LEHRBUCH

Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Anwendungen entwickeln mit Standard-Technologien

5. Auflage

EXTRAS ONLINE

Springer Vieweg

Masterkurs Client/Server-Programmierung mit Java

Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Anwendungen entwickeln mit Standard-Technologien

5. Auflage

Springer Vieweg

Dietmar Abts
FB Wirtschaftswissenschaften
Hochschule Niederrhein
Mönchengladbach, Deutschland

Ergänzendes Material zu diesem Buch finden Sie auf http://extras.springer.com.

ISBN 978-3-658-25924-2 ISBN 978-3-658-25925-9 (eBook)
https://doi.org/10.1007/978-3-658-25925-9

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg
1. Auflage 2003 erschien unter dem Titel „Aufbaukurs JAVA“
2. Auflage 2007 erschien unter dem Titel „Masterkurs JAVA“
© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2003, 2007, 2010, 2015, 2019
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.
Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.
Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Vorwort zur 5. Auflage

»Wir behalten von unsern Studien am Ende doch nur das,
was wir praktisch anwenden.«

J. W. von Goethe, zu Johann Peter Eckermann, 24. Februar 1824

Das vorliegende Buch bietet eine Einführung in Standard-Technologien zur Entwicklung von modernen Client/Server-Anwendungen auf der Basis von Java SE mit zahlreichen Programmbeispielen (56 Java-Projekte) und 45 Übungsaufgaben. Alle Projekte, die Lösungen zu den Aufgaben und die benötigten Bibliotheken sind als Download verfügbar (siehe Kapitel 1.2.3).

Gegenüber der vorhergehenden Auflage wurden zahlreiche Verbesserungen und Aktualisierungen vorgenommen sowie neue Aspekte aufgenommen. Insbesondere wurde das Kapitel zu den REST-basierten Web Services stark erweitert. In Verbindung mit HTML5, CSS3 und Javascript sind insbesondere bei Single-Page-Webanwendungen und Progressive Web Apps REST-fähige Backends und die asynchrone Lieferung von Daten über das REST-API von großer Bedeutung.

Die mit Java 9 als *deprecated* markierten Module sind in Java 11 nicht mehr vorhanden. Hierbei handelt es sich um Module und Frameworks, die insbesondere bei Java EE bzw. Jakarta EE eine Rolle spielen und auch in den Projekten des vorliegenden Buches gebraucht werden. Diese sind jedoch als separate Bibliotheken außerhalb des JDK verfügbar und können in die Projekte eingebunden werden.

Die beigefügten Skripte zur Ausführung der Programme und die Beschreibungen hierzu im Buch berücksichtigen diese Tatsache. Alle Programmbeispiele wurden mit Java 11 und den zum Zeitpunkt der Drucklegung aktuellen Versionen der Bibliotheken ausführlich getestet.

Die verschiedenen Themen des Buches können mit guten Java-Grundkenntnissen erarbeitet werden. Geeignete Zielgruppen sind Studierende der Informatik und Wirtschaftsinformatik, IT-Beschäftigte in Ausbildung und Beruf sowie Java-Programmierer in der Praxis.

Danken möchte ich meinen Leserinnen und Lesern für die konstruktive Kritik und Frau Sybille Thelen, Lektorin des Verlags, für die stets gute Beratung und Zusammenarbeit.

Nun wünsche ich Ihnen viel Spaß beim Lesen und praktischen Erarbeiten der Themen.

Ratingen, Februar 2019

Dietmar Abts
abts@hs-niederrhein.de
www.dietmar-abts.de

Inhaltsverzeichnis

1 Einleitung.....1
1.1 Zielsetzung und Aufbau des Buches.....1
1.2 Tools zur Entwicklung.....3
1.2.1 Java.....3
1.2.2 Entwicklungsumgebung.....3
1.2.3 Materialien zum Buch.....4
1.3 Grundlagen Verteilter Systeme.....4
1.3.1 Definition Verteiltes System.....5
1.3.2 Vor- und Nachteile Verteilter Systeme.....6
1.3.3 Das Client/Server-Modell.....7
1.3.4 Systemarchitekturen.....9
1.3.5 Ziele Verteilter Systeme.....12
1.4 Grundlagen zu TCP/IP.....14
1.4.1 Das Schichtenmodell.....14
1.4.2 IP-Adressen, DNS und Ports.....16
1.5 Java-Klassen für IP-Adressen und Sockets.....19
1.6 Aufgaben.....22

2 Datenaustausch mit JSON.....23
2.1 Java API for JSON Binding.....24
2.2 Serialisierung eines Objektgraphs.....30
2.3 Aufgaben.....35

3 Kommunikation mit UDP.....37
3.1 Das Protokoll UDP.....37
3.2 DatagramSocket und DatagramPacket.....38
3.3 Nachrichten senden und empfangen.....41
3.4 Anfrage und Antwort.....44
3.5 Punkt-zu-Mehrpunkt-Verbindungen.....48
3.6 Aufgaben.....51

4 Netzwerkprogrammierung mit TCP-Sockets.....53
4.1 Das Protokoll TCP.....53
4.2 TCP-Sockets.....54
4.3 Iterative und parallele Server.....58
4.3.1 Iterativer Server.....58

4.3.2 Paralleler Server....62
4.3.3 Thread-Pooling....64
4.4 Ein Framework für TCP-Server....67
4.5 Beenden eines Datenstroms....71
4.6 Ein Chat-Programm....73
4.7 Asynchrone Kommunikation....81
4.8 Aufgaben....87

5 Nachrichtentransfer mit JMS....89
5.1 Java Message Service....90
5.2 Das Point-to-Point-Modell....93
5.2.1 Installation und Konfiguration von ActiveMQ....93
5.2.2 Wichtige Typen und Methoden des JMS-API....95
5.2.3 Das Pull- und Push-Prinzip....97
5.2.4 Der QueueBrowser....102
5.3 Das Request/Reply-Modell....105
5.4 Das Publish/Subscribe-Modell....108
5.5 Dauerhafte Subscriber....111
5.6 Nachrichten filtern....114
5.7 Transaktionen....117
5.8 Aufgaben....121

6 Remote Method Invocation....123
6.1 Vom lokalen zum entfernten Methodenaufruf....123
6.2 Die Architektur einer RMI-Anwendung....126
6.2.1 Protokoll und Komponenten....126
6.2.2 Implementierungsvarianten....133
6.2.3 Dienstauskunft....134
6.2.4 Aufrufparameter und Rückgabewerte....135
6.3 Übertragung von Remote Objects....136
6.4 Mobile Agenten....140
6.5 Der Einsatz von Callback-Methoden....143
6.6 Aufgaben....149

7 Kommunikation über HTTP....153
7.1 Die Protokolle HTTP/1.0 und HTTP/1.1....153
7.2 Ein einfacher Webserver....162
7.3 Ein Monitor für Anfragen und Antworten....170

7.4 Ein einfacher Webserver mit CRUD-Operationen....174
7.5 HTTP/2 und SSL/TLS im Beispiel....182
7.6 Aufgaben....188

8 Bidirektionale Verbindung mit WebSocket....191
8.1 Das WebSocket-Protokoll....191
8.2 Eine einfache WebSocket-Anwendung....193
8.3 Server-Push....199
8.4 Eine Chat-Anwendung....203
8.5 Aufgaben....206

9 SOAP-basierte Web Services mit JAX-WS....209
9.1 Grundlagen....209
9.2 Ein erstes Beispiel....215
9.3 Ein Web Service mit CRUD-Operationen....220
9.4 Web Services mit Apache Tomcat veröffentlichen....225
9.5 Oneway-Operationen....226
9.6 Asynchrone Aufrufe....228
9.7 Transport von Binärdaten....232
9.8 Filterketten....239
9.9 Contract First....251
9.10 Aufgaben....256

10 REST-basierte Web Services mit JAX-RS....259
10.1 Die Grundprinzipien von REST....259
10.2 Ein erstes Beispiel....262
10.2.1 Die erforderlichen Bibliotheken....262
10.2.2 Der Web Service....263
10.2.3 Test mit cURL....271
10.2.4 Die Beschreibungssprache WADL....273
10.2.5 Ein Java-Client....275
10.2.6 Das Client-API von JAX-RS....279
10.3 CRUD-Operationen mit Datenbank....282
10.4 REST-Services mit Apache Tomcat veröffentlichen....294
10.5 Upload und Download von Dateien....295
10.6 Zuverlässigkeit....300
10.6.1 PUT statt POST....300
10.6.2 Kombination von POST und PUT....304

10.7 Parallelzugriffe....307
10.8 Caching....313
10.8.1 Expirationsmodell....315
10.8.2 Validierungsmodell....317
10.9 Asynchrone Verarbeitung....321
10.9.1 Polling....321
10.9.2 Callback....327
10.9.3 Client/Server-API für asynchrone Verarbeitung....331
10.10 Server-Sent Events (SSE)....337
10.11 Filterketten....345
10.12 Rollenbasierter Zugriffsschutz und Verschlüsselung....354
10.13 Aufgaben....363

Anhang
A Tools und Bibliotheken....371
B Globale Bibliotheken in IntelliJ IDEA....375
C IntelliJ-IDEA-Projekte in Eclipse-Projekte konvertieren....377

Literaturhinweise....379
Sachwortverzeichnis....381

1 Einleitung

In diesem Kapitel erfahren Sie zunächst, was Sie von diesem Buch erwarten können, welche Themen behandelt werden und wie es aufgebaut ist. Es folgen Hinweise zu vorausgesetzten Kenntnissen, zur verwendeten Entwicklungsumgebung und zu den zur Verfügung gestellten Quellcodes und Tools.

Des Weiteren erfahren Sie, was man unter einem v*erteilten System* versteht, und lernen Grundlagen zu TCP/IP, zu IP-Adressen und Sockets kennen.

1.1 Zielsetzung und Aufbau des Buches

Das vorliegende Buch beschäftigt sich mit der Implementierung von *Client/Server-Anwendungen* auf Basis der Internet-Protokolle und der darauf aufsetzenden Standard-Technologien. Die dazu gehörenden Spezifikationen sind zum großen Teil unabhängig von der verwendeten Programmiersprache (wie z. B. bei HTTP, WebSocket, SOAP- und REST-basierten Web Services). Wir nutzen für die Implementierung der Beispiele durchweg Java und bekannte Referenzimplementierungen der oben angesprochenen Spezifikationen.

Angesichts des Umfangs der Themen musste eine Auswahl getroffen werden, die beispielsweise in einem vier Semesterwochenstunden umfassenden Kurs behandelt werden kann.

Zielsetzung

Ziel des Buches ist es, über die Themenvielfalt angemessen zu informieren und in die Einzelthemen systematisch mit der für die Anwendungsentwicklung nötigen Tiefe einzuführen. Besonderer Wert wurde dabei auf praxisnahe Programmbeispiele und Übungsaufgaben gelegt.

Die Technologien *JMS* (für die asynchrone Nachrichtenverarbeitung), *WebSocket* (verwendet, um aus serverseitig laufenden Java-Programmen Nachrichten an Clients zu senden), *JAX-WS* (hilft bei der Erstellung von Web Services und zugehörigen Clients, die über SOAP kommunizieren) und *JAX-RS* (verwendet den Architekturstil REST im Rahmen von Web Services) sind Teil der Enterprise-Edition *Java EE*. Sie werden hier *stand-alone* verwendet.

Aufbau des Buches

Dieses Buch ist in zehn Kapitel gegliedert, die jeweils einen Schwerpunkt behandeln. Obwohl die Kapitel weitestgehend in sich geschlossen sind und unabhängig voneinander bearbeitet werden können, wird empfohlen, diese in der hier vorgegebenen Reihenfolge durchzuarbeiten, da in manchen Abschnitten auf frühere Kapitel verwiesen wird. Die vorgestellten Themen werden anhand vollständiger, lauffähiger Programmbeispiele verdeutlicht. Übungsaufgaben am Ende eines jeden Kapitels regen zur selbständigen Beschäftigung mit dem dargebotenen Stoff an.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_1

Wir haben die verschiedenen Themen anhand der verwendeten Kommunikationsverfahren klassifiziert und in drei Kategorien eingeteilt:

- Nachrichten-basierte Kommunikation
- Objekt-basierte Kommunikation
- Web-basierte Kommunikation

Zu den *Nachrichten-basierten Verfahren* gehören die synchrone Nachrichtenübertragung über UDP- und TCP-Sockets sowie Nachrichtensysteme zur Realisierung eines asynchronen Nachrichtenaustausches mit Zwischenspeicherung (Message Oriented Middleware).

Zum *Objekt-basierten Ansatz* gehört RMI (Remote Method Invocation). RMI ermöglicht die Kommunikation zwischen Java-Objekten in verschiedenen JVM-Instanzen (JVM = Java Virtual Machine).

In die Kategorie der *Web-basierten Verfahren* haben wir die Verfahren aufgenommen, die auf der Basis des Transportprotokolls HTTP und damit verwandter Techniken arbeiten. Diese sind alle unabhängig von der verwendeten Plattform und Programmiersprache.

Einordnung der Themen

Nachrichten-basierte Kommunikation
- Kommunikation mit UDP
- Netzwerkprogrammierung mit TCP-Sockets
- Nachrichtentransfer mit JMS

Objekt-basierte Kommunikation
- Remote Method Invocation

Web-basierte Kommunikation
- Kommunikation über HTTP
- Bidirektionale Verbindung mit WebSocket
- SOAP-basierte Web Services mit JAX-WS
- REST-basierte Web Services mit JAX-RS

Vorausgesetzte Kenntnisse

Von den Leserinnen und Lesern werden erwartet:

- Grundlegende Java-Kenntnisse, insbesondere: Konzepte der objektorientierten Programmierung, Dateiverarbeitung und Thread-Programmierung,
- das Wissen um Internet und WWW sowie Erfahrung im Umgang mit einem Webbrowser,
- Grundlagenkenntnisse in HTML, CSS und JavaScript.

Buchtipps hierzu finden Sie am Ende des Buches.

1.2 Tools zur Entwicklung

1.2.1 Java

Ab *Java 11* sind einige Module und Frameworks nicht mehr im JDK (Java Development Kit) enthalten, so sind z. B. entfallen:

- Oberfächen-Framework JavaFX,
- Java API for XML Web Services (JAX-WS),
- Java Architecture for XML Binding (JAXB),
- JavaBeans Activation Framework (JAF)

sowie alle weiteren Tools, die zu den entfernten Teilen gehören.

Die entfernten Module und Frameworks können aber als separate Bibliotheken aus dem Internet bezogen werden und in Projekte importiert werden.

Die meisten dieser Bibliotheken sind Bestandteil der Materialien zu diesem Buch. Im Anhang zu diesem Buch ist aufgeführt, welche Bibliotheken zur Compilierung bzw. Programmausführung benötigt werden und wie diese in *IntelliJ IDEA* als *Global Libraries* eingerichtet werden können.

Mit *Java 9* wurde die Sprache um Module als neues Feature erweitert. Viele hier benötigten Bibliotheken sind (noch) nicht modularisiert. Die beigefügten Skripte zur Ausführung der Programme berücksichtigen diese Tatsache. Die Beispielprojekte dieses Buches sind nicht im Sinne dieses Features modularisiert.

1.2.2 Entwicklungsumgebung

Als Entwicklungsumgebung wird *IntelliJ IDEA* eingesetzt. Die Beispielprojekte sind also auf diese Entwicklungsumgebung ausgerichtet. Häufig werden innerhalb eines Projekts Unterprojekte (so genannte *Module*, nicht zu verwechseln mit dem Java-Begriff *Modul*) verwendet, um klar zwischen Client und Server unterscheiden und doch beides in einem einzigen Projekt zusammenfassen zu können.

Mit geringen Anpassungen kann auch eine andere Java-Entwicklungsumgebung (z. B. *Eclipse*) genutzt werden. Der Anhang enthält eine Kurzanleitung zur Konvertierung in Eclipse-Projekte.

Alle Beispielprogramme und Lösungen zu den Übungsaufgaben wurden mit Hilfe von Java 11 unter Windows 10 entwickelt. Selbstverständlich sind die hier vorgestellten Programme ohne Änderung auch auf UNIX/Linux-Systemen lauffähig.

Einzelheiten zu den verwendeten Tools und Bibliotheken können den entsprechenden Kapiteln entnommen werden.

Hinweis zur Eingabe von Kommandos im Terminal unter Windows

Mitunter besteht ein Kommando zur Programmausführung aus einer sehr langen Zeile. Nach Belieben kann es auch auf mehrere Zeilen verteilt werden. Dazu kann

das Zeichen ^ als Fortsetzungszeichen genutzt werden. Nach Drücken der entsprechenden Taste muss ein Leerzeichen eingegeben werden.

Beispiel:

```
java --add-opens java.base/java.lang=ALL-UNNAMED ^
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;^
../../libs/activation/*;../../libs/jaxb/* Server
```

1.2.3 Materialien zum Buch

Auf der Webseite zu diesem Buch

`https://www.springer.com`

oder

`https://www.dietmar-abts.de`

stehen alle Projekte, Lösungen zu den Übungsaufgaben, benötigte Bibliotheken und Sonstiges zum Download bereit.

1.3 Grundlagen Verteilter Systeme

Jede (klassische) Anwendung kann grob in drei Schichten eingeteilt werden:

Präsentation
: Diese Schicht (auch Benutzungsschnittstelle genannt) hat die Aufgabe, den Dialog mit dem Benutzer zu steuern sowie Daten und Befehle an das System zu übermitteln.

Verarbeitung
: Diese Schicht enthält die Verarbeitungslogik. Hier wird der eigentliche Nutzen der Anwendung erzielt.

Datenhaltung
: Diese Schicht hat die Aufgabe, die Daten abzuspeichern und bei Bedarf wieder zur Verfügung zu stellen.

Abbildung 1-1: Beispiel Webanwendung

Abbildung 1-1 zeigt eine Webanwendung, die dem Benutzer das Produktangebot eines Unternehmens präsentiert.

Die drei Schichten *Präsentation*, *Verarbeitung* und *Datenhaltung* sind jeweils auf eigenen Rechnern implementiert.

Der *Webbrowser* ist die Benutzungsoberfläche, die die abgefragten Informationen grafisch präsentiert.

Der *Webserver* erzeugt Datenbankabfragen, bereitet die Abfrageergebnisse als Webseite auf und übermittelt diese an den Client.

Der *Datenbankserver* verwaltet den Produktkatalog und führt die Datenbankabfragen aus.

Webbrowser, Webserver und Datenbankserver sind autonome Teilsysteme, die koordiniert miteinander kooperieren.

1.3.1 Definition Verteiltes System

Ein V*erteiltes System* ist ein System, bei dem einzelne Komponenten auf verschiedene, räumlich getrennte Knoten eines Netzwerks verteilt sind, die miteinander kooperieren, um eine gemeinsame Aufgabe zu erfüllen.

Da es im Allgemeinen mehrere Möglichkeiten für die Verteilung der Komponenten im Netz gibt, besteht das *Konfigurationsproblem*, die *logische* Struktur der Anwendung auf die *physische* Struktur (Rechner im Netz) so abzubilden, dass bei vorgegebenen Randbedingungen ein maximaler Nutzen erzielt wird (siehe Abbildung 1-2).

Ob eine Verteilung überhaupt möglich ist, entscheidet sich bereits beim Entwurf der Anwendungssoftware. Der Entwurf muss unabhängig von einer späteren physischen Konfiguration sein. Dem Benutzer muss die konkrete physische Aufteilung verborgen bleiben.

Abbildung 1-2: Abbildung der logischen auf die physische Struktur

1.3.2 Vor- und Nachteile Verteilter Systeme

Die verteilte Verarbeitung hat folgende Vor- und Nachteile:

Vorteile

- *Abbild der Realität*
  Leistungen werden dort erbracht, wo sie benötigt werden. Daten werden dort erfasst, wo sie im Geschäftsprozess entstehen.
- *Wirtschaftlichkeit*
  Teure Ressourcen (Geräte, Softwarekomponenten) stehen mehreren Rechnern zur Verfügung und können gemeinsam genutzt werden. Automatisierte Aufgaben werden möglichst dort ausgeführt, wo sie am wirtschaftlichsten sind.
- *Lastverteilung*
  Da mehrere Rechner mit jeweils eigenem Betriebssystem arbeiten, kann durch teilweise parallele Verarbeitung die Gesamtbearbeitungszeit verkürzt werden.
- *Skalierbarkeit*
  Einzelne Komponenten können leichter an einen steigenden Bedarf angepasst werden.
- *Fehlertoleranz*
  Beim Ausfall eines Rechners bleiben die anderen betriebsbereit. Die Aufgaben des ausgefallenen Rechners können oft von einem anderen Rechner übernommen werden.

Nachteile

- *Komplexität durch Verteilung und Heterogenität*
  Die Verteilung der Komponenten großer Systeme und die Heterogenität bei Betriebssystemen, Programmiersprachen, Datenformaten und Kommunikationsprotokollen stellen hohe Anforderungen an die Entwickler, wenn das Gesamtsystem überschaubar bleiben soll.
- *Komplexe Netzinfrastrukturen*
  Netzarchitektur und Netzmanagement sind das Rückgrat der Systeme eines Unternehmens. Die Integration heterogener Systeme mit Komponenten unterschiedlicher Hersteller- und Standardarchitekturen stellt hohe Anforderungen an die Administration.
- *Sicherheitsrisiken*
  Verteilte Systeme bieten Möglichkeiten für unberechtigte Zugriffe. Im Netz übertragene Nachrichten können abgehört oder sogar verändert werden. Der Einsatz von Verschlüsselungsverfahren und Firewall-Systemen sind wirksame Schutzmaßnahmen.

Aufgabe von so genannten Verteilungsplattformen ist es, die Komplexität verteilter Systeme beherrschbar zu machen.

1.3.3 Das Client/Server-Modell

Das *Client/Server-Modell* ist das am weitesten verbreitete Modell für die verteilte Verarbeitung.

Client/Server

Client/Server bezeichnet die Beziehung, in der zwei Programme zueinander stehen. Der *Client* stellt eine Anfrage an den Server, eine gegebene Aufgabe zu erledigen. Der *Server* erledigt die Aufgabe und liefert das Ergebnis beim Client ab. Ein Kommunikationsdienst verbindet Client und Server miteinander.

Ein *Client/Server-System* besteht also aus

- einem oder mehreren Clients (Auftraggebern),
- einem Server (Auftragnehmer) und
- einem Kommunikationsdienst (Netzwerk, Vermittler).

Software-Sicht

Client und Server sind Programme, die auch auf demselben Rechner laufen können. Sie müssen also nicht notwendig über ein Netz verbunden sein. Das macht deutlich, dass es sich hierbei um eine Software- und keine Hardwarearchitektur handelt. In der Praxis werden natürlich Client und Server auf unterschiedlichen Rechnern verteilt, um die bekannten Vorteile eines verteilten Systems zu erzielen.

Abbildung 1-3: Client/Server-Modell

Hardware-Sicht

Gelegentlich werden auch die *Trägersysteme* (Rechner), auf denen die Programme laufen, als Client bzw. Server bezeichnet. Wir nutzen die Begriffe Client bzw. Server als Homonyme. Aus dem Zusammenhang wird dann klar, welche Bedeutung gemeint ist.

Ein Server kann die Erledigung einer Aufgabe weiter delegieren und dazu auf andere Server zugreifen. Er spielt dann selbst die Rolle eines Clients. Der Webserver in Abbildung 1-1 hat diese Doppelfunktion.

Im Folgenden sind charakteristische Merkmale von Client und Server zusammengefasst:

Client-Merkmale

- Ein Programm wird vorübergehend zum Client, wenn es einen Dienst von einem anderen Programm (Server) anfordert. Darüber hinaus kann das Programm andere Aufgaben lokal ausführen.
- Der Client leitet den Kontakt mit einem Server *aktiv* ein.
- Der Client kann im Laufe einer Sitzung auf mehrere Server zugreifen, kontaktiert aber aktiv nur jeweils einen Server.

Server-Merkmale

- Der Server ist ein Programm, das einen bestimmten Dienst bereitstellt. Er kann in der Regel gleichzeitig mehrere Clients bedienen.
- Häufig werden Server automatisch beim Hochfahren des Rechners gestartet und beim Herunterfahren beendet.
- Der Server wartet *passiv* auf die Verbindungsaufnahme durch einen Client.
- Da Server eigenständige Programme sind, können mehrere Server unabhängig voneinander auf einem einzigen Rechner als Trägersystem laufen.
- *Parallelbetrieb* ist ein grundlegendes Merkmal von Servern. Mehrere Clients können einen bestimmten Dienst in Anspruch nehmen, ohne warten zu müssen, bis der Server mit der Erledigung der laufenden Anfrage fertig ist. In den späteren Java-Programmbeispielen werden die Client-Anfragen jeweils in einem neuen Thread bedient.

Abbildung 1-4: Interaktionsarten

synchron/asynchron

Die Interaktion zwischen Client und Server kann synchron oder asynchron erfolgen (siehe Abbildung 1-4).

Bei der *synchronen* Kommunikation wartet der Client nach Absenden der Anfrage an den Server so lange, bis er eine Antwort erhält, und kann dann erst andere Aktivitäten ausführen.

Im *asynchronen* Fall sendet der Client die Anfrage an den Server und arbeitet sofort weiter. Beim Eintreffen der Antwort werden dann bestimmte Ereignisbehandlungsroutinen beim Client aufgerufen. Die durch den Server initiierten Rückrufe (*Callbacks*) erfordern allerdings zusätzliche Kontrolle beim Client, wenn ungewünschte Unterbrechungen der Arbeit vermieden werden sollen.

1.3.4 Systemarchitekturen

Bei *einstufigen Architekturen* wird die gesamte Anwendung (Präsentation, Verarbeitung und Datenhaltung) auf *einem* Rechner implementiert. Die Benutzerinteraktion erfolgt über Terminals, die direkt oder über Kommunikationsserver an den Rechner angeschlossen sind.

Multi-Tier-Architekturen

Verteilte Anwendungen haben eine *mehrstufige Architektur* (*Multi-Tier-Architektur*). Beispielsweise werden Applikations- und Datenbankserver verschiedenen Ebenen zugeordnet.

Bei der *zweistufigen Architektur* ist die Gesamtanwendung zwei Ebenen zugeordnet: Client und Server.

Abbildung 1-5: Eine zweistufige Architektur

Für die Arbeitsteilung zwischen Client und Server existieren verschiedene Alternativen, je nachdem, wo die Schichten *Präsentation*, *Verarbeitung* und *Datenhaltung* angesiedelt sind.

Abbildung 1-6 zeigt fünf Alternativen der Aufgabenverteilung bei der zweistufigen Architektur eines Client/Server-Systems. Von Alternative 1 bis 5 wird immer mehr Funktionalität auf den Client verlagert.

Thin und Rich Clients

Alternative 1 und 2 stellen die extreme Form eines *Thin Clients* dar. Der Client übernimmt nur die Aufgaben einer Benutzungsschnittstelle. Im Gegensatz dazu steht der *Rich Client*, der neben der Präsentation auch (teilweise) die Anwendungslogik und ggf. die lokale Datenhaltung übernimmt.

Einige Merkmale von Thin und Rich Clients:

- Ein Thin Client belastet den Server stärker als ein Rich Client.
- Die Netzbelastung ist bei Thin Clients stärker als bei Rich Clients.
- Rich Clients können meist auch offline arbeiten. Unter Umständen müssen dann aber Daten repliziert und synchronisiert werden.
- Die Versorgung mit neuen Software-Releases ist bei der Verwendung von Thin Clients einfacher, da hier im Allgemeinen nur zentrale Komponenten auf dem Server auf den neuesten Stand gebracht werden müssen.

Abbildung 1-6: Alternativen der Aufgabenverteilung

Moderne Anwendungen werden mehr und mehr in den Webbrowser verlagert. Dies bedeutet keine lokalen Installationen mehr und Zugriff von überall. Frameworks machen den Webbrowser zum Rich Client. Daten und Geschäftslogik liegen auf dem Server. Hier liegt also Alternative 3 vor.

Bei der *dreistufigen Architektur* wird die Gesamtanwendung auf drei Ebenen aufgeteilt: z. B. Client, Applikationsserver und Datenbankserver. Dieses Modell ist auch Basis vieler ERP-Systeme (Enterprise Resource Planning).

Abbildung 1-7: Eine dreistufige Architektur

Insbesondere bei Internet-basierten Anwendungen findet man eine *vierstufige Architektur*, bei der ein Webserver dem Applikationsserver vorgeschaltet ist.

Abbildung 1-8: Eine vierstufige Architektur

Abbildung 1-8 zeigt eine klassische webbasierte Anwendung. Dabei wird die Präsentation in Form von HTML-Seiten auf dem Webserver erzeugt, die dann per HTTP an den Client ausgeliefert werden.

Dank leistungsfähiger Technologien wie HTML5, CSS3 und JavaScript-Frameworks setzen sich immer mehr so genannte *Single-Page-Webanwendungen* durch.

Eine solche Anwendung besteht in der Regel aus einer einzigen HTML-Seite, deren Dateninhalte asynchron nachgeladen werden.

Die HTML-Elemente werden dynamisch mit JavaScript im Browser verändert. Die Validierung von Formulareingaben und die Navigationssteuerung werden ebenfalls mit JavaScript im Browser ausgeführt. Es ist kein klassischer Seitenwechsel mehr nötig. Die Daten werden meist im XML- oder JSON-Format vom Server übertragen. Dazu werden vorzugsweise REST-basierte Web Services eingesetzt.

1.3.5 Ziele Verteilter Systeme

In mehrstufigen Architekturen sind die Komponenten eines Anwendungssystems auf potentiell heterogene Trägersysteme in einem Netzwerk verteilt. Um die Komplexität des Gesamtsystems bei der Bedienung oder Entwicklung der Anwendung vor dem Benutzer bzw. Entwickler verborgen zu halten, bedarf es einer geeigneten Software-Infrastruktur, die die Interaktion zwischen den Komponenten in besonderer Weise unterstützt.

Middleware

Abbildung 1-9: Middleware

Middleware ist Software, die den Austausch von Informationen zwischen den verschiedenen Komponenten eines verteilten, heterogenen Systems unterstützt. Die

Anwendungen werden dabei von den komplexen Details der internen Vorgänge abgeschirmt.

Verteilungstransparenz

In verteilten Systemen sollen interne Abläufe und Implementierungsdetails für den Benutzer oder Anwendungsentwickler nicht sichtbar sein. *Verteilungstransparenz* verbirgt die Komplexität verteilter Systeme.

Es existieren mehrere untergeordnete Transparenzbegriffe, die jeweils einen bestimmten Aspekt bezeichnen. Beispiele hierfür sind:

- *Ortstransparenz*
  Der Ort einer Ressource ist dem Benutzer nicht bekannt. Er identifiziert sie über einen Namen, der keine Information über ihren Aufenthaltsort enthält.
- *Zugriffstransparenz*
  Die Form des Zugriffs auf eine Ressource ist einheitlich und unabhängig davon, ob die Ressource lokal oder auf einem entfernten Rechner zur Verfügung steht. Unterschiede verschiedener Betriebssysteme und Dateisysteme werden verborgen.
- *Nebenläufigkeitstransparenz*
  Der gleichzeitige Zugriff mehrerer Benutzer auf dieselbe Ressource (z. B. Datenbanktabelle) erfolgt ohne gegenseitige Beeinflussung und lässt keine falschen Ergebnisse entstehen.
- *Replikationstransparenz*
  Sind aus Verfügbarkeits- oder Performance-Gründen mehrere Kopien einer Ressource (z. B. eines Datenbestandes) vorhanden, so merkt der Benutzer nicht, ob er auf das Original oder eine Kopie zugreift. Das System sorgt dafür, dass alle Kopien bei Änderungen letztendlich konsistent bleiben.
- *Migrationstransparenz*
  Ressourcen können von einem Rechner auf einen anderen verlagert werden, ohne dass der Benutzer dies bemerkt.
- *Fehlertransparenz*
  Der Benutzer soll nicht mit allen auftretenden Fehlern im System konfrontiert werden. Das Auftreten von Fehlern und die Fehlerbehebung sollen vor dem Benutzer weitestgehend verborgen sein.

Middleware-Produkte stellen dem Entwickler die für die Anwendung benötigten Funktionen in Form eines *API* (Application Programming Interface) zur Verfügung.

Datenbank-Middleware ermöglicht den Zugriff auf unterschiedliche Datenbanksysteme, ohne das Anwendungsprogramm beim Wechsel des Datenbanksystems ändern zu müssen. Das *JDBC-API* erlaubt es einem Java-Programm, SQL-Anweisungen an beliebige relationale Datenbanken zu schicken.

Andere Middleware-Produkte verwenden eine synchrone Verbindung zwischen Client und Server, um Prozeduren nach dem *RPC-Modell* aufzurufen.

Der *Remote Procedure Call* (*RPC*) versteckt den Aufruf einer auf einem anderen Rechner im Netz implementierten Prozedur hinter einem lokalen Prozeduraufruf und bietet damit ein sehr vertrautes Programmiermodell an. Das Programm, das die Prozedur aufruft, agiert als Client, das Programm, das die aufgerufene Prozedur ausführt, als Server.

Remote Method Invocation (*RMI*) ist die objektorientierte Umsetzung des RPC-Modells für Java-Programme (siehe Kapitel 6).

Message Oriented Middleware (*MOM*) unterstützt den zeitversetzten (asynchronen) Austausch von Nachrichten mit Hilfe eines Vermittlers (Message Broker), der eine Warteschlange verwaltet. Die Nachricht des Senders wird vom Vermittler in die Warteschlange des Empfängers gelegt. Der Empfänger kann diese Nachricht zu einem späteren Zeitpunkt aus der Warteschlange holen. Sender und Empfänger agieren unabhängig voneinander und sind also über den Vermittler nur lose gekoppelt (siehe Kapitel 5).

1.4 Grundlagen zu TCP/IP

Das *Internet* ist ein weltumspannendes Rechnernetz, das aus einer Vielzahl von internationalen und nationalen öffentlichen Netzen sowie privaten lokalen Netzen besteht. Spezielle Kopplungselemente, so genannte Router, verbinden diese Teilnetze miteinander und ermöglichen so den Datenaustausch zwischen Rechnern, die sich in unterschiedlichen Teilnetzen befinden. Das Internet ist ein offenes Netz, zu dem Unternehmen, nicht-kommerzielle Organisationen sowie Privatpersonen gleichermaßen Zugang haben.

Kommunikationsprotokolle beinhalten Sprach- und Handlungsregeln für den Datenaustausch. Sie sind das Verständigungsmittel in einem Rechnernetz.

TCP/IP

TCP/IP (*Transmission Control Protocol/Internet Protocol*) bezeichnet eine Familie von einzelnen, aufeinander abgestimmten Protokollen für die Kommunikation im Internet. TCP und IP sind die beiden wichtigsten Protokolle dieser Familie.

Im Gegensatz zum offenen Internet ist ein *Intranet* ein privates, innerbetriebliches Rechnernetz auf Basis der Internet-Protokolle ("privates Internet").

1.4.1 Das Schichtenmodell

Zur Beschreibung der Kommunikation werden allgemein Schichtenmodelle eingesetzt. Im Internet werden vier aufeinander aufbauende Schichten unterschieden. Jede Schicht hat ihre eigene Funktionalität, die sie der jeweils höheren Schicht zur Verfügung stellt.

Abbildung 1-10: TCP/IP-Schichtenmodell

1. Die *Verbindungsschicht* umfasst die Netzwerkhardware und Gerätetreiber. Sie ist die Basis für eine zuverlässige physische Verbindung zwischen zwei benachbarten Systemen.

2. Die *Vermittlungsschicht* enthält das Protokoll IP (*Internet Protocol*), das die Internet-Adressen (IP-Adressen) definiert, die als Basis für die Wahl der Route im Internet dienen. Einzelne Datenpakete werden vom Sender zum Empfänger über verschiedene Teilnetze des Internets geleitet. Das hier angesiedelte Protokoll *ARP* (*Address Resolution Protocol*) übersetzt logische IP-Adressen in physische Adressen (Hardwareadressen) eines Geräts im lokalen Netz.

3. Die *Transportschicht* enthält die beiden Protokolle *TCP* (*Transmission Control Protocol*) und *UDP* (*User Datagram Protocol*).

   TCP stellt der Anwendung ein verlässliches, verbindungsorientiertes Protokoll zur Verfügung. Vor der eigentlichen Datenübertragung in Form von Datenpaketen wird eine virtuelle Ende-zu-Ende-Verbindung zwischen Sender und Empfänger aufgebaut.

   UDP ist ein so genanntes verbindungsloses Protokoll. Datenpakete werden ins Internet geschickt, ohne dass vorher eine Verbindung mit dem Empfänger hergestellt wurde.

   UDP und TCP werden in den Kapiteln 3 und 4 behandelt.

4. Beispiele für die Protokolle der *Anwendungsschicht* sind: *SMTP* (*Simple Mail Transfer Protocol*), *Telnet*, *FTP* (*File Transfer Protocol*) und *HTTP* (*Hypertext Transfer Protocol*), das die Basis für das *World Wide Web* (*WWW*) ist.

Abbildung 1-11 zeigt den Ablauf einer Kommunikation zwischen Client und Server, die ihre Daten direkt über TCP austauschen.

Abbildung 1-11: Datenübertragung mit TCP

Der Client übergibt seine Daten an die Transportschicht. TCP sorgt für die Aufteilung der Daten in Pakete, versieht jeweils die Nutzdaten eines Pakets mit einem Datenkopf (Header) und gibt die Pakete weiter an die Vermittlungsschicht. Die Vermittlungsschicht fügt ihrerseits einen IP-Header hinzu. Schließlich packt die Netzwerkkarte einen Rahmen um die Daten. Beim Empfänger werden in umgekehrter Reihenfolge die Header Schicht für Schicht wieder entfernt und ausgewertet, bevor die Daten an die übergeordnete Schicht weitergegeben werden.

1.4.2 IP-Adressen, DNS und Ports

IP-Adressen

Eine Aufgabe des Protokolls IP ist die Bereitstellung eines Adressierungsschemas, das von den physischen Adressen der Netzwerkhardware unabhängig ist.

Jeder Rechner im Internet (genauer: der Rechner als Komponente eines Teilnetzes) hat eine eindeutige *IP-Adresse*, die in der Version *IPv4* aus 32 Bit besteht und durch eine Folge von vier durch Punkte getrennte Zahlen zwischen 0 und 255 dargestellt wird.

Eine IP-Adresse besteht aus zwei Teilen: der *Netzadresse*, die das Netz eindeutig identifiziert, und der *Rechneradresse*, die den Rechner in diesem Netz eindeutig identifiziert. Offizielle Netzadressen werden zentral vergeben, Rechneradressen können frei vom Netzwerkadministrator der jeweiligen Institution vergeben werden.

Die Adressen können in fünf verschiedene Klassen eingeteilt werden. Adressen der *Klasse A* nutzen 8 Bit für die Netzadresse (beginnend mit 0), Adressen der *Klasse B* 16 Bit (beginnend mit 10) und Adressen der *Klasse C* 24 Bit (beginnend mit 110). Adressen der *Klasse D* sind für Multicast-Anwendungen (siehe Kapitel 3), Adressen der *Klasse E* sind für zukünftige Zwecke reserviert.

Adressbereiche:

Klasse A:	`0.0.0.0 - 127.255.255.255`
Klasse B:	`128.0.0.0 - 191.255.255.255`
Klasse C:	`192.0.0.0 - 223.255.255.255`
Klasse D:	`224.0.0.0 - 239.255.255.255`
Klasse E:	`240.0.0.0 - 247.255.255.255`

Beispiel:

Die IP-Adresse `194.94.124.236` gehört zur Klasse C und identifiziert einen Rechner in dem durch die Netzadresse `194.94.124.0` identifizierten Netz.

Die starre Klasseneinteilung der IP-Adressen wird durch das heute übliche Verfahren *CIDR* (*Classless Inter-Domain Routing*) aufgehoben. Hierbei erfolgt die Aufteilung in Netz- und Rechneradresse bitweise mit Hilfe von so genannten Subnetzmasken.

Jeder Rechner im Internet kann sich selbst mit der Adresse `127.0.0.1` adressieren (*Loopback-Adresse*).

Private IP-Adressen wie z. B. `10.0.0.0` bis `10.255.255.255` oder `192.168.0.0` bis `192.168.255.255` werden nie im Internet geroutet. Jede Organisation kann sie frei im internen Netz verwenden. Um eine Verbindung mit dem Internet zu ermöglichen, werden private Adressen in öffentliche Adressen übersetzt und umgekehrt.

Die neue Version *IPv6* bietet vor allem einen größeren Adressraum. Es werden 128 Bit zur Darstellung der Adresse verwendet. IPv6-Adressen werden hexadezimal notiert, wobei die Adresse in 8 Blöcken zu jeweils 16 Bit unterteilt wird. Führende Nullen innerhalb eines Blockes dürfen weggelassen werden. Blöcke werden durch einen Doppelpunkt getrennt. Blöcke mit lauter Nullen können durch zwei Doppelpunkte ersetzt werden. Diese Notation ist aber nur an einer Stelle der Adresse erlaubt, damit die Eindeutigkeit nicht verloren geht.

Beispiel:

`2001:0db8:0:0:0:0:1428:57ab` ist gleichbedeutend mit `2001:db8::1428:57ab`

Domain Name System

Zur besseren Handhabbarkeit wird das Nummernsystem durch ein Namensystem, dem *Domain Name System* (*DNS*), überlagert. IP-Adressen werden einem sprechenden Namen, dem Domain-Namen, zugeordnet. Domain-Namen sind hierarchisch aufgebaut.

Beispiel: `www.hs-niederrhein.de`

Hier handelt es sich um den Webserver der Hochschule Niederrhein mit `de` (Deutschland) als *Top Level Domain*.

Der Hostname *localhost* identifiziert den eigenen Rechner und entspricht der IP-Adresse `127.0.0.1`.

DNS-Server sind spezielle Verzeichnisdienste, die die Zuordnung von IP-Adressen zu DNS-Namen und umgekehrt unterstützen. Hierüber kann z. B. die IP-Adresse zu einem Namen erfragt werden.

Zu beachten ist, dass mehrere Rechner mit jeweils eigener IP-Adresse denselben DNS-Namen haben können und zu einer IP-Adresse mehrere DNS-Namen gehören können.

Portnummern

Um einen bestimmten Dienst (Server) im Internet zu identifizieren, reicht die IP-Adresse des Rechners, auf dem der Server läuft, nicht aus, da mehrere Server auf demselben Rechner gleichzeitig laufen können.

Zur Identifizierung eines Servers dient neben der IP-Adresse des Rechners die so genannte *Portnummer*. Portnummern werden auch vom Server benutzt, um den anfragenden Client für die Rückantwort zu adressieren. Portnummern sind ganze Zahlen von `0` bis `65535`. Sie werden von den Protokollen der Transportschicht verwendet. TCP und UDP können Portnummern unabhängig voneinander verwenden, d. h. ein TCP-Server und ein UDP-Server können auf einem Rechner unter derselben Portnummer zur gleichen Zeit laufen.

Die Portnummern von `0` bis `1023` (*System Ports*) sind für so genannte *well-known services* reserviert.

Im Bereich von `1024` bis `49151` sind weitere Portnummern (*User Ports*) registriert, z. B. `1099` für die RMI-Registry und `3306` für den MySQL-Datenbankserver.

Der Bereich `49152 - 65535` steht zur freien Verfügung.

Beispiele für System Ports:

Port	Protokoll	Service	Beschreibung
7	TCP/UDP	Echo	liefert die Eingabe als Antwort zurück
13	TCP/UDP	Daytime	liefert die aktuelle Zeit des Servers
20	TCP	FTP (Data)	überträgt Daten zum Server
21	TCP	FTP (Control)	sendet FTP-Kommandos zum Server
23	TCP	Telnet	Terminalemulation
25	TCP	SMTP	überträgt E-Mails zwischen Rechnern
53	TCP/UDP	DNS	Domain Name System
80	TCP	HTTP	Webserver

1.5 Java-Klassen für IP-Adressen und Sockets

InetAddress

Die Klasse `java.net.InetAddress` repräsentiert eine IP-Adresse. Subklassen hiervon sind `Inet4Address` und `Inet6Address`, die IP-Adressen der Version 4 bzw. 6 repräsentieren.

Die folgenden drei statischen Methoden erzeugen `InetAddress`-Objekte:

`static InetAddress getLocalHost() throws UnknownHostException`
: ermittelt die IP-Adresse des Rechners, auf dem diese Methode aufgerufen wird. `java.net.UnknownHostException` ist Subklasse von `java.io.IOException`. Eine Ausnahme dieses Typs wird ausgelöst, wenn die IP-Adresse eines Rechners nicht ermittelt werden kann.

`static InetAddress getByName(String host) throws UnknownHostException`
: liefert die IP-Adresse zu einem Hostnamen. Enthält das Argument `host` die numerische Adresse anstelle eines Namens, so wird nur die Gültigkeit des Adressformats geprüft.

`static InetAddress[] getAllByName(String host) throws UnknownHostException`
: liefert ein Array von IP-Adressen, die alle zum vorgegebenen Hostnamen gehören.

Weitere `InetAddress`-Methoden:

`String getHostAddress()`
: liefert die IP-Adresse eines `InetAddress`-Objekts als Zeichenkette.

`String getHostName()`
: liefert den Hostnamen zu dieser IP-Adresse. Falls das `InetAddress`-Objekt mittels einer numerischen Adresse erzeugt wurde und der zugehörige Hostname nicht ermittelt werden konnte, wird diese Adresse wieder zurückgegeben.

→ Projekt address

Das Programm `Localhost` zeigt den Hostnamen und die IP-Adresse des eigenen Rechners an.

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Localhost {
   public static void main(String[] args) throws UnknownHostException {
      InetAddress address = InetAddress.getLocalHost();
      System.out.printf("%s/%s%n",
            address.getHostName(),
            address.getHostAddress());
   }
}
```

Das Programm `Lookup` liefert die IP-Adresse zu einem Hostnamen.

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Lookup {
   public static void main(String[] args) throws UnknownHostException {
      InetAddress address = InetAddress.getByName(args[0]);
      System.out.printf("Looking up %s -> %s/%s%n",
            args[0],
            address.getHostName(),
            address.getHostAddress());
   }
}
```

Aufruf:

```
java -cp out/production/address Lookup www.hs-niederrhein.de
```

Ausgabe des Programms:

```
Looking up www.hs-niederrhein.de -> www.hs-niederrhein.de/194.94.120.88
```

Wird beim Aufruf des Programms eine numerische Adresse anstelle eines Namens mitgegeben und konnte der Hostname nicht ermittelt werden, so wird als Hostname die vorgegebene Adresse ausgegeben.

InetSocketAddress

Java-Programme nutzen so genannte *Sockets* als Programmierschnittstelle für die Kommunikation über UDP/IP und TCP/IP. Sockets stellen die Endpunkte einer Kommunikationsbeziehung zwischen Prozessen dar. Sie basieren auf einer IP-Socket-Adresse, bestehend aus einer IP-Adresse und einer Portnummer.

Die Klasse `java.net.InetSocketAddress` repräsentiert eine IP-Socket-Adresse (IP-Adresse und Portnummer). Diese Klasse ist von der abstrakten Klasse `java.net.SocketAddress` abgeleitet.

Abbildung 1-12: Kommunikationsverbindung zwischen zwei Programmen

Abbildung 1-12 visualisiert die Verbindung von zwei Programmen über Sockets, hier z. B. die Verbindung eines Browsers auf Rechner A mit einem Webserver auf Rechner B.

Um ein `InetSocketAddress`-Objekt zu erzeugen, stehen die folgenden Konstruktoren zur Verfügung:

`InetSocketAddress(int port)`
: erzeugt eine Socket-Adresse mit der so genannten *Wildcard-Adresse* `0.0.0.0`. Hiermit ist irgendeine IP-Adresse des Rechners, auf dem der Konstruktor aufgerufen wird, gemeint. Zu beachten ist, dass ein Rechner mehrere Netzwerkadapter mit jeweils eigener IP-Adresse haben kann.

`InetSocketAddress(InetAddress addr, int port)`

`InetSocketAddress(String hostname, int port)`

Einige `InetSocketAddress`-Methoden:

`InetAddress getAddress()`
: liefert die IP-Adresse.

`String getHostName()`
: liefert den Hostnamen.

`int getPort()`
: liefert die Portnummer.

Das Programm `CreateSocketAddress` verwendet die verschiedenen Konstruktoren zur Veranschaulichung.

```
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.UnknownHostException;

public class CreateSocketAddress {
   public static void main(String[] args) {
      InetSocketAddress socketAddress = new InetSocketAddress(80);
      output(socketAddress);

      try {
         InetAddress address = InetAddress.getByName("www.google.de");
         socketAddress = new InetSocketAddress(address, 80);
         output(socketAddress);
      } catch (UnknownHostException e) {
         System.out.println(e);
      }

      socketAddress = new InetSocketAddress("www.google.de", 80);
      output(socketAddress);

      socketAddress = new InetSocketAddress("localhost", 80);
      output(socketAddress);
   }
```

```
   private static void output(InetSocketAddress socketAddress) {
      System.out.println(socketAddress.getAddress());
      System.out.println(socketAddress.getHostName());
      System.out.println(socketAddress.getPort());
      System.out.println();
   }
}
```

Ausgabe:

```
0.0.0.0/0.0.0.0
0.0.0.0
80

www.google.de/108.177.126.94
www.google.de
80

www.google.de/108.177.126.94
www.google.de
80

localhost/127.0.0.1
localhost
80
```

1.6 Aufgaben

1. Entwickeln Sie eine verbesserte Version des Programms `Lookup` mit den folgenden Eigenschaften:

   a) Sind einem Hostnamen mehrere IP-Adressen zugeordnet, so sollen alle angezeigt werden.

   b) Wird statt eines Namens eine numerische Adresse beim Aufruf mitgegeben, so soll eine Fehlermeldung ausgegeben werden, wenn der Hostname nicht ermittelt werden konnte, ansonsten soll der zugeordnete Hostname angezeigt werden.

2. Schreiben Sie das Programm `Localhost`, das den Hostnamen und die IP-Adresse des lokalen Rechners in einem Fenster anzeigt.

   Um `Localhost` mit einem Doppelklick auf einem grafischen Symbol starten zu können, muss eine jar-Datei erzeugt werden:

   ```
   jar --create --file localhost.jar --main-class Localhost
          -C out/production/Aufgaben-1 Localhost.class
   ```

3. Entwickeln Sie ein Programm, das die Netzklasse (A, B, C, D oder E) einer IP-Adresse ermittelt.

2 Datenaustausch mit JSON

JSON (*JavaScript Object Notation*) ist ein platzsparendes Datenaustauschformat, das für Menschen einfach zu lesen und zu schreiben ist. Für alle gängigen Programmiersprachen gibt es Tools, um Datenstrukturen im JSON-Format zu analysieren und zu generieren. JSON-Dokumente können direkt von JavaScript interpretiert werden.[1]

In den folgenden Kapiteln werden JSON-Dokumente häufig für den Austausch von Daten zwischen Client und Server benutzt.

Struktur und Datentypen

Ein JSON-Dokument beruht im Allgemeinen auf einer ungeordneten Menge von Name/Wert-Paaren (*Objekte*) und einer geordneten Liste von Werten (*Arrays*).

Ein Wert kann

- ein Objekt,
- ein Array,
- eine Zeichenkette,
- eine Zahl,
- ein boolescher Wert (`true`, `false`) oder
- `null` sein.

Strukturen können auch ineinander verschachtelt sein. Ein Objekt beginnt mit `{` und endet mit `}`. Jedem Namen in doppelten Anführungszeichen folgt ein Doppelpunkt. Paare sind durch Komma getrennt. Ein Array beginnt mit `[` und endet mit `]`. Die Werte sind durch Komma getrennt.

Beispiel:
Das folgende JSON-Dokument beschreibt eine Person mit Id, Name, Alter und Hobbys.

```
{
  "id": 4711,
  "name": "Meier",
  "age": 28,
  "hobbies": [
    "Sport",
    "Lesen",
    "Musik"
  ]
}
```

Um Java-Objekte leicht in JSON-Dokumente zu konvertieren bzw. aus JSON-Dokumenten wiederum Objekte zu erhalten, verwenden wir ein spezielles Java-API für das so genannte *JSON Binding*.

1 Siehe auch: `https://www.json.org/json-de.html`

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_2

2.1 Java API for JSON Binding

JSON-B (*Java API for JSON Binding*) ist ein Standard (spezifiziert in JSR 367) zur Konvertierung von Java Objekten in JSON-Dokumente und umgekehrt.[2] Die Referenzimplementierung *Yasson* wurde von *Eclipse.org* entwickelt.[3]

Das Begleitmaterial zu diesem Buch enthält im Verzeichnis *libs/json* die erforderlichen jar-Dateien:

```
javax.json-1.1.2.jar
javax.json-api-1.1.2.jar
javax.json.bind-api-1.0.jar
yasson-1.0.jar
```

In IntelliJ IDEA kann unter *Project Structure* eine *Global Library* z. B. mit dem Namen *json* eingerichtet werden, die auf dieses Verzeichnis verweist. Diese kann dann im folgenden Projekt bzw. in den späteren Projekten, die JSON-B benötigen, leicht als *Dependency* zugewiesen werden (siehe auch den Anhang).

→ Projekt jsonb

Die Programme des Projekts *jsonb* bieten eine Einführung in die Verwendung von JSON-B anhand von Beispielen.

Das folgende Programm zeigt, wie Java-Objekte einfacher Typen wie Array, `List`, `Set` und `Map` konvertiert werden können.

Die statische Methode `create` des Interface `javax.json.bind.JsonbBuilder` erzeugt eine Instanz vom Typ des Interface `javax.json.bind.Jsonb`, mit der Objekte geschrieben (`toJson`) und JSON-Dokumente gelesen (`fromJson`) werden können.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class DefaultTypesTest {
   public static void main(String[] args) {
      Jsonb jsonb = JsonbBuilder.create();

      String[] array1 = {"one", "two", "three"};
      List<String> list1 = List.of("one", "two", "three");
      Set<String> set1 = Set.of("one", "two", "three");
      Map<String, Integer> map1 = Map.ofEntries(
            Map.entry("Hugo", 22), Map.entry("Emil", 25), Map.entry("Tim", 18));

      System.out.println("--- Serialize");
      String json1 = jsonb.toJson(array1);
      System.out.println(json1);
```

2 http://json-b.net/

3 https://projects.eclipse.org/projects/ee4j.yasson

```
        String json2 = jsonb.toJson(list1);
        System.out.println(json2);

        String json3 = jsonb.toJson(set1);
        System.out.println(json3);

        String json4 = jsonb.toJson(map1);
        System.out.println(json4);

        System.out.println("--- Deserialize");
        List<String> array2 = jsonb.fromJson(json1, List.class);
        System.out.println(array2);

        List<String> list2 = jsonb.fromJson(json2, List.class);
        System.out.println(list2);

        Set<String> set2 = jsonb.fromJson(json3, Set.class);
        System.out.println(set2);

        Map<Integer, String> map2 = jsonb.fromJson(json4, Map.class);
        System.out.println(map2);
    }
}
```

Ausgabe des Programms:

```
--- Serialize
["one","two","three"]
["one","two","three"]
["three","one","two"]
{"Hugo":22,"Tim":18,"Emil":25}
--- Deserialize
[one, two, three]
[one, two, three]
[one, three, two]
{Hugo=22, Tim=18, Emil=25}
```

Mit Annotationen des Paktes `javax.json.bind.annotation` kann die Konvertierung angepasst werden:

`@JsonbProperty`

Hiermit kann der Name des Feldes bzw. der JavaBean Property für die Darstellung im JSON-Dokument geändert werden. Mit `nillable = true` werden auch `null`-Werte serialisiert (Default ist `false`).

`@JsonbTransient`

verhindert die Serialisierung des Feldes bzw. der JavaBean Property.

`@JsonbNumberFormat`

spezifiziert die Darstellung einer Zahl.

`@JsonbDateFormat`

spezifiziert die Darstellung eines Datums.

Das folgende Beispiel zeigt auch, wie mit Hilfe einer Instanz vom Typ `JsonbConfig` die Konvertierung beeinflusst werden kann:

Binärdaten werden mit Hilfe des Verfahrens *Base64* codiert.[4] Feldnamen werden im JSON-Dokument auf besondere Weise dargestellt: Hier wird z. B. der Name `publicString` in `public-string` konvertiert. Das JSON-Dokument wird formatiert ausgegeben.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;
import javax.json.bind.annotation.JsonbDateFormat;
import javax.json.bind.annotation.JsonbNumberFormat;
import javax.json.bind.annotation.JsonbProperty;
import javax.json.bind.annotation.JsonbTransient;
import javax.json.bind.config.BinaryDataStrategy;
import javax.json.bind.config.PropertyNamingStrategy;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.Arrays;
import java.util.Optional;

public class ConvertTest {
   public String publicString;
   private String privateString;

   @JsonbProperty("writer")
   private String author;

   private String comment1;

   @JsonbProperty(nillable = true)
   private String comment2;

   @JsonbTransient
   private String internal;

   private Priority prio;
   private Optional<String> person1;
   private Optional<String> person2;

   @JsonbNumberFormat(value="#.00", locale="Locale.ENGLISH" )
   private double price;

   @JsonbDateFormat("dd.MM.yyyy")
   private LocalDate date;

   private LocalTime time;
   private LocalDateTime timestamp;
   private byte[] token;
```

4 https://de.wikipedia.org/wiki/Base64

```
public enum Priority {
   LOW, MIDDLE, HIGH
}

public ConvertTest() {
   internal = "internal";
}

public String getPrivateString() {
   return privateString;
}

public void setPrivateString(String privateString) {
   this.privateString = privateString;
}

public String getAuthor() {
   return author;
}

public void setAuthor(String author) {
   this.author = author;
}

public String getComment1() {
   return comment1;
}

public void setComment1(String comment1) {
   this.comment1 = comment1;
}

public String getComment2() {
   return comment2;
}

public void setComment2(String comment2) {
   this.comment2 = comment2;
}

public String getInternal() {
   return internal;
}

public void setInternal(String internal) {
   this.internal = internal;
}

public Priority getPrio() {
   return prio;
}

public void setPrio(Priority prio) {
   this.prio = prio;
}

public Optional<String> getPerson1() {
   return person1;
}
```

```
    public void setPerson1(Optional<String> person1) {
       this.person1 = person1;
    }

    public Optional<String> getPerson2() {
       return person2;
    }

    public void setPerson2(Optional<String> person2) {
       this.person2 = person2;
    }

    public double getPrice() {
       return price;
    }

    public void setPrice(double price) {
       this.price = price;
    }

    public LocalDate getDate() {
       return date;
    }

    public void setDate(LocalDate date) {
       this.date = date;
    }

    public LocalTime getTime() {
       return time;
    }

    public void setTime(LocalTime time) {
       this.time = time;
    }

    public LocalDateTime getTimestamp() {
       return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
       this.timestamp = timestamp;
    }

    public byte[] getToken() {
       return token;
    }

    public void setToken(byte[] token) {
       this.token = token;
    }

    @Override
    public String toString() {
       return "ConvertTest{" +
             "\npublicString='" + publicString + '\'' +
             ", \nprivateString='" + privateString + '\'' +
             ", \nauthor='" + author + '\'' +
```

```
            ", \ncomment1='" + comment1 + '\'' +
            ", \ncomment2='" + comment2 + '\'' +
            ", \ninternal='" + internal + '\'' +
            ", \nprio=" + prio +
            ", \nperson1=" + person1 +
            ", \nperson2=" + person2 +
            ", \nprice=" + price +
            ", \ndate=" + date +
            ", \ntime=" + time +
            ", \ntimestamp=" + timestamp +
            ", \ntoken=" + Arrays.toString(token) +
            '}';
   }

   public static void main(String[] args) throws FileNotFoundException {
      ConvertTest t1 = new ConvertTest();

      t1.publicString = "My public string";
      t1.setPrivateString("My private string");
      t1.setAuthor("Joseph Roth");
      t1.setPrio(Priority.HIGH);
      t1.setPerson1(Optional.of("Hugo"));
      t1.setPerson2(Optional.empty());
      t1.setPrice(10.986);
      t1.setDate(LocalDate.now());
      t1.setTime(LocalTime.of(17, 43));
      t1.setTimestamp(LocalDateTime.of(2018, 1, 27, 17, 46, 10));
      t1.setToken("Mein Token".getBytes());

      JsonbConfig jsonbConfig = new JsonbConfig()
            .withBinaryDataStrategy(BinaryDataStrategy.BASE_64)
            .withPropertyNamingStrategy(
                  PropertyNamingStrategy.LOWER_CASE_WITH_DASHES)
            .withFormatting(true);

      Jsonb jsonb = JsonbBuilder.create(jsonbConfig);
      jsonb.toJson(t1, new FileOutputStream("test.json"));

      ConvertTest t2 = jsonb.fromJson(new FileInputStream("test.json"),
                                 ConvertTest.class);
      System.out.println(t2);
   }
}
```

Das Programm erzeugt die Datei *test.json*:

```
{
  "comment2": null,
  "date": "08.01.2019",
  "person1": "Hugo",
  "price": "10.99",
  "prio": "HIGH",
  "private-string": "My private string",
  "public-string": "My public string",
  "time": "17:43:00",
  "timestamp": "2018-01-27T17:46:10",
  "token": "TWVpbiBUb2tlbg==",
  "writer": "Joseph Roth"
}
```

Ausgabe auf der Konsole:

```
ConvertTest{
publicString='My public string',
privateString='My private string',
author='Joseph Roth',
comment1='null',
comment2='null',
internal='internal',
prio=HIGH,
person1=Optional[Hugo],
person2=null,
price=10.99,
date=2019-01-08,
time=17:43,
timestamp=2018-01-27T17:46:10,
token=[77, 101, 105, 110, 32, 84, 111, 107, 101, 110]}
```

2.2 Serialisierung eines Objektgraphs

Eine Liste von Bestellungen soll serialisiert werden. Dann soll das erzeugte JSON-Dokument wieder als Objektgraph deserialisiert werden.

Zugrunde liegt das folgende Datenmodell:

Abbildung 2-1: Datenmodell Bestellung

→ Projekt jsonb

```
package model;

import javax.json.bind.annotation.JsonbPropertyOrder;

@JsonbPropertyOrder({"id", "name"})
public class Kunde {
   private int id;
   private String name;

   public Kunde() {
   }

   public Kunde(int id, String name) {
      this.id = id;
      this.name = name;
   }
```

```
   // get-/set-Methoden

   @Override
   public String toString() {
      // ...
   }
}


package model;

import javax.json.bind.annotation.JsonbPropertyOrder;

@JsonbPropertyOrder({"id", "preis", "bestand"})
public class Artikel {
   private int id;
   private double preis;
   private int bestand;

   public Artikel() {
   }

   public Artikel(int id, double preis, int bestand) {
      this.id = id;
      this.preis = preis;
      this.bestand = bestand;
   }

   // get-/set-Methoden

   public void addBestand(int menge) {
      this.bestand += menge;
   }

   @Override
   public String toString() {
      // ...
   }
}


package model;

import javax.json.bind.annotation.JsonbPropertyOrder;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@JsonbPropertyOrder({"id", "datum", "kunde", "positionen"})
public class Bestellung {
   private int id;
   private LocalDate datum;
   private Kunde kunde;
   private List<Bestellposition> positionen;

   public Bestellung() {
   }
```

```
   public Bestellung(int id, LocalDate datum) {
      this.id = id;
      this.datum = datum;
      positionen = new ArrayList<Bestellposition>();
   }

   // get-/set-Methoden

   public void addPosition(Bestellposition pos) {
      positionen.add(pos);
   }

   @Override
   public String toString() {
      // ...
   }
}


package model;

import javax.json.bind.annotation.JsonbPropertyOrder;

@JsonbPropertyOrder({"posnr", "menge", "artikel"})
public class Bestellposition {
   private int posnr;
   private int menge;
   private Artikel artikel;

   public Bestellposition() {
   }

   public Bestellposition(int posnr, int menge) {
      this.posnr = posnr;
      this.menge = menge;
   }

   // get-/set-Methoden


   @Override
   public String toString() {
      // ...
   }
}
```

Die Annotation `javax.json.bind.annotation.JsonbPropertyOrder` sorgt dafür, dass die angegebenen Attribute in der vorgegebenen Reihenfolge serialisiert werden.

Serialisierung

```
import model.Artikel;
import model.Bestellposition;
import model.Bestellung;
import model.Kunde;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.time.LocalDate;
import java.util.List;

public class Serialize {
   public static void main(String[] args) throws FileNotFoundException {
      Artikel a1 = new Artikel(4711, 10., 100);
      Artikel a2 = new Artikel(4712, 20., 200);
      Artikel a3 = new Artikel(4713, 30., 300);

      Kunde k1 = new Kunde(1, "Hugo Meier");
      Kunde k2 = new Kunde(2, "Emil Schulz");

      Bestellung b1 = new Bestellung(101, LocalDate.now());
      b1.setKunde(k1);

      Bestellposition p11 = new Bestellposition(1, 3);
      p11.setArtikel(a1);
      b1.addPosition(p11);

      Bestellposition p12 = new Bestellposition(2, 10);
      p12.setArtikel(a2);
      b1.addPosition(p12);

      Bestellung b2 = new Bestellung(102, LocalDate.now());
      b2.setKunde(k2);

      Bestellposition p21 = new Bestellposition(1, 1);
      p21.setArtikel(a1);
      b2.addPosition(p21);

      Bestellposition p22 = new Bestellposition(2, 5);
      p22.setArtikel(a3);
      b2.addPosition(p22);

      List<Bestellung> bestellungen = List.of(b1, b2);

      Jsonb jsonb = JsonbBuilder.create();
      jsonb.toJson(bestellungen, new FileOutputStream("bestellungen.json"));
   }
}
```

Das Programm erzeugt die Datei *bestellung.json*:

```
[
  {
    "id": 101,
    "datum": "2019-01-08",
    "kunde": {
      "id": 1,
      "name": "Hugo Meier"
    },
    "positionen": [
      {
        "posnr": 1,
        "menge": 3,
        "artikel": {
          "id": 4711,
```

```
          "preis": 10.0,
          "bestand": 100
        }
      },
      {
        "posnr": 2,
        "menge": 10,
        "artikel": {
          "id": 4712,
          "preis": 20.0,
          "bestand": 200
        }
      }
    ]
  },
  {
    "id": 102,
    "datum": "2019-01-08",
    "kunde": {
      "id": 2,
      "name": "Emil Schulz"
    },
    "positionen": [
      {
        "posnr": 1,
        "menge": 1,
        "artikel": {
          "id": 4711,
          "preis": 10.0,
          "bestand": 100
        }
      },
      {
        "posnr": 2,
        "menge": 5,
        "artikel": {
          "id": 4713,
          "preis": 30.0,
          "bestand": 300
        }
      }
    ]
  }
]
```

Deserialisierung

```
import model.Bestellposition;
import model.Bestellung;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;
```

```
public class Deserialize {
   public static void main(String[] args) throws FileNotFoundException {
      Jsonb jsonb = JsonbBuilder.create();
      Type type = new ArrayList<Bestellung>() {}.getClass().getGenericSuperclass();
      List<Bestellung> bestellungen = jsonb.fromJson(
                        new FileInputStream("bestellungen.json"), type);

      for (Bestellung b : bestellungen) {
         System.out.println(b.getKunde());
         System.out.println("\t" + b);
         for (Bestellposition bpos : b.getPositionen()) {
            System.out.println("\t\t" + bpos);
            System.out.println("\t\t\t" + bpos.getArtikel());
         }
         System.out.println();
      }
   }
}
```

Das zweite Argument im Aufruf der Methode `fromJson` ist normalerweise der Typ des so genannten Root-Objekts der JSON-Struktur. `List.class` an dieser Stelle funktioniert jedoch nicht, da die Information zum Typparameter `Bestellung` zur Laufzeit nicht mehr vorhanden ist. Diese wird jedoch für die Deserialisierung benötigt.

```
ArrayList<Bestellung> list = new ArrayList<Bestellung>() {};
Type type = list.getClass().getGenericSuperclass();
```

`getGenericSuperclass` liefert hier den Parametertyp der Superklasse:

```
java.util.ArrayList<model.Bestellung>
```

Ausgabe des Programms:

```
Kunde{id=1, name='Hugo Meier'}
   Bestellung{id=101, datum=2019-01-08}
      Bestellposition{posnr=1, menge=3}
         Artikel{id=4711, preis=10.0, bestand=100}
      Bestellposition{posnr=2, menge=10}
         Artikel{id=4712, preis=20.0, bestand=200}

Kunde{id=2, name='Emil Schulz'}
   Bestellung{id=102, datum=2019-01-08}
      Bestellposition{posnr=1, menge=1}
         Artikel{id=4711, preis=10.0, bestand=100}
      Bestellposition{posnr=2, menge=5}
         Artikel{id=4713, preis=30.0, bestand=300}
```

2.3 Aufgaben

1. Entwickeln Sie die Klasse `Person` mit den Eigenschaften *id*, *name*, *age*, *hobbies* (Liste aus Strings). Erzeugen Sie aus einem `Person`-Objekt ein in einer Datei gespeichertes JSON-Dokument. Konvertieren Sie dann umgekehrt dieses Dokument in ein `Person`-Objekt.

2. Entwickeln Sie die Klasse `MyImage` mit den Eigenschaften *name* (`String`), *type* (`String`), *data* (`byte[]`). Schreiben Sie ein Programm, das eine Liste von kleinen Bildern vom Typ `MyImage` in ein JSON-Dokument konvertiert. Dabei sollen die Binärdaten mit *Base64* codiert werden. Entwickeln Sie auch ein Programm, das aus dem JSON-Dokument Grafikdateien erzeugt.

3 Kommunikation mit UDP

Das *User Datagram Protocol* (*UDP*) stellt grundlegende Funktionen zur Verfügung, um mit geringem Aufwand Daten zwischen kommunizierenden Prozessen austauschen zu können. UDP ist als Transportprotokoll der dritten Schicht im TCP/IP-Schichtenmodell zugeordnet und nutzt den Vermittlungsdienst IP.

3.1 Das Protokoll UDP

UDP ist ein verbindungsloses Protokoll, d. h. es bietet keinerlei Kontrolle über die sichere Ankunft versendeter Datenpakete, so genannter Datagramme. UDP ist im *RFC* (*Request for Comments*) 768 der *IETF* (*Internet Engineering Task Force*) spezifiziert.[1]

Datagramme müssen nicht den Empfänger in der Reihenfolge erreichen, in der sie abgeschickt werden. Datagramme können verloren gehen, weder Sender noch Empfänger werden über den Verlust informiert. Fehlerhafte Datagramme werden nicht nochmals übertragen. Abbildung 3-1 zeigt den Aufbau des *UDP-Datagramms*, das in ein IPv4-Paket eingebettet ist.

Abbildung 3-1: Aufbau des UDP-Datagramms

1 https://tools.ietf.org/html/rfc768

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_3

Der UDP-Header enthält vier 16 Bit lange Felder:

- die Portnummer des Senders,
- die Portnummer des Empfängers,
- die Gesamtlänge des UDP-Datagramms und
- die Prüfsumme.

Darauf folgen dann die eigentlichen Nutzdaten des Datagramms.

Die Maximallänge eines UDP-Datagramms ist begrenzt. Anwendungen sollten nicht mehr als 508 Byte Nutzdaten in einem Datagramm versenden, um Probleme infolge einer evtl. nötigen Fragmentierung des IP-Pakets durch Router zu vermeiden.

UDP ist ein einfaches Mittel, um Nachrichten, die keine zuverlässige Übertragung erfordern, zu versenden. UDP hat gegenüber TCP den Vorteil eines viel geringeren Kontroll-Overheads.

UDP-Anwendungen

Einige bekannte Anwendungen, die UDP nutzen, sind:

- das *Domain Name System* (*DNS*) zur automatischen Konvertierung von Domain-Namen (Rechnernamen) in IP-Adressen und umgekehrt,
- das *Simple Network Management Protocol* (*SNMP*) zur Verwaltung von Ressourcen in einem TCP/IP-basierten Netz,
- das *Trivial File Transfer Protocol* (*TFTP*), das einen einfachen Dateitransferdienst zum Sichern und Laden von System- und Konfigurationsdateien bietet,
- das *Dynamic Host Configuration Protocol* (*DHCP*) zur dynamischen Zuweisung von IP-Adressen.

Die unidirektionale Übertragung von Video- und Audiosequenzen zur Wiedergabe in Echtzeit (*Multimedia Streaming*) ist besonders zeitkritisch. Der gelegentliche Verlust eines Datenpakets kann aber toleriert werden. Hier ist UDP sehr gut geeignet.

3.2 DatagramSocket und DatagramPacket

Ein *UDP-Socket* wird an eine Portnummer auf dem lokalen Rechner gebunden. Er kann dann Datagramme an jeden beliebigen anderen UDP-Socket senden und von jedem beliebigen UDP-Socket empfangen (siehe Abbildung 3-2).

DatagramSocket

Die Klasse `java.net.DatagramSocket` repräsentiert einen UDP-Socket zum Senden und Empfangen von UDP-Datagrammen. Ein UDP-Socket muss an einen Port des lokalen Rechners gebunden werden.

Abbildung 3-2: UDP-Sockets

Hierzu stehen Konstruktoren zur Verfügung:

`DatagramSocket() throws SocketException`
: erzeugt einen UDP-Socket und bindet ihn an einen verfügbaren Port des lokalen Rechners.

`DatagramSocket(int port) throws SocketException`
: erzeugt einen UDP-Socket und bindet ihn an die Portnummer `port` des lokalen Rechners. Hierdurch kann ein Server den Port, den er nutzen möchte, festlegen.

`DatagramSocket(int port, InetAddress laddr) throws SocketException`
`DatagramSocket(SocketAddress bindaddr) throws SocketException`
: Diese beiden Konstruktoren binden den Socket an eine vorgegebene Adresse.

Alle Konstruktoren können bei Fehlern im zugrunde liegenden Protokoll die Ausnahme `java.net.SocketException` auslösen. `SocketException` ist Subklasse von `java.io.IOException`.

Einige `DatagramSocket`-Methoden:

`void close()`
: schließt den Socket.

`InetAddress getLocalAddress()`
: liefert die IP-Adresse, an die der Socket gebunden ist.

`int getLocalPort()`
: liefert die Portnummer, an die der Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
: liefert die Socket-Adresse, an die der Socket gebunden ist.

DatagramPacket

Die Klasse `java.net.DatagramPacket` repräsentiert ein UDP-Datagramm.

Der Konstruktor

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

erzeugt ein UDP-Datagramm zum Senden von Daten. `buf` enthält die Daten, `address` ist die IP-Adresse und `port` die Portnummer des Empfängers. Die Anzahl `length` der zu übertragenden Bytes muss kleiner oder gleich der Pufferlänge sein.

Ein ähnlicher Konstruktor, der eine Socket-Adresse verwendet:

```
DatagramPacket(byte[] buf, int length, SocketAddress address)
    throws SocketException
```

Der Konstruktor

```
DatagramPacket(byte[] buf, int length)
```

erzeugt ein UDP-Datagramm zum Empfangen von Daten. `buf` ist der Puffer, der die empfangenen Daten aufnimmt. `length` legt fest, wie viele Bytes maximal gelesen werden sollen. `length` muss kleiner oder gleich der Pufferlänge sein.

Wendet man auf dieses Paket die Methoden `setAddress` und `setPort` (siehe weiter unten) an, so kann es auch gesendet werden.

Senden und empfangen

Die `DatagramSocket`-Methode

```
void send(DatagramPacket p) throws IOException
```

sendet ein UDP-Datagramm von diesem Socket aus.

Die `DatagramSocket`-Methode

```
void receive(DatagramPacket p) throws IOException
```

empfängt ein UDP-Datagramm von diesem Socket. `p` enthält die Daten sowie die IP-Adresse und die Portnummer des Senders. Die Methode blockiert so lange, bis ein Datagramm empfangen wurde.

Weitere `DatagramPacket`-Methoden sind:

`byte[] getData()`
: liefert den Datenpufferinhalt.

`int getLength()`
: liefert die Länge der Daten, die empfangen wurden bzw. gesendet werden sollen.

`InetAddress getAddress()`
: liefert die (entfernte) IP-Adresse des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`int getPort()`
: liefert die (entfernte) Portnummer des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`SocketAddress getSocketAddress()`
: liefert die (entfernte) Socket-Adresse des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`void setData(byte[] buf)`
: setzt den Datenpuffer des Datagramms.

`void setLength(int length)`
: bestimmt die Anzahl Bytes, die gesendet werden sollen bzw. die maximale Anzahl Bytes, die empfangen werden sollen. `length` muss kleiner oder gleich der Pufferlänge des Datagramms sein.

`void setAddress(InetAddress address)`
: setzt die IP-Adresse des Rechners, an den das Datagramm gesendet werden soll.

`void setPort(int port)`
: setzt die Portnummer, an die das Datagramm gesendet werden soll.

`void setSocketAddress(SocketAddress address)`
: setzt die Socket-Adresse des Rechners, an den das Datagramm gesendet werden soll.

In allen Beispielen dieses Kapitels ist die Bearbeitungszeit des Servers nach Eintreffen eines Pakets relativ kurz. Deshalb ist eine iterative Implementierung (Paket empfangen – Anfrage bearbeiten – Paket senden) ausreichend.

Bei längeren Bearbeitungszeiten sollten für die Bearbeitung und das Senden der Antwort zu einer Anfrage Threads benutzt werden (parallele Implementierung), um die Wartezeit eines Clients zu reduzieren.

3.3 Nachrichten senden und empfangen

Das erste Programmbeispiel ist eine einfache Client/Server-Anwendung, die den Umgang mit den Methoden des vorigen Abschnitts zeigen soll.

→ Projekt messages

Der Client sendet eine Nachricht, die der Server entgegennimmt und auf der Konsole ausgibt.

Die Nachricht ist ein Objekt vom Typ `Message`. Der Client konvertiert das Objekt für die Übertragung in ein JSON-Dokument (*Serialisierung*). Der Server konvertiert dieses Dokument wiederum in ein Objekt (*Deserialisierung*).

Das IntelliJ-Projekt *messages* enthält die beiden Unterprojekte (Module) *client* und *server*, die beide die Library *libs/json* (aus Kapitel 2) als Dependency einbinden.

```
import java.time.LocalDateTime;

public class Message {
   private String user;
   private LocalDateTime timestamp;
   private String message;

   public Message() {
   }

   public Message(String user, LocalDateTime timestamp, String message) {
      this.user = user;
      this.timestamp = timestamp;
      this.message = message;
   }

   public String getUser() {
      return user;
   }

   public void setUser(String user) {
      this.user = user;
   }

   public LocalDateTime getTimestamp() {
      return timestamp;
   }

   public void setTimestamp(LocalDateTime timestamp) {
      this.timestamp = timestamp;
   }

   public String getMessage() {
      return message;
   }

   public void setMessage(String message) {
      this.message = message;
   }

   @Override
   public String toString() {
      return "Message{" +
            "user='" + user + '\'' +
            ", timestamp=" + timestamp +
            ", message='" + message + '\'' +
            '}';
   }
}
```

Klasse `ReceiveMessages` aus Modul *server*:

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
```

```
public class ReceiveMessages {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      int port = 50000;

      try (DatagramSocket socket = new DatagramSocket(port)) {
         DatagramPacket packet = new DatagramPacket(new byte[BUFSIZE], BUFSIZE);

         while (true) {
            socket.receive(packet);
            String data = new String(packet.getData(), 0, packet.getLength());
            Message message = deserialize(data);
            System.out.println(message);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private static Message deserialize(String data) {
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.fromJson(data, Message.class);
   }
}
```

Klasse `SendMessage` aus Modul *client*:

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.time.LocalDateTime;

public class SendMessage {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) throws Exception {
      String host = "localhost";
      int port = 50000;

      String message = createMessage();

      try (DatagramSocket socket = new DatagramSocket()) {
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket packet = new DatagramPacket(
               new byte[BUFSIZE], BUFSIZE, addr, port);
         byte[] data = message.getBytes();
         packet.setData(data);
         packet.setLength(data.length);
         socket.send(packet);
      }
   }
```

```
   private static String createMessage() {
      Message msg = new Message("Hugo", LocalDateTime.now(), "Das ist ein Test.");
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.toJson(msg);
   }
}
```

Zu Beginn des Programms `ReceiveMessages` wird ein UDP-Socket erzeugt und an die Portnummer 50000 gebunden. Es wird ein Datagramm zum Empfangen von Daten erzeugt. In einer Endlos-Schleife werden Daten empfangen, konvertiert und ausgegeben. Das Programm kann über Tastatur mit *Strg + C* abgebrochen werden.

Aufruf des Servers:

```
java -cp out/production/server;../../libs/json/* ReceiveMessages
```
[2]

Das Programm `SendMessage` erzeugt einen UDP-Socket sowie ein Datagramm zum Senden von Daten an den Server. Hierzu werden IP-Adresse und Portnummer des Servers eingetragen. Das Datagramm wird mit den Daten (JSON-Dokument) bestückt und an den Server gesendet.

Aufruf des Clients:

```
java -cp out/production/client;../../libs/json/* SendMessage
```

3.4 Anfrage und Antwort

Im folgenden Beispiel schickt der Client einen Text zum Server und erhält den mit dem Algorithmus SHA-256 erzeugten Hash-Wert des Textes zurück. Anfrage und Antwort werden in einem JSON-Dokument transportiert.

Das Beispiel zeigt auch, wie eine Verbindung auf eine bestimmte Adresse (IP-Adresse und Portnummer) eingeschränkt werden kann.

Erhält der Client nicht innerhalb von 10 Sekunden eine Antwort vom Server, wird eine Ausnahme ausgelöst und das Programm beendet.

Verbindung einschränken

Die Klasse `DatagramSocket` enthält eine Methode, die es ermöglicht, die Adresse, an die Datagramme gesendet werden bzw. von der Datagramme empfangen werden, gezielt auszuwählen und alle anderen Datagramme zu verwerfen:

---

2 In IntelliJ-Projekten liegt der Bytecode standardmäßig im Verzeichnis `out/production/<Projekt bzw. Modul>`.

Das Verzeichnis *libs* enthält "globale" Bibliotheken wie z. B. *json*. Projekte eines Kapitels befinden sich in dem jeweiligen Ordner *Kap01* usw. *libs* ist in dem Ordner, in dem auch die Ordner eines Kapitels wie *Kap01* etc. gespeichert sind, abgelegt.

`void connect(InetAddress address, int port)`
`void connect(SocketAddress addr) throws SocketException`

Datagramme können nur an diese IP-Adresse/Portnummer gesendet bzw. von dieser IP-Adresse/Portnummer empfangen werden.

`void disconnect()`

hebt die durch `connect` hergestellte Restriktion auf.

Mit den `DatagramSocket`-Methoden

```
InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
```

können IP-Adresse/Portnummer, die in `connect` verwendet wurden, abgefragt werden. Wurde `connect` nicht benutzt, wird `null` bzw. `-1` geliefert.

Timeout

Die Zeitsteuerung wird durch Aufruf der folgenden `DatagramSocket`-Methode erreicht:

`void setSoTimeout(int timeout) throws SocketException`

setzt ein Timeout in Millisekunden. `receive` für diesen Socket blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert. `SocketTimeoutException` ist Subklasse von `java.io.InterruptedIOException`.

`int getSoTimeout() throws SocketException`

liefert die Timeout-Angabe.

→ Projekt digest

```
public class Digest {
   private String text;
   private String hashValue;

   public Digest() {
   }

   public Digest(String text) {
      this.text = text;
   }

   public String getText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }

   public String getHashValue() {
      return hashValue;
   }
```

```
   public void setHashValue(String hashValue) {
      this.hashValue = hashValue;
   }
}



import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class DigestServer {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      int port = 50000;
      String remoteHost = "localhost";
      int remotePort = 40000;

      try (DatagramSocket socket = new DatagramSocket(port)) {
         // Es werden nur Datagramme dieser Adresse entgegengenommen
         socket.connect(InetAddress.getByName(remoteHost), remotePort);

         // Pakete zum Empfangen bzw. Senden
         DatagramPacket packetIn = new DatagramPacket(
               new byte[BUFSIZE], BUFSIZE);
         DatagramPacket packetOut = new DatagramPacket(
               new byte[BUFSIZE], BUFSIZE);

         System.out.println("Server gestartet ...");

         while (true) {
            // Paket empfangen
            socket.receive(packetIn);
            System.out.println("Received: " + packetIn.getLength() + " bytes");

            String json = new String(packetIn.getData(), 0, packetIn.getLength());

            Digest digest = deserialize(json);
            String hashValue = digest(digest.getText());
            digest.setHashValue(hashValue);

            byte[] data = serialize(digest).getBytes();

            // Daten und Länge im Antwortpaket speichern
            packetOut.setData(data);
            packetOut.setLength(data.length);

            // Zieladresse und Zielport im Antwortpaket setzen
            packetOut.setSocketAddress(packetIn.getSocketAddress());

            // Antwortpaket senden
            socket.send(packetOut);
         }
```

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

   private static Digest deserialize(String data) {
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.fromJson(data, Digest.class);
   }

   private static String serialize(Digest digest) {
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.toJson(digest);
   }

   private static String digest(String data) throws NoSuchAlgorithmException {
      MessageDigest md = MessageDigest.getInstance("SHA-256");
      md.update(data.getBytes());
      byte[] digest = md.digest();
      StringBuilder sb = new StringBuilder();
      for (byte b : digest) {
         sb.append(String.format("%02x", b));
      }
      return sb.toString();
   }
}
```

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketTimeoutException;

public class DigestClient {
   private static final int BUFSIZE = 508;
   private static final int TIMEOUT = 10000;

   public static void main(String[] args) {
      int localPort = 40000;
      String host = "localhost";
      int port = 50000;
      String text = "Das ist ein Test.";

      try (DatagramSocket socket = new DatagramSocket(localPort)) {
         // Maximal TIMEOUT Millisekunden auf Antwort warten
         socket.setSoTimeout(TIMEOUT);

         byte[] data = serialize(new Digest(text)).getBytes();

         // Paket an Server senden
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket packetOut = new DatagramPacket(
               data, data.length, addr, port);
         socket.send(packetOut);
```

```
            // Antwortpaket empfangen
            DatagramPacket packetIn = new DatagramPacket(
                  new byte[BUFSIZE],
                  BUFSIZE);
            socket.receive(packetIn);

            String json = new String(packetIn.getData(), 0, packetIn.getLength());
            Digest digest = deserialize(json);
            System.out.println(digest.getText());
            System.out.println(digest.getHashValue());
         } catch (SocketTimeoutException e) {
            System.err.println("Timeout: " + e.getMessage());
         } catch (Exception e) {
            System.err.println(e);
         }
      }

      private static Digest deserialize(String data) {
         Jsonb jsonb = JsonbBuilder.create();
         return jsonb.fromJson(data, Digest.class);
      }

      private static String serialize(Digest digest) {
         Jsonb jsonb = JsonbBuilder.create();
         return jsonb.toJson(digest);
      }
}
```

Der Client bindet den UDP-Socket an die vorgegebene Portnummer 40000. Wird diese Nummer geändert (z. B. 40001), so erhält der Client keine Antwort und bricht nach 10 Sekunden mit der Meldung *Timeout: Receive timed out* ab.

3.5 Punkt-zu-Mehrpunkt-Verbindungen

Bisher haben wir in diesem Kapitel ausschließlich *Punkt-zu-Punkt-Verbindungen* (*Unicast*) betrachtet. Unter *Multicast* versteht man die Übertragung von einem Teilnehmer zu einer Gruppe von Teilnehmern (*Punkt-zu-Mehrpunkt-Verbindung*).

Der Vorteil dieses Verfahrens liegt darin, dass gleichzeitig Pakete über eine einzige Adresse an mehrere Teilnehmer gesendet werden. Die Vervielfältigung der Pakete findet nicht beim Sender statt, sondern erst an jedem Verteiler (Switch, Router) auf dem Übertragungsweg. So wird *Multicasting* insbesondere zur Übertragung von Audio- und Videoströmen eingesetzt, um die Netzbelastung zu reduzieren.

Im Internet werden für das Multicasting bei IPv4 die IP-Adressen der Klasse D verwendet: `224.0.0.0 - 239.255.255.255`

Eine *Multicast-Gruppe* wird durch eine Adresse dieses Bereichs (*Multicast-Adresse*) und eine UDP-Portnummer repräsentiert. Adressen im Bereich `224.0.0.0 - 224.255.255.255` sind reserviert und sollten nicht benutzt werden.

Teilnehmer können jederzeit einer Multicast-Gruppe beitreten oder diese wieder verlassen. Wird eine Nachricht an eine Multicast-Gruppe gesendet, erhalten alle Gruppenteilnehmer diese Nachricht, sofern sie sich im Time-to-Live-Bereich des Pakets befinden. Das Feld *Time-to-Live* (*TTL*) im Header eines IP-Pakets gibt an, wie lange ein Paket im Internet verbleiben darf. Es handelt sich um einen Zähler, der von jedem Router, den das Paket passiert, um 1 dekrementiert wird. Bei einem Wert von 0 wird das Paket zerstört. Die Reichweite der gesendeten Pakete kann also durch die Angabe des Time-to-Live-Wertes eingeschränkt werden.

Abbildung 3-3: Multicasting

Die Klasse `java.net.MulticastSocket` (Subklasse von `DatagramSocket`) wird zum Senden und Empfangen von Mulitcast-Datagrammen verwendet.

`void setTimeToLive(int ttl) throws IOException`
: setzt den Time-To-Live-Wert. Zulässiger Wert: `0 <= ttl <= 255`. Voreinstellung ist `1`.

`void joinGroup(InetAddress mcastaddr) throws IOException`
: tritt der Multicast-Gruppe bei.

`void leaveGroup(InetAddress mcastaddr) throws IOException`
: verlässt die Multicast-Gruppe.

Ein Client, der ein Datagramm an eine Multicast-Gruppe sendet, muss selbst nicht Teilnehmer dieser Gruppe sein.

→ Projekt multicast

Im folgenden Beispiel bilden zwei Empfänger eine Multicast-Gruppe (IP-Adresse: `228.5.6.7`, Portnummer: `50000`). Zu diesem Zweck wird im Test das Programm `ReceiveMessages` zweimal gestartet.

Für den Datenaustausch verwenden wir die Klasse `Message` aus dem Projekt *messages*.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.time.LocalDateTime;

public class SendMessage {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) throws Exception {
      String host = "228.5.6.7";
      int port = 50000;

      String message = createMessage();

      try (DatagramSocket socket = new DatagramSocket()) {
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket packet = new DatagramPacket(
               new byte[BUFSIZE], BUFSIZE, addr, port);
         byte[] data = message.getBytes();
         packet.setData(data);
         packet.setLength(data.length);
         socket.send(packet);
      }
   }

   private static String createMessage() {
      Message msg = new Message("Hugo", LocalDateTime.now(), "Das ist ein Test.");
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.toJson(msg);
   }
}
```

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.time.LocalDateTime;

public class ReceiveMessages {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      String multicastAddress = "228.5.6.7";
      int port = 50000;

      try (MulticastSocket socket = new MulticastSocket(port)) {
         socket.setTimeToLive(1);
         InetAddress group = InetAddress.getByName(multicastAddress);
         socket.joinGroup(group);

         DatagramPacket packet = new DatagramPacket(
               new byte[BUFSIZE], BUFSIZE);
```

```
            System.out.println("Receiver gestartet: " + LocalDateTime.now());

            while (true) {
               socket.receive(packet);
               String data = new String(packet.getData(), 0, packet.getLength());
               Message message = deserialize(data);
               System.out.println(message);
            }
         } catch (IOException e) {
            System.err.println(e);
         }
      }

      private static Message deserialize(String data) {
         Jsonb jsonb = JsonbBuilder.create();
         return jsonb.fromJson(data, Message.class);
      }
   }
```

Zum Testen auf einem lokalen Windows-Rechner muss das Netzwerk-Interface mit einer IP-Adresse konfiguriert sein. Ggf. sind Firewall-Einstellungen anzupassen.

Das Programm `ReceiveMessages` wird zweimal gestartet (jeweils in einem eigenen Terminal-Fenster):

```
java -cp out/production/server;../../libs/json/* ReceiveMessages

java -cp out/production/client;../../libs/json/* SendMessage
```

Ausgabebeispiel:

```
Receiver gestartet: 2019-01-10T14:40:22.433715200
Message{user='Hugo', timestamp=2019-01-10T14:40:40.691324700,
        message='Das ist ein Test.'}

Receiver gestartet: 2019-01-10T14:40:28.897193300
Message{user='Hugo', timestamp=2019-01-10T14:40:40.691324700,
        message='Das ist ein Test.'}
```

3.6 Aufgaben

1. Ein Server (`QuoteServer`) liefert zufallsgesteuert Zitate. Alle Zitate befinden sich in einer Datei *quotes.txt*, die den folgenden Zeilenaufbau hat:

   ```
   Zitat#Autor
   ```

   Erhält der Server ein "leeres" Datagramm, so bestimmt eine Zufallszahl, welches Zitat (mit Autor) an den Client gesendet wird.

   Programmieren Sie auch eine Anwendung mit grafischer Oberfläche (*JavaFX*) zur Anzeige von Zitaten in regelmäßigen Zeitabständen.

   Aufruf:

   ```
   java -Dfile.encoding=UTF-8 -cp out/production/client;../../libs/json/*
   -p %PATH_TO_FX% --add-modules=javafx.controls QuoteApp
   ```

Die Umgebungsvariable `PATH_TO_FX` zeigt hier auf das *lib*-Verzeichnis der JavaFX-Installation.

2. Entwickeln Sie das Programm `Talk`, mit dem zwei Benutzer an verschiedenen Rechnern eine Unterhaltung online führen können. Das Programm soll eine grafische Oberfläche haben. Verwenden Sie z. B. *JavaFX*.

   Szenario:

   Benutzer "Hugo" sitzt am Rechner A (IP-Adresse `192.168.2.90`), Benutzer "Emil" am Rechner B (IP-Adresse `192.168.2.91`).

   Benutzer "Hugo" ruft das Programm wie folgt auf (die Umgebungsvariable `PATH_TO_FX` zeigt auf das *lib*-Verzeichnis der JavaFX-Installation):

   ```
   java -cp out/production/Aufgabe-3.2 -p %PATH_TO_FX%
   --add-modules=javafx.controls Talk --user=Hugo --localPort=50000 --
   remoteHost=192.168.2.91 --remotePort=50000
   ```

   Benutzer "Emil" gibt das folgende Kommando ein:

   ```
   java -cp out/production/Aufgabe-3.2 -p %PATH_TO_FX%
   --add-modules=javafx.controls Talk --user=Emil --localPort=50000 --
   remoteHost=192.168.2.90 --remotePort=50000
   ```

   Der UDP-Socket wird hier jeweils an die explizit vorgegebene Portnummer `50000` gebunden.

   Zum Testen kann das Programm auch zweimal auf demselben Rechner (`localhost`) gestartet werden. Allerdings müssen dann die Portnummern unterschiedlich sein:

   ```
   java -cp out/production/Aufgabe-3.2 -p %PATH_TO_FX%
   --add-modules=javafx.controls Talk --user=Hugo --localPort=40000 --
   remoteHost=localhost --remotePort=50000
   ```

   ```
   java -cp out/production/Aufgabe-3.2 -p %PATH_TO_FX%
   --add-modules=javafx.controls Talk --user=Emil --localPort=50000 --
   remoteHost=localhost –remotePort=40000
   ```

3. Erstellen Sie eine Anwendung mit zwei Servern und einem Client, die Suchanfragen an eine verteilte Dateiablage simuliert. Die beiden Server sollen zu einer *Multicast-Gruppe*n gehören und jeweils eine Map verwalten mit dem Schlüssel "Datei-Id" und dem Wert "Dateiname". Der Client richtet Suchanfragen an beide Server, indem er die Id der Datei, nach der er suchen will, mitgibt. Derjenige Server, der die Datei in seiner Hashtable verwaltet, antwortet mit einer entsprechenden Nachricht. Ein Timeout von 10 Sekunden signalisiert, dass die Datei auf keinem der beiden Server vorhanden ist.

4 Netzwerkprogrammierung mit TCP-Sockets

Das wichtigste Protokoll der Transportschicht im TCP/IP-Schichtenmodell ist das *Transmission Control Protocol* (*TCP*). Es ist aufwändiger als UDP, stellt aber dafür eine verlässliche Verbindung zwischen Client und Server her. Viele bekannte Internet-Dienste wie FTP (File Transfer Protocol), Telnet, SMTP (Simple Mail Transfer Protocol), POP (Post Office Protocol) und HTTP (Hypertext Transfer Protocol) nutzen TCP.

4.1 Das Protokoll TCP

TCP ist im Gegensatz zu UDP ein verbindungsorientiertes Protokoll. Zwischen zwei Prozessen (Client und Server) wird eine virtuelle Verbindung hergestellt, über die in Form eines bidirektionalen Datenstroms Bytefolgen beliebiger Länge geschickt werden können.

Vor der Übertragung von Daten muss der Client eine Verbindung zum Server anfordern. Wird die Verbindung nicht mehr gebraucht, fordert einer der beteiligten Prozesse TCP auf, sie abzubauen.

Abbildung 4-1: TCP-Verbindung

TCP sorgt für die Aufteilung der Daten in einzelne Pakete, garantiert, dass die Pakete den Empfänger in der richtigen Reihenfolge erreichen, und initiiert die Neuübertragung von verloren gegangenen oder defekten Paketen. TCP ist im *RFC* 793 der *IETF* spezifiziert.[1]

Abbildung 4-2 zeigt den Aufbau eines TCP-Pakets, das Bestandteil der Nutzdaten des IPv4-Pakets ist.

Der TCP-Header enthält:

- die Portnummer des Senders (2 Byte),
- die Portnummer des Empfängers (2 Byte),
- weitere Felder mit einer Gesamtlänge von 16 Byte wie z. B. Reihenfolgenummer und Prüfsumme,
- optionale Felder variabler Länge.

Darauf folgen dann die eigentlichen Nutzdaten des TCP-Pakets.

1 https://tools.ietf.org/html/rfc793

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_4

Abbildung 4-2: Aufbau des TCP-Pakets

4.2 TCP-Sockets

Im Gegensatz zu UDP wird bei TCP zwischen Client-Socket und Server-Socket unterschieden.

Der Server-Socket versetzt den Server in die Lage, die Verbindungsanforderungen der Clients abzuhören und diese dann zu bedienen.

Bevor Daten zwischen Client und Server ausgetauscht werden können, muss eine Verbindung zwischen Sockets hergestellt werden. Der Aufruf der Methode `accept` des Server-Sockets blockiert den Server so lange, bis ein Client versucht, Kontakt aufzunehmen. `accept` liefert als Ergebnis einen Socket, über den dann die eigentliche Kommunikation stattfindet.

Die Kontaktaufnahme auf der Client-Seite geschieht automatisch bei der Erzeugung des Client-Sockets mit einem geeigneten Konstruktor. Eine Verbindung kann natürlich nur hergestellt werden, wenn der Server zeitlich vor dem Start des Clients `accept` aufgerufen hat. Über Ein- und Ausgabeströme können nun Daten empfangen bzw. gesendet werden.

Die Verbindung wird abgebaut, wenn einer der beiden Sockets geschlossen wird.

Die Klasse Socket

Die Klasse `java.net.Socket` implementiert einen Client-Socket.

`Socket(InetAddress address, int port) throws IOException`
: erzeugt einen Client-Socket und bindet ihn an die vorgegebene IP-Adresse und Portnummer.

`Socket(String host, int port) throws java.net.UnknownHostException, IOException`
erzeugt einen Client-Socket und bindet ihn an den vorgegebenen Host und an die vorgegebene Portnummer.

`void close() throws IOException`
schließt den Socket und auch die mit ihm verbundenen Ein- und Ausgabeströme (siehe weiter unten).

`boolean isClosed()`
liefert true, falls der Socket geschlossen ist, sonst false.

`void setSoTimeout(int timeout) throws SocketException`
setzt ein Timeout in Millisekunden. `read` für den Eingabestrom dieses Sockets blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert.

`int getSoTimeout() throws SocketException`
liefert die Timeout-Angabe.

`InetAddress getLocalAddress()`
liefert die lokale IP-Adresse, an die der Socket gebunden ist.

`int getLocalPort()`
liefert die lokale Portnummer, an die der Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
liefert die lokale Socket-Adresse, an die der Socket gebunden ist.

`InetAddress getInetAddress()`
liefert die entfernte IP-Adresse, mit der der Socket verbunden ist.

`int getPort()`
liefert die entfernte Portnummer, mit der der Socket verbunden ist.

`SocketAddress getRemoteSocketAddress()`
liefert die entfernte Socket-Adresse, mit der der Socket verbunden ist.

Input/Output

`InputStream getInputStream() throws IOException`
liefert einen Eingabestrom für diesen Socket.

`OutputStream getOutputStream() throws IOException`
liefert einen Ausgabestrom für diesen Socket.

Wird einer der beiden Datenströme mit der entsprechenden `close`-Methode geschlossen, so wird auch der zugehörige Socket geschlossen.

Mit einer der folgenden Methoden kann jeweils nur einer der beiden Datenströme (Eingabe bzw. Ausgabe) geschlossen werden, ohne damit den Socket selbst zu schließen:

`void shutdownInput() throws IOException`
setzt den Eingabestrom für diesen Socket auf EOF (End Of File).

`void shutdownOutput() throws IOException`
deaktiviert den Ausgabestrom für diesen Socket.

Die Klasse ServerSocket

Die Klasse `java.net.ServerSocket` implementiert einen Server-Socket.

`ServerSocket()`
erzeugt einen ungebundenen Server-Socket.

`ServerSocket(int port) throws IOException`
erzeugt einen Server-Socket und bindet ihn an die vorgegebene Portnummer.

`void bind(SocketAddress addr) throws IOException`
bindet den Server-Socket an eine Socket-Adresse (IP-Adresse und Portnummer).

`void close() throws IOException`
schließt den Server-Socket.

`boolean isClosed()`
liefert `true`, falls der Server-Socket geschlossen ist, sonst `false`.

`Socket accept() throws IOException`
nimmt einen Verbindungswunsch an und erzeugt einen neuen Socket für diese Verbindung. `accept` blockiert so lange, bis eine neue Verbindung aufgenommen wurde.

`void setSoTimeout(int timeout) throws SocketException`
setzt ein *Timeout* in Millisekunden. `accept` blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert.

`int getSoTimeout() throws IOException`
liefert die Timeout-Angabe.

`InetAddress getInetAddress()`
liefert die IP-Adresse, an die der Server-Socket gebunden ist.

`int getLocalPort()`
liefert die Portnummer, an die der Server-Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
liefert die Socket-Adresse, an die der Server-Socket gebunden ist.

Backlog

Verbindungswünsche, für die mittels `accept` noch kein Socket erzeugt werden konnte, werden in einer Warteschlange (*backlog*) verwaltet. Falls die Warteschlange voll ist und ein neuer Verbindungswunsch eingeht, wird beim Client die Ausnahme `java.net.ConnectException` ausgelöst. Die maximale Länge der Warteschlange ist standardmäßig 50.

Mit dem Konstruktor

```
ServerSocket(int port, int backlog) throws IOException
```

kann die maximale Länge der Warteschlange explizit festgelegt werden.

Der folgende Konstruktor bindet den Socket an die explizit vorgegeben IP-Adresse:

```
ServerSocket(int port, int backlog, InetAddress bindAddr)
       throws IOException

void bind(SocketAddress addr, int backlog) throws IOException
```

bindet den Server-Socket an eine Socket-Adresse (IP-Adresse und Portnummer).

Herstellen einer TCP-Verbindung

Abbildung 4-3: Herstellen einer TCP-Verbindung

1. Der Server erzeugt ein `ServerSocket`-Objekt, das an einen vorgegebenen Port gebunden wird.
2. Der Server ruft die Methode `accept` des Server-Sockets auf und wartet auf den Verbindungswunsch eines Clients.
3. Der Client erzeugt ein `Socket`-Objekt mit der Adresse und der Portnummer des Servers und versucht damit, eine Verbindung zum Server aufzunehmen.
4. Der Server erzeugt ein `Socket`-Objekt, das das serverseitige Ende der Kommunikationsverbindung darstellt. Die Methode `accept` gibt dieses Objekt zurück.

5. Um eine bidirektionale Verbindung zwischen Client und Server aufzubauen, stellen Client und Server jeweils einen Aus- und Eingabestrom mit Hilfe ihrer `Socket`-Objekte bereit.
6. Nun können Daten mittels Lese- und Schreiboperationen hin- und hergeschickt werden.

Abbildung 4-3 zeigt den Ablauf.

4.3 Iterative und parallele Server

In den beiden folgenden Projekten entwickeln wir verschiedene Varianten eines TCP-Servers, der Informationen zu Artikeln bereitstellt.

4.3.1 Iterativer Server

→ Projekt info_v1

Die *erste Variante* kann nicht mehrere Clients gleichzeitig bedienen. Werden z. B. zwei Clients kurz hintereinander gestartet, so werden sie der Reihe nach bedient. Erst wenn der erste Client beendet wurde, kommt der zweite zum Zuge. Es handelt sich also um einen so genannten *iterativen Server*.

```
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.HashMap;
import java.util.Map;

public class InfoServer1 {
   private int port;
   private Map<Integer, Artikel> map;

   public InfoServer1(int port) {
      this.port = port;
      load();
   }

   public void startServer() {
      try (ServerSocket serverSocket = new ServerSocket(port)) {
         System.out.println("InfoServer1 gestartet ...");
         process(serverSocket);
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private void process(ServerSocket serverSocket) throws IOException {
      while (true) {
         SocketAddress socketAddress = null;
```

```
        try (Socket socket = serverSocket.accept();
            ObjectOutputStream out = new ObjectOutputStream(
                  socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(
                  socket.getInputStream())) {

           socketAddress = socket.getRemoteSocketAddress();
           System.out.println("Verbindung zu " + socketAddress + " hergestellt");

           out.flush();

           while (true) {
              int id = in.readInt();
              Artikel artikel = map.get(id);
              out.writeObject(artikel);
              out.flush();
           }
        } catch (EOFException e) {
        } catch (IOException e) {
           System.err.println(e);
        } finally {
           System.out.println("Verbindung zu " + socketAddress + " beendet");
        }
     }
   }

   private void load() {
      map = new HashMap<>();
      map.put(4711, new Artikel(4711, "Hammer", 2.99));
      map.put(4712, new Artikel(4712, "Zange", 3.99));
      map.put(4713, new Artikel(4713, "Schraubendreher", 1.50));
   }

   public static void main(String[] args) {
      new InfoServer1(50000).startServer();
   }
}
```

In der Methode `startServer` wird der Server-Socket erzeugt und an die Portnummer 50000 gebunden.

Je Schleifendurchgang in der Methode `process` erfolgen diese Schritte:

- Aufruf der Methode `accept`, die so lange blockiert, bis ein Client versucht, Verbindung aufzunehmen,
- Bereitstellung der Ein- und Ausgabeströme `in` und `out`,
- In einer Schleife werden Artikelnummern gelesen und gefundene Artikel-Objekte in den Ausgabestrom geschrieben. Die Schleife läuft so lange, bis der Client die Verbindung und damit seinen Ausgabestrom geschlossen hat.

Zum Lesen und Schreiben wird der `ObjectInputStream` bzw. `ObjectOutputStream` verwendet. Die vom Client gesendete Artikelnummer wird mit `readInt` gelesen. In der Map wird der zur Artikelnummer passende Artikel (eine Instanz der Klasse `Artikel`) gelesen und serialisiert an den Client geschickt.

```
import java.io.Serializable;

public class Artikel implements Serializable {
   private int id;
   private String name;
   private double preis;

   public Artikel() {
   }

   public Artikel(int id, String name, double preis) {
      this.id = id;
      this.name = name;
      this.preis = preis;
   }

   // get-/set-Methoden

   @Override
   public String toString() {
      return "Artikel{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", preis=" + preis +
            '}';
   }
}
```

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Scanner;

public class InfoClient {
   public static void main(String[] args) {
      String host = "localhost";
      int port = 50000;

      try (Socket socket = new Socket(host, port);
          ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
          ObjectOutputStream out = new ObjectOutputStream(
                socket.getOutputStream())) {

         Scanner scanner = new Scanner(System.in);
         while (true) {
            System.out.print("> ");
            int id = 0;
            try {
               id = Integer.parseInt(scanner.next());
            } catch (NumberFormatException e) {
               continue;
            }
            if (id == 0)
               break;
            out.writeInt(id);
            out.flush();
            Artikel artikel = (Artikel) in.readObject();
```

```
            if (artikel != null)
               System.out.println(artikel);
            else
               System.out.println("Artikel " + id + " nicht vorhanden");
         }
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Zunächst wird ein Client-Socket erzeugt und damit versucht, die Verbindung zum Server herzustellen. Das `ObjectInputStream`-Objekt `in` ist der Eingabestrom, das `ObjectOutputStream`-Objekt `out` der Ausgabestrom für diesen Socket.

In einer Schleife werden nun von der Tastatur Artikelnummern eingelesen, in den Ausgabestrom `out` geschrieben und damit zum Server geschickt. `in.readObject()` blockiert so lange, bis die Antwort vom Server vorliegt. Diese wird dann angezeigt.

Der Server erzeugt zuerst den `ObjectOutputStream` und danach den `ObjectInputStream`. Der Client macht das genau umgekehrt. Dass hier die Reihenfolge bei der Erstellung der Streams wichtig ist, geht aus den folgenden Erläuterungen hervor:

Der Konstruktor

```
ObjectOutputStream(OutputStream out) throws IOException
```

erstellt einen `ObjectOutputStream`, der in den angegebenen `OutputStream` schreibt. Dieser Konstruktor schreibt den so genannten Serialisierungs-Stream-Header in den zugrunde liegenden Stream. Aufrufer sollten den Stream sofort leeren, um sicherzustellen, dass Konstruktoren zum Empfangen von `ObjectInputStreams` beim Lesen der Kopfzeile nicht blockieren.

Der Konstruktor

```
public ObjectInputStream(InputStream in) throws IOException
```

erstellt einen `ObjectInputStream`, der aus dem angegebenen `InputStream` liest. Ein Serialisierungs-Stream-Header wird aus dem Stream gelesen und verifiziert. Dieser Konstruktor blockiert, bis der entsprechende `ObjectOutputStream` den Header geschrieben und den Stream geleert hat.

Testablauf

Aufruf des Servers:

```
java -cp out/production/server InfoServer1
```

Ausgabe des Servers:

```
InfoServer1 gestartet ...
```

Aufruf des Clients (hier auf demselben Rechner):

```
java -cp out/production/client InfoClient
```

Der Server meldet den Aufbau der Verbindung:

```
Verbindung zu /127.0.0.1:50459 hergestellt
```

Eingabe des Clients:

```
> 4711
Artikel{id=4711, name='Hammer', preis=2.99}
> 4712
Artikel{id=4712, name='Zange', preis=3.99}
> 4799
Artikel 4799 nicht vorhanden
> 0
```

Der Server meldet den Abbau der Verbindung:

```
Verbindung zu /127.0.0.1:50459 beendet
```

Der Server kann über Tastatur mit *Strg + C* abgebrochen werden.

Wird während der Client-Sitzung das Client-Programm nochmals in einem neuen Terminal-Fenster gestartet, so wird die angeforderte Verbindung zum Server erst dann aufgebaut, wenn der erste Client die Sitzung beendet hat.

4.3.2 Paralleler Server

Die im letzten Absatz geschilderte Einschränkung ist in der nächsten Variante nicht vorhanden. Hier wir die Client-bezogene Verarbeitung in einem eigenen Thread ausgeführt. Sofort nach dem Start dieses Threads ist der `main`-Thread bereit weitere Verbindungen per `accept` entgegenzunehmen. Der Server ermöglicht also die parallele Bedienung mehrerer Clients (*paralleler Server*).

→ Projekt info_v1

```
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.HashMap;
import java.util.Map;

public class InfoServer2 {
   private int port;
   private Map<Integer, Artikel> map;

   public InfoServer2(int port) {
      this.port = port;
      load();
   }
```

```
   public void startServer() {
      try (ServerSocket serverSocket = new ServerSocket(port)) {
         System.out.println("InfoServer2 gestartet ...");
         while (true) {
            Socket socket = serverSocket.accept();
            new Thread(new Handler(socket)).start();
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private class Handler implements Runnable {
      private Socket socket;

      public Handler(Socket socket) {
         this.socket = socket;
      }

      @Override
      public void run() {
         SocketAddress socketAddress = null;

         try (ObjectOutputStream out = new ObjectOutputStream(
               socket.getOutputStream());
             ObjectInputStream in = new ObjectInputStream(
               socket.getInputStream())) {

            socketAddress = socket.getRemoteSocketAddress();
            System.out.println("Verbindung zu " + socketAddress + " hergestellt");

            out.flush();

            while (true) {
               int id = in.readInt();
               Artikel artikel = map.get(id);
               out.writeObject(artikel);
               out.flush();
            }
         } catch (EOFException e) {
         } catch (IOException e) {
            System.err.println(e);
         } finally {
            System.out.println("Verbindung zu " + socketAddress + " beendet");
         }
      }
   }

   private void load() {
      map = new HashMap<>();
      map.put(4711, new Artikel(4711, "Hammer", 2.99));
      map.put(4712, new Artikel(4712, "Zange", 3.99));
      map.put(4713, new Artikel(4713, "Schraubendreher", 1.50));
   }

   public static void main(String[] args) {
      new InfoServer2(50000).startServer();
   }
}
```

4.3.3 Thread-Pooling

Die Fähigkeit des Servers, mehrere Clients quasi gleichzeitig zu bedienen, wurde bisher so gelöst, dass für jede Anfrage eines Clients ein neuer Thread erzeugt wurde. Im Vergleich zu Prozessen ist die Erzeugung eines Threads weniger aufwändig. Trotzdem sollte man hiermit sparsam umgehen; insbesondere dann, wenn kurzzeitig sehr viele Threads mit kurzer Laufzeit benötigt werden.

Ein *Thread-Pool* bietet die Möglichkeit, mehrere separate Aufgaben vom selben Thread nacheinander ausführen zu lassen. Nicht mehr benutzte Threads werden in den Pool zurückgelegt und können wiederverwendet werden.

Wir benutzen hier eine spezielle Thread-Pool-Variante, einen so genannten *Cached Thread Pool*. Ein solcher Pool wächst bzw. schrumpft nach Bedarf. Steht eine neue Aufgabe (hier eine Client-Anfrage) zur Bearbeitung an und gibt es keinen "freien" Thread im Pool, so wird ein neuer erzeugt, der die Ausführung übernimmt. Ist die Aufgabe ausgeführt, steht dieser Thread als wieder "freier" Thread im Pool zur Verfügung. Wird er *innerhalb von 60 Sekunden* nicht benötigt, so wird er terminiert und ist dann nicht mehr verwendbar. Der Pool passt sich also dynamisch den momentanen Anforderungen an und ist optimal für kleinere Aufgaben, die in hoher Zahl kurzfristig anstehen.

Für unsere Zwecke benutzen wir die Klasse `java.util.concurrent.Executors` und das Interface `java.util.concurrent.ExecutorService`.

Die statische `Executors`-Methode

```
static ExecutorService newCachedThreadPool()
```

erzeugt einen *Cached Thread Pool* und liefert diesen als Objekt vom Typ `ExecutorService` zurück.

Das Interface `ExecutorService` enthält u. a. die folgenden Methoden:

`void execute(Runnable task)`
: führt die `run`-Methode von `task` in einem Thread des Pools aus.

`void shutdown()`
: bewirkt, dass vor dem Aufruf dieser Methode übergebene Aufgaben noch ausgeführt, neue aber nicht mehr akzeptiert werden; alle vom Pool verwalteten Threads werden dann terminiert.

→ Projekt info_v2

Der folgende Server nutzt zur Parallelisierung Thread-Pooling. Zudem werden `Artikel`-Objekte für die Übertragung in JSON-Dokumente konvertiert. Die beiden Unterprojekte *client* und *server* haben also die Dependency *json* (siehe auch Kapitel 2). Die Klasse Artikel ist dieselbe wie im Projekt *info_v1*, allerdings muss die Klasse nicht das Interface `Serializable` implementieren.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class InfoServer {
   private int port;
   private Map<Integer, Artikel> map;
   private ExecutorService pool;

   public InfoServer(int port) {
      this.port = port;
      load();
   }

   public void startServer() {
      try (ServerSocket serverSocket = new ServerSocket(port)) {
         System.out.println("InfoServer gestartet ...");
         pool = Executors.newCachedThreadPool();
         while (true) {
            Socket socket = serverSocket.accept();
            pool.execute(new Handler(socket));
         }
      } catch (IOException e) {
         System.err.println(e);
         pool.shutdown();
      }
   }

   private class Handler implements Runnable {
      private Socket socket;

      public Handler(Socket socket) {
         this.socket = socket;
      }

      @Override
      public void run() {
         SocketAddress socketAddress = null;

         try (BufferedReader in = new BufferedReader(new InputStreamReader(
                  socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

            socketAddress = socket.getRemoteSocketAddress();
            System.out.println("Verbindung zu " + socketAddress + " hergestellt");

            String line;
            while ((line = in.readLine()) != null) {
               int id = Integer.parseInt(line);
```

```
                Artikel artikel = map.get(id);
                artikel = (artikel != null) ? artikel : new Artikel();
                out.println(serialize(artikel));
            }
        } catch (IOException e) {
            System.err.println(e);
        } finally {
            System.out.println("Verbindung zu " + socketAddress + " beendet");
        }
    }
}

private void load() {
    map = new HashMap<>();
    map.put(4711, new Artikel(4711, "Hammer", 2.99));
    map.put(4712, new Artikel(4712, "Zange", 3.99));
    map.put(4713, new Artikel(4713, "Schraubendreher", 1.50));
}

private static String serialize(Artikel artikel) {
    Jsonb jsonb = JsonbBuilder.create();
    return jsonb.toJson(artikel);
}

public static void main(String[] args) {
    new InfoServer(50000).startServer();
}
}


import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class InfoClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 50000;

        try (Socket socket = new Socket(host, port);
            BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

            Scanner scanner = new Scanner(System.in);
            while (true) {
                System.out.print("> ");
                int id = 0;
                try {
                    id = Integer.parseInt(scanner.next());
                } catch (NumberFormatException e) {
                    continue;
                }
                if (id == 0)
                    break;
```

```
            out.println(id);

            Artikel artikel = deserialize(in.readLine());
            if (artikel.getId() == 0)
               System.out.println("Artikel " + id + " nicht vorhanden");
            else
               System.out.println(artikel);
         }
      } catch (Exception e) {
         System.err.println(e);
      }
   }

   private static Artikel deserialize(String data) {
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.fromJson(data, Artikel.class);
   }
}
```

4.4 Ein Framework für TCP-Server

Die Beispiele der vorhergehenden Abschnitte zeigen, dass die Implementierungen der TCP-Server im Großen und Ganzen fast immer dem gleichen Muster folgen. Es liegt also nahe, die immer wiederkehrenden Codeteile zu standardisieren und als Framework für eigene Server-Implementierungen anzubieten.

→ Projekt framework

Das Framework

Das Framework besteht aus den beiden Klassen `TCPServer` und `AbstractHandler`, die zum Paket `tcpframework` gehören.

Der Konstruktor von `TCPServer` erwartet eine Portnummer und die Klasse des Handlers, der die eigentliche Kommunikation mit dem Client durchführt. Hier wird das `Class`-Objekt des Handlers angegeben. Anstelle der Handler-Klasse kann auch eine Handler-Instanz angegeben werden. Der Handler ist eine Subklasse von `AbstractHandler`.

Innerhalb des Konstruktors werden u. a. ein `ServerSocket`-Objekt und ein Thread-Pool (siehe Kapitel 4.3.3) erzeugt.

`TCPServer` ist von `Thread` abgeleitet. Innerhalb der `run`-Methode wird in einer Schleife die Methode `accept` aufgerufen. Für die im Konstruktor erzeugte bzw. übergebene Handler-Instanz wird die Methode `handle` mit den Referenzen für das `Socket`-Objekt und den Thread-Pool aufgerufen.

Die Methode `stopServer` schließt das `ServerSocket`-Objekt und terminiert den Thread-Pool.

```
package tcpframework;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TCPServer extends Thread {
   private AbstractHandler handler;
   private ServerSocket serverSocket;
   private ExecutorService pool;

   public TCPServer(int port, Class<?> handlerClass) throws Exception {
      handler = (AbstractHandler) handlerClass.getDeclaredConstructor()
                     .newInstance();
      serverSocket = new ServerSocket(port);
      pool = Executors.newCachedThreadPool();
   }

   public TCPServer(int port, Object handlerObject) throws IOException {
      handler = (AbstractHandler) handlerObject;
      serverSocket = new ServerSocket(port);
      pool = Executors.newCachedThreadPool();
   }

   public void run() {
      try {
         while (true) {
            Socket socket = serverSocket.accept();
            handler.handle(socket, pool);
         }
      } catch (SocketException e) {
         // Beim Aufruf von stopServer() wird eine SocketException ausgelöst
      } catch (Exception e) {
         System.err.println(e);
      }
   }

   public void stopServer() {
      try {
         serverSocket.close();
      } catch (IOException e) {
      }
      pool.shutdown();
   }
}
```

Ein konkreter Handler ist von `AbstractHandler` abgeleitet und implementiert die Methode `runTask`.

```
package tcpframework;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketAddress;
```

```
import java.util.concurrent.ExecutorService;

public abstract class AbstractHandler {
   public void handle(final Socket socket, ExecutorService pool) {
      pool.execute(new Runnable() {

         @Override
         public void run() {
            SocketAddress socketAddress = socket.getRemoteSocketAddress();
            System.out.println("Verbindung zu " + socketAddress + " hergestellt");

            runTask(socket);

            try {
               if (socket != null)
                  socket.close();
            } catch (IOException e) {
            }

            System.out.println("Verbindung zu " + socketAddress + " beendet");
         }
      });
   }

   public abstract void runTask(Socket socket);
}
```

Anwendung des Frameworks

Das Framework wird nun in einem Beispiel verwendet. Hierbei handelt es sich um einen Server, der einen empfangenen Text als Echo wieder zurücksendet.

```
import tcpframework.TCPServer;

public class EchoServer {
   public static void main(String[] args) throws Exception {
      int port = 50000;
      TCPServer server = new TCPServer(port, EchoHandler.class);
      server.start();
      System.out.println("EchoServer gestartet.");
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      server.stopServer();
      System.out.println("EchoServer gestoppt.");
   }
}
```

Der Server wird nach Betätigung der ENTER-Taste gestoppt; allerdings erst dann, wenn alle momentan aktiven Client-Bearbeitungen beendet sind.

```
import tcpframework.AbstractHandler;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
```

```
public class EchoHandler extends AbstractHandler {

   @Override
   public void runTask(Socket socket) {
      try (BufferedReader in = new BufferedReader(new InputStreamReader(
               socket.getInputStream()));
          PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

         out.println("Server ist bereit ...");

         String input;
         while ((input = in.readLine()) != null) {
            out.println(input);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

Hierzu ein passender Client:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class EchoClient {
   public static void main(String[] args) {
      String host = "localhost";
      int port = 50000;

      try (Socket socket = new Socket(host, port);
          BufferedReader in = new BufferedReader(new InputStreamReader(
               socket.getInputStream()));
          PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
          BufferedReader input = new BufferedReader(new InputStreamReader(
               System.in))) {

         String msg = in.readLine();
         System.out.println(msg);

         String line;
         while (true) {
            System.out.print("> ");
            line = input.readLine();
            if (line.equals("q"))
               break;
            out.println(line);
            System.out.println(in.readLine());
         }
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Das Framework kann als jar-Datei wie folgt bereitgestellt werden:

```
jar --create --file tcpframework.jar -C out/production/server tcpframework
```

4.5 Beenden eines Datenstroms

Die `Socket`-Methode `close` schließt nicht nur den Socket, sondern auch gleichzeitig die mit ihm verbundenen Datenströme. Umgekehrt führt das Schließen eines der beiden Datenströme zum Schließen des Sockets.

Will man nur einen der Datenströme beenden, ohne auch den Socket zu schließen, kann die `Socket`-Methode `shutdownInput` bzw. `shutdownOutput` verwendet werden (siehe Kapitel 4.2).

→ Projekt upload

Wir nutzen diese Möglichkeit im folgenden Programm. Der `UploadClient` sendet den Inhalt einer Datei zum Server und beendet dann den Ausgabestrom. Der `UploadServer` liest Bytes aus dem Eingabestrom, bis dieser beendet ist. Danach schickt er eine Meldung zum Client.

Würde der Ausgabestrom vom Client nicht durch `shutdownOutput` beendet, so würde der Server in Erwartung weiterer Daten in der Schleife zum Einlesen verbleiben und so das Programm nicht zum Ende kommen.

Der Client sendet den Dateinamen, danach ein Newline-Zeichen und dann den Inhalt der Datei. Die Datenstruktur ist also wie folgt:

Dateiname\n*Dateiinhalt*

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.Socket;

public class UploadClient {
   public static void main(String[] args) {
      String host = "localhost";
      int port = 50000;
      String filename = "test.pdf";

      try (Socket socket = new Socket(host, port);
          BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
          OutputStream out = socket.getOutputStream()) {

         out.write(filename.getBytes());
         out.write('\n');

         try (InputStream is = new FileInputStream(filename)) {
            byte[] buffer = new byte[1024];
```

```
            int c;
            while ((c = is.read(buffer)) != -1) {
               out.write(buffer, 0, c);
            }
            System.out.println(filename + " wurde hochgeladen.");
         }

         socket.shutdownOutput();

         String msg = in.readLine();
         System.out.println(msg);
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Für den Server nutzen wir das Framework aus Kapitel 4.4.

```
import tcpframework.TCPServer;

public class UploadServer {
   public static void main(String[] args) throws Exception {
      int port = 50000;
      String dir = "D:/temp";

      TCPServer server = new TCPServer(port, new UploadHandler(dir));

      server.start();
      System.out.println("UploadServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("UploadServer wird gestoppt.");
   }
}
```

```
import tcpframework.AbstractHandler;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;

public class UploadHandler extends AbstractHandler {
   private String dir;

   public UploadHandler(String dir) {
      this.dir = dir;
   }
```

```
   @Override
   public void runTask(Socket socket) {
      try (InputStream in = socket.getInputStream();
           PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

         StringBuilder builder = new StringBuilder();
         int c;
         while ((c = in.read()) != '\n') {
            builder.append((char) c);
         }
         String filename = builder.toString();
         File file = new File(dir, filename);

         String message;
         try (OutputStream os = new FileOutputStream(file)) {
            byte[] buffer = new byte[8192];
            while ((c = in.read(buffer)) != -1) {
               os.write(buffer, 0, c);
            }
            message = "Die Datei '" + filename + "' wurde in '" + dir
                                    + "' gespeichert.";
         } catch (IOException e) {
            System.err.println(e);
            message = e.getMessage();
         }

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

Alle Zeichen bis zum ersten Newline-Zeichen bilden den Dateinamen. Alle Bytes, die danach folgen, werden in die Datei mit dem gebildeten Namen im Verzeichnis *D:\temp* gespeichert. Im Erfolgsfall wird eine entsprechende Nachricht als Antwort gesendet.

4.6 Ein Chat-Programm

→ Projekt chat

Mit dem Programm `ChatClient` ist das so genannte "Chatten" mit mehreren Teilnehmern im Netz möglich. Der Teilnehmer kann sich an- und abmelden und Texte durch Eingabe von ENTER im Eingabefeld an alle anderen aktiven Teilnehmer senden. Der Server registriert die angemeldeten Teilnehmer und verteilt eingehende Nachrichten an alle registrierten Teilnehmer.

Zur Compile-und Laufzeit benötigt der Server das Framework aus Kapitel 4.4 und die Bibliothek *libs/json* (siehe Kapitel 2). Der Client benötigt zur Compile-und Laufzeit nur die Bibliothek *libs/json*.

Daten werden im JSON-Format zwischen Client und Server ausgetauscht. Diese werden programmintern als Objekte vom Typ `Message` verwaltet.

```
public class Message {
   public enum Action {
      LOGIN, JOIN, LEAVE, SAY, JOIN_ERROR
   }

   public Action action;
   public String user;
   public String text;
}
```

Die Klasse `Message` enthält die Attribute `user` und `text` sowie einen Operationscode `action` (hier vom Aufzählungstyp `Action`), der die Art der Nachricht bestimmt:

- `LOGIN`
  wird bei der Anmeldung eines Benutzers verwendet.
- `JOIN`
  wird bei erfolgreicher Anmeldung verwendet.
- `LEAVE`
  wird bei der Abmeldung eines Benutzers verwendet.
- `SAY`
  wird für den Austausch einer Nachricht eines angemeldeten Benutzers verwendet.
- `JOIN_ERROR`
  wird verwendet, wenn die Anmeldung nicht erfolgreich war.

```
import tcpframework.TCPServer;

public class ChatServer {
   public static void main(String[] args) throws Exception {
      int port = 50000;

      TCPServer server = new TCPServer(port, ChatHandler.class);

      server.start();
      System.out.println("ChatServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("ChatServer wird gestoppt.");
   }
}
```

Der Server verwaltet zwei Listen:

- die Namen der angemeldeten User (Liste `users`) und
- die aktiven Verbindungen (Liste `connections`).

Beide Listen sind vom Typ `CopyOnWriteArrayList`. Diese Klasse verhindert Ausnahmen vom Typ `ConcurrentModificationException` während der Iteration über die Elemente der Liste.

Nach Aufnahme einer Verbindung wird die Referenz auf den Ausgabestrom vom Typ `PrintWriter` für den angemeldeten Teilnehmer in der Liste `connections` gespeichert und am Ende wieder entfernt.

```
import tcpframework.AbstractHandler;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class ChatHandler extends AbstractHandler {
   private final static int TIMEOUT = 600000; // 10 Min.
   private List<PrintWriter> connections;
   private List<String> users;
   private Jsonb jsonb;

   public ChatHandler() {
      connections = new CopyOnWriteArrayList<>();
      users = new CopyOnWriteArrayList<>();
      jsonb = JsonbBuilder.create();
   }

   @Override
   public void runTask(Socket socket) {
      String user = null;
      BufferedReader in = null;
      PrintWriter out = null;

      boolean joined = false;
      try {
         socket.setSoTimeout(TIMEOUT);
         in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
         out = new PrintWriter(socket.getOutputStream(), true);

         String line;
         while ((line = in.readLine()) != null) {
            Message message = jsonb.fromJson(line, Message.class);
            switch (message.action) {
               case LOGIN:
                  user = message.user;
                  joined = join(user, out);
                  break;
               case SAY:
                  say(user, message.text, out);
                  break;
            }
         }
      } catch (IOException e) {
         System.err.println(e);
      } finally {
```

```
            try {
               socket.close();
            } catch (IOException e) {
            }

            if (joined)
               leave(user, out);
         }
      }

      private boolean join(String user, PrintWriter out) {
         if (users.contains(user)) {
            Message message = new Message();
            message.action = Message.Action.JOIN_ERROR;
            message.user = user;
            String jsonStr = jsonb.toJson(message);
            out.println(jsonStr);
            out.close();
            return false;
         } else {
            connections.add(out);
            users.add(user);
            Message message = new Message();
            message.action = Message.Action.JOIN;
            message.user = user;
            broadcast(message);
            System.out.println("Anzahl Verbindungen: " + connections.size());
            System.out.println("Angemeldete User: " + users);
            return true;
         }
      }

      private void leave(String user, PrintWriter out) {
         connections.remove(out);
         users.remove(user);
         Message message = new Message();
         message.action = Message.Action.LEAVE;
         message.user = user;
         broadcast(message);
         System.out.println("Anzahl Verbindungen: " + connections.size());
         System.out.println("Angemeldete User: " + users);
      }

      private void say(String user, String text, PrintWriter out) {
         Message message = new Message();
         message.action = Message.Action.SAY;
         message.user = user;
         message.text = text;
         broadcast(message);
      }

      private void broadcast(Message message) {
         String jsonStr = jsonb.toJson(message);
         for (PrintWriter out : connections) {
            out.println(jsonStr);
         }
      }
   }
```

Kollidiert beim Versuch der Anmeldung der Name mit dem Namen eines bereits angemeldeten Benutzers, so wird eine Fehlermeldung (`JOIN_ERROR`) gesendet und die Verbindung beendet.

Die Methode `broadcast` sendet Nachrichten an alle angemeldeten Teilnehmer.

Nach der An- bzw. Abmeldung eines Teilnehmers werden auf der Server-Konsole die Anzahl aktiver Verbindungen und die Namen der momentan angemeldeten User angezeigt.

Hier folgt nun der Client, dessen Oberfläche mit JavaFX realisiert ist:[2]

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class ChatClient extends Application implements Runnable {
   private static String HOST = "localhost";
   private static int PORT = 50000;

   private TextArea ta;
   private TextField tf;
   private Button btn;
   private boolean loggedIn;
   private String user;
   private Socket socket;
   private BufferedReader in;
   private BufferedWriter out;
   private Stage stage;
   private volatile Thread t;
   private Jsonb jsonb;

   @Override
   public void start(Stage stage) {
      this.stage = stage;

      VBox box = new VBox();
      ta = new TextArea();
      tf = new TextField();
      btn = new Button("Anmelden");
```

2 Die jar-Dateien von JavaFX müssen zur Compilierung dem Projekt zugeordnet werden.

```
        box.getChildren().addAll(ta, tf, btn);

        ta.setWrapText(true);
        ta.setStyle("-fx-font: 16pt \"Arial\";");
        ta.setEditable(false);
        ta.setPrefHeight(500);

        tf.setStyle("-fx-font: 16pt \"Arial\";");
        tf.setOnAction(event -> {
            if (loggedIn)
                say();
        });

        btn.setStyle("-fx-font: 16pt \"Arial\";");
        btn.setOnAction(event -> btnAction());

        stage.setScene(new Scene(box, 800, 500));
        stage.setTitle("Chat");
        stage.show();

        jsonb = JsonbBuilder.create();
    }

    @Override
    public void stop() throws Exception {
        logout();
    }

    private void btnAction() {
        if (loggedIn) {
            logout();
        } else {
            user = tf.getText();
            if (user.length() != 0) {
                login();
            }
        }
    }

    private void login() {
        try {
            socket = new Socket(HOST, PORT);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new BufferedWriter(new OutputStreamWriter(
                      socket.getOutputStream()));

            send(Message.Action.LOGIN, user, "");

            t = new Thread(this);
            t.start();
        } catch (IOException e) {
            ta.appendText(e.getMessage() + "\n");
            logout();
        } finally {
            tf.setText("");
            tf.requestFocus();
        }
    }
```

```
private void logout() {
   try {
      loggedIn = false;
      stage.setTitle("Chat");
      btn.setText("Anmelden");
      t = null;
      if (socket != null)
         socket.close();
      if (in != null)
         in.close();
      if (out != null)
         out.close();
   } catch (IOException e) {
   }
}

private void say() {
   try {
      send(Message.Action.SAY, user, tf.getText());
   } catch (IOException e) {
      ta.appendText(e.getMessage() + "\n");
      logout();
   } finally {
      tf.setText("");
      tf.requestFocus();
   }
}

private void send(Message.Action action, String user, String text)
                throws IOException {
   Message message = new Message();
   message.action = action;
   message.user = user;
   message.text = text;
   String jsonStr = jsonb.toJson(message);

   out.write(jsonStr);
   out.newLine();
   out.flush();
}

@Override
public void run() {
   try {
      while (Thread.currentThread() == t) {
         String line = in.readLine();
         if (line == null)
            break;

         Message message = jsonb.fromJson(line, Message.class);
         String msg = "";
         switch (message.action) {
            case JOIN:
               Platform.runLater(() -> {
                    loggedIn = true;
                    stage.setTitle("Chat - " + user);
                    btn.setText("Abmelden");
               });
               msg = message.user + " ist angemeldet.";
```

```
                    break;
                case JOIN_ERROR:
                    msg = message.user + " ist bereits angemeldet.";
                    break;
                case LEAVE:
                    msg = message.user + " ist abgemeldet.";
                    break;
                case SAY:
                    msg = message.user + ": " + message.text;
                    break;
            }

            String output = msg;
            Platform.runLater(() -> {
                ta.appendText(output);
                ta.appendText("\n");
            });
        }
    } catch (IOException e) {
    }
}

public static void main(String[] args) {
    launch(args);
}
}
```

In der Methode `login` wird ein Thread gestartet, der die eingehenden Nachrichten verarbeitet.

Abbildung 4-4: Der Chat-Client

```
java -cp out/production/server;../../libs/tcpframework.jar;
../../libs/json/* ChatServer
```

Nachdem der Server gestartet wurde, kann das Client-Programm mehrfach aufgerufen werden (siehe Abbildung 4-4):

```
java -cp out/production/client;../../libs/json/* -p %PATH_TO_FX%
--add-modules=javafx.controls ChatClient
```

Die Umgebungsvariable `PATH_TO_FX` zeigt auf das *lib*-Verzeichnis der JavaFX-Installation.

Ausgabe des Servers:

```
ChatServer gestartet.
ENTER stoppt den Server.
Verbindung zu /127.0.0.1:53138 hergestellt
Anzahl Verbindungen: 1
Angemeldete User: [Hugo]
Verbindung zu /127.0.0.1:53139 hergestellt
Anzahl Verbindungen: 2
Angemeldete User: [Hugo, Emil]
```

4.7 Asynchrone Kommunikation

In diesem Kapitel stellen wir ein Beispiel vor, in dem Client und Server *asynchron* miteinander kommunizieren. Senden und Empfangen von Daten sind zeitlich versetzt. Der Client ist nicht blockiert durch Warten auf die Antwort des Servers.

Der Client erteilt einen Auftrag an den Server. Nach der Empfangsbestätigung durch den Server wir die Verbindung beendet. Der Server bearbeitet nun den Auftrag und baut nach Beendigung der Bearbeitung seinerseits eine Verbindung zum Client auf, um ihm die Fertigstellung mitzuteilen.

Dazu muss der Client selbst einen Server bereitstellen. Die Portnummer dieses im Client integrierten Servers wird dem Server, der den Auftrag bearbeiten soll, zu Beginn mitgeteilt.

Abbildung 4-5: Asynchrone Kommunikation

→ Projekt async

Die Daten zum Auftrag werden in einem Objekt vom Typ `Job` gespeichert. Dieses wird für die Übertragung in eine JSON-Dokument konvertiert.

Der Auftrag enthält die Job-Id (`id`), die eigentliche Anfrage (`request`), das zugehörige Ergebnis (`response`), Adresse und Portnummer für die Kontaktaufnahme (`addr`, `port`) und die Dauer der serverseitigen Bearbeitung (`duration`).

```
public class Job {
   private String id;
   private String request;
   private String response;
   private String addr;
   private int port;
   private long duration;

   // get-/set-Methoden

   @Override
   public String toString() {
      return "Job{" +
            "id='" + id + '\'' +
            ", request='" + request + '\'' +
            ", response='" + response + '\'' +
            ", addr='" + addr + '\'' +
            ", port=" + port +
            ", duration=" + duration +
            '}';
   }
}
```

```
import tcpframework.TCPServer;

public class JobServer {
   public static void main(String[] args) throws Exception {
      int port = 50000;

      TCPServer server = new TCPServer(port, JobHandler.class);

      server.start();
      System.out.println("JobServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("JobServer wird gestoppt.");
   }
}
```

```
import tcpframework.AbstractHandler;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class JobHandler extends AbstractHandler {
   @Override
   public void runTask(Socket socket) {
      try (BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
          PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

         String data = in.readLine();
         if (data != null) {
            Job job = deserialize(data);
            out.println(job.getId() + " erhalten.");
            new Worker(job).start();
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private static Job deserialize(String data) {
      Jsonb jsonb = JsonbBuilder.create();
      return jsonb.fromJson(data, Job.class);
   }
}
```

Die eigentliche Bearbeitung des Auftrags wird in einem separaten Thread (`Worker`) ausgeführt. Dieser nimmt nach der Bearbeitung die Verbindung zum Client auf.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.time.Duration;
import java.time.Instant;
import java.util.Random;

public class Worker extends Thread {
   private Job job;

   public Worker(Job job) {
      this.job = job;
   }

   @Override
   public void run() {
      System.out.println(job.getId() + " wird bearbeitet.");
      Instant start = Instant.now();

      try {
         Random random = new Random();
         Thread.sleep(8000 + random.nextInt(8000));
```

```
        } catch (InterruptedException e) {
        }

        Instant end = Instant.now();
        long millis = Duration.between(start, end).toMillis();
        job.setResponse("Hier steht das Ergebnis.");
        job.setDuration(millis);

        System.out.println(job.getId() + " ist fertig.");

        try (Socket socket = new Socket(job.getAddr(), job.getPort());
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

           out.println(serialize(job));
        } catch (IOException e) {
           System.err.println(e);
        }
    }

    private static String serialize(Job job) {
        Jsonb jsonb = JsonbBuilder.create();
        return jsonb.toJson(job);
    }
}
```

Der Client startet zu Beginn in der Methode `startReceiver` einen Server, der an die Portnummer 40000 gebunden ist. Mit diesem integrierten Server nimmt der `Worker` (siehe oben) Kontakt auf, um das Bearbeitungsergebnis zurückzusenden.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class JobClient {
    private final static String HOST = "localhost";
    private final static int PORT = 50000;
    private final static int SERVER_PORT = 40000;
    private ServerSocket serverSocket;

    public static void main(String[] args) throws Exception {
        JobClient client = new JobClient();
        client.startReceiver();

        Job job1 = new Job();
        job1.setId("Job 1");
        job1.setRequest("Hier steht die Auftragsbeschreibung.");
        job1.setAddr(InetAddress.getLocalHost().getHostAddress());
        job1.setPort(SERVER_PORT);
```

```
    Job job2 = new Job();
    job2.setId("Job 2");
    job2.setRequest("Hier steht die Auftragsbeschreibung.");
    job2.setAddr(InetAddress.getLocalHost().getHostAddress());
    job2.setPort(SERVER_PORT);

    client.process(job1);
    Thread.sleep(3000);
    client.process(job2);

    System.in.read();
    client.serverSocket.close();
}

private void process(Job job) {
    try (Socket socket = new Socket(HOST, PORT);
        BufferedReader in = new BufferedReader(
              new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

       out.println(serialize(job));
       System.out.println("Server: " + in.readLine());
    } catch (Exception e) {
       System.err.println(e);
    }
}

private void startReceiver() {
    Thread t = new Thread() {
       @Override
       public void run() {
          try {
             serverSocket = new ServerSocket(SERVER_PORT);

             while (true) {
                try (Socket socket = serverSocket.accept();
                      BufferedReader in = new BufferedReader(
                              new InputStreamReader(socket.getInputStream()))) {

                     String data = in.readLine();
                     if (data != null) {
                        Job job = deserialize(data);
                        System.out.println(job);
                     }
                } catch (SocketException e) {
                     break;
                } catch (IOException e) {
                     System.err.println(e);
                }
             }
          } catch (IOException e) {
             System.err.println(e);
          }
       }
    };

    t.start();
}
```

```
    private static String serialize(Job job) {
        Jsonb jsonb = JsonbBuilder.create();
        return jsonb.toJson(job);
    }

    private static Job deserialize(String data) {
        Jsonb jsonb = JsonbBuilder.create();
        return jsonb.fromJson(data, Job.class);
    }
}
```

In diesem Szenario werden zwei Aufträge (Job 1 und Job 2) kurz hintereinander gesendet.

Zur Compile- und Laufzeit benötigt der Server das Framework aus Kapitel 4.4 und die Bibliothek *json* (siehe Kapitel 2). Der Client benötigt zur Compile- und Laufzeit nur die Bibliothek *json*.

Aufruf des Servers:

```
java -cp out/production/server;../../libs/tcpframework.jar;
../../libs/json/* JobServer
```

Aufruf des Clients:

```
java -cp out/production/client;../../libs/json/* JobClient
```

Hier die Ausgabe des Servers:

```
Verbindung zu /127.0.0.1:53360 hergestellt
Verbindung zu /127.0.0.1:53360 beendet
Job 1 wird bearbeitet.
Verbindung zu /127.0.0.1:53361 hergestellt
Verbindung zu /127.0.0.1:53361 beendet
Job 2 wird bearbeitet.
Job 1 ist fertig.
Job 2 ist fertig.
```

Und hier die Ausgabe des Clients:

```
Job 1 erhalten.
Job 2 erhalten.
Job{id='Job 1', request='Hier steht die Auftragsbeschreibung.',
response='Hier steht das Ergebnis.', addr='192.168.2.91', port=40000,
duration=10380}
Job{id='Job 2', request='Hier steht die Auftragsbeschreibung.',
response='Hier steht das Ergebnis.', addr='192.168.2.91', port=40000,
duration=14011}
```

4.8 Aufgaben

1. Erweitern Sie den Client und Server aus dem Projekt *info_v2* um die Möglichkeit, Artikelpreise zu ändern. Zum Datenaustausch soll die folgende Klasse dienen, die neben der Referenz auf den Artikel die Operation (`READ` oder `UPDATE`) enthält sowie einen Text, der über die erfolgreiche oder nicht erfolgreiche Änderung informiert:

```
public class Message {
   public enum Op {
      READ, UPDATE
   }

   public Op op;
   public String info;
   public Artikel artikel;
}
```

2. Entwickeln Sie ein Programm, mit dem Dateien von einem Rechner auf einen anderen kopiert werden können.

   Aufruf beim Empfänger (Server):

```
java -cp out/production/Aufgabe-4.2 NetCopy > aus.pdf
```

   Aufruf beim Sender (Client):

```
java -cp out/production/Aufgabe-4.2 NetCopy <IP-Nummer> < test.pdf
```

3. Entwickeln Sie einen Server `SecretMessageReceiver`, der verschlüsselte Nachrichten empfangen kann und diese entschlüsselt auf der Konsole ausgibt, sowie einen Client `SecretMessageSender`, der Nachrichten verschlüsselt an den Server sendet.

   Aufruf des Servers:

```
java -cp out/production/Aufgabe-4.3 SecretMessageReceiver
```

   Aufruf des Clients (Beispiel):

```
java -cp out/production/Aufgabe-4.3 SecretMessageSender < klar.txt
```

   Benutzen Sie den symmetrischen Verschlüsselungsalgorithmus *Blowfish*.[3]

   Das Programm `CryptDemo` zeigt, wie Datenströme mit Blowfish verschlüsselt bzw. entschlüsselt werden können:

```
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.spec.SecretKeySpec;
import java.io.InputStream;
import java.io.OutputStream;
import java.security.Key;

public class CryptDemo {
   private static void encode(byte[] password) throws Exception {
      Cipher cipher = Cipher.getInstance("Blowfish");
      Key key = new SecretKeySpec(password, "Blowfish");
```

3 https://de.wikipedia.org/wiki/Blowfish

```
        cipher.init(Cipher.ENCRYPT_MODE, key);

        try (OutputStream out = new CipherOutputStream(System.out, cipher)) {
            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = System.in.read(buffer)) != -1) {
                out.write(buffer, 0, c);
            }
        }
    }

    private static void decode(byte[] password) throws Exception {
        Cipher cipher = Cipher.getInstance("Blowfish");
        Key key = new SecretKeySpec(password, "Blowfish");
        cipher.init(Cipher.DECRYPT_MODE, key);

        try (InputStream in = new CipherInputStream(System.in, cipher)) {
            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
                System.out.write(buffer, 0, c);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        char op = args[0].charAt(0);
        byte[] password = args[1].getBytes();

        if (op == 'e')
            encode(password);
        else
            decode(password);
    }
}
```

5 Nachrichtentransfer mit JMS

Die Kommunikation zwischen Client und Server in den vorangegangenen Kapiteln ist dadurch gekennzeichnet, dass die Teilnehmer direkt und zeitgleich miteinander in Verbindung treten. In der Regel ist der Client solange blockiert, bis der Server die Verarbeitung abgeschlossen und die Antwort zurückgesendet hat (*synchrone Kommunikation*). Die Kommunikation auf der Basis von nachrichtenorientierter Middleware (*Message Oriented Middleware*, *MOM*) macht sich von dieser engen Kopplung frei. Der Austausch von Nachrichten erfolgt *asynchron* mit Hilfe eines Vermittlers (*Message Broker*, *MOM-Server*), der Warteschlangen verwaltet. Die Nachricht des Senders wird vom Vermittler in die Warteschlange des Empfängers gelegt. Der Empfänger kann diese Nachricht zu einem späteren Zeitpunkt aus der Warteschlange holen. Sender und Empfänger agieren unabhängig voneinander und sind also über den Vermittler nur lose gekoppelt.

Abbildung 5-1: Nachrichtenaustausch

Vorteile dieser *asynchronen Kommunikation* sind:

- Sender und Empfänger arbeiten unabhängig voneinander. Auch bei Ausfall eines Teils (Sender, Empfänger oder Vermittler) kann die Nachricht noch nachträglich zugestellt werden. Die beteiligten Komponenten können getrennt voneinander wieder gestartet werden. Daraus ergibt sich eine hohe Fehlertoleranz.
- Eine Nachricht kann dank des Vermittlers an mehrere Empfänger gesendet werden, ohne dass jeweils eine explizite Verbindung vom Client zu jedem Empfänger aufgebaut werden muss.
- Mehrere gesendete Nachrichten können gebündelt werden und zur Steigerung der Effizienz erst zu einem späteren Zeitpunkt in einem Rutsch abgeholt und verarbeitet werden.
- Durch Verwendung nachrichtenorientierter Middleware ist die Kommunikation unabhängig von Programmiersprache und Plattform.
- Die weitgehende Unabhängigkeit von Sender und Empfänger fördert den Einsatz dieser Technologie im Bereich der Anwendungsintegration.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_5

5.1 Java Message Service

Java Message Service (*JMS*) wurde 1998 von Sun Microsystems veröffentlicht und ist Teil der Java Enterprise Edition. JMS ist ein API, das Syntax und Semantik für den Zugriff von in Java geschriebenen Clients auf nachrichtenorientierte Middleware definiert. Verschiedene Hersteller bieten Implementierungen an, die diese Spezifikation umsetzen. Hier existieren sowohl kommerzielle Produkte als auch Open-Source-Projekte.

Wir nutzen die Open-Source-Implementierung

Apache ActiveMQ in der Version 5.15.8.

Sie kann über die Webseite `http://activemq.apache.org` heruntergeladen werden.

ApacheMQ implementiert vollständig JMS 1.1.[1] Wir verwenden hier dieses *Classic JMS API*.

Komponenten einer JMS-Anwendung

Abbildung 5-2 zeigt die JMS-Architektur.

Abbildung 5-2: JMS-Architektur

Eine JMS-Anwendung besteht aus einem JMS-Provider und mehreren JMS-Clients.

- *JMS-Provider*
  Der JMS-Provider ist der MOM-Server, in unserem Fall also ActiveMQ.
- *JMS-Clients*
  JMS-Clients sind die Java-Programme, die Nachrichten senden (Message Producer) bzw. empfangen (Message Consumer).

1 Das Begleitmaterial enthält jar-Datei und API-Dokumentation zu JMS.

- *Messages*
  Messages (Nachrichten) haben ein festgelegtes Format und werden von JMS-Clients erzeugt, versandt und empfangen.
- *Administrierte Objekte*
  Administrierte Objekte werden vom JMS-Provider bereitgestellt und in einem so genannten Namensdienst veröffentlicht. Diese Objekte werden von JMS-Clients über *JNDI* (*Java Naming and Directory Interface*)[2] angefordert, um Nachrichten senden oder empfangen zu können. *ConnectionFactory* und *Destination* sind administrierte Objekte.

Die Interfaces und Klassen des JMS-API gehören alle zum Paket `javax.jms`. Im Folgenden werden die in Abbildung 5-3 aufgeführten Interfaces kurz beschrieben.

Abbildung 5-3: Wichtige JMS-Schnittstellen

`ConnectionFactory`

Eine Instanz dieses Typs wird vom JMS-Client genutzt, um Verbindungen zu einem JMS-Provider aufzubauen. Sie wird vom Administrator konfiguriert (administriertes Objekt) und über JNDI zur Verfügung gestellt.

2 https://de.wikipedia.org/wiki/Java_Naming_and_Directory_Interface

`Connection`

Eine Verbindung wird von einer `ConnectionFactory` erzeugt und stellt einen Kommunikationskanal zum JMS-Provider dar.

`Session`

Eine Sitzung wird von einer geöffneten Verbindung erstellt und repräsentiert einen Kontext, in dem Nachrichten, Sender oder Empfänger erzeugt werden.

`Message`

Eine Nachricht besteht aus einem Kopf (*Header*), Eigenschaften (*Properties*) und einem Rumpf (*Body*).

Der Kopf enthält verschiedene Felder, die zur Identifizierung und Verwaltung genutzt werden und zum Teil vom JMS-Provider automatisch gesetzt werden. Einige Header werden in den Beispielen dieses Kapitels verwendet.

Abbildung 5-4: Aufbau einer Nachricht

Anwendungsbezogen können Properties hinterlegt werden. Hierzu stehen die folgenden `Message`-Methoden zur Verfügung:

```
void setXxxProperty(String name, xxx value) throws JMSException
xxx getXxxProperty(String name) throws JMSException
```

Hierbei steht *xxx* für `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` oder `String`.

Der Rumpf speichert die Nutzdaten. Es existieren unterschiedliche Nachrichtentypen in Form von Subinterfaces wie z. B.:

`BytesMessage`, `MapMessage` und `TextMessage`.

`MessageProducer`

Der `MessageProducer` hat die Aufgabe, Nachrichten an ein Ziel zu versenden.

`MessageConsumer`

Der MessageConsumer empfängt Nachrichten vom JMS-Provider.

`Destination`

Ein Nachrichtenziel (`Destination`-Objekt) repräsentiert je nach Nachrichtenmodell (*Point-to-Point*, *Publish/Subscribe*) eine Warteschlange des JMS-Providers. Das Nachrichtenziel kann vom Administrator eingerichtet (administriertes Objekt) und über JNDI bereitgestellt werden.

5.2 Das Point-to-Point-Modell

Beim *Point-to-Point-Modell* (*P2P*) wird eine vom Sender erzeugte Nachricht über eine *Queue* an genau einen Empfänger übermittelt (siehe Abbildung 5-1).

Sobald der Empfänger den Erhalt der Nachricht bestätigt hat, gilt sie als verbraucht und kann nicht erneut geholt werden. MOM-Server bieten das Konzept der persistenten Warteschlange. Nachrichten werden bis zu ihrer Auslieferung dauerhaft in einer Datenbank gespeichert, sodass sie ihren Empfänger auf jeden Fall erreichen, wenn er wieder aktiv ist.

In den Programmbeispielen dieses Abschnitts werden einfache Textnachrichten gesendet und empfangen.

5.2.1 Installation und Konfiguration von ActiveMQ

Abbildung 5-5: ActiveMQ Console

Im Installationsverzeichnis von *ActiveMQ* kann der Message Broker gestartet werden:

```
cd <Installationsverzeichnis>
bin\activemq start
```

Er läuft standardmäßig auf der Portnummer 61616. Durch Eingabe von *Strg + C* kann er später beendet werden.

Die Aktivitäten des Brokers sowie Nachrichtenziele (*Queues*, *Topics*) können mit Hilfe eines Browsers überwacht werden:

```
http://localhost:8161/admin
```

Diese Anwendung ist geschützt.

Nutzername: admin
Passwort: admin

Ein JMS-Client muss zunächst `ConnectionFactory` und `Destination` über *JNDI* abfragen (*lookup*).

jndi.properties

Die Verbindungsdaten für diesen Zugriff befinden sich in der Datei *jndi.properties*, die im Sourcen-Verzeichnis bzw. Verzeichnis der Bytecodes liegt:

```
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616
# queue.[jndiName]=[physicalName]
queue.myQueue=demo.queue1
```

Arbeitet man mit verschiedenen Rechnern, so kann natürlich statt `localhost` die IP-Adresse des Rechners, auf dem ActiveMQ läuft, angegeben werden.

JNDI-Name und physischer Name für die Queue können frei vergeben werden. Der JNDI-Name der Queue ist hier `myQueue`. Im Fall, dass die zugehörige physische Ressource `demo.queue1` nicht existiert, wird sie von ActiveMQ erzeugt, sobald eine Nachricht an dieses Ziel gesendet wird. Gleiches gilt auch für Topics im Publish/Subscribe-Modell.

jar-Datei zur Compilierung und Ausführung

Zur Compilierung und Ausführung der Programme wird die jar-Datei

```
activemq-all-5.15.8.jar
```

aus der ActiveMQ-Installation benötigt.

Zunächst werden die in den Programmbeispielen benutzten Typen und Methoden vorgestellt.

5.2.2 Wichtige Typen und Methoden des JMS-API

Die meisten Methoden des JMS-API lösen bei Fehlern eine Ausnahme vom Typ `JMSException` (Subklasse von `Exception`) aus.

`ConnectionFactory`-Methode:

`Connection createConnection() throws JMSException`
: baut eine Verbindung auf.

`Connection`-Methoden:

```
Session createSession(boolean transacted, int acknowledgeMode)
       throws JMSException
```
: erzeugt eine Session. `transacted` gibt an, ob das Senden bzw. Empfangen von Nachrichten innerhalb einer Transaktion stattfinden soll. `acknowledgeMode` legt fest, wer für die Empfangsbestätigung zuständig ist.
In den Beispielen wird `Session.AUTO_ACKNOWLEDGE` (automatische Bestätigung durch die Session) bzw. `Session.CLIENT_ACKNOWLEDGE` (explizite Bestätigung durch den Client) genutzt.

`void start() throws JMSException`
: startet die Auslieferung eingegangener Nachrichten beim Message Consumer.

`void stop() throws JMSException`
: stoppt den Empfang von Nachrichten beim Message Consumer. Mit `start` kann der Nachrichtenempfang fortgesetzt werden.

`void close() throws JMSException`
: schließt die Verbindung.

`Session`-Methoden:

`TextMessage createTextMessage() throws JMSException`
: erzeugt eine Textnachricht.

`Message createMessage() throws JMSException`
: erzeugt eine Nachricht ohne Rumpf.

```
MessageProducer createProducer(Destination destination)
       throws JMSException
```
: erzeugt einen Message Producer für das Nachrichtenziel. `destination` kann auch `null` sein, dann muss beim Aufruf der `MessageProducer`-Methode `send` (siehe unten) festgelegt werden, an welches Ziel die Nachricht geschickt werden soll.

```
MessageConsumer createConsumer(Destination destination)
       throws JMSException
```
: erzeugt einen Message Consumer für das Nachrichtenziel.

`void close() throws JMSException`
: schließt die Sitzung.

`MessageProducer`-Methoden:

`void setTimeToLive(long timeToLive) throws JMSException`
: setzt die Gültigkeitsdauer für Nachrichten in Millisekunden (0 steht für unbegrenzt und ist die Standard-Einstellung).

`void setPriority(int priority)`
: setzt die Priorität der Nachricht. 0 ist die niedrigste, 9 die höchste Priorität.

`void send(Message message) throws JMSException`
: sendet die Nachricht `message`.

`void send(Destination destination, Message message) throws JMSException`
: sendet die Nachricht `message` an das Ziel `destination`, falls der Message Producer ohne Nachrichtenziel erzeugt wurde.

`void close() throws JMSException`
: beendet den Message Producer.

`MessageConsumer`-Methoden:

`Message receive() throws JMSException`
: wartet auf das Eintreffen einer Nachricht und kehrt dann zurück.

`Message receive(long timeout) throws JMSException`
: wartet maximal `timeout` Millisekunden auf das Eintreffen einer Nachricht und kehrt dann zurück. Der Aufruf mit `0` ist gleichbedeutend mit dem parameterlosen Aufruf `receive()`.

`void setMessageListener(MessageListener listener) throws JMSException`
: registriert ein Objekt vom Typ `MessageListener`, das asynchron Nachrichten empfängt.

`void close() throws JMSException`
: beendet den Message Consumer.

Das Interface `MessageListener` definiert die Methode

`void onMessage(Message message)`

`TextMessage` ist Subinterface von `Message`.

`TextMessage`-Methoden:

`void setText(String string) throws JMSException`
: setzt den Inhalt der Textnachricht.

`String getText() throws JMSException`
: liefert den Inhalt der Textnachricht.

`Session`-, `MessageProducer`- und `MessageConsumer`-Objekte einer geschlossenen Verbindung müssen nicht explizit geschlossen werden. Beim Schließen einer Verbindung werden auch alle temporären Queues (siehe Kapitel 5.3) gelöscht.

5.2.3 Das Pull- und Push-Prinzip

→ Projekt p2p

Im folgenden ersten Beispiel demonstrieren wir, auf welche Arten ein Consumer Nachrichten empfangen kann. Der Producer sendet eine Textnachricht an den JMS-Provider ActiveMQ. Optional kann eine Gültigkeitsdauer in Millisekunden angegeben werden, im Beispiel: 0 = unbegrenzt.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Producer {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageProducer messageProducer = session.createProducer(queue);

         TextMessage message = session.createTextMessage();
         message.setText("Das ist ein Test.");
         message.setStringProperty("Status", "Test");
         messageProducer.setTimeToLive(0);
         messageProducer.send(message);
      } catch (NamingException | JMSException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }
}
```

Pull-Prinzip

Im folgenden Programm wartet der Empfänger aktiv auf das Eintreffen von Nachrichten (*Pull-Prinzip*). Nach 30 Sekunden Wartezeit wird das Programm beendet. Hier wird der Empfang einer Nachricht nach Erhalt automatisch bestätigt.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer1 {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session.createConsumer(queue);
         connection.start();

         Message message;
         while ((message = messageConsumer.receive(30000)) != null) {
            if (message instanceof TextMessage) {
               TextMessage textMessage = (TextMessage) message;
               System.out.println("Nachricht: " + textMessage.getText());
               System.out.println("Status: " +
                  message.getStringProperty("Status"));
            }
         }
      } catch (NamingException | JMSException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }
}
```

Push-Prinzip

Im Programm `Consumer2` wird der Empfänger vom MOM-Server durch die Callback-Methode `onMessage` über das Eintreffen einer Nachricht informiert (*Push-Prinzip*). Das Programm wird nach 30 Sekunden beendet.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer2 {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session.createConsumer(queue);
         messageConsumer.setMessageListener(message -> handleMessage(message));
         connection.start();

         Thread.sleep(30000);
      } catch (NamingException | JMSException | InterruptedException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }

   private static void handleMessage(Message message) {
      try {
         if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println("Nachricht: " + textMessage.getText());
            System.out.println("Status: " + message.getStringProperty("Status"));
         }
```

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

Redelivery

Das Programm `Consumer3` entspricht im Aufbau dem Programm `Consumer1`. Jedoch muss hier der Empfang einer Nachricht explizit bestätigt werden:

```
Session.CLIENT_ACKNOWLEDGE.
```

Hierzu dient die `Message`-Methode

```
void acknowledge() throws JMSException
```

Kurz nach Empfang der ersten Nachricht wird ein Abbruch simuliert und die `Session`-Methode

```
void recover() throws JMSException
```

aufgerufen. Das führt dazu, dass die bisher nicht bestätigten Nachrichten erneut ausgeliefert werden (*Redelivery*).

Ob eine Nachricht erneut ausgeliefert wurde, kann mit der `Message`-Methode

```
boolean getJMSRedelivered()throws JMSException
```

abgefragt werden.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer3 {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.CLIENT_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session.createConsumer(queue);
         connection.start();
```

```
            Message message;
            int count = 0;
            while ((message = messageConsumer.receive(30000)) != null) {
               try {
                  if (message instanceof TextMessage) {
                     count++;

                     TextMessage textMessage = (TextMessage) message;
                     System.out.println("\nNachricht: " + textMessage.getText());
                     System.out.println("Status: " +
                          message.getStringProperty("Status"));
                     System.out.println("Redelivered: " +
                          textMessage.getJMSRedelivered());

                     System.out.println("Nachricht wird verarbeitet");
                     try {
                          Thread.sleep(3000);
                     } catch (InterruptedException e) {
                     }

                     if (count == 1)
                          throw new RuntimeException("Abbruch!");

                     System.out.println("Nachricht wurde erfolgreich verarbeitet");
                     message.acknowledge();
                  }
               } catch (RuntimeException e) {
                  System.err.println(e);
                  if (session != null) {
                     try {
                          session.recover();
                     } catch (JMSException e1) {
                          System.err.println(e);
                     }
                  }
               }
            }
         } catch (NamingException | JMSException e) {
            System.err.println(e);
         } finally {
            if (connection != null) {
               try {
                  connection.close();
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         }
      }
}
```

Setzen Sie die Umgebungsvariable `ACTIVEMQ_HOME` dauerhaft auf den Installationsort von ActiveMQ (z. B. `D:\Programme\apache-activemq-5.15.8`), so können die Programme wie folgt aufgerufen werden. Vorher muss IntelliJ IDEA neu gestartet werden, damit die neue Umgebungsvariable im Terminal der Entwicklungsumgebung verfügbar ist.

Aufruf des Producers

```
java -cp out/production/p2p;%ACTIVEMQ_HOME%/activemq-all-5.15.8.jar Producer
```

Die folgenden Programme können *alternativ* aufgerufen werden.

```
java -cp out/production/p2p;%ACTIVEMQ_HOME%/activemq-all-5.15.8.jar Consumer1
java -cp out/production/p2p;%ACTIVEMQ_HOME%/activemq-all-5.15.8.jar Consumer2
java -cp out/production/p2p;%ACTIVEMQ_HOME%/activemq-all-5.15.8.jar Consumer3
```

Da im Point-to-Point-Modell nur immer ein Consumer eine Nachricht empfängt, muss der Producer wiederholt gestartet werden.

Hier die Ausgabe beim Aufruf von `Consumer3`:

```
Nachricht: Das ist ein Test.
Status: Test
Redelivered: false
Nachricht wird verarbeitet
java.lang.RuntimeException: Abbruch!

Nachricht: Das ist ein Test.
Status: Test
Redelivered: true
Nachricht wird verarbeitet
Nachricht wurde erfolgreich verarbeitet
```

5.2.4 Der QueueBrowser

Das folgende Programm ermöglicht die Abfrage des Inhalts einer Warteschlange, ohne die Nachrichten zu löschen.

Hierzu gibt es das Interface `QueueBrowser` mit den folgenden Methoden:

`java.util.Enumeration getEnumeration() throws JMSException`
: liefert ein `Enumeration`-Objekt zum sequentiellen Durchlaufen der in der Warteschlange enthaltenen Nachrichten.

`void close() throws JMSException`
: schließt den Browser.

Mit der `Session`-Methode

```
QueueBrowser createBrowser(Queue queue) throws JMSException
```

wird ein Browser für die Warteschlange `queue` erzeugt. `Queue` ist Subinterface von `Destination`.

Header-Felder einer Nachricht können mit get-Methoden des Interface `Message` abgefragt werden. Das Programm verwendet die folgenden vom JMS-Provider automatisch gesetzten Felder:

`JMSDestination`

Das Nachrichtenziel, an das die Nachricht gesendet wurde (Typ `Destination`).

`JMSMessageID`

Eindeutiger Bezeichner zur Identifikation einer Nachricht (Typ `String`).

`JMSPriority`

Wert zwischen 0 (niedrig) und 9 (hoch), der die Priorität der Auslieferung einer Nachricht festlegt (Typ `int`). Die Priorität kann mit der `MessageProducer`-Methode `setPriority` (siehe oben) gesetzt werden.

`JMSTimestamp`

Zeitpunkt der Übergabe der Nachricht an den JMS-Provider in Millisekunden (Typ `long`).

`JMSExpiration`

Zeitpunkt des Verfalls einer Nachricht in Millisekunden (Typ `long`). Dieser Zeitpunkt kann mit der `MessageProducer`-Methode `setTimeToLive` (siehe oben) bestimmt werden.

Die `Message`-Methode

```
java.util.Enumeration getPropertyNames() throws JMSException
```

liefert ein `Enumeration`-Objekt mit den Namen der in der Nachricht hinterlegten Properties.

→ Projekt p2p

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Enumeration;

public class QueueInfo {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");
```

```
            connection = factory.createConnection();
            connection.start();
            Session session = connection
               .createSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueBrowser browser = session.createBrowser((Queue) queue);

            Enumeration e = browser.getEnumeration();
            SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
            int cnt = 0;

            while (e.hasMoreElements()) {
               cnt++;

               Message message = (Message) e.nextElement();

               System.out.print(cnt + ".");
               System.out.println("\tDestination: " + message.getJMSDestination());
               System.out.println("\tMessageID: " + message.getJMSMessageID());
               System.out.println("\tTimestamp: " + f.format(
                  new Date(message.getJMSTimestamp())));
               System.out.println("\tPriority: " + message.getJMSPriority());

               long expiration = message.getJMSExpiration();

               if (expiration == 0) {
                  System.out.println("\tExpiration: 0");
               } else {
                  System.out.println("\tExpiration: " + f.format(
                     new Date(expiration)));
               }

               System.out.println("\tProperties:");

               Enumeration names = message.getPropertyNames();

               while (names.hasMoreElements()) {
                  String name = (String) names.nextElement();
                  System.out.println("\t\t" + name + " = " +
                     message.getStringProperty(name));
               }

               System.out.println();
            }
         } catch (NamingException | JMSException e) {
            System.err.println(e);
         } finally {
            if (connection != null) {
               try {
                  connection.close();
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         }
      }
}
```

Ausgabebeispiel:

```
1.      Destination: queue://demo.queue1
        MessageID: ID:MSI-53363-1548167973341-1:1:1:1:1
        Timestamp: 22.01.2019 15:39:33
        Priority: 4
        Expiration: 0
        Properties:
                Status = Test
```

5.3 Das Request/Reply-Modell

Ein Spezialfall des Point-to-Point-Modells ist das *Request/Reply-Modell*, das eine synchrone Kommunikation zwischen Sender und Empfänger simuliert. Der Sender wartet solange, bis die Antwortnachricht eintrifft. Hierzu wird eine temporäre Warteschlange für die Antwortnachricht genutzt. Diese wird für die Dauer der Kommunikation erzeugt und dem Empfänger über den Nachrichten-Header `JMSReplyTo` (Typ `Destination`) mitgeteilt.

Abbildung 5-6: Das Request/Reply-Modell

Das Interface `TemoraryQueue` ist Subinterface von `Queue` und enthält die Lösch-Methode

```
void delete() throws JMSException
```

Die folgende `Session`-Methode erzeugt eine temporäre Warteschlange:

```
TemporaryQueue createTemporaryQueue() throws JMSException
```

→ Projekt request_reply

Wir realisieren zur Demonstration einen synchronen Echo-Dienst.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
```

```
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;

public class Replier {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();

         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session.createConsumer(queue);
         connection.start();

         Thread t = new Thread() {
            @Override
            public void run() {
               System.out.println("Replier gestartet ...");

               try {
                  while (true) {
                        TextMessage request =
                                (TextMessage) messageConsumer.receive();
                        if (request == null)
                                break;

                        String text = request.getText();
                        Destination tempQueue =
                                (Destination) request.getJMSReplyTo();
                        TextMessage response = session.createTextMessage();
                        response.setText("Anwort: " + text);
                        MessageProducer producer =
                                session.createProducer(tempQueue);
                        producer.send(response);
                        producer.close();
                  }
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         };

         t.start();

         System.out.println("ENTER eingeben, um das Programm zu beenden.");
         System.in.read();
         messageConsumer.close();
      } catch (NamingException | JMSException | IOException e) {
         System.err.println(e);
      } finally {
```

```
            if (connection != null) {
               try {
                  connection.close();
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         }
      }
}
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TemporaryQueue;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Scanner;

public class Requestor {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         TemporaryQueue tempQueue = session.createTemporaryQueue();
         MessageProducer messageProducer = session.createProducer(queue);
         MessageConsumer messageConsumer = session.createConsumer(tempQueue);
         connection.start();

         Scanner scanner = new Scanner(System.in);
         while (true) {
            System.out.print("> ");
            String text = scanner.nextLine();
            if (text.equals("q"))
               break;

            TextMessage request = session.createTextMessage();
            request.setText(text);
            request.setJMSReplyTo(tempQueue);
            messageProducer.send(request);

            TextMessage response = (TextMessage) messageConsumer.receive();
            System.out.println(response.getText());
         }
      } catch (NamingException | JMSException e) {
         System.err.println(e);
      } finally {
```

```
            if (connection != null) {
               try {
                  connection.close();
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         }
      }
}
```

5.4 Das Publish/Subscribe-Modell

Beim *Publish/Subscribe-Modell* sendet der *Publisher* Nachrichten zu einem bestimmten Thema (*Topic*), die vom Vermittler (MOM-Server) verteilt werden. *Subscriber* können beim Vermittler ein bestimmtes Thema abonnieren. Alle Subscriber erhalten dann die Nachrichten, die zu diesem Thema veröffentlicht wurden. Im Unterschied zum P2P-Modell erhalten sie aber nur die Nachrichten, die *während ihrer aktiven Verbindung* mit dem Vermittler versandt wurden. Publisher und Subscriber sind vollständig unabhängig voneinander.

Es können jedoch auch dauerhafte Abonnements eingerichtet werden. Ein *Subscriber* erhält dann später auch die Nachrichten zu einem Thema, die veröffentlicht wurden, während er keine Verbindung zum Vermittler hatte (siehe Kapitel 5.5).

Abbildung 5-7: Das Publish/Subscribe-Modell

→ Projekt non_durable_topic

Das folgende Programmbeispiel nutzt den Nachrichtentyp `MapMessage`. Eine solche Nachricht besteht im Nachrichtenrumpf aus Schlüssel-Wert-Paaren. Der Schlüssel ist vom Typ `String`, der Wert vom Typ `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `String` oder `byte[]`.

Zu jedem Datentyp existieren die get- und set-Methoden:

```
void setXxx(xxx value) throws JMSException
xxx getXxx(String name) throws JMSException
```

Für den Typ `byte[]` heißen die Methoden `setBytes` und `getBytes`.

Eine `MapMessage` wird mit der folgenden `Session`-Methode erzeugt:

```
MapMessage createMapMessage() throws JMSException
```

Der zur Queue analoge Eintrag für ein Topic in der Datei *jndi.properties* ist hier:

```
topic.myTopic = demo.topic1
```

Beispiel:

Der Publisher veröffentlicht in regelmäßigen Abständen Nachrichten mit Zeitstempel. Subscriber zeigen diese Nachrichten an.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.time.LocalTime;

public class Publisher {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageProducer messageProducer = session.createProducer(topic);

         for (int i = 0; i < 24; i++) {
            MapMessage message = session.createMapMessage();
            message.setString("Time", LocalTime.now().toString());
            message.setString("Message", "Message " + i);
            messageProducer.send(message);
            Thread.sleep(5000);
         }
      } catch (NamingException | JMSException | InterruptedException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session.createConsumer(topic);
         messageConsumer.setMessageListener(message -> handleMessage(message));
         connection.start();
         Thread.sleep(60000);
      } catch (NamingException | JMSException | InterruptedException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }

   private static void handleMessage(Message message) {
      try {
         if (message instanceof MapMessage) {
            MapMessage mapMessage = (MapMessage) message;
            System.out.println(mapMessage.getString("Time"));
            System.out.println(mapMessage.getString("Message"));
            System.out.println();
         }
      } catch (JMSException e) {
         System.err.println(e);
      }
   }
}
```

Mehrere Subscriber können in verschiedenen Konsolen gestartet werden. Sie sind jeweils 60 Sekunden aktiv. Sie erhalten nur die Nachrichten, die während ihrer Verbindung mit dem Server (ActiveMQ) bei diesem eingegangen sind.

5.5 Dauerhafte Subscriber

Ein Subscriber erhält standardmäßig nur solche Nachrichten, die an ein Topic geschickt werden, während er eine Verbindung zum MOM-Server hat.

Ein *dauerhafter Subscriber* kann auch nachträglich Nachrichten abrufen, die geschickt wurden, während er inaktiv war. Der Server muss dann für jeden dauerhaften Subscriber eine Instanz der Nachricht vorhalten, da jeder dieser Subscriber eine eigene Kopie der Nachricht erhält.

Hierzu muss eine Subskription eindeutig identifiziert werden können. Zur Identifizierung einer Subskription dienen

- die *Client-Identifikation* für die Verbindung und
- der *Name*, der die Subskription innerhalb dieser Client-Identifikation eindeutig identifiziert.

Innerhalb ein und derselben Verbindung kann ein Client mehrere Subskriptionen (bei verschiedenen Topics) haben.

Die Client-Identifikation wird mit der folgenden `Connection`-Methode gesetzt, unmittelbar nachdem das `Connection`-Objekt erzeugt wurde:

```
void setClientID(String clientID) throws JMSException
```

Die `Connection`-Methode

```
String getClientID() throws JMSException
```

liefert die Client-Identifikation der Verbindung.

Die `Session`-Methode

```
TopicSubscriber createDurableSubscriber(Topic topic,  String name)
    throws JMSException
```

erzeugt einen dauerhaften Subscriber. `name` ist der oben erwähnte Subskriptionsname. `Topic` ist Subinterface von Destination. `TopicSubscriber` ist Subinterface von `MessageConsumer`.

Mit der `Session`-Methode

```
void unsubscribe(String name) throws JMSException
```

kann sich der Subscriber vom Server abmelden. `name` ist der Subskriptionsname.

→ Projekt durable_topic

Das folgende Beispiel demonstriert Anmeldung, Abruf von Nachrichten und Abmeldung eines dauerhaften Subscribers.

Der Publisher entspricht dem aus Kapitel 5.4.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.Topic;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class DurableSubscriber {
   public static void main(String[] args) {
      String clientID = "0001";
      String subscriptionName = "test";

      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");
         connection = factory.createConnection();
         connection.setClientID(clientID);
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer messageConsumer = session
            .createDurableSubscriber((Topic) topic, subscriptionName);
         messageConsumer.setMessageListener(message -> handleMessage(message));
         connection.start();
         Thread.sleep(60000);
      } catch (NamingException | JMSException | InterruptedException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }

   private static void handleMessage(Message message) {
      try {
         if (message instanceof MapMessage) {
            MapMessage mapMessage = (MapMessage) message;
            System.out.println(mapMessage.getString("Time"));
            System.out.println(mapMessage.getString("Message"));
            System.out.println();
         }
      } catch (JMSException e) {
         System.err.println(e);
      }
   }
}
```

Die Identifikation des Subscribers erfolgt über die Client-Id "0001" und den Subskriptionsnamen "test".

Testszenario

Starten Sie den Publisher, danach in einer neuen Konsole den Subscriber. Der Publisher sendet alle 5 Sekunden eine Nachricht an den Server. Die Ausgabe des Subschribers könnte so aussehen:

```
12:48:24.128784800
Message 3

12:48:29.190096
Message 4

12:48:34.251500500
Message 5

12:48:39.281497900
Message 6
```

Warten Sie ca. 10 Sekunden und starten Sie den Subscriber erneut. Es werden auch die Nachrichten ausgegeben, die den Server zur inaktiven Zeit des Subscribers erreicht haben, also:

```
12:48:44.311505300
Message 7

12:48:49.372818100
Message 8

12:48:54.434301200
Message 9
...
```

Mit der ActiveMQ Console (siehe Kapitel 5.2) können die aktiven und nicht aktiven Subscriber abgefragt werden.

Das folgende Programm meldet den Subscriber vom Server ab.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Session;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Unsubscribe {
   public static void main(String[] args) {
      String clientID = "0001";
      String subscriptionName = "test";

      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");

         connection = factory.createConnection();
         connection.setClientID(clientID);
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);

         session.unsubscribe(subscriptionName);
      } catch (NamingException | JMSException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }
}
```

5.6 Nachrichten filtern

Um nur die Nachrichten zu empfangen, die bestimmte Bedingungen erfüllen, kann ein Filtermechanismus verwendet werden.

Die Nachrichten werden beim JMS-Provider auf der Basis von Bedingungen, die sich auf *Header-Felder* und *Properties* beziehen können, selektiert.

Zur Formulierung dieser Bedingungen wird eine *SQL-ähnliche Notation* verwendet. Es können Vergleichsoperatoren (`=`, `<`, `<=`, `>`, `>=`), logische Operatoren (`AND`, `OR`, `NOT`), sowie `IN`, `LIKE` und `IS NULL` benutzt werden.

Ein *Message Selector* ist ein String, der einen so gebildeten Bedingungsausdruck enthält.

Im Programmbeispiel wird z. B. der folgende *Message Selector* verwendet:

```
"Priority = 'normal' OR Priority = 'high'"
```

Beim Erzeugen des Empfängers kann der *Message Selector* festgelegt werden. Hierzu stehen die folgenden `Session`-Methoden zur Verfügung:

```
MessageConsumer createConsumer(Destination destination, String
       messageSelector, boolean noLocal) throws JMSException
```

```
TopicSubscriber createDurableSubscriber(Topic topic, String name, String
       messageSelector, boolean noLocal) throws JMSException
```

Für den Fall, dass über dieselbe Verbindung für ein Topic Nachrichten sowohl gesendet als auch empfangen werden, kann der Empfänger festlegen, ob von ihm selbst gesendete Nachrichten auch von ihm wieder empfangen werden sollen. Hat `noLocal` den Wert `true`, so wird dies ausgeschlossen.

→ Projekt filter

Im folgenden Beispiel erzeugt der Publisher zufallsgesteuert Nachrichten unterschiedlicher Priorität (`low`, `normal`, `high`).

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Publisher {
   public static void main(String[] args) {
      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");

         connection = factory.createConnection();
         Session session = connection
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageProducer messageProducer = session.createProducer(topic);

         for (int i = 0; i < 60; i++) {
            TextMessage message = session.createTextMessage();
            message.setText("Nachricht " + i);

            if (Math.random() < 0.25) {
               message.setStringProperty("Priority", "low");
            } else if (Math.random() < 0.75) {
               message.setStringProperty("Priority", "normal");
            } else {
               message.setStringProperty("Priority", "high");
            }
```

```
            messageProducer.send(message);
            System.out.println(message.getText() + ": "
               + message.getStringProperty("Priority"));
            Thread.sleep(1000);
         }
      } catch (NamingException | JMSException | InterruptedException e) {
         System.err.println(e);
      } finally {
         if (connection != null) {
            try {
               connection.close();
            } catch (JMSException e) {
               System.err.println(e);
            }
         }
      }
   }
}
```

Der Subscriber kann

- alle Nachrichten,
- Nachrichten mit normaler und hoher Priorität oder
- nur Nachrichten mit hoher Priorität

empfangen.

Das Programm hierzu existiert in drei Varianten, die sich nur im *Message Selector* unterscheiden.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber1 {
   public static void main(String[] args) {
      String messageSelector = "";

      Connection connection = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination topic = (Destination) ctx.lookup("myTopic");

         connection = factory.createConnection();
```

```
            Session session = connection
                .createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer messageConsumer = session
                .createConsumer(topic, messageSelector);
            messageConsumer.setMessageListener(message -> handleMessage(message));
            connection.start();

            Thread.sleep(30000);
        } catch (NamingException | JMSException | InterruptedException e) {
            System.err.println(e);
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    System.err.println(e);
                }
            }
        }
    }

    private static void handleMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                TextMessage textMessage = (TextMessage) message;
                System.out.print(textMessage.getText());
                System.out.println(": " + textMessage.getStringProperty("Priority"));
            }
        } catch (JMSException e) {
            System.err.println(e);
        }
    }
}
```

Das Programm `Subcriber2` hat den *Message Selector*

```
String messageSelector = "Priority = 'normal' OR " + "Priority = 'high'";
```

`Subcriber3` hat den *Message Selector*

```
String messageSelector = "Priority = 'high'";
```

5.7 Transaktionen

Die Kommunikation mit dem JMS-Provider kann *transaktionsorientiert* erfolgen.

Der *Transaktionsmodus* (`false` oder `true`) wird durch den ersten Aufrufparameter der `Connection`-Methode `createSession` eingestellt (siehe Kapitel 5.2).

Es wird zwischen Transaktionssteuerung beim Senden und Empfangen unterschieden.

Mehrere Nachrichten können auf der Senderseite in einer Transaktion gebündelt werden. Der Versand der Nachrichten wird solange zurückgehalten, bis die Transaktion mit dem Aufruf der `Session`-Methode `commit` abgeschlossen wird. Die

gesamte Transaktion kann durch den Aufruf der `Session`-Methode `rollback` rückgängig gemacht werden. `commit` bzw. `rollback` öffnet sofort wieder eine neue Transaktion.

Auf der Empfängerseite können innerhalb einer Transaktion nacheinander mehrere Nachrichten empfangen werden. Der Aufruf von `commit` bestätigt den Empfang und schließt die Transaktion ab. Wird die Transaktion mit `rollback` beendet, wird versucht, alle Nachrichten, die in der letzten Transaktion empfangen wurden, erneut zuzustellen. Ob eine Nachricht schon einmal zugestellt wurde, kann durch Auswerten des Header-Feldes `JMSRedelivered` (`true` oder `false`) ermittelt werden.

`Session-Methoden:`

`void commit() throws JMSException`

`void rollback() throws JMSException`

`boolean getTransacted() throws JMSException`
prüft, ob die aktuelle Session mit Transaktionssteuerung arbeitet.

→ Projekt transaction

Das folgende Beispiel demonstriert das Arbeiten mit Transaktionen. Der *Producer* schickt innerhalb einer Transaktion vier Textnachrichten und eine weitere Nachricht als Abschlussnachricht, die das Ende einer Nachrichtenfolge signalisiert.

Hat die Programmvariable `exception` den Wert `true`, so wird eine Fehlersituation simuliert, die zum Abbruch der Transaktion mit anschließendem `rollback` führt.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.time.LocalTime;

public class Producer {
   public static void main(String[] args) {
      boolean exception = false;

      Connection connection = null;
      Session session = null;
      try {
         Context ctx = new InitialContext();
         ConnectionFactory factory =
            (ConnectionFactory) ctx.lookup("ConnectionFactory");
         Destination queue = (Destination) ctx.lookup("myQueue");
```

```
            connection = factory.createConnection();
            session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer = session.createProducer(queue);

            for (int i = 0; i < 4; i++) {
               TextMessage message = session.createTextMessage();
               message.setText("Nachricht " + i + " " + LocalTime.now().toString());
               messageProducer.send(message);
            }

            // Leere Nachricht signalisiert den Abschluss
            messageProducer.send(session.createMessage());

            if (exception) {
               throw new RuntimeException("Simulierter Fehler");
            }

            session.commit();
            System.out.println("Committed");
         } catch (RuntimeException e) {
            System.err.println(e);
            try {
               session.rollback();
               System.out.println("Rolled back");
            } catch (JMSException e1) {
               System.err.println(e1);
            }
         } catch (NamingException | JMSException e) {
            System.err.println(e);
         } finally {
            if (connection != null) {
               try {
                  connection.close();
               } catch (JMSException e) {
                  System.err.println(e);
               }
            }
         }
      }
}



import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer {
   public static void main(String[] args) {
      boolean exception = true;
```

```
        Connection connection = null;
        try {
            Context ctx = new InitialContext();
            ConnectionFactory factory =
                (ConnectionFactory) ctx.lookup("ConnectionFactory");
            Destination queue = (Destination) ctx.lookup("myQueue");

            connection = factory.createConnection();
            Session session = connection
                .createSession(true, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer messageConsumer = session.createConsumer(queue);
            connection.start();

            while (true) {
                try {
                    Message message = messageConsumer.receive(30000);
                    if (message == null)
                        break;

                    if (message instanceof TextMessage) {
                        TextMessage textMessage = (TextMessage) message;
                        System.out.println(textMessage.getText() +
                             " (Redelivered: " + textMessage.getJMSRedelivered() + ")");
                    } else {
                        session.commit();
                        System.out.println("Committed");
                        continue;
                    }

                    if (exception) {
                        exception = false;
                        throw new RuntimeException("Simulierter Fehler");
                    }
                } catch (RuntimeException e) {
                    System.err.println(e);
                    try {
                        session.rollback();
                        System.out.println("Rolled back");
                    } catch (JMSException e1) {
                        System.err.println(e1);
                    }
                }
            }
        } catch (NamingException | JMSException e) {
            System.err.println(e);
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    System.err.println(e);
                }
            }
        }
    }
}
```

Die Nachrichten werden in derselben Reihenfolge empfangen wie sie gesendet wurden (*FIFO-Prinzip*).

Nach dem Empfang der ersten Nachricht wird ein Abbruch simuliert, der zum *Rollback* führt. Diese Nachricht wird wieder ausgeliefert. Die letzte Nachricht (leere Nachricht vom Typ `Message`) signalisiert das Ende und löst *Commit* aus.

Ausgabe:

```
Nachricht 0 14:01:33.722057900 (Redelivered: false)
java.lang.RuntimeException: Simulierter Fehler
Rolled back
Nachricht 0 14:01:33.722057900 (Redelivered: true)
Nachricht 1 14:01:33.722057900 (Redelivered: false)
Nachricht 2 14:01:33.722057900 (Redelivered: false)
Nachricht 3 14:01:33.722057900 (Redelivered: false)
Committed
```

5.8 Aufgaben

1. Entwickeln Sie auf Basis des *Publish/Subscribe*-Modells einen Zeitangabe-Service, der jede Minute die aktuelle Uhrzeit übermittelt. Ein Client soll diese Zeitangabe ausgeben.

2. Durch Vergabe von Prioritäten (siehe Kapitel 5.2) kann die Reihenfolge der ausgelieferten Nachrichten beeinflusst werden. Testen Sie diesen Sachverhalt, indem Sie einen geeigneten Producer bzw. Consumer entwickeln.

   Die Prioritätensteuerung für Queues muss eigens konfiguriert werden. Hierzu muss die XML-Datei

   <ActiveMQ-Installationsverzeichnis>/conf/activemq.xml

   angepasst werden:

```
<destinationPolicy>
    <policyMap>
      <policyEntries>
        ...
        <policyEntry queue=">" prioritizedMessages="true" />
      </policyEntries>
    </policyMap>
</destinationPolicy>
```

   `queue=">"` bedeutet, dass für alle Queues Prioritäten berücksichtigt werden.

3. Implementieren Sie auf Basis des *Publish/Subscribe*-Modells ein einfaches Chat-Programm. Die Bedienung soll über Kommandozeilen erfolgen.

   Damit Nachrichten sowohl gesendet als auch empfangen werden können, muss das Chat-Programm einen Publisher und einen Subscriber enthalten.

4. Mit dem Nachrichtentyp `BytesMessage` können Daten byteweise übertragen werden. Diese Bytes werden als Rohdaten behandelt und von der *Java Virtual Machine* nicht interpretiert. Erstellen Sie JMS-Clients, die Binärdaten (z. B. ein GIF-Bild) senden bzw. empfangen.

   Die `Session`-Methode

   ```
   BytesMessage createBytesMessage() throws JMSException
   ```

   erzeugt eine `BytesMessage`.

   Zur Lösung dieser Aufgabe sollen die folgenden `BytesMessage`-Methoden verwendet werden:

   ```
   void writeBytes(byte[] value, int offset, int length) throws JMSException
   int readBytes(byte[] value) throws JMSException
   ```

5. Realisieren Sie Programme, die das folgende Szenario umsetzen:

   Ein Händler (`Vendor`) bestellt mehrere Artikel in bestimmten Stückzahlen bei einem Lieferanten (`Supplier`). Die Bestellung (`TextMessage` mit JSON) wird an die Warteschlage *orderQueue* versandt. Für die Antwort des Lieferanten wird eine temporäre Warteschlage (*replyQueue*) erzeugt.

   Der Lieferant prüft für jeden bestellten Artikel, ob er lieferbar ist. Ggf. wird eine kleinere Menge geliefert. Er sendet das Ergebnis an die Warteschlange *replyQueue* und wartet dann auf weitere Aufträge.

   Die Session beim Lieferanten arbeitet transaktionsorientiert.

6 Remote Method Invocation

In diesem Kapitel stellen wir ein System vor, das die Kommunikation zwischen Objekten, die sich auf verschiedenen Rechnern (JVMs) befinden, ermöglicht. Neben (fast) beliebigen Objekten kann auch das Verhalten (Bytecode) übertragen werden, so dass flexible Anwendungen realisiert werden können. Allerdings muss sowohl der Client als auch der Server in Java programmiert sein.

Das klassische *RPC-Modell* (RPC = Remote Procedure Call) versteckt den Aufruf einer auf einem anderen Rechner im Netz (bzw. einer anderen JVM) implementierten Prozedur hinter einem lokalen Prozeduraufruf und bietet damit ein sehr vertrautes Programmiermodell an. Das Programm, das die Prozedur aufruft, agiert als Client, das Programm, das die aufgerufene Prozedur ausführt, als Server.

Abbildung 6-1: Entfernter Prozeduraufruf im RPC-Modell

RMI (*Remote Method Invocation*) ist die objektorientierte Umsetzung des RPC-Modells in Java.

6.1 Vom lokalen zum entfernten Methodenaufruf

Das folgende Programmbeispiel zeigt, wie einfach eine Anwendung mit *lokalem* Methodenaufruf in eine Client/Server-Anwendung mit *entferntem* Methodenaufruf (RMI) umgewandelt werden kann. Die im Beispiel benutzte Methode berechnet die Summe zweier Zahlen.

→ Projekt intro

Hier zunächst die lokale Anwendung (Modul *intro1*):

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_6

```
public class AddServer {
   public double add(double x, double y) {
      return x + y;
   }
}


public class AddClient {
   public static void main(String[] args) {
      AddServer service = new AddServer();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

Die beiden Klassen werden nun mittels eines Interface entkoppelt (Modul *intro2*). Um ein Server-Objekt zu erzeugen, wird eine statische Methode benutzt, die den konkreten Namen der Implementierungsklasse (hier: `AddServer`) gegenüber dem Client verbirgt (*Factory-Methode*).

Das Klassendiagramm in Abbildung 6-2 verdeutlicht die Zusammenhänge.

Abbildung 6-2: Klassendiagramm

```
public interface Add {
   double add(double x, double y);
}


public class AddServer implements Add {
   public double add(double x, double y) {
      return x + y;
   }
}
```

```
public class AddFactory {
   public static Add getInstance() {
      return new AddServer();
   }
}


public class AddClient {
   public static void main(String[] args) {
      Add service = AddFactory.getInstance();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

Die RMI-Version der Anwendung entsteht nun durch leichte Änderung (Modul *intro3*).

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Add extends Remote {
   double add(double x, double y) throws RemoteException;
}
```

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AddServer extends UnicastRemoteObject implements Add {
   public AddServer() throws RemoteException {
   }

   public double add(double x, double y) {
      return x + y;
   }

   public static void main(String[] args) throws Exception {
      AddServer server = new AddServer();
      Naming.rebind("add", server);
   }
}
```

```
import java.rmi.Naming;

public class AddFactory {
   public static Add getInstance() throws Exception {
      return (Add) Naming.lookup("//localhost/add");
   }
}
```

```
public class AddClient {
   public static void main(String[] args) throws Exception {
      Add service = AddFactory.getInstance();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

`Add`, `AddServer` und `AddFactory` importieren RMI-spezifische Typen (Klassen und Interface).

Das Interface `Add` ist von `java.rmi.Remote` abgeleitet, die Methode `add` enthält eine `throws`-Klausel mit der Ausnahme `java.rmi.RemoteException`.

Die Klasse `AddServer` ist von `java.rmi.server.UnicastRemoteObject` abgeleitet und besitzt eine `main`-Methode, die das erzeugte Server-Objekt bei einem Verzeichnisdienst registriert. Die Methode `getInstance` der Klasse `AddFactory` erhält ein `Add`-Objekt mit Hilfe dieses Verzeichnisdienstes.

Zur Vereinfachung sollen Client und Server auf demselben Rechner (`localhost`) laufen.

Test

Start des Verzeichnisdienstes:

```
set CLASSPATH=out/production/intro3
rmiregistry
```

Start des Servers:

```
java -cp out/production/intro3 AddServer
```

Aufruf des Clients:

```
java -cp out/production/intro3 AddClient
```

Verzeichnisdienst und Server müssen am Ende mit *Strg + C* abgebrochen werden.

6.2 Die Architektur einer RMI-Anwendung

6.2.1 Protokoll und Komponenten

Das Protokoll *Remote Method Invocation* (RMI) setzt auf TCP/IP auf und verbirgt die Details einer Netzverbindung. RMI hat folgende Eigenschaften:

- Mit RMI können Methoden für Objekte aufgerufen werden, die von einer anderen virtuellen Maschine (JVM) erzeugt und verwaltet werden – in der Regel auf einem anderen Rechner.
- Für den Entwickler sieht der entfernte Methodenaufruf wie ein ganz normaler lokaler Aufruf aus.

- Entfernt aufrufbare Methoden werden in einem *Interface* deklariert. Nur hierüber kann der Client mit einem entfernten Objekt kommunizieren. Das Interface stellt einen so genannten *Vertrag* zwischen Client und Server dar.
- *Netzspezifischer Code*, der die Codierung und Übertragung von Aufrufparametern und Rückgabewerten ermöglicht, wird ab Java SE 5 dynamisch zur Laufzeit generiert.
- Um für den ersten Aufruf einer entfernten Methode eine "Referenz" auf das entfernte Objekt, das diese Methode anbietet, zu erhalten, kann der Client einen so genannten *Namensdienst* (Registry) nutzen.
- RMI bietet Mechanismen sowohl für die Übertragung von Objekten als auch für die Übertragung und das Laden des Bytecodes der zugehörigen Klassen, falls diese lokal nicht vorhanden sind.
- RMI ist eine rein Java-basierte Lösung, d. h. Client und Server müssen in Java programmiert sein.

Mit Hilfe eines *Namensdienstes* können Dienste zentral veröffentlicht werden, sodass Clients diese über ein Netz finden und nutzen können. Namensdienste ordnen den Adressen von Ressourcen (hier entfernten Objekten) eindeutige Namen zu. Die Adresse einer Ressource enthält alle Informationen, die ein Client braucht, um mit der Ressource zu kommunizieren.

Um eine "Referenz" auf die gewünschte Ressource zu erhalten, übergibt der Client dem Namensdienst den Namen, unter dem die Ressource angemeldet ist. Als Ergebnis erhält er die Referenz, mit der nun eine Verbindung zur Ressource aufgebaut werden kann.

Abbildung 6-3 zeigt den allgemeinen Fall einer mit RMI realisierten Client/Server-Anwendung.

Abbildung 6-3: RMI-Anwendung

Der Client erhält über die *Registry* eine "Referenz" auf ein entferntes Objekt, das der Server vorher dort registriert hat. Für dieses Objekt ruft der Client dann eine

Methode auf. Ggf. kann das RMI-System auch einen Webserver nutzen, um Bytecode für Objekte vom Server zum Client bzw. umgekehrt zu übertragen.

Die Zusammenhänge, Begriffe und die erforderlichen Klassen und Methoden zur Entwicklung einer RMI-Anwendung werden nun im Folgenden schrittweise erläutert.

→ Projekt echo_v1

Das Beispiel zeigt die Implementierung eines *Echo-Dienstes*.

Remote Interface

Das *Remote Interface* definiert die Sicht des Clients auf das entfernte Objekt. Dieses Interface enthält die Methoden, die für dieses Objekt entfernt aufgerufen werden können.

Ein *Remote Interface* ist von dem Interface

`java.rmi.Remote`

abgeleitet.

`Remote` dient dazu, Interfaces zu kennzeichnen, deren Methoden entfernt aufgerufen werden sollen. `Remote` muss von allen Interfaces erweitert werden, die entfernte Methoden deklarieren.

`java.rmi.RemoteException` ist von `java.io.IOException` abgeleitet und ist Superklasse einer Reihe von Ausnahmen, die beim Aufruf einer entfernten Methode bei Netz- bzw. Protokollfehlern ausgelöst werden können. Jede Methode eines von `Remote` abgeleiteten Interfaces (*Remote Interface*) muss diese Klasse in der `throws`-Klausel aufführen.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Echo extends Remote {
   String getEcho(String s) throws RemoteException;
}
```

Remote Object

Jedes Objekt, dessen Klasse ein Remote Interface implementiert, ist ein so genanntes entferntes Objekt (*Remote Object*), d. h. es implementiert die vorgeschriebenen entfernt aufrufbaren Methoden.

Konstruktoren müssen `RemoteException` in der `throws`-Klausel enthalten. Methoden können in der Regel hierauf verzichten, falls sie selbst keine entfernten Methoden aufrufen.

Damit eine Verbindung zwischen Client und Server aufgenommen werden kann und Methoden des Objekts entfernt aufgerufen werden können, muss das entfernte Objekt *exportiert* und damit *remote-fähig* gemacht werden.

Dies geschieht z. B. durch Ableiten von der Klasse

`java.rmi.server.UnicastRemoteObject`

Bei Erzeugung des entfernten Objekts wird der Konstruktor dieser Superklasse aufgerufen, der das Objekt *exportiert*. Das exportierte Objekt kann nun über TCP/IP eingehende Nachrichten erhalten.

Implementierung des Remote Interface

Wir nutzen die folgende *Namenskonvention*:

Der Name der Klasse, die das Interface `Xxx` implementiert, ist `XxxImpl`.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class EchoImpl extends UnicastRemoteObject implements Echo {
   public EchoImpl() throws RemoteException {
   }

   public String getEcho(String s) {
      return s;
   }
}
```

Abbildung 6-4: Aufruf einer entfernten Methode

Stub und Skeleton

Die Codierung bzw. Decodierung der Aufrufparameter und Rückgabewerte von Methodenaufrufen und die Übermittlung der Daten zwischen Client und Server wird vom RMI-System und von generierten Klassen, beim Client *Stub* und beim Server *Skeleton* genannt, geregelt (siehe Abbildung 6-4).

Ein Stub-Objekt fungiert als *lokaler Stellvertreter* (Proxy) des entfernten Objekts. Ein Stub implementiert dasselbe Remote Interface, das auch die Klasse des entfernten Objekts implementiert. Das *Java Remote Method Protocol* (JRMP) wird vom Stub genutzt, um mit dem Server zu kommunizieren.

Remote Reference

Ein Client erhält Zugriff auf ein entferntes Objekt durch eine so genannte *entfernte Referenz* (*Remote Reference*). Diese wird beim Erzeugen des lokalen Stub-Objekts bereitgestellt. Das Stub-Objekt kapselt Informationen, die für den Zugriff auf das entfernte Objekt benötigt werden. Der Client ruft eine Methode des Stub-Objekts auf, die dann für den Methodenaufruf des entfernten Objekts auf der Serverseite sorgt (siehe Abbildung 6-5). Kurz gesagt wird eine entfernte Referenz durch eine Referenz auf ein entsprechendes Stub-Objekt realisiert.

Abbildung 6-5: Remote Reference

Die Klasse `EchoServer` (siehe unten) erzeugt ein entferntes Objekt vom Typ `EchoImpl` und registriert dieses bei einem Namensdienst (Registry).

Registry

Das vom JDK bereitgestellte Programm *rmiregistry* stellt einen einfachen Dienst zur Verfügung, der es dem Client erlaubt, eine *erste Referenz* auf ein entferntes Objekt als *Einstiegspunkt* zu erhalten. Weitere Referenzen auf andere entfernte Objekte können von hier aus dann anwendungsspezifisch, z. B. als Rückgabewerte von entfernten Methodenaufrufen, geliefert werden.

Jeder Eintrag in der Registry besteht aus einem *Namen* und einer *Objektreferenz*.

Der Name hat die Form eines URL:

```
//host:port/Dienstname
```

Bis auf `Dienstname` können alle Bestandteile entfallen. Der Rechnername ist in diesem Fall `localhost` und die Portnummer `1099`.

Soll für die Registry eine andere als die standardmäßig vorgesehene Portnummer `1099` benutzt werden, so muss sie als Aufrufparameter beim Start von *rmiregistry* angegeben werden.

Die Klasse `java.rmi.Naming` wird von Clients und Servern benutzt, um mit der Registry zu kommunizieren.

```
static void bind(String name, Remote obj)
       throws java.rmi.AlreadyBoundException,
       java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. `AlreadyBoundException` wird ausgelöst, wenn `name` bereits eingetragen ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

```
static void rebind(String name, Remote obj)
       throws java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. Besteht bereits ein Eintrag zu diesem Namen, so wird der bestehende Eintrag überschrieben. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

```
static void unbind(String name)
       throws java.rmi.NotBoundException, java.net.MalformedURLException,
       java.rmi.RemoteException
```

entfernt den Eintrag zu `name`. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

Diese drei `Naming`-Methoden können nur auf dem Rechner ausgeführt werden, auf dem auch der Namensdienst läuft. Registry und Server müssen also auf demselben Rechner gestartet werden.

```
import java.rmi.Naming;
import java.rmi.Remote;

public class EchoServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new EchoImpl();
      Naming.rebind("echo", remote);
      System.out.println("EchoServer gestartet ...");
   }
}
```

Das RMI-System sorgt dafür, dass der Server läuft, auch wenn alle Anweisungen der `main`-Methode ausgeführt sind.

```
import java.rmi.Naming;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      String host = args[0];

      Echo remote = (Echo) Naming.lookup("//" + host + "/echo");
      String received = remote.getEcho("Das ist ein Test.");
      System.out.println(received);
   }
}
```

Mittels der `Naming`-Methode `lookup` erhält der Client das Stub-Objekt zum entfernten Objekt, das unter dem Namen "`echo`" in der Registry eingetragen ist.

```
static Remote lookup(String name) throws java.rmi.NotBoundException,
       java.net.MalformedURLException, java.rmi.RemoteException
```

liefert die Referenz auf das Stub-Objekt für das unter `name` eingetragene entfernte Objekt. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist.

Test

Start des Verzeichnisdienstes:

```
set CLASSPATH=out/production/intro3
rmiregistry
```

Der Aufruf

```
rmiregistry -J-Djava.rmi.server.logCalls=true
```

bewirkt, dass der Kommunikationsfluss zwischen Client und Server protokolliert wird.

Start des Servers:

```
java -cp out/production/echo_v1 EchoServer
```

Aufruf des Clients:

```
java -cp out/production/echo_v1 EchoClient localhost
```

Verzeichnisdienst und Server müssen am Ende mit *Strg + C* abgebrochen werden.

Client und Server können natürlich auch auf unterschiedlichen Rechnern installiert und getestet werden.

6.2.2 Implementierungsvarianten

Die folgende Variante des Echo-Beispiels zeigt, dass die implementierende Klasse `EchoImpl` nicht unbedingt von `UnicastRemoteObject` abgeleitet sein muss.

Stattdessen wird im `EchoServer` die statische `UnicastRemoteObject`-Methode `export` mit einer expliziten Portnummer aufgerufen:

```
UnicastRemoteObject.exportObject(remote, 50000);
```

Mit dem Aufruf

```
UnicastRemoteObject.unexportObject(remote, true);
```

ist das Objekt nicht mehr *remote* aufrufbar, wobei der zweite Parameter angibt, ob das Objekt entfernt werden soll, auch wenn noch RMI-Aufrufe ausgeführt werden.

→ Projekt echo_v2

`Echo` und `EchoClient` sind wie im Projekt *echo_v1* realisiert.

```
public class EchoImpl implements Echo {
   public String getEcho(String s) {
      return s;
   }
}
```

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.server.UnicastRemoteObject;

public class EchoServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new EchoImpl();
      UnicastRemoteObject.exportObject(remote, 50000);
      Naming.rebind("echo", remote);
      System.out.println("EchoServer gestartet ...");
   }
}
```

LocateRegistry

Eine individuelle Registry kann aber auch innerhalb des Servers gestartet werden. Dazu steht die Klasse `LocateRegistry` im Paket `java.rmi.registry` zur Verfügung.

Die Methode

```
static Registry createRegistry(int port)
    throws java.rmi.RemoteException
```

erzeugt eine Registry die auf dem Port `port` Anfragen akzeptiert.

→ Projekt locateRegistry

Wir demonstrieren den Einsatz von `LocateRegistry` Gegenüber dem vorhergehenden Beispiel (Projekt *echo_v2*) werden `EchoServer` und `EchoClient` wie folgt angepasst:

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class EchoServer {
   public static void main(String args[]) throws Exception {
      int registryPort = 40000;

      LocateRegistry.createRegistry(registryPort);
      Remote remote = new EchoImpl();
      UnicastRemoteObject.exportObject(remote, 50000);
      Naming.rebind("//:" + registryPort + "/echo", remote);
      System.out.println("EchoServer gestartet ...");
   }
}
```

```
import java.rmi.Naming;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      String host = "localhost";
      int port = 40000;
      String text = "Das ist ein Test.";

      Echo remote = (Echo) Naming.lookup("//" + host + ":" + port + "/echo");
      String received = remote.getEcho(text);
      System.out.println(received);
   }
}
```

6.2.3 Dienstauskunft

Das folgende Programm gibt eine Liste aller in der Registry gebundenen Namen aus.

Die `Naming`-Methode

```
static String[] list(String name) throws java.rmi.RemoteException,
       java.net.MalformedURLException
```

liefert ein Array von Namen, die an entfernte Objekte gebunden sind. `name` spezifiziert die Registry in der Form `//host:port`.

→ Projekt list

```
import java.rmi.ConnectException;
import java.rmi.Naming;

public class ListRegistry {
   public static void main(String args[]) throws Exception {
      String[] registryNames = {
            "//localhost:1099",
            "//localhost:40000"
      };

      for (String registryName : registryNames) {
         try {
            String list[] = Naming.list(registryName);
            for (String name : list) {
               System.out.println(name);
            }
         } catch (ConnectException e) {
            System.err.println("Keine Verbindung zu: " + registryName);
         }
      }
   }
}
```

Die Ausgabe hat die Form:

```
//host:port/Dienstname
```

6.2.4 Aufrufparameter und Rückgabewerte

Die Aufrufparameter und der Rückgabewert einer entfernten Methode können von einem einfachen Datentyp, Referenzen auf "normale" lokale Objekte oder Referenzen auf entfernte Objekte sein.

Für *entfernte Methoden* gelten die folgenden Übertragungsregeln:

- *Werte von einfachem Datentyp* (z. B. `int`, `double`) werden wie bei lokalen Methodenaufrufen *by value* übertragen.
- *Lokale Objekte* werden serialisiert, übertragen und vom Server deserialisiert. Dafür sorgen Stub und Skeleton. Lokale Objekte werden, anders als beim lokalen Methodenaufruf, als Kopie *by value* übertragen. Diese Objekte müssen also das Interface `java.io.Serializable` implementieren.
- Exportierte *entfernte Objekte* werden *by reference* übertragen, d. h. es werden die Stub-Objekte, nicht Kopien der Originale übertragen.

Die folgende Tabelle fasst die Regeln zusammen:

Typ	lokale Methode	entfernte Methode
einfacher Typ	by value	by value
Objekt	by reference	by value (Serialisierung)
entferntes Objekt	by reference	by remote reference (Stub-Objekt)

6.3 Übertragung von Remote Objects

Beim Aufruf entfernter Methoden werden lokale Objekte als Kopie in serialisierter Form übertragen. Das folgende Beispiel zeigt eine entfernte Methode, die als Rückgabewert eine *entfernte Referenz* liefert, mit deren Hilfe eine entfernte Methode eines weiteren entfernten Objekts vom Client aufgerufen werden kann.

→ Projekt bank

Der RMI-Server soll Konten mit den Attributen Kontonummer (id), PIN und Saldo verwalten. Der Client kann ein neues Konto mit Angabe von Kontonummer und PIN anlegen oder ein bereits unter seiner Kontonummer eingerichtetes Konto öffnen. Für dieses Konto kann er dann einen Betrag einzahlen oder abheben sowie sich seinen Kontostand anzeigen lassen.

Für dieses Beispiel nutzen wir eine eigene Exception-Klasse.

KontoException

```
public class KontoException extends Exception {
   public KontoException() {
   }

   public KontoException(String msg) {
      super(msg);
   }
}
```

Remote Interface Konto

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Konto extends Remote {
   int getSaldo() throws RemoteException;

   void add(int betrag) throws RemoteException, KontoException;
}
```

Remote Interface KontoManager

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface KontoManager extends Remote {
   Konto getKonto(int id, int pin) throws RemoteException, KontoException;
}
```

KontoImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements Konto {
   private int pin;
   private int saldo;

   public KontoImpl(int pin) throws RemoteException {
      this.pin = pin;
   }

   public int getSaldo() {
      return saldo;
   }

   public void add(int betrag) throws KontoException {
      if (saldo + betrag < 0) {
         throw new KontoException("Das Konto kann nicht überzogen werden.");
      }
      saldo += betrag;
   }

   public int getPin() {
      return pin;
   }
}
```

KontoManagerImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;

public class KontoManagerImpl extends UnicastRemoteObject implements KontoManager {
   private Hashtable<Integer, KontoImpl> hashtable;

   public KontoManagerImpl() throws RemoteException {
      hashtable = new Hashtable<>();
   }

   public Konto getKonto(int id, int pin) throws RemoteException, KontoException {
      KontoImpl konto = hashtable.get(id);

      if (konto == null) {
         konto = new KontoImpl(pin);
         hashtable.put(id, konto);
```

```
            System.out.println("Konto " + id + " wurde eingerichtet.");
            return konto;
         } else {
            if (konto.getPin() == pin)
               return konto;
            else
               throw new KontoException("PIN ist ungültig.");
         }
      }
}
```

Die Konten werden in einer `Hashtable` unter der jeweiligen Kontonummer gespeichert. Ist bereits unter der angegebenen Kontonummer ein Konto vorhanden, so wird geprüft, ob die angegebene PIN gültig ist. Ist die PIN nicht korrekt, wird eine Ausnahme ausgelöst. Ist die Kontonummer neu, so wird ein neues Konto eingerichtet. Bei fehlerfreier Verarbeitung wird in beiden Fällen die entfernte Referenz auf ein `Konto`-Objekt zurückgeliefert.

BankServer

```
import java.rmi.Naming;
import java.rmi.Remote;

public class BankServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new KontoManagerImpl();
      Naming.rebind("bank", remote);
      System.out.println("BankServer gestartet ...");
   }
}
```

Das "Einstiegsobjekt" vom Typ `KontoManagerImpl` wird registriert.

BankClient

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.Scanner;

public class BankClient {
   public static void main(String args[]) {
      String host = args[0];
      int id = Integer.parseInt(args[1]);
      int pin = Integer.parseInt(args[2]);

      Konto konto = null;
      try {
         KontoManager manager = (KontoManager) Naming
            .lookup("//" + host + "/bank");
         konto = manager.getKonto(id, pin);
      } catch (KontoException e) {
         System.err.println(e.getMessage());
         return;
      } catch (Exception e) {
```

```
            System.err.println(e.getMessage());
            return;
        }

        Scanner scanner = new Scanner(System.in);
        String input;
        while (true) {
            try {
                System.out.println("get | <zahl> | q:");
                input = scanner.nextLine();
                if (input.length() == 0 || input.equals("q"))
                    break;
                if (input.equals("get")) {
                    System.out.println("Aktueller Kontostand: " + konto.getSaldo());
                } else {
                    int betrag = Integer.parseInt(input);
                    konto.add(betrag);
                }
            } catch (NumberFormatException e) {
            } catch (KontoException e) {
                System.err.println(e.getMessage());
            } catch (RemoteException e) {
                System.err.println(e.getMessage());
            }
        }
        scanner.close();
    }
}
```

Mit `lookup` erhält der Client Zugriff auf das entfernte `KontoManager`-Objekt. Hierfür ruft er die Methode `getKonto` auf und erhält als Rückgabewert eine entfernte Referenz auf ein `Konto`-Objekt, für das er dann die entfernten `Konto`-Methoden aufruft.

Registry und Server werden wie in Kapitel 6.2.1 aufgerufen.

Aufruf des Clients:

```
java -cp out/production/bank BankClient localhost 4711 12345

get | <zahl> | q:
1000
get | <zahl> | q:
get
Aktueller Kontostand: 1000
get | <zahl> | q:
-2000
Das Konto kann nicht überzogen werden.
get | <zahl> | q:
get
Aktueller Kontostand: 1000
get | <zahl> | q:
q
```

6.4 Mobile Agenten

Ein (serialisierbares) Objekt kann auch dann vom Client zum Server transportiert werden, wenn der Server den Bytecode der zugehörigen Klasse noch nicht zur Verfügung hat. Der Code muss dann ebenfalls zur Laufzeit ad hoc übertragen werden.

Wir nutzen hier diese Möglichkeit, um Methoden dieser Klasse auf dem Server *lokal* auszuführen. Derartige Nachrichten entsprechen von ihrem Wesen her einem *mobilen Agenten*, der im Auftrag eines Clients bestimmte Aufgaben auf einem anderen Rechner erledigt und danach zurückkehrt.

→ Projekt agent

Wir entwickeln einen Server, der beliebige Aufgaben vom Client entgegennimmt, diese ausführt und die Ergebnisse zurückliefert. Das ist z. B. dann hilfreich, wenn der Server auf einer sehr schnellen Maschine läuft und komplexe mathematische Berechnungen ausgeführt werden müssen.

Eine Aufgabe kann durch ein beliebiges Objekt repräsentiert werden, dessen Klasse das Interface `Agent` implementiert.

Client und Server sind in verschiedenen Unterprojekten realisiert.

Interface Agent

```
package agent;

import java.io.Serializable;

public interface Agent extends Serializable {
   void execute();
}
```

Remote Interface ServerAgent

```
package agent;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerAgent extends Remote {
   Agent execute(Agent agent) throws RemoteException;
}
```

ServerAgentImpl

Die entfernte Methode `execute` des Remote Interface `ServerAgent` initiiert den Transport des `Agent`-Objekts und ruft die `Agent`-Methode `execute` auf:

```
import agent.Agent;
import agent.ServerAgent;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ServerAgentImpl extends UnicastRemoteObject implements ServerAgent {
   public ServerAgentImpl() throws RemoteException {
   }

   public Agent execute(Agent agent) {
      agent.execute();
      return agent;
   }
}
```

Server

```
import java.rmi.Naming;
import java.rmi.Remote;

public class Server {
   public static void main(String args[]) throws Exception {
      Remote remote = new ServerAgentImpl();
      Naming.rebind("agent", remote);
      System.out.println("Server gestartet ...");
   }
}
```

DemoAgent

Die Klasse `DemoAgent` implementiert das Interface `Agent`. Die Methode `execute` berechnet die Summe der Zahlen von `1` bis zu einer vorgegebenen Zahl `n`. Diese Klasse soll später über das Netz zum Server transportiert werden.

```
import agent.Agent;

public class DemoAgent implements Agent {
   private int n;
   private int sum;

   public DemoAgent(int n) {
      this.n = n;
   }

   public void execute() {
      for (int i = 1; i <= n; i++) {
         sum += i;
      }
   }

   public int getResult() {
      return sum;
   }
}
```

Client

```
import agent.Agent;
import agent.ServerAgent;

import java.rmi.Naming;

public class Client {
   public static void main(String args[]) throws Exception {
      String host = args[0];

      ServerAgent remote = (ServerAgent) Naming.lookup("//" + host + "/agent");
      Agent demo = new DemoAgent(100);
      DemoAgent result = (DemoAgent) remote.execute(demo);
      System.out.println(result.getResult());
   }
}
```

Die Klasse `DemoAgent` im Unterprojekt *client* ist für den Server im Unterprojekt *server* über seinen `CLASSPATH` nicht erreichbar. Um den Bytecode von `DemoAgent` herunterladen zu können, nutzt der Server einen HTTP-Server.

Hierzu kann ein beliebiger Webserver eingesetzt werden. Wir nutzen einen Webserver, der im Kapitel 7 entwickelt wird. Hierzu muss die jar-Datei

```
libs/grizzly/grizzly-http-all-2.4.4.jar
```

in den Klassenpfad aufgenommen werden.

Für den Start des Clients ist der URL für den zu übertragenden Bytecode als Wert der Property `java.rmi.server.codebase` anzugeben.

Diese Information wird zum Server übertragen, sodass dieser dann die geeignete HTTP-Anfrage stellen kann.

Der Server muss einen *Security Manager* nutzen, da das Laden von Bytecode im Allgemeinen eine unsichere Aktivität ist.[1]

Die *Policy-Datei policy.txt* hat für unsere Zwecke den folgenden Inhalt:

```
grant {
  permission java.net.SocketPermission "*:1024-", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
};
```

Der HTTP-Server wird an die Portnummer `80` gebunden. Abbildung 6-6 zeigt die Konfiguration. Insbesondere geht aus der Abbildung hervor, welche Klassen und Interfaces dem Client, welche dem Server lokal zur Verfügung stehen.

Test

Aufruf der Registry:

```
set CLASSPATH=out/production/server
rmiregistry
```

1 Siehe auch `https://docs.oracle.com/en/java/javase/11/rmi/`

Aufruf des RMI-Servers (in einer Zeile):

```
java -Djava.rmi.server.useCodebaseOnly=false -Djava.security.manager -
Djava.security.policy=policy.txt -cp out/production/server Server
```

Aufruf des HTTP-Servers (in einer Zeile):

```
java -cp out/production/webserver;../../libs/grizzly/grizzly-http-all-
2.4.4.jar Webserver out/production/client 80 true
```

`out/production/client` ist das Root-Verzeichnis des Webservers (hier liegt die herunterzuladende Klasse `DemoAgent`, `80` ist die Portnummer und `true` gibt an, dass der Webserver HTTP-Anfragen protokollieren soll (siehe *access.log*).

Aufruf des Clients (in einer Zeile):

```
java -Djava.rmi.server.codebase=http://localhost:80/
-cp out/production/client Client localhost
```

Abbildung 6-6: Dynamisches Laden einer Klasse

6.5 Der Einsatz von Callback-Methoden

In diesem Kapitel sehen wir, dass ein Client auch zeitweise selbst Dienste anbieten kann. Der Server ruft eine entfernte Methode des Clients auf.

Polling oder Callback

Eine typische Anwendungssituation ist: Der Client will Ereignisse beobachten, die auf dem Server eintreten. Statt nun regelmäßig in bestimmten Abständen eine Anfrage an den Server zu stellen, ob das interessierende Ereignis eingetreten ist

oder nicht (*Polling*, *Pull-Prinzip*), lässt sich der Client vom Server über das Eintreten des Ereignisses informieren (*Callback*, *Push-Prinzip*).

Damit dieser *Callback-Mechanismus* funktioniert, muss der Client sich beim Server registrieren (der Server speichert eine *entfernte Referenz auf ein Remote-Objekt des Clients*). Dann kann der Server bei Eintreten des Ereignisses eine entfernte Methode des Clients aufrufen. Bei dieser Lösung fällt im Vergleich zum Polling unnötige Rechenzeit und Netzlast weg.

→ Projekt callback

Zur Veranschaulichung dieses Mechanismus entwickeln wir eine Anwendung (Client und Server), mit der Textnachrichten, die ein so genannter *Publisher* veröffentlicht, an interessierte Abonnenten (*Subscriber*) gesendet werden können. Der Server hat die Aufgabe, eine an ihn gerichtete Nachricht sofort an alle Abonnenten weiterzuleiten. Zu diesem Zweck muss er diese "kennen".

Textnachrichten werden in ein `Message`-Objekt verpackt, das zusätzlich den Zeitpunkt der Veröffentlichung enthält.

Das Paket `message.api`

```
package message.api;

import java.io.Serializable;
import java.time.LocalDateTime;

public class Message implements Serializable {
   private LocalDateTime timestamp;
   private String text;

   public LocalDateTime getTimestamp() {
      return timestamp;
   }

   public void setTimestamp(LocalDateTime timestamp) {
      this.timestamp = timestamp;
   }

   public void setText(String text) {
      this.text = text;
   }

   public String getText() {
      return text;
   }
}
```

Das *entfernte Objekt des Servers* (vom Typ `MessageManager`) hat drei Methoden:

`void setMessageListener(MessageListener listener)`
: meldet einen Subscriber an.

`void removeMessageListener(MessageListener listener)`
: meldet einen Subscriber ab.

`void send(Message msg)`
sendet eine Nachricht.

```
package message.api;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MessageManager extends Remote {
   void setMessageListener(MessageListener listener) throws RemoteException;

   void removeMessageListener(MessageListener listener) throws RemoteException;

   void send(Message message) throws RemoteException;
}
```

Das *entfernte Objekt des Clients* (vom Typ `MessageListener`) hat die Methode:

`void onMessage(Message msg)`
gibt die Nachricht aus.

```
package message.api;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MessageListener extends Remote {
   void onMessage(Message message) throws RemoteException;
}
```

Abbildung 6-7 zeigt den Zusammenhang.

Abbildung 6-7: Callback-Mechanismus

Wir implementieren zunächst den RMI-Server. Die Liste `listeners` vom Typ `CopyOnWriteArrayList` speichert die entfernten Referenzen auf die `MessageListener`-Objekte der Abonnenten. `CopyOnWriteArrayList` verhindert Ausnahmen vom Typ `ConcurrentModificationException` während der Iteration über die Ele-

mente der Liste. Die Methode `send` ruft für jede in der Liste gespeicherte Referenz die entfernte `MessageListener`-Methode `onMessage` auf. Wird hierbei (z. B. aufgrund des Abbruchs eines Subscribers) eine `RemoteException` ausgelöst, so wird die entsprechende Referenz aus der Liste entfernt.

Ein Thread gibt alle 5 Sekunden die Anzahl der Abonnenten am Bildschirm aus.

Modul *server*

```
import message.api.Message;
import message.api.MessageListener;
import message.api.MessageManager;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class MessageManagerImpl extends UnicastRemoteObject implements
       MessageManager {
   private List<MessageListener> listeners;

   public MessageManagerImpl() throws RemoteException {
      listeners = new CopyOnWriteArrayList<>();
      new ControlThread().start();
   }

   public void setMessageListener(MessageListener listener) {
      listeners.add(listener);
   }

   public void removeMessageListener(MessageListener listener) {
      listeners.remove(listener);
   }

   public synchronized void send(Message message) {
      for (MessageListener listener : listeners) {
         try {
            listener.onMessage(message);
         } catch (RemoteException e) {
            System.err.println(e.getMessage());
         }
      }
   }

   private class ControlThread extends Thread {
      public void run() {
         while (true) {
            try {
               Thread.sleep(5000);
            } catch (InterruptedException e) {
            }

            System.out.println("Anzahl Subscribers: " + listeners.size());
         }
      }
   }
}
```

```
import java.rmi.Naming;
import java.rmi.Remote;

public class MessageServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new MessageManagerImpl();
      Naming.rebind("message", remote);
      System.out.println("MessageServer gestartet ...");
   }
}
```

Nun implementieren wir *Publisher* und *Subscriber*.

Modul *pub*

```
import message.api.Message;
import message.api.MessageManager;

import java.rmi.Naming;
import java.time.LocalDateTime;

public class Publisher {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      String text = args[1];

      MessageManager manager = (MessageManager) Naming
         .lookup("//" + host + "/message");
      Message message = new Message();
      message.setTimestamp(LocalDateTime.now());
      message.setText(text);
      manager.send(message);
   }
}
```

Modul *sub*

```
import message.api.Message;
import message.api.MessageListener;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MessageListenerImpl extends UnicastRemoteObject implements
MessageListener {
   public MessageListenerImpl() throws RemoteException {
   }

   public void onMessage(Message message) {
      System.out.println(message.getTimestamp());
      System.out.println("\t" + message.getText());
   }
}


import message.api.MessageListener;
import message.api.MessageManager;
```

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Subscriber {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      int millis = Integer.parseInt(args[1]);

      final MessageManager manager = (MessageManager) Naming
         .lookup("//" + host + "/message");

      final MessageListener listener = new MessageListenerImpl();
      manager.setMessageListener(listener);

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            if (manager != null) {
               try {
                  manager.removeMessageListener(listener);
               } catch (RemoteException e) {
                  System.err.println(e);
               }
            }
         }
      });

      try {
         Thread.sleep(millis);
      } catch (InterruptedException e) {
      }

      System.exit(0);
   }
}
```

Der Subscriber wird nach einer vorgegebenen Anzahl Millisekunden beendet.

Test

Aufruf der Registry:

```
set CLASSPATH=out/production/server
rmiregistry
```

Aufruf des Servers:

```
java -cp out/production/server MessageServer
```

Aufruf des Subscribers:

```
java -cp out/production/sub Subscriber localhost 60000
```

Aufruf eines Publishers:

```
java -cp out/production/pub Publisher localhost "Das ist ein Test"
```

6.6 Aufgaben

1. Programmieren Sie einen RMI-Dienst, dessen entfernte Methode
   ```
   String getDaytime()
   ```
   die aktuelle Systemzeit des Servers liefert.

2. Erstellen Sie für den RMI-Server aus Kapitel 6.4 einen neuen Agenten, der zu einer vorgegebenen Zahl `n` die Fakultät `n!` ermittelt. Zur Berechnung soll die Klasse `java.math.BigInteger` genutzt werden.

3. Entwickeln Sie eine auf RMI basierende Chat-Anwendung mit Hilfe des Callback-Mechanismus.

   Remote-Methoden des Interface `ChatManager`:
   ```
   void login(String user, Callback callback) throws RemoteException;
   void logout(String user, Callback callback) throws RemoteException;
   void notifyClients(String user, String message)
          throws RemoteException;
   ```
   Remote-Methoden des Interface `Callback`:
   ```
   void showLogin(String user) throws RemoteException;
   void showLogout(String user) throws RemoteException;
   void showMessage(String user, String message) throws RemoteException;
   ```
   Angemeldete Clients sollen in einer Liste vom Typ
   ```
   CopyOnWriteArrayList<Callback>
   ```
   beim Server gespeichert werden.

   Der RMI-Client soll eine einfache Kommandozeilen-gesteuerte Anwendung sein.

4. Erstellen Sie einen universellen RMI-Server (`MultiServer`), der mehrere Dienste gleichzeitig anbieten kann. Zur Laufzeit sollen bestehende Dienste beendet und neue Dienste hinzugefügt werden können. Ebenso soll der Server kontrolliert beendet werden können.

   Der Server soll das Remote-Interface `MultiServerManager` mit den folgenden Methoden implementieren:
   ```
   void shutdown()
   void reconfigure()
   ```
   `shutdown` entfernt in einem Thread alle Dienste (`Naming`-Methode `unbind`) und beendet den Server mit der Anweisung `System.exit(0)`.

   `reconfigure` entfernt alle Dienste, liest die zu bindenden Dienste aus einer `Properties`-Datei ein und bindet diese (`Naming`-Methode `bind`).

   Die neu zu bindenden Dienste sind in einer Textdatei in folgender Form gespeichert (Beispiel):
   ```
   echo      EchoImpl
   daytime   DaytimeImpl
   ```

Jede Zeile entspricht einem Dienst. Sie enthält den Dienstnamen und den Namen der Klasse, die den Dienst implementiert.

Für jeden Eintrag soll mittels *Reflection* (Nutzung der `Class`-Methoden `forName` und `getDeclaredConstructor().newInstance()`) ein neues entferntes Objekt erzeugt und bei der Registry angemeldet werden.

Des Weiteren ist ein Client `MultiServerManagerClient` zu erstellen, der den Aufruf der Methoden `shutdown` und `reconfigure` ermöglicht.

5. Realisieren Sie eine Mail-Anwendung auf Basis von RMI, die Mails senden und empfangen kann.

   Erstellen Sie hierzu die Klasse `Mail` mit den Attributen `sender`, `recipient` und `message`. Benutzer können sich anmelden und abmelden. Die Klasse `Credentials` enthält die Attribute `user` und `password`.

   Das Remote-Interface `MailService` enthält die folgenden Methoden:

```
void register(Credentials credentials) throws RemoteException,
       MailException;
void unregister(Credentials credentials) throws RemoteException,
       MailException;
void sendMail(Credentials credentials, Mail mail) throws
       RemoteException, MailException;
List<Mail> getMails(Credentials credentials) throws RemoteException,
       MailException;
```

   Funktionalität:

   Zur Speicherung von User (Schlüssel ist der Username) und Mails (Schlüssel ist der Empfängername) enthält die implementierende Klasse `MailServiceImpl` zwei Tabellen vom Typ
   `Map<String, Credentials>` bzw. `Map<String, List<Mail>>`.

   `register`
   Ist der User noch nicht eingetragen, wird er gespeichert, ansonsten wird eine Ausnahme ausgelöst.

   `unregister`
   Ein User kann aus der Liste nur gelöscht werden, wenn User und Passwort gültig sind.

   `sendMail`
   Die Mail eines Users wird gespeichert, wenn User und Empfänger bereits registriert sind und der User mit dem Absender übereinstimmt. Die Mail wird der Empfängerliste hinzugefügt.

   `getMails`
   Ein bereits registrierter User kann nur die für ihn bestimmten Mails abfragen. Die Mail-Liste wird anschließend gelöscht.

Bei Beendigung des Servers sollen die beide oben erwähnten Tabellen serialisiert gespeichert werden. Hierzu ist ein *Shutdown hook* zu nutzen:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
   public void run() {
      try {
         service.store();
      } catch (IOException e) {
      }
   }
});
```

Beim Starten des Servers sollen die beiden Tabellen wieder durch Deserialisierung zur Verfügung gestellt werden.

Des Weiteren soll für die Bedienung ein Kommandozeilen-gesteuerter RMI-Client erstellt werden mit den Kommandos: `register`, `unregister`, `get`, `send` und `exit`.

7 Kommunikation über HTTP

Ziel dieses Kapitels ist es, eine Einführung in das *Hypertext Transfer Protocol* (*HTTP*) zu geben und einen einfachen Webserver zu entwickeln.

7.1 Die Protokolle HTTP/1.0 und HTTP/1.1

HTTP ist ein Protokoll der Anwendungsschicht im TCP/IP-Schichtenmodell und regelt insbesondere, wie ein Webbrowser mit einem Webserver im Internet kommuniziert. HTTP verwendet auf der Transportschicht TCP.

Damit ein Webbrowser eine Webseite im Web abrufen kann, muss er sie zunächst adressieren. Ein *Uniform Resource Locator* (*URL*) ist eine standardisierte Adresse, mit der eine beliebige Ressource (z. B. eine HTML-Seite, ein GIF-Bild, eine PDF-Datei, ein Programm) lokalisiert werden kann.

So lokalisiert z. B. `http://www.hs-niederrhein.de` die Website der Hochschule Niederrhein.

Der URL hat im Allgemeinen den folgenden Aufbau:[1]

```
protocol://server[:port][/resource]
```

Hierbei sind die eingeklammerten Teile optional. Die Angaben bedeuten:

`protocol`	Protokollname, hier: `http`
`server`	Host-Name oder IP-Adresse des Servers
`port`	Portnummer, unter der der Server läuft; die standardmäßige Portnummer für einen HTTP-Server ist 80 und muss nicht angegeben werden.
`resource`	Bezeichnung der Ressource, z. B. Pfad- und Dateiname

Pfad- und Dateiname müssen keine reale Datei im Dateisystem bezeichnen. Sie können auf der Serverseite als logischer Name zur eindeutigen Bezeichnung einer Ressource interpretiert werden.

Die Interaktion zwischen HTTP-Client und HTTP-Server für eine Anfrage umfasst im Normalfall die folgenden Schritte:

1. Der Server wartet auf eine eingehende *HTTP-Anfrage*.
2. Der Client erzeugt einen URL `http://...`
3. Der Client versucht, eine TCP-Verbindung zum Server aufzubauen.
4. Der Server akzeptiert den Verbindungswunsch.

1 Angaben zum Account (Username, Passwort), wie sie z. B. beim Protokoll FTP verwendet werden, werden hier nicht betrachtet.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_7

5. Der Client sendet eine Nachricht (*HTTP-Anfrage*) an den Server und fordert die Ressource mit dem spezifizierten URL an.
6. Der Server verarbeitet die Anfrage (z. B. Ausführung einer Datenbankabfrage und Generierung einer HTML-Seite mit dem Abfrageergebnis).
7. Der Server sendet eine Rückantwort (*HTTP-Antwort*) an den Client, die die angeforderte Ressource oder eine Fehlermeldung enthält.
8. Der Client verarbeitet die Antwort.
9. Der Client und/oder der Server schließen die TCP-Verbindung.

Der Server hat keine Kenntnis über vorangegangene Anfragen desselben Clients. Jede Anfrage wird unabhängig von vorhergehenden Anfragen bearbeitet. HTTP ist also ein *zustandsloses Protokoll*.

Für jede HTTP-Anfrage wird bei HTTP/1.0 in der Regel eine neue TCP-Verbindung aufgebaut. Enthält z. B. eine angeforderte HTML-Seite mehrere Grafiken, so muss jede dieser Grafiken separat angefordert werden (jeweils mit Verbindungsaufbau und -abbau).

Abbildung 7-1: Eine HTTP-Transaktion

HTTP/1.0 ist im RFC 1945 der IETF spezifiziert.[2]

Die Version *HTTP/1.1* unterstützt so genannte *persistente Verbindungen*. Während die TCP-Verbindung steht, können *mehrere* HTTP-Anfragen über diese Verbindung durchgeführt werden. Die Verbindung kann vom Client oder Server abgebaut werden. HTTP/1.1 ist im RFC 2616 der IETF spezifiziert sowie in den RFCs 7230 - 7235 überarbeitet.[3]

2 https://tools.ietf.org/html/rfc1945

3 https://tools.ietf.org/html/rfc2616

Die neue Version HTTP/2 beschleunigt und optimiert die Übertragung durch Zusammenfassung mehrerer Anfragen, Datenkompression und binäre Übertragung.[4]

→ Projekt request

Das folgende Programm zeigt den Inhalt der Nachricht (HTTP-Anfrage), die ein Webbrowser an den Webserver schickt. Das verwendete Format der HTTP-Antwort wird später erklärt.

```
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

public class ShowRequest {
   public static void main(String[] args) {
      try (ServerSocket server = new ServerSocket(50000)) {
         while (true) {
            try (Socket client = server.accept();
                BufferedReader in = new BufferedReader(
                  new InputStreamReader(client.getInputStream()));
                BufferedOutputStream out = new BufferedOutputStream(
                  client.getOutputStream())) {

               out.write("HTTP/1.1 200 OK\r\n".getBytes());
               out.write("Content-Type: text/plain\r\n\r\n".getBytes());

               int n = 0;
               int length = 0;
               String line;
               while (true) {
                  line = in.readLine();
                  out.write((++n + "\t" + line + "\n").getBytes());

                  if (line == null || line.length() == 0)
                       break;

                  if (line.startsWith("Content-Length")) {
                       length = Integer.parseInt(line.substring(
                               line.indexOf(':') + 1).trim());
                  }
               }

               out.write((char) '\t');
               for (int i = 0; i < length; i++) {
                  int c = in.read();
                  out.write((char) c);
               }
            }
         }
```

4 https://tools.ietf.org/html/rfc7540
https://tools.ietf.org/html/rfc7541

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

Aufruf des Programms:

```
java -cp out/production/request ShowRequest
```

Anforderung einer fiktiven HTML-Seite im Webbrowser (hier: Firefox):

```
http://localhost:50000/abc/xyz.html
```

Ausgabe im Browser:

```
1  GET /abc/xyz.html HTTP/1.1
2  Host: localhost:50000
3  Connection: keep-alive
4  Upgrade-Insecure-Requests: 1
5  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...
6  Accept: text/html,application/xhtml+xml,application/xml;...
7  Accept-Encoding: gzip, deflate, br
8  Accept-Language: de,de-DE;q=0.9,en;q=0.8,en-US;q=0.7,lb;q=0.6
9
```

Eine *HTTP-Anfrage* hat den folgenden Aufbau:

- *Kopfzeile*
  Sie enthält die *HTTP-Methode* (im Beispiel GET), den Namen der angeforderten Ressource ohne Protokoll und Domain-Namen (im Beispiel `/abc/xyz.html`) und die verwendete Protokollversion (im Beispiel HTTP/1.1).
- *Anfrageparameter (optional)*
  Anfrageparameter liefern dem Server zusätzliche Informationen über den Client und seine Anfrage. Jeder Parameter benutzt eine eigene Zeile und besteht aus dem Namen, einem Doppelpunkt und dem Wert.
- eine *Leerzeile*
- *Nutzdatenteil (optional)*
  Hier stehen z. B. die in einem Formular eingetragenen Daten bei Anwendung der HTTP-Methode POST.

Kopfzeile, Anfrageparameter und Leerzeile enden jeweils mit *Carriage Return* und *Linefeed*: `\r\n`.

Im einfachsten Fall reicht eine Anfrage der Form

```
GET /index.html HTTP/1.1
```

aus, um bereits von einem Webserver verstanden zu werden.

Abbildung 7-2: Aufbau der HTTP-Anfrage

Die HTTP-Methode spezifiziert die vom Server durchzuführende Aktion.

Wichtige HTTP-Methoden sind:

- `GET`
  Diese Methode fordert eine Ressource an.
- `POST`
  Diese Methode überträgt Benutzerdaten an den Server.
- `PUT`
  Diese Methode wird verwendet, um eine Ressource auf dem Server zu erstellen bzw. zu ändern.
- `DELETE`
  Diese Methode wird verwendet, um eine Ressource auf dem Server zu löschen.

Anfrageparameter können in beliebiger Reihenfolge angegeben werden. Groß- und Kleinschreibung wird ignoriert.

Die gebräuchlichsten Anfrageparameter sind (siehe obiges Testbeispiel):

- `Host`
  Rechnername und optionale Portnummer des Servers
- `User-Agent`
  Kenndaten über den HTTP-Client
- `Connection`
  Dieser Parameter wird benutzt, um eine persistente TCP-Verbindung anzufordern bzw. zu schließen.
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil
- `Accept-Language`
  Dieser Parameter gibt die vom Client bevorzugte Sprache an. Der Server kann dann z. B. eine HTML-Seite, die in mehreren Sprachvarianten vorliegt, in der vom Client gewünschten Sprache senden.

- `Accept-Encoding`
  Mit diesem Parameter gibt der Client an, welche Komprimierungsalgorithmen er versteht. Der Server kann z. B. große Dateien komprimieren, um die Übertragungszeit zu minimieren.
- `Accept`
  Dieser Parameter gibt die vom Client akzeptierten Medientypen an. Die gültigen Parameterwerte sind durch den MIME-Standard definiert.

MIME steht für den Standard *Multipurpose Internet Mail Extension*, der ursprünglich für E-Mails entworfen wurde.

MIME-Formatangaben werden von HTTP-Clients und HTTP-Servern benutzt. Clients nutzen sie, um dem Server mitzuteilen, welche Medientypen sie handhaben können. Server nutzen sie, um den Client über den Inhaltstyp der gesendeten Ressource zu informieren.

Die MIME-Formatangabe besteht aus einer Typ- und einer Subtypangabe:

```
typ/subtyp
```

Der MIME-Standard ist im RFC 1521 der IETF spezifiziert.[5]

Beispiele für Medientypen:

Typ/Subtyp	Beschreibung und übliche Erweiterung
`text/html`	HTML-Datei (*.htm, *.html)
`text/plain`	ASCII-Text (*.txt)
`text/xml`	XML-Datei (*.xml)
`image/gif`	GIF-Bild (*.gif)
`image/jpeg`	JPEG-Bild (*.jpeg, *.jpg)
`image/png`	PNG-Bild (*.png)
`application/pdf`	PDF-Datei (*.pdf)
`application/octet-stream`	Binärdaten (*.bin, *.exe)
`application/zip`	ZIP-Datei (*.zip)

Für nicht standardisierte Subtypen wird das Präfix `x-` benutzt, z. B. bezeichnet `audio/x-wav` Audio-Dateien `*.wav`.

Mit der HTTP-Methode `POST` können Benutzerdaten zum Server geschickt werden.

Das folgende Beispiel zeigt ein HTML-Formular, dessen Daten durch Betätigen des Buttons "Senden" zum Server gesendet werden. Die HTML-Seite *post.html* soll aus der lokalen Ablage vom Browser geladen werden.

5 `https://tools.ietf.org/html/rfc1521`

HTML-Code des Formulars:

```
<html>
<head>
    <meta charset="utf-8">
    <title>POST</title>
</head>
<body>
<form action="http://localhost:50000/xxx" method="POST">
<pre>
Artikelnummer: <input type="text" name="nr" size="5">
Bezeichnung:   <input type="text" name="bez" size="30">
Preis:         <input type="text" name="preis" size="10">
</pre>
Beschreibung:<br>
<textarea name="beschr" cols="60" rows="5"></textarea>
<p>
<input type="submit" value="Senden">
<input type="reset" value="Zurücksetzen">
</form>
</body>
</html>
```

Abbildung 7-3: Ein Eingabeformular

Ausgabe im Browser:

```
1  POST /xxx HTTP/1.1
2  Host: localhost:50000
...
8  Content-Type: application/x-www-form-urlencoded
9  Content-Length: 120
10 Connection: keep-alive
11
   nr=4711&bez=Akku-Handstaubsauger&preis=15.99&beschr=3+Zellen.+Kabellos.+Wandhalt
   erung.+Fugend%C3%BCse+und+B%C3%BCrste.
```

Die im Formular eingetragenen Daten werden im *URL-codierten Format* übertragen (`application/x-www-form-urlencoded`).

Die im Formular definierten Variablennamen (im Beispiel: `nr`, `bez`, `preis`, `beschr`) sind mit den vom Benutzer eingegebenen Werten verknüpft.

- Variable und Wert werden jeweils durch ein Gleichheitszeichen voneinander getrennt.
- Die einzelnen Variable/Wert-Paare sind durch das Zeichen `&` getrennt.
- Alle Zeichen außer `a-z`, `A-Z`, `0-9`, `.`, `-`, `*` werden zuerst in Bytes nach einem Codierungsschema (z. B. `ISO-8859-1` oder `UTF-8`) konvertiert.
- Jedes Byte wird dann durch ein Prozentzeichen, gefolgt vom Hexadezimalwert des Bytes dargestellt.
- Leerzeichen werden durch `+` ersetzt.

Formulare können auch die `GET`-Methode nutzen, um Daten zu übertragen. Die URL-codierten Daten werden nach einem Fragezeichen `?` an den URL angehängt. Im HTML-Code des obigen Formulars muss nur `POST` durch `GET` ersetzt werden. Die Kopfzeile der HTTP-Anfrage hat dann das folgende Aussehen:

```
GET /xxx?nr=4711&bez=Akku-Handstaubsauger&preis=15.99&beschr=3+Zellen.+Kab
ellos.+Wandhalterung.+Fugend%C3%BCse+und+B%C3%BCrste. HTTP/1.1
```

Die so codierte Anfragezeichenkette nach dem Fragezeichen wird auch als *Query String* bezeichnet.

Eine *HTTP-Antwort* hat den folgenden Aufbau:

- *Kopfzeile*
  Sie besteht aus der Protokollversion, dem Status-Code und einer optionalen Status-Meldung.
- *Antwortparameter (optional)*
  Antwortparameter liefern dem Client zusätzliche Informationen über den Server und die Antwort.
- eine *Leerzeile*
- *Nutzdatenteil (optional)*
  Dieser enthält die angeforderte Ressource.

Beispiel:

```
HTTP/1.1 200 OK
Date: Fri, 25 Jan 2019 16:19:10 GMT
Server: Apache-Coyote/1.1
...
Content-Length: 3205
Content-Type: text/html

<html>...</html>
```

Abbildung 7-4: Aufbau der HTTP-Antwort

Der *Status-Code*[6] teilt dem Client mit, wie die gewünschte Aktion vom Server ausgeführt wurde.

Die Status-Codes sind wie folgt gruppiert:

100 - 199	Informative Meldungen
200 - 299	Die Anfrage war erfolgreich
300 - 399	Die Anfrage wurde weitergeleitet
400 - 499	Die Anfrage war fehlerhaft
500 - 599	Server-Fehler

Beispiele:

200	OK
201	Created
204	No Content
400	Bad Request
404	Not Found
500	Internal Server Error
501	Not Implemented

Gebräuchliche Antwortparameter sind:

- `Date`
  Aktuelles Datum des Servers zum Zeitpunkt der Beantwortung der Anfrage. Ein Beispiel:
  `Date: Fri, 25 Jan 2019 16:19:10 GMT`

6 https://tools.ietf.org/html/rfc2616#section-10

- `Server`
  Kenndaten über den HTTP-Server
- `Content-Type`
  MIME-Format der Nutzdaten
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil. Werden Webseiten dynamisch erzeugt, ist die Länge oft nicht bekannt, weshalb dieser Parameter dann weggelassen wird.
- `Last-Modified`
  Zeitpunkt der letzten Änderung der Ressource. Ein Beispiel:
  `Last-Modified: Fri, 25 Jan 2019 16:19:10 GMT`
- `Connection`
  Legt für diese Verbindung gewünschte Optionen fest. Ein HTTP/1.1-Server, der persistente Verbindungen nicht unterstützt, muss den Header "`Connection: close`" verwenden.

Gängige Browser (z. B. Firefox, Google Chrome) haben Web-Konsolen, mit denen der Netzwerkverkehr (Anfragen, Antworten) analysiert werden kann.

7.2 Ein einfacher Webserver

Im Folgenden entwickeln wir einen einfachen Webserver, der nur die HTTP-Methode GET versteht, Query Strings aber nicht verarbeitet.

→ Projekt webserver1

Der Konstruktor der Klasse `SimpleWebserver` wird mit drei Parametern aufgerufen:

- die Wurzel des Webverzeichnisses, das alle abrufbaren Ressourcen enthält,
- die Portnummer und
- ein boolescher Wert, der angibt, ob Zugriffe protokolliert werden sollen.

`SimpleWebserver`-Methoden:

`void start()`
startet den Webserver.

`void shutdown()`
beendet den Webserver.

```
import webserver.SimpleWebserver;

import java.io.File;
import java.io.IOException;
```

```
public class MyWebserver {
   public static void main(String[] args) {
      String root = args[0];
      int port = Integer.parseInt(args[1]);
      boolean log = Boolean.parseBoolean(args[2]);

      try {
         SimpleWebserver server = new SimpleWebserver(new File(root), port, log);
         server.start();
         System.out.println("MyWebserver gestartet ...");
         System.out.println("ENTER oder Strg + C stoppt den Server.");
         System.in.read();
         System.exit(0);
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}


package webserver;

import java.io.File;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class SimpleWebserver extends Thread {
   private static final String LOG_FILE = "access.log";
   private static final int SOCKET_TIMEOUT = 30000;
   private File root;
   private boolean log;
   private ServerSocket serverSocket;

   public SimpleWebserver(File root, int port, boolean log) throws IOException {
      this.root = root.getCanonicalFile();
      this.log = log;

      if (!root.isDirectory()) {
         throw new IOException("No directory");
      }

      if (log)
         AccessLog.initializeLogger(new File(LOG_FILE));

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            shutdown();
         }
      });

      serverSocket = new ServerSocket(port);
   }

   @Override
   public void run() {
      while (true) {
         try {
```

```
            Socket socket;
            try {
               socket = serverSocket.accept();
            } catch (SocketException e) {
               break;
            }

            socket.setSoTimeout(SOCKET_TIMEOUT);

            Request request = new Request(socket, root, log);
            request.start();
         } catch (IOException e) {
            System.err.println(e);
         }
      }
   }

   public void shutdown() {
      try {
         if (serverSocket != null) {
            serverSocket.close();
         }

         if (AccessLog.logger != null) {
            AccessLog.logger.close();
         }
      } catch (IOException e) {
      }
   }
}
```

Die Bearbeitung einer HTTP-Anfrage erfolgt in einem Thread (siehe Klasse `Request`).

Ein *Shutdown-Hook* sorgt für das ordnungsgemäße Beenden des Servers, auch wenn das Programm mit *Strg + C* abgebrochen wird. In Eclipse und IntelliJ IDEA wird der Shutdown-Hook nicht gestartet, wenn das Programm über den Stop-Button beendet wird. Dieser Button "killt" den JVM-Prozess direkt, sodass der Shutdown-Hook nicht mehr ausgeführt werden kann.

```
package webserver;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.net.URLConnection;
import java.net.URLDecoder;
import java.text.SimpleDateFormat;
import java.util.Calendar;
```

```
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class Request extends Thread {
   private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
         "yyyy-MM-dd HH:mm:ss");
   private File root;
   private boolean log;
   private Socket socket;

   public Request(Socket socket, File root, boolean log) {
      this.socket = socket;
      this.root = root;
      this.log = log;
   }

   @Override
   public void run() {
      try (BufferedReader in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
            BufferedOutputStream out = new BufferedOutputStream(
                  socket.getOutputStream())) {

         String request = in.readLine();
         if (request == null || request.trim().length() == 0)
            return;

         if (log) {
            AccessLog.logger.log("[" + simpleDateFormat.format(new Date())
                  + " " + socket.getInetAddress().getHostAddress() + ":"
                  + socket.getPort() + "] " + request);
         }

         if (!request.startsWith("GET")) {
            sendError(out, Status.NOT_IMPLEMENTED,
                  "Nur die HTTP-Methode GET ist implementiert.");
            return;
         }

         // Query Strings werden ignoriert.
         String path = getPath(request);
         int idx = path.indexOf("?");
         if (idx >= 0) {
            path = path.substring(0, idx);
         }

         File file = new File(root, URLDecoder.decode(path, "UTF-8"))
               .getCanonicalFile();

         if (file.isDirectory()) {
            File indexFile = new File(file, "index.html");
            if (indexFile.exists() && !indexFile.isDirectory()) {
               file = indexFile;
            } else {
               sendError(out, Status.FORBIDDEN, "");
               return;
            }
         }
```

```
        // Zugriff außerhalb von root ist nicht erlaubt.
        if (!file.getCanonicalPath().startsWith(root.getCanonicalPath())) {
           sendError(out, Status.FORBIDDEN, "");
           return;
        }

        if (!file.exists()) {
           sendError(out, Status.NOT_FOUND, "");
           return;
        }

        try (InputStream is = new BufferedInputStream(new FileInputStream(
              file))) {

           String contentType = URLConnection.getFileNameMap()
                 .getContentTypeFor(file.getName());

           if (contentType == null) {
              contentType = "application/octet-stream";
           }

           sendHeader(out, Status.OK, contentType, file.length(),
              file.lastModified());

           byte[] buffer = new byte[8192];
           int bytesRead;
           while ((bytesRead = is.read(buffer)) != -1) {
              out.write(buffer, 0, bytesRead);
           }
        }

        out.flush();
     } catch (SocketTimeoutException e) {
     } catch (IOException e) {
        System.err.println(e);
     }
  }

  private void sendError(BufferedOutputStream out, Status status, String reason)
        throws IOException {

     String msg = status.getMessage() + ": " + reason;
     sendHeader(out, status, "text/html", msg.length(),
           System.currentTimeMillis());
     out.write(msg.getBytes());
     out.flush();
  }

  private void sendHeader(BufferedOutputStream out, Status status,
        String contentType, long length, long time) throws IOException {

     String header =
           "HTTP/1.1 " + status.getCode() + " " + status.getMessage()
           + "\r\nDate: " + getTime(System.currentTimeMillis())
           + "\r\nServer: SimpleWebserver"
           + "\r\nContent-Type: " + contentType
           + "\r\nContent-Length: " + length
           + "\r\nLast-Modified: " + getTime(time)
```

```
            + "\r\nConnection: close"
            + "\r\n\r\n";

      out.write(header.getBytes());
   }

   private static String getPath(String request) {
      return request.substring(4, request.length() - 9);
   }

   private static String getTime(long time) {
      Calendar calendar = Calendar.getInstance();
      calendar.setTimeInMillis(time);
      SimpleDateFormat dateFormat = new SimpleDateFormat(
            "EEE, dd MMM yyyy HH:mm:ss 'GMT'", Locale.US);
      dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
      return dateFormat.format(calendar.getTime());
   }
}
```

Die einzelne HTTP-Anfrage wird in der `run`-Methode der von Thread abgeleiteten Klasse `Request` bearbeitet. Die Kopfzeile der Anfrage wird gelesen und analysiert. Nur Anfragen mit der HTTP-Methode GET werden berücksichtigt.

Die Methode `getPath` extrahiert den Pfad-/Dateinamen `xxx` aus der Kopfzeile:

```
GET /xxx HTTP/1.1
```

Ein URL-codierter Pfadname (siehe Kapitel 7.1) wird mit Hilfe der Klassenmethode `decode` der Klasse `java.net.URLDecoder` decodiert und dieser dann mit dem Wurzelverzeichnis verbunden.

Handelt es sich bei dieser File-Ressource um ein Verzeichnis und enthält dieses die Datei `index.html`, so wird der Inhalt von `index.html` gesendet.

Der Versuch, mit Hilfe von Angaben wie `../` innerhalb des Pfadnamens auf Dateien außerhalb des Wurzelverzeichnisses zuzugreifen, führt zu einer Fehlermeldung.

Zur Bestimmung des Content-Type in der HTTP-Antwort wird die Methode

```
URLConnection.getFileNameMap().getContentTypeFor(...)
```

aufgerufen.

Die abstrakte Klasse `java.net.URLConnection` ist die Superklasse derjenigen Klassen, die eine Netzverbindung zu einer durch den URL adressierten Ressource repräsentieren.

`static FileNameMap getFileNameMap()`
: versucht eine benutzerspezifische Tabelle mit MIME-Typen über die System-Property `-Dcontent.types.user.table=<dateiname>`, zu laden. Gelingt dies nicht wird eine systeminterne Tabelle verwendet.

Aufbau der Tabelle (Beispiel):

```
text/plain: \
    description=Plain Text;\
    file_extensions=.text,.txt,.java,;\
    icon=text;\
    action=browser
```

Das Begleitmaterial enthält eine Beispieldatei.

Das Interface `java.net.FileNameMap` enthält die Methode

```
String getContentTypeFor(String fileName)
```

Sie liefert den MIME-Typ zu einem Dateinamen.

`Status` vom Typ `enum` enthält die hier benötigten Fehlercodes.

```
package webserver;

public enum Status {
   OK(200, "OK"), FORBIDDEN(403, "Forbidden"), NOT_FOUND(404, "Not Found"),
         NOT_IMPLEMENTED(501, "Not Implemented");

   private int code;
   private String message;

   private Status(int code, String message) {
      this.code = code;
      this.message = message;
   }

   public int getCode() {
      return code;
   }

   public String getMessage() {
      return message;
   }
}
```

```
package webserver;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class AccessLog {
   public static AccessLog logger;
   private BufferedWriter logfileWriter;

   private AccessLog(File logfile) throws IOException {
      logfileWriter = new BufferedWriter(new FileWriter(logfile, true));
   }

   public static void initializeLogger(File logfile) throws IOException {
      if (logger == null)
```

```
            logger = new AccessLog(logfile);
    }

    public void log(String info) throws IOException {
        synchronized (logfileWriter) {
            logfileWriter.write(info);
            logfileWriter.newLine();
            logfileWriter.flush();

            System.out.println(info);
        }
    }

    public void close() {
        try {
            logfileWriter.flush();
            logfileWriter.close();
        } catch (IOException e) {
        }
    }
}
```

Aufruf des Servers (in einer Zeile einzugeben):

```
java -cp out/production/webserver1 MyWebserver web 80 true
```

oder

```
java -Dcontent.types.user.table=content-types.properties
-cp out/production/webserver1 MyWebserver web 80 true
```

Alternative Implementierung

→ Projekt webserver2

Wir verwenden das *Grizzly HTTP Server Framework*, um mit Hilfe dieses API einen Webserver mit wenigen Zeilen zu implementieren.

Dazu muss die jar-Datei `libs/grizzly/grizzly-http-all-2.4.4.jar`[7] in den Klassenpfad eingebunden werden.

```
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.grizzly.http.server.ServerConfiguration;
import org.glassfish.grizzly.http.server.StaticHttpHandler;
import org.glassfish.grizzly.http.server.accesslog.AccessLogBuilder;

import java.io.IOException;

public class Webserver {
    public static void main(String[] args) {
        String root = args[0];
        int port = Integer.parseInt(args[1]);
        boolean log = Boolean.parseBoolean(args[2]);
```

7 Diese jar-Datei finden Sie im Begleitmaterial zu diesem Buch.

```
      try {
         HttpServer server = HttpServer.createSimpleServer(null, port);
         ServerConfiguration config = server.getServerConfiguration();
         StaticHttpHandler handler = new StaticHttpHandler(root);
         handler.setFileCacheEnabled(false);
         config.addHttpHandler(handler, "/");

         if (log) {
            AccessLogBuilder builder = new AccessLogBuilder("access.log");
            builder.instrument(config);
         }

         Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
               server.shutdownNow();
            }
         });

         server.start();
         System.out.println("ENTER oder Strg + C stoppt den Server.");
         System.in.read();
         System.exit(0);
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

Die `HttpServer`-Methode `createSimpleServer` erzeugt eine Server-Instanz und bindet sie an den vorgegebenen Port. Über eine `ServerConfiguration`-Instanz kann der Server konfiguriert werden. Der `StaticHttpHandler` verarbeitet Anfragen für statische Ressourcen im vorgegebenen Verzeichnis. Mit dem `AccessLogBuilder` wird die Protokollierung der Zugriffe festgelegt. Die `HttpServer`-Methode `start` startet den Server, `shutdownNow` fährt ihn herunter.

7.3 Ein Monitor für Anfragen und Antworten

Das Programm `Monitor` kann genutzt werden, um die zwischen Client und Server ausgetauschten Daten aufzuzeichnen.

Ein Shutdown-Hook sorgt u. a. für das ordnungsgemäße Schließen der Log-Dateien, wenn das Programm mit *Strg + C* beendet wird (siehe auch Kapitel 7.2).

→ Projekt monitor

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class Monitor extends Thread {
   private static OutputStream logRequest;
   private static OutputStream logResponse;
```

```
    private static ServerSocket srv;

    public static void main(String[] args) {
        int localPort = Integer.parseInt(args[0]);
        final String host = args[1];
        final int serverPort = Integer.parseInt(args[2]);
        String request = args[3];
        String response = args[4];

        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                try {
                    if (logRequest != null) {
                        logRequest.close();
                    }
                    if (logResponse != null) {
                        logResponse.close();
                    }
                    if (srv != null)
                        srv.close();
                } catch (IOException e) {
                }
            }
        });

        try {
            logRequest = new FileOutputStream(request);
            logResponse = new FileOutputStream(response);
            srv = new ServerSocket(localPort);

            System.out.println("Beenden mit Strg + C");

            while (true) {
                Socket client = srv.accept();
                new ConnectionHandler(client, host, serverPort,
                        logRequest, logResponse).start();
            }
        } catch (SocketException e) {
        } catch (IOException e) {
            System.err.println("Monitor::main: " + e);
        }
    }
}
```

Abbildung 7-5: Monitor zeichnet HTTP-Anfragen und -Antworten auf

Das Programm `Monitor` fungiert hier als *Proxy*. Dieser leitet die Datenströme weiter an den Server (die Ausgabe des Clients als Eingabe für den Server) bzw. an den Client (die Ausgabe des Servers als Eingabe für den Client). Anfragen und Antworten werden in getrennten Dateien protokolliert.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class ConnectionHandler extends Thread {
   private static final int SOCKET_TIMEOUT = 0;

   private Socket client;
   private String host;
   private int serverPort;
   private OutputStream logRequest;
   private OutputStream logResponse;

   public ConnectionHandler(Socket client, String host, int serverPort,
         OutputStream logRequest, OutputStream logResponse) {

      this.client = client;
      this.host = host;
      this.serverPort = serverPort;
      this.logRequest = logRequest;
      this.logResponse = logResponse;
   }

   @Override
   public void run() {
      try (InputStream fromClient = client.getInputStream();
            OutputStream toClient = client.getOutputStream();
            Socket server = new Socket(host, serverPort);
            InputStream fromServer = server.getInputStream();
            OutputStream toServer = server.getOutputStream()) {

         client.setSoTimeout(SOCKET_TIMEOUT);
         server.setSoTimeout(SOCKET_TIMEOUT);

         Thread forward = new StreamThread(fromClient, toServer, logRequest);
         Thread reverse = new StreamThread(fromServer, toClient, logResponse);

         forward.start();
         reverse.start();

         forward.join();
         reverse.join();
      } catch (InterruptedException e) {
      } catch (IOException e) {
         System.err.println("ConnectionHandler::run: " + e);
      }
   }
}
```

Die `run`-Methode von `ConnectionHandler` verarbeitet die Eingabe eines Clients. `StreamThread` kopiert den Eingabestrom in den Ausgabestrom sowie in die Log-

Datei jeweils für eine Richtung (vom Client zum Server bzw. umgekehrt). Die beiden `run`-Methoden der Threads `forward` und `reverse` laufen quasi gleichzeitig.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.SocketException;

public class StreamThread extends Thread {
   private static final int BUFFER_SIZE = 8192;

   private InputStream from;
   private OutputStream to;
   private OutputStream log;

   public StreamThread(InputStream from, OutputStream to, OutputStream log) {
      this.from = from;
      this.to = to;
      this.log = log;
   }

   @Override
   public void run() {
      try {
         byte[] buffer = new byte[BUFFER_SIZE];
         int bytesRead;
         while ((bytesRead = from.read(buffer)) != -1) {
            to.write(buffer, 0, bytesRead);
            to.flush();
            log.write(buffer, 0, bytesRead);
            log.flush();
         }
      } catch (SocketException e) {
      } catch (IOException e) {
         System.err.println("StreamThread::run " + e);
      } finally {
         try {
            from.close();
            to.close();
         } catch (IOException e) {
         }
      }
   }
}
```

Das Programm kann wie folgt gestartet werden:

```
java -cp bin out/production/monitor Monitor localPort remoteHost
remotePort logRequest logResponse
```

Beispiel:

```
java -cp out/production/monitor Monitor 8080 localhost 80 logRequest.txt
logResponse.txt
```

7.4 Ein einfacher Webserver mit CRUD-Operationen

In diesem Kapitel entwickeln wir eine Stammdatenpflege (CRUD = Create, Read, Update und Delete) für Kontaktdaten (siehe Klasse `Contact`), indem wir neben den HTTP-Methoden `GET` und `POST` auch `PUT` und `DELETE` einsetzen. Um das Beispiel einfach zu halten, verwenden wir nur wenige Attribute: Id, Name und Mailadresse.

→ Projekt crud

```
public class Contact {
   private int id;
   private String name;
   private String email;

   public Contact() {
   }

   public Contact(String name, String email) {
      this.name = name;
      this.email = email;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
      this.email = email;
   }

   @Override
   public String toString() {
      return "Contact{" +
             "id=" + id +
             ", name='" + name + '\'' +
             ", email='" + email + '\'' +
             '}';
   }
}
```

Die folgende Tabelle gibt eine Übersicht darüber, mit welchem URL (Resource-Teil, siehe Kapitel 7.1) und welcher HTTP-Methode Daten gelesen, eingefügt, geändert oder gelöscht werden können.

URL (Resource)	HTTP-Methode	Operation
`/contacts`	`POST`	Anlegen eines neuen Kontakt. Die Id wird vom Service vergeben.
`/contacts`	`GET`	Lesen aller Kontakte.
`/contacts/1`	`GET`	Lesen des Kontakts mit einer vorgegebenen Id (hier: 1).
`/contacts/1`	`PUT`	Ändern des Kontakts mit der Id 1.
`/contacts/1`	`DELETE`	Löschen den Kontakts mit der Id 1.

Der Server wird mit Hilfe des im Kapitel 7.2 vorgestellten *Grizzly HTTP Server Framework* realisiert, außerdem werden JSON-Dokumente zwischen Client und Server ausgetauscht.

Zu diesem Zweck müssen die jar-Datei `libs/grizzly/grizzly-http-all-2.4.4.jar` und die jar-Dateien aus dem Verzeichnis *libs/json* in den Klassenpfad eingebunden werden.

```
import org.glassfish.grizzly.http.server.HttpServer;

import java.io.IOException;

public class ContactServer {
   public static void main(String[] args) {
      try {
         HttpServer server = HttpServer.createSimpleServer(null, 80);
         server.getServerConfiguration()
            .addHttpHandler(new ContactHandler(), "/");

         Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
               server.shutdownNow();
            }
         });

         server.start();
         System.out.println("ENTER oder Strg + C stoppt den Server.");
         System.in.read();
         System.exit(0);
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

```
import org.glassfish.grizzly.http.server.HttpHandler;
import org.glassfish.grizzly.http.server.Request;
import org.glassfish.grizzly.http.server.Response;
import org.glassfish.grizzly.http.util.HttpStatus;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ContactHandler extends HttpHandler {
   private Map<Integer, Contact> contacts = Collections
      .synchronizedMap(new TreeMap<>());
   private AtomicInteger counter;
   private Jsonb jsonb;

   public ContactHandler() {
      counter = new AtomicInteger();
      jsonb = JsonbBuilder.create();
   }

   @Override
   public void service(Request request, Response response) throws Exception {
      String method = request.getMethod().getMethodString();
      String path = request.getRequestURI();

      switch (method) {
         case "GET":
            doGet(path, response);
            break;
         case "POST":
            doPost(path, request, response);
            break;
         case "PUT":
            doPut(path, request, response);
            break;
         case "DELETE":
            doDelete(path, response);
            break;
         default:
            response.setStatus(HttpStatus.BAD_REQUEST_400);
      }
   }

   private void doPost(String path, Request request, Response response)
         throws IOException {

      String result = match(path, "^/contacts$", 0);
      if (result == null) {
         response.setStatus(HttpStatus.BAD_REQUEST_400);
```

```
            return;
        }

        Contact contact = jsonb.fromJson(new InputStreamReader(
            request.getInputStream(), "UTF-8"), Contact.class);
        int id = counter.incrementAndGet();
        contact.setId(id);
        contacts.put(id, contact);
        response.setStatus(HttpStatus.CREATED_201);
        response.setHeader("X-Id", String.valueOf(id));
    }

    private void doGet(String path, Response response) throws IOException {
        String result = match(path, "^/contacts$", 0);
        if (result != null) {
            List<Contact> list = new ArrayList<>();
            synchronized (contacts) {
                for (int key : contacts.keySet()) {
                    list.add(contacts.get(key));
                }
            }
            response.setStatus(HttpStatus.OK_200);
            response.setContentType("application/json");
            jsonb.toJson(list, new OutputStreamWriter(
                response.getOutputStream(), "UTF-8"));
            return;
        }

        String id = match(path, "^/contacts/(\\d+)$", 1);
        if (id == null) {
            response.setStatus(HttpStatus.BAD_REQUEST_400);
            return;
        }

        Contact contact = contacts.get(Integer.parseInt(id));
        if (contact == null) {
            response.setStatus(HttpStatus.NOT_FOUND_404);
            return;
        }

        response.setStatus(HttpStatus.OK_200);
        response.setContentType("application/json");
        jsonb.toJson(contact, new OutputStreamWriter(
            response.getOutputStream(), "UTF-8"));
    }

    private void doPut(String path, Request request, Response response)
            throws IOException {
        String id = match(path, "^/contacts/(\\d+)$", 1);
        if (id == null) {
            response.setStatus(HttpStatus.BAD_REQUEST_400);
            return;
        }

        Contact contact = contacts.get(Integer.parseInt(id));
        if (contact == null) {
            response.setStatus(HttpStatus.NOT_FOUND_404);
            return;
        }
```

```
        Contact c = jsonb.fromJson(new InputStreamReader(
            request.getInputStream(), "UTF-8"), Contact.class);
        contact.setName(c.getName());
        contact.setEmail(c.getEmail());
        response.setStatus(HttpStatus.NO_CONTENT_204);
    }

    private void doDelete(String path, Response response) throws IOException {
        String id = match(path, "^/contacts/(\\d+)$", 1);
        if (id == null) {
            response.setStatus(HttpStatus.BAD_REQUEST_400);
            return;
        }

        Contact contact = contacts.get(Integer.parseInt(id));
        if (contact == null) {
            response.setStatus(HttpStatus.NOT_FOUND_404);
            return;
        }

        contacts.remove(contact.getId());
        response.setStatus(HttpStatus.NO_CONTENT_204);
    }

    private String match(String s, String pattern, int groupIdx) {
        String result = null;
        Matcher m = Pattern.compile(pattern).matcher(s);
        if (m.find()) {
            result = m.group(groupIdx);
        }
        return result;
    }
}
```

Die Methode `doPost` verwendet den "erfundenen" Header `X-Id`, um dem Client die Id des erstellten Kontakts mitzuteilen.

Im Handler wird je nach Methode in der HTTP-Anfrage die `doPost`, `doGet`, `doPut` bzw. `doDelete` aufgerufen, in denen zunächst die Pfadangaben (Resource-Teil des URL) auf Gültigkeit geprüft werden (Methode `match`).

Die Umwandlung von Objekten in JSON-Dokumente und umgekehrt wurde in Kapitel 2 behandelt.

Mit Hilfe des Monitors aus Kapitel 7.3 können diese Strukturen analysiert werden.

Der Client `ContactClient` nutzt die Methoden der Klasse `HttpURLConnection` aus dem Paket `java.net` um Daten mit HTTP zu senden und zu empfangen.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.io.OutputStreamWriter;
import java.lang.reflect.Type;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;

public class ContactClient {
   private static String BASE_URL = "http://localhost";
   private static Jsonb jsonb = JsonbBuilder.create();

   public static void main(String[] args) throws IOException {
      int port = Integer.parseInt(args[0]);
      BASE_URL += ":" + port;

      System.out.println("\n--- Neuen Kontakt anlegen");
      Contact contact = new Contact("Flick, Pit", "pit.flick@gmx.de");
      int id = create(contact);
      System.out.println("Kontakt mit Id=" + id + " angelegt.");

      System.out.println("\n--- Neuen Kontakt anlegen");
      contact = new Contact("Meier, Hugo", "hugo.meier@gmx.de");
      id = create(contact);
      System.out.println("Kontakt mit Id=" + id + " angelegt.");

      System.out.println("\n--- Alle Kontakte lesen");
      List<Contact> list = read();
      for (Contact c : list) {
         System.out.println(c);
      }

      System.out.println("\n--- Kontakt " + id + " lesen");
      contact = read(id);
      System.out.println(contact);

      System.out.println("\n--- Kontakt " + contact.getId() + " ändern");
      contact.setEmail("hugo.meier@web.de");
      update(contact);
      System.out.println("Kontakt geändert");

      System.out.println("\n--- Kontakt " + id + " lesen");
      contact = read(id);
      System.out.println(contact);

      System.out.println("\n--- Kontakt " + contact.getId() + " löschen");
      delete(contact);
      System.out.println("Kontakt gelöscht");

      System.out.println("\n--- Kontakt " + id + " lesen");
      contact = read(id);
      System.out.println(contact);
   }

   private static int create(Contact contact) throws IOException {
      URL url = new URL(BASE_URL + "/contacts");
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setRequestMethod("POST");
      System.out.println(connection.getURL() + " "
         + connection.getRequestMethod());
```

```
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json");
        jsonb.toJson(contact, new OutputStreamWriter(
            connection.getOutputStream(), "UTF-8"));

        int code = connection.getResponseCode();
        if (code == HttpURLConnection.HTTP_CREATED) {
            String id = connection.getHeaderField("X-Id");
            return Integer.parseInt(id);
        } else {
            throw new RuntimeException(code + " " + connection.getResponseMessage());
        }
    }

    private static Contact read(int id) throws IOException {
        URL url = new URL(BASE_URL + "/contacts/" + id);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        System.out.println(connection.getURL() + " "
            + connection.getRequestMethod());

        int code = connection.getResponseCode();
        if (code == HttpURLConnection.HTTP_OK) {
            return jsonb.fromJson(new InputStreamReader(
                connection.getInputStream(), "UTF-8"), Contact.class);
        } else if (code == HttpURLConnection.HTTP_NOT_FOUND) {
            return null;
        } else {
            throw new RuntimeException(code + " " + connection.getResponseMessage());
        }
    }

    private static List<Contact> read() throws IOException {
        URL url = new URL(BASE_URL + "/contacts");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        System.out.println(connection.getURL() + " "
            + connection.getRequestMethod());

        int code = connection.getResponseCode();
        if (code == HttpURLConnection.HTTP_OK) {
            Type type = new ArrayList<Contact>() {}.getClass().getGenericSuperclass();
            return jsonb.fromJson(new InputStreamReader(
                connection.getInputStream(), "UTF-8"), type);
        } else {
            throw new RuntimeException(code + " " + connection.getResponseMessage());
        }
    }

    private static void update(Contact contact) throws IOException {
        URL url = new URL(BASE_URL + "/contacts/" + contact.getId());
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("PUT");
        System.out.println(connection.getURL() + " "
            + connection.getRequestMethod());
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json");
        jsonb.toJson(contact, new OutputStreamWriter(
            connection.getOutputStream(), "UTF-8"));
```

```
        int code = connection.getResponseCode();
        if (code != HttpURLConnection.HTTP_NO_CONTENT) {
           throw new RuntimeException(code + " " + connection.getResponseMessage());
        }
    }

    private static void delete(Contact contact) throws IOException {
        URL url = new URL(BASE_URL + "/contacts/" + contact.getId());
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("DELETE");
        System.out.println(connection.getURL() + " "
           + connection.getRequestMethod());

        int code = connection.getResponseCode();
        if (code != HttpURLConnection.HTTP_NO_CONTENT) {
           throw new RuntimeException(code + " " + connection.getResponseMessage());
        }
    }
}
```

Aufruf des Servers:

```
java -cp out/production/server;../../libs/grizzly/grizzly-http-all-2.4.4.jar;
../../libs/json/* ContactServer
```

Aufruf des Clients:

```
java -cp out/production/client;../../libs/json/* ContactClient
```

Hier die Ausgabe eines beispielhaften Programmablaufs:

```
--- Neuen Kontakt anlegen
http://localhost/contacts POST
Kontakt mit Id=1 angelegt.

--- Neuen Kontakt anlegen
http://localhost/contacts POST
Kontakt mit Id=2 angelegt.

--- Alle Kontakte lesen
http://localhost/contacts GET
Contact{id=1, name='Flick, Pit', email='pit.flick@gmx.de'}
Contact{id=2, name='Meier, Hugo', email='hugo.meier@gmx.de'}

--- Kontakt 2 lesen
http://localhost/contacts/2 GET
Contact{id=2, name='Meier, Hugo', email='hugo.meier@gmx.de'}

--- Kontakt 2 ändern
http://localhost/contacts/2 PUT
Kontakt geändert

--- Kontakt 2 lesen
http://localhost/contacts/2 GET
Contact{id=2, name='Meier, Hugo', email='hugo.meier@web.de'}
```

```
--- Kontakt 2 löschen
http://localhost/contacts/2 DELETE
Kontakt gelöscht

--- Kontakt 2 lesen
http://localhost/contacts/2 GET
null
```

Sie haben in diesem Kapitel die Grundzüge des Programmierparadigmas REST (Representational State Transfer) für Webservices kennen gelernt. REST wird im Kapitel 10 vertieft.

7.5 HTTP/2 und SSL/TLS im Beispiel

Im Mai 2015 wurde HTTP/2 als Nachfolger von HTTP/1.1 verabschiedet. Das HTTP/2-Protokoll enthält drei wichtige Neuerungen, durch die die bestehende Bandbreite bei der Datenübertragung besser ausgenutzt wird:

- Zusammenfassung mehrerer Anfragen (Request-Multiplexing),
- vom Server eingeleitete Datenübertragung (Server-Push),
- verbesserte Datenkompression.

Request-Multiplexing

Während in der Vorgängerversion über eine TCP-Verbindung nur jeweils eine HTTP-Anfrage, gefolgt von einer HTTP-Antwort zu einer Zeit fließen können, können nun simultan mehrere HTTP-Konversationen (Anfragen und Antworten) über eine TCP-Verbindung zwischen Client und Server abgewickelt werden.

Dazu sind HTTP-Anfragen und -Antworten in Streams, Messages und Frames organisiert. *Streams* bestehen aus ein oder mehreren Messages und lassen sich simultan über dieselbe TCP-Verbindung übertragen. Eine *Message* enthält wiederum mehrere Frames. Ein *Frame* enthält binär codierte Header- bzw. Nutzdaten.

Server-Push

Server-Push kann HTTP-Anfragen des Clients einsparen. Beispielsweise dann, wenn eine HTML-Seite mit verlinktem Bild angefordert wird, kann der Server die Übertragung des Bildes anbieten, ohne dass der Client eigens hierfür eine Anfrage stellen muss.

Datenkompression

HTTP/2 kann neben Nutzdaten auch erstmalig Header-Daten (also auch Cookies) mit Hilfe des Algorithmus *HPACK*[8] komprimieren und verringert damit den Umfang dieser Daten erheblich.

---

8 https://http2.github.io/http2-spec/compression.html

Aktuelle Implementierungen unterstützen HTTP/2 nur über verschlüsselte Verbindungen (SSL/TLS).

Zur Demonstration verwenden wir *Apache Tomcat 9.0.14* in der eingebetteten Version.[9] Siehe *libs/tomcat_embed* im Begleitmaterial.

→ Projekt http2

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;
import org.apache.coyote.http2.Http2Protocol;

import java.io.File;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Webserver {
   public static void main(String[] args) throws Exception {
      int port = 8443;
      String contextPath = "";
      String docBase = "web";
      String keystoreFile = "keystore.jks";
      String keystorePass = "secret";

      Logger.getLogger("org").setLevel(Level.SEVERE);

      Tomcat tomcat = new Tomcat();
      tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
      Context ctx = tomcat.addWebapp(contextPath,
         new File(docBase).getAbsolutePath());

      Connector con = new Connector();
      con.setPort(port);
      con.setScheme("https");
      con.setSecure(true);
      con.addUpgradeProtocol(new Http2Protocol());
      con.setAttribute("protocol", "HTTP/1.1");
      con.setAttribute("SSLEnabled", true);
      con.setAttribute("maxThreads", "150");
      con.setAttribute("clientAuth", "false");
      con.setAttribute("sslProtocol", "TLS");
      con.setAttribute("keystoreFile", new File(keystoreFile).getAbsolutePath());
      con.setAttribute("keystorePass", keystorePass);

      Service service = tomcat.getService();
      service.addConnector(con);

      tomcat.start();

      System.out.println("DocBase: " + ctx.getDocBase());
      String url = con.getScheme() + "://localhost:" + con.getPort() +
         ctx.getPath();
      System.out.println("URL: " + url);
      System.out.println("ENTER drücken, um den Server zu stoppen.");
```

9 https://tomcat.apache.org/tomcat-9.0-doc/api/

```
        System.in.read();
        tomcat.stop();
        tomcat.destroy();
    }
}
```

Mit Hilfe von `org.apache.catalina.connector.Connector` wird der Server konfiguriert.

`org.apache.catalina.startup.Tomcat` bietet die Möglichkeit zum Starten und Stoppen des Servers, zur Festlegung der Konfiguration und des so genannten Webkontexts.

Der Server bietet hier statische Seiten aus dem Verzeichnis *web* zum Abruf an.

Sichere Datenübertragung mit SSL/TLS

Secure Sockets Layer (*SSL*), die alte Bezeichnung für *Transport Layer Security* (*TLS*), ist ein Netzwerkprotokoll um Daten verschlüsselt über *HTTPS* (*HyperText Transfer Protocol Secure*) zu übertragen.[10] Der Server muss bei diesem Verfahren seine Identität gegenüber dem Client beweisen. Das geschieht mit einem Zertifikat, das von einer vertrauenswürdigen Zertifizierungsstelle stammen sollte. Wir nutzen hier allerdings zur Vereinfachung ein selbst signiertes Zertifikat. Wurde das Zertifikat vom Client akzeptiert und ist die Verbindung aufgebaut, werden die Daten zwischen Client und Server verschlüsselt übertragen.

Für den Server muss ein Zertifikat erzeugt werden. Hierzu nutzen wir das Java-Tool *keytool*. Das Kommando muss jeweils in einer einzigen Zeile eingegeben werden.

a) Schlüsselpaar (public key/private key) erstellen:

```
keytool -genkeypair -keyAlg RSA -dname "CN=demo"
-ext san=dns:localhost,ip:127.0.0.1,ip:192.168.2.91
-alias demo -validity 90 -keystore keystore.jks -keypass secret
-storepass secret
```

Der Speicher zur Aufbewahrung der Schlüssel wird mit dem Namen *keystore.jks* eingerichtet. Antragsteller ist *demo*, der alternative Antragstellername (*san* = subject alternative name) ist der DNS-Name `localhost`, die IP-Adresse `127.0.0.1` sowie die im Netz gültige IP-Adresse des Rechners, auf dem der Server läuft, hier beispielsweise `192.168.2.91`. Das erzeugte Zertifikat ist 90 Tage gültig.

b) Zertifikat exportieren:

```
keytool -exportcert -alias demo -keystore keystore.jks -file server.cer
-storepass secret
```

Das Zertifikat wird in `server.cer` gespeichert.

---

10 http://de.wikipedia.org/wiki/Transport_Layer_Security
https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol_Secure

c) Zertifikat im Truststore des Clients aufnehmen:

Das Zertifikat wird dem Client ausgehändigt. In unserem Beispiel befinden sich Client und Server auf demselben Rechner, deshalb kann *server.cer* an derselben Stelle bleiben.

Der Client importiert nun das Zertifikat in seinen Truststore:

```
keytool -importcert -noprompt -alias demo -file server.cer
-keystore certs.jks -storepass secret
```

Abbildung 7-6: HTTPS-Authentifizierung mit SSL/TLS[11]

Test

Der Server wird wie folgt gestartet:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.io=ALL-UNNAMED
--add-opens java.rmi/sun.rmi.transport=ALL-UNNAMED
-cp out/production/server;../../libs/tomcat_embed/* Webserver
```

Wir nutzen im Weiteren das mit Java 9 eingeführte *HTTP-Client-API*. Wir statten dafür den Client mit SSL/TLS-Funktionalität aus.

Wir demonstrieren das blockierende (Methode `testBlocking`) und das nicht-blockierende (Methode `testNonBlocking`) Verhalten des Clients beim Senden der HTTP-Anfrage.

11 Quelle: `https://www.elektronik-kompendium.de/sites/net/1811281.htm`

`HttpResponse.BodyHandler.asString()` bestimmt, dass die zurückgegebenen Daten als Zeichenkette behandelt werden.

Die Methode `sendAsync` liefert ein Objekt vom Typ

`CompletableFuture<HttpResponse<String>>`.

Mit Hilfe von `CompletableFuture` können asynchrone Abläufe erzeugt und behandelt werden.[12]

Im vorliegenden Beispiel werden zwei verschiedene Anfragen quasi parallel verarbeitet. Der Aufruf von `future.get()` wartet ggf. auf das Vorliegen des Ergebnisses und gibt es dann zurück.

```
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.stream.Collectors;

public class TestClient {
   public static void main(String[] args) {
      System.setProperty("javax.net.ssl.trustStore", "certs.jks");
      System.setProperty("javax.net.ssl.trustStorePassword", "secret");

      try {
         HttpClient client = HttpClient.newBuilder()
               .version(HttpClient.Version.HTTP_2)
               .build();
         testBlocking(client);
         testNonBlocking(client);
      } catch (Exception e) {
         System.err.println(e);
      }
   }

   private static void testBlocking(HttpClient client)
         throws URISyntaxException, IOException, InterruptedException {

      System.out.println("--- Synchron");

      HttpRequest request = HttpRequest.newBuilder()
            .uri(new URI("https://localhost:8443/page1.txt"))
            .GET()
            .build();

      HttpResponse<String> response = client.send(request,
         HttpResponse.BodyHandlers.ofString());
      System.out.println("Status: " + response.statusCode());
```

12 Siehe: Abts, Dietmar: Grundkurs Java. Springer Vieweg, 10. Aufl. 2018 (Kapitel 9.5)

```
      System.out.println("Version: " + response.version());
      System.out.println(response.body());
   }

   private static void testNonBlocking(HttpClient client)
            throws URISyntaxException, ExecutionException, InterruptedException {

      System.out.println("--- Asynchron");

      List<URI> targets = Arrays.asList(
            new URI("https://localhost:8443/page1.txt"),
            new URI("https://localhost:8443/page2.txt"));

      List<CompletableFuture<String>> futures = targets.stream()
            .map(target -> client
               .sendAsync(
                  HttpRequest.newBuilder(target).GET().build(),
                  HttpResponse.BodyHandlers.ofString())
               .thenApply(resp -> resp.body()))
            .collect(Collectors.toList());

      System.out.println("Hier kann der Client etwas anderes tun.");

      for (CompletableFuture<String> future : futures) {
         System.out.println(future.get());
      }
   }
}
```

Aufruf des Clients:

```
java -cp out/production/client TestClient
```

Ausgabe:

```
--- Synchron
Status: 200
Version: HTTP_2
Das steht auf Seite 1.
--- Asynchron
Hier kann der Client etwas anderes tun.
Das steht auf Seite 1.
Das steht auf Seite 2.
```

Zertifikat im Browser importieren

Die Webseiten können auch im Browser aufgerufen werden. Allerdings akzeptiert der Browser in der Regel keine *selbst unterschriebenen Zertifikate* (wie in unserem Beispiel).

Eine Lösung für *Google Chrome* ist, das Zertifikat *server.cer* über die Zertifikatsverwaltung des Browsers in den erweiterten Einstellungen als vertrauenswürdiges Zertifikat zu importieren. Die einzelnen Schritte sind:

- Im Chrome Menü *Einstellungen* wählen,
- auf *Erweitert* ganz unten klicken,
- *Zertifikate verwalten* wählen,
- Reiter *Vertrauenswürdige Stammzertifizierungsstellen* wählen,
- dort *server.cer* importieren.

Danach sollte der Browser neu gestartet werden.

Das Zertifikat kann auch beim Aufruf von `https://localhost:8443/` heruntergeladen werden. Die einzelnen Schritte sind:

- Auf das Symbol *Nicht sicher* in der Adresszeile klicken,
- auf *Zertifikat (Ungültig)* klicken,
- Reiter *Details* wählen,
- dann *In Datei kopieren…* .

7.6 Aufgaben

1. Entwickeln Sie einen Client, mit dem Dateien vom Webserver mittels HTTP herunter geladen werden können. Nutzen Sie hierzu die Socket-Programmierung mit TCP/IP (siehe Kapitel 4).
2. Realisieren Sie eine Variante zum Downloadprogramm in Aufgabe 1, indem Sie die folgenden Methoden der Klassen `URL` und `URLConnection` benutzen.

   Die Klasse `java.net.URL` repräsentiert einen *Uniform Resource Locator*.

   Die `URL`-Methode

   ```
   URLConnection openConnection() throws IOException
   ```

   liefert ein `java.net.URLConnection`-Objekt, das eine Verbindung zu einer durch den URL adressierten Ressource repräsentiert.

   `URLConnection`-Methoden:

   `int getContentLength()`
   liefert den Wert des Antwortparameters `Content-Length`.

   `InputStream getInputStream() throws IOException`
   liefert einen Eingabestrom zum Lesen über diese Verbindung.
3. Entwickeln Sie einen HTTP-Server, der als einzigen Dienst nach Verbindungsaufnahme mit einem Browser immer dieselbe PDF-Datei sendet. Der Server wird mit der Portnummer und dem Dateinamen als Parameter aufgerufen. Die HTTP-Antwort besteht aus der Kopfzeile mit Status-Code 200, den beiden Antwortparametern `Content-Type` und `Content-Length`, einer Leerzeile und dem Inhalt der Datei.

4. Entwickeln Sie einen HTTP-Server, der einen User auf eine andere Website umleitet (*Redirection*). Beispiel:

   Start des Servers:

   ```
   java -cp out/production/Aufgabe-7.4 Redirector 80
   http://de.wikipedia.org/wiki
   ```

   URL im Browser: `http://localhost/HTTP-Statuscode`

   Die Weiterleitung erfolgt hier an:

   `http://de.wikipedia.org/wiki/HTTP-Statuscode`

   Nutzen Sie hierzu in der HTTP-Antwort den Status "307 Temporary Redirect" und den `Location`-Header, im Beispiel:

   ```
   Location: http://de.wikipedia.org/wiki/HTTP-Statuscode
   ```

5. Entwickeln Sie einen Server (*Forwarder* genannt), der Anfragen an einen anderen Webserver im Netz weiterleitet. Passen Sie hierzu das Programm aus dem Projekt *monitor* an.

   Beispiel:

   ```
   java -cp out/production/Aufgabe-7.5 Forward 80 192.168.2.91:8080
   ```

   Aufruf im Browser: `http://localhost`

   Anwendung:
   Um *Progressive Web Apps*[13] mit einem selbst entwickelten Server im lokalen Netz testen zu können, muss der Server über *HTTPS* oder auf demselben Rechner im Browser mit `http://localhost` aufgerufen werden. Ist der Server nicht mit SSL/TLS konfiguriert, kann dieser mit Hilfe des *Forwarder* auf einem Testrechner im Browser lokal aufgerufen werden, so als ob der Server auch auf diesem Rechner laufen würde.

13 https://de.wikipedia.org/wiki/Progressive_Web_App

8 Bidirektionale Verbindung mit WebSocket

Bei einer HTTP-Verbindung wird jede HTTP-Anfrage mit einer HTTP-Antwort abgeschlossen (siehe Kapitel 7). Anfrage und Antwort bestehen jeweils aus Header-Informationen und Nutzdaten. Jede vom Server an den Client gesendete Nachricht erfordert also eine vorhergehende Anfrage des Clients. Die Kommunikation wird also stets vom Client initiiert.

Viele modere Webanwendungen, z. B. Anzeige von Börsenkursen, Verkehrsinformationen, Online-Spiele) müssen aber auf Ereignisse, die beim Server eintreten, reagieren. Hier ist eine vom Server initiierte Übertragung (*Server-Push*) erforderlich.

Bisher wurden hierfür proprietäre Verfahren eingesetzt wie z. B. Long Polling. Beim *Long Polling* antwortet der Server verzögert auf eine vom Client initiierte Anfrage. Er antwortet erst, wenn ein Ereignis vorliegt oder eine gewisse Zeitspanne verstrichen ist.

8.1 Das WebSocket-Protokoll

Das *WebSocket*-Protokoll[1] ist ein auf TCP basierendes Netzwerkprotokoll, das eine Ergänzung zu HTTP darstellt. Der Client startet wie bei HTTP eine Anfrage. Die zugrundeliegende TCP/IP-Verbindung bleibt aber nach der Übertragung der Client-Daten bestehen. Client und Server können nun jederzeit Nachrichten übertragen (bidirektionale Kommunikation). Ein ständiger Verbindungsauf- und -abbau entfällt. Zudem entfallen die HTTP-Header-Informationen bei jeder Anfrage und Antwort.

Das WebSocket-Protokoll besteht aus zwei Phasen: Handshake und Datenverbindung.

Der *Handshake* wird vom Client eingeleitet. Er führt zu einem Protokollwechsel (Upgrade). Die zufällig generierte Zeichenkette `Sec-WebSocket-Key` dient der Überprüfung (Challenge-Response-Verfahren), ob beide Seiten den Wechsel wollen.

Der Server wendet eine beiden Seiten bekannte Operation auf den Wert von `Sec-WebSocket-Key` an und erzeugt damit den Wert von `Sec-WebSocket-Accept`. Der Client führt dieselbe Operation aus. Stimmt das Ergebnis mit dem vom Server gesendeten Wert überein, ist die Verbindung erfolgreich hergestellt.

Daten werden in Frames transportiert. Jeder Frame enthält einen 2 - 14 Byte langen Header und die Nutzdaten. Der Header besteht u. a. aus dem Operationscode und der Angabe zur Länge der Nutzdaten.

1 https://tools.ietf.org/html/rfc6455

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_8

Abbildung 8-1: Ablauf des WebSocket-Protokolls

WebSocket Handshake

```
GET /ws/echo HTTP/1.1
Host: localhost:8080
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: nvCM37i2AJqj40X8HbbkyQ==

        HTTP/1.1 101
        Upgrade: websocket
        Connection: upgrade
        Sec-WebSocket-Accept: b2DcRzHgMiYPwKTgvuxvkyr2iRI=
```

Analog zu HTTP hat der URL den folgenden Aufbau:

```
ws://server[:port][/resource]
```

Die gängigen Browser unterstützen das WebSocket-Protokoll im Rahmen von HTML5.

8.2 Eine einfache WebSocket-Anwendung

Apache Tomcat unterstützt das WebSocket-Protokoll. Wir nutzen hier die Tomcat-Version 9.0.14 mit dem API Java WebSocket 1.1 (JSR-356). Die Dokumentation hierzu ist im Internet verfügbar (siehe auch Kapitel 7.5).[2]

→ Projekt ws

Beispielhaft wollen wir zunächst einen einfachen Echo-Dienst realisieren. Die Klasse `ws.echo.Echo` implementiert diesen Dienst.

```
package ws.echo;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.concurrent.atomic.AtomicInteger;

@ServerEndpoint("/echo")
public class Echo {
   private AtomicInteger counter = new AtomicInteger();

   @OnOpen
   public void onOpen(Session session) {
      System.out.println(session.getId() + ": onOpen");
      try {
         session.getBasicRemote().sendText("Verbindung wurde hergestellt.");
      } catch (IOException e) {
      }
   }

   @OnClose
   public void onClose(Session session) {
      System.out.println(session.getId() + ": onClose");
   }

   @OnError
   public void onError(Session session, Throwable error) {
      System.out.println(session.getId() + ": " + error.getMessage());
   }

   @OnMessage
   public void onMessage(Session session, String message) {
      try {
         System.out.println(session.getId() + ": " + message);
         session.getBasicRemote().sendText(counter.incrementAndGet()
            + ". " + message);
```

2 https://tomcat.apache.org/tomcat-9.0-doc/api/
https://tomcat.apache.org/tomcat-9.0-doc/websocketapi/index.html

```
      } catch (IOException e) {
         try {
            session.close();
         } catch (IOException e1) {
         }
      }
   }
}
```

Die Annotation `ServerEndpoint` legt den Pfad fest, mit dem die Anwendung aufgerufen werden kann. Die mit den Annotationen `OnOpen`, `OnClose`, `OnError` und `OnMessage` versehenen Callback-Methoden werden beim Öffnen einer neuen WebSocket-Session, beim Schließen der Session, beim Auftreten eines Fehlers (z. B. bei Verbindungsproblemen) und beim Empfang von Daten aufgerufen.

Das Interface `Session` repräsentiert eine Kommunikationsverbindung zwischen Client und Server.

`Session`-Methoden:

`String getId()`
: liefert die eindeutige Session-Id.

`boolean isOpen()`
: prüft, ob der Socket offen ist.

`void close()`
: schließt die Verbindung.

`RemoteEndpoint.Basic getBasicRemote()`
: liefert ein Objekt vom Typ `RemoteEndpoint.Basic`. Dieses Interface repräsentiert die Gegenseite der Verbindung.

Die `RemoteEndpoint.Basic`-Methode

```
void sendText(String text) throws IOException
```

sendet eine Textnachricht.

Für *jede neue Verbindung* erzeugt Tomcat eine neue Instanz der Klasse `Echo`, sodass Client-spezifische Status-Informationen in dieser Instanz gespeichert werden können.

Außer Textdaten können auch Binärdaten (z. B. als `byte`-Array) empfangen und gesendet werden. Diese Möglichkeit betrachten wir hier nicht.

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;

import java.io.File;
```

```
import java.net.InetAddress;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      int port = 8080;
      String contextPath = "ws";
      String docBase = "web";

      Logger.getLogger("").setLevel(Level.SEVERE);

      Tomcat tomcat = new Tomcat();
      tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
      Context ctx = tomcat.addWebapp(contextPath,
         new File(docBase).getAbsolutePath());

      Connector con = new Connector();
      con.setPort(port);

      Service service = tomcat.getService();
      service.addConnector(con);

      tomcat.start();

      System.out.println("DocBase: " + ctx.getDocBase());
      String url = con.getScheme() + "://" +
         InetAddress.getLocalHost().getHostAddress() + ":" +
         con.getPort() + ctx.getPath();
      System.out.println("URL: " + url);
      System.out.println("ENTER stoppt den Server.");

      Thread t = new News.NewsThread(); // wird in Kapitel 8.3 benötigt
      t.start();

      System.in.read();
      t.interrupt();
      tomcat.stop();
      tomcat.destroy();
   }
}
```

Die Klasse `ws.echo.Echo` muss im Klassenpfad der Java-Anwendung zu finden sein. Das genügt bereits. Zur Laufzeit werden alle Klassen mit der Annotation `@ServerEndpoint` automatisch registriert.

Der Server (siehe auch Kapitel 7.5) wird wie folgt gestartet (Kommando in einer Zeile):

```
java --add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.rmi/sun.rmi.transport=ALL-UNNAMED
--add-opens java.base/java.io=ALL-UNNAMED
-cp out/production/server;../../libs/tomcat_embed/* Server
```

Wir testen die Funktionalität zunächst mit Hilfe des Browsers und einer HTML-Seite, die JavaScript-Code enthält.

Datei *web/echo.html*:

```
<html>
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>EchoClient</title>

   <script>
      var uri = "ws://" + window.location.host + "/ws/echo";
      var socket;

      function $(id) {
         return document.getElementById(id);
      }

      function sendMessage() {
         socket.send($("userInput").value);
      }

      window.onload = function () {
         if (! "WebSocket" in window) {
            $("status").innerHTML = "Der Browser unterstützt keine WebSockets.";
            return;
         }

         socket = new WebSocket(uri);

         socket.onopen = function () {
            $("status").innerHTML = "onOpen";
         };

         socket.onclose = function () {
            $("status").innerHTML = "onClose";
         };

         socket.onerror = function (error) {
            $("status").innerHTML = error;
         };

         socket.onmessage = function (message) {
            $("data").innerHTML = message.data;
         };
      }

      window.onunload = function () {
         socket.close();
      }
   </script>
</head>

<body>
   <input id="userInput" type="text">
   <button onclick="sendMessage()">Senden</button><p />
   <div id="status"></div><p />
   <div id="data"></div>
</body>
</html>
```

Beim Laden der HTML-Seite werden die verschiedenen Callback-Funktionen registriert. Die Ereignisse `onopen`, `onclose`, `onerror` und `onmessage` haben die gleiche Bedeutung wie weiter oben.

Im Browser wird nun der URL aufgerufen:

```
http://localhost:8080/ws/echo.html
```

Schaltet man z. B. in *Google Chrome* die Entwicklertools (Taste F12) ein, so erhält man Informationen über die Netzwerkkommunikation zwischen Browser und Server (insbesondere über Headers und Frames).

Java unterstützt mit `HttpClient` auch das WebSocket-API.

Das folgende Programm sendet Zufallszahlen über das WebSocket-Protokoll und gibt jeweils die Antwort des Servers aus.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.WebSocket;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.ExecutionException;

public class EchoClient {
   public static void main(String[] args) {
      String URL = "ws://localhost:8080/ws/echo";

      WebSocket.Listener listener = new WebSocket.Listener() {
         @Override
         public void onOpen(WebSocket webSocket) {
            System.out.println("onOpen");
         }

         @Override
         public CompletionStage<?> onText(WebSocket webSocket, CharSequence data,
               boolean last) {
            webSocket.request(1);
            System.out.println(data);
            return null;
         }

         @Override
         public CompletionStage<?> onClose(WebSocket webSocket, int statusCode,
               String reason) {
            System.out.println("onClose: " + statusCode + " " + reason);
            return null;
         }

         @Override
         public void onError(WebSocket webSocket, Throwable error) {
            System.out.println("onError: " + error.getMessage());
         }
      };
```

```
        HttpClient client = HttpClient.newHttpClient();
        CompletableFuture<WebSocket> future = client.newWebSocketBuilder()
              .buildAsync(URI.create(URL), listener);

        try {
           WebSocket ws = future.get();
           ws.request(1);

           for (int i = 0; i < 9; i++) {
              ws.sendText(String.valueOf(Math.random()), true);
              Thread.sleep(1000);
           }

           ws.sendClose(WebSocket.NORMAL_CLOSURE, "Closed");
        } catch (InterruptedException | ExecutionException e) {
           System.err.println(e);
        }
    }
}
```

Nach Erzeugung einer `HttpClient`-Instanz mit `newHttpClient()` wird für diese mit `newWebSocketBuilder()` eine `WebSocket.Builder`-Instanz erstellt.

Die `WebSocket.Builder`-Methode

```
CompletableFuture<WebSocket> buildAsync(
        URI uri, WebSocket.Listener listener)
```

stellt eine WebSocket-Verbindung zum vorgegebenen URI her und verknüpft sie mit einem Listener vom Typ `WebSocket.Listener`.

Die Methode gibt eine `CompletableFuture`-Instanz[3] zurück, deren Methode `get()` auf die Herstellung der WebSocket-Verbindung wartet und dann eine `WebSocket`-Instanz liefert.

Die hier verwendeten `WebSocket`-Methoden sind:

```
CompletableFuture<WebSocket> sendText(CharSequence message,
       boolean isLast)
```

sendet einen Text. Die gesamte Nachricht kann aus mehreren Teilen bestehen, `isLast` markiert, ob es sich um den letzten Teil dieser Nachricht handelt (`true`). Die Methode liefert eine `CompletableFuture`-Instanz zurück, die "abgeschlossen" ist, wenn die Nachricht gesendet wurde.

```
CompletableFuture<WebSocket> sendClose(int statusCode, String reason)
```

sendet eine "Close-Nachricht" mit Status und Grund. Die Methode liefert eine `CompletableFuture`-Instanz zurück, die "abgeschlossen" ist, wenn diese Nachricht gesendet wurde.

Ein Listener vom Typ `WebSocket.Listener` kann die Methoden `onOpen`, `onText`, `onClose` und `onError` überschreiben (die Signaturen können dem obigen Quellcode

3 Siehe: Abts, Dietmar: Grundkurs Java. Springer Vieweg, 10. Aufl. 2018 (Kapitel 9.5)

entnommen werden). Diese Methoden werden beim Empfang von Nachrichten bzw. beim Eintreten von entsprechenden Events automatisch aufgerufen.

`void request(long n)`

fordert `n` weitere Aufrufe von Listener-Methoden (`onText`, `onClose`) an.

Aufruf des Clients:

```
java -cp out/production/client EchoClient
```

8.3 Server-Push

Das Beispiel in diesem Kapitel demonstriert ein serverseitiges Push-Verfahren. Clients nehmen Verbindung mit dem Server auf und erhalten vom Server dann automatisch die Nachrichten, die beim Server eingegangen sind.

→ Projekt ws

```
package ws.news;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.time.LocalDateTime;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;

@ServerEndpoint("/news")
public class News {
   private static List<Session> connections = new CopyOnWriteArrayList<>();

   @OnOpen
   public void onOpen(Session session) {
      System.out.println(session.getId() + ": onOpen");
      connections.add(session);
   }

   @OnClose
   public void onClose(Session session) {
      System.out.println(session.getId() + ": onClose");
      connections.remove(session);
   }

   @OnError
   public void onError(Session session, Throwable error) {
      System.out.println(session.getId() + ": " + error.getMessage());
   }
```

```
   public static class NewsThread extends Thread {
      @Override
      public void run() {
         AtomicInteger counter = new AtomicInteger();
         System.out.println("NewsThread gestartet.");

         while (true) {
            int z = 5 + (int) (Math.random() * 6);
            try {
               Thread.sleep(1000 * z);
            } catch (InterruptedException e) {
               break;
            }

            broadcast("[" + LocalDateTime.now() + "] News "
               + counter.incrementAndGet());
         }
      }

      private void broadcast(String message) {
         for (Session session : connections) {
            try {
               session.getBasicRemote().sendText(message);
            } catch (IOException e) {
               connections.remove(session);
               try {
                  session.close();
               } catch (IOException e1) {
               }
            }
         }
      }
   }
}
```

Der vom Server gestartete Thread `NewsThread` generiert in zufälligen Zeitabständen neue Nachrichten.

Die bei Herstellung der Verbindung erzeugten Session-Instanzen werden in einer Liste vom Typ `CopyOnWriteArrayList` gespeichert. `CopyOnWriteArrayList` verhindert Ausnahmen vom Typ `ConcurrentModificationException` während der Iteration über die Elemente der Liste. Jeder einzelne Client ist durch eine `Session`-Instanz repräsentiert. Der Server sendet bei Eintreffen einer neuen Nachricht, diese an alle Clients (Methode `broadcast`).

Datei *web/news.html*:

```
<html>

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>NewsClient</title>
```

```
   <script>
      var uri = "ws://" + window.location.host + "/ws/news";
      var socket;

      function $(id) {
         return document.getElementById(id);
      }

      function showMessage(text) {
         var element = $("message");
         element.innerHTML = element.innerHTML + "<br />" + text;
      }

      window.onload = function () {
         if (! "WebSocket" in window) {
            showMessage("Der Browser unterstützt keine WebSockets.");
            return;
         }

         socket = new WebSocket(uri);

         socket.onopen = function () {
            showMessage("onOpen");
         };

         socket.onclose = function () {
            showMessage("onClose");
         };

         socket.onerror = function (error) {
            showMessage(error);
         };

         socket.onmessage = function (message) {
            showMessage(message.data);
         };
      }

      window.onunload = function () {
         socket.close();
      }
   </script>
</head>

<body>
   <div id="message"></div>
</body>

</html>
```

Aufruf im Browser:

`http://localhost:8080/ws/news.html`

Java-Client:

```
import java.io.IOException;
import java.net.URI;
```

```
import java.net.http.HttpClient;
import java.net.http.WebSocket;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.ExecutionException;

public class NewsClient {
   public static void main(String[] args) {
      String URL = "ws://localhost:8080/ws/news";

      WebSocket.Listener listener = new WebSocket.Listener() {
         @Override
         public void onOpen(WebSocket webSocket) {
            System.out.println("onOpen");
         }

         @Override
         public CompletionStage<?> onText(WebSocket webSocket, CharSequence data,
               boolean last) {
            System.out.println(data);
            webSocket.request(1);
            return null;
         }

         @Override
         public CompletionStage<?> onClose(WebSocket webSocket, int statusCode,
               String reason) {
            System.out.println("onClose: " + statusCode + " " + reason);
            return null;
         }

         @Override
         public void onError(WebSocket webSocket, Throwable error) {
            System.out.println("onError: " + error.getMessage());
         }
      };

      HttpClient client = HttpClient.newHttpClient();
      CompletableFuture<WebSocket> future = client.newWebSocketBuilder()
            .buildAsync(URI.create(URL), listener);

      try {
         WebSocket ws = future.get();
         ws.request(1);

         System.out.println("Eingabe von ENTER beendet den Client.");
         System.in.read();
         ws.sendClose(WebSocket.NORMAL_CLOSURE, "Closed");
      } catch (InterruptedException | IOException | ExecutionException e) {
         System.err.println(e);
      }
   }
}
```

Aufruf des Clients:

```
java -cp out/production/client NewsClient
```

8.4 Eine Chat-Anwendung

Mit dem Chat-Client (Browser) ist das so genannte "Chatten" mit mehreren Teilnehmern im Netz möglich. Der Teilnehmer kann sich an- und abmelden und Textzeilen an alle anderen aktiven Teilnehmer senden. Der Chat-Server registriert die angemeldeten Teilnehmer, repräsentiert durch Session-Instanzen, und verteilt eingehende Nachrichten (siehe Methode `message`) an alle registrierten Teilnehmer (Methode `broadcast`). Die Session-Instanzen werden wieder wie im letzten Beispiel in einer (statischen) Liste vom Typ `CopyOnWriteArrayList` gespeichert.

→ Projekt ws

```
package ws.chat;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;

@ServerEndpoint("/chat")
public class Chat {
   private static List<Session> connections = new CopyOnWriteArrayList<>();
   private static AtomicInteger id = new AtomicInteger();
   private String user;

   public Chat() {
      user = "User " + id.incrementAndGet();
   }

   @OnOpen
   public void onOpen(Session session) {
      System.out.println(session.getId() + ": onOpen");
      connections.add(session);
      login();
   }

   @OnClose
   public void onClose(Session session) {
      System.out.println(session.getId() + ": onClose");
      connections.remove(session);
      logout();
   }

   @OnError
   public void onError(Session session, Throwable error) {
      System.out.println(session.getId() + ": " + error.getMessage());
   }

   @OnMessage
   public void onMessage(Session session, String message) {
```

```
      broadcast(user + ": " + message);
   }

   private void login() {
      broadcast("Login: " + user);
      System.out.println("Login: " + user);
      System.out.println("Anzahl User: " + connections.size());
   }

   private void logout() {
      broadcast("Logout: " + user);
      System.out.println("Logout: " + user);
      System.out.println("Anzahl User: " + connections.size());
   }

   private void broadcast(String message) {
      for (Session session : connections) {
         if (session.isOpen()) {
            try {
               session.getBasicRemote().sendText(message);
            } catch (IOException e) {
               connections.remove(session);
               try {
                  session.close();
               } catch (IOException e1) {
               }
               logout();
            }
         }
      }
   }
}
```

Datei *web/chat.html*:

```
<html>
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>ChatClient</title>

   <script>
      var uri = "ws://" + window.location.host + "/ws/chat";
      var socket;

      function $(id) {
         return document.getElementById(id);
      }

      function showMessage(text) {
         var element = $("message");
         element.innerHTML = element.innerHTML + "<br />" + text;
      }

      function sendMessage() {
         if ($("userInput").value.trim().length == 0)
            return;
```

```
            socket.send($("userInput").value);
        }

        window.onload = function () {
            if (! "WebSocket" in window) {
                showMessage("Der Browser unterstützt keine WebSockets.");
                return;
            }

            socket = new WebSocket(uri);

            socket.onopen = function () {
                showMessage("Sitzung gestartet");
            };

            socket.onclose = function () {
                showMessage("Sitzung beendet");
            };

            socket.onerror = function (error) {
                showMessage(error);
            };

            socket.onmessage = function (message) {
                showMessage(message.data);
            };
        }

        window.onunload = function () {
            socket.close();
        }
    </script>
</head>
<body>
    <input id="userInput" type="text" size="80">
    <button onclick="sendMessage();">Senden</button><p />
    <div id="message"></div>
</body>
</html>
```

Aufruf im Browser (in zwei Tabs):

`http://localhost:8080/ws/chat.html`

Abbildung 8-2: Zwei Verbindungen zum Chat-Server

8.5 Aufgaben

1. Entwickeln Sie analog zu Kapitel 8.3 einen Push-Dienst mit Hilfe von Apache Tomcat und dem WebSocket-API.

   Das Java-Programm `Publisher` erzeugt eine Nachricht (bestehend aus Zeitstempel und Text) und sendet diese im JSON-Format mit der HTTP-Methode POST an den Server (siehe Kapitel 7.4).

   Der Server registriert das Servlet `PushServlet`, das die POST-Anfrage entgegennimmt.

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;
import servlet.PushServlet;

import java.io.File;
import java.net.InetAddress;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
  public static void main(String[] args) throws Exception {
     int port = 8080;
     String contextPath = "demo";
     String docBase = "web";

     Logger.getLogger("").setLevel(Level.SEVERE);

     Tomcat tomcat = new Tomcat();
     tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
     Context ctx = tomcat.addWebapp(contextPath, new
File(docBase).getAbsolutePath());

     Connector con = new Connector();
     con.setPort(port);

     Service service = tomcat.getService();
     service.addConnector(con);

     tomcat.addServlet(ctx, "PushServlet", new PushServlet());
     ctx.addServletMappingDecoded("/push", "PushServlet");

     tomcat.start();
     System.out.println("DocBase: " + ctx.getDocBase());
     String url = con.getScheme() + "://" +
            InetAddress.getLocalHost().getHostAddress() + ":" +
            con.getPort() + ctx.getPath();
     System.out.println("URL: " + url);
     System.out.println("ENTER stoppt den Server.");

     System.in.read();
     tomcat.stop();
     tomcat.destroy();
  }
}
```

Das `PushServlet`:

```
package servlet;

import ws.Push;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class PushServlet extends HttpServlet {
  @Override
  public void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws IOException {
      try (BufferedReader in = new BufferedReader(
             new InputStreamReader(req.getInputStream(), "UTF-8"))) {
         String line = in.readLine();
         Push.broadcast(line);
      }
  }
}
```

Die `ServerEndpoint`-Klasse `Push` speichert die aktiven Sessions, bietet die Callback-Methoden und die statische Methode `broadcast` analog zum Beispiel in Kapitel 8.3 an.

Das Programm `Subscriber` entspricht zum großen Teil dem `NewsClient` aus Kapitel 8.3.

Das Begleitmaterial zu diesem Buch enthält auch eine Android-App, die die vom Server gepushten Nachrichten anzeigt.

9 SOAP-basierte Web Services mit JAX-WS

Verfahren für den Zugriff auf entfernte Dienste gibt es schon lange: DCOM (*Distributed Component Object Model*) für Windows-Systeme, CORBA (*Common Objekt Request Broker Architecture*) für nahezu alle Plattformen und RMI (*Remote Method Invocation*) für Java-Anwendungen. Allerdings sind die Formate der zu übertragenden Daten nur innerhalb eines Modells standardisiert.

Die Zusammenarbeit von Anwendungen, die auf unterschiedlichen Verfahren basieren, kann nur mit Hilfe von proprietären Brücken-Programmen sichergestellt werden. Web Services sollen durch Nutzung von Internet-Standards auf eine einfache Art und Weise eine bessere Interoperabilität heterogener Anwendungen bieten.

9.1 Grundlagen

Services implementieren Teile eines Geschäftsprozesses aus Verarbeitungssicht. Die folgende Aufzählung enthält *allgemeine Anforderungen an Services*: [1]

- *Verteilung*
  Ein Service steht in einem Netzwerk zur Verfügung und hat ggf. Zugriff auf weitere Services.
- *Geschlossenheit*
  Ein Service implementiert eine in sich geschlossene Funktion und kann unabhängig von anderen Services aufgerufen werden.
- *Zustandslosigkeit*
  Ein Service kann unabhängig von früheren Aufrufen genutzt werden. Er startet bei jedem Aufruf im gleichen Zustand.
- *Lose Kopplung*
  Abhängigkeiten zwischen Services sind auf ein Minimum reduziert. Ein Service kann dynamisch zur Laufzeit vom Nutzer gesucht und aufgerufen werden.
- *Austauschbarkeit*
  Ein Service kann relativ leicht gegen einen anderen Service mit gleicher Schnittstelle ausgetauscht werden.
- *Ortstransparenz*
  Auf welchem Rechner ein Service läuft, ist für den Nutzer nicht von Belang.

1 Siehe: Goll, J: Architektur- und Entwurfsmuster der Softwaretechnik. Springer Vieweg, 2. Auflage 2014

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_9

- *Plattformunabhängigkeit*
  Serviceanbieter und Servicenutzer können verschiedene Systeme (Rechner, Betriebssysteme) verwenden, ebenso kann sich die eingesetzte Programmiersprache unterscheiden.

- *Zugriff über eine Schnittstelle*
  Die Funktionalität eines Service wird den Clients über eine Schnittstelle bereitgestellt. Die Implementierung bleibt verborgen.

- *Verzeichnisdienst*
  Services sowie fachliche und technische Informationen zu Services können aus einem Verzeichnis abgerufen werden.

Definition Web Service

Eine technische Erläuterung des Begriffs *Web Service* stellt den Zusammenhang zu Schnittstellen, Programmierung und Standards her:

"*Web-Services sind selbstbeschreibende, gekapselte Software-Komponenten, die eine Schnittstelle anbieten, über die ihre Funktionen entfernt aufgerufen und die lose durch den Austausch von Nachrichten miteinander gekoppelt werden können. Zur Erreichung universeller Interoperabilität werden für die Kommunikation die herkömmlichen Kanäle des Internets verwendet.*" [2]

"*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*" [3]

Die zuletzt aufgeführte Definition weist auf die wichtigsten Standards für Web Services hin: HTTP, SOAP und WSDL.

Ein Web Service wird innerhalb einer speziellen Laufzeitumgebung (Applikationsserver) ausgeführt, die die notwendigen Protokolle unterstützt und die Nutzung weiterer Dienste wie beispielsweise Authentifizierung und Anbindung an Datenbanken erlaubt.

Die Nutzung eines Web Service soll anhand des einfachen Beispiels "Ermittlung des aktuellen Artikelpreises" verdeutlicht werden. Die in Java implementierte Client-Anwendung enthält den an den Web Service gerichteten Aufruf der Methode `getPreis` mit der Artikelnummer 4711 als Argument:

---

2 Gesellschaft für Informatik: Symposium auf der 33. GI-Jahrestagung, August 2003

3 `http://www.w3.org/TR/ws-gloss/`

```
PreisImplService service = new PreisImplService();
Preis port = service.getPreisImplPort();
double preis = port.getPreis("4711");
```

Der Server enthält die Implementierung der Methode:

```
public double getPreis(String artikelnummer) { ... }
```

Abbildung 9-1: Aufruf eines Web Service

Die Methode sucht nach der vorgegebenen Artikelnummer und liefert als Ergebnis den zugehörigen Artikelpreis bzw. `-1`, falls der Artikel nicht gefunden wurde.

Analog zum Abruf einer Webseite im Webbrowser sendet der Client, der die Leistung des Web Service nutzen will, eine Anfrage via HTTP an die Adresse des Web Service (im Beispiel `http://localhost:8080/ws/preis`). Diese Anfrage wird vom Web Service verarbeitet und das Ergebnis wird als HTTP-Antwort an den Client zurückgeschickt.

Im Unterschied zum Abruf einer Webseite enthalten Anfrage und Antwort XML-Dokumente. Die Auszeichnungssprache XML (Extensible Markup Language) ist dazu geeignet, die zu übertragenden Daten exakt zu beschreiben. SOAP definiert die Struktur dieser XML-Dokumente.

SOAP

Eine *SOAP-Nachricht* besteht aus einem Umschlag (*Envelope*), der sich aus einem optionalen *Header* und einem notwendigen *Body* zusammensetzt. Der Body enthält die eigentliche Nachricht für den Empfänger, der Header zusätzliche Verarbeitungsinformationen.

SOAP-Anfrage:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getPreis xmlns:ns2="http://artikel.demo/">
         <artikelnummer>4711</artikelnummer>
      </ns2:getPreis>
   </S:Body>
</S:Envelope>
```

SOAP-Antwort:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getPreisResponse xmlns:ns2="http://artikel.demo/">
         <return>110.0</return>
      </ns2:getPreisResponse>
   </S:Body>
</S:Envelope>
```

Die Verwendung von HTTP als Transportprotokoll zur Übertragung der SOAP-Nachrichten ist am weitesten verbreitet. Gründe hierfür sind:

- HTTP-Server sind ausgereift und weit verbreitet,
- HTTP ermöglicht Verbindungen über Firewalls (Port 80),
- SOAP-Nachrichten können einfach in HTTP-Aufrufe eingebunden werden,
- Sicherheitsmechanismen für HTTP können auf SOAP-Nachrichten angewendet werden.

SOAP-Nachrichten können auch mit anderen Protokollen (z. B. SMTP, FTP, JMS) übertragen werden. Das ermöglicht beispielsweise einen zeitversetzten (asynchronen) Datenaustausch zwischen Client und Web Service.

Aufgrund der hier eingesetzten Technologien sind Web Services unabhängig vom Betriebssystem und von der für die Implementierung gewählten Programmiersprache.

WSDL

Mit Hilfe der XML-basierten Sprache *WSDL* (*Web Service Description Language*) werden Web Services beschrieben. WSDL-Dokumente enthalten eine Spezifikation der Funktionsschnittstellen sowie technische Details zum Web Service, z. B. das verwendete Transportprotokoll und die Adresse, unter der der Web Service erreicht werden kann. Nachfolgend ist ein Ausschnitt des WSDL-Dokuments des hier benutzten Beispiels aufgeführt.

WSDL-Dokumente können von Client-Anwendungen automatisch ausgewertet werden, um die genauen Aufrufmodalitäten eines Web Service zu erfahren. Auf dieser Basis können Clients dann zur Laufzeit den Programmcode zum Aufruf des gerade benötigten Web Service dynamisch einbinden. Damit können die Abhängigkeiten zwischen Service-Anbietern und Service-Nutzern auf ein notwendiges Minimum reduziert werden (*lose Kopplung*).

```
<definitions targetNamespace="http://artikel.demo/" name="PreisImplService" ... >
   <types>
      <xsd:schema version="1.0" targetNamespace="http://artikel.demo/">
         <xsd:element name="getPreis" type="tns:getPreis" />
         <xsd:element name="getPreisResponse" type="tns:getPreisResponse" />

         <xsd:complexType name="getPreis">
            <xsd:sequence>
               <xsd:element name="artikelnummer" type="xsd:string" minOccurs="0" />
            </xsd:sequence>
         </xsd:complexType>

         <xsd:complexType name="getPreisResponse">
            <xsd:sequence>
               <xsd:element name="return" type="xsd:double" />
            </xsd:sequence>
         </xsd:complexType>
      </xsd:schema>
   </types>

   <message name="getPreis">
      <part name="parameters" element="tns:getPreis" />
   </message>

   <message name="getPreisResponse">
      <part name="parameters" element="tns:getPreisResponse" />
   </message>

   <portType name="Preis">
      <operation name="getPreis">
         <input ... message="tns:getPreis" />
         <output ... message="tns:getPreisResponse" />
      </operation>
   </portType>

   <binding name="PreisImplPortBinding" type="tns:Preis">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
         style="document" />
      <operation name="getPreis">
         <soap:operation soapAction="" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>

   <service name="PreisImplService">
      <port name="PreisImplPort" binding="tns:PreisImplPortBinding">
         <soap:address location="http://localhost:8080/ws/preis" />
      </port>
   </service>
</definitions>
```

Abbildung 9-2 zeigt, wie ein WSDL-Dokument strukturiert ist.

Abbildung 9-2: Aufbau eines WSDL-Dokuments (WSDL 1.1)

Das `port`-Element enthält die Adresse des Web Service und einen Verweis auf das `binding`-Element. Dort ist definiert, mit welchem Protokoll und in welcher Form Daten übertragen werden. `binding` verweist auf das `portType`-Element. Dieses definiert die Operationen des Service. Jedes `operation`-Element enthält ein `input`-Element und ggf. ein `output`- und `fault`-Element. Alle drei Elemente werden über `message`-Elemente genauer spezifiziert. Das `types`-Element enthält die Datentypen. Sie werden durch ein XML-Schema beschrieben.

SOA

Das Konzept der Web Services bietet die Möglichkeit, die Fähigkeiten einer so genannten *Service-orientierten Architektur* (*SOA*) in Anwendungen zu nutzen.

In einer Service-orientierten Architektur können drei Rollen unterschieden werden:

- Service-Anbieter,
- Service-Verzeichnis und
- Service-Nutzer.

Der *Service-Anbieter* stellt einen Web Service bereit und veröffentlicht seine Verfügbarkeit in einem Verzeichnis.

Das *Service-Verzeichnis* (Registry) stellt Beschreibungen der Web Services (WSDL) bereit und unterstützt das Veröffentlichen und Auffinden von Web Services.

Der *Service-Nutzer* sucht und findet den gewünschten Web Service im Verzeichnis und nutzt ihn, indem er sich mit dem Anbieter verbindet.

Abbildung 9-3: Rollen einer Service-orientierten Architektur

9.2 Ein erstes Beispiel

Es existieren mehrere Open Source Frameworks zur Entwicklung von Web Services mit Java: z. B. *Apache CXF* und *Apache Axis2*.

JAX-WS

Wir nutzen das im Rahmen des *Java Community Process* entwickelte *Java API for XML Web Services* (*JAX-WS*) und die zugehörige Implementierung JAX-WS RI 2.3.0.[4]

Alle jar-Dateien der ZIP-Datei `jaxws-ri-2.3.0.zip` müssen in den Klassenpfad der Anwendung eingebunden werden. Das Begleitmaterial zu diesem Buch enthält im Verzeichnis *libs/jaxws-ri* die erforderlichen jar-Dateien. Am besten richtet man in der Entwicklungsumgebung IntelliJ IDEA eine Global Library mit dem Namen *jaxws-ri* ein, die alle diese jar-Dateien enthält und in jedes Projekt eingebunden wird.

→ Projekt bmi

Als erstes Beispiel soll ein Web Service entwickelt werden, der den *Body-Mass-Index* (BMI) für die Bewertung des Körpergewichts eines Menschen (in kg) in Relation zu seiner Körpergröße (in m) berechnet.

Die hierfür zu implementierende Methode

```
String bmi(double gewicht, double groesse)
```

wird im *Service Endpoint Interface* (*SEI*) deklariert.

4 `https://javaee.github.io/metro-jax-ws/`
`https://github.com/eclipse-ee4j/metro-jax-ws`

Service Endpoint Interface (SEI)

```
package demo.bmi;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Bmi {
   @WebMethod
   String bmi(@WebParam(name = "gewicht") double gewicht,
            @WebParam(name = "groesse") double groesse);
}
```

Die Annotation `@WebService` gibt an, dass es sich um ein SEI für einen Web Service handelt. Mit `@WebMethod` wird die nachfolgende Methode als Service-Operation gekennzeichnet. `@WebParam` sorgt dafür, dass der Wert des Attributs `name` als Name des XML-Elements, das den Parameter beschreibt, verwendet wird (siehe WSDL).

Service Implementation Bean (SIB)

Die folgende Klasse implementiert das Interface.

```
package demo.bmi;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.bmi.Bmi")
public class BmiImpl implements Bmi {
   @Override
   public String bmi(double gewicht, double groesse) {
      double bmi = gewicht / (groesse * groesse);

      if (bmi < 18.5) {
         return "Untergewicht: " + bmi;
      } else if ((bmi >= 18.5) && (bmi < 25)) {
         return "Normalgewicht: " + bmi;
      } else if ((bmi >= 25) && (bmi < 30)) {
         return "Übergewicht: " + bmi;
      } else if (bmi >= 30) {
         return "Adipositas: " + bmi;
      }

      return null;
   }
}
```

Das Attribut `endpointInterface` der Annotation `@WebService` verbindet die Implementierung mit dem SEI. Das Gewicht ist in Kilogramm, die Größe in Meter anzugeben.

Die Klasse `javax.xml.ws.Endpoint` enthält Methoden, mit denen ein Web Service bereitgestellt, veröffentlicht und gestoppt werden kann:

```
static Endpoint create(Object implementor)
void publish(String address)
void stop()
```

```
import demo.bmi.BmiImpl;

import javax.xml.ws.Endpoint;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class BmiPublisher {
   public static void main(String[] args) throws IOException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      String address = "http://localhost:8080/ws/bmi";
      Object implementor = new BmiImpl();

      Endpoint endpoint = Endpoint.create(implementor);
      endpoint.publish(address);

      System.out.println("Service gestartet: " + address);
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      endpoint.stop();
   }
}
```

Die oben aufgeführten Quellcodes liegen im Modul *server* des IntelliJ-IDEA-Projekts *bmi*.

Die Quellcodes können auch in der Konsole wie folgt compiliert werden:

```
javac -encoding UTF-8 -d out/production/server -cp ../../libs/jaxws-ri/*
server/src/demo/bmi/*.java server/src/*.java
```

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/server;../../libs/jaxws-ri/*;
../../libs/activation/* BmiPublisher
```

Das hier benötigte *JavaBeans Activation Framework* (JAF) wird im Begleitmaterial zu diesem Buch zur Verfügung gestellt: `javax.activation-1.2.0.jar`

Nun kann im Browser das WSDL-Dokument abgerufen werden:

`http://localhost:8080/ws/bmi?wsdl`

Das zugehörige XML-Schema ist ausgelagert und kann unter der Adresse

`http://localhost:8080/ws/bmi?xsd=1`

erreicht werden.

Folgender Zusammenhang besteht zwischen den Java-Artefakten und den im WSDL-Dokument verwendeten Namen:

Java-Artefakt		**WSDL-Dokument**	
Paket	`demo.bmi`	`targetNamespace`	`http://bmi.demo/`
SEI	`Bmi`	`portType`	`Bmi`
Methode in SEI	`bmi`	`operation`	`bmi`
SIB	`BmiImpl`	`service` `port`	`BmiImplService` `BmiImplPort`

Nun können mit Hilfe des Tools `WsImport` Klassen und Interfaces zur Unterstützung der Entwicklung des Service Client auf der Basis des WSDL-Dokuments generiert werden (Kommando in einer Zeile):

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp ../../libs/jaxws-ri/jaxws-tools.jar;../../libs/activation/*
com.sun.tools.ws.WsImport -Xnocompile -encoding UTF-8 -s client/src -p gen
http://localhost:8080/ws/bmi?wsdl
```

Der Web Service muss gestartet sein, damit das WSDL-Dokument unter der oben genannten Adresse abgerufen werden kann. Die erzeugten Artefakte werden dem Paket *gen* im Modul *client* zugeordnet.

Web Service Client

```
import gen.Bmi;
import gen.BmiImplService;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class BmiClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      BmiImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new BmiImplService(
            new URL("http://localhost:8888/ws/bmi?wsdl"));
      else
         service = new BmiImplService();

      Bmi port = service.getBmiImplPort();

      System.out.println(port.bmi(46, 1.7));
      System.out.println(port.bmi(56, 1.7));
      System.out.println(port.bmi(78, 1.76));
   }
}
```

Bei gen.`Bmi` und gen.`BmiImplService` handelt es sich um die zuvor generierten Artefakte. Um abweichend von der generierten Adresse

```
http://localhost:8080/ws/bmi?wsdl
```

eine andere zu verwenden kann der Konstruktor mit einer URL-Instanz als Argument aufgerufen werden. Das wird hier benutzt, um eine andere Portnummer wählen zu können (siehe unten).

Nun kann der Client compiliert werden:

```
javac -encoding UTF-8 -d out/production/client -cp ../../libs/jaxws-ri/*
client/src/gen/*.java client/src/*.java
```

Die Ausführung des Programms `BmiClient` mit

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/client;../../libs/jaxws-ri/*;../../libs/activation/*
BmiClient
```

liefert die folgende Ausgabe:

```
Untergewicht: 15.91695501730104
Normalgewicht: 19.377162629757787
Übergewicht: 25.180785123966942
```

Mit Hilfe des Monitors aus Kapitel 7 können die ein- und ausgehenden SOAP-Nachrichten protokolliert werden.

```
java -cp ../../libs/monitor.jar Monitor 8888 localhost 8080 logRequest.txt
logResponse.txt
```

Der Client muss dann mit dem Kommandozeilenparameter `-m` aufgerufen werden:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/client;../../libs/jaxws-ri/*;../../libs/activation/*
BmiClient -m
```

Hier ein Auszug aus dem Protokoll *logRequest.txt* bzw. *logResponse.txt*:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
   <S:Body>
      <ns2:bmi xmlns:ns2="http://bmi.demo/">
         <gewicht>46.0</gewicht>
         <groesse>1.7</groesse>
      </ns2:bmi>
   </S:Body>

</S:Envelope>
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:bmiResponse xmlns:ns2="http://bmi.demo/">
         <return>Untergewicht: 15.91695501730104</return>
      </ns2:bmiResponse>
   </S:Body>
</S:Envelope>
```

Auch mit *Windows PowerShell* kann sehr einfach der Web Service aufgerufen werden.

Das Skript *client.ps1* mit dem Inhalt

```
$url = "http://localhost:8080/ws/bmi?wsdl"
$proxy = New-WebServiceProxy -Uri $url
$response = $proxy.bmi(78, 1.76)
$response
```

kann in der Konsole wie folgt aufgerufen werden:

```
powershell ./client.ps1
```

Ausgabe:

```
Übergewicht: 25.180785123966942
```

9.3 Ein Web Service mit CRUD-Operationen

Dieser Abschnitt zeigt, dass nicht nur einfache Datentypen als Argument- oder Rückgabetypen genutzt werden können. Außerdem erfahren Sie, wie Ausnahmen behandelt werden.

Der folgende Web Service ermöglicht die Verwaltung von Artikeln: Speichern eines neuen Artikels, Suche nach einem Artikel, Ausgabe aller Artikel, Änderung eines Artikels und Löschen eines Artikels.

Ein Objekt vom Typ Artikel enthält Artikel-ID, -bezeichnung und -preis. JAX-WS verwendet intern das *Java API for XML-Binding* (JAXB), um Java-Objekte in XML-Dokumente bzw. Java-Klassen in XML-Schema-Dokumente zu transformieren. In der folgenden Klasse `Artikel` wird die Annotation genutzt, um die Reihenfolge der Elemente in einem komplexen XML-Schema-Typ festzulegen.

→ Projekt crud

```
package demo.artikel;

import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = { "id", "bezeichnung", "preis" })
public class Artikel {
   private int id;
   private String bezeichnung;
   private double preis;

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getBezeichnung() {
      return bezeichnung;
   }
```

```
   public void setBezeichnung(String bezeichnung) {
      this.bezeichnung = bezeichnung;
   }

   public double getPreis() {
      return preis;
   }

   public void setPreis(double preis) {
      this.preis = preis;
   }
}
```

Der Web Service soll die folgenden Methoden anbieten (SEI):

```
package demo.artikel;

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

@WebService
public interface ArtikelManager {
   @WebMethod
   void createArtikel(@WebParam(name = "artikel") Artikel artikel)
      throws ArtikelFault;

   @WebMethod
   Artikel getArtikel(@WebParam(name = "id") int id) throws ArtikelFault;

   @WebMethod
   List<Artikel> getArtikelListe();

   @WebMethod
   void updateArtikel(@WebParam(name = "artikel") Artikel artikel)
      throws ArtikelFault;

   @WebMethod
   void deleteArtikel(@WebParam(name = "id") int id) throws ArtikelFault;
}
```

```
package demo.artikel;

public class ArtikelFault extends Exception {
   public ArtikelFault() {
   }

   public ArtikelFault(String msg) {
      super(msg);
   }
}
```

Hier folgt die Implementierung (SIB). Aus Vereinfachungsgründen werden die Artikel nur im Hauptspeicher in einer Map gespeichert.

```
package demo.artikel;

import javax.jws.WebService;
import java.util.*;
import java.util.stream.Collectors;

@WebService(endpointInterface = "demo.artikel.ArtikelManager")
public class ArtikelManagerImpl implements ArtikelManager {
   private Map<Integer, Artikel> map = new Hashtable<>();

   @Override
   public void createArtikel(Artikel artikel) throws ArtikelFault {
      int id = artikel.getId();
      if (find(id) == null) {
         map.put(id, artikel);
      } else
         throw new ArtikelFault("Artikel " + id + " bereits vorhanden");
   }

   @Override
   public Artikel getArtikel(int id) throws ArtikelFault {
      Artikel artikel = find(id);
      if (artikel == null)
         throw new ArtikelFault("Artikel " + id + " nicht vorhanden");
      return artikel;
   }

   @Override
   public List<Artikel> getArtikelListe() {
      return map.values()
            .stream()
            .sorted((a1, a2) -> a1.getId() - a2.getId())
            .collect(Collectors.toList());
   }

   @Override
   public void updateArtikel(Artikel artikel) throws ArtikelFault {
      int id = artikel.getId();
      Artikel a = find(id);
      if (a == null)
         throw new ArtikelFault("Artikel " + id + " nicht vorhanden");
      else
         map.put(id, artikel);
   }

   @Override
   public void deleteArtikel(int id) throws ArtikelFault {
      Artikel a = find(id);
      if (a == null)
         throw new ArtikelFault("Artikel " + id + " nicht vorhanden");
      else
         map.remove(id);
   }
```

```
   private Artikel find(int id) {
      Artikel artikel = map.get(id);
      return artikel;
   }
}
```

Die Vorgehensweise zur Entwicklung des Web Service entspricht der des vorhergehenden Abschnitts.

Im WSDL-Dokument ist für jede Operation, die eine Ausnahme (`ArtikelFault`) auslösen kann, eine Fehlernachricht vorgesehen, z. B.

```
<operation name="createArtikel">
    <input wsam:Action="..." message="tns:createArtikel"/>
    <output wsam:Action="..." message="tns:createArtikelResponse"/>
    <fault message="tns:ArtikelFault" name="ArtikelFault" wsam:Action="..."/>
</operation>
```

Hier nun der Client:

```
import gen.Artikel;
import gen.ArtikelManager;
import gen.ArtikelManagerImplService;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ArtikelManagerClient {
   private static ArtikelManager port;

   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      ArtikelManagerImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new ArtikelManagerImplService(
            new URL("http://localhost:8888/ws/artikel?wsdl"));
      else
         service = new ArtikelManagerImplService();

      port = service.getArtikelManagerImplPort();

      try {
         System.out.println("Neue Artikel einfügen");
         for (int i = 0; i < 10; i++) {
            Artikel artikel = new Artikel();
            artikel.setId(4711 + i);
            artikel.setBezeichnung("Bezeichnung " + (i + 1));
            artikel.setPreis(1.99 + i);
            port.createArtikel(artikel);
         }

         System.out.println("Artikel 4715 lesen und ändern");
         Artikel artikel = port.getArtikel(4715);
```

```
            print(artikel);
            artikel.setPreis(10.);
            port.updateArtikel(artikel);

            printList(port.getArtikelListe());

            System.out.println("Artikel 4713 löschen");
            port.deleteArtikel(4713);

            System.out.println("Artikel 4713 lesen");
            artikel = port.getArtikel(4713);
            print(artikel);
         } catch (Exception e) {
            System.err.println(e.getMessage());
         }
      }

      private static void print(Artikel artikel) {
         System.out.println("Artikel{" +
               "id=" + artikel.getId() +
               ", bezeichnung='" + artikel.getBezeichnung() + '\'' +
               ", preis=" + artikel.getPreis() +
               '}');
      }

      private static void printList(List<Artikel> list) {
         System.out.println("\nListe aller Artikel:");
         for (Artikel a : list) {
            print(a);
         }
         System.out.println();
      }
   }
```

Da der Artikel 4713, nachdem er gelöscht wurde, nicht vorhanden ist, wird eine Ausnahme ausgelöst.

Die folgende SOAP-Nachricht wird in diesem Fall zum Client geschickt:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Artikel 5000 nicht vorhanden</faultstring>
         <detail>
            <ns2:ArtikelFault xmlns:ns2="http://artikel.demo/">
               <message>Artikel 4713 nicht vorhanden</message>
            </ns2:ArtikelFault>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

Das Begleitmaterial enthält auch *Windows-PowerShell*-Skripte zum Testen der einzelnen Operationen.

9.4 Web Services mit Apache Tomcat veröffentlichen

Mit jedem HTTP-Server, der die Servlet-Technologie unterstützt, kann ein JAX-WS Web Service veröffentlicht werden, z. B. mit *Apache Tomcat*. Gegenüber der bisher genutzten Möglichkeit, einen Web Service zu starten, hat der Einsatz von Tomcat mehrere Vorteile:

- Veröffentlichung mehrerer Web Services innerhalb derselben Webanwendung,
- Nutzung von Container-gesteuerten Sicherheitsfunktionen (Authentifizierung),
- höhere Performance und Skalierbarkeit.

Wir nutzen hier *Apache Tomcat 9.0.14* in der eingebetteten Version (siehe auch Kapitel 7.5).

→ Projekt tomcat

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;

import java.io.File;
import java.net.InetAddress;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      int port = 8080;
      String contextPath = "ws";
      String docBase = "web";

      Logger.getLogger("").setLevel(Level.SEVERE);

      Tomcat tomcat = new Tomcat();
      tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
      Context ctx = tomcat.addWebapp(contextPath,
         new File(docBase).getAbsolutePath());

      Connector con = new Connector();
      con.setPort(port);

      Service service = tomcat.getService();
      service.addConnector(con);

      tomcat.start();

      System.out.println("DocBase: " + ctx.getDocBase());
      String url = con.getScheme() + "://" +
            InetAddress.getLocalHost().getHostAddress() + ":" +
            con.getPort() + ctx.getPath();
      System.out.println("URL: " + url);
      System.out.println("ENTER stoppt den Server.");
```

```
        System.in.read();
        tomcat.stop();
        tomcat.destroy();
    }
}
```

Das Verzeichnis *web* enthält das Unterverzeichnis *WEB-INF* mit dem Verzeichnis *lib* und der Datei *sun-jaxws.xml*.

Die Datei *web/WEB-INF/sun-jaxws.xml* enthält die Endpoint-Beschreibungen der zu veröffentlichenden Web Services (hier wird nur ein Web Service veröffentlicht):

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
   <endpoint name="artikel"
         implementation="demo.artikel.ArtikelManagerImpl"
         url-pattern="/artikel" />
</endpoints>
```

Die Bytecodes der zu veröffentlichenden Web Services werden jeweils in einer jar-Datei zusammengefasst (hier nur der Web Service aus Kapitel 9.3) und in *web/WEB-INF/lib* abgelegt:

```
jar --create --file web/WEB-INF/lib/artikel.jar
-C ../crud/out/production/server demo/artikel/
```

Nun kann der Server gestartet werden:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.io=ALL-UNNAMED
--add-opens java.rmi/sun.rmi.transport=ALL-UNNAMED
-cp out/production/tomcat;../../libs/tomcat_embed/*;../../libs/jaxws-ri/*;
../../libs/activation/* Server
```

9.5 Oneway-Operationen

Mit der Annotation `@Oneway` wird für eine `void`-Methode festgelegt, dass die Operation *keine Nachricht* an den Aufrufer als Antwort sendet. Beim Aufruf einer solchen Operation wird die Kontrolle an den Aufrufer zurückgegeben, bevor die Methode ausgeführt wird.

→ Projekt oneway

Das folgende Beispiel zeigt den Unterschied zwischen dem Aufruf einer normalen `void`-Methode und einer Oneway-Operation.

```
package demo.oneway;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
```

```
@WebService
public interface Send {
   @WebMethod
   void send1(@WebParam(name = "message") String message);

   @WebMethod
   @Oneway
   void send2(@WebParam(name = "message") String message);
}
```

```
package demo.oneway;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.oneway.Send")
public class SendImpl implements Send {
   @Override
   public void send1(String message) {
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      System.out.println(message);
   }

   @Override
   public void send2(String message) {
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      System.out.println(message);
   }
}
```

Die Ausgabe der Nachricht `message` auf der Server-Seite wird in beiden Fällen um 10 Sekunden verzögert.

```
import gen.Send;
import gen.SendImplService;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class SendClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      SendImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new SendImplService(
            new URL("http://localhost:8888/ws/oneway?wsdl"));
      else
```

```
         service = new SendImplService();

      Send port = service.getSendImplPort();

      System.out.println("--- send1 ---");
      port.send1("Hallo");

      System.out.println("--- send2 ---");
      port.send2("Auf Wiedersehen");

      System.out.println("--- Ende ---");
   }
}
```

Generierung der Artefakte und Starten von Server und Client erfolgt wie in Kapitel 9.2. Ebenso können hier mit Hilfe des Monitors die SOAP-Nachrichten protokolliert werden.

Ruft man den Client auf, sieht man deutlich den Unterschied: `send1` kehrt erst nach 10 Sekunden zurück, `send2` praktisch sofort.

SOAP-Antwort von `send1`:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:send1Response xmlns:ns2="http://oneway.demo/" />
   </S:Body>
</S:Envelope>
```

Bei der Ausführung von `send2` wird lediglich der HTTP-Header

```
HTTP/1.1 202 Accepted
```

als Antwort übertragen, keine SOAP-Nachricht.

9.6 Asynchrone Aufrufe

Der Client kann eine Methode so aufrufen, dass der Aufruf nicht so lange blockiert, bis das Ergebnis vorliegt. Vielmehr kann der Client weiter arbeiten. Er "wartet" *asynchron* auf das Ergebnis.

Polling oder Callback

Hierzu gibt es zwei Möglichkeiten:

- Der Client fragt von Zeit zu Zeit nach, ob die Antwort da ist (*polling*, *Pull-Prinzip*), oder
- er stellt eine *Callback*-Methode bereit, die dann automatisch aufgerufen wird, wenn die Antwort vorliegt (*Push-Prinzip*).

→ Projekt async

Wir demonstrieren diese beiden Möglichkeiten anhand eines einfachen Beispiels. Der Web Service ermittelt, ob eine bestimmte Menge eines Produkts verfügbar ist. Die Bearbeitung kann unterschiedlich lange dauern (3 bis 8 Sekunden).

```
package demo.product;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Product {
   @WebMethod
   boolean isAvailable(@WebParam(name = "id") int id,
      @WebParam(name = "amount") int amount);
}
```

```
package demo.product;

import javax.jws.WebService;
import java.util.Random;

@WebService(endpointInterface = "demo.product.Product")
public class ProductImpl implements Product {
   @Override
   public boolean isAvailable(int id, int amount) {
      Random r = new Random();
      int delay = 3 + r.nextInt(6);

      try {
         Thread.sleep(delay * 1000);
      } catch (InterruptedException e) {
      }

      return r.nextBoolean();
   }
}
```

Damit Code für den asynchronen Aufruf (Polling bzw. Callback) generiert wird, muss das folgende XML-Dokument erstellt und beim Aufruf der Generierung der Artefakte berücksichtigt werden:

Bindings-Dokument async.xml

```
<bindings wsdlLocation="http://localhost:8080/ws/product?wsdl"
   xmlns="http://java.sun.com/xml/ns/jaxws">
   <enableAsyncMapping>true</enableAsyncMapping>
</bindings>
```

Das Starten von Server und Client erfolgt wie in den früheren Beispielen.

Kommando zur Generierung:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp ../../libs/jaxws-ri/jaxws-tools.jar;../../libs/activation/*
com.sun.tools.ws.WsImport -Xnocompile -encoding UTF-8 -s client/src -p gen
-b async.xml http://localhost:8080/ws/product?wsdl
```

Es folgt der Quellcode des Client-Programms, der im Anschluss erläutert wird.

```
import gen.IsAvailableResponse;
import gen.Product;
import gen.ProductImplService;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ProductClient {
   private Product port;

   public ProductClient() {
      ProductImplService service = new ProductImplService();
      port = service.getProductImplPort();
   }

   public static void main(String[] args) {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      ProductClient client = new ProductClient();
      client.invokeBlocking();
      client.invokeNonblockingPoll();
      client.invokeNonblockingCallback();
      System.out.println("Warte auf Antwort");
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }
   }

   private void invokeBlocking() {
      System.out.println("--- blocking ---");
      boolean isAvailable = port.isAvailable(4711, 10);
      System.out.println("isAvailable: " + isAvailable);
   }

   private void invokeNonblockingPoll() {
      System.out.println("--- nonblocking, polling ---");
      Response<IsAvailableResponse> res = port.isAvailableAsync(4711, 10);

      while (!res.isDone()) {
         System.out.println("Warte auf Antwort");
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
         }
      }

      try {
         IsAvailableResponse response = res.get();
         boolean isAvailable = response.isReturn();
         System.out.println("isAvailable: " + isAvailable);
      } catch (Exception e) {
         System.err.println(e);
      }
   }
```

```
   private void invokeNonblockingCallback() {
      System.out.println("--- nonblocking, callback ---");
      port.isAvailableAsync(4711, 10, new MyHandler());
   }

   private class MyHandler implements AsyncHandler<IsAvailableResponse> {
      public void handleResponse(Response<IsAvailableResponse> res) {
         try {
            IsAvailableResponse response = res.get();
            boolean isAvailable = response.isReturn();
            System.out.println("isAvailable: " + isAvailable);
         } catch (Exception e) {
            System.err.println(e);
         }
      }
   }
}
```

Nicht-blockierend mit Polling

Das generierte Interface `gen.Product` enthält die Methode

```
Response<IsAvailableResponse> isAvailableAsync(int id, int amount)
```

Das Interface `javax.xml.ws.Response<T>` repräsentiert die Antwort des Aufrufs. Es enthält u. a. die beiden folgenden Methoden:

```
boolean isDone()
T get() throws InterruptedException,
       java.util.concurrent.ExecutionException
```

`isDone` liefert `true`, sobald die Antwort vorliegt, sonst `false`. `get` liefert die Antwort, hier ein Objekt vom Typ `IsAvailableResponse`. Diese Klasse enthält die `boolean`-Property `return`.

Innerhalb einer Schleife wird im Abstand von einer Sekunde nachgefragt, ob die Antwort vorliegt (polling). Ist die Schleife beendet, so kann die Antwort mit `get` geholt werden.

Nicht-blockierend mit Callback

`gen.Product` enthält ebenfalls die Methode

```
Future<?> isAvailableAsync(int id, int amount,
       AsyncHandler<IsAvailableResponse> asyncHandler)
```

`Response` (siehe oben) ist ein Subinterface von `Future`.

Das Interface `javax.xml.ws.AsyncHandler` deklariert die Methode

```
       void handleResponse(Response<T> res),
```

die automatisch aufgerufen wird, sobald die Antwort vorliegt.

Die innere Klasse `MyHandler` implementiert diese Methode für den Typparameter `IsAvailableResponse`.

Nach dem Aufruf von `invokeNonblockingCallback` wartet das Programm 10 Sekunden, damit die `main`-Methode nicht endet, bevor der Rückruf per Callback erfolgt ist.

Beispielhafte Ausgabe des Client-Programms:

```
--- blocking ---
isAvailable: false

--- nonblocking, polling ---
Warte auf Antwort
Warte auf Antwort
Warte auf Antwort
Warte auf Antwort
isAvailable: false
--- nonblocking, callback ---
Warte auf Antwort
isAvailable: true
```

9.7 Transport von Binärdaten

Bei SOAP-Nachrichten handelt es sich um XML-Dokumente, die ausschließlich Textzeichen enthalten dürfen. Will man nun Nachrichten mit binärem Inhalt übertragen, so kann man die binären Daten vor der Übertragung in eine Textform überführen (Codierung) und beim Empfänger wieder dekodieren.

Wir demonstrieren diesen Sachverhalt anhand eines Web Service, der den Upload und Download von Dateien anbietet.

→ Projekt base64

```
package demo.transfer;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import java.io.FileNotFoundException;

@WebService
public interface Transfer {
   @WebMethod
   void upload(@WebParam(name = "name") String name,
         @WebParam(name = "data") byte[] data);

   @WebMethod
   byte[] download(@WebParam(name = "name") String name)
         throws FileNotFoundException;
}
```

Die `byte`-Arrays enthalten den Inhalt der Datei, `name` enthält den Dateinamen.

```
package demo.transfer;

import javax.jws.WebService;
import javax.xml.ws.WebServiceException;
import java.io.*;

@WebService(endpointInterface = "demo.transfer.Transfer")
public class TransferImpl implements Transfer {
   private static final String DIR = "D:/temp";

   @Override
   public void upload(String name, byte[] data) {
      String path = DIR + "/" + name;
      try (BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(path))) {

         out.write(data);
         out.flush();
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Upload");
      }
   }

   @Override
   public byte[] download(String name) throws FileNotFoundException {
      String path = DIR + "/" + name;
      File file = new File(path);
      if (!file.exists())
         throw new FileNotFoundException("Datei existiert nicht.");

      try (ByteArrayOutputStream out = new ByteArrayOutputStream();
          BufferedInputStream in = new BufferedInputStream(
                new FileInputStream(path))) {

         byte[] buffer = new byte[8192];
         int n = 0;
         while ((n = in.read(buffer)) != -1) {
            out.write(buffer, 0, n);
         }

         return out.toByteArray();
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Download.");
      }
   }
}
```

`javax.xml.ws.WebServiceException` ist die Basisklasse aller JAX-WS-Laufzeitfehler (`RuntimeException`).

Im generierten XML-Schema ist für den Parameter `data` der XML-Schema-Datentyp `base64Binary` eingetragen.

Der `UploadClient` überträgt die Datei `duke.png` an den Web Service, der sie dann im Verzeichnis `D:/temp` speichert.

```
import gen.Transfer;
import gen.TransferImplService;
```

```
import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class UploadClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      TransferImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new TransferImplService(
            new URL("http://localhost:8888/ws/transfer?wsdl"));
      else
         service = new TransferImplService();

      Transfer port = service.getTransferImplPort();

      String name = "duke.png";
      try (ByteArrayOutputStream out = new ByteArrayOutputStream();
          BufferedInputStream in = new BufferedInputStream(
                new FileInputStream(name))) {
         byte[] buffer = new byte[8192];
         int n = 0;
         while ((n = in.read(buffer)) != -1) {
            out.write(buffer, 0, n);
         }

         port.upload(name, out.toByteArray());

         System.out.println(name + " wurde hochgeladen.");
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Die Binärdaten werden nach dem *Base64-Verfahren* codiert.[5] Dabei werden Bytes in eine Folge von ASCII-Zeichen umgewandelt.

SOAP-Request

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:upload xmlns:ns2="http://transfer.demo/">
         <name>duke.png</name>
         <data>iVBORw0KGgoAA ... SUVORK5CYII=</data>
      </ns2:upload>
   </S:Body>
</S:Envelope>
```

5 https://de.wikipedia.org/wiki/Base64

Diese Codierung führt zu einer Vergrößerung des Datenvolumens um ca. ein Drittel.

```
import gen.Transfer;
import gen.TransferImplService;

import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class DownloadClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      TransferImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new TransferImplService(
            new URL("http://localhost:8888/ws/transfer?wsdl"));
      else
         service = new TransferImplService();

      Transfer port = service.getTransferImplPort();

      String name = "duke.png";
      try {
         byte[] data = port.download(name);

         try (BufferedOutputStream out = new BufferedOutputStream(
                  new FileOutputStream(name))) {
            out.write(data);
            out.flush();
         }

         System.out.println(name + " wurde heruntergeladen.");
      } catch (Exception e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Das Starten von Server und Client sowie die Generierung der Artefakte erfolgt wie in den früheren Beispielen.

MTOM

SOAP Message Transmission Optimization Mechanism (*MTOM*) ist ein Verfahren zur Optimierung der Übertragung binärer Daten.[6] Der in der SOAP-Nachricht im Base64-Format enthaltene Inhalt wird aus der Nachricht entfernt und in den

6 https://de.wikipedia.org/wiki/SOAP_Message_Transmission_Optimization_Mechanism

Anhang verschoben. In der Nachricht selbst befindet sich dann eine *Referenz* auf die Originaldaten (*raw bytes*) als HTTP-Attachment.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
       <ns2:upload xmlns:ns2="http://transfer.demo/">
          <name>duke.png</name>
          <data>
             <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
                 href="cid:34585f69-58e2-4e8c-bb ... 1ef@example.jaxws.sun.com" />
          </data>
       </ns2:upload>
   </S:Body>
</S:Envelope>
```

→ Projekt mtom1

Mit der Annotation `@javax.xml.ws.soap.MTOM` in der Implementierung des Web Service wird die Optimierung eingeschaltet.

```
@WebService(endpointInterface = "demo.transfer.Transfer")
@MTOM
public class TransferImpl implements Transfer {
   ...
}
```

Das generierte WSDL-Dokument enthält einen *Policy-Eintrag*, der aussagt, dass MTOM verwendet wird.

```
<wsp:Policy wsu:Id="TransferImplPortBinding_MTOM_Policy">
   <ns1:OptimizedMimeSerialization wsp:Optional="true"
      xmlns:ns1="http://.../policy/optimizedmimeserialization"/>
</wsp:Policy>
```

Im `binding`-Element des WSDL-Dokuments wird auf diesen Eintrag verwiesen:

```
<binding name="TransferImplPortBinding" type="tns:Transfer">
    <wsp:PolicyReference URI="#TransferImplPortBinding_MTOM_Policy"/>
    ...
</binding>
```

Sollen sehr große Datenmengen ausgetauscht werden, können diese nicht mehr komplett im Hauptspeicher gehalten werden, sodass `byte[]` als Datentyp ausscheidet. Mit Hilfe von `javax.activation.DataHandler` können Datenströme verarbeitet werden.

→ Projekt mtom2

```
package demo.transfer;

import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlMimeType;
```

```
import java.io.FileNotFoundException;

@WebService
public interface Transfer {
   @WebMethod
   void upload(
         @WebParam(name = "name") String name,
         @WebParam(name = "data") @XmlMimeType("application/octet-stream")
               DataHandler data);

   @WebMethod
   @XmlMimeType("application/octet-stream")
   DataHandler download(@WebParam(name = "name") String name)
         throws FileNotFoundException;
}
```

Die Annotation `@XmlMimeType("application/octet-stream")` sorgt dafür, dass im WSDL-Dokument (XML-Schema) für die Elemente `data` und `return` das Attribut

```
xmime:expectedContentTypes="application/octet-stream"
```

erzeugt wird.

Dies ermöglicht durch entsprechende Generierung die Nutzung des `DataHandler` in den Client-Programmen.[7]

Die spezielle Implementierung `com.sun.xml.ws.developer.StreamingDataHandler` von JAX-WS RI bietet die Methode `moveTo`, um die übertragenen Daten in eine Datei zu speichern.

```
package demo.transfer;

import com.sun.xml.ws.developer.StreamingDataHandler;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.jws.WebService;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.soap.MTOM;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;

@WebService(endpointInterface = "demo.transfer.Transfer")
@MTOM
public class TransferImpl implements Transfer {
   private static final String DIR = "D:/temp";

   @Override
   public void upload(String name, DataHandler data) {
      String path = DIR + "/" + name;
      try (StreamingDataHandler dh = (StreamingDataHandler) data) {
         dh.moveTo(new File(path));
```

7 Die Generierung erfordert hier eine Internet-Verbindung zum Zugriff auf das Schema `http://www.w3.org/2005/05/xmlmime`.

```
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Upload");
      }
   }

   @Override
   public DataHandler download(String name) throws FileNotFoundException {
      String path = DIR + "/" + name;
      File file = new File(path);
      if (!file.exists())
         throw new FileNotFoundException("Datei existiert nicht.");
      DataHandler dh = new DataHandler(new FileDataSource(file));
      return dh;
   }
}
```

Zur Compilierung wird hier auch das *JavaBeans Activation Framework* (JAF) benötigt: `javax.activation-1.2.0.jar`

Der `UploadClient` nutzt die Transfercodierung `chunked`, die die zu übertragenen Daten als Folge mehrerer Teile (Chunks) codiert. Der Sender kann damit bereits mit der Übertragung der Daten beginnen, ohne ihre Gesamtlänge zu kennen

```
import com.sun.xml.ws.developer.JAXWSProperties;
import gen.Transfer;
import gen.TransferImplService;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.xml.ws.BindingProvider;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

public class UploadClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      TransferImplService service;
      if (args.length > 0 && args[0].equals("-m"))
         service = new TransferImplService(
            new URL("http://localhost:8888/ws/transfer?wsdl"));
      else
         service = new TransferImplService();

      Transfer port = service.getTransferImplPort();

      // Transfer-Encoding: chunked
      Map<String, Object> ctx = ((BindingProvider) port).getRequestContext();
      ctx.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8129);

      String name = "duke.png";
      DataHandler dh = new DataHandler(new FileDataSource(name));
      port.upload(name, dh);
```

```
        System.out.println(name + " wurde hochgeladen.");
    }
}


import com.sun.xml.ws.developer.StreamingDataHandler;
import gen.Transfer;
import gen.TransferImplService;

import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class DownloadClient {
    public static void main(String[] args) throws MalformedURLException {
        Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

        TransferImplService service;
        if (args.length > 0 && args[0].equals("-m"))
            service = new TransferImplService(
                new URL("http://localhost:8888/ws/transfer?wsdl"));
        else
            service = new TransferImplService();

        Transfer port = service.getTransferImplPort();

        String name = "duke.png";
        try (StreamingDataHandler dh = (StreamingDataHandler) port.download(name)) {
            dh.moveTo(new File(name));
            System.out.println(name + " wurde heruntergeladen.");
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

9.8 Filterketten

JAX-WS unterstützt so genannte *Interzeptoren* (hier *Handler* genannt), die die Funktionalität auf Client- und/oder Serverseite erweitern. Hiermit können ausgehende und eingehende SOAP-Nachrichten beim Client bzw. Server ausgewertet und auch geändert werden (Nachverarbeitung, Vorverarbeitung).

So kann zum Beispiel auf der Clientseite ein Handler implementiert werden, der unabhängig von der Funktionalität des Web Service die ausgehende SOAP-Nachricht um Authentifizierungsdaten im SOAP-Header ergänzt.

Auf der Serverseite kann dann die eingehende SOAP-Nachricht geprüft werden, sodass der eigentliche Web Service gar nicht aufgerufen wird, falls die Authentifizierung fehlgeschlagen ist. Solche Handler können in der Regel mehrfach wiederverwendet werden und die fachliche Logik muss nicht zusätzlich aufgebläht werden.

Mögliche Einsatzbereiche sind:

- Authentifizierung,
- Verschlüsselung und Entschlüsselung,
- Protokollierung,
- Auditing (wer hat zu welcher Zeit was getan?)

Mehrere Handler können in einer Kette angeordnet sein, die dann in der Reihenfolge ihres Auftretens durchlaufen werden.

Während *SOAP Handler* Zugriff auf die komplette SOAP-Nachricht (Header und Body) haben, können *Logical Handler* nur auf die Nutzdaten des SOAP Body zugreifen. Wir betrachten hier nur SOAP Handler.

→ Projekt filter

Wir wollen im Folgenden drei Handler implementieren:

- Der `ClientHashHandler` schreibt den Usernamen und das Passwort des Benutzers in den SOAP-Header. Passwort ist hier der Hashwert (nach dem Verfahren SHA-256) des Klartextpassworts.
- Der `ServiceHashHandler` extrahiert den Usernamen und den Hashwert und prüft diese Angaben gegen eine Tabelle (Klasse `DataStore`), die alle registrierten User mit dem Hashwert des Klartextpassworts enthält.
- Der `LogHandler` zeichnet aus- und eingehende SOAP-Nachrichten auf. Er wird sowohl auf der Client- als auch auf der Serverseite eingesetzt.

Abbildung 9-4: SOAP Handler des Beispiels

Wir demonstrieren die Wirkungsweise der Handler anhand eines sehr einfachen Web Service. Hier das SEI:

```
package demo.echo;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Echo {
   @WebMethod
   String echo(@WebParam(name = "text") String text);
}
```

Der `ClientHashHandler` implementiert das Interface `SOAPHandler` mit vier Methoden, wobei hier nur die Methode `handleMessage` relevant ist.

`handleMessage` und `handleFault` liefern einen booleschen Wert zurück. Bei `true` wird die Verarbeitung mit evtl. weiteren Handlern fortgesetzt, bei `false` werden andere Handler nicht mehr ausgeführt.

`handleMessage` prüft zunächst, ob es sich um eine ausgehende (Request) oder eine eingehende Nachricht (Response) handelt. In diesem Beispiel muss die ausgehende SOAP-Nachricht geändert werden.

Der Hashwert wird mit Hilfe der Methoden der Klasse `java.security.MessageDigest` berechnet.

Das von `ClientHashHandler` erzeugte SOAP-Header-Element sieht wie folgt aus:

```
<credentials xmlns="http://handler.demo">
   <username>username</username>
   <password>Hashwert des Passworts (hexadezimal)</password>
</credentials>
```

```
package demo.handler;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
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.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Set;

public class ClientHashHandler implements SOAPHandler<SOAPMessageContext> {
   private static final String ALGORITHM = "SHA-256";
   private String username;
   private String password;

   public ClientHashHandler(String username, String password) {
      this.username = username;
      this.password = password;
   }
```

```
   @Override
   public void close(MessageContext ctx) {
   }

   @Override
   public boolean handleFault(SOAPMessageContext ctx) {
      return true;
   }

   @Override
   public boolean handleMessage(SOAPMessageContext ctx) {
      Boolean outbound = (Boolean) ctx.get(
                MessageContext.MESSAGE_OUTBOUND_PROPERTY);

      if (outbound) {
         try {
            SOAPMessage msg = ctx.getMessage();
            SOAPEnvelope env = msg.getSOAPPart().getEnvelope();

            if (env.getHeader() == null)
               env.addHeader();
            SOAPHeader hdr = env.getHeader();
            SOAPHeaderElement elem = hdr.addHeaderElement(new QName(
                  "http://handler.demo", "credentials"));
            elem.addChildElement(
                  new QName("http://handler.demo", "username"))
                  .addTextNode(username);
            elem.addChildElement(
                  new QName("http://handler.demo", "password"))
                  .addTextNode(digest(password));
            msg.saveChanges();
         } catch (Exception e) {
            throw new RuntimeException(e.getMessage());
         }
      }

      return true;
   }

   @Override
   public Set<QName> getHeaders() {
      return null;
   }

   private String digest(String text) throws NoSuchAlgorithmException {
      MessageDigest md = MessageDigest.getInstance(ALGORITHM);
      md.update(text.getBytes());
      byte[] digest = md.digest();
      StringBuilder sb = new StringBuilder();
      for (byte b : digest) {
         sb.append(String.format("%02x", b));
      }
      return sb.toString();
   }
}
```

Analog arbeitet der `ServiceHashHandler`. Aus einer eingehenden SOAP-Nachricht (Request) werden der Username und der Hashwert des Passworts extrahiert und geprüft, ob der Username registriert ist und ob der gesendete Hashwert mit dem beim Server gespeicherten Hashwert übereinstimmt. Diverse Fehler, die hier auftreten können, werden als *SOAP Fault* an den Client geschickt.

```
package demo.handler;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPHeader;
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.ws.soap.SOAPFaultException;
import java.util.Iterator;
import java.util.Set;

public class ServiceHashHandler implements SOAPHandler<SOAPMessageContext> {
   @Override
   public void close(MessageContext ctx) {
   }

   @Override
   public boolean handleFault(SOAPMessageContext ctx) {
      return true;
   }

   @Override
   public boolean handleMessage(SOAPMessageContext ctx) {
      Boolean outbound = (Boolean) ctx.get(
                MessageContext.MESSAGE_OUTBOUND_PROPERTY);

      if (!outbound) {
         SOAPMessage msg = ctx.getMessage();
         SOAPHeader hdr = null;

         try {
            hdr = msg.getSOAPHeader();
         } catch (SOAPException e) {
         }

         if (hdr == null) {
            generateSOAPFault(msg, "Kein Header vorhanden.");
         }

         QName qn = new QName("http://handler.demo", "credentials");
         Iterator it = hdr.getChildElements(qn);
         if (!it.hasNext()) {
            generateSOAPFault(msg, "Header 'credentials' ist nicht vorhanden.");
         }
```

```
            Node node = (Node) it.next();

            NodeList list = node.getChildNodes();
            if (list.getLength() != 2) {
               generateSOAPFault(msg, "Header 'credentials' ist fehlerhaft.");
            }

            String username = list.item(0).getTextContent();
            String password = list.item(1).getTextContent();

            if (username == null || password == null
                  || username.trim().length() == 0
                  || password.trim().length() == 0) {
               generateSOAPFault(msg, "Username bzw. Passwort fehlt.");
            }

            String digest = DataStore.get(username);
            if (digest == null) {
               generateSOAPFault(msg, username + " ist nicht registriert.");
            }

            if (!digest.equals(password)) {
               generateSOAPFault(msg, "Passwort ist falsch.");
            }
         }

         return true;
      }

      @Override
      public Set<QName> getHeaders() {
         return null;
      }

      private void generateSOAPFault(SOAPMessage msg, String reason) {
         try {
            SOAPBody body = msg.getSOAPBody();
            SOAPFault fault = body.addFault();
            fault.setFaultString(reason);
            throw new SOAPFaultException(fault);
         } catch (SOAPException e) {
         }
      }
   }
```

`DataStore` enthält die registrierten User mit den Hashwerten ihrer Passwörter.[8] Diese In-Memory-Lösung simuliert eine Datenbanktabelle.

```
package demo.handler;

import java.util.HashMap;
import java.util.Map;
```

8 Die Seite `http://www.freeformatter.com/message-digest.html` bietet die Möglichkeit, Hashwerte mit verschiedenen Algorithmen berechnen zu lassen.

```
public class DataStore {
   private static final Map<String, String> map;

   static {
      map = new HashMap<>();
      map.put("hugo",
            "5c479f371e72181f28fc221d354a555cd2a070d68b04527d970322cc600b54d8");
      map.put("emil",
            "efbaa8cbfffc1af3afcf8082a3732e4d5111104ae3f6ef8b2545975e50497505");
   }

   public static String get(String key) {
      return map.get(key);
   }
}
```

Auch beim `LogHandler` ist nur die Methode `handleMessage` relevant. Hier wird aber sowohl auf ausgehende als auch eingehende Nachrichten reagiert. Wir nutzen das *Transformer API for XML*, um die Ausgabe des XML-Dokuments mit Zeilenumbrüchen und Einrückungen aufzubereiten, da sonst das gesamte Dokument in einer einzigen Zeile erscheinen würde.

```
package demo.handler;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import java.io.ByteArrayOutputStream;
import java.util.Set;
import java.util.logging.Logger;

public class LogHandler implements SOAPHandler<SOAPMessageContext> {
   private static final Logger log = Logger.getLogger(LogHandler.class.getName());

   @Override
   public void close(MessageContext ctx) {
   }

   @Override
   public boolean handleFault(SOAPMessageContext ctx) {
      log(ctx);
      return true;
   }

   @Override
   public boolean handleMessage(SOAPMessageContext ctx) {
      log(ctx);
      return true;
   }
```

```
   private static String prettyPrint(SOAPMessage msg) {
      try {
         TransformerFactory factory = TransformerFactory.newInstance();
         Transformer transformer = factory.newTransformer();

         transformer.setOutputProperty(OutputKeys.INDENT, "yes");
         transformer.setOutputProperty(
            "{http://xml.apache.org/xslt}indent-amount", "3");

         Source source = msg.getSOAPPart().getContent();

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         StreamResult result = new StreamResult(out);
         transformer.transform(source, result);

         return out.toString();
      } catch (Exception e) {
         throw new RuntimeException(e.getMessage());
      }
   }

   @Override
   public Set<QName> getHeaders() {
      return null;
   }

   private void log(SOAPMessageContext ctx) {
      Boolean outbound = (Boolean)
ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

      if (outbound) {
         log.info(" --- outgoing ---");
      } else {
         log.info(" --- incoming ---");
      }

      try {
         SOAPMessage msg = ctx.getMessage();
         String strMsg = prettyPrint(msg);
         log.info(strMsg);
      } catch (Exception e) {
         throw new RuntimeException(e.getMessage());
      }
   }
}
```

Nun müssen die Handler beim Client und beim Web Service registriert werden.

Handler registrieren

Die Implementierung des Web Service enthält die Annotation `@HandlerChain`. Die Datei *server-handler-chain.xml* liegt in dem Verzeichnis, in dem auch `EchoImpl.class` liegt.

```
package demo.echo;

import javax.jws.HandlerChain;
import javax.jws.WebService;

@WebService(endpointInterface = "demo.echo.Echo")
@HandlerChain(file = "server-handler-chain.xml")
public class EchoImpl implements Echo {
   @Override
   public String echo(String text) {
      return text;
   }
}
```

Inhalt der Datei *server-handler-chain.xml*:

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
   <handler-chain>
      <handler>
         <handler-class>demo.handler.ServiceHashHandler</handler-class>
      </handler>
      <handler>
         <handler-class>demo.handler.LogHandler</handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

Beim Client wird die Handler-Kette durch Aufruf der `Service`-Methode

```
void setHandlerResolver(HandlerResolver handlerResolver)
```

registriert.

Ausführungsreihenfolge

Die Ausführungsreihenfolge in der Handler-Kette ist wie folgt festgelegt: Bei ausgehenden Nachrichten startet die Verarbeitung mit dem ersten Handler in der Kette. Bei eingehenden Nachrichten ist die Reihenfolge umgekehrt (vgl. Abbildung 9-4).

```
import gen.Echo;
import gen.EchoImplService;

import javax.xml.ws.soap.SOAPFaultException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EchoClient {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      String username = "hugo";
      String password = "oguh";
```

```
      EchoImplService service;
      if (args.length > 0 && args[0].equals("-ssl"))
         service = new EchoImplService(new URL(
            "https://localhost:8443/ws/echo?wsdl"));
      else
         service = new EchoImplService();

      service.setHandlerResolver(new ClientHandlerResolver(username, password));
      Echo port = service.getEchoImplPort();

      try {
         String echo = port.echo("Hallo");
         System.out.println(echo);
      } catch (SOAPFaultException e) {
         System.err.println(e.getFault().getFaultString());
      }
   }
}
```

Wird beim Start des Programms `EchoClient` der Kommandozeilenparameter `-ssl` mitgegeben, dann nutzt die Anwendung die SSL-Verschlüsselung (siehe weiter unten).

Der `ClientHandlerResolver` implementiert die Methode `getHandlerChain` des Interfaces `HandlerResolver`.

Die Reihenfolge der `add`-Aufrufe bestimmt die Reihenfolge in der Kette bei ausgehenden Nachrichten (siehe auch "Ausführungsreihenfolge" oben).

```
import demo.handler.ClientHashHandler;
import demo.handler.LogHandler;

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

public class ClientHandlerResolver implements HandlerResolver {
   private String username;
   private String password;

   public ClientHandlerResolver(String username, String password) {
      this.username = username;
      this.password = password;
   }

   @Override
   public List<Handler> getHandlerChain(PortInfo portInfo) {
      List<Handler> handlerChain = new ArrayList<>();
      handlerChain.add(new ClientHashHandler(username, password));
      handlerChain.add(new LogHandler());
      return handlerChain;
   }
}
```

Test

Klartextpasswort für "hugo" ist "oguh", für "emil" ist "lime".

Der in `LogHandler` verwendete `Logger` kann so konfiguriert werden, dass die Ausgabe in eine Datei gelenkt wird.

logging.properties hat den folgenden Inhalt:

```
handlers = java.util.logging.FileHandler
#handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
.level = SEVERE
java.util.logging.FileHandler.pattern = server.log
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.append = true
demo.handler.level = INFO
```

Der Server (`EchoPublisher`) enthält die Anweisung

```
System.setProperty("java.util.logging.config.file", "logging.properties");
```

Hier wird nur die Ausgabe beim Server gezeigt (Datei: *server.log*). Das fettgedruckte XML-Fragment wurde vom `ClientHashHandler` eingefügt.

```
Jan. 30, 2019 7:55:03 NACHM. demo.handler.LogHandler log
INFO:  --- incoming ---
Jan. 30, 2019 7:55:03 NACHM. demo.handler.LogHandler log
INFO: <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header xmlns="http://handler.demo">
      <credentials>
         <username>hugo</username>
         <password>5c479f371e72181f28fc221d354a555cd2a ... 600b54d8</password>
      </credentials>
   </SOAP-ENV:Header>
   <S:Body xmlns:ns2="http://echo.demo/">
      <ns2:echo>
         <text>Hallo</text>
      </ns2:echo>
   </S:Body>
</S:Envelope>

Jan. 30, 2019 7:55:03 NACHM. demo.handler.LogHandler log
INFO:  --- outgoing ---
Jan. 30, 2019 7:55:03 NACHM. demo.handler.LogHandler log
INFO: <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body xmlns:ns2="http://echo.demo/">
      <ns2:echoResponse>
         <return>Hallo</return>
      </ns2:echoResponse>
   </S:Body>
</S:Envelope>
```

SSL/TLS-Verschlüsselung

Wir wollen nun die Übertragung zwischen Client und Server verschlüsseln. Hierzu nutzen wir *Apache Tomcat*, erzeugen ein Server-Zertifikat und konfigurieren Server und Client so, dass die Daten verschlüsselt übertragen werden können.

Die Vorgehensweise entspricht der Darstellung in Kapitel 7.5. Wie dort geschildert werden Schlüsselpaar, Zertifikat und Truststore erzeugt.

Wir nutzen *Apache Tomcat 9.0.14* in der eingebetteten Version (siehe Kapitel 9.4 und Kapitel 7.5).

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;

import java.io.File;

public class TomcatSSL {
   public static void main(String[] args) throws Exception {
      int port = 8443;
      String contextPath = "ws";
      String docBase = "web";
      String keystoreFile = "keystore.jks";
      String keystorePass = "secret";

      System.setProperty("java.util.logging.config.file", "logging.properties");

      Tomcat tomcat = new Tomcat();
      tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
      Context ctx = tomcat.addWebapp(contextPath,
         new File(docBase).getAbsolutePath());

      Connector con = new Connector();
      con.setPort(port);
      con.setScheme("https");
      con.setSecure(true);
      con.setAttribute("protocol", "HTTP/1.1");
      con.setAttribute("SSLEnabled", true);
      con.setAttribute("maxThreads", "150");
      con.setAttribute("clientAuth", "false");
      con.setAttribute("sslProtocol", "TLS");
      con.setAttribute("keystoreFile", new File(keystoreFile).getAbsolutePath());
      con.setAttribute("keystorePass", keystorePass);

      Service service = tomcat.getService();
      service.addConnector(con);

      tomcat.start();

      System.out.println("DocBase: " + ctx.getDocBase());
      String url = con.getScheme() + "://localhost:" + con.getPort() +
                        ctx.getPath();
      System.out.println("URL: " + url);
      System.out.println("ENTER drücken, um den Server zu stoppen.");
```

```
        System.in.read();
        tomcat.stop();
        tomcat.destroy();
    }
}
```

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.io=ALL-UNNAMED
--add-opens java.rmi/sun.rmi.transport=ALL-UNNAMED
-cp out/production/server;../../libs/tomcat_embed/*;../../libs/jaxws-ri/*;
../../libs/activation/* TomcatSSL
```

Start des Clients:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/client;../../libs/jaxws-ri/*;../../libs/activation/*
EchoClient -ssl
```

9.9 Contract First

Das WSDL-Dokument ist das zentrale Bindeglied zwischen Service-Nutzer und Service-Anbieter. Es beschreibt den Web Service plattform- und programmiersprachenunabhängig.

Das WSDL-Dokument kann auf zwei Arten erstellt werden:

- *Code First*
  Es wird zunächst der Web Service in Java programmiert, daraus werden dann das WSDL-Dokument und andere Artefakte generiert. Diese Vorgehensweise haben wir in allen Beispielen bisher gewählt.
- *Contract First*
  Es wird zuerst das WSDL-Dokument (z. B. von Hand oder mit geeigneten Tools) erstellt und hieraus der notwendige Java-Code generiert. Auf dieser Basis wird dann die Anwendungslogik des Web Service implementiert.

Bei beiden Vorgehensweisen werden die Artefakte zur Entwicklung des Clients aus dem WSDL-Dokument generiert.

Nachteile des Code-First-Ansatzes

Ein WSDL-Dokument manuell zu erstellen, ist relativ aufwändig. Warum dieser Weg aber doch sinnvoll ist, zeigen die folgenden Gefahren des Code-First-Ansatzes:

- Mit der Änderung des Web Service ändert sich automatisch das WSDL-Dokument. Die Gefahr ist vorhanden, dass sich dann auch Teile der Client-Artefakte mit ändern. Eine einmal erstellte Schnittstellenbeschreibung

sollte aber stabil sein. Client-Anwendungen, die gegen die veröffentlichte Schnittstelle programmiert sind, sollten nicht geändert werden müssen.

- Auch beim Wechsel des Frameworks zur Entwicklung von Web Services (Einsatz anderer Generierungstools) kann sich das WSDL-Dokument ändern. Getroffene Annahmen bei der Generierung aus der Java-Implementierung können leicht zu Inkompatibilitäten führen.
- Eine komplizierte, ungeschickte Implementierung des Web Service wirkt sich möglicherweise auf das generierte WSDL-Dokument aus, sodass dieses auch nur schwer verständlich ist.

Dem Aufwand bei der manuellen Erstellung steht die Einfachheit und Schnelligkeit der automatischen WSDL-Generierung gegenüber.

Ein *Kompromiss* wäre, den Code-First-Ansatz für die erstmalige Erstellung des WSDL-Dokuments zu verwenden und bei späteren Iterationen zum Contract-First-Ansatz zu wechseln. Darüber hinaus bietet JAX-WS eine Reihe von Möglichkeiten, mit Annotationen Inhalte des WSDL-Dokuments festzulegen.

→ Projekt contract_first

Im Folgenden wird das Vorgehen beim *Contract-First-Ansatz* demonstriert. Ausgangsbasis ist das folgende WSDL-Dokument `TimeService.wsdl`. Die Datei liegt im Verzeichnis *web/WEB-INF/wsdl* und hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://time.demo/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="TimeService"
   targetNamespace="http://time.demo/">
   <wsdl:documentation>Test Contract First</wsdl:documentation>
   <wsdl:types>
      <xsd:schema targetNamespace="http://time.demo/">
         <xsd:element name="getTime">
            <xsd:complexType>
               <xsd:sequence />
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="getTimeResponse">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="time" minOccurs="1" maxOccurs="1">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:string">
                                        <xsd:length value="19"></xsd:length>
                                </xsd:restriction>
                        </xsd:simpleType>
                  </xsd:element>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
      </xsd:schema>
   </wsdl:types>
```

```
   <wsdl:message name="getTimeRequest">
      <wsdl:part element="tns:getTime" name="parameters" />
   </wsdl:message>
   <wsdl:message name="getTimeResponse">
      <wsdl:part element="tns:getTimeResponse" name="parameters" />
   </wsdl:message>
   <wsdl:portType name="Time">
      <wsdl:operation name="getTime">
         <wsdl:input message="tns:getTimeRequest" />
         <wsdl:output message="tns:getTimeResponse" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="TimeServiceSOAP" type="tns:Time">
      <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="getTime">
         <soap:operation soapAction="http://time.demo/getTime" />
         <wsdl:input>
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output>
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="TimeService">
      <wsdl:port binding="tns:TimeServiceSOAP" name="TimeServiceSOAP">
         <soap:address location="http://localhost:8080/ws/time" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Für das Weitere können wir dieser Beschreibung entnehmen:

Element	Name	Artefakt
`targetNamespace`	`http://time.demo/`	Paket: `demo.time`
`portType`	`Time`	SEI
`operation`	`getTime`	`Time`-Methode: `getTime`
`service`	`TimeService`	Klasse: `TimeService` (Client)
`port`	`TimeServiceSOAP`	`TimeService`-Methode: `getTimeServiceSOAP` (Client)

Der fett gedruckte Teil des obigen XML-Schemas zeigt, dass der Datentyp des Rückgabewerts der Operation `getTime` gegenüber den Möglichkeiten der automatischen Generierung weiter eingeschränkt ist: `getTime` liefert eine von `null` verschiedene Zeichenkette der Länge 19.

Das WSDL-Dokument als *Contract* kann also Service-Anforderungen spezifizieren, die mit der Angabe der Java-Datentypen alleine nicht ausgedrückt werden können.

Implementierung des Web Service

Zunächst werden die Java-Artefakte zur Implementierung des Web Service generiert. Der generierte Quellcode befindet sich im Paket `gen`. Das Kommando muss in einer einzigen Zeile eingegeben werden:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp ../../libs/jaxws-ri/jaxws-tools.jar;../../libs/activation/*
com.sun.tools.ws.WsImport -Xnocompile -encoding UTF-8 -s server/src
-p gen web/WEB-INF/wsdl/TimeService.wsdl
```

```
package demo.time;

import gen.Time;

import javax.jws.WebService;
import java.time.LocalDateTime;

@WebService(
   endpointInterface = "gen.Time",
   serviceName = "TimeService",
   portName = "TimeServiceSOAP",
   wsdlLocation = "WEB-INF/wsdl/TimeService.wsdl"
)
public class TimeServiceImpl implements Time {
   @Override
   public String getTime() {
      return LocalDateTime.now().toString();
   }
}
```

Der Web Service kann nun wie bekannt mit Apache Tomcat veröffentlicht werden (siehe Kapitel 9.4).

Die Attribute der Annotation `@WebService` in `TimeServiceImpl` sorgen dafür, dass nicht das aus den Artefakten generierte, sondern das selbst erstellte WSDL-Dokument über `http://localhost:8080/ws/time?wsdl` zur Verfügung gestellt wird.

Implementierung des Client

Die Generierung der Artefakte für den Client erfolgt mit:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp ../../libs/jaxws-ri/jaxws-tools.jar;../../libs/activation/*
com.sun.tools.ws.WsImport -Xnocompile -encoding UTF-8 -s client/src
-p gen http://localhost:8080/ws/time?wsdl
```

```
import gen.Time;
import gen.TimeService;

import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class TimeServiceClient1 {
   public static void main(String[] args)  {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      TimeService service = new TimeService();
      Time port = service.getTimeServiceSOAP();
      String time = port.getTime();
      System.out.println(time);
   }
}
```

Die WSDL-Adresse und der qualifizierte Service-Name sind in der Klasse `client.TimeService` codiert.

Um den Service-Aufruf flexibler zu gestalten, können die Parameter zur Herstellung der Verbindung mit dem Web Service erst zur Laufzeit bestimmt werden:

```
import gen.Time;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TimeServiceClient2 {
   public static void main(String[] args) throws MalformedURLException {
      Logger.getLogger("javax.xml").setLevel(Level.SEVERE);

      Service service = Service.create(
            new URL("http://localhost:8080/ws/time?wsdl"),
            new QName("http://time.demo/", "TimeService"));

      Time port = service.getPort(Time.class);
      String time = port.getTime();
      System.out.println(time);
   }
}
```

```
static Service create(java.net.URL wsdlDocumentLocation,
       javax.xml.namespace.QName serviceName)
```

erzeugt eine `Service`-Instanz.

```
<T> T getPort(Class<T> sei)
```

liefert zum vorgegebenen SEI (Service Endpoint Interface) einen Proxy.

9.10 Aufgaben

1. Die Mosteller-Formel ist eine mathematische Formel zur Abschätzung der Körperoberfläche (m^2) aus der Körpergröße (cm) und dem Körpergewicht (kg) beim Menschen:

   ```
   Oberfläche = Math.sqrt(Größe * Gewicht) / 60.
   ```[9]

   Erzeugen Sie nach dem *Code-First-Ansatz* einen Web Service und einen Client.

2. Durch eine Hash-Funktion wird eine Nachricht auf einen Hashwert mit einer festen Länge abgebildet. In Java steht hierfür die Klasse `java.security.MessageDigest` zur Verfügung.
   - Entwickeln Sie ein Testprogramm, das zu einer Textnachricht den Hashwert mit Hilfe der Algorithmen MD5, SHA-1, SHA-256 oder SHA-512 berechnet und diesen im Hexadezimalformat ausgibt.
   - Entwickeln Sie einen Web Service nach dem Code-First-Ansatz mit der Operation

     ```
     String digest(String text, String algorithm)
     ```

   - Entwickeln Sie dann einen Client, der diesen Web Service aufruft.

3. Veröffentlichen Sie die Web Services aus den Aufgaben 1 und 2 mit Apache Tomcat. Das Begleitmaterial zu diesem Buch enthält auch eine einfache Webanwendung, in der der Browser mit dem Web Service aus Aufgabe 2 kommuniziert.

4. Erstellen Sie einen Web Service mit der `Oneway`-Operation

   ```
   void send(Message message),
   ```

   die eine Nachricht auf der Server-Konsole ausgibt. `message` enthält den Zeitstempel, den Benutzernamen und den eigentlichen Text. Implementieren Sie auch einen Client.

5. Erstellen Sie einen Client, der den Web Service aus Aufgabe 2 asynchron aufruft und für die Antwort eine Callback-Methode zur Verfügung stellt.

6. Ein Webservice soll die Operation

   ```
   DataHandler download()
   ```

   anbieten, mit der ein bestimmtes PDF-Dokument (z. B. Auftragsformular) heruntergeladen werden kann. Nutzen Sie MTOM.

7. Entwickeln Sie einen Handler für einen Service Ihrer Wahl, der protokolliert, wie lange der Service (in Milliskunden) gelaufen ist. Die Startzeit soll im `SOAPMessageContext` festgehalten werden. Hier der relevante Ausschnitt aus dem Handler:

---

9 Siehe `http://de.wikipedia.org/wiki/Mosteller-Formel`

```
@Override
public boolean handleMessage(SOAPMessageContext ctx) {
   Boolean outbound = (Boolean)
      ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

   if (outbound) {
      Instant end = Instant.now();
      Instant start = (Instant) ctx.get("START");
      Duration duration = Duration.between(start, end);
      String message = String.format(
               "Start: %s Ende: %s Dauer: %5d ms", start.toString(),
                        end.toString(), duration.toMillis());
      log.info(message);
   } else {
      ctx.put("START", Instant.now());
   }

   return true;
}
```

8. Erstellen Sie nach dem *Contract-First-Ansatz* einen Web Service, der die folgende Operation anbietet:

   ```
   List<Integer> rand(int bound, int length)
   ```

   Hier das zugehörige XML-Schema:

   ```
   <xsd:schema targetNamespace="http://rand.demo/">
      <xsd:element name="rand" type="tns:rand" />
      <xsd:element name="randResponse" type="tns:randResponse" />

      <xsd:complexType name="rand">
         <xsd:sequence>
               <xsd:element name="bound" type="xsd:int" />
               <xsd:element name="length" type="xsd:int" />
         </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="randResponse">
         <xsd:sequence>

               <xsd:element name="return" type="xsd:int" minOccurs="0"
                        maxOccurs="unbounded" />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:schema>
   ```

   Die Operation soll `length` Zufallszahlen liefern, die zwischen `0` und `bound` liegen. Entwickeln Sie auch einen Client hierzu.

9. Entwickeln Sie die folgenden beiden Services:
   - Verfügbarkeitsprüfung für Artikel:

     ```
     boolean isAvailable(@WebParam(name = "id") int id,
        @WebParam(name = "amount") int amount);
     ```

   - Bestellung entgegennehmen und bei Verfügbarkeit der benötigten Menge Ausgabe des Rechnungsbetrags:

```
double order(@WebParam(name = "id") int id,
   @WebParam(name = "amount") int amount)
      throws OrderFault;
```

Der erste Service verwendet den zweiten. Der Client soll mit JavaFX implementiert werden und den ersten Service asynchron aufrufen.

10 REST-basierte Web Services mit JAX-RS

SOAP-basierte Web Services und die im Umfeld vorhandenen Standards sind gut geeignet, die Implementierung von SOA zu ermöglichen.

Allerdings haben die XML-Nachrichten einen recht großen Overhead: viele Metadaten, wenig Nutzdaten, was u. a. Analyse und Synthese solcher Nachrichten aufwändig macht. Zudem macht die Vielzahl von so genannten "WS-*"-Standards (Spezifikationen im Kontext von Web Services mit SOAP/WSDL) einen guten Überblick fast unmöglich.

In den letzten Jahren ist eine leichtgewichtige und vom Datenformat unabhängige Alternative zur Realisierung von Service-basierten Anwendungen hervorgetreten: REST (Representational State Transfer). Im Folgenden werden die Grundlagen von REST erläutert und die Implementierung von REST-basierten Web Services (hier kurz *REST-Services* genannt) auf der Basis von JAX-RS vorgestellt.

10.1 Die Grundprinzipien von REST

REST (*Representational State Transfer*) ist ein Architekturstil, dessen Prinzipien wichtige Eigenschaften des Web – abstrahiert von der konkreten Architektur, die HTTP zugrunde liegt – zusammenfassen.[1] Das Web ist demgemäß eine Ausprägung des REST-Architekturstils.

Im Folgenden werden die Prinzipien, die REST zugrunde liegen, vorgestellt.

Ressourcenorientierung

Jede Client-Anfrage ist auf eine *Ressource* bezogen. Ressource steht hier für eine Informationseinheit, z. B. ein Textdokument, ein Bild, ein Auftrag in einem Auftragssystem usw., kann aber auch eine Sammlung von mehreren anderen Ressourcen sein.

Adressierbarkeit

Jede Ressource wird über einen eindeutigen *URI* (*Uniform Resource Identifier*) identifiziert.

Beispiel:

`http://www.test.de/customers/4711`

identifiziert den Kunden mit dem Schlüssel 4711.

1 Der Begriff REST taucht erstmalig in der im Jahr 2000 beendeten Dissertation von Roy Thomas Fielding auf: R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD thesis, University of California 2000, `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9_10

Unterschiedliche Repräsentationen

Es können verschiedene *Repräsentationen* ein- und derselben Ressource existieren. So kann ein Client eine Ressource beispielsweise im XML-, HTML- oder JSON-Format anfordern.

Verknüpfungen

Jede Repräsentation einer Ressource versetzt den Client in einen neuen Zustand (*State*). Die Repräsentation kann einen Hyperlink enthalten, der mitteilt, welche Aktion er als nächste ausführen kann. Die Verwendung des Hyperlinks gibt eine weitere Repräsentation einer anderen Ressource zurück und der Client wird in einen neuen Zustand versetzt (Zustandsübergang, *State Transfer*). Diese Eigenschaft wird auch als "Hypermedia As The Engine Of Application State" (*HATEOAS*) bezeichnet.

Statuslose Kommunikation

Um die Skalierbarkeit der Anwendung zu erhöhen und die Kopplung des Clients an den Server zu verringern, wird auf einen clientspezifischen Sitzungsstatus verzichtet. Es existieren also keine Benutzersitzungen in Form von Sessions oder Cookies. Bei jeder Anfrage werden alle notwendigen Informationen wieder neu mitgeschickt, der Zustand wird also vom Client gehalten oder der Server erstellt für den Status eine eigene Ressource (Beispiel: Warenkorb).

Einheitlicher Zugriff auf Ressourcen

REST definiert einen einheitlichen Satz von Standardmethoden, die auf Ressourcen angewendet werden können: GET, POST, PUT, DELETE, HEAD und OPTIONS.[2]

GET

Mit GET wird die Repräsentation einer Ressource abgefragt. Da nur lesend zugegriffen wird, kann eine Ressource auf dem Server nicht verändert werden.

POST

Mit POST kann eine neue Ressource mit einem vom Server bestimmten URI erstellt werden. Mit POST kann aber auch eine beliebige Verarbeitung auf dem Server angestoßen werden.

PUT

PUT wird verwendet, um eine Ressource, deren URI bereits bekannt ist, zu erstellen oder zu ändern.

---

2 Es existieren noch die HTTP-Methoden PATCH, CONNECT und TRACE. Diese werden hier nicht behandelt.

DELETE

Mit DELETE wird eine Ressource gelöscht.

HEAD

Im Unterschied zu GET wird mit HEAD keine Repräsentation zurückgeliefert, sondern nur Metadaten über die Ressource (Status-Code, Header). Hiermit kann z. B. die Existenz einer Ressource geprüft werden.

OPTIONS

OPTIONS liefert Informationen über eine Ressource, z. B. welche Repräsentationsformate unterstützt werden und welche Methoden erlaubt sind.

Diese Methoden können nach den folgenden Gesichtspunkten eingeordnet werden:

Sicher

Eine Methode heißt *sicher* (*safe*), wenn der Aufruf dieser Methode den Zustand der Ressource, auf die zugegriffen wird, nicht verändern kann. In diesem Sinne sind nur die Methoden GET, HEAD und OPTIONS sicher.

Idempotent

Eine Methode heißt *idempotent*, wenn ein mehrmaliges Aufrufen der Methode den gleichen Effekt bewirkt wie ein einmaliges. In diesem Sinne sind alle Methoden mit Ausnahme von POST idempotent.

Ein Beispiel

Ein Service verwaltet Textnachrichten, die vom Client erzeugt und abgefragt werden können.

Durch eine POST-Anfrage an den URL `http://www.test.de/messages` kann eine neue Nachricht erstellt werden:

```
POST /messages HTTP/1.1
Content-Type: application/json
Content-Length: 22

{"text":"Nachricht A"}
```

Service-Antwort:

```
HTTP/1.1 201 Created
Location: http://www.test.de/messages/1
```

Der Aufruf von `http://www.test.de/messages/1` führt zur GET-Anfrage

```
GET /messages/1 HTTP/1.1
Accept: application/json
```

mit der Antwort:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 74

{"id":1,"text":"Nachricht A","timestamp":"2019-02-
01T16:35:50.780903500Z"}
```

REST und SOAP im Vergleich

Im Gegensatz zu REST sind SOAP-basierte Web Services *operationsorientiert*, d. h. für jede Operation ist die Zugriffsschnittstelle operationsspezifisch. REST dagegen definiert immer dieselben oben aufgeführten Standardmethoden, um auf einer Ressource operieren zu können. REST ist *ressourcenorientiert*.

10.2 Ein erstes Beispiel

Ein erstes Beispiel soll zeigen, wie REST-Services in Java mit JAX-RS entwickelt werden können.

10.2.1 Die erforderlichen Bibliotheken

Wir nutzen das im Rahmen des Java Community Process entwickelte *Java API for RESTful Web Services* (*JAX-RS*). Das Open-Source-Projekt *Jersey*[3] stellt die Referenzimplementierung für JAX-RS bereit. Im Internet sind die *API-Dokumentation* und ein *User Guide* verfügbar. Wir nutzen hier die Version 2.28.

Zunächst werden die Bibliotheken vorgestellt, die hier und in späteren Beispielen verwendet werden. Diese sind auch im Begleitmaterial zu diesem Buch vorhanden.

Bibliothek	**Beschreibung**
libs/jaxrs	Jersey 2.28, Implementierung von JAX-RS 2.1
libs/grizzly	Grizzly HTTP Server 2.4.4 mit Jersey Container 2.28 (siehe auch Kapitel 7.2)

3 `https://jersey.github.io/`
`https://jersey.github.io/apidocs/latest/jersey/index.html`
`https://jersey.github.io/documentation/latest/index.html`
`https://projects.eclipse.org/projects/ee4j.jersey`

Bibliothek	Beschreibung
libs/activation	JavaBeans Activation Framework 1.2.0 (siehe auch Kapitel 9.2)
libs/jaxb	Java Architecture for XML Binding (JAXB) 2.3.0

libs/activation und *libs/jaxb* werden ausschließlich zur Laufzeit benötigt, wobei *libs/jaxb* nur zur Verwendung von XML bei der HTTP-Methode OPTIONS und beim Abruf von WADL (siehe später) erforderlich ist.

10.2.2 Der Web Service

→ Projekt messages

Das Projekt besteht aus den Modulen *service*, *client1* und *client2*. Wir stellen zunächst den Service vor.

Der Service `MessagesResource` verwaltet Nachrichten vom Typ `Message` zur Vereinfachung in einer `Map` im Hauptspeicher. Eine Message besteht aus einer eindeutigen Id, der eigentlichen Textnachricht und einem Zeitstempel.

```
package demo.messages;

import java.time.Instant;

public class Message {
   private int id;
   private String text;
   private Instant timestamp;

   public Message() {
   }

   public Message(int id, String text, Instant timestamp) {
      this.id = id;
      this.text = text;
      this.timestamp = timestamp;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }
```

```
    public Instant getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(Instant timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return "Message{" +
                "id=" + id +
                ", text='" + text + '\'' +
                ", timestamp=" + timestamp +
                '}';
    }
}
```

Ressourcen des Dienstes

Ressource	URI	Methode
Alle Nachrichten	`messages`	GET *), POST
Einzelne Nachricht	`messages/{id}`	GET *), PUT, DELETE
Die letzten 3 Nachrichten	`messages/limited?n=3`	GET

*) Alternative Repräsentationsformen: Text, HTML, JSON-Format

JAX-RS benutzt diverse Annotationen, um eine Service-Methode einer HTTP-Anfrage eindeutig zuordnen zu können. Diese Annotationen betreffen vor allem den Ressourcenpfad, die HTTP-Methode und den Content-Type der HTTP-Anfrage bzw. -Antwort.

Die folgende Klasse `MessageRessource` implementiert alle Service-Methoden. Erläuterungen finden Sie am Ende des Quellcodes.

```
package demo.messages;

import javax.ws.rs.*;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.net.URISyntaxException;
import java.time.Instant;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.Collectors;
```

```
@Path("messages")
public class MessageResource {
   private static Map<Integer, Message> map;
   private static AtomicInteger counter = new AtomicInteger();

   static {
      map = new ConcurrentHashMap<>();
      Instant instant = Instant.now();
      map.put(1, new Message(1, "Nachricht A", instant));
      map.put(2, new Message(2, "Nachricht B", instant.plusSeconds(1)));
      map.put(3, new Message(3, "Nachricht C", instant.plusSeconds(2)));
      counter.set(3);
   }

   @GET
   @Produces(MediaType.TEXT_PLAIN)
   public String getAllMessagesAsText() {
      StringBuilder stringBuilder = new StringBuilder();
      map.values().stream()
            .sorted((m1, m2) -> -m1.getTimestamp().compareTo(m2.getTimestamp()))
            .forEach(m -> stringBuilder.append(m.toString() + "\n"));
      return stringBuilder.toString();
   }

   @GET
   @Path("{id}")
   @Produces(MediaType.TEXT_PLAIN)
   public Response getMessageAsText(@PathParam("id") int id) {
      Message message = map.get(id);
      if (message == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      return Response.ok(message.toString()).build();
   }

   @GET
   @Produces(MediaType.TEXT_HTML)
   public String getAllMessagesAsHtml() {
      StringBuilder stringBuilder = new StringBuilder(
         "<html><head><meta charset=\"UTF-8\"></head><body><table border='1'>");
      map.values().stream()
            .sorted((m1, m2) -> -m1.getTimestamp().compareTo(m2.getTimestamp()))
            .forEach(m -> stringBuilder.append("<tr><td>")
                  .append(m.getId())
                  .append("</td><td>")
                  .append(m.getText())
                  .append("</td><td>")
                  .append(m.getTimestamp())
                  .append("</td></tr>")
            );
      stringBuilder.append("</table></body></html>");
      return stringBuilder.toString();
   }

   @GET
   @Path("{id}")
   @Produces(MediaType.TEXT_HTML)
   public Response getMessageAsHtml(@PathParam("id") int id) {
```

```
    Message message = map.get(id);
    if (message == null) {
        return Response.noContent().status(Response.Status.NOT_FOUND).build();
    }
    StringBuilder stringBuilder = new StringBuilder(
        "<html><head><meta charset=\"UTF-8\"></head><body>");
    stringBuilder
            .append(message.getId())
            .append("<br>")
            .append(message.getText())
            .append("<br>")
            .append(message.getTimestamp())
            .append("</body></html>");
    return Response.ok(stringBuilder.toString()).build();
}

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Message> getAllMessagesAsJson() {
    List<Message> list = map.values().stream()
            .sorted((m1, m2) -> -m1.getTimestamp().compareTo(m2.getTimestamp()))
            .collect(Collectors.toList());
    return list;
}

@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getMessageAsJson(@PathParam("id") int id) {
    Message message = map.get(id);
    if (message == null) {
        return Response.noContent().status(Response.Status.NOT_FOUND).build();
    }
    return Response.ok(message).build();
}

@GET
@Path("limited")
@Produces(MediaType.APPLICATION_JSON)
public List<Message> getAllMessagesAsJsonLimited(
        @QueryParam("n") @DefaultValue("3") int n) {

    n = n <= 0 ? 3 : n;
    StringBuilder stringBuilder = new StringBuilder();
    List<Message> list = map.values().stream()
            .sorted((m1, m2) -> -m1.getTimestamp().compareTo(m2.getTimestamp()))
            .limit(n)
            .collect(Collectors.toList());
    return list;
}

@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response newMessage(@Context UriInfo uriInfo, Message message) {
    int id = counter.incrementAndGet();
    message.setId(id);
    message.setTimestamp(Instant.now());
    map.put(id, message);
    URI location = uriInfo.getAbsolutePathBuilder()
```

```
            .path(String.valueOf(id)).build();
        return Response.created(location).build();
    }

    @PUT
    @Path("{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response updateMessage(@PathParam("id") int id, Message message) {
        if (map.get(id) == null) {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        } else {
            message.setId(id);
            message.setTimestamp(Instant.now());
            map.put(id, message);
            return Response.noContent().status(Response.Status.OK).build();
        }
    }

    @DELETE
    @Path("{id}")
    public Response deleteMessage(@PathParam("id") int id) {
        if (map.get(id) != null) {
            map.remove(id);
            return Response.noContent().status(Response.Status.OK).build();
        } else {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        }
    }

    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Response newMessageFromForm(
            @Context UriInfo uriInfo, @FormParam("text") String text) {
        int id = counter.incrementAndGet();
        Message message = new Message(id, text, Instant.now());
        map.put(id, message);
        URI location = uriInfo.getAbsolutePathBuilder()
            .path(String.valueOf(id)).build();
        return Response.created(location).status(Response.Status.MOVED_PERMANENTLY)
            .build();
    }
}
```

Die Map wird mit drei Messages initialisiert.

Die Annotation `@Path("messages")` vor der Definition der Klasse legt den Ressourcenpfad fest. Dieser kann für verschiedene Methoden ergänzt werden.

`MessagesResource` verwendet zur Kennzeichnung der HTTP-Methoden die Annotationen `@GET`, `@POST`, `@PUT` und `@DELETE`.

Die Annotation `@GET` vor der Methode `getAllMessagesAsText` bestimmt, dass eine GET-Anfrage zur Ausführung dieser Methode führt.

Hier stehen aber mehrere Methoden zur Auswahl. Welche Methode wird gewählt?

Die Annotation `@Produces` legt fest, welchen HTTP-Content-Type die GET-Antwort hat. Die Klasse `javax.ws.rs.core.MediaType` enthält String-Konstanten zur

Bezeichnung von Medientypen. Der Client kann die gewünschte Repräsentationsform über den Wert des HTTP-Headers `Accept` bestimmen. In unserem Beispiel kommt als Wert *text/plain*, *text/html* oder *application/json* infrage. Diese Form der Anforderung einer Repräsentation vonseiten des Clients wird als *Content Negotiation* bezeichnet.

Die einzige mit `@GET` annotierte Methode, die als Repräsentationsform *text/plain* liefert, ist `getAllMessagesAsText`. Hiermit ist diese dann eindeutig bestimmt.

Die Annotation `@Path("{id}")` enthält einen so genannten *Template-Parameter*, hier `{id}`. Lautet der (relative) Ressourcenpfad der Anfrage z. B. `message/1`, so wird 1 dem Parameter `id` zugeordnet.

Die Annotation `@PathParam` bei einem Methodenparameter injiziert den Wert des entsprechenden Template-Parameters. Im obigen Beispiel erhält der Parameter `id` den Wert 1.

Die Klasse `javax.ws.rs.core.Response` repräsentiert die HTTP-Antwort. Sie ermöglicht, außer den reinen Nutzdaten noch Status und Header-Informationen zu liefern.

Die hier verwendeten `Response`-Methoden sind:

`static Response.ResponseBuilder noContent()`
: erzeugt einen `ResponseBuilder` für eine "leere" Antwort.

`static Response.ResponseBuilder ok(Object entity)`
: erzeugt einen `ResponseBuilder`, der eine Repräsentation enthält.

`static Response.ResponseBuilder created(URI location)`
: erzeugt einen `ResponseBuilder` für die neu angelegte Ressource und setzt den Location-Header für die HTTP-Antwort.

`ResponseBuilder`-Methoden:

`Response.ResponseBuilder status(Response.Status status)`
: setzt den HTTP-Status für die Antwort.

`Response build()`
: erzeugt eine `Response`-Instanz.

Die Methode `getAllMessagesAsJsonLimited` im obigen Quellcode enthält zwei Annotationen zu einem Parameter:

`@QueryParam`
Der Wert des Query-Parameters (hier `n`) wird in den Java-Parameter injiziert.
Beispiel: `messages/limited?n=5`

`@DefaultValue`
Der angegebene Wert (hier 3) gilt, wenn ein Wert vom Client nicht mitgeliefert wurde.

Die Annotation `@Consumes` legt fest, welchen Medientyp die so annotierte Methode für die Daten in der HTTP-Anfrage erwartet. In unserem Beispiel:

`MediaType.APPLICATION_JSON` (JSON-Format),
`MediaType.APPLICATION_FORM_URLENCODED` (URL-codierte Daten).

Die Annotation `@Context` zu einem Methodenparameter (hier bei `newMessage` und `newMessageFromForm`) sorgt dafür, dass das Laufzeitsystem bestimmte Objekte injiziert.

Das Interface `javax.ws.rs.core.UriInfo` bietet verschiedene Methoden zur Abfrage von Informationen über den URI der Anfrage.

Die Anweisung

```
URI location = info.getAbsolutePathBuilder()
        .path(String.valueOf(id)).build();
```

in der Methode `newMessage` ermittelt den absoluten Pfad der Anfrage in einem `UriBuilder`, erweitert diesen um den Pfad mit der Id der Message und baut damit den kompletten Pfad auf.

Die Methode `newMessage` erzeugt eine neue Nachricht und speichert ihren Ressourcenpfad im Location-Header der HTTP-Antwort, sodass der Client nach Auswertung dieses Headers insbesondere die vom Service vergebene Id der neuen Nachricht erfährt.

Die in der Methode `newMessageFromForm` verwendete Annotation `@FormParam` erlaubt die Extraktion von per POST gesendeten Feldwerten eines HTML-Formulars. Ist der Client ein Browser, so führt der Status *Moved Permanently* (301) zur sofortigen Weiterleitung (Redirect mit GET) an die im Location-Header angegebene Adresse.

Die von `javax.ws.rs.core.Application` abgeleitete Klasse `MyApplication` sorgt für die Registrierung der JAX-RS-Ressourcen beim Server.

```
package demo.messages;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApplication extends Application {
   private Set<Object> singletons = new HashSet<>();
   private Set<Class<?>> classes = new HashSet<>();

   public MyApplication() {
      singletons.add(new MessageResource());
   }

   @Override
   public Set<Class<?>> getClasses() {
      return classes;
   }
```

```
   @Override
   public Set<Object> getSingletons() {
      return singletons;
   }
}
```

Die Methode `getClasses()` liefert eine Menge von JAX-RS-Ressourcenklassen. Für jede HTTP-Anfrage, die sich an eine dieser Klassen richtet, wird *eine neue Instanz* dieser Klasse vom Laufzeitsystem erzeugt.

Die alternative Methode `getSingletons()` liefert eine Menge von bereits erstellten Instanzen. Diese Instanzen werden im Gegensatz zum oben beschriebenen Verhalten für alle Anfragen an die jeweilige Klasse beibehalten.

Zur Bereitstellung des Service benutzen wir den erweiterten Grizzly HTTP Server:

```
import demo.messages.MyApplication;
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.grizzly.http.server.StaticHttpHandler;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);

      // listen on every available network interface
      URI baseUri = new URI("http://0.0.0.0:8080/demo/rest");
      ResourceConfig config = ResourceConfig
         .forApplicationClass(MyApplication.class);
      HttpServer server = GrizzlyHttpServerFactory
         .createHttpServer(baseUri, config);
      StaticHttpHandler handler = new StaticHttpHandler("web");
      handler.setFileCacheEnabled(false);
      server.getServerConfiguration().addHttpHandler(handler, "/demo");

      System.out.println("http://localhost:8080/demo/");
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      server.shutdownNow();
   }
}
```

Der *Context Root* der Web-Applikation ist `demo`. Der Basis-URI für alle Ressourcen ist `demo/rest`.

Damit der Server auch statische Ressourcen (wie HTML-Seiten, CSS-Stylesheets und JavaScript-Dateien) bereitstellen kann, kann er mit einem statischen HTTP-

Handler zusätzlich konfiguriert werden. Die statischen Ressourcen liegen hier im Verzeichnis *web*.

Hierbei handelt es sich um eine kleine Webanwendung, die den bereitgestellten REST-Service nutzt. Diese befindet sich im Begleitmaterial zu diesem Buch, sie wird hier allerdings nicht beschrieben.

Der Service kann auf Konsolenebene wie folgt kompiliert werden:

```
javac -encoding UTF-8 -d out/production/service
-cp ../../libs/grizzly/*;../../libs/jaxrs/*
service/src/demo/messages/*.java service/src/*.java
```

Aufruf des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/*;../../libs/jaxb/* Server
```

Ruft man die Adresse

`http://localhost:8080/demo/rest/messages`

im Browser auf, erhält man die HTML-Tabelle aller gespeicherten Nachrichten. Es wird also die Methode `getAllMessagesAsHtml` der Klasse `MessageResource` ausgeführt, da der Browser in der Regel *text/html* als ersten *Accept-Header* mitschickt.

10.2.3 Test mit cURL

Mit dem Kommandozeilenprogramm `cURL` (Client for URLs) kann der Service getestet werden.[4]

```
curl -i -H "Accept: text/plain" http://localhost:8080/demo/rest/messages/1
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 75

Message{id=1, text='Nachricht A', timestamp=2019-02-02T14:23:55.609266600Z}

curl -i -H "Accept: text/plain" http://localhost:8080/demo/rest/messages
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 228

Message{id=3, text='Nachricht C', timestamp=2019-02-02T14:23:57.609266600Z}
Message{id=2, text='Nachricht B', timestamp=2019-02-02T14:23:56.609266600Z}
Message{id=1, text='Nachricht A', timestamp=2019-02-02T14:23:55.609266600Z}

curl -i -H "Accept: text/html" http://localhost:8080/demo/rest/messages/1
```

4 https://curl.haxx.se

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 111

<html><head><meta charset="UTF-8"></head><body>1<br>Nachricht A<br>2019-02-
02T14:23:55.609266600Z</body></html>
```

curl -i -H "Accept: text/html" http://localhost:8080/demo/rest/messages

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 321

<html><head><meta charset="UTF-8"></head><body><table border='1'>
<tr><td>3</td><td>Nachricht C</td><td>2019-02-02T14:23:57.609266600Z</td></tr>
<tr><td>2</td><td>Nachricht B</td><td>2019-02-02T14:23:56.609266600Z</td></tr>
<tr><td>1</td><td>Nachricht A</td><td>2019-02-02T14:23:55.609266600Z</td></tr>
</table></body></html>
```

curl -i -H "Accept: application/json" http://localhost:8080/demo/rest/messages/1

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 74

{"id":1,"text":"Nachricht A","timestamp":"2019-02-02T14:23:55.609266600Z"}
```

curl -i -H "Accept: application/json" http://localhost:8080/demo/rest/messages

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 226

[{"id":3,"text":"Nachricht C","timestamp":"2019-02-02T14:23:57.609266600Z"},
{"id":2,"text":"Nachricht B","timestamp":"2019-02-02T14:23:56.609266600Z"},
{"id":1,"text":"Nachricht A","timestamp":"2019-02-02T14:23:55.609266600Z"}]
```

curl -i -H "Content-Type: application/json"
http://localhost:8080/demo/rest/messages -d @data.json

```
HTTP/1.1 201 Created
Location: http://localhost:8080/demo/rest/messages/4
Content-Length: 0
```

curl -i -H "Content-Type: application/json" -X PUT
http://localhost:8080/demo/rest/messages/1 -d @data.json

```
HTTP/1.1 200 OK
Content-Length: 0
```

curl -i -X DELETE http://localhost:8080/demo/rest/messages/2

```
HTTP/1.1 200 OK
Content-Length: 0
```

curl -i -H "Accept: text/plain" http://localhost:8080/demo/rest/messages

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 234

Message{id=1, text='Nachricht curl', timestamp=2019-02-02T14:24:28.304753300Z}
Message{id=4, text='Nachricht curl', timestamp=2019-02-02T14:24:27.820471800Z}
Message{id=3, text='Nachricht C', timestamp=2019-02-02T14:23:57.609266600Z}
```

```
curl -i -H "Accept: application/json"
http://localhost:8080/demo/rest/messages/limited?n=2
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 157

[{"id":1,"text":"Nachricht curl","timestamp":"2019-02-02T14:24:28.304753300Z"},
{"id":4,"text":"Nachricht curl","timestamp":"2019-02-02T14:24:27.820471800Z"}]

curl -i -d "text=Neue Nachricht" http://localhost:8080/demo/rest/messages
HTTP/1.1 301 Moved Permanently
Location: http://localhost:8080/demo/rest/messages/5
Content-Length: 0
```

Die Datei *data.json* hat den Inhalt:

```
{
  "text": "Nachricht curl"
}
```

Ebenso kann das grafische Tool *Postman* genutzt werden, um REST-Anfragen zu senden.[5]

10.2.4 Die Beschreibungssprache WADL

Der Aufruf von

```
<Basis-URI>/application.wadl
```

(im Beispiel `http://localhost:8080/demo/rest/application.wadl`) im Web Browser liefert die Beschreibung des REST-Service in einem XML-basierten Format: *Web Application Description Language* (*WADL*). Diese Beschreibung wird aus der Implementierung des Service automatisch generiert.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/"
      jersey:generatedBy="Jersey: 2.28 2019-01-25 15:18:13"/>
    <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="..."/>
    <grammars>
        <include href="application.wadl/xsd0.xsd">
            <doc title="Generated" xml:lang="en"/>
        </include>
    </grammars>
    <resources base="http://localhost:8080/demo/rest/">
        <resource path="messages">
            <method id="newMessageFromForm" name="POST">
                <request>
                    <representation mediaType="application/x-www-form-urlencoded">
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                name="text" style="query" type="xs:string"/>
                    </representation>
                </request>
```

5 https://www.getpostman.com

```
        <response>
            <representation mediaType="*/*"/>
        </response>
    </method>
    <method id="newMessage" name="POST">
        <request>
            <representation mediaType="application/json"/>
        </request>
        <response>
            <representation mediaType="*/*"/>
        </response>
    </method>
    <method id="getAllMessagesAsHtml" name="GET">
        <response>
            <representation mediaType="text/html"/>
        </response>
    </method>
    <method id="getAllMessagesAsJson" name="GET">
        <response>
            <representation mediaType="application/json"/>
        </response>
    </method>
    <method id="getAllMessagesAsText" name="GET">
        <response>
            <representation mediaType="text/plain"/>
        </response>
    </method>
    <resource path="{id}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
           name="id" style="template" type="xs:int"/>
        <method id="getMessageAsText" name="GET">
            <response>
                <representation mediaType="text/plain"/>
            </response>
        </method>
        <method id="getMessageAsHtml" name="GET">
            <response>
                <representation mediaType="text/html"/>
            </response>
        </method>
        <method id="getMessageAsJson" name="GET">
            <response>
                <representation mediaType="application/json"/>
            </response>
        </method>
        <method id="updateMessage" name="PUT">
            <request>
                <representation mediaType="application/json"/>
            </request>
            <response>
                <representation mediaType="*/*"/>
            </response>
        </method>
        <method id="deleteMessage" name="DELETE">
            <response>
                <representation mediaType="*/*"/>
            </response>
        </method>
    </resource>
```

```
            <resource path="limited">
                <method id="getAllMessagesAsJsonLimited" name="GET">
                    <request>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                name="n" style="query" type="xs:int" default="3"/>
                    </request>
                    <response>
                        <representation mediaType="application/json"/>
                    </response>
                </method>
            </resource>
        </resource>
    </resources>
</application>
```

10.2.5 Ein Java-Client

Mit Java-Bordmitteln können wir einen Java-Client realisieren. Die hier zentral verwendete Klasse ist `java.net.HttpURLConnection`. Um Objekte ins JSON-Format zu konvertieren und umgekehrt verwenden wir das *Java API for JSON Binding* aus Kapitel 2.

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;

public class Client1 {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";

   public static void main(String[] args) throws IOException {
      get("/messages/1", "text/plain");
      get("/messages", "text/plain");

      get("/messages/1");
      get("/messages");

      head("/messages/1");
      head("/messages");

      options("/messages/1");
      options("/messages");

      post("/messages", "Neue Nachricht");
      put("/messages/1", "Nachricht AA");
      delete("/messages/2");

      get("/messages/limited?n=2");
   }
```

```
private static void get(String uri, String mediaType) throws IOException {
   System.out.println("\n--- GET " + BASE_URI + uri + " " + mediaType);
   URL url = new URL(BASE_URI + uri);
   HttpURLConnection connection = (HttpURLConnection) url.openConnection();
   connection.setRequestMethod("GET");
   connection.setRequestProperty("Accept", mediaType);
   if (status(connection) == 200) {
      output(connection);
   }
   connection.disconnect();
}

private static void get(String uri) throws IOException {
   System.out.println("\n--- GET " + BASE_URI + uri);
   URL url = new URL(BASE_URI + uri);
   HttpURLConnection connection = (HttpURLConnection) url.openConnection();
   connection.setRequestMethod("GET");
   connection.setRequestProperty("Accept", "application/json");
   if (status(connection) == 200) {
      Jsonb jsonb = JsonbBuilder.create();
      if (uri.endsWith("messages") || uri.contains("limited")) {
         List<Message> list = jsonb.fromJson(connection.getInputStream(),
               new ArrayList<Message>() {
               }.getClass().getGenericSuperclass());
         for (Message message : list) {
            System.out.println(message.getId() + ";" +
                     message.getText() + ";" + message.getTimestamp());
         }
      } else {
         Message message = jsonb.fromJson(
            connection.getInputStream(), Message.class);
         System.out.println(message.getId() + ";" +
               message.getText() + ";" + message.getTimestamp());
      }
   }
   connection.disconnect();
}

private static void head(String uri) throws IOException {
   System.out.println("\n--- HEAD " + BASE_URI + uri);
   URL url = new URL(BASE_URI + uri);
   HttpURLConnection connection = (HttpURLConnection) url.openConnection();
   connection.setRequestMethod("HEAD");
   status(connection);
   connection.disconnect();
}

private static void options(String uri) throws IOException {
   System.out.println("\n--- OPTIONS " + BASE_URI + uri);
   URL url = new URL(BASE_URI + uri);
   HttpURLConnection connection = (HttpURLConnection) url.openConnection();
   connection.setRequestMethod("OPTIONS");
   connection.setRequestProperty("Accept", "*/*");
   output(connection);
   connection.disconnect();
}

private static void post(String uri, String text) throws IOException {
   System.out.println("\n--- POST " + BASE_URI + uri);
```

```
        URL url = new URL(BASE_URI + uri);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json");
        Message message = new Message();
        message.setText(text);
        Jsonb jsonb = JsonbBuilder.create();
        jsonb.toJson(message, connection.getOutputStream());
        if (status(connection) == 201) {
            String location = connection.getHeaderField("Location");
            System.out.println("Location: " + location);
        }
        connection.disconnect();
    }

    private static void put(String uri, String text) throws IOException {
        System.out.println("\n--- PUT " + BASE_URI + uri);
        URL url = new URL(BASE_URI + uri);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("PUT");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json");
        Jsonb jsonb = JsonbBuilder.create();
        Message message = new Message();
        message.setText(text);
        jsonb.toJson(message, connection.getOutputStream());
        status(connection);
        connection.disconnect();
    }

    private static void delete(String uri) throws IOException {
        System.out.println("\n--- DELETE " + BASE_URI + uri);
        URL url = new URL(BASE_URI + uri);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("DELETE");
        status(connection);
        connection.disconnect();
    }

    private static int status(HttpURLConnection connection) throws IOException {
        int code = connection.getResponseCode();
        String reason = connection.getResponseMessage();
        System.out.println("Status: " + code + " " + (reason == null ? "" : reason));
        return code;
    }

    private static void output(HttpURLConnection connection) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String line = reader.readLine();
        while (line != null) {
            System.out.println(line);
            line = reader.readLine();
        }
        reader.close();
    }
}
```

Aufruf des Clients:

```
java -cp out/production/client1;../../libs/json/* Client1
```

Ausgabe:

```
--- GET http://localhost:8080/demo/rest/messages/1 text/plain
Status: 200 OK
Message{id=1, text='Nachricht A', timestamp=2019-02-02T14:58:38.912145100Z}

--- GET http://localhost:8080/demo/rest/messages text/plain
Status: 200 OK
Message{id=3, text='Nachricht C', timestamp=2019-02-02T14:58:40.912145100Z}
Message{id=2, text='Nachricht B', timestamp=2019-02-02T14:58:39.912145100Z}
Message{id=1, text='Nachricht A', timestamp=2019-02-02T14:58:38.912145100Z}

--- GET http://localhost:8080/demo/rest/messages/1
Status: 200 OK
1;Nachricht A;2019-02-02T14:58:38.912145100Z

--- GET http://localhost:8080/demo/rest/messages
Status: 200 OK
3;Nachricht C;2019-02-02T14:58:40.912145100Z
2;Nachricht B;2019-02-02T14:58:39.912145100Z
1;Nachricht A;2019-02-02T14:58:38.912145100Z

--- HEAD http://localhost:8080/demo/rest/messages/1
Status: 200 OK

--- HEAD http://localhost:8080/demo/rest/messages
Status: 200 OK

--- OPTIONS http://localhost:8080/demo/rest/messages/1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/"
      jersey:generatedBy="Jersey: 2.28 2019-01-25 15:18:13"/>
    <grammars>
        <include href="http://localhost:8080/demo/rest/application.wadl/xsd0.xsd">
            <doc title="Generated" xml:lang="en"/>
        </include>
    </grammars>
    <resources base="http://localhost:8080/demo/rest/">
        <resource path="messages/1">
            <method id="getMessageAsJson" name="GET">
                <response>
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <method id="updateMessage" name="PUT">
                <request>
                    <representation mediaType="application/json"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
            <method id="getMessageAsHtml" name="GET">
```

```
                <response>
                    <representation mediaType="text/html"/>
                </response>
            </method>
            <method id="getMessageAsText" name="GET">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
            <method id="deleteMessage" name="DELETE">
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
        </resource>
    </resources>
</application>

--- OPTIONS http://localhost:8080/demo/rest/messages
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">

   entspricht der Ausgabe (WADL) in Kapitel 10.2.4

</application>

--- POST http://localhost:8080/demo/rest/messages
Status: 201 Created
Location: http://localhost:8080/demo/rest/messages/4

--- PUT http://localhost:8080/demo/rest/messages/1
Status: 200 OK

--- DELETE http://localhost:8080/demo/rest/messages/2
Status: 200 OK

--- GET http://localhost:8080/demo/rest/messages/limited?n=2
Status: 200 OK
1;Nachricht AA;2019-02-02T15:05:48.509630400Z
4;Neue Nachricht;2019-02-02T15:05:48.494007700Z
```

Das Begleitmaterial zu diesem Buch enthält auch ein mit dem mit Java 11 neu eingeführten *Http-Client-API* analog zu `Client1` realisiertes Programm.

10.2.6 Das Client-API von JAX-RS

JAX-RS bietet ein Client-API, dessen Implementierung intern Objekte ins JSON-Format konvertiert und umgekehrt.

Mit Hilfe der Klassenmethode `javax.ws.rs.client.ClientBuilder.newClient()` wird eine Instanz vom Typ des Interface `javax.ws.rs.client.Client` erzeugt.

Die `Client`-Methode

```
WebTarget target(String uri)
```

liefert zum vorgegebenen URI `uri` ein Ressourcenziel.

Das Interface `javax.ws.rs.client.WebTarget` enthält u. a. die Methode

```
Invocation.Builder request(),
```

die den Aufbau einer Anfrage startet.

Das Interface `javax.ws.rs.client.Invocation.Builder` bietet diverse Methoden. Hier nutzen wir:

`Invocation.Builder accept(String... mediaTypes)`
: legt den gewünschten Medientyp fest.

`Response get()`
: ruft die GET-Methode auf und liefert die Antwort zurück.

Die Klasse `javax.ws.rs.core.Response` repräsentiert die HTTP-Antwort.

Die `Response`-Methode

```
<T> T readEntity(Class<T> entityType)
```

stellt den Inhalt im entsprechenden Java-Typ bereit.

Die Klasse `javax.ws.rs.client.Entity<T>` repräsentiert die Nutzdaten, die gesendet werden.

`static <T> Entity<T> entity(T entity, MediaType mediaType)`
: erzeugt ein `Entity`-Objekt mit den Daten `entity` und dem Content-Type `mediaType`.

Weitere `Response`-Methoden:

`URI getLocation()`
: liefert den Wert des Location-Headers.

`int getStatus()`
: liefert den HTTP-Status der Antwort.

`Response.StatusType getStatusInfo()`
: liefert die komplette Status-Information der Antwort.

Die Klasse `javax.ws.rs.core.GenericType<T>` repräsentiert einen generischen Entity-Typ `T` (vgl. auch Kapitel 2.2 "Deserialisierung").

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.List;

public class Client2 {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      get("/messages/1", MediaType.TEXT_PLAIN);
```

```
    get("/messages", MediaType.TEXT_PLAIN);

    get("/messages/1");
    get("/messages");

    head("/messages/1");
    head("/messages");

    options("/messages/1");
    options("/messages");

    post("/messages", "Neue Nachricht");
    put("/messages/1", "Nachricht AA");
    delete("/messages/2");

    get("/messages/limited?n=2");
}

private static void get(String uri, String mediaType) {
    System.out.println("\n--- GET " + BASE_URI + uri + " " + mediaType);
    WebTarget target = client.target(BASE_URI + uri);
    Response response = target.request().accept(mediaType).get();
    if (status(response) == 200)
        System.out.println(response.readEntity(String.class));
}

private static void get(String uri) {
    System.out.println("\n--- GET " + BASE_URI + uri);
    WebTarget target = client.target(BASE_URI + uri);
    Response response = target.request()
        .accept(MediaType.APPLICATION_JSON).get();
    if (status(response) == 200) {
        if (uri.endsWith("messages") || uri.contains("limited")) {
            List<Message> list = response.readEntity(
                new GenericType<List<Message>>() {});
            for (Message message : list) {
                System.out.println(message.getId() + ";" +
                        message.getText() + ";" + message.getTimestamp());
            }
        } else {
            Message message = response.readEntity(Message.class);
            System.out.println(message.getId() + ";" +
                   message.getText() + ";" + message.getTimestamp());
        }
    }
}

private static void head(String uri) {
    System.out.println("\n--- HEAD " + BASE_URI + uri);
    WebTarget target = client.target(BASE_URI + uri);
    Response response = target.request().head();
    status(response);
}

private static void options(String uri) {
    System.out.println("\n--- OPTIONS " + BASE_URI + uri);
    WebTarget target = client.target(BASE_URI + uri);
    Response response = target.request()
        .accept(MediaType.WILDCARD_TYPE).options();
```

```
        System.out.println(response.readEntity(String.class));
    }

    private static void post(String uri, String text) {
        System.out.println("\n--- POST " + BASE_URI + uri);
        WebTarget target = client.target(BASE_URI + uri);
        Message message = new Message();
        message.setText(text);
        Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
        Response response = target.request().post(entity);
        if (status(response) == 201) {
            String location = response.getLocation().toString();
            System.out.println("Location: " + location);
        }
    }

    private static void put(String uri, String text) {
        System.out.println("\n--- PUT " + BASE_URI + uri);
        WebTarget target = client.target(BASE_URI + uri);
        Message message = new Message();
        message.setText(text);
        Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
        Response response = target.request().put(entity);
        status(response);
    }

    private static void delete(String uri) {
        System.out.println("\n--- DELETE " + BASE_URI + uri);
        WebTarget target = client.target(BASE_URI + uri);
        Response response = target.request().delete();
        status(response);
    }

    private static int status(Response response) {
        int code = response.getStatus();
        String reason = response.getStatusInfo().getReasonPhrase();
        System.out.println("Status: " + code + " " + reason);
        return code;
    }
}
```

Aufruf des Clients:

```
java -cp out/production/client2;../../libs/jaxrs/* Client2
```

Die Ausgabe entspricht der Ausgabe des Programms `Client1`.

10.3 CRUD-Operationen mit Datenbank

Im Folgenden entwickeln wir einen Service zur Verwaltung von Kontaktdaten in einer Datenbank. Hierzu benutzen wir das relationale Datenbanksystem H2.[6]

Wir betreiben die H2-Datenbank mit dem Namen *contacts_db* im eingebetteten Modus, d. h. Datenbank und Java-Anwendung laufen im selben Prozess. Der Zugriff auf die Datenbank erfolgt über das JDBC-API. Der JDBC-Treiber von H2

6 http://www.h2database.com

h2-x.x.x.jar muss zur Laufzeit des Programms in den Klassenpfad der Anwendung eingebunden sein.

Der URL der Datenbank lautet:

```
jdbc:h2:~/contacts_db
```

~ zeigt auf das User-Verzeichnis, z. B. `C:\Users\`*`username`* bei Windows 10.

→ Projekt contacts

Ein Kontakt besteht aus einer eindeutigen Id, einem Namen, einer Mailadresse und einem Kommentar:

```
package demo.contacts.model;

public class Contact {
   private int id;
   private String name;
   private String email;
   private String comment;

   public Contact() {
   }

   public Contact(String name, String email, String comment) {
      this.name = name;
      this.email = email;
      this.comment = comment;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }


   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
      this.email = email;
   }

   public String getComment() {
      return comment;
   }
```

```
    public void setComment(String comment) {
        this.comment = comment;
    }

    @Override
    public String toString() {
        return "Contact{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", comment='" + comment + '\'' +
                '}';
    }
}
```

Ressourcen des Dienstes

Ressource	URI	Methode
Alle Kontakte	`contacts`	GET, POST
Einzelner Kontakt	`contacts/{id}`	GET, PUT, DELETE
Suche über Teil des Namens	`contacts/search?name=...`	GET

Der Datenaustausch erfolgt im JSON-Format.

Der Service

Die Datenbankzugriffsmethoden `create`, `insert`, `update`, `delete` und Methoden zur Auswertung (`select`) befinden sich alle in der Klasse `DBManager`.

Wir setzen hier grundlegende SQL- und JDBC-Kenntnisse voraus und erläutern den Quellcode deshalb nicht.

```
package demo.contacts.db;

import demo.contacts.model.Contact;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class DBManager {
   private static Connection getConnection()
         throws SQLException, ClassNotFoundException {
      Class.forName("org.h2.Driver");
      return DriverManager.getConnection("jdbc:h2:~/contacts_db");
   }

   public static void create() {
      try (Connection con = getConnection()) {
         String sql = "create table if not exists contact (" +
               "id int identity," +
               "name varchar(50)," +
               "email varchar(50)," +
               "comment varchar(100)," +
               "primary key (id))";
         Statement stmt = con.createStatement();
         stmt.executeUpdate(sql);
         stmt.close();
      } catch (Exception e) {
         Logger.getAnonymousLogger().log(Level.SEVERE, "create: "
            + e.getMessage());
         throw new RuntimeException(e);
      }
   }

   public static int insert(Contact contact) {
      try (Connection con = getConnection()) {
         String sql = "insert into contact (name, email, comment) values (?,?,?)";
         PreparedStatement stmt = con.prepareStatement(
            sql, Statement.RETURN_GENERATED_KEYS);
         stmt.setString(1, contact.getName());
         stmt.setString(2, contact.getEmail());
         stmt.setString(3, contact.getComment());
         stmt.executeUpdate();
         ResultSet rs = stmt.getGeneratedKeys();
         int id = 0;
         if (rs.next())
            id = rs.getInt(1);
         stmt.close();
         return id;
      } catch (Exception e) {
         Logger.getAnonymousLogger().log(Level.SEVERE, "insert: "
            + e.getMessage());
         throw new RuntimeException(e);
      }
   }

   public static int update(int id, Contact contact) {
      try (Connection con = getConnection()) {
         String sql =
            "update contact set name = ?, email = ?, comment = ? where id = ?";
         PreparedStatement stmt = con.prepareStatement(sql);
         stmt.setString(1, contact.getName());
         stmt.setString(2, contact.getEmail());
         stmt.setString(3, contact.getComment());
         stmt.setInt(4, id);
         int n = stmt.executeUpdate();
         stmt.close();
         return n;
```

```
    } catch (Exception e) {
        Logger.getAnonymousLogger().log(Level.SEVERE, "update: "
            + e.getMessage());
        throw new RuntimeException(e);
    }
}

public static int delete(int id) {
    try (Connection con = getConnection()) {
        String sql = "delete from contact where id = ?";
        PreparedStatement stmt = con.prepareStatement(sql);
        stmt.setInt(1, id);
        int n = stmt.executeUpdate();
        stmt.close();
        return n;
    } catch (Exception e) {
        Logger.getAnonymousLogger().log(Level.SEVERE, "delete: "
            + e.getMessage());
        throw new RuntimeException(e);
    }
}

public static List<Contact> query() {
    try (Connection con = getConnection()) {
        String sql = "select id, name, email, comment from contact order by name";
        PreparedStatement stmt = con.prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();
        List<Contact> list = new ArrayList<Contact>();
        while (rs.next()) {
            Contact c = new Contact();
            c.setId(rs.getInt(1));
            c.setName(rs.getString(2));
            c.setEmail(rs.getString(3));
            c.setComment(rs.getString(4));
            list.add(c);
        }
        rs.close();
        stmt.close();
        return list;
    } catch (Exception e) {
        Logger.getAnonymousLogger().log(Level.SEVERE, "query: " + e.getMessage());
        throw new RuntimeException(e);
    }
}

public static List<Contact> query(String name) {
    try (Connection con = getConnection()) {
        String sql = "select id, name, email, comment from contact " +
              "where name like ? order by name";
        PreparedStatement stmt = con.prepareStatement(sql);
        stmt.setString(1, "%" + name + "%");
        ResultSet rs = stmt.executeQuery();
        List<Contact> list = new ArrayList<Contact>();
        while (rs.next()) {
            Contact c = new Contact();
            c.setId(rs.getInt(1));
            c.setName(rs.getString(2));
            c.setEmail(rs.getString(3));
            c.setComment(rs.getString(4));
```

```
            list.add(c);
         }
         rs.close();
         stmt.close();
         return list;
      } catch (Exception e) {
         Logger.getAnonymousLogger().log(Level.SEVERE, "query(name): "
            + e.getMessage());
         throw new RuntimeException(e);
      }
   }

   public static Contact find(int id) {
      try (Connection con = getConnection()) {
         String sql = "select id, name, email, comment from contact where id = ?";
         PreparedStatement stmt = con.prepareStatement(sql);
         stmt.setInt(1, id);
         ResultSet rs = stmt.executeQuery();
         Contact contact = null;
         if (rs.next()) {
            contact = new Contact();
            contact.setId(rs.getInt(1));
            contact.setName(rs.getString(2));
            contact.setEmail(rs.getString(3));
            contact.setComment(rs.getString(4));
         }
         rs.close();
         stmt.close();
         return contact;
      } catch (Exception e) {
         Logger.getAnonymousLogger().log(Level.SEVERE, "find: " + e.getMessage());
         throw new RuntimeException(e);
      }
   }
}
```

Die in `MyApplication` registrierte Ressource `ContactListResource` enthält einen Konstruktor, der die DB-Tabelle *contact* in der Datenbank anlegt, sofern sie noch nicht existiert (siehe `DBManager.create()`). Man beachte, dass dieser Konstruktor nur ein einziges Mal mit dem Start des Servers aufgerufen wird.

```
package demo.contacts;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApplication extends Application {
   private Set<Object> singletons = new HashSet<>();
   private Set<Class<?>> classes = new HashSet<>();

   public MyApplication() {
      singletons.add(new ContactListResource());
   }
```

```
   @Override
   public Set<Class<?>> getClasses() {
      return classes;
   }

   @Override
   public Set<Object> getSingletons() {
      return singletons;
   }
}



package demo.contacts;

import demo.contacts.db.DBManager;
import demo.contacts.model.Contact;

import javax.ws.rs.Consumes;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.util.List;

@Path("contacts")
public class ContactListResource {
   public ContactListResource() {
      DBManager.create();
   }

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newContact(@Context UriInfo uriInfo, Contact contact) {
      try {
         int id = DBManager.insert(contact);
         URI location = uriInfo.getAbsolutePathBuilder().path(String.valueOf(id))
            .build();
         return Response.created(location).build();
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public List<Contact> getContacts() {
      try {
         return DBManager.query();
```

```
        } catch (RuntimeException e) {
            System.err.println(e);
            throw new WebApplicationException();
        }
    }

    @GET
    @Path("search")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Contact> search(@QueryParam("name") @DefaultValue("") String name) {
        try {
            return DBManager.query(name);
        } catch (RuntimeException e) {
            System.err.println(e);
            throw new WebApplicationException();
        }
    }

    @Path("{id}")
    public ContactResource getContact(@PathParam("id") int id) {
        return new ContactResource(id);
    }
}
```

`javax.ws.rs.WebApplicationException` ist eine von `RuntimeException` abgeleitete Klasse. Bei Auslösung einer solchen Exception wird eine HTTP-Antwort mit dem angegeben Status-Code zurückgegeben. Fehlt ein solcher Parameter wird der Status-Code 500 (Internal Server Error) zurückgegeben.

Die mit `@Path("{id}")` annotierte Methode getContact erzeugt eine neue Ressource vom Typ `ContactResource` und gibt diese als Ergebnis zurück. JAX-RS nutzt diese Ressource, um die Anfrage weiter zu bearbeiten.

Eine solche Ressource, die gewissermaßen Teil einer anderen Ressource ist, wird als *Subressource* bezeichnet. Diese muss in `MyApplication` nicht registriert werden.

```
package demo.contacts;

import demo.contacts.db.DBManager;
import demo.contacts.model.Contact;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class ContactResource {
    private int id;
```

```
   public ContactResource(int id) {
      this.id = id;
   }

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public Contact getContact() {
      Contact contact;
      try {
         contact = DBManager.find(id);
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
      if (contact == null) {
         throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
      return contact;
   }

   @PUT
   @Consumes(MediaType.APPLICATION_JSON)
   public Response updateContact(Contact contact) {
      try {
         int n = DBManager.update(id, contact);
         if (n == 0) {
            return Response.noContent().status(Response.Status.NOT_FOUND)
                   .build();
         } else {
            return Response.noContent().status(Response.Status.OK).build();
         }
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @DELETE
   public Response deleteContact() {
      try {
         int n = DBManager.delete(id);
         if (n == 0) {
            return Response.noContent().status(Response.Status.NOT_FOUND)
                   .build();
         } else {
            return Response.noContent().status(Response.Status.OK).build();
         }
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }
}



import demo.contacts.MyApplication;
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
```

```
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);

      URI baseUri = new URI("http://0.0.0.0:8080/demo/rest");
      ResourceConfig config = ResourceConfig
         .forApplicationClass(MyApplication.class);
      HttpServer server = GrizzlyHttpServerFactory.createHttpServer(
         baseUri, config);

      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      server.shutdownNow();
   }
}
```

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/*;../../libs/jaxb/*;../../libs/h2-1.4.197.jar Server
```

Der Client

Wir zeigen hier nur die mit JAX-RS entwickelte Variante:

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.io.IOException;
import java.util.List;

public class Client2 {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";
   private static Client client;

   public static void main(String[] args) throws Exception {
      client = ClientBuilder.newClient();

      Contact contact1 = new Contact(
         "Mustermann, Max", "max.mustermann@web.de", "Test");
      Contact contact2 = new Contact(
         "Musterfrau, Maria", "maria.musterfrau@web.de", "Test");

      int id1 = post(contact1);
      int id2 = post(contact2);
```

```
    listContacts();

    Contact c1 = getContact(id1);
    System.out.println(c1);

    Contact c2 = getContact(id2);
    System.out.println(c2);

    c1.setName("Mustermann, Otto");
    c1.setEmail("otto.mustermann@web.de");
    c1.setComment("geändert");
    put(c1);

    search("Mustermann");

    delete(id2);
    listContacts();
}

private static void listContacts() throws IOException {
    System.out.println("\n--- listContacts");
    WebTarget target = client.target(BASE_URI + "/contacts");
    Response response = target.request().accept(MediaType.APPLICATION_JSON)
        .get();
    if (status(response) == 200) {
        List<Contact> list = response.readEntity(
                                new GenericType<List<Contact>>() {});
        for (Contact contact : list) {
            System.out.println(contact);
        }
    }
}

private static Contact getContact(int id) throws IOException {
    System.out.println("\n--- getContact");
    WebTarget target = client.target(BASE_URI + "/contacts/" + id);
    Response response = target.request().accept(MediaType.APPLICATION_JSON)
        .get();
    Contact contact = null;
    if (status(response) == 200) {
        contact = response.readEntity(Contact.class);
    }
    return contact;
}

private static int post(Contact contact) throws IOException {
    System.out.println("\n--- post");
    WebTarget target = client.target(BASE_URI + "/contacts");
    Entity<Contact> entity = Entity.entity(contact, MediaType.APPLICATION_JSON);
    Response response = target.request().post(entity);
    if (status(response) == 201) {
        String location = response.getLocation().toString();
        System.out.println("Location: " + location);
        String id = location.substring(location.lastIndexOf("/") + 1);
        return Integer.parseInt(id);
    }
    return 0;
}
```

```
   private static void put(Contact contact) throws IOException {
      System.out.println("\n--- put");
      WebTarget target = client.target(BASE_URI + "/contacts/" + contact.getId());
      Entity<Contact> entity = Entity.entity(contact, MediaType.APPLICATION_JSON);
      Response response = target.request().put(entity);
      status(response);
   }

   private static void search(String name) throws IOException {
      System.out.println("\n--- search");
      WebTarget target = client.target(
         BASE_URI + "/contacts/search").queryParam("name", name);
      Response response = target.request().accept(MediaType.APPLICATION_JSON)
         .get();
      if (status(response) == 200) {
         List<Contact> list = response.readEntity(
                                 new GenericType<List<Contact>>() {});
         for (Contact contact : list) {
            System.out.println(contact);
         }
      }
   }

   private static void delete(int id) throws IOException {
      System.out.println("\n--- delete");
      WebTarget target = client.target(BASE_URI + "/contacts/" + id);
      Response response = target.request().delete();
      status(response);
   }

   private static int status(Response response) {
      int code = response.getStatus();
      String reason = response.getStatusInfo().getReasonPhrase();
      System.out.println("Status: " + code + " " + reason);
      return code;
   }
}
```

Die `WebTarget`-Methode

```
WebTarget queryParam(String name, Object... values)
```

erzeugt einen Query-Parameter.

Im Begleitmaterial ist das Skript *H2WebConsole.bat* im Verzeichnis *scripts* enthalten, mit dem die Datenbank verwaltet werden kann. Im Startdialog muss die Variante *Generic H2 (Embedded)* gewählt werden. Zu beachten ist, dass der Server nicht läuft.

Das Begleitmaterial zu diesem Buch enthält auch eine Android-App, die den Service nutzt.

10.4 REST-Services mit Apache Tomcat veröffentlichen

Wir nutzen hier *Apache Tomcat 9.0.14* in der eingebetteten Version (siehe auch Kapitel 7.5). und veröffentlichen den REST-Service aus Kapitel 10.3.

→ Projekt tomcat

```
import org.apache.catalina.startup.Tomcat;

import java.io.File;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      int port = 8080;
      String contextPath = "demo";
      String docBase = "web";

      Logger.getLogger("").setLevel(Level.SEVERE);

      Tomcat tomcat = new Tomcat();
      tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
      Context ctx = tomcat.addWebapp(contextPath,
         new File(docBase).getAbsolutePath());

      Connector con = new Connector();
      con.setPort(port);

      Service service = tomcat.getService();
      service.addConnector(con);

      tomcat.start();

      System.out.println("DocBase: " + ctx.getDocBase());
      String url = con.getScheme() + "://" +
            InetAddress.getLocalHost().getHostAddress() + ":" +
            con.getPort() + ctx.getPath();
      System.out.println("URL: " + url);
      System.out.println("ENTER stoppt den Server.");

      System.in.read();
      tomcat.stop();
      tomcat.destroy();
   }
}
```

Der *Context Root* der Web-Applikation ist `demo`. Der Basis-URI für alle Ressourcen soll `demo/rest` sein.

Hierzu muss der Applikationspfad *rest* mit Hilfe der Annotation `@ApplicationPath` für die Klasse `demo.contact.MyApplication` (siehe Projekt *contacts*) angegeben werden:

```
@ApplicationPath("rest")
public class MyApplication extends Application { ... }
```

Damit ist der Basis-URI `demo/rest` festgelegt.

Im Verzeichnis *web/WEB-INF/lib* wird die jar-Datei *contacts.jar*, die die Bytecodes des Services enthält, gespeichert:

```
web
└── WEB-INF
    └── lib
        └── contacts.jar
```

```
jar --create --file web/WEB-INF/lib/contacts.jar
-C ../contacts/out/production/service demo/contacts/
```

Der Server wird wie folgt gestartet:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.io=ALL-UNNAMED
--add-opens java.rmi/sun.rmi.transport=ALL-UNNAMED
-cp out/production/tomcat;../../libs/tomcat_embed/*;../../libs/jaxrs/*;
../../libs/activation/*;../../libs/jaxb/*;../../libs/h2-1.4.197.jar Server
```

10.5 Upload und Download von Dateien

Die Klasse `java.io.InputStream` kann in den Zugriffsmethoden des REST-Service zum Lesen von Anfrage (Input-Daten) und für die Antwort (Output-Daten) genutzt werden.

Im folgenden Beispiel handelt es sich um einen Service zum Upload und Download von Dateien beliebigen Inhalts.

→ Projekt files

Der Service

```
package demo.files;

import org.glassfish.jersey.media.multipart.FormDataBodyPart;
import org.glassfish.jersey.media.multipart.FormDataContentDisposition;
import org.glassfish.jersey.media.multipart.FormDataMultiPart;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.List;

@Path("")
public class FileResource {
   private final static String PATH = "D:/temp";

   @GET
   @Path("download/{filename}")
   @Produces(MediaType.APPLICATION_OCTET_STREAM)
   public InputStream download(@PathParam("filename") String filename) {
      try {
         return new FileInputStream(PATH + "/" + filename);
      } catch (FileNotFoundException e) {
         System.err.println(e.getMessage());
         throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
   }

   @PUT
   @Path("upload/{filename}")
   @Consumes(MediaType.APPLICATION_OCTET_STREAM)
   public Response upload(@PathParam("filename") String filename, InputStream is) {
      try {
         String filePath = PATH + "/" + filename;
         Files.copy(is, Paths.get(filePath), StandardCopyOption.REPLACE_EXISTING);
         System.out.println(filePath + " gespeichert");
         return Response.noContent().status(Response.Status.OK).build();
      } catch (IOException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @POST
   @Path("upload")
   @Consumes(MediaType.MULTIPART_FORM_DATA)
   public Response upload(FormDataMultiPart multiPart) {
      try {
         List<FormDataBodyPart> parts = multiPart.getFields("file");
         String files = "";
         for (FormDataBodyPart part : parts) {
            InputStream is = part.getValueAs(InputStream.class);
            FormDataContentDisposition disposition = part
               .getFormDataContentDisposition();
            String filePath = PATH + "/" + disposition.getFileName();
            Files.copy(is, Paths.get(filePath),
               StandardCopyOption.REPLACE_EXISTING);
            System.out.println(filePath + " gespeichert");
            files += disposition.getFileName() + " ";
         }
         return Response.ok(files).build();
      } catch (Exception e) {
```

```
            System.err.println(e);
            throw new WebApplicationException();
        }
    }
}
```

`MediaType.APPLICATION_OCTET_STREAM` legt fest, dass es sich um eine beliebige binäre Datei handelt.

JAX-RS bietet auch Unterstützung für die Verarbeitung von Dateien, die über ein Formular hochgeladen werden.

```
<form action="/demo/rest/upload" method="post" enctype="multipart/form-data">
      <input type="file" name="file" multiple>
      <button type="submit">Upload</button>
      <button type="reset">Reset</button>
</form>
```

Die Codierung der HTTP-Anfrage erfolgt gemäß dem Medientyp `multipart/form-data`.

Beispiel (Zwei kleine Textdateien werden hochgeladen):

Header:

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarye9DzwNt4YzY1NciX
```

Body:

```
------WebKitFormBoundarye9DzwNt4YzY1NciX
Content-Disposition: form-data; name="file"; filename="test1.txt"
Content-Type: text/plain

Hallo
------WebKitFormBoundarye9DzwNt4YzY1NciX
Content-Disposition: form-data; name="file"; filename="test2.txt"
Content-Type: text/plain

Welt
------WebKitFormBoundarye9DzwNt4YzY1NciX--
```

Die `FormDataMultiPart`-Methode

```
getFields
```

liefert zu einem Feldnamen eine Liste von `FormDataBodyPart`-Objekten.

Die `FormDataBodyPart`-Methode

```
getFormDataContentDisposition
```

liefert den Content-Disposition-Header `FormDataContentDisposition` und der Aufruf von

```
getValueAs(InputStream.class)
```

stellt den Dateiinhalt als `InputStream` zur Verfügung.

Im Konstruktor der Klasse `MyApplication` muss der *Multipart Provider* registriert werden:

```
public MyApplication() {
   singletons.add(new FileResource());
   singletons.add(new MultiPartFeature());
}
```

Das Begleitmaterial zu diesem Buch enthält auch eine etwas komfortablere Webanwendung.

Der Server wird wiederum mit *Grizzly* realisiert:

```
import demo.files.MyApplication;
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.grizzly.http.server.StaticHttpHandler;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
   public static void main(String[] args) throws Exception {
      Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);

      // listen on every available network interface
      URI baseUri = new URI("http://0.0.0.0:8080/demo/rest");
      ResourceConfig config = ResourceConfig
         .forApplicationClass(MyApplication.class);
      HttpServer server = GrizzlyHttpServerFactory
         .createHttpServer(baseUri, config);
      StaticHttpHandler handler = new StaticHttpHandler("web");
      handler.setFileCacheEnabled(false);
      server.getServerConfiguration().addHttpHandler(handler, "/demo");

      System.out.println("http://localhost:8080/demo/");
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      server.shutdownNow();
   }
}
```

Zur Compilierung und Ausführung wird zusätzlich die Bibliothek

libs/media_multipart

(*Jersey Media Multipart 2.28* und *MIME Streaming Extension 1.9.11*) benötigt. Sie befindet sich im Begleitmaterial zu diesem Buch.

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/*;../../libs/jaxb/*;../../libs/media_multipart/*
Server
```

Der Client

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

public class Client2 {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      upload("test1.pdf");
      download("test1.pdf");
   }

   private static void upload(String filename) {
      WebTarget target = client.target(BASE_URI + "/upload/" + filename);
      try {
         InputStream is = new FileInputStream(filename);
         Entity<InputStream> entity = Entity.entity(is,
            MediaType.APPLICATION_OCTET_STREAM);
         Response response = target.request().put(entity);
         status(response);
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }

   private static void download(String filename) {
      WebTarget target = client.target(BASE_URI + "/download/" + filename);
```

```
        Response response = target.request()
            .accept(MediaType.APPLICATION_OCTET_STREAM).get();
        if (status(response) == 200) {
            try {
                InputStream is = response.readEntity(InputStream.class);
                Files.copy(is, Paths.get(filename),
                    StandardCopyOption.REPLACE_EXISTING);
            } catch (IOException e) {
                System.err.println(e);
            }
        }
    }

    private static int status(Response response) {
        int code = response.getStatus();
        String reason = response.getStatusInfo().getReasonPhrase();
        System.out.println("Status: " + code + " " + reason);
        return code;
    }
}
```

Aufruf des Clients:

```
java -cp out/production/client2;../../libs/jaxrs/*;../../libs/activation/*
Client2
```

10.6 Zuverlässigkeit

Werden die HTTP-Methoden so verwendet, wie es in Kapitel 10.1 beschrieben ist, ist POST die einzige Methode, die nicht *idempotent* und damit prinzipiell nicht zuverlässig ist.

Besteht beim Client nach der Anfrage mit POST für die Neuanlage einer Ressource Unklarheit darüber, ob die Nachricht beim Server angekommen und verarbeitet worden ist und wird diese Anfrage wiederholt, kann im ungünstigsten Fall die Ressource zweimal erzeugt worden sein.

Zur Lösung dieser Problematik bestehen verschiedene Ansätze.[7]

10.6.1 PUT statt POST

Dieser Ansatz verwendet die idempotente Methode PUT. Allerdings muss der Client nun selbst den URI mit einem eindeutigen Schlüssel für die neue Ressource erzeugen. Dazu kann das UUID-Verfahren angewandt werden.

Der *UUID* (*Universally Unique Identifier*) besteht aus einer 16-Byte-Zahl, die mittels eines Zufallszahlenalgorithmus generiert wird. Die Wahrscheinlichkeit, dass

7 Siehe Tilkov, S. u. a.: REST und HTTP. dpunkt.verlag, 3. Auflage 2015, Kapitel 13.2

zwei gleiche UUIDs erzeugt werden, ist verschwindend gering. Der UUID wird hexadezimal notiert und ist in fünf Gruppen unterteilt.[8]

Beispiel:

```
d55b57bb-307d-423c-a11f-047a6ed40bc1
```

Ist die Ressource mit diesem URI noch nicht vorhanden, wird sie erzeugt, ansonsten aktualisiert.

Wir demonstrieren den Sachverhalt anhand eines einfachen REST-Service, der Aufträge verwaltet.

→ Projekt reliable1

```
package demo.order;

import java.time.LocalDate;

public class Order {
   private String id;
   private String customer;
   private String product;
   private int quantity;
   private LocalDateTime dateTime;

   public Order() {
   }

   public Order(String customer, String product, int quantity) {
      this.customer = customer;
      this.product = product;
      this.quantity = quantity;
   }

   public Order(String id, String customer, String product, int quantity) {
      this(customer, product, quantity);
      this.id = id;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public String getCustomer() {
      return customer;
   }

   public void setCustomer(String customer) {
      this.customer = customer;
   }
```

8 https://tools.ietf.org/html/rfc4122

```
   public String getProduct() {
      return product;
   }

   public void setProduct(String product) {
      this.product = product;
   }

   public int getQuantity() {
      return quantity;
   }

   public void setQuantity(int quantity) {
      this.quantity = quantity;
   }

   public LocalDateTime getDateTime() {
      return dateTime;
   }

   public void setDateTime(LocalDateTime dateTime) {
      this.dateTime = dateTime;
   }

   @Override
   public String toString() {
      return "Order{" +
             "id='" + id + '\'' +
             ", customer='" + customer + '\'' +
             ", product='" + product + '\'' +
             ", quantity=" + quantity +
             ", dateTime=" + dateTime +
             '}';
   }
}



package demo.order;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.time.LocalDate;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Path("orders")
public class OrderResource {
   private static Map<String, Order> map = new ConcurrentHashMap<>();

   @PUT
   @Path("{id}")
   @Consumes(MediaType.APPLICATION_JSON)
```

```
   public Response putOrder(@PathParam("id") String id, Order order) {
      order.setDateTime(LocalDateTime.now());
      map.put(id, order);
      if (map.get(id) == null) {
         return Response.noContent().status(Response.Status.CREATED).build();
      } else {
         return Response.noContent().status(Response.Status.OK).build();
      }
   }

   @GET
   @Path("{id}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response getOrder(@PathParam("id") String id) {
      Order order = map.get(id);
      if (order == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      } else {
         return Response.ok(order).build();
      }
   }
}


import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.UUID;

public class JerseyClient {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      UUID uuid = UUID.randomUUID();
      Order order = new Order(uuid.toString(), "4711", "P001", 10);
      put(order);

      order = get(uuid);
      System.out.println(order);

      order.setQuantity(20);
      put(order);

      order = get(uuid);
      System.out.println(order);
   }

   private static void put(Order order) {
      WebTarget target = client.target(BASE_URI + "/orders/" + order.getId());
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().put(entity);
      status(response);
   }
```

```
   private static Order get(UUID uuid) {
      WebTarget target = client.target(BASE_URI + "/orders/" + uuid);
      Response response = target.request().get();
      Order order = null;
      if (status(response) == 200) {
         order = response.readEntity(Order.class);
      }
      return order;
   }

   private static int status(Response response) {
      int code = response.getStatus();
      String reason = response.getStatusInfo().getReasonPhrase();
      System.out.println("Status: " + code + " " + reason);
      return code;
   }
}
```

10.6.2 Kombination von POST und PUT

Das Anlegen einer neuen Ressource läuft hier in mehreren Schritten ab:

- Der Client legt per POST auf dem Server eine neue, "leere" Ressource an.
- Der Server liefert den URI der neuen Ressource im Location-Header.
- Der Client übermittelt nun die eigentlichen Daten per PUT an diesen URI.
- Der Server aktualisiert die Ressource.

POST ohne anschließendes PUT würde beim Server eine ungenutzte, "leere" Ressource übrig lassen. Der Server muss deshalb diese "leeren" Ressourcen in regelmäßigen Abständen entfernen.

Im Vergleich zu einem einzelnen POST oder dem vorhergehenden Verfahren erhöht sich die Anzahl der Netzwerkinteraktionen.

→ Projekt reliable2

```
package demo.order;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.time.LocalDate;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
```

```
@Path("orders")
public class OrderResource {
   private static Map<String, Order> map = new ConcurrentHashMap<>();
   private static AtomicInteger counter = new AtomicInteger();

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response emptyOrder(@Context UriInfo uriInfo, Order order) {
      String id = String.valueOf(counter.incrementAndGet());
      map.put(id, order);
      URI location = uriInfo.getAbsolutePathBuilder().path(id).build();
      return Response.created(location).build();
   }

   @PUT
   @Path("{id}")
   @Consumes(MediaType.APPLICATION_JSON)
   public Response updateOrder(@PathParam("id") String id, Order order) {
      if (map.get(id) == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      order.setId(id);
      order.setDateTime(LocalDateTime.now());
      map.put(id, order);
      return Response.noContent().status(Response.Status.OK).build();
   }

   @GET
   @Path("{id}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response getOrder(@PathParam("id") String id) {
      Order order = map.get(id);
      if (order == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      } else {
         return Response.ok(order).build();
      }
   }

   @DELETE
   public void clean() {
      for (String key : map.keySet()) {
         Order order = map.get(key);
         String id = order.getId();
         if (id == null) {
            map.remove(key);
            System.out.println("Eintrag zu " + key + " wurde entfernt.");
         }
      }
   }
}
```

POST legt eine neue, aber "leere" Ressource an und liefert den URI dieser Ressource an den Client zurück. Mit PUT wird diese Ressource "gefüllt", insbesondere wird das Feld `id` von `Order` belegt.

Die Methode `clean` sorgt dafür, alle leeren Ressourcen zu entfernen.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class JerseyClient {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      Order order = new Order("4711", "P001", 10);

      String uri = post();
      System.out.println(uri);

      put(uri, order);

      order = get(uri);
      System.out.println(order);

      // Diese neue Ressource bleibt leer.
      uri = post();
      System.out.println(uri);
   }

   private static String post() {
      WebTarget target = client.target(BASE_URI + "/orders");
      Entity<Order> entity = Entity.entity(
         new Order(), MediaType.APPLICATION_JSON);
      Response response = target.request().post(entity);
      String location = null;
      if (status(response) == 201) {
         location = response.getLocation().toString();
      }
      return location;
   }

   private static void put(String uri, Order order) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(uri);
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().put(entity);
      status(response);
   }

   private static Order get(String uri) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(uri);
      Response response = target.request().get();
      Order order = null;
      if (status(response) == 200) {
         order = response.readEntity(Order.class);
      }
      return order;
   }
```

```
   private static int status(Response response) {
      int code = response.getStatus();
      String reason = response.getStatusInfo().getReasonPhrase();
      System.out.println("Status: " + code + " " + reason);
      return code;
   }
}
```

Mit der folgenden REST-Anfrage wird die Methode `clean` aus `OrderResource` aufgerufen.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class CleanClient {
   private static final String BASE_URI = "http://localhost:8080/demo";

   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/orders");
      Response response = target.request().delete();
      int status = response.getStatus();
      System.out.println("Status: " + status + " "
         + response.getStatusInfo().getReasonPhrase());
   }
}
```

10.7 Parallelzugriffe

Wenn mehrere Clients zur selben Zeit dieselbe Ressource aktualisieren wollen, können Probleme (*Lost Update*) auftreten.

Beispiel:

> Client A liest den Auftrag 1 mit dem *quantity*-Wert 10 in der Absicht diesen zu ändern. Client B liest ebenfalls den Auftrag 1 und erhöht die Menge von 10 um 1 auf 11. Danach erhöht Client A die Menge um 2. Die Menge enthält am Ende den Wert 12 und nicht 13. Wer zuletzt schreibt, gewinnt!

Bei relationalen Datenbanksystemen werden üblicherweise derartige Aktionen in einer Datenbank-Transaktion ausgeführt. Je nach verwendetem Sperrmechanismus können dann solche Phänomene verhindert werden.

In unserem REST-Beispiel können wir den *Lost Update* durch den Einsatz einer *optimistischen Nebenläufigkeitskontrolle* verhindern:[9]

9 Siehe Tilkov, S. u. a.: REST und HTTP. dpunkt.verlag, 3. Auflage 2015, Kapitel 13.4

Zu diesem Zweck wird ein Hashwert vom Server erzeugt und im Header *ETag* (*Entity-Tag*) gesendet. Die Klasse `javax.ws.rs.core.EntityTag` repräsentiert den *ETag-Header*. Zur Berechnung des Hashwerts können verschiedene Algorithmen eingesetzt werden: MD5, SHA-1, SHA-256 oder SHA-512.

GET liefert nun neben der Repräsentation auch das aktuelle *ETag* der Ressource. PUT sendet dieses *ETag* in einem *If-Match-Header*. Der Service vergleicht nun das aktuelle *ETag* der zu ändernden Ressource mit dem gerade erhaltenen *ETag*. Stimmen beide überein, wird die Änderung vollzogen, ansonsten erhält der Client die Fehlermeldung *412 Precondition Failed*.

→ Projekt parallel

Die Klasse `Order` wurde dem Beispiel in Kapitel 10.6 entnommen.

```
package demo.order;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.time.LocalDateTime;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

@Path("orders")
public class OrderResource {
   private static Map<String, Order> map = new ConcurrentHashMap<String, Order>();
   private static AtomicInteger counter = new AtomicInteger();

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newOrder(@Context UriInfo uriInfo, Order order) {
      String id = String.valueOf(counter.incrementAndGet());
      order.setId(id);
      order.setDateTime(LocalDateTime.now());
      map.put(id, order);
      URI location = uriInfo.getAbsolutePathBuilder().path(id).build();
      return Response.created(location).build();
   }
```

```
   @PUT
   @Path("{id}")
   @Consumes(MediaType.APPLICATION_JSON)
   public Response updateOrder(
         @HeaderParam("If-Match") String tag,
         @PathParam("id") String id, Order order) {
      Order oldOrder = map.get(id);
      if (oldOrder == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      String digest = computeDigest(oldOrder);
      if (tag.equals("\"" + digest + "\"")) {
         order.setId(id);
         order.setDateTime(LocalDateTime.now());
         map.put(id, order);
         return Response.noContent().status(Response.Status.OK).build();
      }
      return Response.noContent().status(Response.Status.PRECONDITION_FAILED)
         .build();
   }

   @GET
   @Path("{id}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response getOrder(@PathParam("id") String id) {
      Order order = map.get(id);
      if (order == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      String digest = computeDigest(order);
      return Response.ok(order).tag(digest).build();
   }

   private static String computeDigest(Order order) {
      String content = order.getId() + order.getCustomer() + order.getProduct()
            + order.getQuantity() + order.getDateTime();
      try {
         MessageDigest md = MessageDigest.getInstance("MD5");
         md.update(content.getBytes());
         byte[] digest = md.digest();
         StringBuilder sb = new StringBuilder();
         for (byte b : digest) {
            sb.append(String.format("%02x", b));
         }
         return sb.toString();
      } catch (NoSuchAlgorithmException e) {
         return "";
      }
   }
}
```

Der erste Parameter der Methode `updateOrder` enthält das *ETag* inkl. Anführungszeichen. Dieser Header-Wert wird über `@HeaderParam("If-Match")` automatisch injiziert.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class PostClient {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";

   public static void main(String[] args) {
      Order order = new Order("4711", "P001", 10);
      String uri = post(order);
      System.out.println(uri);
   }

   private static String post(Order order) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/orders");
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().post(entity);
      int code = response.getStatus();
      System.out.println("Status: " + code);
      String location = null;
      if (code == 201) {
         location = response.getLocation().toString();
      }
      return location;
   }
}


import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.Scanner;

public class UpdateClient {
   private static String BASE_URI = "http://localhost:8080/demo/rest";
   private static Client client;
   private static String tag;

   public static void main(String[] args) {
      if (args.length > 0 && args[0].equals("-m"))
         BASE_URI = "http://localhost:8888/demo/rest";

      client = ClientBuilder.newClient();

      Scanner sc = new Scanner(System.in);
      System.out.print("> Id: ");
      String id = sc.nextLine();
      Order order = get(BASE_URI + "/orders/" + id);

      if (order != null) {
         System.out.println(order);
         System.out.print("> + ");
```

```
         try {
            int quantity = order.getQuantity();
            quantity += Integer.parseInt(sc.nextLine());
            order.setQuantity(quantity);
            put(BASE_URI + "/orders/" + id, order, tag);
            order = get(BASE_URI + "/orders/" + id);
            System.out.println(order);
         } catch (NumberFormatException e) {
            System.err.println("Keine Zahl");
         }
      }

      sc.close();
   }

   private static Order get(String uri) {
      WebTarget target = client.target(uri);
      Response response = target.request().get();
      Order order = null;
      int code = response.getStatus();
      if (code == 200) {
         tag = response.getHeaderString("Etag");
         System.out.println("ETag: " + tag);
         order = response.readEntity(Order.class);
      } else {
         System.out.println("Status: " + code);
      }
      return order;
   }

   private static void put(String uri, Order order, String tag) {
      WebTarget target = client.target(uri);
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().header("If-Match", tag).put(entity);
      int code = response.getStatus();
      if (code != 200)
         System.out.println("Status: " + code);
   }
}
```

Testablauf

Server starten und `PostClient` ausführen:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
PostClient
```

Ausgabe:

```
Status: 201
http://localhost:8080/demo/rest/orders/1
```

Client A und Client B starten jeweils das Programm `UpdateClient`:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
UpdateClient
```

Im Folgenden sehen Sie die verzahnte Interaktion beider Clients.

Client A

```
> Id: 1
ETag: "aa554791461b091bd926f164514a19da"
Order{id='1', customer='4711', product='P001', quantity=10, dateTime=2019-
02-04T13:00:23.320979200}
> +
```

Client B

```
> Id: 1
ETag: "aa554791461b091bd926f164514a19da"
Order{id='1', customer='4711', product='P001', quantity=10, dateTime=2019-
02-04T13:00:23.320979200}
> + 1
ETag: "a47b345f66dee6da0ae222122cecd274"
Order{id='1', customer='4711', product='P001', quantity=11, dateTime=2019-
02-04T13:01:24.365345100}
```

Client A

```
> + 2
Status: 412
ETag: "a47b345f66dee6da0ae222122cecd274"
Order{id='1', customer='4711', product='P001', quantity=11, dateTime=2019-
02-04T13:01:24.365345100}
```

Das Begleitmaterial zu diesem Buch enthält auch eine kleine Webanwendung zur Demonstration von Parallelzugriffen. Hierzu muss die Anwendung in zwei Tabs gestartet werden:

`http://localhost:8080/demo/`

10.8 Caching

Die HTTP-Methode GET unterstützt ein sehr effizientes *Caching*. Ein REST-Service kann festlegen, unter welchen Bedingungen und für welchen Zeitraum ein Client die Antworten zwischenspeichern kann.

Es existieren zwei Modelle.[10]

Expirationsmodell: Vom Server gesendete HTTP-Header enthalten Informationen über die Gültigkeitsdauer einer Antwort. Innerhalb dieses Zeitraums kann der Client die zwischengespeicherte Antwort wiederverwenden, ohne mit dem Server erneut zu kommunizieren.

Validierungsmodell: Der Client fragt beim Server an, ob eine bereits gelieferte Antwort wiederverwendet werden kann. Hierdurch wird dann ggf. eine Übertragung der Antwortdaten vermieden.

Wir demonstrieren beide Modelle anhand eines einfachen REST-Service, der Aufträge (Klasse `Order` aus den vorhergehenden Abschnitten) verwaltet.

→ Projekt caching

```
package demo.order;

...

@Path("orders")
public class OrderResource {
   private static Map<String, Order> map = new ConcurrentHashMap<>();
   private static AtomicInteger counter = new AtomicInteger();

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newOrder(@Context UriInfo uriInfo, Order order) {
      String id = String.valueOf(counter.incrementAndGet());
      order.setId(id);
      order.setDateTime(LocalDateTime.now());
      map.put(id, order);
      URI location = uriInfo.getAbsolutePathBuilder().path(id).build();
      return Response.created(location).build();
   }

   @PUT
   @Path("{id}")
   @Consumes(MediaType.APPLICATION_JSON)
   public Response updateOrder(@PathParam("id") String id, Order order) {
      if (map.get(id) == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      order.setId(id);
      order.setDateTime(LocalDateTime.now());
      map.put(id, order);
      return Response.noContent().status(Response.Status.OK).build();
   }
```

10 Siehe Tilkov, S. u. a.: REST und HTTP. dpunkt.verlag, 3. Auflage 2015, Kapitel 10

```
   @GET
   @Path("{id}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response getOrder(@PathParam("id") String id) {
      Order order = map.get(id);
      if (order == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }
      return Response.ok(order).build();
   }

   ...
}
```

In den folgenden Szenarien werden die Client-Programme `PostClient` und `UpdateClient` verwendet.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class PostClient {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";

   public static void main(String[] args) {
      Order order = new Order("4711", "P001", 10);
      String uri = post(order);
      System.out.println(uri);
   }

   private static String post(Order order) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/orders");
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().post(entity);
      int code = response.getStatus();
      System.out.println("Status: " + code);
      String location = null;
      if (code == 201) {
         location = response.getLocation().toString();
      }
      return location;
   }
}
```

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
```

```
public class UpdateClient {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      int id = 1;
      Order order = get(id);
      if (order != null) {
         order.setQuantity(order.getQuantity() + 1);
         System.out.println("Neue Menge: " + order.getQuantity());
         put(id, order);
      }
   }

   private static Order get(int id) {
      WebTarget target = client.target(BASE_URI + "/orders/" + id);
      Response response = target.request().get();
      Order order = null;
      int code = response.getStatus();
      System.out.println("Status: " + code);
      if (code == 200) {
         order = response.readEntity(Order.class);
      }
      return order;
   }

   private static void put(int id, Order order) {
      WebTarget target = client.target(BASE_URI + "/orders/" + id);
      Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
      Response response = target.request().put(entity);
      int code = response.getStatus();
      System.out.println("Status: " + code);
   }
}
```

10.8.1 Expirationsmodell

Ausschnitt aus `OrderResource`:

```
   @GET
   @Path("exp/{id}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response getOrderExp(@PathParam("id") String id) {
      Order order = map.get(id);
      if (order == null) {
         return Response.noContent().status(Response.Status.NOT_FOUND).build();
      }

      CacheControl cc = new CacheControl();
      cc.setNoTransform(true);
      cc.setMustRevalidate(true);
      cc.setMaxAge(60); // 60 Sekunden

      return Response.ok(order).cacheControl(cc).build();
   }
```

Die Klasse `javax.ws.rs.core.CacheControl` repräsentiert den Header *Cache-Control* in der HTTP-Antwort.

Mit dem Aufruf der Methode `setMaxAge` wird die Gültigkeitsdauer der Ressource relativ zum Zeitpunkt der Anfrage in Sekunden angegeben.

Mit `setNoTransform(true)` wird eine Transformation oder Konvertierung der Ressource (z. B. durch einen Proxy) verboten.

`setMustRevalidate(true)` bestimmt, dass der Cache den Status der Ressource überprüfen muss, bevor sie verwendet wird.

Die `ResponseBuilder`-Methode `cacheControl` setzt den Cache-Control-Header.

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Abbildung 10-1: Links zur Ausführung der verschiedenen GET-Methoden

Einfügen eines neuen Auftrags:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
PostClient
```

Ausgabe:

```
Status: 201
http://localhost:8080/demo/rest/orders/1
```

Wir testen die Anwendung mit Hilfe eines Browsers, z. B. Google Chrome. Hierzu benutzen wird die Entwicklertools von Chrome, die über das Menü zu erreichen sind oder mittels Tasturkürzel *Strg + Umschalttaste + I* oder mittels *F12*.

Achtung:

Die Option *Disable cache* muss deaktiviert sein.

Unter dem Reiter *Network* können dann die Header der HTTP-Anfragen und -Antworten kontrolliert werden.

Der Aufruf von

```
http://localhost:8080/demo/test.html
```

liefert die HTML-Seite in Abbildung 10-1.

Testablauf

1. Link "Expirationsmodell" ausführen

   … "quantity": 10 …

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 200 OK
   Cache-Control: no-transform, must-revalidate, max-age=60
   ```

2. Update:

   ```
   java -cp out/production/client;../../libs/jaxrs/*;
   ../../libs/activation/* UpdateClient
   ```

3. Wenn noch keine Minute ab 1. vergangen ist:

   Link "Expirationsmodell" ausführen

   … "quantity": 10 …

4. Wenn eine Minute ab 3. vergangen ist:

   Link "Expirationsmodell" ausführen

   … "quantity": 11 …

10.8.2 Validierungsmodell

Das Expirationsmodell ist sehr einfach zu implementieren. Es ist jedoch schwer im Vorhinein festzulegen, wie lange eine Ressource gültig ist.

Das Validierungsmodell setzt auf einen so genannten bedingten GET-Request, der den Server fragt, ob die gecachte Repräsentation der Ressource noch gültig ist. Im Ja-Fall werden die Daten nicht erneut übertragen (HTTP-Status der Antwort: 304 Not Modified).

Variante 1

Validierung auf Basis des Modifikationszeitpunkts der Ressource

```
@GET
@Path("val1/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getOrderV1(
      @Context Request request, @PathParam("id") String id) {
   Order order = map.get(id);
```

```
    if (order == null) {
       return Response.noContent().status(Response.Status.NOT_FOUND).build();
    }

    CacheControl cc = new CacheControl();
    cc.setNoTransform(true);
    cc.setMustRevalidate(true);
    cc.setMaxAge(0);

    Date date = convertFrom(order.getDateTime());
    Response.ResponseBuilder builder = request.evaluatePreconditions(date);
    if (builder == null) {
       // Die Ressource hat sich geändert.
       builder = Response.ok(order).lastModified(date);
    }
    return builder.cacheControl(cc).build();
 }

 private static Date convertFrom(LocalDateTime dateTime) {
    Instant instant = dateTime.toInstant(ZoneOffset.UTC);
    return Date.from(instant);
 }
```

Hier wird mit Hilfe von `@Context` als erster Parameter ein Objekt vom Interface-Typ `javax.ws.rs.core.Request` injiziert.

Die `Request`-Methode

```
Response.ResponseBuilder evaluatePreconditions(Date lastModified)
```

prüft, ob die Vorbedingung erfüllt ist, also das vom Client gesendete Datum mit `lastModified` übereinstimmt. Ist das nicht der Fall, wird `null` zurück geliefert.

Die Repräsentation der Ressource mit dem Cache-Control-Header und ggf. dem Modifikationszeitpunkt wird zum Client gesendet.

Testablauf

1. Link "Validierungsmodell: Modifikationszeitpunkt" ausführen

   … "quantity": 11 …

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 200 OK
   Last-Modified: Mon, 04 Feb 2019 16:51:00 GMT
   Cache-Control: no-transform, must-revalidate, max-age=0
   ```

2. Link "Validierungsmodell: Modifikationszeitpunkt" ausführen

   … "quantity": 11 …

   Die HTTP-Anfrage enthält:

   ```
   If-Modified-Since: Mon, 04 Feb 2019 16:51:00 GMT
   ```

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 304 Not Modified
   Cache-Control: no-transform, must-revalidate, max-age=0
   ```

3. Update:

```
java -cp out/production/client;../../libs/jaxrs/*;
../../libs/activation/* UpdateClient
```

4. Link "Validierungsmodell: Modifikationszeitpunkt" ausführen

   … "quantity": 12 …

   Die HTTP-Anfrage enthält:

```
If-Modified-Since: Mon, 04 Feb 2019 16:51:00 GMT
```

   Die HTTP-Antwort enthält:

```
HTTP/1.1 200 OK
Last-Modified: Mon, 04 Feb 2019 16:54:39 GMT
Cache-Control: no-transform, must-revalidate, max-age=0
```

Variante 2

Validierung auf Basis eines Hashwerts der Ressource

```
@GET
@Path("val2/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getOrderV2(
       @Context Request request, @PathParam("id") String id) {
   Order order = map.get(id);
   if (order == null) {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   }

   CacheControl cc = new CacheControl();
   cc.setNoTransform(true);
   cc.setMustRevalidate(true);
   cc.setMaxAge(0);

   String digest = computeDigest(order);
   EntityTag etag = new EntityTag(digest);
   Response.ResponseBuilder builder = request.evaluatePreconditions(etag);
   if (builder == null) {
      // Die Ressource hat sich geändert.
      builder = Response.ok(order).tag(etag);
   }
   return builder.cacheControl(cc).build();
}


private static String computeDigest(Order order) {
   String content = order.getId() + order.getCustomer() + order.getProduct()
         + order.getQuantity() + order.getDateTime();
   try {
      MessageDigest md = MessageDigest.getInstance("MD5");
      md.update(content.getBytes());
      byte[] digest = md.digest();
```

```
        StringBuilder sb = new StringBuilder();
        for (byte b : digest) {
           sb.append(String.format("%02x", b));
        }
        return sb.toString();
     } catch (NoSuchAlgorithmException e) {
        return "";
     }
  }
```

In diesem Fall wird geprüft, ob die gecachte Repräsentation der Ressource mit der aktuellen übereinstimmt. Zu diesem Zweck wird ein Hashwert (*Entity-Tag*, *ETag*) vom Server erzeugt und im Header *ETag* gesendet.

Die Klasse `javax.ws.rs.core.EntityTag` repräsentiert den ETag-Header. Zur Berechnung des Hashwerts können verschiedene Algorithmen eingesetzt werden: MD5, SHA-1, SHA-256 oder SHA-512.

Die `Request`-Methode

```
Response.ResponseBuilder evaluatePreconditions(EntityTag eTag)
```

prüft analog zur ersten Variante, ob die Vorbedingung erfüllt ist. Ist das nicht der Fall, wird `null` zurück geliefert. Die Repräsentation der Ressource mit dem Cache-Control-Header und ggf. dem ETag werden zum Client gesendet.

Testablauf

1. Link "Validierungsmodell: ETag" ausführen

   … "quantity": 12 …

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 200 OK
   ETag: "3e1ea1246d007cbf330e79d4e2baa9fe"
   Cache-Control: no-transform, must-revalidate, max-age=0
   ```

2. Link "Validierungsmodell: ETag" ausführen

   … "quantity": 12 …

   Die HTTP-Anfrage enthält:

   ```
   If-None-Match: "3e1ea1246d007cbf330e79d4e2baa9fe"
   ```

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 304 Not Modified
   ETag: "3e1ea1246d007cbf330e79d4e2baa9fe"
   Cache-Control: no-transform, must-revalidate, max-age=0
   ```

3. Update:

   ```
   java -cp out/production/client;../../libs/jaxrs/*;
   ../../libs/activation/* UpdateClient
   ```

4. Link "Validierungsmodell: Modifikationszeitpunkt" ausführen

   … "quantity": 13 …

   Die HTTP-Anfrage enthält:

   ```
   If-None-Match: "3e1ea1246d007cbf330e79d4e2baa9fe"
   ```

   Die HTTP-Antwort enthält:

   ```
   HTTP/1.1 200 OK
   ETag: "5f9f88a19b2a2721bdfb2c90da587199"
   Cache-Control: no-transform, must-revalidate, max-age=0
   ```

Das Begleitmaterial zum Buch enthält eine etwas komfortablere weitere Webanwendung, die wie folgt im Browser aufgerufen werden kann:

```
http://localhost:8080/demo/
```

sowie einen Java-Client für GET, mit dem die verschiedenen Testfälle ausprobiert werden können. Dieser ist mit Hilfe von *OkHttp* implementiert.[11]

10.9 Asynchrone Verarbeitung

HTTP ist ein synchrones Anfrage/Antwort-Protokoll. Dauert die Verarbeitung auf dem Server länger und möchte der Client nicht die ganze Zeit blockiert sein und auf das Ergebnis warten, bietet sich eine *asynchrone Verarbeitung* an.

Hierzu existieren verschiedene Mechanismen, die in den folgenden Abschnitten vorgestellt werden:

- Regelmäßige Nachfrage des Clients beim Server, ob das Ergebnis vorliegt (*Polling*)
- Zustellung des Ergebnisses durch den Server (*Callback*)

In diesen beiden Fällen sendet der Server nach der Client-Anfrage nur eine Annahmebestätigung, das eigentliche Verarbeitungsergebnis erfolgt später. Während der Verarbeitungszeit auf dem Server ist der Client nicht blockiert und die Verbindung muss nicht offen gehalten werden.[12]

Des Weiteren stellen wir das *Client/Server-API für asynchrone Verarbeitung* vor.

10.9.1 Polling

→ Projekt polling

Abbildung 10-2 zeigt den Ablauf und den Datenaustausch zwischen Client und Server. Der Client sendet einen Auftrag (Job) an den Server. Sofort nach Eingang des Auftrags antwortet der Server mit einer Empfangsbestätigung und schickt dem Client eine Adresse, unter der er das Ergebnis der Verarbeitung aktiv abfragen

11 `http://square.github.io/okhttp`

12 Siehe Tilkov, S. u. a.: REST und HTTP. dpunkt.verlag, 3. Auflage 2015, Kapitel 13.1

kann. Solange dieses Ergebnis noch nicht vorliegt, erhält der Client den Status 404 (Not Found).

Abbildung 10-2: Polling-Szenario

```
package demo.job;

public class Job {
   private int id;
   private String inputData;
   private String outputData;
   private String uri;

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getInputData() {
      return inputData;
   }

   public void setInputData(String inputData) {
      this.inputData = inputData;
   }

   public String getOutputData() {
      return outputData;
   }
```

```
   public void setOutputData(String outputData) {
      this.outputData = outputData;
   }

   public String getUri() {
      return uri;
   }

   public void setUri(String uri) {
      this.uri = uri;
   }

   @Override
   public String toString() {
      return "Job{" +
            "id=" + id +
            ", inputData='" + inputData + '\'' +
            ", outputData='" + outputData + '\'' +
            ", uri='" + uri + '\'' +
            '}';
   }
}



package demo.job;

import javax.ws.rs.*;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.atomic.AtomicInteger;

@Path("jobs")
public class JobResource {
   private Map<Integer, Job> map = new ConcurrentHashMap<>();
   private AtomicInteger counter = new AtomicInteger();
   private ExecutorService executorService;

   public JobResource(ExecutorService executorService) {
      this.executorService = executorService;
   }

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newJob(@Context UriInfo uriInfo, Job job) {
      int id = counter.incrementAndGet();
      job.setId(id);
      URI location = uriInfo.getAbsolutePathBuilder().path(String.valueOf(id))
         .build();
      job.setUri(location.toString());
      executorService.execute(() -> runTask(job));
      String msg = "Job accepted, check URI " + location + " for results.";
      Response response = Response.accepted(msg).header("Location", location)
```

```
            .build();
        return response;
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getJob(@PathParam("id") int id) {
        Job job = map.get(id);
        if (job == null) {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        }
        map.remove(id);
        return Response.ok(job).build();
    }

    private void runTask(Job job) {
        String inputData = job.getInputData();
        try {
            Thread.sleep(15000);
            String outputData = "Ergebnis zu '" + inputData + "'";
            job.setOutputData(outputData);
            map.put(job.getId(), job);
        } catch (InterruptedException e) {
            System.out.println("Job wurde abgebrochen.");
        }
    }
}
```

Die Ausführung des Auftrags findet in einem eigenen Thread statt. Der Server richtet hierzu einen `ExecutorService` ein, der über die `MyApplication`-Instanz in die `JobResource`-Instanz injiziert wird.

```
import demo.job.MyApplication;
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
    public static void main(String[] args) throws Exception {
        Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        URI baseUri = new URI("http://0.0.0.0:8080/demo");
        ResourceConfig config = ResourceConfig
            .forApplication(new MyApplication(executorService));
        HttpServer server = GrizzlyHttpServerFactory
            .createHttpServer(baseUri, config);
```

```
      System.out.println("http://localhost:8080/demo/");
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
      executorService.shutdownNow();
      server.shutdownNow();
   }
}
```

```
package demo.job;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ExecutorService;

public class MyApplication extends Application {
   private Set<Object> singletons = new HashSet<>();
   private Set<Class<?>> classes = new HashSet<>();

   public MyApplication(ExecutorService executorService) {
      singletons.add(new JobResource(executorService));
   }

   @Override
   public Set<Class<?>> getClasses() {
      return classes;
   }

   @Override
   public Set<Object> getSingletons() {
      return singletons;
   }
}
```

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class JerseyClient {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      String location = post();
      if (location != null) {
         while (true) {
            try {
               Thread.sleep(5000);
            } catch (InterruptedException e) {
            }
```

```
            System.out.println("Polling ...");
            int status = get(location);
            if (status != 404) {
               break;
            }
         }
      }
   }

   private static String post() {
      String location = null;
      WebTarget target = client.target(BASE_URI + "/jobs");
      Job job = new Job();
      job.setInputData("Das ist mein Input!");
      Entity<Job> entity = Entity.entity(job, MediaType.APPLICATION_JSON);
      Response response = target.request().post(entity);
      if (status(response) == 202) {
         location = response.getLocation().toString();
         System.out.println("Location: " + location);
         System.out.println(response.readEntity(String.class));
      }
      return location;
   }

   private static int get(String uri) {
      WebTarget target = client.target(uri);
      Response response = target.request().get();
      int status = status(response);
      if (status == 200) {
         Job job = response.readEntity(Job.class);
         System.out.println("Job-Id: " + job.getId());
         System.out.println("OutputData: " + job.getOutputData());
      }
      return status;
   }

   private static int status(Response response) {
      int code = response.getStatus();
      String reason = response.getStatusInfo().getReasonPhrase();
      System.out.println("Status: " + code + " " + reason);
      return code;
   }
}
```

Nach Erhalt der Empfangsbestätigung startet der Client alle 5 Sekunden eine GET-Anfrage (Polling) solange, bis das Ergebnis vorliegt.

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Start des Clients:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
JerseyClient
```

Ausgabe:

```
Status: 202 Accepted
Location: http://localhost:8080/demo/jobs/1
Job accepted, check URI http://localhost:8080/demo/jobs/1 for results.
Polling ...
Status: 404 Not Found
Polling ...
Status: 404 Not Found
Polling ...
Status: 200 OK
Job-Id: 1
OutputData: Ergebnis zu 'Das ist mein Input!'
```

Fazit

- Durch wiederholte Anfragen an den Server besteht eine zusätzliche Netzwerklast.
- Der Client erhält nicht unmittelbar nach Fertigstellung der Verarbeitung auf dem Server das Ergebnis, sondern erst bei der nächsten Anfrage.
- Das Ergebnis der Verarbeitung ist selbst eine Ressource. Nicht mehr benötigte Ressourcen müssen ggf. gelöscht werden.

10.9.2 Callback

Abbildung 10-3: Callback-Szenario

→ Projekt callback

Das Szenario aus dem vorhergehenden Abschnitt hat sich wie folgt geändert:

Der Client sendet das Job-Objekt nun mit einer Adresse (URI). Diesen URI (hier: `http://localhost:8888/demo/callback/4711`) kann der Server verwenden, um das Ergebnis später zuzustellen. Zu diesem Zweck startet der Client zu Beginn selbst einen REST-fähigen Server auf Port 8888, der die PUT-Anfrage des Job-Servers zur Übermittlung des Ergebnisses entgegennimmt.

Abbildung 10-3 zeigt die Interaktion zwischen Client und Server.

```
package demo.job;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.atomic.AtomicInteger;

@Path("jobs")
public class JobResource {
   private AtomicInteger counter = new AtomicInteger();
   private ExecutorService executorService;

   public JobResource(ExecutorService executorService) {
      this.executorService = executorService;
   }

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newJob(@Context UriInfo uriInfo, Job job) {
      int id = counter.incrementAndGet();
      job.setId(id);
      executorService.execute(() -> runTask(job));
      String uri = job.getUri();
      String msg = "Job accepted, will notify URI " + uri + " when done.";
      Response response = Response.accepted(msg).build();
      return response;
   }

   private void runTask(Job job) {
      String inputData = job.getInputData();
      try {
         Thread.sleep(15000);
         String outputData = "Ergebnis zu '" + inputData + "'";
         job.setOutputData(outputData);
```

```
            // Übermittlung des Ergebnisses an den Client
            Client client = ClientBuilder.newClient();
            WebTarget target = client.target(job.getUri());
            Entity<Job> entity = Entity.entity(job, MediaType.APPLICATION_JSON);
            target.request().put(entity);
         } catch (InterruptedException e) {
            System.out.println("Job wurde abgebrochen.");
         } catch (Exception e) {
            System.err.println(e.getMessage());
            System.err.println(e.getCause().getMessage());
         }
      }
   }
```

`CallbackResource` wird für den Client in `MyApplication` registriert.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("callback")
public class CallbackResource {
   @PUT
   @Path("{notificationId}")
   @Consumes(MediaType.APPLICATION_JSON)
   public Response put(@PathParam("notificationId") int notificationId, Job job) {
      System.out.println("Notification-Id: " + notificationId);
      System.out.println("Job-Id: " + job.getId());
      System.out.println("OutputData: " + job.getOutputData());
      return Response.noContent().status(Response.Status.OK).build();
   }
}
```

```
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

public class JerseyClient {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static final String CALLBACK_BASE_URI = "http://localhost:8888/demo";

   public static void main(String[] args) {
      Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);
```

```
        startServer();

        int notificationId = 4711;
        post(notificationId);
        System.out.println("Der Client arbeitet weiter ...");
    }

    private static void startServer() {
        new Thread() {
            @Override
            public void run() {
            try {
                URI baseUri = new URI(CALLBACK_BASE_URI);
                ResourceConfig config = ResourceConfig
                    .forApplicationClass(MyApplication.class);
                HttpServer server = GrizzlyHttpServerFactory
                    .createHttpServer(baseUri, config);
                System.out.println("ENTER stoppt den Server.");
                System.in.read();
                server.shutdownNow();
            } catch (Exception e) {
                System.err.println(e.getMessage());
            }
            }
        }.start();
    }

    private static void post(int notificationId) {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(BASE_URI + "/jobs");
        Job job = new Job();
        job.setInputData("Das ist mein Input!");
        job.setUri(CALLBACK_BASE_URI + "/callback/" + notificationId);
        Entity<Job> entity = Entity.entity(job, MediaType.APPLICATION_JSON);
        Response response = target.request().post(entity);
        if (status(response) == 202) {
            System.out.println(response.readEntity(String.class));
        }
    }

    private static int status(Response response) {
        int code = response.getStatus();
        String reason = response.getStatusInfo().getReasonPhrase();
        System.out.println("Status: " + code + " " + reason);
        return code;
    }
}
```

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Start des Clients:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/grizzly/*;
../../libs/activation/* JerseyClient
```

Ausgabe:

```
ENTER stoppt den Server.
Status: 202 Accepted
Job accepted, will notify URI http://localhost:8888/demo/callback/4711
when done.
Der Client arbeitet weiter ...
Notification-Id: 4711
Job-Id: 1
OutputData: Ergebnis zu 'Das ist mein Input!'
```

Fazit

- Der Client ist nun nicht nur ein Client, sondern auch ein Server. Das erfordert evtl. entsprechende Firewall-Regeln, damit der Client von außen als Server erreichbar ist.
- Der Client muss die asynchron eingehende Antwort seiner Anfrage zuordnen können. Im Beispiel muss dazu die *Notification-Id* eindeutig sein.

10.9.3 Client/Server-API für asynchrone Verarbeitung

Die Verarbeitung von REST-Anfragen auf dem Server erfolgt standardmäßig synchron. Entgegennahme der Anfrage und ihre Verarbeitung erfolgen jeweils in ein und demselben Thread. Dieses Modell reicht aus, wenn die eigentliche Verarbeitung nur relativ kurze Zeit benötigt.

Das asynchrone Verarbeitungsmodell bietet sich an, wenn die Ausführung längere Zeit benötigt. Hier wird die Entgegennahme der Anfrage und die Auslieferung der Antwort von der eigentlichen Verarbeitung getrennt. Der I/O-Thread, der die Client-Verbindung unterhält, wird ausgesetzt (*suspend*) und nach Vorliegen des Verarbeitungsergebnisses wieder fortgesetzt (*resume*). Zudem kann die Verarbeitung abgebrochen werden, z. B. dann, wenn eine vorgegebene Dauer überschritten wurde (*Timeout*). Durch diese Vorgehensweise wird der Durchsatz des Servers in der Regel erhöht. Der freigegebene I/O-Thread kann für neue eingehende Anforderungen verwendet werden, während die eigentliche Verarbeitung der Anfrage noch in einem anderen Thread ausgeführt wird.

Unabhängig von der serverseitigen asynchronen Verarbeitung kann auch auf der Client-Seite asynchrone Verarbeitung eingesetzt werden. Hier muss der Client nicht auf das Vorliegen der Antwort aktiv warten. Die Fertigstellung kann jederzeit abgefragt werden (`isDone()`) oder der Client wird Event-gesteuert per Callback informiert.

→ Projekt async

Die Klasse `Job` gleicht der aus den vorhergehenden Abschnitten. Das Attribut `uri` wird jedoch nicht gebraucht.

Um verschiedene Verhaltensweisen der Anwendung testen zu können, enthält die Klasse `Job` zusätzlich das `int`-Attribut `timeout`. Hiermit kann die maximale Laufzeit eines Jobs in Sekunden angegeben werden.

```
package demo.job;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.CompletionCallback;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;
import javax.ws.rs.core.MediaType;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

@Path("jobs")
public class JobResource {
   private AtomicInteger numberOfSuccessResponses = new AtomicInteger();
   private AtomicInteger numberOfFailures = new AtomicInteger();
   private ExecutorService executorService;

   public JobResource(ExecutorService executorService) {
      this.executorService = executorService;
   }

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   public void doJob(@Suspended AsyncResponse asyncResponse, Job job) {
      asyncResponse.register(new CompletionCallback() {
         @Override
         public void onComplete(Throwable throwable) {
            if (throwable == null) {
               if (asyncResponse.isCancelled())
                  numberOfFailures.incrementAndGet();
               else
                  numberOfSuccessResponses.incrementAndGet();
            } else {
               System.out.println(throwable);
               numberOfFailures.incrementAndGet();
            }
         }
      });

      final Future<?> task = executorService.submit(
         () -> runTask(asyncResponse, job));

      asyncResponse.setTimeout(job.getTimeout(), TimeUnit.SECONDS);
```

```
        asyncResponse.setTimeoutHandler(new TimeoutHandler() {
            @Override
            public void handleTimeout(AsyncResponse asyncResponse) {
                asyncResponse.cancel();
                task.cancel(true);
            }
        });
    }

    @GET
    @Path("status")
    public String getStatus() {
        return "numberOfSuccessResponses: " + numberOfSuccessResponses +
                ", numberOfFailures: " + numberOfFailures;
    }

    private void runTask(AsyncResponse asyncResponse, Job job) {
        String inputData = job.getInputData();
        try {
            Thread.sleep(15000); // simuliert Zeitaufwand
            String outputData = "Ergebnis zu '" + inputData + "'";
            job.setOutputData(outputData);
            asyncResponse.resume(job);
        } catch (InterruptedException e) {
            System.err.println("Job " + job.getId() + " wurde abgebrochen.");
        }
    }
}
```

Neu sind hier die Interfaces

`AsyncResponse`,
`CompletionCallback` und
`TimeoutHandler`

sowie die Annotation

`Suspended`.

Alle stammen aus dem Paket `javax.ws.rs.container`.

Die Annotation `@Suspended` in der Methode `doJob` sorgt dafür, dass die Antwort asynchron erzeugt wird. Nach Vorliegen des Ergebnisses wird die Verbindung wieder aufgenommen und die Antwort mit der `AsyncResponse`-Methode `resume` zurückgegeben.

Die `AsyncResponse`-Methode `setTimeout` setzt den *Timeout* gemäß der Angabe in der `Job`-Instanz. Der `TimeoutHandler` beendet die Verarbeitung und liefert eine entsprechende Fehlermeldung (HTTP-Status 503) an den Client.

Mit der registrierten `CompletionCallback`-Instanz wird der Status der Fertigstellung ausgewertet.

Der Server wird wie üblich gestartet:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Jederzeit kann während der Laufzeit des Servers die Anzahl der erfolgreichen Antworten und die Fehleranzahl abgefragt werden.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class Status {
   private static final String BASE_URI = "http://localhost:8080/demo";

   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/jobs/status");
      Response response = target.request().get();
      int code = response.getStatus();
      System.out.println("Status: " + code);
      if (code == 200) {
         System.out.println(response.readEntity(String.class));
      }
   }
}
```

Aufruf von `Status`:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
Status
```

Client mit Polling-Funktion

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.concurrent.Future;

public class JerseyClient1 {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Job job;

   public static void main(String[] args) {
      int timeout = Integer.parseInt(args[0]); // Sekunden

      job = new Job();
      job.setId(4711);
      job.setInputData("Das ist mein Input!");
      job.setTimeout(timeout);
```

```
        post();
    }

    private static void post() {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(BASE_URI + "/jobs");
        Entity<Job> entity = Entity.entity(job, MediaType.APPLICATION_JSON);
        Future<Response> responseFuture = target.request().async().post(entity);

        while (!responseFuture.isDone()) {
            System.out.println("Polling ...");
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
            }
        }

        try {
            Response response = responseFuture.get();
            int code = response.getStatus();
            System.out.println("Status: " + code);
            if (code == 200) {
                job = response.readEntity(Job.class);
                System.out.println("Id: " + job.getId());
                System.out.println("OutputData: " + job.getOutputData());
            }
        } catch (Exception e) {
            System.err.println(e);
        } finally {
            client.close();
        }
    }
}
```

Der Aufruf von `async()` liefert ein `AsyncInvoker`-Objekt, mit dem HTTP-Methoden asynchron ausgeführt werden können.
`target.request().async().post(entity)` liefert dann ein `Future`-Objekt.

Aufruf von `JerseyClient1` mit verschiedenen Parametern für den Timeout (z. B. 5 oder 30).

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
JerseyClient1 30
```

Ausgabe:

```
Polling ...
Polling ...
Polling ...
Polling ...
Status: 200
Id: 4711
OutputData: Ergebnis zu 'Das ist mein Input!'
```

Client mit Callback

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.InvocationCallback;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.concurrent.Future;

public class JerseyClient2 {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Job job;
   private static volatile boolean running = true;

   public static void main(String[] args) {
      int timeout = Integer.parseInt(args[0]); // Sekunden

      job = new Job();
      job.setId(4711);
      job.setInputData("Das ist mein Input!");
      job.setTimeout(timeout);
      post();
   }

   private static void post() {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/jobs");
      Entity<Job> entity = Entity.entity(job, MediaType.APPLICATION_JSON);
      Future<Response> responseFuture = target.request().async()
            .post(entity, new InvocationCallback<Response>() {
               @Override
               public void completed(Response response) {
                  int code = response.getStatus();
                  System.out.println("Status: " + code);
                  if (code == 200) {
                        job = response.readEntity(Job.class);
                        System.out.println("Id: " + job.getId());
                        System.out.println("OutputData: " + job.getOutputData());
                  }
                  running = false;
               }

               @Override
               public void failed(Throwable throwable) {
                  System.err.println(throwable);
                  running = false;
               }
            });

      System.out.println("Der Client arbeitet weiter ...");

      while (running) {
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
         }
      }
```

```
        client.close();
    }
}
```

Der Aufruf von `post()` erhält als zweiten Parameter ein Objekt vom Interface-Typ `javax.ws.rs.client.InvocationCallback`. Hier müssen die Methoden `completed` und `failed` implementiert werden.

Aufruf von `JerseyClient2` mit verschiedenen Parametern für den Timeout (z. B. 5 oder 30).

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
JerseyClient2 30
```

Ausgabe:

```
Der Client arbeitet weiter ...
Status: 200
Id: 4711
OutputData: Ergebnis zu 'Das ist mein Input!'
```

Hinweis

Zu beachten ist, dass zwar im Client und im Server die Verarbeitung asynchron (*non-blocking*) erfolgt, die Übertragung von HTTP-Anfrage und -Antwort aber synchron verläuft. Bis die Antwort auf der Server-Seite vorliegt, bleibt die HTTP-Verbindung zwischen Client und Server offen.

10.10 Server-Sent Events (SSE)

Die Technik *Server-Sent Events* (*SSE*) wurde im Rahmen von HTML5 spezifiziert. Dabei handelt es sich um eine *Server-Push-Technik*, bei der der Client eine unidirektionale HTTP-Verbindung zum Server offen hält. Über die offene Verbindung kann der Server jederzeit Nachrichten an den Client schicken, ohne dass die Verbindung zwischendurch getrennt und wieder neu hergestellt werden muss.[13]

Moderne Browser unterstützen diesen Standard und kümmern sich auch um den Wiederaufbau der Verbindung bei Abbrüchen.

SSE ähnelt dem *Long-Polling-Mechanismus*, es wird jedoch nicht nur eine Nachricht pro Verbindung gesendet.

Der Ablauf im Einzelnen:

13 https://www.w3.org/TR/eventsource/

- Der Client sendet eine Anfrage, und der Server hält eine Verbindung, bis eine neue Nachricht bereit ist.
- Dann sendet der Server die Nachricht zurück an den Client, während die Verbindung weiterhin offen bleibt, so dass sie für eine weitere Nachricht verwendet werden kann, sobald sie verfügbar ist.
- Sobald eine neue Nachricht vorliegt, wird sie auf der gleichen ursprünglichen Verbindung zurück an den Client gesendet.
- Der Client verarbeitet die vom Server gesendeten Nachrichten einzeln, ohne die Verbindung nach der Verarbeitung der einzelnen Nachrichten zu schließen.

Abbildung 10-4: Server-Sent Events

Wir stellen im Folgenden Grundzüge des SSE-API anhand eines einfachen Beispiels vor.

→ Projekt sse

Der REST-Service (siehe Abbildung 10-5) hat die Aufgabe, Nachrichten, die der `PostClient` erzeugt und ihm zusendet an Clients (`Consumer` und Browser) als Events zu verteilen. Diese beiden Clients melden sich hierfür als Abonnenten beim so genannten *Broadcaster* des Services an.

Ein weiterer Client (`Status`) erhält für eine kurze Zeit Informationen über die Anzahl der bisher veröffentlichten Nachrichten.

Die mit JavaScript realisierte Webanwendung, die im Browser (z. B. Google Chrome) läuft, ist im Begleitmaterial zu diesem Buch enthalten. Sie wir hier allerdings nicht erläutert.

Die verwendete Klasse `Message` ist dieselbe wie in Kapitel 10.2.

Abbildung 10-5: Szenario des Beispiels im Projekt sse

```
package demo.broadcast;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.sse.OutboundSseEvent;
import javax.ws.rs.sse.Sse;
import javax.ws.rs.sse.SseBroadcaster;
import javax.ws.rs.sse.SseEventSink;
import java.time.Instant;
import java.util.concurrent.atomic.AtomicInteger;

@Path("")
public class MessageResource {
   private AtomicInteger counter = new AtomicInteger();
   private Sse sse;
   private SseBroadcaster broadcaster;

   @Context
   public void setSse(Sse sse) {
      this.sse = sse;
      this.broadcaster = sse.newBroadcaster();
   }

   // Beim Broadcaster registrieren
   @GET
   @Path("subscribe")
   @Produces(MediaType.SERVER_SENT_EVENTS)
   public void getNewMessages(@Context SseEventSink eventSink) {
      broadcaster.register(eventSink);
   }
```

```
   // Empfangene Nachrichten per Broadcast senden
   @POST
   @Path("messages")
   @Consumes(MediaType.APPLICATION_JSON)
   public Response newMessage(Message message) {
      int messageId = counter.incrementAndGet();
      message.setId(messageId);
      message.setTimestamp(Instant.now());

      OutboundSseEvent event = sse.newEventBuilder()
            .id(String.valueOf(messageId)) // Event-Id
            .name("new-message") // Event-Name
            .mediaType(MediaType.APPLICATION_JSON_TYPE) // Typ der Event-Daten
            .data(Message.class, message) // Event-Daten
            .comment("Neue Nachricht") // Kommentar zum Event
            .build();

      broadcaster.broadcast(event);
      return Response.ok().build();
   }

   // Einfache Events senden
   @GET
   @Path("status")
   @Produces(MediaType.SERVER_SENT_EVENTS)
   public void getStatus(@Context SseEventSink eventSink) {
      new Thread(() -> {
         for (int i = 0; i < 5; i++) {
            OutboundSseEvent event = sse.newEventBuilder()
                  .name("status")
                  .mediaType(MediaType.TEXT_PLAIN_TYPE)
                  .data(Integer.class, counter.get())
                  .comment("Anzahl Nachrichten")
                  .build();

            eventSink.send(event);

            try {
               Thread.sleep(10000);
            } catch (InterruptedException e) {
            }
         }
      }).start();
   }
}
```

Das Interface `Sse` ermöglicht die Erzeugung von `OutboundSseEvent`-Instanzen und der `SseBroadcaster`-Instanz. Eine `Sse`-Instanz wird mit der mit `@Context` (siehe Kapitel 10.2) annotierten set-Methode injiziert.

Die Methode `newMessage` empfängt eine Nachricht (vom `PostClient`), macht daraus ein `OutboundSseEvent` und verteilt dieses Event mit der `SseBroadcaster`-Methode `broadcast` an die angemeldeten Clients.

Diverse Methoden `OutboundSseEvent.Builder`-Methoden bauen ein Event auf:

- `id` setzt die eindeutige *Event-Id* (optional).
- `name` setzt einen eindeutigen Namen für den *Event-Typ* (optional). Damit können verschiedene Event-Kanäle auseinander gehalten werden und ein Client kann einen bestimmten Kanal abonnieren.
- `mediaType` legt den *Medientyp* für die Event-Daten fest.
- `data` setzt die *Event-Daten* und bestimmt ihren Java-Typ.
- `comment` legt einen *Kommentar* zum Event fest (optional).

Event-Stream-Format

Der Content-Type des Event-Streams ist: *text/event-stream*. Ein Event ist wie folgt aufgebaut. Hier ein Beispiel:

```
: Neue Nachricht\n
event: new-message\n
id: 1\n
data: {"id":1,"text":"...","timestamp":"..."}\n\n
```

Die Methode `getNewMessages` registriert eine `SseEventSink`-Instanz beim Broadcaster. `SseEventSink` repräsentiert einen Event-Stream.

Events können auch gezielt auf Anfrage eines Clients gesendet werden. Das zeigt die Methode `getStatus`. Hier wird die Anzahl Nachrichten als Event mit Hilfe der `SseEventSink`-Methode `send` gesendet.

SSE unterstützt auch einen *Reconnection-Mechanismus*. Dieser ermöglicht Clients nach einer abgebrochenen Verbindung diese nach einer bestimmten Zeit wieder aufzunehmen. Die Standard-Timeout-Einstellung kann über `retry:` im Event-Stream geändert werden. Auf diese Möglichkeit gehen wir hier nicht ein.

Der `PostClient` erzeugt in zufälligen Zeitabständen zufällige Nachrichtentexte.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.Random;

public class PostClient {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";

   public static void main(String[] args) {
      Random random = new Random();
      for (int i = 0; i < 9; i++) {
         post("/messages", getRandomString());
```

```
            try {
               Thread.sleep(1000 + random.nextInt(5000));
            } catch (InterruptedException e) {
            }
         }
      }

      private static void post(String uri, String text) {
         Client client = ClientBuilder.newClient();
         WebTarget target = client.target(BASE_URI + uri);
         Message message = new Message();
         message.setText(text);
         Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
         Response response = target.request().post(entity);
         int code = response.getStatus();
         System.out.println("Status: " + code);
      }

      public static String getRandomString() {
         int length = 8 + (int) (Math.random() * 23);
         StringBuffer sb = new StringBuffer();
         for (int i = 0; i < length; i++) {
            sb.append((char) (65 + (int) (Math.random() * 26)));
         }
         return sb.toString();
      }
   }
```

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.sse.SseEventSource;
import java.io.IOException;

public class Consumer {
   private static String BASE_URI;

   public static void main(String[] args) {
      if (args.length > 0 && args[0].equals("-m"))
         BASE_URI = "http://localhost:8888/demo/rest";
      else
         BASE_URI = "http://localhost:8080/demo/rest";

      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/subscribe");

      try (SseEventSource eventSource = SseEventSource.target(target).build()) {
         eventSource.register(
            event -> { // event consumer
               System.out.println("Name: " + event.getName());
               System.out.println("Id: " + event.getId());
               System.out.println("Kommentar: " + event.getComment());
               Message message = event
                  .readData(Message.class, MediaType.APPLICATION_JSON_TYPE);
               System.out.println(message.getText() + "(" +
                  message.getTimestamp() + ")");
```

```
                System.out.println();
            },
            t -> System.out.println(t), // error consumer
            () -> System.out.println("Ende") // onComplete handler
         );
         eventSource.open();

         System.out.println("ENTER beendet den Consumer.");
         System.in.read();
      } catch (IOException e) {
      }

      client.close();
      System.exit(0);
   }
}
```

Um beim Client eingehende Events lesen und verarbeiten zu können, muss eine `SseEventSource`-Instanz erzeugt werden. Mit der Methode `register` werden ein *Event Consumer* (Typ `InboundSseEvent`), ein *Error Consumer* (Typ `Throwable`) und ein *OnComplete-Handler* (`Runnable`) registriert.

Die `SseEventSource`-Methode `open` öffnet die Verbindung zum `WebTarget` und start die Verarbeitung eingehender Events. `close` schließt den Event-Stream.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.sse.SseEventSource;

public class Status {
   private static final String BASE_URI = "http://localhost:8080/demo/rest";

   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      WebTarget target = client.target(BASE_URI + "/status");

      try (SseEventSource eventSource = SseEventSource.target(target).build()) {
         eventSource.register(
            event -> {
               System.out.println("Name: " + event.getName());
               System.out.print(event.getComment() + ": ");
               int data = event
                  .readData(Integer.class, MediaType.TEXT_PLAIN_TYPE);
               System.out.println(data);
               System.out.println();
            }, t -> System.out.println(t),
            () -> System.out.println("Ende")
         );
         eventSource.open();

         Thread.sleep(60000);
      } catch (InterruptedException e) {
      }
```

```
        client.close();
        System.exit(0);
    }
}
```

Testablauf

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Start des Clients `Consumer`:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
Consumer
```

Start des Clients `PostClient`:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
PostClient
```

Ausgabe von `Consumer`:

```
Name: new-message
Id: 1
Kommentar: Neue Nachricht
OPECNJTGE(2019-02-06T11:24:23.493666300Z)
...
```

Abbildung 10-6: Webanwendung empfängt Server-Sent Events

Start des Clients `Status`:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
Status
```

Ausgabe von `Status`:

```
Name: status
Anzahl Nachrichten: 2
```

10.11 Filterketten

Auf der Client- und der Server-Seite können Filter zur Vor- bzw. Nachbearbeitung von REST-Anfragen und -antworten eingesetzt werden. Dies ist sinnvoll, um anwendungsneutrale Verarbeitungen, beispielsweise eine Protokollierung, durchzuführen. Damit muss im einzelnen Service nicht jedes Mal der gleiche Code erscheinen. Filter können in unterschiedlichen Anwendungsszenarien wiederverwendet werden.

Grundlage zur Implementierung sind die folgenden Interfaces:

```
javax.ws.rs.container.ContainerRequestFilter
javax.ws.rs.container.ContainerResponseFilter
javax.ws.rs.client.ClientRequestFilter
javax.ws.rs.client.ClientResponseFilter
```

Alle Filter besitzen die einzige Methode `filter`, die zu implementieren ist. Während Response-Filter Zugriff auf Daten der REST-Anfrage und REST-Antwort haben, haben Request-Filter nur Zugriff auf Daten der REST-Anfrage.

Das folgende Beispiel nutzt den Service aus Kapitel 10.2, wobei nur die Methoden zum Anlegen, Ändern, Löschen und Anzeigen von Nachrichten auf Basis von JSON verwendet werden.

Während die Methoden POST und GET von allen Benutzern aufgerufen werden können, sind PUT und DELETE geschützt. Hierzu wird das Authentifizierungsverfahren *HTTP Basic Authentication* verwendet.

Basic Authentication

Jersey unterstützt die Authentifizierung mit Benutzername und Passwort nach dem Verfahren *Basic Authentication*.[14]

Ein Client, der das JAX-RS Client API nutzt, muss dieses Feature zu Beginn registrieren:

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature
    .basic(user, password);
target.register(feature);
```

Benutzername und Passwort werden mit einem Doppelpunkt verkettet, dann nach dem *Base64-Verfahren*[15] codiert und in einem speziellen HTTP-Header an den Server geschickt.

---

14 https://tools.ietf.org/html/rfc7617

15 https://de.wikipedia.org/wiki/Base64

Zu beachten ist, dass es sich hierbei um keine echte Verschlüsselung von Benutzername und Passwort mit dem Ziel der Geheimhaltung handelt. *Base64* bildet 8-Bit-Binärdaten auf eine Zeichenkette aus Buchstaben ohne Umlaute, Ziffern, "+" und "/" ab.

Beispiel:

Ist der Benutzername "`admin`" und das Passwort "`secret`", so lautet der HTTP-Header:

```
Authorization: Basic YWRtaW46c2VjcmV0
```

"`YWRtaW46c2VjcmV0`" wird gemäß dem Base64-Verfahren auf der Serverseite zu "`admin:secret`" decodiert.

Abbildung 10-7 gibt einen Überblick über die auf Client- und Server-Seite eingesetzten Filter.

Abbildung 10-7: Einsatz von Filtern

→ Projekt filter

In der Klasse `MyApplication` werden *vier Filter der Serverseite* registriert (zwei Request-Filter und zwei Response-Filter):

```
public MyApplication() {
   singletons.add(new MessageResource());
   singletons.add(new MyAuthFilter());
   singletons.add(new MyLogFilter());
   singletons.add(new Timer1());
   singletons.add(new Timer2());
}
```

Die Reihenfolge, in der die Filter durchlaufen werden, wird durch die Annotation `@Priority` mit einem numerischen Wert festgelegt.

Request-Filter werden dabei in aufsteigender Folge der Prioritäten durchlaufen, Response-Filter in absteigender Reihenfolge.

Filter auf der Server-Seite

Der Request-Filter `MyAuthFilter` prüft die Gültigkeit von User und Passwort bei PUT und DELETE. `ContainerRequestContext` liefert Request-bezogene Informationen für den Filter, die Methode `abortWith` bricht die Filterkette ab.

```
package demo.filter;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.core.Response;
import java.io.IOException;
import java.util.Base64;
import java.util.logging.Logger;

@Priority(1)
public class MyAuthFilter implements ContainerRequestFilter {
   private static final Logger LOGGER =
Logger.getLogger(MyAuthFilter.class.getName());

   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      String method = requestContext.getMethod();
      if (!method.equals("PUT") && !method.equals("DELETE"))
         return;

      boolean isValid = false;
      String auth = requestContext.getHeaderString("Authorization");
      if (auth != null) {
         byte[] b = Base64.getDecoder().decode(auth.substring(6));
         String[] parts = new String(b).split(":");
         String user = parts[0];
         String password = parts[1];
         if (user.equals("admin") && password.equals("secret")) {
            isValid = true;
         }
      }

      if (!isValid) {
         LOGGER.warning("Nicht autorisiert");
         requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED)
            .build());
      }
   }
}
```

Der Request-Filter `Timer1` ermittelt die aktuelle Zeit in Nanosekunden und speichert diese im Request-Context.

```
package demo.filter;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
```

```
import java.io.IOException;
import java.util.logging.Logger;

@Priority(2)
public class Timer1 implements ContainerRequestFilter {
   private static final Logger LOGGER = Logger.getLogger(Timer1.class.getName());

   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      long start = System.nanoTime();
      requestContext.setProperty("start", start);
   }
}
```

Der Response-Filter `Timer2` liest die Startzeit aus dem Request-Context und ermittelt hiermit die Dauer der Ausführung der Service-Methode. Mit Hilfe der injizierten `ResourceInfo`-Instanz können die beteiligte Ressourcenklasse und Ressourcenmethode ermittelt werden.

```
package demo.filter;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.Context;
import java.io.IOException;
import java.lang.reflect.Method;
import java.util.logging.Logger;

@Priority(2)
public class Timer2 implements ContainerResponseFilter {
   private static final Logger LOGGER = Logger.getLogger(Timer2.class.getName());

   @Context
   ResourceInfo info;

   @Override
   public void filter(ContainerRequestContext requestContext,
                      ContainerResponseContext responseContext) throws IOException {
      long end = System.nanoTime();
      long start = (Long) requestContext.getProperty("start");
      Method resourceMethod = info.getResourceMethod();
      LOGGER.info("Dauer " + resourceMethod.getName() + ": " + (end - start)
         + " ns");
   }
}
```

Der Response-Filter `MyLogFilter` speichert Informationen über Request und Response.

`ContainerResponseContext` liefert Response-bezogene Informationen für den Filter.

```
package demo.filter;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import java.io.IOException;
import java.lang.reflect.Method;
import java.net.URI;
import java.util.logging.Logger;

@Priority(1)
public class MyLogFilter implements ContainerResponseFilter {
   private static final Logger LOGGER = Logger
      .getLogger(MyLogFilter.class.getName());

   @Context
   ResourceInfo info;

   @Override
   public void filter(ContainerRequestContext requestContext,
                   ContainerResponseContext responseContext) throws IOException {
      Class<?> resourceClass = info.getResourceClass();
      Method resourceMethod = info.getResourceMethod();

      StringBuilder sb = new StringBuilder();
      sb.append(resourceClass.getName() + ":" + resourceMethod.getName() + "\n");
      sb.append("   Request: ");
      sb.append(requestContext.getMethod() + " " +
         requestContext.getUriInfo().getAbsolutePath() + "\n");
      sb.append("   Response: ");
      sb.append(responseContext.getStatus() + " " +
         responseContext.getStatusInfo());
      MediaType mediaType = responseContext.getMediaType();
      if (mediaType != null)
         sb.append(", MediaType: " + mediaType);
      URI location = responseContext.getLocation();
      if (location != null)
         sb.append(", Location: " + location);

      LOGGER.info(sb.toString());
   }
}
```

Filter auf der Client-Seite

Auf der Clientseite werden der Request-Filter `MyCheckFilter` und der Response-Filter `MyClientLogFilter` eingesetzt.

`MyCheckFilter` prüft, ob ein *Authorization-Header* verwendet wird und bricht ggf. den Request ab. `ClientRequestContext` liefert Request-bezogene Informationen für den Filter, die Methode `abortWith` bricht die Filterkette ab. `MyClientLogFilter` speichert Informationen über Request und Response. `ClientResponseContext` liefert Response-bezogene Informationen für den Filter.

```
package demo.filter;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.core.Response;
import java.io.IOException;
import java.util.logging.Logger;

public class MyCheckFilter implements ClientRequestFilter {
   private static final Logger LOGGER = Logger
                        .getLogger(MyCheckFilter.class.getName());

   @Override
   public void filter(ClientRequestContext requestContext) throws IOException {
      String method = requestContext.getMethod();
      if (method.equals("PUT") || method.equals("DELETE")) {
         if (requestContext.getHeaderString("Authorization") == null) {
            LOGGER.warning("Header 'Authorization' fehlt");
            requestContext.abortWith(Response.status(Response.Status.BAD_REQUEST)
               .build());
         }
      }
   }
}
```

```
package demo.filter;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientResponseContext;
import javax.ws.rs.client.ClientResponseFilter;
import javax.ws.rs.core.MediaType;
import java.io.IOException;
import java.net.URI;
import java.util.logging.Logger;

public class MyClientLogFilter implements ClientResponseFilter {
   private static final Logger LOGGER = Logger
                        .getLogger(MyClientLogFilter.class.getName());

   @Override
   public void filter(ClientRequestContext requestContext,
         ClientResponseContext responseContext) throws IOException {
      StringBuilder sb = new StringBuilder();
      sb.append("Request: ");
      sb.append(requestContext.getMethod() + " " + requestContext.getUri() + "\n");
      sb.append("   Response: ");
      sb.append(responseContext.getStatus() + " " +
         responseContext.getStatusInfo());
      MediaType mediaType = responseContext.getMediaType();
      if (mediaType != null)
         sb.append(", MediaType: " + mediaType);
      URI location = responseContext.getLocation();
      if (location != null)
         sb.append(", Location: " + location);

      LOGGER.info(sb.toString());
   }
}
```

`JerseyClient` registriert mit der `Client`-Methode `register` die beiden Filter. In den Methoden `put` und `delete` wird jeweils der *Authorization-Header* für *Basic Authentication* gesetzt.

```
import demo.filter.MyCheckFilter;
import demo.filter.MyClientLogFilter;
import org.glassfish.jersey.client.authentication.HttpAuthenticationFeature;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.List;

public class JerseyClient {
   private static final String BASE_URI = "http://localhost:8080/demo";
   private static Client client;

   public static void main(String[] args) {
      setup();

      get("/messages/1");
      get("/messages");

      post("/messages", "Neue Nachricht");
      put("/messages/1", "Nachricht AA", "admin", "secret");
      delete("/messages/2", "admin", "secret");

      get("/messages/limited?n=2");
   }

   private static void setup() {
      System.setProperty("java.util.logging.config.file",
                                 "client/logging.properties");
      client = ClientBuilder.newClient();
      client.register(new MyCheckFilter());
      client.register(new MyClientLogFilter());
   }

   private static void get(String uri) {
      System.out.println("\n--- GET " + BASE_URI + uri);
      WebTarget target = client.target(BASE_URI + uri);
      Response response = target.request().accept(MediaType.APPLICATION_JSON)
         .get();
      if (status(response) == 200) {
         if (uri.endsWith("messages") || uri.contains("limited")) {
            List<Message> list = response.readEntity(
               new GenericType<List<Message>>() {
            });
            for (Message message : list) {
               System.out.println(message.getId() + ";" +
                        message.getText() + ";" + message.getTimestamp());
            }
         } else {
```

```
            Message message = response.readEntity(Message.class);
            System.out.println(message.getId() + ";" +
                  message.getText() + ";" + message.getTimestamp());
         }
      }
   }

   private static void post(String uri, String text) {
      System.out.println("\n--- POST " + BASE_URI + uri);
      WebTarget target = client.target(BASE_URI + uri);
      Message message = new Message();
      message.setText(text);
      Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
      Response response = target.request().post(entity);
      if (status(response) == 201) {
         String location = response.getLocation().toString();
         System.out.println("Location: " + location);
      }
   }

   private static void put(String uri, String text, String user, String password) {
      System.out.println("\n--- PUT " + BASE_URI + uri);
      WebTarget target = client.target(BASE_URI + uri);

      HttpAuthenticationFeature feature = HttpAuthenticationFeature
         .basic(user, password);
      target.register(feature);

      Message message = new Message();
      message.setText(text);
      Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
      Response response = target.request().put(entity);
      status(response);
   }

   private static void delete(String uri, String user, String password) {
      System.out.println("\n--- DELETE " + BASE_URI + uri);
      WebTarget target = client.target(BASE_URI + uri);

      HttpAuthenticationFeature feature = HttpAuthenticationFeature
         .basic(user, password);
      target.register(feature);

      Response response = target.request().delete();
      status(response);
   }

   private static int status(Response response) {
      int code = response.getStatus();
      String reason = response.getStatusInfo().getReasonPhrase();
      System.out.println("Status: " + code + " " + reason);
      return code;
   }
}
```

Der Server setzt die *Logging-Eigenschaften* mit:

```
System.setProperty("java.util.logging.config.file", "service/logging.properties");
```

Aufruf des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server
```

Aufruf des Clients:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
JerseyClient
```

Inhalt von *client.log* (Beispiel):

```
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: GET http://localhost:8080/demo/messages/1
   Response: 200 OK, MediaType: application/json
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: GET http://localhost:8080/demo/messages
   Response: 200 OK, MediaType: application/json
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: POST http://localhost:8080/demo/messages
   Response: 201 Created, Location: http://localhost:8080/demo/messages/4
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: PUT http://localhost:8080/demo/messages/1
   Response: 200 OK
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: DELETE http://localhost:8080/demo/messages/2
   Response: 200 OK
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyClientLogFilter filter
INFO: Request: GET http://localhost:8080/demo/messages/limited?n=2
   Response: 200 OK, MediaType: application/json
```

Inhalt von *service.log* (Beispiel):

```
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer getMessageAsJson: 5756842 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:getMessageAsJson
   Request: GET http://localhost:8080/demo/messages/1
   Response: 200 OK, MediaType: application/json
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer getAllMessagesAsJson: 1432100 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:getAllMessagesAsJson
   Request: GET http://localhost:8080/demo/messages
   Response: 200 OK, MediaType: application/json
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer newMessage: 24694133 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:newMessage
   Request: POST http://localhost:8080/demo/messages
   Response: 201 Created, Location: http://localhost:8080/demo/messages/4
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer updateMessage: 516346 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:updateMessage
   Request: PUT http://localhost:8080/demo/messages/1
   Response: 200 OK
```

```
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer deleteMessage: 179358 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:deleteMessage
   Request: DELETE http://localhost:8080/demo/messages/2
   Response: 200 OK
Feb. 06, 2019 2:16:52 NACHM. demo.filter.Timer2 filter
INFO: Dauer getAllMessagesAsJsonLimited: 3565434 ns
Feb. 06, 2019 2:16:52 NACHM. demo.filter.MyLogFilter filter
INFO: demo.messages.MessageResource:getAllMessagesAsJsonLimited
   Request: GET http://localhost:8080/demo/messages/limited
   Response: 200 OK, MediaType: application/json
```

10.12 Rollenbasierter Zugriffsschutz und Verschlüsselung

Im folgenden Beispiel, in dem wiederum Nachrichten wie im vorhergehenden Kapitel verwaltet werden, soll jeder Client-Zugriff über User und Passwort geschützt werden. Zudem soll anhand der Rolle, die ein User hat, der Zugriff für bestimmte Methoden erlaubt bzw. verboten werden. In einer erweiterten Fassung soll die Verbindung mittel SSL/TLS verschlüsselt werden.

→ **Projekt security**

Rollenkonzept und Security Context

Abbildung 10-8: Klassendiagramm

Die Klasse `User` verwaltet Namen, Passwort und Rollen.

Die Klasse `AuthService` verwaltet User mit Namen, Passwort und optionalen Rollen. Die Methode `authenticate` prüft, ob ein User registriert ist und sich mit dem richtigen Passwort ausgewiesen hat.

Die Klasse `MySecurityContext` implementiert `SecurityContext` und liefert einige sicherheitsrelevante Informationen.

Der Request-Filter `MyAuthFilter` prüft den *Authorization-Header* und setzt im positiven Fall den *Security Context*.

```
package demo.auth;

import java.security.Principal;
import java.util.List;

public class User implements Principal {
   private String name;
   private String password;
   private List<String> roles;

   public User(String name, String password, String... roles) {
      this.name = name;
      this.password = password;
      this.roles = List.of(roles);
   }

   @Override
   public String getName() {
      return name;
   }

   public String getPassword() {
      return password;
   }

   public List<String> getRoles() {
      return roles;
   }
}
```

```
package demo.auth;

import java.util.HashMap;
import java.util.Map;

public class AuthService {
   private static final Map<String, User> map = new HashMap<>();

   static {
      map.put("hugo", new User("hugo", "oguh", "admin"));
      map.put("emil", new User("emil", "lime", "user"));
   }

   public static User authenticate(String name, String pw) {
      User user = map.get(name);
      if (user != null) {
         if (user.getPassword().equals(pw))
            return user;
      }
      return null;
   }
}
```

```
package demo.auth;

import javax.ws.rs.core.SecurityContext;
import java.security.Principal;

public class MySecurityContext implements SecurityContext {
   private User user;
   private String scheme;

   public MySecurityContext(User user, String scheme) {
      this.user = user;
      this.scheme = scheme;
   }

   @Override
   public String getAuthenticationScheme() {
      return SecurityContext.BASIC_AUTH;
   }

   @Override
   public Principal getUserPrincipal() {
      return this.user;
   }


   @Override
   public boolean isSecure() {
      return scheme.equals("https");
   }

   @Override
   public boolean isUserInRole(String role) {
      if (user.getRoles() != null) {
         return user.getRoles().contains(role);
      }
      return false;
   }
}
```

```
package demo.auth;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.Response;
import java.io.IOException;
import java.util.Base64;

@PreMatching
public class MyAuthFilter implements ContainerRequestFilter {
   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      String auth = requestContext.getHeaderString("Authorization");
      if (auth == null) {
         String message = "Header 'Authorization' fehlt";
         System.err.println(message);
         requestContext.abortWith(Response.status(
```

```
            Response.Status.UNAUTHORIZED.getStatusCode(), message).build());
         return;
      }

      byte[] b = Base64.getDecoder().decode(auth.substring(6));
      String[] parts = new String(b).split(":");
      String username = parts[0];
      String password = parts[1];

      User user = AuthService.authenticate(username, password);
      if (user == null) {
         String message = "User '" + username +
                                 "' existiert nicht oder falsches Passwort";
         System.err.println(message);
         requestContext.abortWith(Response.status(
            Response.Status.UNAUTHORIZED.getStatusCode(), message).build());
         return;
      }

      String scheme = requestContext.getUriInfo().getRequestUri().getScheme();
      requestContext.setSecurityContext(new MySecurityContext(user, scheme));
   }
}
```

Die Annotation `@PreMatching` sorgt dafür, dass der Request-Filter ausgeführt wird, bevor die Zuordnung (Matching) zur geeigneten Ressource und Methode erfolgt.

```
package demo.messages;

import javax.annotation.security.RolesAllowed;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.UriInfo;
import java.net.URI;
import java.time.Instant;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.Collectors;

@Path("messages")
public class MessageResource {
   private static Map<Integer, Message> map;
   private static AtomicInteger counter = new AtomicInteger();
```

```
static {
   map = new ConcurrentHashMap<Integer, Message>();
   Instant instant = Instant.now();
   map.put(1, new Message(1, "Nachricht A", instant));
   map.put(2, new Message(2, "Nachricht B", instant.plusSeconds(1)));
   map.put(3, new Message(3, "Nachricht C", instant.plusSeconds(2)));
   counter.set(3);
}

@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getMessage(@PathParam("id") int id) {
   Message message = map.get(id);
   if (message == null) {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   }
   return Response.ok(message).build();
}

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Message> getAllMessage() {
   StringBuilder stringBuilder = new StringBuilder();
   List<Message> list = map.values().stream()
         .sorted((m1, m2) -> -m1.getTimestamp().compareTo(m2.getTimestamp()))
         .collect(Collectors.toList());
   return list;
}

@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response newMessage(@Context UriInfo uriInfo,
                           @Context SecurityContext sec, Message message) {
   if (sec.isUserInRole("user")) {
      System.out.println("Neue Nachricht von User '" +
            sec.getUserPrincipal().getName() + "'");
   }

   int id = counter.incrementAndGet();
   message.setId(id);
   message.setTimestamp(Instant.now());
   map.put(id, message);
   URI location = uriInfo.getAbsolutePathBuilder().path(String.valueOf(id))
      .build();
   return Response.created(location).build();
}

@PUT
@Path("{id}")
@Consumes(MediaType.APPLICATION_JSON)
@RolesAllowed("admin")
public Response updateMessage(@PathParam("id") int id, Message message) {
   if (map.get(id) == null) {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   } else {
      message.setId(id);
      message.setTimestamp(Instant.now());
```

```
            map.put(id, message);
            return Response.noContent().status(Response.Status.OK).build();
        }
    }

    @DELETE
    @Path("{id}")
    @RolesAllowed("admin")
    public Response deleteMessage(@PathParam("id") int id) {
        if (map.get(id) != null) {
            map.remove(id);
            return Response.noContent().status(Response.Status.OK).build();
        } else {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        }
    }
}
```

In der Methode `newMessage` wird der im Filter gesetzte *Security Context* als Parameter injiziert. Wenn der User, der den Request gesendet hat, der Rolle *user* zugeordnet ist, wird eine bestimmte Information auf der Konsole des Servers ausgegeben.

Die beiden Methoden `updateMessage` und `deleteMessage` dürfen nur von Usern mit der Rolle *admin* ausgeführt werden.

Hierfür sorgt die Annotation `@RolesAllowed("admin")`. In der Klasse `MyApplication` muss auch eigens das Feature

`org.glassfish.jersey.server.filter.RolesAllowedDynamicFeature`

registriert werden:

```
    public MyApplication() {
        singletons.add(new MessageResource());
        singletons.add(new MyAuthFilter());
        singletons.add(new RolesAllowedDynamicFeature());
    }
```

Für den Test sind zwei User in `AuthService` eingerichtet.

Name: `hugo`, Passwort: `oguh`, Rolle: `admin`
Name: `emil`, Passwort: `lime`, Rolle: `user`

```
import org.glassfish.jersey.client.authentication.HttpAuthenticationFeature;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.List;

public class JerseyClient1 {
    private static final String BASE_URI = "http://localhost:8080/demo";
```

```
private static Client client;

public static void main(String[] args) {
   // setup("hugo", "oguh");
   setup("emil", "lime");

   get("/messages/1");
   get("/messages");

   post("/messages", "Neue Nachricht");
   put("/messages/1", "Nachricht AA");
   delete("/messages/2");
}

private static void setup(String user, String password) {
   client = ClientBuilder.newClient();
   HttpAuthenticationFeature feature = HttpAuthenticationFeature
      .basic(user, password);
   client.register(feature);
}

private static void get(String uri) {
   System.out.println("\n--- GET " + BASE_URI + uri);
   WebTarget target = client.target(BASE_URI + uri);
   Response response = target.request().accept(MediaType.APPLICATION_JSON)
      .get();
   if (status(response) == 200) {
      if (uri.endsWith("messages") || uri.contains("limited")) {
         List<Message> list = response
            .readEntity(new GenericType<List<Message>>() {});
         for (Message message : list) {
            System.out.println(message.getId() + ";" +
                    message.getText() + ";" + message.getTimestamp());
         }
      } else {
         Message message = response.readEntity(Message.class);
         System.out.println(message.getId() + ";" +
               message.getText() + ";" + message.getTimestamp());
      }
   }
}

private static void post(String uri, String text) {
   System.out.println("\n--- POST " + BASE_URI + uri);
   WebTarget target = client.target(BASE_URI + uri);
   Message message = new Message();
   message.setText(text);
   Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
   Response response = target.request().post(entity);
   if (status(response) == 201) {
      String location = response.getLocation().toString();
      System.out.println("Location: " + location);
   }
}

private static void put(String uri, String text) {
   System.out.println("\n--- PUT " + BASE_URI + uri);
   WebTarget target = client.target(BASE_URI + uri);
   Message message = new Message();
```

```
        message.setText(text);
        Entity<Message> entity = Entity.entity(message, MediaType.APPLICATION_JSON);
        Response response = target.request().put(entity);
        status(response);
    }

    private static void delete(String uri) {
        System.out.println("\n--- DELETE " + BASE_URI + uri);
        WebTarget target = client.target(BASE_URI + uri);
        Response response = target.request().delete();
        status(response);
    }

    private static int status(Response response) {
        int code = response.getStatus();
        String reason = response.getStatusInfo().getReasonPhrase();
        System.out.println("Status: " + code + " " + reason);
        return code;
    }
}
```

Das Feature `HttpAuthenticationFeature` wird wie in Kapitel 10.11 registriert, hier allerdings beim `javax.ws.rs.client.Client`. Es wirkt damit für alle Requests.

Start des Servers:

```
java --add-opens java.base/java.lang=ALL-UNNAMED
-cp out/production/service;../../libs/grizzly/*;../../libs/jaxrs/*;
../../libs/activation/* Server1
```

Start des Clients:

```
java -cp out/production/client;../../libs/jaxrs/*;../../libs/activation/*
JerseyClient1
```

Wird der Client mit dem Usernamen *emil* und dem Passwort *lime* ausgeführt, erhält man das folgende Ergebnis.

Beispiel:

```
--- GET http://localhost:8080/demo/messages/1
Status: 200 OK
1;Nachricht A;2018-12-22T16:30:52.797202600Z

--- GET http://localhost:8080/demo/messages
Status: 200 OK
3;Nachricht C;2019-02-06T13:46:02.019881600Z
2;Nachricht B;2019-02-06T13:46:01.019881600Z
1;Nachricht A;2019-02-06T13:46:00.019881600Z

--- POST http://localhost:8080/demo/messages
Status: 201 Created
Location: http://localhost:8080/demo/messages/4
```

```
--- PUT http://localhost:8080/demo/messages/1
Status: 403 Forbidden

--- DELETE http://localhost:8080/demo/messages/2
Status: 403 Forbidden
```

Beim Server wird ausgegeben:

```
Neue Nachricht von User 'emil'
```

Sichere Verbindung mit SSL/TLS

Wir wollen nun die Verbindung zwischen Client und Server mittels *SSL/TLS* verschlüsseln. Dazu gehen wir wie in Kapitel 7.5 vor:

- Schlüsselpaar (public key/private key) erstellen
- Zertifikat exportieren
- Zertifikat in den Truststore des Client aufnehmen

Das Begleitmaterial enthält die hierzu benötigten Skripte.

`Server2` läuft auf Port 8443 und ist für SSL/TLS konfiguriert.

```
import demo.messages.MyApplication;
import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.grizzly.ssl.SSLContextConfigurator;
import org.glassfish.grizzly.ssl.SSLEngineConfigurator;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server2 {
   public static void main(String[] args) throws Exception {
      Logger.getLogger("org.glassfish").setLevel(Level.SEVERE);

      URI baseUri = new URI("https://0.0.0.0:8443/demo");
      ResourceConfig config = ResourceConfig
         .forApplicationClass(MyApplication.class);

      SSLContextConfigurator sslContext = new SSLContextConfigurator();
      sslContext.setKeyStoreFile("keystore.jks");
      sslContext.setKeyStorePass("secret");
      SSLEngineConfigurator sslEngine = new SSLEngineConfigurator(
         sslContext).setClientMode(false).setNeedClientAuth(false);
      HttpServer server = GrizzlyHttpServerFactory
         .createHttpServer(baseUri, config, true, sslEngine);

      System.out.println("https://localhost:8443/demo/");
      System.out.println("ENTER stoppt den Server.");
      System.in.read();
```

```
        server.shutdownNow();
    }
}
```

Mit Hilfe des `SSLContextConfigurator` werden Keystore-Datei und -Passwort gesetzt. `SSLEngineConfigurator` konfiguriert die *SSLEngine*. Hier muss nur der Server seine Identität gegenüber dem Client beweisen.

Die wesentliche Änderung in `JerseyClient2` gegenüber `JerseyClient1` besteht in:

```
private static void setup(String user, String password) {
    // System.setProperty("javax.net.debug", "ssl:record:handshake");
    System.setProperty("javax.net.ssl.trustStore", "certs.jks");
    System.setProperty("javax.net.ssl.trustStorePassword", "secret");
    HttpAuthenticationFeature feature = HttpAuthenticationFeature
        .basic(user, password);
    client = ClientBuilder.newClient();
    client.register(feature);
}
```

10.13 Aufgaben

1. Durch eine Hash-Funktion wird eine Nachricht auf einen Hashwert mit einer festen Länge abgebildet. In Java steht hierfür die Klasse `java.security.MessageDigest` zur Verfügung.

   Entwickeln Sie einen REST-Service, der für einen vorgegebenen Algorithmus (z. B. MD5, SHA-1, SHA-256 oder SHA-512) und einen Text den Hashwert berechnet und als Antwort hexadezimal codiert liefert. Der Client sendet die Daten im URL-codierten Format über POST.

   Beispiel:
   `text=Das+ist+ein+Test.&algorithm=MD5`

   Entwickeln Sie auch einen Java-Client mit dem JAX-RS Client API sowie ein HTML-Formular.

2. Mit GET sollen vom Client zufällige Sensor-Werte abgefragt werden können. Hierzu soll die folgende Ressourcenmethode zur Verfügung gestellt werden:

   ```
   @GET
   @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   public SensorValue getSensorValue() {
       return new SensorValue();
   }
   ```

   Je nach eingestelltem Medientyp (Methode `accept`) wird das Ergebnis im JSON- oder im XML-Format geliefert.

   Die Klasse `SensorValue` ist vorgegeben:

   ```
   package demo.sensor;

   import javax.xml.bind.annotation.XmlElement;
   import javax.xml.bind.annotation.XmlRootElement;
   ```

```
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.time.Instant;

@XmlRootElement
public class SensorValue {
  @XmlElement
  @XmlJavaTypeAdapter(MyInstantAdapter.class)
  public Instant time;
  public double value;

  public SensorValue() {
     time = Instant.now();
     value = Math.random() * 100;
  }

  @Override
  public String toString() {
     return "SensorValue{" +
            "time=" + time +
            ", value=" + value +
            '}';
  }
}
```

Je nach eingestelltem Medientyp (Methode `accept`) wird das Ergebnis im JSON- oder im XML-Format geliefert. Für die Konvertierung Objekt ↔ XML-Dokument muss das Attribut `time` vom Typ `Instant` mit einem eigenen Adapter ausgestattet werden (siehe obige Annotationen):

```
package demo.sensor;

import javax.xml.bind.annotation.adapters.XmlAdapter;
import java.time.Instant;

public class MyInstantAdapter extends XmlAdapter<String, Instant> {
  @Override
  public Instant unmarshal(String s) throws Exception {
     return s != null ? Instant.parse(s) : null;
  }

  @Override
  public String marshal(Instant instant) throws Exception {
     return instant != null ? instant.toString() : null;
  }
}
```

`unmarshal` konvertiert eine geeignete Zeichenkette in eine `Instant`-Instanz, `marshal` eine `Instant`-Instanz in eine Zeichenkette.

Beachten Sie, dass zur Ausführung des Servers die Bibliothek *libs/jaxb* in den Klassenpfad eingebunden werden muss (siehe Kapitel 10.2).

3. In Apache Tomcat können Datenbanken als *JNDI Datasource* konfiguriert werden. Tomcat arbeitet mit einem *Database Connection Pool*, um die Client-bezogenen Verbindungen zur Datenbank optimal zu verwalten. Ändern Sie hierzu das Projekt *contacts* aus Kapitel 10.3 (zur Nutzung von Tomcat siehe Kapitel 10.4).

Im Verzeichnis *web/META-INF* ist die Datei *context.xml* abgelegt:

```
<Context>
   <JarScanner scanManifest="false"/>
   <Resource url="jdbc:h2:~/contacts_db" driverClassName="org.h2.Driver"
        maxWaitMillis="10000" maxIdle="30"maxTotal="100"
        type="javax.sql.DataSource" auth="Container" name="jdbc/myDB"/>
</Context>
```

Sie enthält Angaben zur Datenbank und zum Pool.
Die Methode `getConnection` in `DBManager` sieht nun wie folgt aus:

```
private static Connection getConnection()
     throws NamingException, SQLException {
  Context ctx = new InitialContext();
  DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/myDB");
  return ds.getConnection();
}
```

Beim Server muss die noch folgende Anweisung eingefügt werden (siehe Kapitel 10.4):

```
tomcat.enableNaming();
```

4. Diese Aufgabe soll die Möglichkeit von so genannten *partiellen Updates* und *Multi-Updates* demonstrieren. Ein REST-Service soll Aufträge (`Order`) in einer Map verwalten.

   Die Klasse `Order` enthält die Attribute:

   ```
   int orderId
   int customerId
   LocalDateTime timeCreated
   LocalDateTime timeChanged
   List<Item> items
   ```

   Die Klasse `Item` enthält die Attribute:

   ```
   int productId
   int quantity
   ```

   `timeCreated` wird bei Neuanlage eines Auftrags gesetzt, `timeChanged` bei jeder Änderung.

   Der Service soll die folgenden Ressourcen (HTTP-Methode, URI) anbieten:

   - Neuanlage eines Auftrags:
     `POST orders`
   - Lesen eines Auftrags:
     `GET orders/{orderId}`
   - Ändern einer Auftragsposition (partieller Update):
     `PUT orders/{orderId}/items/{position}`
   - Ändern aller Auftragspositionen (Multi-Update):
     `PUT orders/{orderId}/items`

   Entwickeln Sie Client und Server.

5. Der REST-Service aus dem Projekt *security* in Kapitel 10.11 soll mit *Apache Tomcat* angeboten werden. SSL/TLS ist für Tomcat so wie im Projekt *filter*

aus Kapitel 9.8 zu konfigurieren. Basic Authentication, User und Rollen sollen auch direkt im Server konfiguriert werden, sodass hierfür insbesondere auch keine Filter benötigt werden.

Das Verzeichnis *web/META-INF* enthält die Datei *context.xml*:

```
<Context>
    <JarScanner scanManifest="false"/>
</Context>
```

Die Klasse `MyApplication` wird mit `@ApplicationPath("rest")` annotiert.

Hier ist der Quellcode des Servers:

```
import org.apache.catalina.Context;
import org.apache.catalina.Service;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;
import org.apache.tomcat.util.descriptor.web.LoginConfig;
import org.apache.tomcat.util.descriptor.web.SecurityCollection;
import org.apache.tomcat.util.descriptor.web.SecurityConstraint;

import java.io.File;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
  public static void main(String[] args) throws Exception {
     int port = 8443;
     String contextPath = "demo";
     String docBase = "web";
     String keystoreFile = "keystore.jks";
     String keystorePass = "secret";

     Logger.getLogger("").setLevel(Level.SEVERE);

     Tomcat tomcat = new Tomcat();
     tomcat.setBaseDir(System.getProperty("java.io.tmpdir"));
     Context ctx = tomcat.addWebapp(contextPath,
        new File(docBase).getAbsolutePath());

     Connector con = new Connector();
     con.setPort(port);
     con.setScheme("https");
     con.setSecure(true);
     con.setAttribute("protocol", "HTTP/1.1");
     con.setAttribute("SSLEnabled", true);
     con.setAttribute("maxThreads", "150");
     con.setAttribute("clientAuth", "false");
     con.setAttribute("sslProtocol", "TLS");
     con.setAttribute("keystoreFile",
        new File(keystoreFile).getAbsolutePath());
     con.setAttribute("keystorePass", keystorePass);

     Service service = tomcat.getService();
     service.addConnector(con);

     // Rollen und User
     tomcat.addRole("hugo", "admin");
     tomcat.addRole("emil", "user");
```

```
        tomcat.addUser("hugo", "oguh");
        tomcat.addUser("emil", "lime");

        // Basic Authentication
        LoginConfig config = new LoginConfig();
        config.setAuthMethod("BASIC");
        ctx.setLoginConfig(config);

        // Security Roles
        ctx.addSecurityRole("admin");
        ctx.addSecurityRole("user");

        // Security Constraint
        SecurityConstraint constraint = new SecurityConstraint();
        constraint.addAuthRole("admin");
        constraint.addAuthRole("user");
        SecurityCollection collection = new SecurityCollection();
        collection.addPattern("/*");
        constraint.addCollection(collection);
        ctx.addConstraint(constraint);

        tomcat.start();

        System.out.println("DocBase: " + ctx.getDocBase());
        String url = con.getScheme() + "://localhost:" +
           con.getPort() + ctx.getPath();
        System.out.println("URL: " + url);
        System.out.println("ENTER drücken, um den Server zu stoppen.");

        System.in.read();
        tomcat.stop();
        tomcat.destroy();
    }
}
```

6. Realisieren Sie einen REST-Service zur Verwaltung von Aufgaben (Todos). Neben einer eindeutigen Id soll eine Aufgabe den Usernamen, die Beschreibung der Aufgabe und den Status für die Erledigung (`true`/`false`) enthalten. Die Daten sollen in einer von Apache Tomcat verwalteten Datenbank gespeichert werden (siehe Aufgabe 3). Jeder User soll seine Aufgaben verwalten können und keinen Zugriff auf Aufgaben anderer User haben. Nutzen Sie die Konfiguration der HTTP-Authentifizierung und von SSL/TLS aus Aufgabe 5. Innerhalb der Java-Anwendung kann der aktuelle User über den *Security Context* abgefragt werden. Die Datenbankzugriffsmethoden müssen den User berücksichtigen (WHERE-Klausel in SQL).

   Entwickeln Sie auch einen Client mit dem JAX-RS Client API.

7. Realisieren Sie die optimistische Nebenläufigkeitskontrolle (siehe Kapitel 10.7) mit Hilfe einer im `Order`-Objekt gespeicherten Versionsnummer, die bei jeder Änderung um 1 hochgezählt wird.

8. Diese Aufgabe soll domänenübergreifende Zugriffe mit *CORS* demonstrieren. Betrachten Sie das in der folgenden Abbildung dargestellte Szenario:

Ein REST-Server (A) bietet einen Service zur Verwaltung von Nachrichten an (siehe REST-Service aus Kapitel 10.2). Über einen Browser wird eine Single-Page-Webanwendung von einem Webserver (B) geladen (HTML-Seite, CSS, JavaScript). Daten (im JSON-Format) sollen mittels AJAX-Requests bzw. *fetch* vom REST-Server abgefragt bzw. zu diesem gesendet werden (vgl. die Webanwendung im Projekt *messages* des Kapitels 10.2).

Als Webserver (B) kann ein Server analog zum Server aus Kapitel 7.2 (Projekt *webserver2*) eingesetzt werden.

Nach Starten von REST-Server auf Port 8080 und Webserver auf Port 80 kann im Browser diese Seite abgerufen werden: `http://localhost/demo/`

Nun erhält man einen Fehler, der in der Web-Konsole (z. B. von Google Chrome) erscheint:

No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost' is therefore not allowed access.

Solche domänenübergreifenden Zugriffe (das Skript stammt von B, soll aber Daten auf A abfragen bzw. ändern) sind in der Regel aus Sicherheitsgründen untersagt (*Same-Origin-Policy*).

Cross-Origin Resource Sharing (*CORS*) ist ein Verfahren, das diese Einschränkungen aufheben kann. Der REST-Server (A) kann hierzu bestimmte Header der Art `Access-Control-*` in seiner Antwort setzen.

Erstellen Sie einen Response-Filter, den Sie für den REST-Server in `MyApplication` registrieren.

```
package demo.messages;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.core.MultivaluedMap;
import java.io.IOException;

public class MyCORSFilter implements ContainerResponseFilter {
  @Override
  public void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext) throws IOException {
```

```
        MultivaluedMap<String, Object> map = responseContext.getHeaders();
        map.add("Access-Control-Allow-Origin", "*");
        map.add("Access-Control-Allow-Methods", "GET,POST,PUT,DELETE");
        map.add("Access-Control-Allow-Headers", "Content-Type");
        map.add("Access-Control-Expose-Headers", "Location");
    }
}
```

Für eine komplexere Anfrage (z. B. mit der HTTP-Methode PUT oder DELETE) wird eine zusätzliche Anfrage mit der HTTP-Methode OPTIONS gestellt, um herauszufinden, ob die geplante Anfrage erlaubt werden darf oder nicht. Aus diesem Grund muss zusätzlich die Bibliothek *libs/jaxb* zur Ausführung des REST-Servers verwendet werden.

Nach Neustart funktioniert die Anwendung fehlerfrei.

9. Um den Inhalt einer sehr großen Datei über einen REST-Service zu lesen, kann der Inhalt der Datei in Seiten einer bestimmten Größe aufgeteilt werden, die dann einzeln angefordert werden können, z. B. mit Query-Parametern:

```
GET /messages?page=3&count=10 HTTP/1.1
```

Testen Sie diese Technik mit Hilfe einer Datenbank, die eine Tabelle mit sehr vielen Datenzeilen enthält. Testdaten können über ein einfaches JDBC-Programm generiert werden.

In H2 können Daten seitenweise wie im folgenden Beispiel abgefragt werden:

```
select message from messages order by id limit 10 offset 20
```

Der Zusammenhang zwischen `page` und `offset` bzw. `count` und `limit` ergibt sich wie folgt:

```
limit = count
offset = (page – 1) * count
```

Anhang A

Tools und Bibliotheken

Die folgende Tabelle enthält die in diesem Buch verwendeten Tools und Bibliotheken mit den zum Zeitpunkt der Drucklegung aktuellen Versionen. In der zweiten Spalte dieser Tabelle sind die Verzeichnisse angegeben, die im Begleitmaterial zu diesem Buch vorhanden sind und die nötigen jar-Dateien enthalten.

Tools/Bibliotheken	Im Begleitmaterial unter
Java 11 (JDK) `https://www.oracle.com/technetwork/java/javase/downloads/`	
JavaFX 11 `https://openjfx.io/`	
IntelliJ IDEA 2018.3.4 `https://www.jetbrains.com/idea/`	
Apache ActiveMQ 5.15.8 `http://activemq.apache.org`	
cURL 7.63.0 `https://curl.haxx.se/`	
Postman 6.7.3 `https://www.getpostman.com/`	

Eclipse Yasson 1.0, Implementierung von Java API for JSON Binding
`http://json-b.net/`
`https://projects.eclipse.org/projects/ee4j.yasson`
	libs/json

Grizzly HTTP Server 2.4.4
`https://mvnrepository.com/artifact/org.glassfish.grizzly/grizzly-http-all`
Jersey Container Grizzly2 HTTP 2.28
`https://mvnrepository.com/artifact/org.glassfish.jersey.containers/jersey-container-grizzly2-http`
	libs/grizzly

Apache Tomcat 9.0.14 Embedded `https://tomcat.apache.org/download-90.cgi`	*libs/tomcat_embed*

Java API for XML Web Services, JAX-WS RI 2.3.0
`https://javaee.github.io/metro-jax-ws/`
`https://github.com/eclipse-ee4j/metro-jax-ws`
	libs/jaxws-ri

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9

Tools/Bibliotheken	Im Begleitmaterial unter
JavaBeans Activation Framework (JAF) 1.2.0 `https://mvnrepository.com/artifact/javax.activation/javax.activation-api/1.2.0`	*libs/activation*

Jersey 2.28, Implementierung von JAX-RS 2.1
`https://jersey.github.io/`
`https://jersey.github.io/apidocs/latest/jersey/index.html`
`https://jersey.github.io/documentation/latest/index.html`
`https://projects.eclipse.org/projects/ee4j.jersey`
	libs/jaxrs

Java Architecture for XML Binding (JAXB) 2.3.0 `https://javaee.github.io/jaxb-v2/`	*libs/jaxb*
H2 Database 1.4.197 `http://www.h2database.com`	*libs/h2-1.4.197.jar*

Jersey Media Multipart 2.28 und MIME Streaming Extension 1.9.11
`https://mvnrepository.com/artifact/org.glassfish.jersey.media/jersey-media-multipart`
`https://mvnrepository.com/artifact/org.jvnet.mimepull/mimepull`
	libs/media_multipart

OkHttp 3.12.1 und Okio 2.2.2 `http://square.github.io/okhttp`	*libs/okhttp*

Die nachfolgende Tabelle, zeigt welche Tools bzw. Bibliotheken in den einzelnen Kapiteln des Buches verwendet werden.

Kapitel	Tools/Bibliotheken
Kapitel 2	*libs/json*
Kapitel 3	*libs/json* *JavaFX*
Kapitel 4	*libs/json* *JavaFX*
Kapitel 5	*libs/json* *ActiveMQ*
Kapitel 6	*libs/grizzly (HTTP Server)*

Kapitel	Tools/Bibliotheken
Kapitel 7	*libs/grizzly (HTTP Server)* *libs/json* *libs/tomcat_embed*
Kapitel 8	*libs/json* *libs/tomcat_embed*
Kapitel 9	*libs/activation* *libs/jaxws-ri* *libs/tomcat_embed* *JavaFX*
Kapitel 10	*libs/activation* *libs/grizzly* *libs/h2-1.4.197.jar* *libs/jaxb* *libs/jaxrs* *libs/json* *libs/media_multipart* *libs/okhttp* *libs/tomcat_embed*

Anhang B

Globale Bibliotheken in IntelliJ IDEA

Die im Anhang A aufgeführten Verzeichnisse können in der Entwicklungsumgebung *IntelliJ IDEA* als *Globale Bibliotheken* eingerichtet werden. Diese stehen projektübergreifend zur Verfügung und können den einzelnen Projekten/Modulen bequem zugewiesen werden.

Das folgende Beispiel zeigt die generelle Vorgehensweise hierzu. Wir richten die globale Bibliothek mit dem Namen *json* ein und weisen dieser das Verzeichnis *libs/json* zu. Anschließend binden wir diese globale Bibliothek in das Projekt *jsonb* aus Kapitel 2 als Abhängigkeit (Dependency) ein.

New Global Library (+) anklicken, dann das Verzeichnis *libs\json* im Dateidialog auswählen und den Namen ggf. anpassen (*json*). Im Beispiel liegt *libs\json* im Verzeichnis *D:\Test\MKJava5*.

Dann das Modul *jsonb* wählen, *Add* (+) anklicken ...

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9

… und die Library *json* auswählen.

Anhang C

IntelliJ-IDEA-Projekte in Eclipse-Projekte konvertieren

IntelliJ IDEA unterstützt die Konvertierung eines Projekts in ein *Eclipse*-Projekt (siehe *File > Export Project to Eclipse...*). Allerdings muss für jedes Modul (Unterprojekt) auf diesem Weg ein eigenes Eclipse-Projekt erzeugt werden.

Es wird auch eine Datei erzeugt, die die *Global Libraries* aufführt. Diese Einträge können in Eclipse als *User Libraries* im entsprechenden Dialog unter *Window > Preferences* importiert werden.

Die entstandenen Eclipse-Projekte sollten umbenannt werden, sofern sie aus einem Unterprojekt (Modul) in IntelliJ IDEA stammen, denn die Modulnamen sind nur innerhalb des IntelliJ-IDEA-Projekts eindeutig.

Beispiel:

Projekt *P1* hat die beiden Module *client* und *service*. Dann sollten die beiden Eclipse-Projekte wie folgt bezeichnet werden: *P1-client* und *P1-service*.

Skripte zum Ausführen der Programme sind in die entsprechenden Projekte zu kopieren und anzupassen.

Hier ist der Ausgabeordner für Bytecodes *out/production/…* in *bin* zu ändern.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9

Literaturhinweise

Die folgenden Quellen sind für eine Vertiefung einzelner Themen dieses Buches gut geeignet.

Abts, D.: Grundkurs Java. Von den Grundlagen bis zu Datenbank- und Netzanwendungen. Springer Vieweg, 10. Auflage 2018

Ackermann, P.: Professionell entwickeln mit JavaScript: Design, Patterns und Praxistipps. Rheinwerk Computing, 2. Auflage 2018

Baun, C.: Computernetze kompakt. Springer Vieweg, 4. Auflage 2018

Bengel, G.; Baun, C.; Kunze, M.; Stucky, K.-U.: Masterkurs Parallele und Verteilte Systeme. Springer Vieweg, 2. Auflage 2015

Epple, A.: JavaFX 8: Grundlagen und fortgeschrittene Techniken. dpunkt.verlag 2015

Goll, J: Architektur- und Entwurfsmuster der Softwaretechnik. Springer Vieweg, 2. Auflage 2014

Gorski, P. L.; Lo Iacono, L.; Nguyen, H. V.: WebSockets. Moderne HTML5-Echtzeitanwendungen entwickeln. Hanser 2015

Harold, E. R.: Java Network Programming. O'Reilly Media, 4. Auflage 2013

Hettel, J.; Tran, M. T.: Nebenläufige Programmierung mit Java. dpunkt.verlag 2016

JDBC Database Access Tutorial: `http://docs.oracle.com/javase/tutorial/jdbc/`

Oechsle, R.: Parallele und verteilte Anwendungen in Java. Hanser, 5. Auflage 2018

Plenk, V.: Angewandte Netzwerktechnik kompakt: Dateiformate, Übertragungsprotokolle und ihre Nutzung in Java-Applikationen. Springer Vieweg 2017

Tilkov, S. u. a.: REST und HTTP. dpunkt.verlag, 3. Auflage 2015

Unterstein. M., Matthiessen, G.: Anwendungsentwicklung mit Datenbanken. Springer Vieweg, 5. Auflage 2013

Unterstein. M., Matthiessen, G.: Relationale Datenbanken und SQL in Theorie und Praxis. Springer Vieweg, 5. Auflage 2012

Wolf, J.: HTML5 und CSS3: Das umfassende Handbuch. Rheinwerk Computing, 3. Auflage 2019

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019

D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9

Sachwortverzeichnis

@

@ApplicationPath 294
@Consumes 269
@Context 269, 340
@DefaultValue 268
@DELETE 267
@FormParam 269
@GET 267
@HandlerChain 246
@HeaderParam 309
@JsonbDateFormat 25
@JsonbNumberFormat 25
@JsonbProperty 25
@JsonbTransient 25
@Oneway 226
@Path 267
@POST 267
@PreMatching 357
@Priority 346
@Produces 267
@PUT 267
@QueryParam 268
@RolesAllowed 359
@WebMethod 216
@WebParam 216
@WebService 216
@XmlMimeType 237

A

Accept 268
Access-Control-* 368
ActiveMQ 90, 93, 94
Address Resolution Protocol 15
AlreadyBoundException 131
Anfrageparameter 156, 157
Antwortparameter 160
Anwendungsschicht 15
Apache ActiveMQ 90
Application 269
ARP 15
AsyncHandler 231
asynchron 9, 81, 89, 228
asynchrone Verarbeitung 321
AsyncResponse 333
Austauschbarkeit 209

B

backlog 56
Base64 26, 234, 345
base64Binary 233
BytesMessage 92

C

CacheControl 316
Cached Thread Pool 64
Caching 313
Callback 9, 144, 228, 321
CIDR 17
Classless Inter-Domain Routing 17
Client 7, 8, 279
Client/Server-Modell 7
Client/Server-System 7
ClientBuilder 279
ClientRequestContext 349
ClientRequestFilter 345
ClientResponseContext 349
ClientResponseFilter 345
Code First 251
CompletionCallback 333
ConcurrentModificationException 74, 145, 200
ConnectException 56
Connection 92, 95, 111, 162
ConnectionFactory 91, 95

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
https://doi.org/10.1007/978-3-658-25925-9

ContainerRequestContext 347
ContainerRequestFilter 345
ContainerResponseContext 348
ContainerResponseFilter 345
Content-Length 162
Content Negotiation 268
Content-Type 162
Context Root 270, 294
Contract First 251
CopyOnWriteArrayList 74, 145, 200, 203
CORS 367, 368
Cross-Origin Resource Sharing 368
CRUD 174
cURL 271

D
DatagramPacket 40
DatagramSocket 38, 39, 40, 44, 45
DataHandler 236
Date 161
Datenbank-Middleware 13
Datenhaltung 4
dauerhafter Subscriber 111
DELETE 157, 261
Destination 93
DHCP 38
DNS 17, 38
Domain Name System 17, 38
dreistufige Architektur 11
Dynamic Host Configuration Protocol 38

E
einstufige Architektur 9
Endpoint 217
Entity 280
Entity-Tag 308, 320
ETag 308, 320
Event-Stream-Format 341
Executors 64
ExecutorService 64
Expirationsmodell 313, 315

F
Fehlertoleranz 6
Fehlertransparenz 13
FIFO-Prinzip 121
FileNameMap 168
File Transfer Protocol 15, 53
FormDataBodyPart 298
FormDataContentDisposition 298
FormDataMultiPart 297
Framework 67
FTP 15, 53
Future 231

G
GenericType 280
Geschlossenheit 209
GET 157, 160, 260

H
H2 282
Handler 239
HandlerResolver 248
Handshake 191
HATEOAS 260
HEAD 261
Heterogenität 6
HPACK 182
HTTP 15, 53, 153, 210, 212
HTTP/1.0 154
HTTP/1.1 154
HTTP/2 155, 182
HTTP-Anfrage 153, 154, 156
HTTP-Antwort 154, 160
HttpAuthenticationFeature 345, 361

HTTP Basic Authentication 345
HttpClient 198
HTTP-Methode 156
HTTPS 184
Hypertext Transfer Protocol 15, 53, 153
HyperText Transfer Protocol Secure 184

I

idempotent 261
IETF 37, 53, 154, 158
If-Match 309
InboundSseEvent 343
Inet4Address 19
Inet6Address 19
InetAddress 19
InetSocketAddress 20
Internet 14
Internet Engineering Task Force 37
Internet Protocol 15
Interzeptor 239
Intranet 14
Invocation.Builder 280
InvocationCallback 337
IP-Adresse 16
IPv4 16
IPv6 17
iterativer Server 58

J

Java API for JSON Binding 24
Java API for RESTful Web Services 262
Java API for XML-Binding 220
Java API for XML Web Services 215
Java Message Service 90
Java Naming and Directory Interface 91
Java Remote Method Protocol 130
JavaScript Object Notation 23
JAXB 220
JAX-RS 262
JAX-WS 215
JDBC 13
Jersey 262
Jersey Media Multipart 299
JMS 90
JMS-Client 90
JMSDestination 103
JMSException 95
JMSExpiration 103
JMSMessageID 103
JMSPriority 103
JMS-Provider 90
JMSRedelivered 118
JMSReplyTo 105
JMSTimestamp 103
JNDI 91, 94
JRMP 130
JSON 23
Jsonb 24
JSON-B 24
JsonbBuilder 24
JsonbConfig 26

K

keytool 184

L

Last-Modified 162
Lastverteilung 6
localhost 18
LocateRegistry 133
Logical Handler 240
Long Polling 191
Loopback-Adresse 17
lose Kopplung 209, 212
Lost Update 307

M
MapMessage 92, 108, 109
MediaType 267, 269
mehrstufige Architektur 9
Message 91, 92, 102, 103
Message Broker 89
MessageConsumer 92, 96
Message Consumer 90
MessageListener 96
Message Oriented Middleware 14, 89
MessageProducer 92, 96
Message Producer 90
Message Selector 114
Message Transmission Optimization Mechanism 235
Middleware 12, 89
Migrationstransparenz 13
MIME 158
MIME Streaming Extension 299
mobiler Agent 140
MOM 14, 89
MOM-Server 89
MTOM 235, 236
Multicast 48
Multicast-Adresse 48
Multicast-Gruppe 48
Multicasting 48
MulticastSocket 49
Multimedia Streaming 38
MultiPartFeature 298
Multipurpose Internet Mail Extension 158
Multi-Tier-Architektur 9

N
Namensdienst 127
Naming 131, 132, 134
Nebenläufigkeitstransparenz 13
Netzinfrastruktur 6
NotBoundException 131, 132

O
OnClose 194
OnError 194
OnMessage 194
OnOpen 194
operationsorientiert 262
optimistische Nebenläufigkeitskontrolle 307
OPTIONS 261
Ortstransparenz 13, 209
OutboundSseEvent 340

P
Parallelbetrieb 8
paralleler Server 62
Plattformunabhängigkeit 210
Polling 144, 321
POP 53
Portnummer 18
POST 157, 158, 260
Postman 273
Post Office Protocol 53
Präsentation 4
Proxy 130, 172
Publish/Subscribe-Modell 108
Publisher 108, 144
Pull-Prinzip 98, 144, 228
Push-Prinzip 99, 144, 228
PUT 157, 260

Q
Query String 160
Queue 102
QueueBrowser 102

R
Redelivery 100
Registry 127, 131
Remote 126, 128

RemoteEndpoint.Basic 194
RemoteException 126, 128
Remote Interface 128
Remote Method Invocation 14, 123, 126
Remote Object 128
Remote Procedure Call 14, 123
Remote Reference 130
Replikationstransparenz 13
Repräsentation 260
Representational State Transfer 182, 259
Request/Reply-Modell 105
Request for Comments 37
Request-Multiplexing 182
ResourceInfo 348
Response 231, 268, 280
Response.StatusType 280
Ressource 259
ressourcenorientiert 262
REST 182, 259
REST-Services 259
RFC 37, 53, 154, 158
Rich Client 10
RMI 14, 123, 126
rmiregistry 131
RolesAllowedDynamicFeature 359
RPC 14, 123
RPC-Modell 14

S

safe 261
Same-Origin-Policy 368
san 184
Secure Sockets Layer 184
SecurityContext 354
Security Manager 142
SEI 215
Server 7, 8, 162
ServerEndpoint 194
Server-Push 182, 191
Server-Push-Technik 337
Server-Sent Events 337
ServerSocket 56
Service 247, 255
Service-Anbieter 214
Service Endpoint Interface 215
Service Implementation Bean 216
Service-Nutzer 214
Service-orientierte Architektur 214
Service-Verzeichnis 214
Session 92, 95, 102, 105, 109, 111, 115, 118, 194
SIB 216
Sicherheitsrisiko 6
Simple Mail Transfer Protocol 15, 53
Simple Network Management Protocol 38
Single-Page-Webanwendung 12
Skalierbarkeit 6
Skeleton 130
SMTP 15, 53
SNMP 38
SOA 214
SOAP 210, 211
SOAP Handler 240
Socket 20, 54
SocketAddress 20
SocketException 39
SocketTimeoutException 45, 55, 56
Sse 340
SSE 337
SseBroadcaster 340
SseEventSink 341
SseEventSource 343
SSL 184, 362
SSLContextConfigurator 363
SSLEngineConfigurator 363
State 260
State Transfer 260

Status-Code 161
StreamingDataHandler 237
Stub 130
subject alternative name 184
Subressource 289
Subscriber 108, 144
Suspended 333
synchron 9
System Port 18

T
TCP 15, 53
TCP/IP 14
Telnet 15, 53
TemoraryQueue 105
Template-Parameter 268
TextMessage 92, 96
TFTP 38
Thin Client 10
Thread-Pool 64
TimeoutHandler 333
Timeout-Steuerung 45, 55, 56
Time-to-Live 49
TLS 184, 362
Topic 108, 111
TopicSubscriber 111
Top Level Domain 17
Transaktionsmodus 117
Transformer API for XML 245
Transmission Control Protocol 15, 53
Transport Layer Security 184
Transportschicht 15
Trivial File Transfer Protocol 38
TTL 49

U
UDP 15, 37
UDP-Datagramm 37
UDP-Socket 38
Unicast 48
UnicastRemoteObject 126, 129
Uniform Resource Identifier 259
Uniform Resource Locator 153
Universally Unique Identifier 300
UnknownHostException 19
URI 259
UriBuilder 269
UriInfo 269
URL 153
URL-codiert 159
URLConnection 167
User Datagram Protocol 15, 37
User Port 18
UUID 300

V
Validierungsmodell 313, 317
Verarbeitung 4
Verbindungsschicht 15
Vermittlungsschicht 15
Verteiltes System 5
Verteilung 209
Verteilungstransparenz 13
Verzeichnisdienst 210
vierstufige Architektur 11

W
WADL 273
Web Application Description Language 273
WebApplicationException 289
Web Service 210
Web Service Description Language 212
WebServiceException 233
WebSocket 191, 198
WebSocket.Builder 198
WebSocket.Listener 198

WebTarget 280
well-known service 18
Wildcard-Adresse 21
Windows PowerShell 220, 224
Wirtschaftlichkeit 6
World Wide Web 15
WSDL 210, 212
WWW 15

X
XML 210

Z
Zugriffstransparenz 13
zustandslos 154
Zustandslosigkeit 209
zweistufige Architektur 9

Springer Vieweg

springer-vieweg.de

Jetzt im Springer-Shop bestellen:
springer.com/978-3-658-21906-2

Springer Vieweg

springer-vieweg.de

Jetzt im Springer-Shop bestellen:
springer.com/978-3-658-16378-5