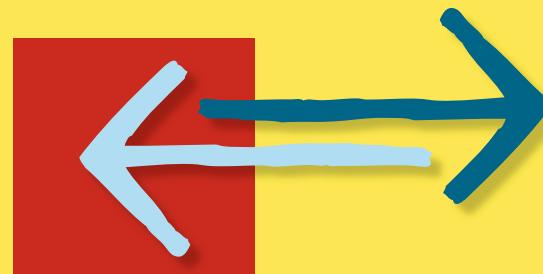


peter leo GORSKI
luigi LO IACONO
hoai viet NGUYEN



WebSockets

Moderne
**HTML5-Echtzeit-
anwendungen
entwickeln**

HANSER

Gorski/Lo Iacono/Nguyen
WebSockets

Bleiben Sie auf dem Laufenden!



Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter



www.hanser-fachbuch.de/newsletter



Hanser Update ist der IT-Blog des Hanser Verlags mit Beiträgen und Praxistipps von unseren Autoren rund um die Themen Online Marketing, Webentwicklung, Programmierung, Softwareentwicklung sowie IT- und Projektmanagement. Lesen Sie mit und abonnieren Sie unsere News unter



www.hanser-fachbuch.de/update

Lizenziert für luxiliph@gmail.com.

© 2015 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Peter Leo Gorski

Luigi Lo Iacono

Hoai Viet Nguyen

WebSockets

Moderne HTML5-
Echtzeitanwendungen
entwickeln

HANSER

Die Autoren:

Peter Leo Gorski, Köln

Prof. Dr.-Ing. Luigi Lo Iacono, Bonn

Hoai Viet Nguyen, Köln

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Bibliografische Information der Deutschen Nationalbibliothek:

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

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2015 Carl Hanser Verlag München, www.hanser-fachbuch.de

Lektorat: Sieglinde Schärl

Herstellung: Irene Weilhart

Copy editing: Sandra Gottmann, Münster

Layout: Peter Leo Gorski mit LaTeX

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Stephan Rönigk

Druck und Bindung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

Print-ISBN: 978-3-446-44371-6

E-Book-ISBN: 978-3-446-44438-6

**»Der Weltuntergang steht bevor,
aber nicht so, wie Sie denken.
Dieser Krieg jagt nicht alles in die Luft,
sondern schaltet alles ab.«**



**Tom DeMarco
Als auf der Welt das Licht ausging**

ca. 560 Seiten. Hardcover

ca. € 19,99 [D] / € 20,60 [A] / sFr 28,90

ISBN 978-3-446-43960-3

Erscheint im November 2014

Hier klicken zur
Leseprobe

Sie möchten mehr über Tom DeMarco und seine Bücher erfahren.
Einfach reinklicken unter www.hanser-fachbuch.de/special/demarco

Inhalt

Vorwort	IX
Danksagung	XI
1 Einleitung	1
1.1 Wie ist das Buch strukturiert?	3
1.2 Wer sollte das Buch lesen?	3
1.3 Wie sollte mit dem Buch gearbeitet werden?	4
2 HTTP: Hypertext Transfer Protocol	5
2.1 Grundlegendes	5
2.1.1 ISO/OSI-Referenzmodell für Kommunikationssysteme	6
2.1.2 Vereinfachtes OSI-Modell	8
2.1.3 Dienst	9
2.1.4 Protokoll	10
2.1.5 Kommunikationsfluss	11
2.2 HTTP en détail	13
2.2.1 Kommunikationsablauf	13
2.2.2 Textbasiert	14
2.2.3 Aufbau von Request-Nachrichten und Protokollelemente	16
2.2.4 Aufbau von Response-Nachrichten und Statuscodes	20
2.2.5 Zustandslos	24
3 Höhere Interaktivität und Echtzeitfähigkeit	25
3.1 XMLHttpRequest (XHR)	25
3.2 Polling	27
3.3 Long-Polling	28
3.4 Comet	29
3.5 Server-Sent Events	31
3.6 Bewertung der Verfahren	33

4 Die Leitung: Das IETF WebSocket-Protokoll	35
4.1 Was bisher geschah	35
4.2 Opening-Handshake – WebSocket-Verbindung aufbauen	36
4.3 WebSocket-Frames	39
4.4 Fragmentierung	42
4.5 Maskierung	43
4.6 Datenframes	45
4.6.1 Frames mit Textdaten	45
4.6.2 Frames mit Binärdaten	47
4.7 Control-Frames	48
4.7.1 Ping-Frame	48
4.7.2 Pong-Frame	49
4.7.3 Close-Frame	49
4.8 Closing-Handshake – WebSocket-Verbindung schließen	53
4.9 Tools zur Protokollanalyse	54
4.9.1 Wireshark	54
4.9.2 Fiddler	60
4.9.3 Chrome Developer Tools	68
4.9.4 Chrome Network Internals	70
5 Der Client: Die W3C WebSocket-API	75
5.1 Was bisher geschah	75
5.2 Browserunterstützung	76
5.3 Namensschema	79
5.4 Objekterzeugung und Verbindungsaufbau	80
5.5 Zustände	81
5.6 Event-Handler	83
5.7 Erstes vollständiges Programmchen	86
5.8 Attribute	89
5.9 Datenübertragung	92
5.9.1 Übertragung textbasierter Daten	93
5.9.2 Übertragung binärer Daten	94
5.10 Verbindungsabbau	96
5.10.1 Verbindung beenden	96
5.10.2 Close-Event	96
5.11 Ausblick: HTTP 2.0/SPDY	97

6 Der Server: Sprachliche Vielfalt	101
6.1 Übersicht verfügbarer WebSocket-Implementierungen	101
6.2 Node.js	102
6.2.1 Installation von Node.js	104
6.2.2 WebSocket.io	106
6.2.3 Socket.io	109
6.2.4 WebSocket-Node	113
6.3 Vert.x.....	119
6.4 Play Framework	122
6.4.1 Installation	122
6.4.2 Anlegen eines neuen Play-Projekts	123
6.4.3 Anatomie einer Play-Anwendung.....	123
6.4.4 Die Play-Konsole	124
6.4.5 Controller in Play.....	124
6.4.6 Views in Play	125
6.4.7 Routes in Play.....	126
6.4.8 Vom Controller zur View	127
6.4.9 Erstellen eines Echo-Servers in Play	129
6.5 JSR 356	132
6.5.1 JSR 356-basierender WebSocket-Server.....	136
6.5.2 JSR 356-basierender WebSocket-Client	140
7 WebSockets in der Praxis	147
7.1 Performance und Skalierbarkeit	147
7.1.1 Test-Echo-Server	148
7.1.2 Testclient für die Auslastung.....	151
7.1.3 Testclient für die Zeitmessung	152
7.1.4 Einrichtung der Testumgebung	153
7.1.5 Ergebnisse.....	157
7.2 Sicherheit	158
7.2.1 Same Origin Policy	158
7.2.2 Vertrauliche Kommunikation	159
7.2.3 Authentifizierung	162
7.2.4 Firewalls und Proxys	183
7.2.5 Mögliche Gefährdungen	186
7.2.6 Bewertung der Sicherheitslage	195
7.3 Mit Haken und Ösen	196
7.3.1 Pings und Pongs.....	196
7.3.2 Das Cache-Problem im Internet Explorer 10	197

8 Beispielanwendungen	201
8.1 Fernbedienung von Webanwendungen mit einer Smartphone-/Tablet-Fernbedienung	202
8.2 Chat-System	211
8.3 Heatmap für Usability-Tests	216
8.4 Überwachungskamera per Webcam	221
Schlusswort	231
A XHR-Objekt	233
B Chrome Developer Tools auf Android	235
C Umgebungsvariablen definieren.....	241
C.1 Mac OS/Linux mit Bash.....	241
C.2 Windows	242
D Express.js	247
D.1 Anatomie einer Express.js-Anwendung	248
D.2 Die Anwendungslogik in der Datei app.js	250
D.3 Die Jade-Template-Engine	252
D.4 Express.js mit WebSocket-Servern verbinden	254
Literatur	257
Stichwortverzeichnis	263

Vorwort

Netzwerke umgeben uns quasi überall. Viele Dinge des täglichen Geschäfts- und Privatlebens sind ohne Kommunikation über ein digitales Datennetz nicht mehr vorstellbar. Zur Lingua franca der Protokolle der Anwendungsschicht hat sich in den letzten Jahren das HTTP gemausert. Angeschoben durch den immensen Erfolg des Web bedienen sich heute eine Vielzahl anderer Anwendungen dem HTTP. Darunter z.B. die Lokalisierung und Ansteuerung von Geräten im Hausnetz via UPnP, die Speicherung von Daten in der Cloud alla Dropbox und vermehrt auch das TV. Die Gründe hierfür sind vielfältig. Der simple Aufbau von HTTP und die zahlreich verfügbaren Implementierungen spielen dabei sicherlich eine Hauptrolle. In den Nebenrollen finden sich Darsteller wie die durchgängige Firewall-Freundlichkeit und neuerlich auch die ubiquitäre Verbreitung von HTTP in Geräten jeden Couleurs. Ist man mit seinen Anwendungen und Diensten an Reichweite interessiert, führt derzeit kein Weg an HTTP vorbei.

Neben all dieser Euphorie ist es ratsam, alle Eigenschaften des neuen Gefährten zu kennen, da sich daraus auch seine Grenzen ableiten lassen. An der Zustandslosigkeit von HTTP lässt sich dies schön verdeutlichen. Erlaubt es auf der einen Seite ein vereinfachtes Caching und globale Skalierbarkeit durch Content Distribution Networks (CDN), erschwert es die Implementierung von Warenkörben in E-Commerce Anwendungen. Letzteres hat man im Laufe der Zeit gut in den Griff bekommen. Für andere Nachteile haben sich dagegen noch keine Patentrezeptheit etabliert. Hierzu zählt z.B. die unidirektionale ausgelegte Kommunikation. Gemäß der Protokollspezifikation meldet sich immer der Client mit einer Anfrage an den Server, der auf diese mit einer entsprechenden Antwort reagiert. Dass sich der Server selbstinitiativ bei einem Client meldet, ist nicht vorgesehen. Dies liegt auch darin begründet, dass die Verbindung zwischen Client und Server nicht dauerhaft, sondern nur für die Zeit des Austausches von Anfragen und ihrer Antworten besteht. In modernen Anwendungen ist das Pushen von Nachrichten oder Daten vom Server zum Client eine häufige Anforderung, die es technisch umzusetzen gilt. Genau an dieser Stelle setzt die WebSocket-Technologie an.

Dem Wildwuchs an Ansätzen, die zum Aufbrechen der Kommunikationseinbahnstraße von HTTP vorgeschlagen wurden und uneinheitlich zur Verwendung gekommen sind, wird mit WebSockets eine standardisierte Lösung entgegengestellt. Der Hunger nach einem derartigen Standard war groß. Dies lässt sich daran ablesen, dass die Unterstützung von WebSockets durch alle einschlägigen Industriegrößen bekannt gegeben wurde, als sich die Standardisierung noch in den Kinderschuhen befand. Bereits heute lässt sich kein Webbrowser vermissen, der nicht WebSockets mit an Bord hat. Obwohl die Standardisierung

zum aktuellen Zeitpunkt noch nicht abgeschlossen ist, kann davon ausgegangen werden, dass es sich hierbei in nicht allzu langer Zeit um einen etablierten Standard handeln wird.

Das vorliegende Buch stellt diesen neuen Grundpfeiler für moderne Anwendungen auf Grundlage von HTTP vor. Dabei geht es auf alle Aspekte der WebSocket-Technologie in angemessener Tiefe ein. Meine vorweggenommene Motivation wird zu Beginn des Buches, angereichert mit dem notwendigen Background, aufgegriffen und somit auf das eigentliche Thema hingeführt. Dann stehen die WebSockets im Rampenlicht. Mit dem unter den Fittichen der IETF stehendem WebSocket-Protokoll geht es los. Hier wird Bit genau beleuchtet, wie sich die verschiedenen Frames zwischen Client und Server bewegen. Das zeigt unter anderem, wie viel effektiver die Übertragung mittels WebSockets im Vergleich zu HTTP ist, wenn der Overhead der HTTP-Header wegfällt. Im Anschluss lernt man die von der W3C verantwortete JavaScript API kennen, mit der sich die Clientseite einer verteilten WebSocket-Anwendung in einem Browser implementieren lässt. Das ist allerdings noch nicht alles, was für ein ganzheitliches Projekt benötigt wird. Die Serverseite fehlt und hier sieht das Bild nicht ganz so einheitlich aus. Fehlende Standards machen es erforderlich, sich die vom ausgewählten Framework entsprechend bereitgestellten APIs anzueignen. Diesem Umstand Rechnung tragend, werden in dem Buch verschiedene serverseitige Frameworks behandelt, die für Java und JavaScript bereitstehen. Abgerundet wird das Ganze durch zahlreiche Beispiele, die auch größere Zusammenhänge nachvollziehbar und verständlich machen.

Mir hat die Lektüre das notwendige Know-how und das Verständnis in die neuen Möglichkeiten vermittelt, mit dem ich nun spannenden neuen Projekten unter Verwendung von WebSockets entgegenblicken. Ich wünsche Ihnen ebenso viel Erkenntnisgewinn beim Lesen, Ihr

Alexander Leschinsky

Geschäftsführer der G&L Systemhaus GmbH

Köln, im November 2014

Danksagung

Die Seiten eines Buches füllen sich maßgeblich durch die Federn der Autoren. Einfluss auf die Formulierungen und Aufbereitungen der produzierten Inhalte haben allerdings eine Vielzahl von weiteren Personen. Auch an diesem Werk haben viele gute Geister konstruktiv mitgewirkt, denen wir hiermit ein herzliches Dankeschön aussprechen möchten (Nennungen in alphabetischer Reichenfolge): Aline Jaritz, Carsten Mörke, Daniel Torkian, Stephan Mattescheck und Sven Wagner

Ein Dankeschön geht außerdem an die Fachhochschule Köln, an den Carl Hanser Verlag, an unsere Lektorin Sieglinde Schärl, für das reibungslose Management, an Irene Weilhart, für die Hilfe beim Buchsatz und selbstverständlich an Sandra Gottmann, die eine Fülle von Fehlern entdeckt hat, die uns beim Schreiben durch die Finger gerutscht sind.

Möchte man eine Webanwendungen oder gar einen Webservice entwickeln, kommt man unweigerlich mit einer Vielzahl von Web-Technologien, Frameworks und anderen Artefakten in Kontakt. Um dem Leser die WebSocket-Technologie näherzubringen, nehmen auch wir daher Bezug auf viele Zeilen Code und Software, die wir nicht immer selbst entwickelt haben. Deshalb möchten wir den Organisationen, Unternehmen und Entwicklern, einen besonderen Dank aussprechen, insbesondere denen, die Ihre Entwicklungen jedem zur freien Verfügung stellen, um etwas daraus lernen und um gemeinsam Technologien voranbringen zu können (erneut in alphabetischer Reihenfolge): Agendaless Consulting and Contributors (supervisor), Alexis Deveria (caniuse.com), Amazon.com Inc., Apple Inc., Automattic (Socket.io), Brian McKelvey (WebSocket-Node), Canonical Foundation, Eclipse Foundation, Eric Lawrence (Fiddler), Firebug Working Group, GNU/Linux, Google Inc., IETF, ISO, Joyent Inc. (Node.js), Kaazing Corporation (WebSocket.org), LearnBoost (WebSocket.io), Microsoft Corporation, Mozilla Corporation, Opera Software, Oracle Corporation, Patrick Wied (heatmap.js), Play Framework, strongloop (Express.js), Tim Fox & VMWare & Red Hat (vert.x), Twitter Inc. (Bootstrap), W3C, Wireshark-Community und alle, die noch im Buch erwähnt werden und die wir versehentlich an dieser Stelle vergessen haben.

Ein Teil der Grafiken, die wir für dieses Buch angefertigt haben, sind mithilfe von lizenzfreien Cliparts der Internetseite www.clker.com/ angefertigt worden. Auch dafür sind wir sehr dankbar.

Die Fertigstellung eines Buchs benötigt vor allem viel Zeit, Geduld, Fleiß und Ausdauer. Auch das wäre für uns ohne die Unterstützung der Menschen, die uns am nächsten stehen, nicht möglich gewesen. Vielen Dank an Anh Trung, Antonio, Ba Tho, Barbara, Em Yen, Fabio, Giuliana, Inge, Marco, Me Chau, Peter und Tati.

An letzter Stelle möchten wir natürlich auch Ihnen für den Kauf dieses Buches und Ihr Interesse an den WebSockets danken!

1

Einleitung

Einer der technischen Hauptakteure im Web ist HTTP, das *Hypertext Transfer Protocol* [FGM⁺99], und das ist so, weil HTTP den genauen Kommunikationsablauf zwischen Webbrowser und Webserver festlegt. Genau wie ein Protokoll für eine königliche Gala zu Hofe alle Regeln in Bezug auf Kleidung, Etikette und Umgangsformen zusammenfasst, so legt ein Kommunikationsprotokoll alle Regeln zum Nachrichtenaustausch fest. Anders als bei Hofe sind das bei einem Kommunikationsprotokoll aber vielmehr Regeln in Bezug auf den Aufbau, das Format und die Codierung der ausgetauschten Nachrichten. Außerdem muss in einem Protokoll festgelegt sein, wer die Kommunikation beginnt und wie diese dann Nachricht für Nachricht bis zum Kommunikationsende abläuft. HTTP verwendet hier einen einfachen Request-Response-Nachrichtenaustausch. Dieser wird immer vom Webclient durch eine Anfrage (engl. *Request*) initiiert und entsprechend vom Webserver mit einer Antwort (engl. *Response*) beantwortet. Sobald Sie also z. B. eine Webadresse – die URL – in die Adresszeile eines Webbrowsers eingeben oder einen Bookmark bzw. Link anklicken, setzen Sie damit eine Anfrage an einen Server ab, der diese dann bearbeitet und die Antwort daraufhin zurücksendet (siehe Bild 1.1).

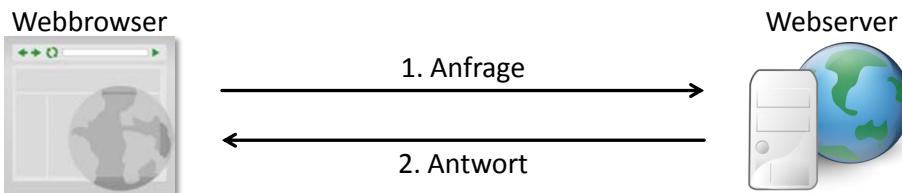


Bild 1.1 Grundlegender HTTP-Nachrichtenaustausch

In diesem einfachen Request-Response-Nachrichtenaustausch liegen viele Gründe für den großen Erfolg von HTTP und nicht zuletzt des Webs. Dazu zählen u. a. die guten Skalierungseigenschaften, denen wir die Größe des Webs verdanken. Wie Sie sich aber sicher jetzt schon denken, bringt das Nachrichtenpaar nicht nur Vorteile mit sich. Und tatsächlich ist die Protokollregel, dass der Request immer vom Client ausgehen muss, eine Zwangsjacke, die bestimmte Bewegungsfreiheiten in Form von Kommunikationsmustern nicht zu lässt. Der Teil einer Webanwendung, der auf dem Server ausgeführt wird, kann sich nämlich nicht von sich aus an einen Client wenden. Diese Eigenschaft des Protokolls schränkt folglich Anwendungsfälle ein, in denen die Serverseite Statusänderungen an die betroffenen Clients weiterreichen können muss. Typische Beispiele hierfür sind Chatanwendun-

gen, Finanzdatenströme, Webmail, Sportticker, das Monitoring von Haushalt und Industrie oder auch Internetauktionen. All diese Anwendungen teilen die Gemeinsamkeit, dass (häufig in relativ kurzen Zeitabschnitten) Daten auf dem Server hinzukommen bzw. sich ändern und dies unmittelbar an die angeschlossenen Clients bekannt gegeben werden muss. Liegt Ihnen eine derartige Anforderung auf dem Tisch, dann würden Sie sich eine standardisierte Technik wünschen, mit der Sie die Schranken der Serverseite öffnen und die entsprechenden Clients ansprechen können.

HTTP bietet für derartige Kommunikationsmuster keine native Unterstützung. Wenn Sie sich in den letzten Jahren aus dieser Zwangsjacke hätten befreien und die Kommunikationseinbahnstraße zur Umsetzung von Serverbenachrichtigungen hätten aufbrechen wollen, so wären Sie auf Grundlage des Verfügbaren auf eine (selbst) konstruierte Lösung angewiesen gewesen. Die Konstruktionen, die findige Webentwickler dabei gefunden haben, sind vielfältig. Prinzipiell können Sie den Standardfunktionsumfang durch Browser-Plug-ins und serverseitige Erweiterungen aufbohren und damit proprietäre Lösungen zur Serverbenachrichtigung implementieren. Der erste Ansatz überhaupt kam tatsächlich auch aus dieser Technologiegattung und bediente sich Java Applets und der LiveConnect-Schnittstelle zur Anbindung an JavaScript-Funktionen im Browser. Seit HTML5 besteht allerdings ein starker Trend weg von Browsererweiterungen à la Plug-in oder Add-on. Schützenhilfe kam zudem aus dem Umfeld der Smartphones und Tablets, das sich gegen eine Unterstützung von Flash und Silverlight entschieden hat, was schließlich selbst Adobe und Microsoft dazu bewegt hat, die Entwicklungen ihrer Technologien für mobile Plattformen einzustellen. Daher wollen wir auch nicht weiter auf das Auslaufmodell der Plug-in-basierten Ansätze eingehen. Weitere Konstruktionen basieren auf Ansätzen, die den Webbrowser in regelmäßigen Zeitabständen beim Webserver nach Veränderungen fragen lassen. Diese aktuell weit verbreiteten Verfahren haben Nachteile, was das Kommunikationsaufkommen anbelangt. Dennoch haben sie unter Umständen ihre Berechtigung. Sie lernen diese Ansätze daher kennen und insbesondere diese zu bewerten, um die richtige Technologie für Ihr Entwicklungsvorhaben auswählen zu können. Nichtsdestotrotz handelt es sich aber weitestgehend um Notlösungen, die durch die in der HTML5-Standardisierung befindlichen alternativen Lösungswege abgelöst werden. Dazu gehören zum einen *Server-Sent Events* (SSE) [Hic12a] und zum anderen *WebSockets* (WS) [FM11a]. Die verschiedenen Bestandteile des Standards, die HTML5 einführt, sind thematisch gruppiert und mit einem Logo versehen worden [W3C14]. SSE und WS sind in der sogenannten Connectivity-Gruppe. Das Logo sehen Sie in einer leicht abgewandelten Form in Bild 1.2.



Bild 1.2 Das Logo für die Connectivity-Standards Server-Sent Events und WebSockets, platziert auf dem HTML5-Schild [W3C14]

WebSockets stehen im Fokus dieses Buches, da Sie damit neben den Serverbenachrichtigungen eine Vielzahl weiterer Anwendungsfälle realisieren können, zu denen z. B. Spiele, jegliche Art von Konversationen (Text, Sprache, Sprache mit zusätzlichem Video), Kollaborationen sowie das Benutzermonitoring im Rahmen von Usabilitystudien zählen.

■ 1.1 Wie ist das Buch strukturiert?

WebSockets und die damit einhergehenden neuen Entwicklungsmöglichkeiten stehen somit im Zentrum des vorliegenden Buches und an diese wollen wir uns mit Ihnen Schritt für Schritt ranpirschen. In [Kapitel 2](#) führen wir uns dazu zunächst die notwendigen Grundlagen in Bezug auf HTTP zu Gemüte. Kennen Sie diese schon, können Sie getrost die Abkürzung zu [Kapitel 3](#) nehmen, in dem wir uns mit den Mechanismen befassen, mit denen eine höhere Interaktivität und Echtzeitfähigkeit bei Webanwendungen erreicht werden kann. In [Kapitel 4](#) widmen wir uns dann dem WebSocket-Protokoll und in [Kapitel 5](#) geht es anschließend ans Eingemachte. Hier lernen wir die WebSocket API kennen und programmieren mit JavaScript erste Beispiele für WebSocket-Clientanwendungen in Webbrowsersn. In [Kapitel 6](#) wenden wir uns den WebSocket Implementierungen auf der Serverseite sowie gebräuchlichen Frameworks zu. Weitere wichtige Aspekte folgen in [Kapitel 7](#), das das Testen von verteilten WebSocket-basierten Applikationen beschreibt und auf Eigenschaften der Performance eingeht. Was niemals vergessen werden darf, sind Sicherheitsaspekte, insbesondere wenn die Anwendung aus verteilten Komponenten zusammengesetzt ist, die über offene Netze miteinander gekoppelt sind, wie das im Web quasi immer der Fall ist. Darauf wollen wir uns damit ebenfalls in [Kapitel 7](#) auseinandersetzen und die Besonderheiten von WebSockets in diesem Kontext herausstellen. In [Kapitel 8](#) werden wir dann das Gelernte einsetzen und verschiedene „größere“ und vollständige Anwendungen implementieren. Diese haben wir dabei so gewählt, dass damit ein möglichst großes Spektrum an Möglichkeiten aufgezeigt wird. Wir spannen den Bogen von einer generischen Fernsteuerung für Webanwendungen, über ein klassisches Chatsystem über eine Heatmap für Usability-Tests bis hin zu einer Überwachungskamera per Webcam.

■ 1.2 Wer sollte das Buch lesen?

In erster Linie richtet sich das vorliegende Buch an den Entwickler in Ihnen. Möchten Sie mehr über WebSockets wissen und verstehen, was Sie mit diesen so alles in Ihren Webprojekten anstellen können, dann sollten Sie hier fündig werden. Da der Koffer an Werkzeugen, Technologien und Sprachen kaum unterschiedlicher gefüllt ist als bei Webentwicklern, stellen wir auf keine spezifische Umgebung ab, sondern versuchen dieser farbenfrohen Vielfalt gerecht zu werden. So finden sich Beispiele auf Basis von Node.js, vertx und JSR 356 im Buch wieder. Als Programmiersprachen verwenden wir Java und JavaScript. Neben der gängigen Vorstellung und Erläuterung der Buchinhalte sind die zahlreichen Beispielanwendungen in [Kapitel 8](#) eine Stärke des vorliegenden Buches, mit denen das Erlernte geübt und nachvollzogen werden kann. Webentwickler sollten somit in die Lage versetzt werden,

einen umfassenden Einstieg in die Thematik zu bekommen und anhand der exemplarischen Anwendungen einen tief gehenden Einblick in den Einsatz von WebSockets zu erhalten.

Neben Webentwicklern eignet sich das Buch auch für Lehrveranstaltungen an Hochschulen und damit für Studierende verschiedener Fachrichtungen, die mit WebSockets innovative Anwendungen im Web entwickeln wollen. Zudem profitieren Informations- und Softwarearchitekten sowie Konzepte und (technische) Projektleiter vom vorliegenden Buch, da ein Grundverständnis der Kommunikationsmuster, die mit WebSockets realisiert werden können, für Architekturen und Konzepte von Bedeutung sein können. Auch hierfür haben wir versucht, mit den in [Kapitel 8](#) beschriebenen Beispielanwendungen den Horizont aufzuzeichnen.

■ 1.3 Wie sollte mit dem Buch gearbeitet werden?

Das Buch führt sukzessive in die Thematik ein und setzt nur wenige Vorkenntnisse voraus. Wenn Sie über kaum bis wenige Erfahrungen in der Webentwicklung verfügen, raten wir Ihnen, sich an diesem Aufbau zu orientieren und sich Kapitel für Kapitel einzuarbeiten. Nach den Grundlagen werden in den einführenden Kapiteln erste kleinere Beispielanwendungen gezeigt und erläutert. Wir empfehlen, diese nachzuprogrammieren und die Quelltexte im Detail nachzuvollziehen. Ist das Fundament gelegt, geht es mit weiterführenden Aspekten weiter, die selektiv durchgearbeitet werden können. Das Durcharbeiten der Beispielanwendungen ist wiederum sehr empfehlenswert. Es trainiert auf der einen Seite die im Buch vermittelten Inhalte. Bei Bedarf können Unklarheiten nochmals dediziert nachgelesen werden. Auf der anderen Seite zeigen sie die Potenziale von WebSockets auf. Leser mit fundiertem Vorwissen können sicherlich die Grundlagenkapitel überspringen und sich direkt mit den originären Inhalten und den Beispielanwendungen befassen.



<http://websocket101.org/>

Wir freuen uns von Ihnen zu hören! Ob Lob oder Kritik, ob Frage oder Anregung, ob Hinweis oder Verbesserung, wir glauben, dass das Buch von all dem profitieren kann, wenn es konstruktiv eingebbracht wird. Dazu wollen wir unser Nötigstes tun und freuen uns auf Ihr Mitwirken. Aktuelles wird dabei zeitnah auf der buchbegleitenden Webseite bereitgestellt. Schauen Sie doch häufiger mal auf

<http://websocket101.org/>

vorbei. Dort finden Sie auch unsere Kontaktinformationen.

An dieser Stelle bleibt uns nur noch, Ihnen viel Spaß beim Lesen und Nachprogrammieren sowie viel Erfolg bei den eigenen Entwicklungen zu wünschen.

2

HTTP: Hypertext Transfer Protocol

Wenn Sie gefragt werden, was das Web ist, so können Sie darauf viele verschiedene Antworten finden. Aus technischer Sicht lässt sich die Antwort allerdings auf ein Kommunikationsprotokoll zurückführen. Es wird Hypertext Transfer Protocol (HTTP) [FGM⁺99] genannt.



Fragen, die dieses Kapitel beantwortet:

- Welche wichtigen Kommunikationsgrundlagen werden zum besseren Verständnis von HTTP benötigt?
- Was sind die wesentlichen Details der Kommunikation im Web und insbesondere von HTTP?
- Welche Einschränkungen hat HTTP in Bezug auf Realzeit-Kommunikation?
- Was für Ad-hoc-Lösungen sind aus diesen Einschränkungen hervorgegangen?

■ 2.1 Grundlegendes

Wenn Ihnen Begriffe wie Protokoll, Schicht und Dienst im Kommunikationskontext noch nicht geläufig sind, dann sollten Sie diesem Kapitel aufmerksam folgen. Hier möchten wir Ihnen die grundlegenden Prinzipien und Konzepte moderner Kommunikationssysteme erläutern, da diese zum Verständnis von HTTP und später von WebSockets notwendig sind. Dazu möchten wir Ihnen zunächst das ISO/OSI-Referenzmodell näherbringen, welches das formale Modell aller gegenwärtigen Kommunikationssysteme darstellt. In den darauf folgenden Unterkapiteln zeigen wir Ihnen dann, wie das ISO/OSI-Modell im Internet angewendet wird und wo welche Protokolle was für Dienste realisieren, um schließlich den Austausch von Nachrichten im Web zu ermöglichen. Wenn wir das geschafft haben, steigen wir Schritt für Schritt tiefer in die Details von HTTP ein und leiten den Übergang in die Welt der WebSockets ein, indem wir Möglichkeiten und Einschränkungen der realzeit-orientierten Kommunikation im Web diskutieren.

2.1.1 ISO/OSI-Referenzmodell für Kommunikationssysteme

Häufig sind „alte“ Standards in der noch jungen Informations- und Kommunikationstechnologie ein Merkmal für deren Relevanz und Bedeutung. Ein herausragendes Beispiel hierfür ist das von der Standardisierungsorganisation ISO (International Organization for Standardization) bereits 1984 als internationaler Standard verabschiedete OSI-Referenzmodell (Open Systems Interconnection) [ISO94].

Ein Referenzmodell beschreibt ganz allgemein die logische Gliederung von Systemen. Dabei müssen Sie durch Abstraktion die Komponenten eines Systems identifizieren und deren Abhängigkeiten zueinander bestimmen. Und mehr noch, ein Referenzmodell versucht dabei die Gemeinsamkeiten und Unterschiede der konkreten Instanziierungen des Modells zu berücksichtigen. Wenn Sie sich mal überlegen, was Sie für Gemeinsamkeiten und Unterschiede in Kommunikationsnetzen zur Telefonie, zu dem Rundfunk oder dem Internet finden und wie Sie diese abstrakten Komponenten in Beziehung bringen können, sind Sie vermutlich schon beim OSI-Modell.

Das OSI-Modell definiert zur Abstraktion von Kommunikationssystemen sieben Schichten, die alle bestimmte festgelegte Aufgaben erledigen (siehe Bild 2.1).



Bild 2.1 ISO/OSI-Referenzmodell [ISO94]

Starten wir von unten. In der ersten Schicht oder in **Schicht 1** finden Sie alle diejenigen Spezifikationen und Funktionen, die für die physikalische Übertragung der Bits benötigt werden. Daraus ergibt sich auch der Name der Schicht, der im Deutschen passend als Bitübertragungsschicht übersetzt ist. Konkrete Ausprägungen dieser Schicht müssen Definitionen und Festlegungen machen, die von Spezifikationen von Kabeln und Steckern bis hin zu Modulationsverfahren reichen. Diese Schicht ist in der Netzwerkkopplungshardware (z. B. der Netzwerkkarte) implementiert.

Die **Schicht 2** hat die maßgebliche Aufgabe, dafür Sorge zu tragen, dass die zu übertragenen Bits fehlerfrei beim Empfänger ankommen. Sie müssen davon ausgehen, dass bei der Übertragung der Bits in vielfältiger Form Verfälschungen auftreten können. Diese können

in Abhängigkeit des konkreten Übertragungsmediums und -verfahrens sehr unterschiedlich sein. Diese Schicht soll übertragungsbedingte Verfälschungen erkennen und ggf. beheben. Eine weitere wichtige Funktion, die Schicht 2 innehat, ist die Steuerung von (zeitgleichen) Zugriffen auf das gemeinsam genutzte physikalische Übertragungsmedium. Hierzu wird je nach Ausprägung eine Identifikation der Netzwerkkarte benötigt, die als MAC-Adresse bekannt ist. MAC steht hier für Medium Access Control. Als Softwareentwickler kommen Sie in der Regel, wie auch schon für Schicht 1, mit dieser Schicht nicht direkt in Berührung, da diese ebenfalls in der Netzwerkkartenhardware implementiert ist.

Die Vermittlungsschicht ist die **Schicht 3**. Sie können sich vorstellen, dass diese Schicht die Verbindung zum Kommunikationspartner herstellt. Dies kann erneut verschiedene Ausprägungen annehmen. Ist das umgebende Kommunikationssystem ein leitungsvermittelndes Netz, wie es z. B. bei der analogen Festnetztelefonie der Fall ist, dann wird die Verbindung durch Schalten von Leitungen hergestellt. Handelt es sich um ein paketvermittelndes Netz, wie es z. B. das Internet ist, dann wird die Verbindung durch Weiterleiten der Pakete an das Zielsystem hergestellt. In beiden Fällen muss der Weg vom Sender zum Empfänger gesucht werden, was als Routing bezeichnet wird. Um einen Weg finden zu können, muss in Schicht 3 ein netzwerkweites Adressierungsschema vorliegen, womit die Kommunikationsendpunkte eindeutig identifiziert werden können. In den Beispielen, die wir gerade erwähnt haben, wären das die Telefonnummern bzw. IP-Adressen.

Die als Transportschicht benannte **Schicht 4** hat die Aufgabe, die Kommunikation von Endpunkt zu Endpunkt zu kontrollieren und zu steuern. Zu den typischen Dingen, die in diesen Aufgabenbereich fallen, zählen u. a., dass der Sender den Empfänger mit den gesendeten Daten nicht überfrachtet und dass die anwendungsorientierten Schichten 5 bis 7 einen einheitlichen Zugang zum Netzwerk bekommen, unabhängig von den tatsächlichen Gegebenheiten. Bei Ausprägungen der Transportschicht, die einen verlässlichen Transport gewährleisten, werden zudem die Reihenfolge, Vollständigkeit und Fehlerfreiheit der Nachrichten bzw. Nachrichtenpakete sichergestellt. Um auch mehrere verschiedene Dienste auf einem Endsystem für das Netzwerk anbieten zu können, muss auf dieser Ebene ein zusätzlicher Adressierungsmechanismus vorliegen. In der Internet-Protokollfamilie sind das die nummerischen Ports. Für viele Standarddienste sind Default-Ports festgelegt worden [TLM⁺14]. Für das unverschlüsselte HTTP ist das der Port 80 und für die SSL-/TLS-gesicherte Variante entsprechend der Port 443.

Schicht 5 ist mit der Etablierung und Verwaltung von Sitzungen befasst. Sitzungen sollen durch entsprechende Maßnahmen gewährleisten, dass im Falle einer unterbrochenen Kommunikation und insbesondere eines nicht vollständig durchgeföhrten Datenaustausches dieser zu einem späteren Zeitpunkt an der unterbrochenen Stelle wieder aufgenommen werden kann.

Um zu gewährleisten, dass unterschiedliche Systeme mit den ausgetauschten Daten umgehen können, stellt **Schicht 6** Funktionen bereit, die dafür sorgen, dass die verwendeten Codierungen für die an der Kommunikation beteiligten Partner bekannt und verständlich sind. Sollte es nicht möglich sein, sich auf einen gemeinsamen Codierungsnenner zu verständigen, verfügt die sogenannte Darstellungsschicht zur Not über Konvertierungsroutinen, mit denen eine Anpassung an die spezifischen Gegebenheiten erfolgen kann.

Die Zuckerschicht des Schichtenkuchens stellt die Anwendungsschicht dar. Auf der 7. Ebene des OSI-Turms thronend, fasst **Schicht 7** alle Protokolle zusammen, die die unterschiedlichen Anwendungen benötigen. Hierzu zählen z. B. Protokolle zum Austausch von E-Mails

oder Dateien und für den Zugriff auf entfernte Systeme via Remote Login oder Virtual Terminal und das für uns so wichtige HTTP.

2.1.2 Vereinfachtes OSI-Modell

Referenzmodelle zeichnen sich dadurch aus, dass sie die Gemeinsamkeiten, aber auch die Unterschiede zwischen den verschiedenen konkreten Modellinstanzen berücksichtigen. Das bedeutet, dass die Anwendung des OSI-Referenzmodells für ein bestimmtes Kommunikationssystem nicht immer alle Aspekte des Modells beinhalten muss. Als wohl bekanntestes Beispiel können Sie sich hierfür das Internet vorstellen. Wie Sie aus Bild 2.2 ersehen können, stimmen das OSI-Modell und seine Inkarnation im Internet in den ersten vier transportorientierten Schichten überein. Erst in den anwendungsorientierten Schichten 5 bis 7 wird es im Internet schwierig, entsprechende Vertreter zu finden. Daher hat sich für das Internet eine entsprechend vereinfachte Instanz des OSI-Modells herausgebildet, die nur über die Anwendungsschicht verfügt und nicht weiter die Sitzungs- und Darstellungs- schicht unterscheidet.

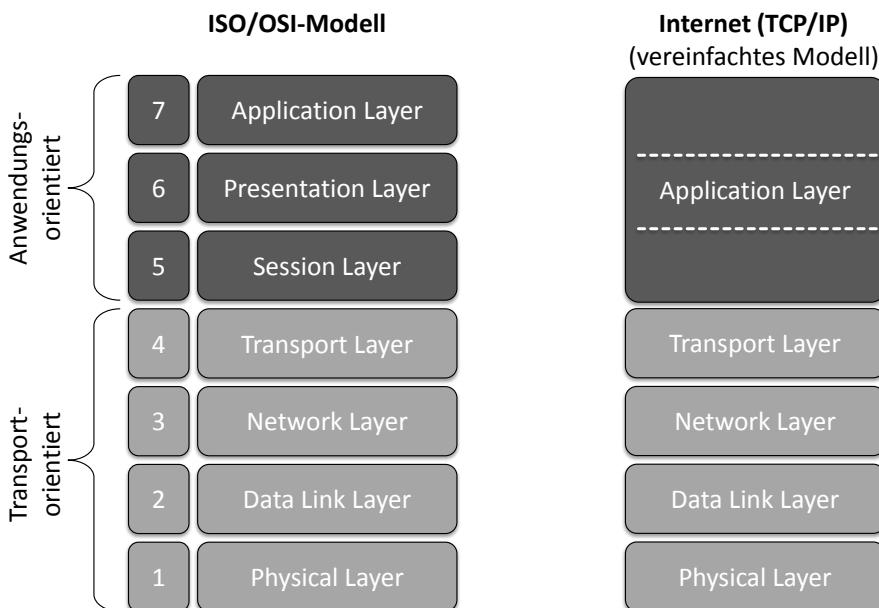


Bild 2.2 Anwendung des OSI-Modells auf das Internet

Das beschriebene Gebilde, ob als allgemeines Referenzmodell oder als Ausprägung für das Internet, allein reicht noch nicht, um zu verstehen, wie Ihnen ein solcher Schichtenhaufen bei der Entwicklung von komplexen Kommunikationssystemen behilflich sein kann.

2.1.3 Dienst

Durch die Gruppierung in sieben Schichten mit abstrakten, aber definierten Aufgaben und Funktionen ist zunächst eine Struktur in das Themenumfeld eingebbracht worden. Damit Sie dieses aber tatsächlich in seiner Komplexität beherrschen können, fehlt es noch an Ordnung. Diese wäre Ihnen nicht gegeben, wenn alle Schichten mit allen anderen Schichten interagieren könnten. Wenn dem so wäre, so stünden die Chancen nicht schlecht, in einem schier undurchdringbaren Wirrwarr an Abhängigkeiten zu enden, das Sie kaum noch durchblicken könnten. Wir benötigen also noch weitere Maßnahmen, um aus den sieben Schichten ein wirkungsvolles Modell zu bekommen, das uns tatsächlich bei der Komplexitätsbewältigung von Kommunikationssystemen unterstützt.

Dass die sieben Schichten im OSI-Referenzmodell häufig als Bauklötzenturm dargestellt werden, der ausgehend vom Fundament der Schicht 1 Stockwerk für Stockwerk in den Himmel ragt und schließlich im Dachgeschoss der Schicht 7 endet, ist kein Zufall. Dies hängt vielmehr mit der Tatsache zusammen, dass sich genau durch diese Anordnung die Abhängigkeiten der Schichten untereinander darstellen lassen. Die Dienste einer Schicht bieten diese genau der darüber liegenden Schicht an und sonst keiner anderen. Abstrakter ist dies nochmals in [Bild 2.3](#) dargestellt.

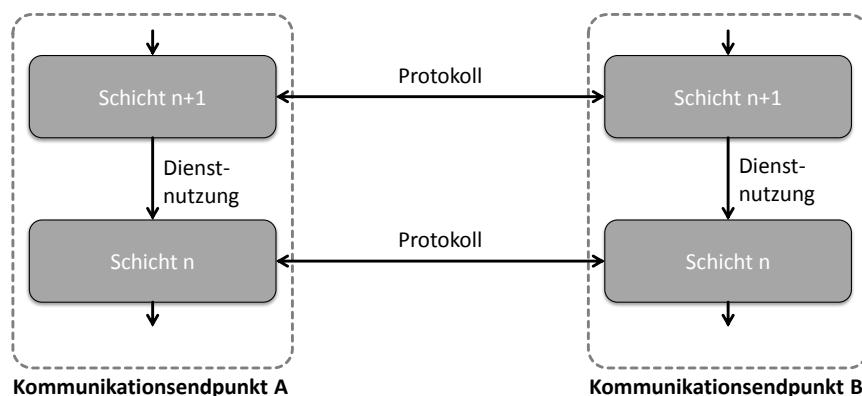


Bild 2.3 Dienste und Protokolle in Kommunikationssystemen

Auf diese Weise lassen sich die Abhängigkeiten unter den Schichten dramatisch reduzieren und die Kontrolle über den tatsächlichen Kommunikationsablauf bewahren. Sie können sich die Wirkung dieses Prinzips an einigen konkreten Konstellationen verdeutlichen. So ist es z. B. für die Schicht-4-Protokolle TCP und UDP völlig irrelevant, wie sich das Netzwerk unterhalb der eigenen Schicht zusammensetzt. Ob drahtgebunden oder kabellos, ob rasend schnell oder schleichend langsam, ob zuverlässig oder fehleranfällig, für TCP und UDP sind nur die Dienste der Netzwerkschicht ausschlaggebend und damit das, was ihnen IP bietet. Auf dieser Ebene muss nicht noch mit anderen Schichten als der direkt darunter liegenden interagiert werden, um die eigenen Funktionalitäten anbieten zu können. Denken Sie sich eigene Situationen aus und überprüfen Sie damit Ihr Verständnis.

2.1.4 Protokoll

Neben den Diensten und den Abhängigkeitsstrukturen zwischen den Schichten im OSI-Modell, kommt in Bild 2.3 zudem der Begriff des Protokolls vor. Nur was ist das bloß?

Wie ein Protokoll bei Hofe die Umgangs- und Benimmregeln festlegt, legt ein Kommunikationsprotokoll eine Menge von Regeln fest, die beschreiben, wie Nachrichten ausgetauscht werden, um eine bestimmte Aufgabe zu erledigen. Sie können noch weitere anschauliche Parallelen finden, mit denen Sie sich dem Protokollbegriff annähern können. Stellen Sie sich z. B. die Konvention vor, wie eine Person einer zweiten Person in einem Straßencafé höflich nach einem freien Sitzplatz fragt (siehe Bild 2.4).

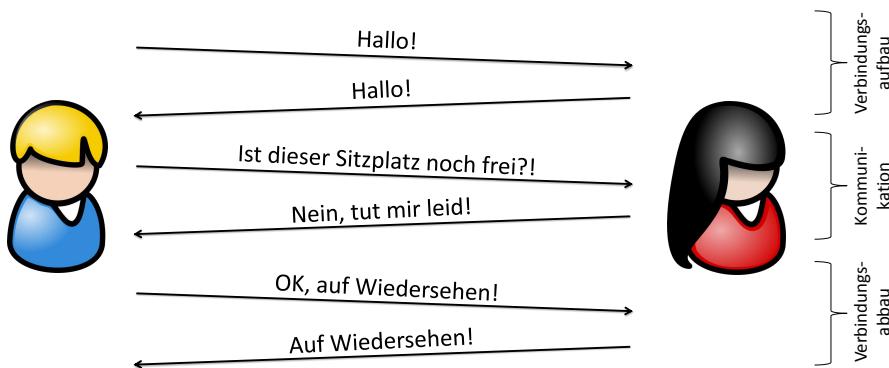


Bild 2.4 Veranschaulichung des Protokollbegriffs

In dem in Bild 2.4 dargestellten Protokoll ist festgelegt, dass derjenige die Verbindung aufbaut, der den freien Sitzplatz im Straßencafé sucht. In unserer Abbildung ist das der junge Mann auf der linken Seite. Dieser beginnt also mit dem Verbindungsauftakt, indem er die junge Frau mit einem freundlichen „Hallo!“ anspricht. Er konnte die junge Frau mit seiner Begrüßung erreichen und diese ist auch gesprächsbereit und antwortet in der Folge ebenfalls mit einem freundlichen „Hallo!“. Hiermit ist der Kommunikationskanal zwischen den beiden Gesprächspartnern aufgebaut und die eigentliche Kommunikation kann über die etablierte Verbindung vollzogen werden. Die eigentliche Frage, die dem jungen Mann am Herzen liegt, folgt sogleich und beinhaltet die Frage nach einem freien Sitzplatz. Auf seine Frage erwidert die junge Frau, dass der Platz nicht frei ist. Damit ist die gerade begonnene Konversation bereits wieder an ihrem Ende angekommen. Zu einer höflichen Anfrage gehört, dass sich der Fragende für die Auskunft bedankt und verabschiedet. Dies erfolgt auch mit dem letzten Nachrichtenpaar, dass damit auch die etablierte Kommunikationsverbindung beendet.

An dieser alltäglichen Lebenssituation können Sie wunderbar nachvollziehen, was alles Gegenstand eines Kommunikationsprotokolls ist. Die Abfolge der auszutauschenden Nachrichten wird festgelegt und damit auch, welcher der beiden Kommunikationspartner die Konversation initiiert und welche Nachrichtensequenz dann bis zur Beendigung der Kommunikation stattfindet. Zudem werden der Aufbau und Inhalt der Nachrichten festgelegt und wie diese codiert sind (in unserem Beispiel höflich). Und tatsächlich folgen Protokollstandards diesen Maßgaben aus dem realen Leben, um technische Kommuni-

kationssysteme umzusetzen. Wir haben für Sie zu einem späteren Zeitpunkt in diesem Kapitel – bei der detaillierten Beschreibung von HTTP – eine weitere Veranschaulichung eingebaut, bei der Sie ein Déjà-vu erleben sollten. Wenn Sie keines gehabt haben sollten, dann schreiben Sie uns!

Protokolle gibt es viele und sie sind immer einer Schicht des OSI-Modells zugeordnet. Bild 2.5 zeigt nochmals die Ausgestaltung des OSI-Modells für das Internet und fügt relevante Protokolle der jeweiligen Schicht an.

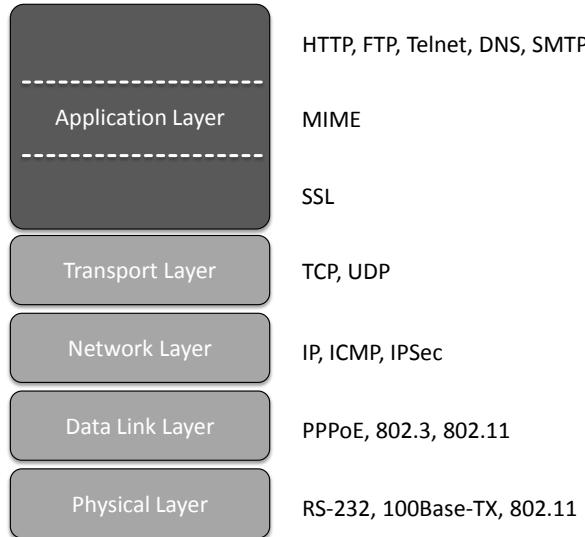


Bild 2.5 Ausgewählte Protokolle bei der Internetkommunikation

In den ersten beiden Schichten haben wir die gebräuchlichen Standards Ethernet (IEEE 802.3) und WLAN (IEEE 802.11) angegeben. Schicht 3 wird im Internet von IP dominiert. Auf der Transportschicht haben Sie die Wahl zwischen den zuverlässigen Transportdiensten von TCP oder dem unzuverlässigen Pendant UDP. In der Anwendungsschicht findet sich eine ganze Reihe von Protokollen, die z. B. für den entfernten Login (Telnet, SSH), für E-Mail (SMTP, IMAP, POP3), für den Datentransfer (FTP, SFTP) und für das Browsen im Web (HTTP) spezifiziert wurden.

2.1.5 Kommunikationsfluss

Warum und wie jetzt das auf den ersten Anblick recht umfangreiche und komplex wirkende OSI-Referenzmodell genau das Gegenteil erreichen will, nämlich die Komplexität moderner Kommunikationssysteme beherrschbar zu machen, können Sie sich an einem einfachen Beispiel vor Augen führen. Stellen Sie sich ein lokales Netzwerk vor, wie es in Bild 2.6 dargestellt ist und das einen Webserver über einen WLAN-Router mit einem Notebook vernetzt. Der Webserver ist dabei via Kupferkabel gemäß den Ethernet-Standards IEEE 802.3 [IEE12a] und das Notebook entsprechend kabellos gemäß den WLAN-Standards IEEE 802.11 [IEE12b] am Router angeschlossen.

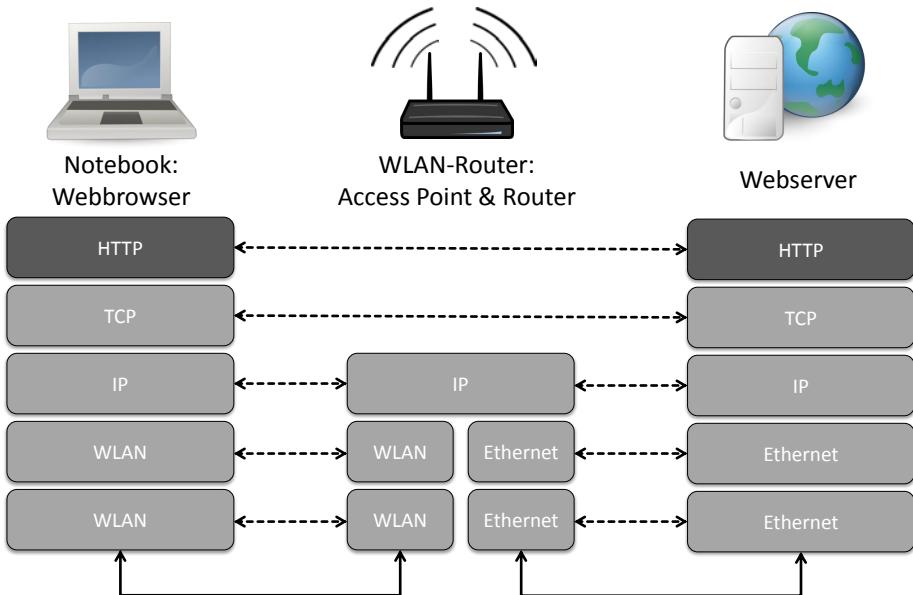


Bild 2.6 Beispielhafter Kommunikationsfluss zwischen Client und Server

Setzt ein Webbrower auf dem Notebook eine Anfrage an den Webserver im lokalen Netz ab, wird die HTTP-Anfrage logisch direkt an den Webserver gerichtet, der daraufhin mit einer HTTP-Antwort darauf reagiert. Dies ist die Protokollsicht. Tatsächlich nutzt der Webbrower den Dienst der darunter liegenden Transportschicht und bittet diesen, eine zuverlässige TCP-Verbindung zu dem in der URL angegebenen Host aufzubauen und den mit dem Port angegebenen Dienst aufzurufen. Die Transportschicht ermittelt die IP-Adresse zum vorliegenden Hostnamen und verpackt die Daten, die sie von der Anwendungsschicht bekommen hat, in TCP-Pakete, die neben den Daten aus der Anwendungsschicht zusätzliche Protokolldaten enthalten, die für die zuverlässige Zustellung der Daten benötigt werden. Die Netzwerkschicht ermittelt die Route, die die Pakete zum Ziel nehmen müssen. An dieser Stelle wird insbesondere ermittelt, an welche Netzwerkkarte die Daten übergeben werden sollen. Im Beispiel in Bild 2.6 ist das die im Notebook eingebaute WLAN-Karte. Ist die Netzwerkkarte ermittelt, werden an diese die IP-Pakete übergeben, in die die Netzwerkschicht die TCP-Pakete verpackt hat. In der Netzwerkkarte angekommen, werden in Abhängigkeit der vorliegenden Netzwerktechnologie der Zugriff auf das Medium (Kupferkabel, Lichtwelle, Funk) sowie die Umwandlung der Daten in übertragungsfähige Signale vorgenommen. Auf diesem Weg gelangen die Daten schließlich physikalisch auf den nächsten Vermittlungsknoten auf der Route zum adressierten Ziel. Im Beispiel ist das der lokale WLAN-Router. Hier werden bis einschließlich zur Schicht 3 die Schritte rückgängig gemacht. Die zurückgewonnenen IP-Pakete enthalten die IP-Adresse des Zielsystems und können anhand dieser weitergeleitet werden. Da der Webserver im LAN angesprochen wird, legt der WLAN-Router diese auf die lokale Ethernetkarte. Die IP-Pakete werden erneut für den physikalischen Transport über das Kupferkabel zur Ethernetkarte im Webserver an-

angepasst und in entsprechende Signale überführt. Im Webserver angekommen, werden die Pakete Schicht für Schicht nach oben gereicht. Die Transportschicht gewährleistet, dass die TCP-Pakete vollständig und fehlerfrei sowie in der richtigen Reihenfolge vorliegen. Daraus werden dann die von der Anwendungsschicht der Clientseite abgeschickten Daten (z. B. die HTTP-Anfrage) rekonstruiert. Die rekonstruierten Daten können dann von der Anwendung verarbeitet werden. Die HTTP-Antwort geht den gleichen Weg zurück.

■ 2.2 HTTP en détail

Im OSI-Referenzmodell finden Sie HTTP in der Anwendungsschicht. Einige der Regeln, die ein Kommunikationsprotokoll ausmachen, haben wir bereits im vorangegangenen Kapitel besprochen. Dazu zählt insbesondere die Reihenfolge der zwischen Client und Server ausgetauschten Nachrichten. Diese Regeln, die ein Protokoll ausmachen, schauen wir uns im Folgenden anhand von HTTP im Detail an.

2.2.1 Kommunikationsablauf

Die erste Frage, die Sie sich dabei vermutlich gerade stellen, ist, wer die Kommunikation zwischen den beiden beteiligten Akteuren beginnt. Tatsächlich ist in der HTTP-Spezifikation festgelegt, dass immer der Client die Kommunikation initiiert. Diese erfolgt durch das Absetzen einer Anfrage an den Server.

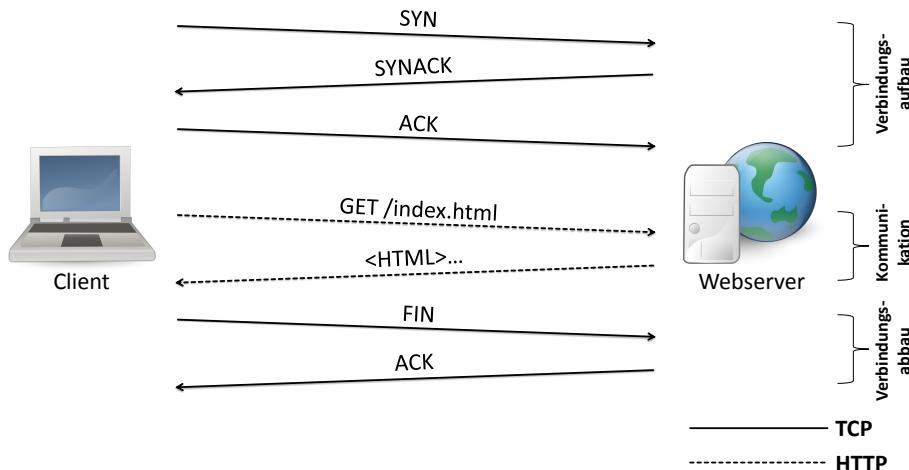


Bild 2.7 HTTP über TCP

Diesen Ablauf können Sie auch einfach mit dem Telnet-Clientprogramm nachvollziehen (in jedem guten Betriebssystem enthalten). Das Telnet-Clientprogramm etabliert nämlich

nichts anderes als eine TCP-Verbindung zu einem entfernten Rechner. Konnte diese hergestellt werden, befindet sich im Normalfall auf der Gegenseite ein Telnet-Server, der die Eingaben des Benutzers im Telnet-Client entgegennimmt, lokal ausführt und die Ausgabe an das Clientprogramm zurücksendet. Der Client erlaubt es, einen vom Telnet spezifizierten Standardport 22 abweichenden Port für den Verbindungsaufbau anzugeben. Setzen Sie diesen nun, wie in [Bild 2.8](#) demonstriert, auf den HTTP-Standardport, so wird eine TCP-Verbindung zum angegebenen Webserver aufgebaut (die ersten drei Nachrichten in [Bild 2.7](#)). Was die Abbildung nicht mehr zeigt, ist die Antwort des Webservers, der die angefragte HTML-Seite zurückliefert. Die Kommunikationsverbindung wird direkt nach Erhalt der HTTP-Antwort vom Webserver geschlossen.

```

user@ubuntu:~$ telnet google.de 80 ← 1. TCP-Verbindung öffnen
Trying 209.85.148.104...
Connected to google.de.
Escape character is '^]'
GET /index.html HTTP/1.0 ← 2. HTTP-Anfrage absetzen
-

```

Bild 2.8 Die Telnet-Probe

An diesem Experiment erkennen Sie auch schon eine wesentliche Eigenschaft von HTTP. Es ist textbasiert, sonst wären Sie nicht in der Lage gewesen, die Anfrage einfach so über Ihre Tastatur in die offene TCP-Verbindung der Telnet-Sitzung zu tippen. Andere Tools, mit denen Sie derartige Tests durchführen können, reichen von allgemeinen Netzwerktools wie z. B. Netcat¹ bis hin zu spezialisierten Werkzeugen speziell für HTTP wie z. B. cURL².

2.2.2 Textbasiert

Die Codierung der Protokollelemente erfolgt als Text. Sie können sich vorstellen, dass das den Nachteil hat, aus Sicht der zu übertragenden Datenmenge nicht optimal zu sein. In diesem Kontext sind binäre Codierungen sicherlich leichtgewichtiger. Im Vergleich zu binären Protokollen ist aber der große Vorteil textbasierter Protokolle, wie es HTTP ist, dass die Protokollelemente einfach zu erzeugen und zu lesen sind. Hierdurch erleichtert sich die Entwicklung ungemein. Wenn Sie schon mal einen Bug in einem binären Protokollformat aufspüren mussten, werden Sie uns in diesem Punkt ohne Weiteres zustimmen.

Um zu verdeutlichen, wie die Textbasiertheit die Entwicklung mit HTTP vereinfacht, wollen wir mit Ihnen einen einfachen HTTP-Client implementieren. Das Beispiel mit dem Telnet-Client aus [Abschnitt 2.2.1](#) wollen wir mit einer weitverbreiteten Programmierschnittstelle, den Sockets, in ein eigenes Programm integrieren. Sockets bieten eine abstrakte Schnittstelle, mit der Sie eine TCP-Verbindung zu einem Server aufbauen und verwalten können. Das folgende Beispiel zeigt, wie Sie mit Java einen Socket (also einen TCP-Kanal) zu einem Server öffnen können. Da Sockets praktisch in jeder Programmiersprache

¹ <http://de.wikipedia.org/wiki/Netcat>

² <http://de.wikipedia.org/wiki/CURL>

verfügbar sind, können Sie das konzeptionelle Verfahren auch in Ihrer LieblingsSprache vorfinden und entsprechend umsetzen.

Wir ersetzen den Telnet-Client zunächst durch einen TCP-Kanal, den wir programmatisch via Sockets öffnen. Das Beispiel ist absichtlich simplifiziert und vernachlässigt zur besseren Lesbarkeit z. B. die Behandlung von Exceptions, um den Blick für das Wesentliche nicht zu versperren.

Listing 2.1 Erzeugen eines Socket-Objekts

```
1 String serverHostname = "google.de";
2 Socket webServer = new Socket(serverHostname, 80);
```

Haben wir auf diese Weise ein Socket-Objekt erzeugt, benötigen wir noch einen InputStream, mit dem wir Daten empfangen, und einen OutputStream, mit dem wir Daten senden können. Die generischen Oberklassen können natürlich direkt verwendet werden. Dies wäre aber nur für Binärdaten sinnvoll und ratsam. Für textbasierte Daten, wie sie für HTTP benötigt werden, sind Klassen wünschenswert, mit denen wir mit den gewohnten und lieb gewonnenen print()- und println()-Methoden arbeiten können. Die PrintWriter-Klasse stellt diese bereit und kann auf Grundlage eines OutputStreams instanziiert werden. Selbiges gilt für die Klasse BufferedReader.

Listing 2.2 Erzeugen eines Input- und OutputStreams

```
1 PrintWriter out = new PrintWriter(webServer.getOutputStream(), true);
2 BufferedReader in = new BufferedReader(new InputStreamReader(
    webServer.getInputStream()));
```

Steht der TCP-Kanal, können wir die HTTP-Anfrage aus [Bild 2.8](#) absetzen. Genau die Tatsache, dass HTTP textbasiert ist, ermöglicht es uns nun, das Protokollelement als einfachen String über den PrintWriter in den Kommunikationskanal zu schreiben.

Listing 2.3 Erzeugen einer HTTP-Anfrage

```
1 out.print("GET /index.html HTTP/1.0\r\n");
2 out.print("\r\n");
```

Das kommt uns doch sehr bekannt vor. Bevor Sie sich jetzt aber anfangen zu wundern, wozu denn die zusätzlichen Escape-Sequenzen (\r\n) im String benötigt werden, möchten wir Sie um einige wenige Absätze Geduld bitten, nach denen das Geheimnis gelüftet wird.

Das Argument mit dem höheren Bedarf an Transfervolumen durch die Codierung als Text verliert zudem an Gewicht, da es durch zusätzliche Mechanismen abgeschwächt wird. Bevor die HTTP-Anfrage über TCP gesendet wird, kann diese komprimiert werden. Auf der Gegenseite wird sie dann entsprechend entpackt, bevor die HTTP-Verarbeitung beginnt. Als Komprimierungsverfahren schreibt HTTP hierfür gzip vor [[FGM+99](#), Kapitel 3].

Schließlich gilt nicht für alle Teile eines HTTP-Protokollelements, dass sie textbasiert sein müssen. Stellen Sie z. B. ein Pixelbild im GIF-, JPG- oder PNG-Format über einen Webserver zum Abruf bereit, so wird dieses nicht in Textform übertragen. Da derartige Daten nativ in einem Binärformat vorliegen, würde das sonst bedeuten, dass diese zunächst in ei-

ne Textrepräsentation überführt werden müssten. Hierfür stünde Ihnen z. B. die Base64-Codierung zur Verfügung. Mit dieser könnten Sie binäre Daten in Textdaten transformieren. Der Nachteil hierbei ist, dass die Base64-Codierung aus jeweils 6 Bit der Ursprungsdaten ein Textzeichen bestimmt, das wiederum mit 8 Bit codiert ist. Hieraus können Sie leicht erkennen, dass durch die Base64-Codierung ein Datenvolumen-Overhead entsteht, der nicht unwesentlich ist. Dieser liegt bei etwa 33 % ($8 / 6 = 1,33$). Daher sieht HTTP für den Transfer von Nutzdaten eine Ausnahme zur Textform vor. Sie steht eng in Verbindung mit dem Aufbau der Anfragen und Antworten, die wir uns im Folgenden genauer anschauen.

2.2.3 Aufbau von Request-Nachrichten und Protokollelemente

Der allgemeine Aufbau einer HTTP-Anfrage ist in drei verschiedene Bereiche aufgeteilt, die durch die folgende allgemeine Notation beschrieben sind [FGM⁺99]:

Listing 2.4 Notation des Aufbaus einer HTTP-Anfrage

```
<Method> <URL> <HTTP Version><CR><LF>
[Header Field Name: Value <CR><LF>]
...
[Header Field Name: Value <CR><LF>]
<CR><LF>
[Payload]
```

Alle Zeilen werden per Spezifikation durch die zwei Steuerzeichen Carriage Return (<CR>) und Line Feed (<LF>) abgeschlossen. Durch diese Festlegung werden insbesondere mögliche Inkompatibilitäten zwischen verschiedenen Systemen, die den Zeilenumbruch verschieden handhaben (Windows: <CR><LF>, Mac OS bis Version 9: <CR>, Mac OS X und Linux: <LF>), aus dem Weg geräumt.

Den wichtigsten Teil einer HTTP-Anfrage finden Sie unmittelbar in der ersten Zeile, die sogenannte Anfragezeile (engl. *Request Line*), die Sie für eine gültige Anfrage auch zwingend angeben müssen.

```
<Method> <URL> <HTTP Version><CR><LF>
```

Hier müssen Sie zunächst eine sogenannte Methode (<Method>) angeben. Sie können sich unter dieser Methode einen Schalter vorstellen, mit dem Sie verschiedene Protokollelemente an den Server senden bzw. verschiedene Aktionen auf dem Server auslösen können. Die bekannteste Methode ist die *GET*-Methode. Wie Sie sicherlich schon aus dem Namen geschlossen haben, bewirkt diese Methode den Abruf einer Ressource von einem Server. Um die angefragte Ressource zu benennen, muss als Nächstes die <URL>-Komponente der Anfragezeile mit der URL befüllt werden, die die angefragte Ressource identifiziert. Zur Erinnerung, eine URL hat den folgenden Aufbau:

```
<scheme>://<user>@<domain>:<port>/<path>?<query>#<fragment>
```

Im Web wird der mit *scheme* bezeichnete Teil mit dem Kürzel *http* ersetzt. Der *user* fällt im Web ersatzlos weg. Die *domain* entspricht dem Domainname der Website. Ist diese über

den Standardport 80 zugänglich, muss der port im Anschluss nicht mehr angegeben werden. Der path gibt den Namen der Ressource an, der sich auf der Website mit dem Domainname befindet. Hierbei kann es sich um eine statische Ressource handeln, die unter einem bestimmten Dateinamen ggf. einer bestimmten Verzeichniss hierarchie zugreifbar ist. Handelt es sich bei der Ressource um ein Programm, das in Abhängigkeit von Benutzereingaben eine Antwort dynamisch erzeugt, kommen zum Namen des Programms u.U. noch query-Parameter zur URL hinzu, die eben diese Benutzereingaben liefern können. Soll in der gelieferten Antwort an eine bestimmte Stelle gesprungen werden, so kann dies mit dem Fragmentbezeichner fragment erfolgen. Dies ist in der Regel allerdings nur mit Auszeichnungssprachen wie HTML und XML möglich, in denen Sie dafür im Dokument Ankerpunkte einbetten können.

Als letzter Eintrag in der Anfragezeile muss die Version von HTTP eingetragen werden, die der Client versteht. Aktuell können Sie hier HTTP/1.0 und HTTP/1.1 als mögliche Einträge vorfinden. Je nachdem, ob der anfragende Client mit dem Protokoll der Version 1.0 bzw. 1.1 arbeiten kann bzw. möchte.

GET

Die vermutlich einfachste HTTP-Anfrage, die Sie hieraus konstruieren können, ist die GET-Anfrage, die wir bereits in [Bild 2.8](#) im Telnet-Beispiel verwendet haben:

```
GET /index.html HTTP/1.0
```

Rufen wir uns die soeben besprochene Struktur einer HTTP-Anfrage in Erinnerung, so können wir die Ersetzung der Platzhalter mit konkreten Werten für eine GET-Anfrage nachvollziehen:

<Method>	<URL>	<HTTP Version>
GET	/index.html	HTTP/1.0

Bild 2.9 Struktur und Platzhalter einer HTTP-Anfrage

Die Methode wird treffenderweise für einen GET-Request mit dem Schlüsselwort GET belegt. Die *URL* bezeichnet die angefragte Ressource durch Angabe des Namens absolut zur Wurzel des Webserververzeichnisses (ohne Domänennamen). Als Version wird vom Client in diesem Beispiel 1.0 gewählt bzw. unterstützt und damit der dafür spezifizierte Wert HTTP/1.0 gesetzt.

Neben der GET-Methode spezifiziert HTTP sieben weitere Methoden, die für unterschiedliche Funktionen bereitstehen und denen wir uns in den nachfolgenden Unterkapiteln zuwenden wollen.

POST

Neben GET gehört die POST-Methode zu den am häufigsten verwendeten HTTP-Methoden. Im Vergleich zu GET können Sie allerdings mit der POST-Methode Daten zum Server schicken. Sie könnten uns jetzt entgegnen, dass dies doch auch über Query-Parameter

via GET möglich ist, und hätten damit auch vollkommen recht. GET Query-Parameter unterliegen allerdings einigen Einschränkungen. Der formale Aufbau der Daten ist bei beiden Methoden identisch. Es handelt sich also um Name-Wert-Paare. Sie werden sich erinnern, dass die GET-Parameter in der Anfragezeile als Teil der URL an den Server übertragen werden. Im Unterschied dazu überträgt die POST-Methode diese Paare aber im Payload-Bereich der Anfrage. Hier unterliegen die Daten praktisch keiner Beschränkung. Sie können somit also auch sehr umfangreiche Daten senden und diese können auch andere Codierungsformen als reinen Text annehmen. Dies macht POST zur bevorzugten Methode, wenn es darum geht, Daten an den Server zu übertragen. Laut der HTTP-Spezifikation sollte POST immer dazu verwendet werden, neue Ressourcen zu erstellen bzw. hinzuzufügen.

PUT

Die PUT-Methode funktioniert ähnlich wie ein POST, allerdings mit einer anderen Bedeutung (der Informatiker sagt auch gerne Semantik). Im Gegensatz zu POST, können Ressourcen durch PUT nicht nur erstellt, sondern auch aktualisiert werden. Der feine Unterschied liegt hier in der URL. Während der Server bei POST-Anfragen eine URL für neu erstelle Ressourcen bestimmt, kann der Client bei PUT selber entscheiden, wie die URL aussehen soll. Dazu fragt er diese mit PUT an. Falls auf dieser URL bereits eine Ressource vorhanden ist, wird diese aktualisiert.

DELETE

Mit der DELETE-Methode kann eine Ressource auf dem Server gelöscht werden. Die Ressource, die Sie löschen wollen, wird durch die URL eindeutig identifiziert. Bei einer DELETE-Nachricht ist der Payload also – ähnlich wie beim GET – leer und sie besteht somit aus der Anfragezeile und optionalen Headern.

HEAD

Die HEAD-Methode funktioniert ähnlich wie ein GET, liefert Ihnen allerdings nur einen Teil der Antwort zurück. Welcher Teil das ist, können Sie aus dem Namen der Methode bereits ganz treffend ableiten. Sie bekommen im Wesentlichen den Header der Antwort auf die Anfrage zurückgeliefert. Wie wir im anschließenden Kapitel noch näher sehen werden, beinhaltet der Antwort-Header typischerweise Informationen über das Betriebssystem und die Webserversoftware sowie Angaben zu den Daten in der Antwort. Häufig findet das HEAD-Protokollelement zur Erhebung von Statistiken Verwendung, da somit z. B. Analysen zur Verbreitung von Webserversoftware effizienter durchgeführt werden können als mit GET-Anfragen, da HEAD nicht den Inhalt der adressierten Ressource mitliefert.

OPTIONS

Welche von diesen hier aufgelisteten Kommunikationsmethoden ein Web- bzw. Applikationsserver bereitstellt, kann mit der OPTIONS-Methode herausgefunden werden. Wird die OPTIONS-Methode unterstützt, so bekommen Sie von ihr eine HTTP-Headerzeile mit dem Namen `Allow`, die mit Ihren Werten angibt, welche Methoden unterstützt werden. So bekommen Sie z. B. auf die Anfrage

```
OPTIONS / HTTP/1.1
Host: www.ebay.de
```

in der Antwort vom Server die Headerzeile

```
Allow: GET, HEAD, POST, TRACE, OPTIONS
```

zurückgeliefert, die angibt, dass die Server von eBay das Gros an HTTP-Methoden unterstützen.

TRACE

Diese Methode ist für Debugging-Zwecke vorgesehen. Wird diese Methode verwendet, antwortet der Webserver mit einem Echo auf die Anfrage. Sprich, Sie bekommen in der Antwort das zu sehen, was Sie zuvor im Request an den Server abgesetzt hatten. Dies kann dann z. B. so aussehen:

Listing 2.5 Beispielantwort eines Servers auf einen TRACE-Request

```
TRACE / HTTP/1.1
Host: www.example.org
```

```
HTTP/1.1 200 OK
...
Connection: close
Content-Type: message/http
Content-Length: 41
```

In der Antwort ist die Anfrage im Payload enthalten. Die auf den ersten Blick so harmlose Methode birgt ihre Gefahren und sollte daher nicht mehr verwendet werden. 2003 veröffentlichte Jeremiah Grossman den von ihm sogenannten Cross-Site-Tracing-(XST)Angriff [Gro03], mit dem Cookies ausgespäht werden können. Daher ist es ratsam, auf diese Methode serverseitig zu verzichten, was auch in der Regel berücksichtigt wird.

PATCH

Wie bereits erläutert, können mit der PUT-Methode Ressourcen auf einem Server aktualisiert werden, indem die Ressource auf einem Server komplett durch den Payload des PUT-Requests ersetzt wird. Partielle Modifikationen sind damit leider nicht möglich. Aus diesem Grund führte die IETF die PATCH-Methode ein, welche die Änderung von einzelnen Teilen ermöglicht [DS13]. Eine PATCH-Anfrage ähnelt daher einer PUT-Anfrage mit dem Unterschied, dass der Payload eine Beschreibung der partiellen Änderung enthält.

Das HTTP mitsamt seinen Methoden bildet zudem *eine* mögliche technische Implementierung des *REST-Konzeptes* von Roy Fielding [Fie00]. REST ist ein wichtiges Architekturkonzept, das den Aufbau von verteilten Systemen auf Basis von Ressourcen und deren Verwaltung mithilfe von festgelegten Methoden beschreibt.

2.2.4 Aufbau von Response-Nachrichten und Statuscodes

Vom Grundsatz her ähnelt der Aufbau der HTTP-Response dem des zuvor besprochenen HTTP-Requests. Auch hier finden Sie eine Dreiteilung vor, die sich aus einer Statuszeile, den Headern und dem Payload ergibt. Im Gegensatz zur HTTP-Anfrage unterscheidet eine HTTP-Antwort allerdings keine verschiedenen Protokollelemente. Sie bekommen auf jede der im vorangegangenen Kapitel beschriebenen Anfragen immer genau eine Antwort, die durch die folgende Notation beschrieben ist [FGM⁺99]:

Listing 2.6 Notation einer HTTP-Antwort

```
<HTTP Version> <Status> <Reason><CR><LF>
[Header Field Name: Value <CR><LF>]
...
[Header Field Name: Value <CR><LF>]
<CR><LF>
[Payload]
```

Die erste, auch als *Status-Line* bezeichnete Zeile ist spezifisch für die Response-Nachricht. Sie beginnt mit der *HTTP Version*, die der Server unterstützt. Die Werte, die dieser Eintrag annehmen kann, entsprechen dem des Requests, also *HTTP/1.0* oder *HTTP/1.1*. Danach folgt der Statuscode, der angibt, wie die Verarbeitung der Anfrage erfolgt ist. Der Status ist ein numerischer Code, der sich aus drei Ziffern zusammensetzt. Die erste Ziffer hat dabei eine bestimmte Bedeutung. Sie unterteilt die Codes in fünf verschiedene Kategorien:

- 1xx: geben informative Nachrichten
- 2xx: signalisieren die erfolgreiche Bearbeitung einer Anfrage
- 3xx: leiten den Client an eine andere Stelle weiter
- 4xx: zeigen einen Fehler aufseiten des Clients an
- 5xx: bedeuten eine Fehlfunktion auf der Serverseite

Da sich numerische Codes so schlecht im Kopf behalten lassen, wird dem Code ein kurzer beschreibender Text an die Seite gestellt (*HTTP!Reason*), der Aufschluss darüber geben soll, was der Statuscode anzeigt.

Statuscodes mit der Startziffer 1: Informatives

Die Gruppe der informativen Statusmeldungen beinhaltet aktuell zwei Mitglieder.

Tabelle 2.1 Statuscodes mit der Startziffer 1

Reservierter Bereich	Spezifizierte Statuscodes	Nachzulesen in
100–199	100 Continue 101 Switching Protocols	RFC2616 Kapitel 10.1

Der Statuscode 100 *Continue* (nur *HTTP/1.1*) informiert den Client z. B. darüber, dass die Verarbeitung der Anfrage erfolgen kann bzw. noch erfolgt. Dieser Antwortcode kommt immer dann zum Tragen, wenn die Verarbeitung einer Anfrage zeitaufwendig ist. Bevor z. B.

eine größere Datei zum Server übertragen wird, kann mit einer Anfrage, die den eigentlichen Inhalt zunächst ausspart und einen zusätzlichen Header einfügt, geprüft werden, ob die Datenmenge vom Server entgegengenommen wird. [Listing 2.7](#) gibt ein konkretes Beispiel.

Listing 2.7 Anwendungsbeispiel für den Statuscode 100

```
POST /videos HTTP/1.1
Host: media.server.net
Content-Type: video/mp4
Content-Length: 105910000
Authorization: Basic ZHVkZTpoYW5kc19vZmY=
Expect: 100-continue
```

Kann der Server die Datenmenge verarbeiten, dann kommt

HTTP/1.1 100 Continue

zurück, ansonsten wird durch einen Statuscode aus der 4xx- bzw. 5xx-Gruppe signalisiert, warum es nicht geht, wie z. B.

HTTP/1.1 413 Request Entity Too Large

oder allgemeiner

HTTP/1.1 417 Expectation Failed.

Die prinzipielle Funktionsweise sollte an diesem Beispiel klar geworden sein. Andere Beispiele können für POST- oder PUT- Anfragen zur Prüfung von Autorisierung konstruiert werden.

Das zweite Mitglied der 1xx-Gruppe ist der Statuscode 101 Switching Protocols (nur HTTP/1.1). Diesem Code werden wir noch häufiger begegnen, da er für WebSockets und insbesondere beim Aufbau einer WebSocket-Verbindung (siehe [Abschnitt 4.2](#)) eine große Rolle spielt. Der 101 Switching Protocols-Statuscode wird durch einen *Upgrade*-Header in der Anfrage ausgelöst. Das Umschalten auf ein anderes Protokoll erfolgt allerdings nur, wenn der Webserver das im Upgrade-Header der Clientanfrage angegebene Protokoll unterstützt. Außerdem sollte der Server der Umschaltungsanfrage nur dann nachkommen, wenn diese von Vorteil ist. Es wurde z. B. in der Vergangenheit häufig dazu verwendet, auf eine neuere HTTP-Version umzuschalten. Wenn der Switch vom Server durchgeführt wurde, dann findet er direkt schon in der positiv beschiedenen Antwort Anwendung, und zwar unverzüglich nach der Leerzeile, die Header von Nutzdaten trennt.

Das Umschalten von Protokollen war zunächst für die verschiedenen HTTP-Versionen vorgesehen und kam kaum zur Verwendung. Weitere Verwendungen kamen durch HTTP-Erweiterungen hinzu, die es z. B. ermöglichen, HTTP-Nachrichtenpaare verschlüsselt zu übertragen, ohne dass die gesamte Kommunikation auf SSL/TLS umgestellt werden mussten. Auch diese Erweiterungen finden in der Praxis bisher kaum Verwendung. Wie wir in [Kapitel 4](#) detaillierter besprechen werden, ist dieser Mechanismus sehr hilfreich für unsere WebSockets. Hier braucht es nämlich genau diese Funktion, um aus HTTP heraus auf ein anderes Protokoll umzuschwenken. Vorgreifend können Sie sich schon mal merken, dass

der Aufbau einer WebSocket-Verbindung vom Client aus mittels HTTP initiiert und dabei signalisiert wird, dass auf ein anderes Protokoll umgeschaltet werden soll, was in unserem Fall das WebSocket-Protokoll sein wird.

Statuscodes mit der Startziffer 2: Alles in Ordnung

Diese Statuscodes signalisieren, dass die zugehörige Anfrage des Client erfolgreich bearbeitet werden konnte. Je nachdem, um was für einen Anfragetyp es sich dabei gehandelt hat, besagt die Antwort Folgendes (siehe [Tabelle 2.2](#)):

Tabelle 2.2 Statuscodes mit der Startziffer 2

Reservierter Bereich	Spezifizierte Statuscodes	Nachzulesen in
200–299	200 OK 201 Created 202 Accepted 203 Non-Authoritative Information 204 No Content 205 Reset Content 206 Partial Content	RFC2616 Kapitel 10.2

Statuscodes mit der Startziffer 3: Hier geht es lang

Die Statuscodes der Gruppe mit der Startziffer 3 geben dem Client Auskunft darüber, wo er die angefragte Ressource auffinden kann. Dies können Umleitungen auf andere Server sein, aber auch Hinweise darüber, dass sich die Ressource noch unverändert im Cache des Clients befindet und noch von dort bezogen werden kann.

Tabelle 2.3 Statuscodes mit der Startziffer 3

Reservierter Bereich	Spezifizierte Statuscodes	Nachzulesen in
300–399	300 Multiple Choices 301 Moved Permanently 302 Found 303 See Other* 304 Not Modified 305 Use Proxy* 306 Switch Proxy 307 Temporary Redirect* *(ab HTTP/1.1)	RFC2616 Kapitel 10.3

Statuscodes mit der Startziffer 4: Clientseitige Schieflage

Die meisten Statuscodes finden Sie in der Gruppe mit der Startziffer 4, die Fehler in der Anfrage des Clients aufzeigen. Der bekannteste darunter, der Ihnen mit Sicherheit schon oft untergekommen ist, ist der Code 404 Not Found. Dieser teilt Ihnen mit, dass sich die von Ihnen angefragte Ressource nicht auf dem Server befindet.

Tabelle 2.4 Statuscodes mit der Startziffer 4

Reservierter Bereich	Spezifizierte Statuscodes	Nachzulesen in
400–499	400 Bad Request 401 Unauthorized 402 Payment Required 403 Forbidden 404 Not Found 405 Method Not Allowed 406 Not Acceptable 407 Proxy Authentication Required 408 Request Timeout 409 Conflict 410 Gone 411 Length Required 412 Precondition Failed 413 Request Entity Too Large 414 Request-URI Too Long 415 Unsupported Media Type 416 Requested Range Not Satisfiable 417 Expectation Failed	RFC2616 Kapitel 10.4

Statuscodes mit der Startziffer 5: Serverseitige Schieflage

Last but not least bekommen Sie durch Statuscodes der Gruppe mit der Startziffer 5 gesagt, dass die Verarbeitung Ihrer Anfrage durch einen Fehler auf dem Server nicht bearbeitet werden konnte. Als Webentwickler fürchtet man sich vor dem 500 Internal Server Error, da diese Antwort auf einen Programmierfehler in den eigenen serverseitigen Komponenten hinweist.

Tabelle 2.5 Statuscodes mit der Startziffer 5

Reservierter Bereich	Spezifizierte Statuscodes	Nachzulesen in
500–599	500 Internal Server Error 501 Not Implemented 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout 505 HTTP Version Not Supported	RFC2616 Kapitel 10.5

Neben diesen im HTTP-Standard spezifizierten Statuscodes kann es noch weitere geben, die von Erweiterungen definiert werden. Ein Beispiel hierfür können Sie im WebDAV-Protokoll finden.

Response-Header

Viele der Header, die Sie in eine Anfrage eintragen können, stehen auch für HTTP-Antworten bereit. Einige nicht, nämlich genau die, die für Anfragen vorbehalten sind. Derartige exklusive Header gibt es auch für Antworten. Die [Tabelle 2.6](#) zeigt eine Auflistung von dedizierten Response-Headern, die im WebSocket-Kontext relevant sind.

Tabelle 2.6 Ausgewählte HTTP Response-Header

Name	Beschreibung
Cache-Control Beispiel: Cache-Control: no-cache	Hier kann der Server definieren, wie lange und ob der Client die Ressource im Cache speichern darf.
Connection Beispiel: Connection: Upgrade	Hier wird dem Client die Art der Verbindung mitgeteilt. Bei einem WebSocket-Handshake kann hier der Server dem Client mitteilen, dass er die Art der Verbindung ändern will.
WWW-Authenticate Beispiel: WWW-Authenticate: Basic	Der Server kann dem Client mit diesem Header mitteilen, dass er sich für den Zugriff auf diese Ressource über einen HTTP-Authentifizierungsmechanismus (Basic oder Digest) authentifizieren muss.
Set-Cookie Beispiel: Set-Cookie: Play_FLASH=; Expires 13-Jun-2003; Path=/	Durch diesen Header kann der Server dem Client einen Cookie zuteilen.
Server Beispiel: Server: Apache	Der Server kann durch diesen Header seinen Namen bekannt geben.

2.2.5 Zustandslos

Der Austausch von HTTP-Nachrichtenpaaren ist **zustandslos**. Das bedeutet, dass eine Anfrage nichts von zuvor ausgetauschten Nachrichten weiß. Sie können sich vorstellen, dass das Protokoll kein Gedächtnis hat. Es vergisst, sobald eine Anfrage abgearbeitet ist, alles, was damit zu tun gehabt hat. Diese Eigenschaft bringt einige Vorteile mit sich, darunter z. B. eine schier unbegrenzte Skalierbarkeit.

Auf der anderen Seite hat die Zustandslosigkeit des Protokolls auch eine Kehrseite. Die mangelhafte Gedächtnisleistung macht das Etablieren und Aufrechthalten einer Sitzung zwischen Client und Server unmöglich. Damit so nützliche Dinge wie das Einloggen in eine Webanwendung oder auch ein Warenkorb realisiert werden können, muss daher in übergeordneten Schichten geackert werden.

3

Höhere Interaktivität und Echtzeitfähigkeit

Durch das besprochene Request-/Response-Modell, in dem immer der Client die Anfrage initiiert und eine darauf zurückgelieferte Antwort vom Server zur Folge hat, dass die dargestellte Webseite vollständig neu gerendert wird, ist die Umsetzung von Webanwendungen mit einer höheren Interaktivität und Echtzeitfähigkeit zunächst nicht realisierbar. Hierfür bedarf es zusätzlicher Mechanismen, mit denen diese Anforderungen bedient werden können.



Fragen, die dieses Kapitel beantwortet:

- Wie sind die Anforderungen an eine höhere Interaktivität und Echtzeitfähigkeit von Webanwendungen über die Zeit in neue Ansätze und Technologien gemündet?
- Was für Verfahren sind daraus hervorgegangen (im Wesentlichen XMLHttpRequest, Polling, Long-Polling und COMET/HTTP-Streaming) und wie funktionieren diese?
- Wie haben sich schließlich die aktuellen Ansätze in der HTML5-Standardfamilie Server-Sent Events und WebSockets daraus entwickelt?

■ 3.1 XMLHttpRequest (XHR)

Durch den Einzug von XMLHttpRequest (XHR) [KASS14] in den Browser steht dem Webentwickler eine vom W3C standardisierte API zur Verfügung, mit der JavaScript-gesteuert HTTP-Requests abgesetzt werden können. Der JavaScript-Codeschnipsel in Listing 3.1 zeigt Ihnen exemplarisch, wie dies für eine GET-Anfrage erfolgt.

Listing 3.1 Exemplarische GET-Anfrage eines JavaScript-gesteuerten HTTP-Requests

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("GET", "http://www.example.com", true);
3 xhr.onreadystatechange = function() {
4     if(this.readyState == 4 && this.status == 200) {
5         console.log(this.responseText);
6     }
7 }
8 xhr.send();
```

Als Erstes brauchen Sie ein xhr-Objekt, das Sie durch den Konstruktorauftrag – wie in Zeile 1 abgedruckt – erzeugen können. Steht Ihnen eine Instanz eines xhr-Objekts zur Verfügung, können Sie darüber HTTP-Anfragen absetzen. In Zeile 2 wird eine GET-Anfrage an den Host `www.example.com` vorbereitet. Damit die Kommunikation den Programmablauf nicht blockiert, registrieren Sie als Nächstes eine Funktion, die aufgerufen werden soll, sobald sich der Zustand, in dem sich das xhr-Objekt befindet, ändert. Hat sich dieser geändert, können Sie einige Dinge prüfen. Im Beispiel wird geprüft, ob sich das Objekt im Zustand mit der Nummer 4 befindet. Wenn Sie im genannten W3C-Standard nachschauen, sehen Sie, dass dieser Zustand angenommen wird, sobald die ganze Antwort des Servers auf der Clientseite eingegangen ist und verarbeitet wurde. Ist dem so und war die GET-Anfrage zudem erfolgreich (der Statuscode in der Response ist 200), werden die empfangenen Nutzdaten in der JavaScript-Konsole ausgegeben. Bis jetzt ist aber noch nichts passiert. Die Kommunikation wurde bisher nur vorbereitet. Der entsprechende Request wird erst durch das Aufrufen der `send()`-Methode ausgelöst. Die Kommunikation läuft dann asynchron ab, was bedeutet, dass der Programmablauf durch die Kommunikation nicht blockiert wird. Andere JavaScript-Komponenten auf der aktuell angezeigten Webseite werden also nicht beeinflusst und laufen wie gewohnt weiter. Diese Möglichkeit, asynchrone Anfragen an den Server stellen zu können, ohne dass die dargestellte Webseite blockiert oder gar neu geladen bzw. neu gerendert wird, bildet die technische Grundlage für moderne Web-Frontends, die auch unter dem Oberbegriff Ajax (*Asynchronous JavaScript and XML*) geführt werden.

Bild 3.1 veranschaulicht den Lebenszyklus eines XHR-Objekts und die Zustände, die es dabei durchlaufen kann. Einen Überblick über die Konstruktoren, Methoden, Attribute und Events eines XHR-Objekts finden Sie in Anhang A.

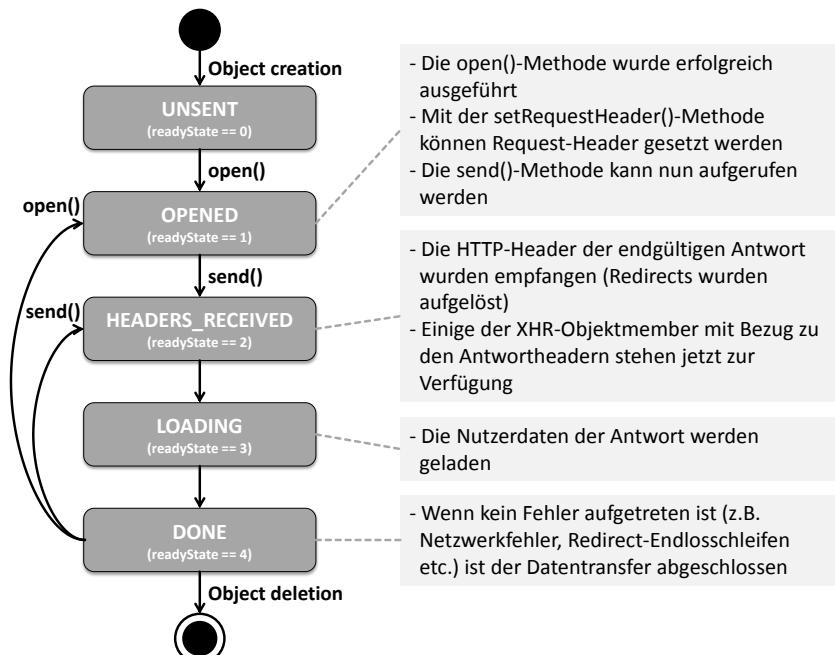


Bild 3.1 UML-Zustandsdiagramm eines XHR-Objekts gemäß dem W3C Working Draft [KASS14]

■ 3.2 Polling

Viele Echtzeit-Webanwendungen werden heute durch *Polling* oder Long-Polling realisiert. Beim einfachen Polling werden in einem festgelegten Zeitabstand (z. B. alle zwei Sekunden) Anfragen an den Server gestellt, auf die umgehend geantwortet wird. Das zugrunde liegende Prinzip können Sie an dem folgenden JavaScript-Codefragment nachvollziehen:

Listing 3.2 Grundprinzip von Polling

```
1 function poll() {
2     xhr.send();
3     clearTimeout(timeoutId);
4     timeoutId = setTimeout(poll(), 2000);
5 }
6 timeoutId = setTimeout(poll(), 2000);
```

Das Objekt `xhr` ist so instanziert und initialisiert worden wie zuvor beschrieben. Der Unterschied besteht hier im wiederholten Aufrufen der `send()`-Methode, durch die `setTimeout()`-Funktion. Auf diese Weise können Sie eine Ressource auf dem Server periodisch anfragen und die Antwort auf der Clientseite verarbeiten, was ggf. eine Aktualisierung der dem Benutzer angezeigten Daten mit sich bringt.

Haben sich seit der letzten Anfrage keine Änderungen mehr ergeben, so wird dies durch eine leere Antwort signalisiert. **Bild 3.2** zeigt ein Polling-Kommunikationsszenario, aus dem Sie die Funktionsweise und Eigenschaften des Ansatzes ablesen können.

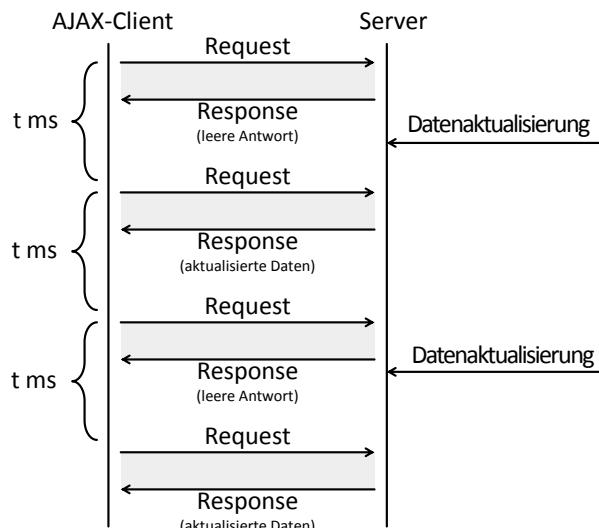


Bild 3.2 Polling

Polling ist einfach und browserdeckend zu implementieren. Der sich damit einstellende Overhead, der durch unnötige Anfragen und den damit verbundenen Kommunikations-

und Verarbeitungsaufwand zu Buche schlägt, ist allerdings nicht zu unterschätzen. Bei einer Entscheidung für oder wider Polling gilt es insbesondere, die benötigte zeitliche Auflösung zu berücksichtigen. Für Internetauktionen und Chat-Systeme muss diese naturgemäß recht fein sein, damit sich Veränderungen am Gebot rechtzeitig oder abgesetzte Nachrichten zeitnah auf dem Bildschirm der Clients einfinden. Derartige Anforderungen an die Aktualität der im Browser dargestellten Daten sind mit Polling nur durch einen erhöhten Ressourceneinsatz zu bedienen. Ein Dashboard zur Kontrolle des Wasserbedarfs von Grünpflanzen hingegen, kann ohne Weiteres effizient mit Polling implementiert werden. Hier können die Zeitintervalle großzügig definiert werden, da es keine gravierenden Auswirkungen hat, wenn z. B. im Stundenrhythmus ein Polling-Request abgesetzt wird und die darin evtl. enthaltenen Daten im ungünstigsten Fall fast eine Stunde alt sind. Das sollte weder den Server noch die Grünpflanzen zu sehr beanspruchen.

■ 3.3 Long-Polling

Um den Overhead des Polling-Verfahrens zu reduzieren, tritt das *Long-Polling* mit der Abwandlung an, dass der Server nur dann prompt antwortet, wenn auch eine Änderung vorliegt, die er an den Client kommunizieren kann. Liegt nichts Neues vor, so antwortet er zunächst nicht und wartet eine festgelegte Zeit bei offener Verbindung auf zwischenzeitlich eintreffende Neuigkeiten (siehe [Bild 3.3](#)).

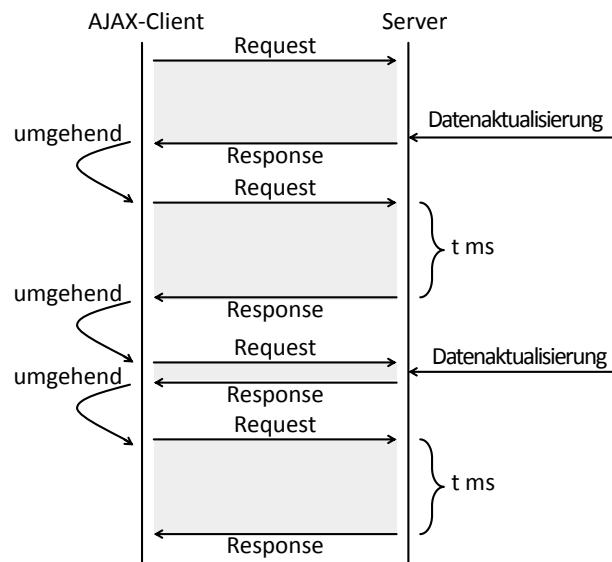


Bild 3.3 Long-Polling

Sobald Änderungen im Wartezustand eintrudeln, werden diese direkt über die offene Verbindung in der Response mitgeteilt und die Verbindung wie gewohnt geschlossen. Kommt

es zu keinen Änderungen bzw. Neuigkeiten im Wartezustand, so wird nach Ablauf der vordefinierten Wartezeit eine leere Antwort geschickt und die Verbindung geschlossen. Der Client öffnet in beiden Fällen postwendend eine neue Long-Polling-Verbindung, sodass der Server über eine quasi andauernde Verbindung zum Client verfügt.

■ 3.4 Comet

Mit Comet [Rus06], dem zweiten gebräuchlichen Haushaltsputzmittel in den USA neben Ajax, steht ein Programmiermodell zur Verfügung, das ebenfalls die Long-Polling-Mechanismen beinhaltet, um Serverbenachrichtigungen in Webanwendungen zu integrieren, und ergänzt diese durch einige zusätzliche Spielarten. Comet sieht z. B. die Methode vor, anstelle von XMLHttpRequest JSONP [Sim10] zu verwenden. Die Idee hierbei ist, ein `<script>`-Element dynamisch zu erzeugen und dem DOM der geladenen Webseite hinzuzufügen. Das angefragte Skript enthält dabei vornehmlich JSON-Daten, die geeignet verpackt werden müssen, um sie mit JavaScript verarbeiten zu können. Die Verarbeitung wird durch Events initiiert, die im geladenen Skript ebenfalls enthalten sind. Sind die Daten lokal im Client verarbeitet, so wird zu guter Letzt, dem Polling-Muster entsprechend, das nächste Script-Element generiert und der identische Prozess erneut durchlaufen. Interessant an dieser als *Script Tag Long Polling* bekannten Methode ist, dass die JSON-Daten über das `src`-Attribut des `<script>`-Elements von einer (Sub-)Domain „gepollt“ werden können, die sich von der geladenen Webseite unterscheiden kann. Damit zählt das Script Tag Long Polling zu den wenigen Verfahren, die die Same Origin Policy des Webbrowsers zu überwinden vermögen, was durch CORS (Cross-Origin Resource Sharing) [Kes14] abgelöst werden soll. Auch in der neuen Version des XHR-Objekts im Browser – XHR in Level 2 – sind Mechanismen definiert, mit denen die Prüfung des Ursprungsortes einer Ressource selektiv außer Kraft gesetzt werden kann.

Außerdem sieht Comet auch Varianten vor, die auf persistenten HTTP-Verbindungen (engl. *Persistent Connection*) aufbauen. Der Webserver schließt hierbei die TCP-Verbindung nicht umgehend nach Beantwortung eines HTTP-Requests, sondern hält diese eine vorbestimmte Zeit offen, damit der TCP-Kanal zur Abarbeitung weiterer HTTP-Anfragen des Browsers an den Server verwendet werden kann. Dies ist insbesondere für Hypertext-Dokumente wie HTML sinnvoll, die eine Vielzahl von Ressourcen vom Server laden müssen, damit sie vollständig gerendert und angezeigt werden können. Nachrichtenseiten wie z. B. heise¹ können dabei leicht auf über 100 Ressourcen pro Seite kommen (Bilder, Skripte, Styles, Schriften usw.). Gesteuert wird dies über den HTTP-Header `Connection`. Trägt dieser den Wert `close`, so wird der TCP-Kanal nach Auslieferung der Response geschlossen (Default für HTTP 1.0). Ist der Wert des `Connection`-Headers auf `keep-alive` gesetzt, bleibt die TCP-Verbindung geöffnet (Default für HTTP 1.1). Eine aufrechterhaltene Verbindung kann von einer Comet-Umgebung verwendet werden, um mit einem unsichtbaren Inline-Frame, der kontinuierlich mit JavaScript-Fragmenten gefüllt wird, einen Server-Push zu realisieren. Die tröpfchenweise eintreffenden JavaScript-Anweisungen werden

¹ www.heise.de/

vom Interpreter im Browser sukzessive ausgeführt und können somit zur servergesteuerten Aktualisierung des Clients herangezogen werden.

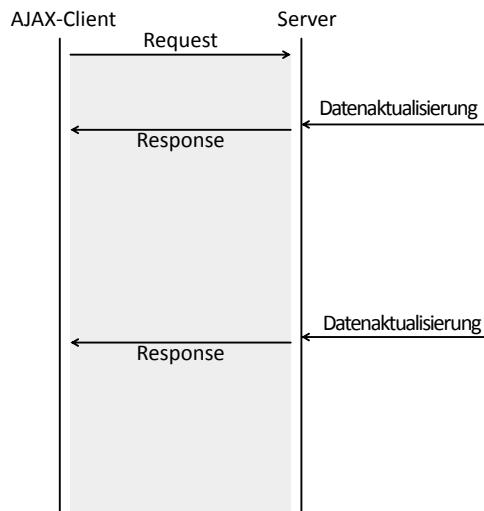


Bild 3.4 HTTP-Streaming

Einen großen Nachteil von Comet können Sie in der Tatsache finden, dass es sich hierbei nicht um einen Standard handelt. Das Fehlen von verbindlichen Spezifikationen steht häufig einer breiten Verwendung entgegen. Um dem zu begegnen, ist mit der Definition des Bayeux-Protokolls [RWDN07] eine Initiative ins Leben gerufen worden, die eine entsprechende Norm für die Entwicklung Comet-basierter Webbenachrichtigungen festlegt, die Entwicklern in erster Linie eine leichtere Umsetzung und höhere Interoperabilität gewährleisten soll. Um insbesondere den letzten Punkt realisieren zu können, hat die Dojo Foundation im Rahmen des CometD-Projekts [The14] eine Reihe von Implementierungen des Bayeux-Protokolls in verschiedenen Programmiersprachen – darunter z. B. JavaScript, Java, Perl und Python – bereitgestellt. Da sich das Bayeux-Protokoll allerdings nicht den Einzug in etablierte Standardisierungsorganisationen bahnen konnte, es aber nunmehr alternative Entwicklungen in der IETF und dem W3C gibt, empfehlen wir Ihnen, sich auf Letztere zu konzentrieren. Das W3C fokussiert sich dabei auf Standardisierungsarbeiten rund um JavaScript-APIs für die HTML5-Standardfamilie. In unserem Kontext sind die Kommunikationserweiterungen von Interesse, die in der Gruppe HTML5 Connectivity zusammengefasst sind. In dieser Kategorie finden Sie die Standards Server-Sent Events (SEE, siehe Abschnitt 3.5) und WebSockets (siehe Bild 1.2 und Bild 3.5).

Für jede Gruppe von inhaltlich verwandten Standards gibt es neben einem Gruppennamen auch ein eigenes Logo. Diese Logos für die verschiedenen als *Technology Classes* bezeichneten Gruppen können Sie Bild 3.6 entnehmen. Weitere Informationen hierzu und über die Möglichkeit, Ihr eigenes HTML5-Logo zusammenzustellen, haben Sie auf der extra dafür veröffentlichten Webseite des W3C.²

² <http://www.w3.org/html/logo/>



Bild 3.5 HTML5 Connectivity-Logo [W3C14]



Bild 3.6 HTML5-Logo mit allen Technology Classes [W3C14]

■ 3.5 Server-Sent Events

Die bewährten Teile aus den aufgezeigten Server-Push-Technologien sind in HTML5 im Server-Sent Event (SSE) [Hic12a] Standard verschmolzen worden. SSE definiert eine API, mit der sich HTTP-Verbindungen herstellen lassen, die der Webserver zum spontanen Senden an den Client verwenden kann. In der W3C „Candidate Recommendation“ ist dafür das EventSource-Objekt eingeführt und seine Schnittstelle spezifiziert worden. Ein EventSource-Objekt repräsentiert die Clientseite, die Benachrichtigungen empfangen kann. Die Benachrichtigungen werden in Form von DOM-Events an die Anwendung weitergegeben. Es werden drei Event-Handler spezifiziert, die vom Browser entsprechend anzubieten sind. Der onopen-Handler wird aufgerufen, wenn die SSE-Verbindung geöffnet worden ist. Tritt ein Fehler auf, wird der onerror-Handler aktiviert. Das Eintreffen von Servernachrichten wird durch den onmessage-Event signalisiert und an die registrierte Funktion zur Weiterverarbeitung weitergereicht (siehe Listing 3.3).

Listing 3.3 Aufbau eines SSE-Kanals und das Lauschen auf Serverbenachrichtigungen

```

1  <html>
2      <head>
3          <script>
4              var stats = new EventSource('stats');
5              /*

```

```

6          * Für jede eintreffende Nachricht vom Server
7          * wird die hier registrierte Funktion aufgerufen
8          * und ausgeführt.
9          */
10         stats.onmessage = function (event) {
11             statsDiv = document.getElementById('stats');
12             statsDiv.innerHTML += "<br>" + event.data;
13         };
14     </script>
15 </head>
16 <body>
17     <div id="stats"></div>
18 </body>
19 </html>
```

Wird die in [Listing 3.3](#) dargestellte Webseite in einen Browser geladen, so wird durch das Erzeugen des EventSource-Objekts ein HTTP GET-Request an die im Aufruf angegebene URL abgesetzt, der den SSE-Kanal etabliert (siehe [Listing 3.4](#)). Für den SSE-Request sind keine spezifischen HTTP-Header zwingend erforderlich. Der Working Draft rät allerdings, den für SSE registrierten MIME-Typ `text/event-stream` im Accept-Header einzutragen, das Caching der SSE-Requests zu unterbinden sowie sicherzustellen, dass es sich um eine persistente Verbindung handelt (`Connection: keep-alive`).

Listing 3.4 Wesentliche Komponenten des GET-Requests, der bei der Erzeugung eines EventSource-Objekts generiert wird

```

GET /stats HTTP/1.1
[...]
Accept: text/event-stream
Last-Event-Id: 6
Cache-Control: no-cache
Connection: keep-alive
```

Auf der Serverseite muss nicht viel getan werden. Wesentlich ist hier, dass die Verbindung zum Client nicht geschlossen wird und dass die definierten Felder in der vorgegebenen Syntax in die offene Response geschrieben werden. [Listing 3.5](#) zeigt eine dortige (noch offene) Response. Auch in diesem Beispiel sind nur die für SSE relevanten Header enthalten. Der Content-Type muss auf den für SSE registrierten MIME-Type `text/event-stream` gesetzt sein und das Caching der SSE-Response unterbunden werden.

Listing 3.5 Offene HTTP-Response, die es dem Server erlaubt, spontane Nachrichten an den Client zu versenden

```

HTTP/1.1 200 OK
[...]
Content-Type: text/event-stream
Expires: Mon, 01 Jan 2015 00:00:00 GMT
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Connection: close
```

```

id: 7
data: {"servername": "tatooine", "cpu-load": "45", "time": "14:33:48"}

id: 8
data: {"servername": "tatooine", "mem-usage": "78", "time": "14:33:48"}

... weitere Serverbenachrichtigungen können folgen ...

```

In der Response aus Listing 3.5 sind zwei Events enthalten, die in dem Beispiel Werte zur Auslastung von Servern bereitstellen. Gemäß den Definitionen zum text/event-stream MIME-Type besteht ein Event aus keiner, einer oder mehreren Kommentarzeilen, die mit einem Doppelpunkt beginnen und einer oder mehreren Zeilen mit Datenfeldern. Datenfelder sind durch einen Doppelpunkt getrennte Name-Wert-Paare, wobei aktuell die vier Namen event, data, id und retry spezifiziert sind. Andere Feldnamen werden ignoriert. Als Wert kann ein beliebiger Textstring eingesetzt werden. In Listing 3.5 sehen Sie z. B. einen JSON-formatierten String, der die Messdaten bereitstellt. Events werden durch eine Leerzeile voneinander getrennt.

Durch die Zugehörigkeit zur HTML5-Standardfamilie ist eine breite Browserunterstützung für SSE zu erwarten. Tatsächlich hat gemäß „Can I use“ auch ein Großteil der neueren Browser SSE mit an Bord [Dev14a]. Die einzige Ausnahme stellt hier der Internet Explorer dar, der SSE bisher nicht unterstützt. Laut dem angegebenen Status auf der Internetseite modernIE stehen die SSE für die kommende Version 12 des Microsoft User Agents zur Diskussion.³ Als Fallback kann auf die eingangs diskutierten wegbereitenden Polling-Spielarten zurückgegriffen werden, was insbesondere für die Internet Explorer-Unterstützung vorgesehen werden muss. Abschließend kann festgehalten werden, dass mit SSE ein unidirektionaler Kanal vom Server zum Client aufgebaut werden kann, der für Anwendungsfälle interessant ist, die eine Serverbenachrichtigungsfunktion benötigen. Das Erklingen eines freundlichen „Sie haben Post“-Hinweises ist ein Paradebeispiel für den Einsatz von SSE. Weitere sind Newsfeeds, Finanzdaten- und Sportticker sowie Monitoring von Haushalt und Industrie.

■ 3.6 Bewertung der Verfahren

Polling bzw. Long-Polling hat unter anderem den Nachteil, dass pro Request-/Response-Nachrichtenpaar jeweils ein Header benötigt wird, was zu sehr hohen Datenmengen führen kann. Hat der Server keine neuen Daten, wird eine informationslose Nachricht mit einem Request- und Response-Header erzeugt.

Trotz der angerissenen Variantenvielfalt bleiben Nachteile dieser konstruierten Lösungen für Serverbenachrichtigungen bestehen. Im Detail finden sich spezifische Schwächen hinsichtlich der Kontrolle der Kommunikationsverbindung, der Behandlung von Fehlern und der Sicherheit. Zudem bereitet der bunte Strauß an herangewachsenen Technologien nicht

³ <https://status.modern.ie/serversenteventseventsources?term=Server-Sent%20Events>

gerade einen geradlinigen Einstieg in die Thematik, da es zunächst darum geht, ein Verständnis für den Kessel Buntes zu entwickeln. Der Hauptnachteil ist und bleibt aber im kontinuierlichen Anfragen von Daten begründet. Ein besserer Ansatz würde eine tatsächliche Verbindung etablieren, die vom Server genutzt werden kann, anstatt eine scheinbare Verbindung durch das sich stetig wiederholende Anfragen des Servers zu emulieren. Eine derartige Technologie ist durch WebSockets gegeben, die in der HTML5-Standardfamilie enthalten sind und denen wir für die restliche gemeinsame Zeit mit diesem Buch unsere volle Aufmerksamkeit zukommen lassen.

4

Die Leitung: Das IETF WebSocket-Protokoll

Grundsätzlich können Serverbenachrichtigungen in Form der am Ende des vorherigen Kapitels genannten Beispiele auch mit WebSockets realisiert werden, da dieser Standard einen bidirektionalen Kanal zwischen Webclient und Webserver einführt. Ein WebSocket ermöglicht aber weit mehr, als nur Benachrichtigungen vom Server zum Client zu senden. Über den bidirektionalen Kanal kann auch der Client an den Server Daten schicken und diese können – im Unterschied zu allen bisherigen Technologien – auch Daten in binärer Form sein. Was Sie vorher unter sehr viel Schweißeinsatz nur hätten selbst bauen können, steht nun in standardisierter Form bereit. Dies öffnet die Tür für eine Palette von Anwendungen, die über ein simples Server-Push hinausgehen, zu denen z. B. Spiele, jegliche Art von Konversationen (Text, Sprache, Sprache und Video) und Kollaborationen sowie das Benutzermonitoring im Rahmen von Usability-Studien zählen. Ein Vorteil dieser Technologie ist auch ein kleiner Overhead. Pro Nachricht werden nur 2 bis 6 Bytes benötigt.



Fragen, die dieses Kapitel beantwortet:

- Wie ist die Entstehungsgeschichte des WebSocket-Protokolls?
- Wie funktioniert der Nachrichtenaustausch für den Verbindungsaufbau?
- Wie ist der Aufbau der Protokollnachrichten?
- Wie funktioniert der Nachrichtenaustausch für den Verbindungsabbau?
- Was gibt es für Werkzeuge zur Analyse der WebSocket-Kommunikation?

■ 4.1 Was bisher geschah

Das WebSocket-Protokoll hat einen weiten Weg zurückgelegt, bis es schließlich im Dezember 2011 als RFC (Request For Comments) 6455 [FM11a] das Licht der Welt erblickte. Bereits im Jahr 2009 wurde es von Google der IETF (Internet Engineering Task Force) vorgeschlagen, als Internet Draft in der Version *draft-hixie-thewebsketchprotocol-00* [Hic09]. Google und andere arbeiteten und verbesserten das noch junge Protokoll kontinuierlich weiter. Im Mai 2010 fanden die Arbeiten mit dem *draft-hixie-thewebsketchprotocol-76* [Hic10] zunächst ein Ende. Ab August 2010 wurde die offizielle Weiterentwicklung durch

die „*BiDirectional or Server-Initiated HTTP (HyBi)*“-Arbeitsgruppe der IETF fortgeführt. In etwas mehr als einem Jahr wurden hier erhebliche Fortschritte gemacht. Neben der Einführung des Supports für den Austausch von Binärdaten wurden auch Sicherheitsprobleme in Verbindung mit älteren Proxy-Servern behoben [FM11b]. Im Dezember 2011 war es dann so weit und das WebSocket-Protokoll wurde von der IETF in den Stand eines RFC gehoben und mit der Nummer 6455 versehen.

■ 4.2 Opening-Handshake – WebSocket-Verbindung aufbauen

Der WebSocket-Standard setzt sich aus der Spezifikation des Protokolls (standardisiert durch die IETF in RFC 6455 [FM11a]) und der JavaScript-Webbrowser-API (genormt durch das W3C [Hic12b]) zusammen. Wird im Browser – wie wir noch in [Kapitel 5](#) besprechen werden – eine WebSocket-Verbindung aufgebaut, so kommt es zunächst zum Verbindungsauftakt, dem sogenannten Opening-Handshake oder auch WebSocket-Handshake. Auf Basis von HTTP wird dabei ein WebSocket-Kanal geöffnet. Hierfür setzt der Client einen speziell aufgebauten HTTP-GET-Request ab, der z. B. die in [Listing 4.1](#) gezeigte Gestalt haben kann.

Listing 4.1 WebSocket Opening-Handshake-Request

```
GET /chat HTTP/1.1
Host: domain.net
Connection: Upgrade
upgrade: websocket
Sec-WebSocket-Key: dGhIHNhbXBsZSBub25jZQ==
Origin: http://domain.net
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Extensions: x-webkit-deflate-stream
[...]
```

Mit GET /chat wird der Endpoint des jeweiligen WebSocket-Servers angesprochen. Ein Server kann mehrere WebSocket-Endpoints anbieten. Mit Connection: Upgrade wird der Server aufgefordert, vom HTTP-Protokoll auf ein anderes Protokoll umzuschalten. Auf welches Protokoll umgestellt werden soll, wird mittels des Upgrade-Headers signalisiert; im Beispiel also durch Upgrade: websocket auf das WebSocket-Protokoll. Der Header Sec-WebSocket-Key ist eine Base64-kodierte Zeichenkette, die eine Zufallszahl beinhaltet. Dieser Header dient zur Überprüfung, ob der Server das WebSocket-Protokoll unterstützt. Durch den Origin kann der Client dem Server mitteilen, von welcher Ursprungs-Domain der Request versendet wurde. Der Server hat demzufolge die Möglichkeit zu entscheiden, ob er die Verbindung annimmt. Mit dem Header Sec-WebSocket-Version: 13 wird die Versionsnummer des WebSocket-Protokolls angegeben, wobei sich der Zahlenwert an der Nummerierung der unterschiedlichen IETF Drafts orientiert. Draft Nummer 17 wurde schließlich zu RFC 6455. Der Client kann außerdem optional mit dem

Sec-WebSocket-Protocol-Header den Server darüber in Kenntnis setzen, welche Subprotokolle über den WS-Kanal verarbeitet werden können. Auch Erweiterungen, die der Client unterstützt, können angefragt werden. Unterstützt der Server dieselbe Erweiterung, gibt er diese in der Antwort an. Im obigen Beispiel fragt ein Webkit-Browser den Server an, ob er die Erweiterung `x-webkit-deflate-stream` unterstützt. Dabei handelt es sich um ein Deflate-Kompressionsverfahren [Yos14]. Falls der Server diese Erweiterung nicht kennt, wird dieser Teil des Request-Headers einfach weggelassen. Der Client hat auch die Möglichkeiten, mehrere Kompressionsverfahren bzw. Erweiterungen anzufragen. Der Server kann dann auf eine dieser Erweiterungen antworten.

Wenn mehrere Erweiterungen angefragt werden, kann der Server zugleich dem Client antworten, dass er alle diese unterstützt. Die Reihenfolge der Felder im Handshake wird vom Client zufällig bestimmt und ist von daher nicht von großer Bedeutung. Enthält ein Handshake-Request nicht alle geforderten Felder, kommt die Verbindung nicht zustande. Unbekannte Header-Felder werden vom WebSocket-Server ignoriert. Akzeptiert der Server die Handshake-Anfrage des Clients, antwortet dieser mit einer HTTP-Response, die mit dem Statuscode `101 Switching Protocols` bestätigt, dass der Protokollwechsel stattfinden kann (siehe [Listing 4.2](#)).

Listing 4.2 WebSocket Opening-Handshake-Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Der Sec-WebSocket-Accept wird durch das Anhängen eines Globally Unique Identifiers (der in diesem Fall als

`258EAFA5-E914-47DA-95CA-C5AB0DC85B11`

definiert ist) an den Sec-WebSocket-Key des Handshake-Requests des Clients erstellt (siehe [Bild 4.1](#)). Aus der somit konstruierten Zeichenkette der Form

`dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11`

wird mittels der kryptografischen Hashfunktion SHA-1 ein Hashwert erzeugt, der anschließend durch eine Base64-Codierung [Jos06] zu einem Textstring konvertiert wird. Die so gebildete Zeichenkette ergibt dann den Wert für den Sec-WebSocket-Accept-Header. Der Client besitzt jetzt die Möglichkeit, mit seinem Sec-WebSocket-Key und dem zurückgelieferten Sec-WebSocket-Accept zu prüfen, ob die Antwort wirklich von dem WebSocket-Server kommt. Hierzu verfährt der Client, wie es [Bild 4.2](#) wiedergibt.

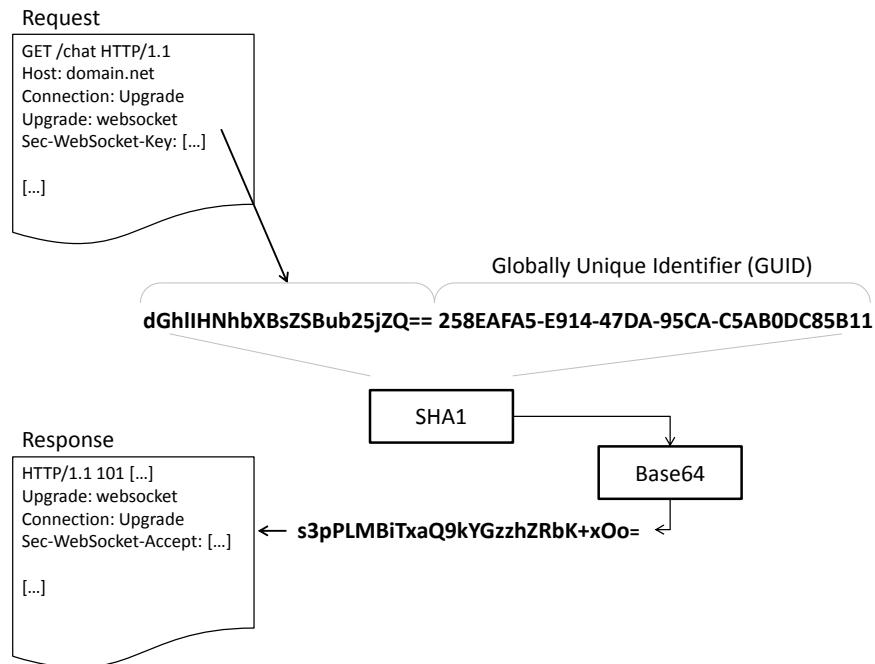


Bild 4.1 Konstruktion des Sec-WebSocket-Accept-Wertes auf der Serverseite

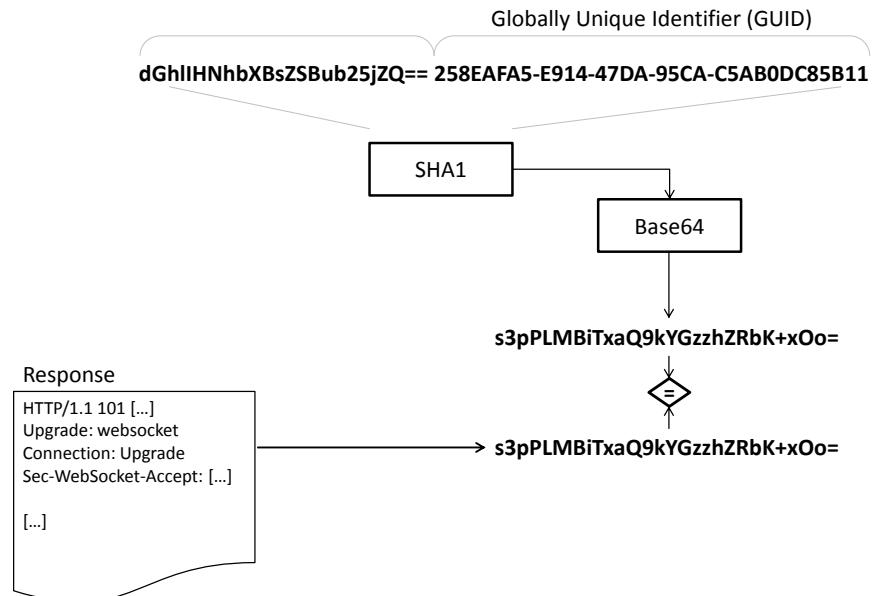


Bild 4.2 Prüfen des Sec-WebSocket-Accept-Wertes auf der Clientseite

Sind Subprotokolle bei der Handshake-Anfrage des Clients mit angegeben, so teilt der Server mit dem Eintrag Sec-WebSocket-Protocol mit, welches der Subprotokolle er versteht bzw. unterstützt. Versteht der Server die angeforderten Subprotokolle nicht, soll er gemäß RFC 6455 den Verbindungsaufbau abbrechen. Antwortet ein Server mit einem Subprotokoll, das nicht vom Client angefragt wurde, soll wiederum gemäß den Festlegungen im RFC der Client den Verbindungsaufbau abbrechen. Viele der verfügbaren Implementierungen halten sich an diese Spezifikation, andere nicht.

Wurden alle genannten Schritte und die darin enthaltenen Prüfungen erfolgreich durchlaufen, ist der Handshake abgeschlossen und der WebSocket-Kanal etabliert. Über diesen können nun Daten ausgetauscht werden, die als Frames den folgenden standardisierten Aufbau haben.

■ 4.3 WebSocket-Frames

Die Daten, die über einen WebSocket-Kanal versendet werden, werden als eine Sequenz von Frames transportiert. Der generelle Aufbau ist wie bei Protokollnachrichten üblich, eine Zweiteilung in einen Header mit Kontrolldaten und den Nutzdaten im Payload. Eine vereinfachte Darstellung des Aufbaus eines WebSocket-Frames gemäß der Spezifikationen im RFC 6455 können Sie Bild 4.3 entnehmen.

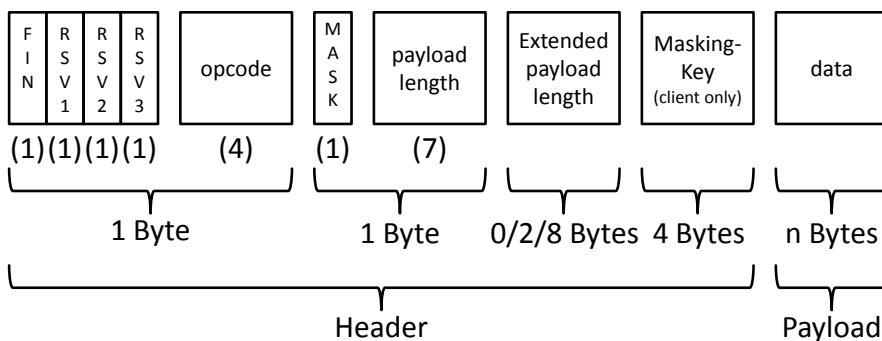


Bild 4.3 Überblick über den Aufbau eines WebSocket-Frames

Der überblicksartigen Darstellung können Sie sofort entnehmen, wie schlank das Protokoll ist. Allein der Header eines WebSocket-Frames beträgt nur einen Bruchteil des Headers einer HTTP-Anfrage. Ein WebSocket-Frame, der vom Server zum Client gesendet wird, beträgt, wenn er mit der einfachen Längenangabe hinkommt, gerade mal 2 Bytes. Wie wir noch sehen werden, müssen WebSocket-Frames, die vom Client auf die Leitung gegeben werden, maskiert werden. Dazu muss der Masking-Key im Header hinterlegt werden. Da dieser 4 Bytes beträgt, ist der kleinste Header, der von einem Client ausgeht, 6 Bytes groß. Da die Länge der Nutzdaten ebenfalls im Header vermerkt wird, kann der Header für WebSocket-Frames, die viele Nutzdaten transportieren, sich entsprechend um 2 bzw. 8 Bytes vergrößern. Die maximale Länge eines Headers beträgt somit 14 Bytes.

Der bit-genaue Aufbau eines WebSocket-Frames ist in [Bild 4.4](#) dargestellt, den wir uns im Folgenden auch bit-genau zu Gemüte führen wollen.

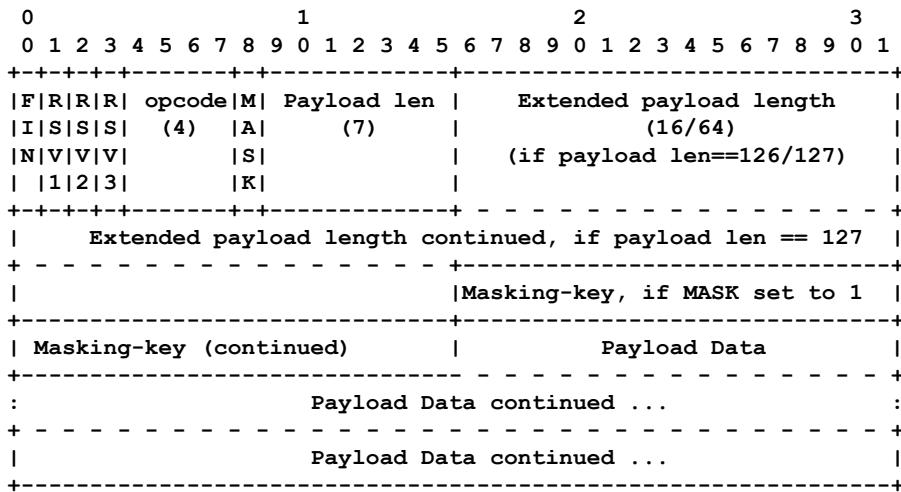


Bild 4.4 Aufbau eines WebSocket-Frames [[FM11a](#)]

Wir wollen uns zunächst kurz die Bedeutung der einzelnen Header-Felder anschauen, bevor wir diese dann in den nachfolgenden Unterkapiteln konkreter beleuchten und bestimmte Ausprägungen von WebSocket-Frames anhand von Beispielen nachvollziehen.

Der Header eines WebSocket-Frames besteht aus folgenden Feldern:

- **FIN (1 Bit)**
Hiermit wird angegeben, ob es sich um einen vollständigen Frame (final) oder ein Fragment handelt. Bei stückchenweiser Übertragung der Nutzdaten in Fragmenten ist das FIN-Flag auf 0 gesetzt. Den Wert 1 nimmt FIN an, wenn es sich um das letzte Fragment handelt. Bei einem einzigen, vollständigen Frame ist das erste Fragment gleichzeitig das letzte und damit FIN auf 1 gesetzt.
- **RSV1, RSV2, RSV3 (jeweils 1 Bit)**
Diese drei Flags sind für zukünftige Verwendungen reserviert und daher in der Regel auf den Wert 0 gesetzt. Erweiterungen, die besondere Signalisierungsmechanismen benötigen, können sich diese mittels dieser Flags definieren. Beim Deflate-Verfahren wird z. B. der RSV1 gesetzt (siehe [Abschnitt 5.8](#)). Falls ein Wert empfangen wird, der nicht 0 ist, und keine Erweiterung für die Bedeutung dieses Wertes definiert ist, muss die WebSocket-Verbindung geschlossen werden.
- **opcode (4 Bit)**
Dieser Headereintrag definiert die Semantik und Interpretation eines WebSocket-Frames. Da WebSockets sowohl Binär- als auch Textdaten übertragen können, wird mittels dieses Opcodes bestimmt, welche Art von Nutzdaten im Frame übertragen werden. Die Frames, die für die eigentliche Datenübertragung zuständig sind, werden auch als *Datenframes* oder *Non-Control-Frames* bezeichnet. Außerdem werden mit dem opcode

noch weitere Frame-Arten unterschieden. Dazu gehören z. B. Frames, die die Beendigung einer WebSocket-Verbindung anstoßen (Close-Frame), oder auch Frames, mit denen der Verbindungsstatus überprüft werden kann (Ping-/Pong-Frames). Da die primäre Funktion dieser Frames in der Kontrolle einer WebSocket-Verbindung liegt, werden diese als Control-Frames bezeichnet. Folgende Opcodes sind im Standard festgelegt:

- 0x0 definiert ein Fortsetzungs-Frame
- 0x1 definiert ein Textframe
- 0x2 definiert einen Binärframe
- 0x3–7 sind reserviert für weitere Non-Control-Frames
- 0x8 definiert ein Connection-Close-Frame
- 0x9 definiert ein Ping-Frame
- 0xA definiert ein Pong-Frame
- 0xB–F sind reserviert für weitere Control-Frames

Wird ein unbekannter Opcode empfangen, muss die WebSocket-Verbindung geschlossen werden.

▪ MASK (1 Bit)

Gibt an, ob die Nutzdaten maskiert sind. Ist dieses Feld auf den Wert 0 gesetzt, ist der Frame maskiert. In diesem Fall ist in einem zusätzlichen Header-Feld ein Masking-Key enthalten, mit dem die Nutzdaten maskiert wurden und mit dem die Maskierung auch wieder entfernt werden kann. Alle Frames, die vom Client zum Server gesendet werden, müssen maskiert werden. Frames, die vom Server zum Client gesendet werden, werden nicht maskiert.

▪ Payload len (7 Bit)

Gibt die Länge der Nutzdaten in Byte an. Mit diesem Feld können Nutzdaten bis zu einer Länge von 125 Bytes versendet werden. Wenn größere Frames versendet werden sollen, kann die Längenangabe entsprechend erweitert werden.

- Payload len == 126

Steht der Wert dieses Feldes auf 126, so bedeutet dies, dass die Längenangabe durch die folgenden 2 Bytes des Extended payload length-Headerfeldes angegeben wird.

- Payload len == 127

Steht der Wert dieses Feldes auf 127, so bedeutet dies, dass die Längenangabe durch die folgenden 8 Bytes des Extended payload length-Headerfeldes angegeben wird.

▪ Extended payload length (16 Bit oder 64 Bit)

Dieser Bereich im Header kommt hinzu, wenn die Nutzdatenlänge 125 Bytes überschreitet. Wie wir in den vorangegangenen Spiegelpunkten gesehen haben, kann mit der einfachen Payload len die Länge dieser Erweiterung beeinflusst werden (entweder 16 oder 64 Bit). Die maximale Länge der Nutzdaten in einem WebSocket-Frame beträgt somit $2^{64} - 1$ Bytes, was sich theoretisch bis in den Exabyte-Bereich erstrecken kann.

▪ Masking-Key (0 oder 32 Bit)

Alle Frames, die vom Client zum Server gesendet werden, müssen mit einem 32-Bit-Wert maskiert werden. Dieser soll vom Client zufällig gewählt werden und im Frame enthalten sein. Für Nachrichten, die vom Client gesendet werden, steht der Masking-Key in diesem

Feld. Für Nachrichten, die vom Server gesendet werden, existiert dieses Feld im Frame nicht.

- **Payload Data (n Byte)**

Die eigentlichen Nutzdaten. Die Anzahl an Bytes ist durch den entsprechenden Längenwert im Header bestimmt. Die Nutzdaten können sich bei der Verwendung von Extension noch in Extension Data und Application Data aufteilen.

Hieraus können Sie bereits ersehen, dass der Overhead durch den Header sehr begrenzt ist. Je nachdem, wie viele Nutzdaten in einem Frame versendet werden sollen und ob der Frame vom Client zum Server oder umgekehrt geschickt wird, beträgt die minimale Größe des Headers 2 Bytes (keine Maskierung und Nutzdatengröße kleiner als 126 Bytes) und die maximale Größe 14 Bytes (mit Maskierung und Nutzdatengröße größer als $2^{16} = 65536$ Bytes).

■ 4.4 Fragmentierung

Der WebSocket-Protokoll-Standard sieht eine stückchenweise Übertragung der Nutzdaten vor. Dies ist insbesondere für Szenarien von Bedeutung, in denen die Nutzdaten zum Zeitpunkt der Übertragung nicht vollständig vorliegen oder nicht komplett gepuffert werden können.

Da in diesen Fällen die vollständigen Daten nicht vorliegen, können diese auch nicht in einen WebSocket-Frame eingebettet werden, geschweige denn die Datenlänge dieser im Header des Frames angegeben werden. Um in diesen Szenarien dennoch Daten versenden zu können, steht die Möglichkeit bereit, die Daten in den vorliegenden Häppchen zu übertragen. Über den entsprechenden Schalter im Header kann signalisiert werden, dass die Daten noch nicht vollständig sind und dass noch Frames kommen.

Im einfachsten Fall stehen alle Daten bereit und können in einem Schwung übertragen werden. Dann ist der „erste“ Frame auch gleichzeitig der „letzte“ Frame. Für die Textnachricht „Hello Galileo!“ ist das z. B. der Fall. Diese kann in einem einzigen WebSocket-Frame erfolgen, der folgenden Aufbau hat (Darstellung der Bytes als hexadezimale Werte):

```
81 0E 48616C6C6F2047616C696C656F21
```

Der für die Fragmentierung wesentliche Teil befindet sich im ersten Byte. Da es sich hierbei um einen vollständigen Frame handelt, ist das FIN-Bit auf den Wert 1 gesetzt. Mit den reservierten Bits RSV1, RSV2 und RSV3 sowie dem 4 Bit großen opcode, der für Textnachrichten als 0x1 definiert ist, ergibt sich das folgende Bitmuster für die dargestellte Nachricht 1000 0001. In hexadezimaler Darstellung entspricht das 0x81.

Wollen Sie die Daten einer großen Datei vom Server zum Client übertragen und Sie lesen diese in Schüben von 4096 Bytes vom Dateisystem ein, dann würde sich auf der Leitung etwas in der folgenden Art abzeichnen:

- Erster Frame: 02 7E 1000 [4096 Byte binäre Daten]
- Zweiter Frame: 02 7E 1000 [4096 Byte binäre Daten]
- Dritter Frame: 02 7E 1000 [4096 Byte binäre Daten]
- ...
- Vorletzter Frame: 02 7E 1000 [4096 Byte binäre Daten]
- Letzter Frame: 82 7E 08E3 [2275 Byte binäre Daten]

Bitte richten Sie Ihre Aufmerksamkeit maßgeblich auf das erste Byte der nach und nach auf die Leitung gegebenen WebSocket-Frames. Solange die Nachrichtenbytes nicht vollständig übertragen wurden, ist das FIN-Bit auf 0 gesetzt. Mit den reservierten Bits und dem opcode für Binärdaten (0x1) ergibt sich das Bitmuster 0000 0010, was in hexadezimaler Darstellung 0x02 entspricht. Das FIN-Bit bleibt bis zum letzten Frame auf 0 gesetzt. Erst wenn das letzte Fragment auf die Leitung gegeben wird, wird das FIN-Bit auf den binären Wert 1 gesetzt, woraus sich dann das Bitmuster 1000 0010 (0x82) ergibt.

■ 4.5 Maskierung

Alle Frames, die vom Client zum Server geschickt werden, müssen aus Sicherheitsgründen maskiert sein (siehe [Abschnitt 7.2.4](#)). WebSocket-Frames vom Server wiederum dürfen nicht maskiert sein. Empfängt ein WebSocket-Client einen maskierten Frame, muss dieser die Verbindung sofort beenden. Ist umgekehrt ein Frame von einem Client unmaskiert bei einem WebSocket-Server eingegangen, muss die Verbindung vom Server ebenfalls geschlossen werden. In beiden Fällen wird von dem RFC 6455 vorgeschlagen, den spezifizierten Fehlercode 1002 protocol error zurückzugeben.

Zur Maskierung muss der Client für jeden Frame eine neue und bis dato noch nicht verwendete 32-Bit-Zufallszahl wählen. Dieser als Masking-Key bezeichnete Wert wird in das entsprechende Headerfeld eingetragen. Anschließend müssen mit dem Masking-Key die Nutzdaten maskiert werden. Den Algorithmus dazu möchten wir Ihnen zunächst mit dem folgenden Codeschnipsel näherbringen:

Listing 4.3 Veranschaulichung des Maskierungsalgorithmus

```

1 byte[] maskingKey; // 4 Bytes zufälliger Masking-Key
2 byte[] payloadData; // Daten, die zum Server übertragen werden sollen
3 byte[] maskedData; // maskierte Daten, die erzeugt werden sollen
4
5 for(int i=0; i<payloadData.length; i++)
    maskedData[i] = payloadData[i] ^ maskingKey[i%4];

```

Wie Sie sehen, wird der Maskierungsschlüssel `maskingKey` zyklisch auf die Nutzdaten `payloadData` angewendet. Dazu werden die Bytes bitweise XOR-verknüpft. Das Ergebnis sind die maskierten Nutzdaten, die im Veranschaulichungscode als `maskedData` bezeichnet sind. Da die XOR-Verknüpfung selbstinvers ist, d. h.

$$\text{payloadData}[i] \wedge \text{maskingKey}[i \% 4] \wedge \text{maskingKey}[i \% 4] = \text{payloadData}[i]$$

gilt, erfolgt das Demaskieren entsprechend umgekehrt, indem der Masking-Key zyklisch in der gleichen Reihenfolge auf die maskierten Daten angewendet wird. Anhand des folgenden Beispiels können Sie dieses Prinzip nachvollziehen. Wir nehmen an, dass der Client die folgende 32-Bit-Zufallszahl als Masking-Key gewählt hat:

37fa213d

Soll der Client nun die Textnachricht „Hello Galileo!“ an den Server schicken, muss die Byte-Repräsentation der Nachricht (hier also die Bytes folgender Textnachricht) bitweise mit dem Masking-Key „xored“ werden (siehe [Bild 4.5](#)):

48616c6c6f2047616c696c656f21

Nachricht	H	e	I	I	o		G	a	I	i	I	e	o	!
Bytes	48	61	6c	6c	6f	20	47	61	6c	69	6c	65	6f	21
Operator	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Key	37	fa	21	3d	37	fa	21	3d	37	fa	21	3d	37	fa
Maskierte Bytes	7f	9b	4d	51	58	da	66	5c	5b	93	4d	a2	58	db

Bild 4.5 Berechnung der maskierten Nutzdaten

Das in [Bild 4.5](#) verwendete Symbol in der Operator-Zeile ist die grafische Repräsentation des XOR-Operators. Beachten Sie auch wie sich der Maskierungsschlüssel in der Key-Zeile periodisch wiederholt, um auf alle Bytes der Nachricht angewendet werden zu können. In der Abbildung können Sie die zyklische Wiederholung der Masking-Key-Bytes an den abwechselnden Grautönen erkennen. Im angegebenen Pseudocode auf der vorangegangenen Seite ist dies durch den Modulo-Operator (%) implementiert. Da es sich beim Masking-Key gemäß dem RFC 6455 immer um 4 Bytes handelt, können die Indizes 0 bis 3 eines entsprechenden byte-Arrays durch den Einsatz des Modulo-Operators auf einen Laufindex angewendet werden:

```
for(int i=0; i<payloadData.length; i++)
    maskedData[i] = payloadData[i] ^ maskingKey[i%4];
```

Die auf diese Weise der Nachricht hinzugefügte Maskierung kann auf der Serverseite entsprechend wieder umgekehrt und damit entfernt werden (siehe [Bild 4.6](#)). Alles, was benötigt wird, um die ursprünglichen Datenbytes zurückzugewinnen, steht im WebSocket-Frame bereit. Dafür bedarf es der maskierten Bytes und des Masking-Keys.

Maskierte Bytes	7f	9b	4d	51	58	da	66	5c	5b	93	4d	a2	58	db
Operator	\oplus													
Key	37	fa	21	3d	37	fa	21	3d	37	fa	21	3d	37	fa
Bytes	48	61	6c	6c	6f	20	47	61	6c	69	6c	65	6f	21
Nachricht	H	e	I	I	o		G	a	I	i	I	e	o	!

Bild 4.6 Rückgewinnung der Nutzdaten aus den maskierten Daten

In [Bild 4.5](#) können Sie schön sehen, wie die selbstinverse XOR-Verknüpfung die maskierten Bytes demaskiert. Wenn jedes maskierte Byte mit dem gleichen Masking-Key-Byte erneut „gexort“ wird, kommt wieder das ursprüngliche unmaskierte Byte zutage. So wird die ursprüngliche Nachricht Schritt für Schritt zurückgewonnen. In einem Java-basierten Code-schnipsel würde die Demaskierung dann entsprechend wie in [Listing 4.4](#) aussehen.

Listing 4.4 Veranschaulichung des Demaskierungsalgorithmus

```

1 byte[] maskingKey; // 4 Bytes zufälliger Masking-Key, dem WebSocket
2 // Frame entnommen
3 byte[] maskedData; // maskierte Daten, dem WebSocket Frame entnommen
4 byte[] payloadData; // demaskierte Daten, die erzeugt werden sollen
5
6 for(int i=0; i<maskedData.length; i++)
    payloadData[i] = maskedData[i] ^ maskingKey[i%4];

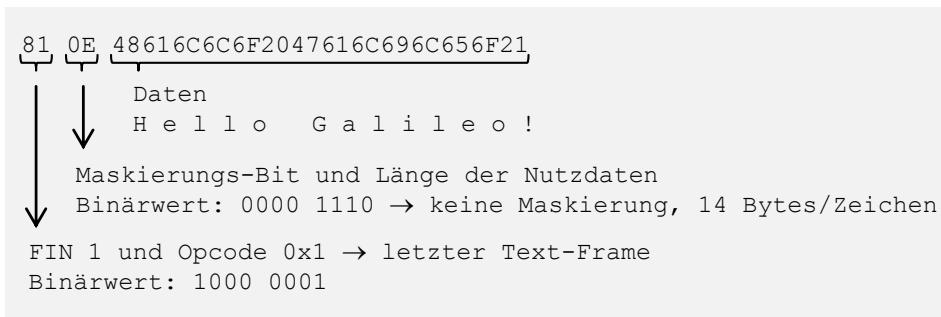
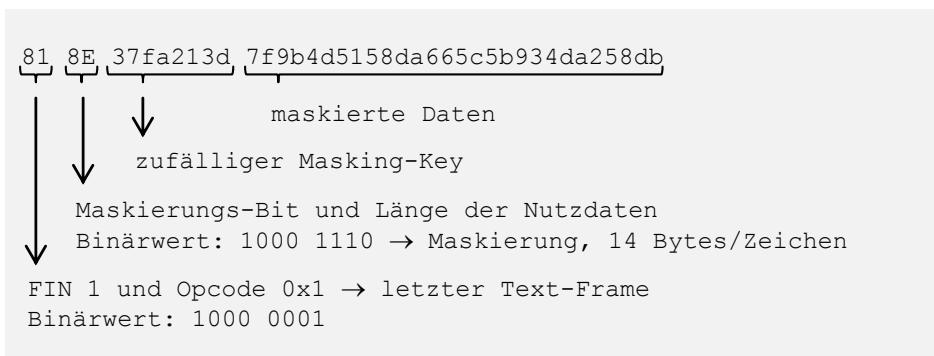
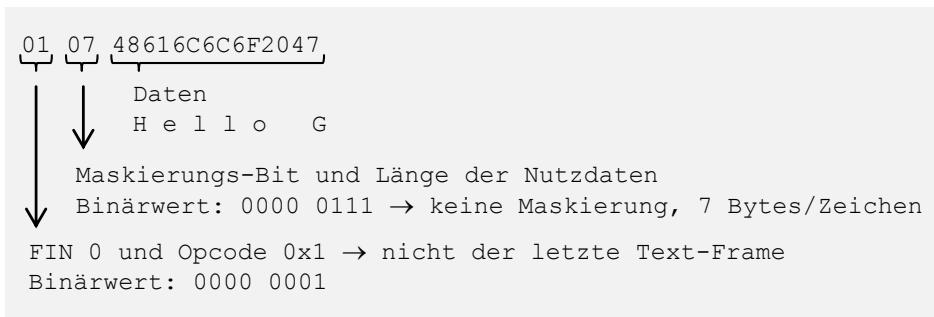
```

■ 4.6 Datenframes

Wie wir bereits angesprochen haben, können WebSockets sowohl textbasierte als auch binäre Daten transportieren.

4.6.1 Frames mit Textdaten

Textdaten-Frames, die mit dem Opcode `0x1` versehen sind, bestehen aus UTF-8-kodierten Zeichen. Um die Funktionsweise verdeutlichen zu können, spielen wir mit Ihnen ein Szenario durch, das die Textnachricht „Hello Galileo!“ in allen möglichen Kombinationen darstellt (siehe [Bild 4.7](#) bis [Bild 4.10](#)).

**Bild 4.7** Server → Client (unmaskiert), eine Nachricht (nicht fragmentiert)**Bild 4.8** Client → Server (maskiert), eine Nachricht (nicht fragmentiert)**Bild 4.9** Server → Client (unmaskiert), zwei Nachrichten (fragmentiert), erster Frame

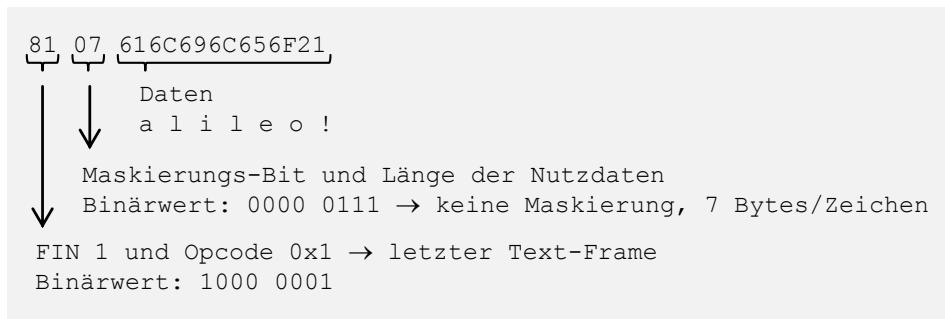


Bild 4.10 Server → Client (unmaskiert), zwei Nachrichten (fragmentiert), zweiter Frame

4.6.2 Frames mit Binärdaten

Binärdaten werden mit dem Opcode 0x2 versehen. Die Variation an Frames, die Sie dabei auf der Leitung beobachten können, hängt davon ab, wie groß die Datenmenge ist und ob sie fragmentiert und/oder maskiert übertragen werden. **Bild 4.11** zeigt ein Beispiel für einen einzelnen zusammenhängenden Frame, der 125 Bytes binäre Nutzdaten enthält.

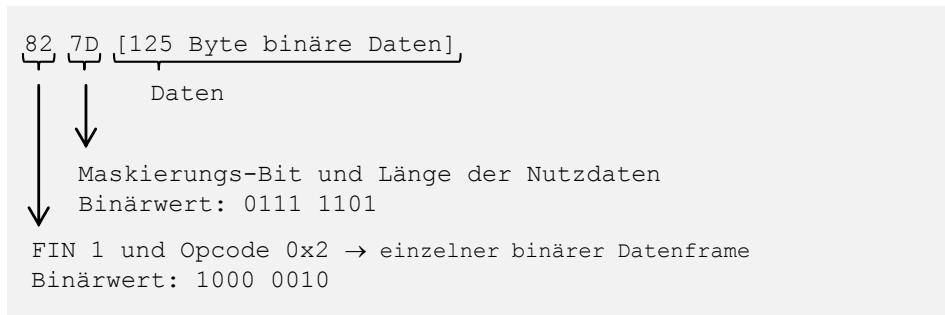


Bild 4.11 125 Bytes Binärdaten in einem vollständigen und unmaskierten Frame

Übersteigt die Anzahl an Bytes die Marke 125, so muss dieses Feld entsprechend erweitert werden, damit die Länge als Zahlenwert darstellbar ist. Das in **Bild 4.12** dargestellte Beispiel zeigt eine solche Codierung, die einen einzelnen WebSocket-Frame mit 255 Bytes binärer Daten repräsentiert.

Übersteigen die Nutzdaten eine Größe von $2^{16} - 1 = 65535$ Bytes, so erlaubt der nächste *Extended payload length* Header eine Größenangabe mit 8 Bytes, womit Nutzdatengrößen bis in den Exabyte-Bereich angegeben und damit in einem einzigen WebSocket-Frame übertragen werden können. Das sollte vorerst ausreichen. Falls nicht, besteht noch die Möglichkeit der fragmentierten Übertragung.

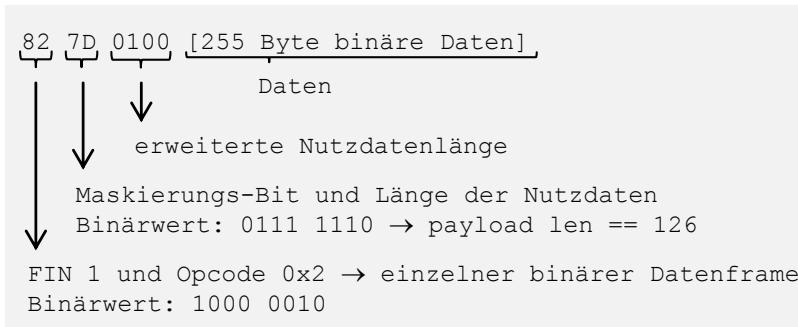


Bild 4.12 255 Bytes Binärdaten in einem vollständigen und unmaskierten Frame

■ 4.7 Control-Frames

Control-Frames werden dazu verwendet, den Status einer WebSocket-Verbindung zu prüfen oder zu beeinflussen. In der Spezifikation sind drei solcher Control-Frames festgelegt worden, die Sie anhand des Opcodes unterscheiden können. Allgemein ist für Control-Frames im RFC 6455 festgelegt worden, dass das höchstwertige Bit des Opcodes immer 1 ist und somit Control-Frames die Opcodes 0x8 bis 0xF für sich vereinnahmen. Eine weitere Gemeinsamkeit der Control-Frames liegt in der Eigenschaft, auch Nutzdaten mit sich führen zu können. Im Gegensatz zu den Datenframes sind diese aber bis auf wenige Ausnahmen nur Sequenzen von UTF-8-kodierten Zeichen.

Die folgende Auflistung zeigt die gegenwärtig im RFC 6455 spezifizierten Opcodes:

- 0x8 Close-Frame
- 0x9 Ping-Frame
- 0xA Pong-Frame
- 0xB–0xF Reserviert für zukünftige Control-Frames

4.7.1 Ping-Frame

Ein Ping-Frame kann genutzt werden, um zu prüfen, ob eine etablierte WebSocket-Verbindung noch besteht und ansprechbar ist. Außerdem können Pings dazu verwendet werden, die Latenz zwischen Client und Server zu messen. Wird ein Endpunkt mit einem Ping angesprochen, muss dieser so schnell wie möglich mit einem Pong-Frame antworten. Mit Pong-Frames befassen wir uns direkt im Anschluss an dieses Kapitel.

Ein Ping-Frame hat den Opcode 0x9. Typischerweise enthält ein Ping-Frame keine Nutzdaten, kann aber laut der Spezifikation solche enthalten. Wenn der Ping Nutzdaten enthält, müssen diese vom korrespondierenden Pong zurückgesendet werden. Enthält der Ping also z. B. einen Zeitstempel, so ist dieser auch im Pong enthalten, wodurch Sie Laufzeiten für einen Nachrichtenumlauf ermitteln können. Ein Ping-Frame, der keine Nutzdaten enthält

und vom Server zum Client gesendet wird, ist somit 2 Bytes groß und hat die in [Bild 4.13](#) dargestellte Anatomie.

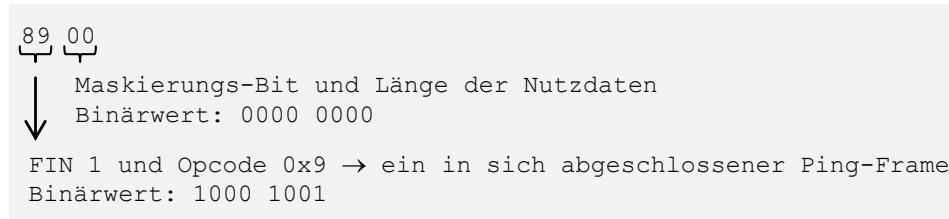


Bild 4.13 Ping-Frame ohne Nutzdaten und ohne Maskierung

Derzeit gibt es noch keine Schnittstelle, um Ping-Frames selbst zu versenden. Diese werden von den verschiedenen WebSocket-Implementierungen aktuell als rein interner Mechanismus zur Verwaltung offener WebSocket-Verbindungen – als eine Art Heartbeat – gehandhabt. Die eingangs erwähnten Latenzmessungen sind daher über diesen Mechanismus noch nicht umsetzbar.

4.7.2 Pong-Frame

Der Pong ist die Antwort auf einen Ping. In einem Pong-Frame müssen die gleichen Nutzdaten enthalten sein wie im empfangenen Ping-Frame, auf den der Pong erwidert werden soll, sofern im Ping überhaupt Nutzdaten enthalten waren. Ein Pong-Frame hat den Opcode 0xA. Ein Pong, der keine Nutzdaten enthält und vom Server zum Client gesendet wird, ist genau wie sein vorangegangener Ping 2 Bytes groß. Den Aufbau eines derartigen Pongs können Sie [Bild 4.14](#) entnehmen.

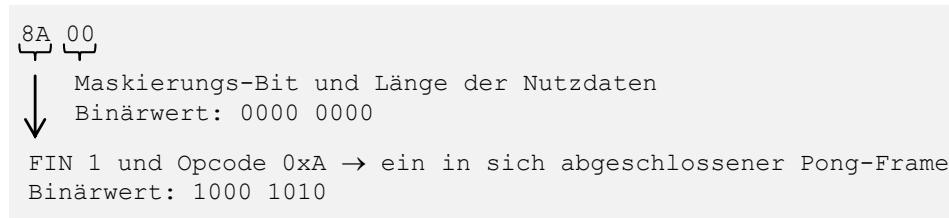


Bild 4.14 Pong-Frame ohne Nutzdaten und ohne Maskierung

Empfängt ein Endpunkt mehrere Pings hintereinander und hatte noch keine Zeit, auf diese zu antworten, so muss er nur den letzten Ping beantworten.

4.7.3 Close-Frame

Der Close-Frame wird, wie der Name schon sagt, benutzt, um eine bestehende WebSocket-Verbindung zu schließen. Empfängt ein Endpoint – Client oder Server – einen Close-Frame, muss dieser darauf ebenfalls mit einem eigenen Close-Frame antworten und alles Weitere

unternehmen, um die betreffende Verbindung zu schließen. Wird ein Close-Frame versendet, dürfen von diesem Zeitpunkt an beide Seiten, sowohl Server als auch Client, keine weiteren Daten mehr verschicken bzw. zur Weiterverarbeitung entgegennehmen.

Close-Frames vom Client zum Server müssen auch, wie wir das bereits in [Abschnitt 4.5](#) für Datenframes gesehen haben, maskiert sein. Ein Close-Frame, der keine Nutzdaten enthält und vom Server zum Client gesendet wird, ist 2 Byte groß. Den Aufbau können Sie [Bild 4.15](#) entnehmen.

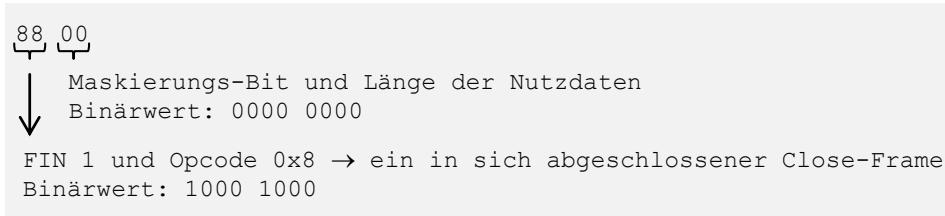


Bild 4.15 Close-Frame ohne Nutzerdaten und ohne Maskierung

Ein Close-Frame kann auch Nutzdaten in Form eines UTF-8-Strings enthalten. Diese Nutzdaten können einen Statuscode und einen Grund für das Beenden der Verbindung beinhalten. Wird eine Verbindung normal geschlossen, ist laut Spezifikation der Statuscode 1000 Normal Closure zu verwenden. Ein solcher Close-Frame hat dann den folgenden Aufbau (unmaskiert in hexadzimaler Darstellung):

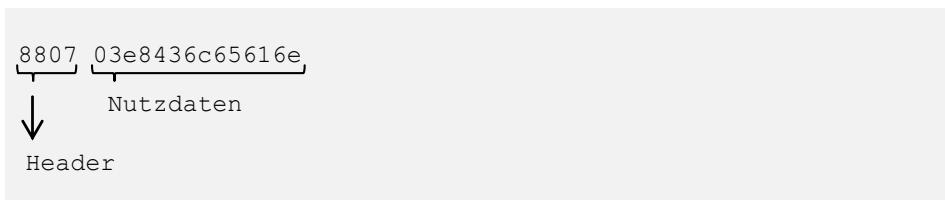


Bild 4.16 Close-Frame ohne Maskierung in hexadzimaler Darstellung

Die sieben Bytes des Payloads enthalten den Statuscode und die dazu passende textuelle Beschreibung. Der Statuscode setzt sich immer aus den ersten beiden Bytes zusammen, die einen vorzeichenlosen Integer codieren, im Beispiel also 0x03e8, was dem Integer-Wert 1000 entspricht. Die nachfolgenden Bytes enthalten die UTF-8-codierten Zeichen der textuellen Begründung; im angegebenen Beispiel also 0x436c65616e, was dem String „Clean“ entspricht. Das hier abgedruckte Beispiel ist vom Chrome-Browser abgesetzt worden und wurde von uns demaskiert.

[Tabelle 4.1](#) gibt die im RFC 6455 definierten Statuscodes und ihre Bedeutung wieder.

Tabelle 4.1 Statuscodes für Close-Frames

Statuscode	Bedeutung	Beschreibung
1000	Normal Closure	Dieser Statuscode zeigt einen ordnungsgemäßen Verbindungsabbau an.
1001	Going Away	Gibt Ihnen Auskunft darüber, dass ein Endpunkt „weg gegangen“ ist. Als Beispiele können Sie hierfür das Herunterfahren des Servers oder das Verlassen der Webseite heranziehen.
1002	Protocol Error	Dieser Statuscode signalisiert Ihnen, dass ein Endpunkt die Verbindung aufgrund eines Protokollfehlers abgebrochen hat.
1003	Unsupported Data	Aus diesem Statuscode können Sie ablesen, dass ein Endpunkt die Verbindung geschlossen hat, weil dieser mit den empfangenen Daten nichts anfangen konnte. Dies kann z. B. dann auftreten, wenn ein Endpunkt, der nur Textdaten verarbeiten kann, einen Binärframe erhält und umgekehrt (erwartet Binärframe und erhält Textframe).
1004	Reserviert	Dieser Statuscode ist für eine eventuell zukünftige Verwendung vorgesehen und bis dahin ohne spezifische Bedeutung belegt.
1005	No Status Rcvd	Für Anwendungen reserviert, die anzeigen können, dass kein Statuscode im empfangenen Close-Frame enthalten war. Dieser Statuscode wird nicht von der Protokollimplementierung gesetzt.
1006	Abnormal Closure	Wird wie der Statuscode 1005 nicht von der Protokollimplementierung gesetzt, sondern ist für Anwendungen vorgesehen, die damit signalisieren können, dass die Verbindung auf „ungewöhnliche“ Art und Weise unterbrochen worden ist, z. B. ohne dass ein Close-Frame gesendet oder empfangen wurde.
1007	Invalid Frame Payload Data	Dieser Statuscode wird gesetzt, wenn die empfangenen Daten nicht mit dem Nachrichtentyp übereinstimmen. Dies ist z. B. der Fall, wenn ein Textframe Bytes enthält, die nicht UTF-8 [Yer03] konform sind.
1008	Policy Violation	Dieser sehr universelle Statuscode wird immer dann verwendet (evtl. auch von Ihnen), wenn kein passenderer Code wie z. B. 1003 oder 1009 zur Anwendung kommen kann.
1009	Message Too Big	Gibt an, dass ein Endpunkt die WebSocket-Verbindung schließt, weil dieser eine Nachricht geschickt bekommen hat, die aufgrund der großen Datenmenge nicht verarbeitet werden kann.

Ihnen wir aufgefallen sein, dass die Statuscodes sich allesamt um den ersten Statuscode (der 1000) tummeln. Tatsächlich definiert der RFC 6455 Statuscode-Gruppen, die sich wie folgt gliedern:

Statuscode	Bedeutung	Beschreibung
1010	Mandatory Ext.	Mit diesem Statuscode signalisiert ein WebSocket-Client, dass die Verbindung geschlossen wird, weil eine oder mehrere Erweiterungen, die vom Client im Opening-Handshake angegeben wurden, fehlen. Der Server braucht von diesem Statuscode keinen Gebrauch zu machen, da er auf einen Erweiterungsmissstand durch einen abgebrochenen Opening-Handshake reagieren kann.
1011	Internal Server Error	Reine Serversache ist dieser Statuscode, mit dem der Server mitteilen kann, dass sich bei der Verarbeitung ein unerwarteter Zustand eingestellt hat, der die ordnungsgemäße Bearbeitung der Anfrage unterbindet.
1015	TLS Handshake	Dieser Statuscode ist der Anwendung überlassen. Die Anwendung kann damit Validierungsfehler der im TLS-Handshake ausgetauschten TLS-Zertifikate anzeigen (z. B. wenn das Server-Zertifikat nicht validiert werden konnte; in Abschnitt 7.2.2 schauen wir uns TLS genauer an).

- 0–999

Die Statuscodes im Bereich von 0–999 sind nicht in Gebrauch.

- 1000–2999

Statuscodes im Bereich von 1000–2999 sind für die Festlegungen des Protokolls im RFC 6455 und zukünftige Weiterentwicklungen reserviert sowie für standardisierte offene Erweiterungen.

- 3000–3999

Bibliotheken, Frameworks und Anwendungen können (idealerweise allgemeingültige) Codes im Bereich von 3000–3999 definieren. Diese werden der IANA zur Standardisierung vorgelegt.

- 4000–4999

Statuscodes, die für proprietäre Belange benötigt werden, können im Bereich von 4000–4999 liegen.

Ist eine Verbindung durch aufgetretene Widrigkeiten unerwartet getrennt worden, d. h., einer der beteiligten Endpunkte (entweder Server oder Client) hat keinen Close-Frame erhalten, so wird der Statuscode 1006 Abnormal closure zurückgegeben. Eine von den Kommunikationspartnern herbeigeführte Trennung kann also z. B. folgenden Statuscode und verbalen Grund haben:

Code: 1000

Reason: Session Timeout

Ein unerwartet eingetroffener Verbindungsabbruch liefert hingegen z. B. den folgenden Statuscode mit dazugehöriger verbaler Beschreibung:

Code: 1006

Reason: Connection to server lost

Die Zuordnung der Close-Codes ist im RFC 6455 nur als Vorschlag enthalten und muss folglich nicht zwingend eingehalten werden. Entwickler von WebSocket-Implementierungen können daher selber entscheiden, welchen Code sie für welches Szenario verwenden. Hieraus ist bereits in der Praxis ein gewisser Wildwuchs entstanden, der unter Umständen zu Widrigkeiten führen kann.

■ 4.8 Closing-Handshake – WebSocket-Verbindung schließen

Ein Closing-Handshake wird durch einen Close-Frame initiiert. Dieser kann sowohl vom Client als auch vom Server abgesetzt werden. Der somit gestartete Closing-Handshake ist erst dann abgeschlossen, wenn beide Parteien jeweils einen Close-Frame gesendet und empfangen haben.

Ein Closing-Handshake, der vom Client initiiert wird, ist in [Bild 4.17](#) dargestellt. Wenn ein Endpunkt einen Close-Frame empfangen und dieser selbst zuvor keinen gesendet hat, muss ein Close-Frame erwidert werden. Wenn ein Close-Frame in Folge eines empfangenen Close-Frames gesendet wird, ist im zweiten Frame der Statuscode des ersten Frames enthalten. Die textuelle Begründung wird im zweiten Close-Frame weggelassen.

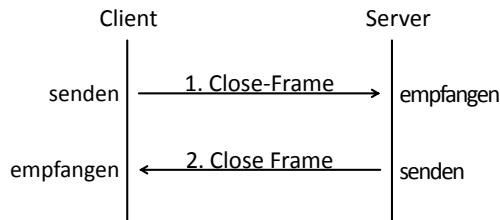


Bild 4.17 Closing-Handshake initiiert durch den Client

Die Abfolge des Closing-Handshakes, der vom Server ausgeht, ist dementsprechend genau umgekehrt und entspricht der in [Bild 4.18](#) dargestellten Sequenz.

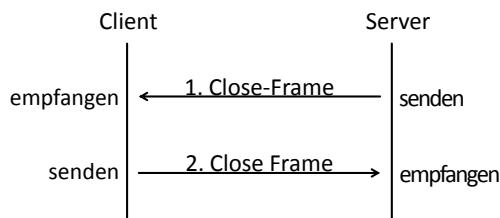


Bild 4.18 Closing Handshake initiiert durch den Server

Nachdem beide Seiten des WebSockets jeweils einen Close-Frame gesendet und empfangen haben, gilt die Verbindung als beendet und der darunter liegende TCP-Kanal wird ge-

schlossen. Dies sollte, unabhängig von wem der erste Close-Frame abgesetzt wurde, vom Server initiiert werden. Die Gründe hierfür liegen innerhalb des TCP. Wenn der Server die Verbindung schließt, können auch im Fehlerfall ohne Verzögerungen neue Verbindungen zwischen dem Client und dem Server aufgebaut werden.

Geht einer der Close-Frames beim Schließen der WebSocket-Verbindung verloren, ist die Verbindung logisch zwar geschlossen, allerdings nur einseitig. Da nach dem Versenden eines Close-Frames vom jeweiligen Endpunkt keine weiteren Daten mehr versendet und angenommen werden, ist dieser Endpunkt nicht mehr Teil der Kommunikationsbeziehung. Durch den unvollständigen Close-Handshake ist einer der Beteiligten allerdings noch im Glauben, dass die Verbindung besteht. Je nachdem, wer den ersten Close-Frame gesendet hat (Client oder Server) und welcher Frame verloren gegangen ist, wird die Verbindung erst nach einem *Timeout* geschlossen. Dies führt dann unter Umständen dazu, dass für den noch in der Kommunikation verbliebenen Teilnehmer abrupt der TCP-Kanal geschlossen wird.

■ 4.9 Tools zur Protokollanalyse

Eine genaue Analyse des Protokolls kann mittels entsprechender Toolunterstützung erfolgen. In diesem Kapitel wollen wir Ihnen einige dieser Werkzeuge vorstellen, die wir für die Analyse von WebSockets für geeignet und nützlich erachten. Wir haben dabei darauf geachtet, jeweils ein Analysetool aus der Gruppe der allgemeinen Netzwerkanalysatoren (Wireshark, siehe [Abschnitt 4.9.1](#)), eines aus der Gruppe der proxy-basierten Tools (Fiddler, siehe [Abschnitt 4.9.2](#)) und eines aus der Gruppe der direkt im Browser integrierten Protokolltools (Chrome Developer Tools, siehe [Abschnitt 4.9.3](#)) zu wählen.

4.9.1 Wireshark

Wireshark ist eine freie Netzwerkmonitoranwendung zur Analyse von Netzwerkaktivitäten. Dadurch haben Sie die Möglichkeit, alle Netzwerkpakete, die durch Ihr lokales Netzwerk übertragen werden, auszulesen und zu speichern. Wireshark ist quelloffen und steht unter der GNU GPL-Lizenz. Diese Software wird u. a. von Angreifern benutzt, um Passwörter und andere sensible Daten zu erspähen. Deswegen ist dieses Tool sehr umstritten und sollte nur in genehmigten oder eigenen Netzwerken genutzt werden. Für Administratoren und Entwickler kann dieses Tool aber sehr hilfreich sein, um z. B. Fehlerquellen und Sicherheitslücken zu ermitteln. Deshalb lernen Sie Wireshark in diesem Kapitel näher kennen. Wir zeigen Ihnen, wie man mit Wireshark aufzeichnet, bestimmte Netzwerkpakete filtert und diese auswertet. Selbstverständlich wird Ihnen auch nähergebracht, wie Sie WebSockets mithilfe dieser Anwendung analysieren können.

Installation von Wireshark

Wireshark ist für Windows, Mac OS, Linux, Solaris und viele andere Betriebssysteme verfügbar.¹ Für Linux-basierende Betriebssysteme können Sie Wireshark in der Regel auch über die jeweiligen Paketmanager beziehen.

Starten der Protokollierung

Unter der Menüauswahl *Interface List* werden alle Netzwerkschnittstellen Ihres Rechners aufgelistet. Ein Klick darauf und eine Liste vorhandener Schnittstellen erscheint (siehe Bild 4.19).

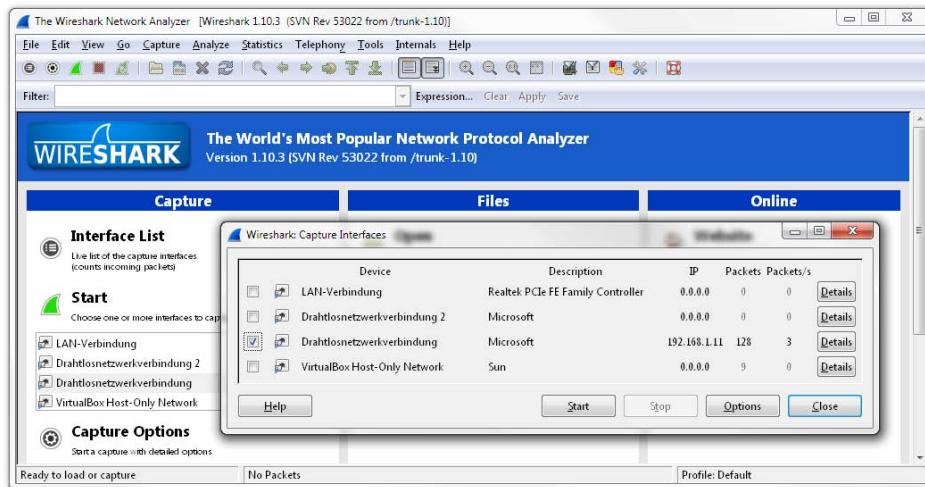


Bild 4.19 Wireshark Netzwerk-Interface-Liste

Nun können Sie eine Netzwerkschnittstelle auswählen, die Sie analysieren möchten. Aktivieren Sie dafür die entsprechende Checkbox. Wireshark beginnt mit der Aufzeichnung des Netzwerkverkehrs, sobald Sie auf den *Start*-Button klicken. Nun öffnet sich das Hauptfenster und die Aufzeichnung Ihrer Pakete startet. Das Hauptfenster besteht aus drei Teilen, der Paketliste, den Paketdetails und der hexadezimalen Paketanzeige (siehe Bild 4.20).

¹ <https://www.wireshark.org/download.html>

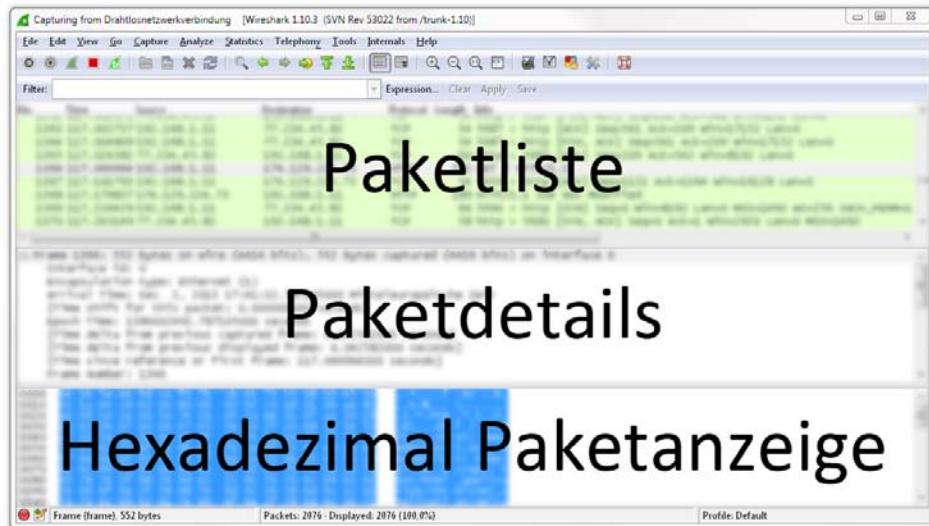


Bild 4.20 Wireshark-Hauptfenster

Im Paketlisten-Fenster werden alle aufgezeichneten Pakete aufgelistet. Wenn Ihre bereits installierten Programme übers Netzwerk kommunizieren, können sehr viele Daten auf einmal zusammen kommen, sodass Sie schnell den Überblick verlieren können. Wireshark bietet Ihnen für diesen Fall die Möglichkeit, nach bestimmten Paketen zu filtern. Oben links in der Toolbar befindet sich ein Formularfeld mit dem Label *Filter*. Dort können Sie ein Filterargument eingeben, das Wireshark veranlasst, nur sich darauf beziehende Pakete zu protokollieren. Wenn Sie z. B. `ip.src==192.168.2.1` eingeben und *Enter* drücken, werden alle Pakete in den Analyselog aufgenommen, die von der Quelle (*engl. Source*), abgekürzt als `src`) von der IP 192.168.2.1 stammen. Sie können auch mehrere Filter auf einmal eingeben, indem Sie mit dem `&&`-Operator zwei Filter verknüpfen. Der Filter `ip.src==192.168.2.1 && tcp` z. B. zeichnet Ihnen alle Pakete auf, die die Quell-IP 192.168.2.1 besitzen und mit dem TCP-Protokoll kommunizieren. Ein logischer Oder-Operator (`||`) steht Ihnen in entsprechender Art und Weise ebenfalls zur Verfügung. Weitere Beispiele und Unterstützung bei der Formulierung von Filterausdrücken bekommen Sie, wenn Sie auf den *Expression*-Button rechts neben dem Filtertextfeld klicken.

Wenn Sie eines der aufgezeichneten Pakete nun näher analysieren möchten, können Sie in dieses „eintauchen“, indem Sie darauf klicken. Daraufhin werden Ihnen die Paketinformationen im Paketdetails-Fenster angezeigt (siehe Bild 4.21).

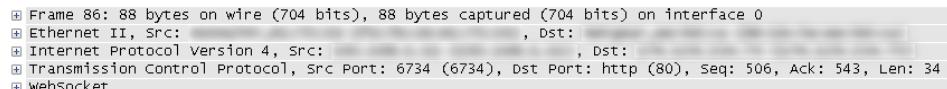
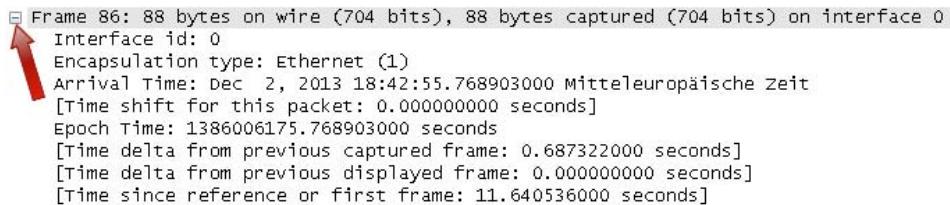


Bild 4.21 Paketdetails-Fenster

Durch das Klicken auf das Dreieckssymbol (bei Windows Plussymbol) am linken Rand einer jeden Zeile erweitert sich jeweils die Informationsansicht. An erster Stelle der Paket-

details sehen Sie die Information des gesamten Pakets bzw. Frames. Dort kann z. B. die Framegröße oder auch die Zeitdifferenz zwischen den vorhergehenden Frames betrachtet werden (siehe Bild 4.22).



```

Frame 86: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
Interface id: 0
Encapsulation type: Ethernet (1)
Arrival Time: Dec 2, 2013 18:42:55.768903000 Mitteleuropäische Zeit
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1386006175.768903000 seconds
[Time delta from previous captured frame: 0.687322000 seconds]
[Time delta from previous displayed frame: 0.000000000 seconds]
[Time since reference or first frame: 11.640536000 seconds]

```

Bild 4.22 WebSocket-Frame im Detail

Die Paketdetails geben Informationen über die jeweiligen ISO/OSI-Layers (siehe Abschnitt 2.1.1) an, die Sie wiederum alle durch das Dreieckssymbol erweitern können. Unter dem Punkt Ethernet II können Sie weitere Details über dieses Protokoll des Layers 2 (Data-Link) erfahren. Weitere Informationen über andere Protokolle in höheren Layern wie z. B. IP, TCP oder auch HTTP finden Sie weiter unten.

Der unterste Teil des Hauptfensters in Bild 4.20 ist die Hexadezimalanzeige der Frames. Wie Sie vielleicht schon gesehen haben, werden die angeklickten Punkte im Paketdetails-Fenster der Hexadezimaldarstellung blau hervorgehoben.

In Bild 4.23 sehen Sie einen Frame in Rohform. Diese Darstellung wird auch als Hexdump bezeichnet, da die einzelnen Bytes des Pakets in hexadezimaler Darstellung angegeben sind. Links sehen Sie die Positionsangabe oder auch die Adresse der Daten. In der Mitte werden die Daten als hexadezimale Werte dargestellt. Rechts werden die Daten in ASCII-Zeichen dargestellt. Nicht darstellbare Zeichen werden in der Ansicht mit einem Punkt (.) ersetzt.

0000	00 4a 73 74 40 00 80 06 36 bb	08 00 45 00	.Jst@... 6. .E.
0010	1a 4e 00 50 b1 e8 f7 97 c9 a2 6e a5 50 18		.N.P.n.P.
0020	00 42 c7 a0 00 00 81 9c 8a 08 8e 92 d8 67 ed f9		.B..... .g..
0030	aa 61 fa b2 fd 61 fa fa aa 40 da df c6 3d ae c5		.a....a.. .@...=..
0040	ef 6a dd fd e9 63 eb e6		.j....c..

Bild 4.23 WebSocket-Frame in Hexdump bzw. ASCII-Dump

Wenn Sie in der Hexadezimalanzeige eine bestimmte Stelle markieren, wird diese in der Paketdetailanzeige grau markiert. Mit *Rechtsklick* öffnet sich ein Kontextmenü, in dem Sie zwischen Binär- und Hexadezimalanzeige wechseln können.

Filtern und Analysieren von WebSocket-Nachrichten

Wireshark bietet auch die Möglichkeit, WebSockets zu analysieren. Dazu müssen Sie als Erstes eine WebSocket-Verbindung aufbauen. Eine WebSocket-Demo-Anwendung können Sie auf der Webseite [WebSocket.org²](http://www.websocket.org/echo.html) starten (siehe Bild 4.24).

² <http://www.websocket.org/echo.html>

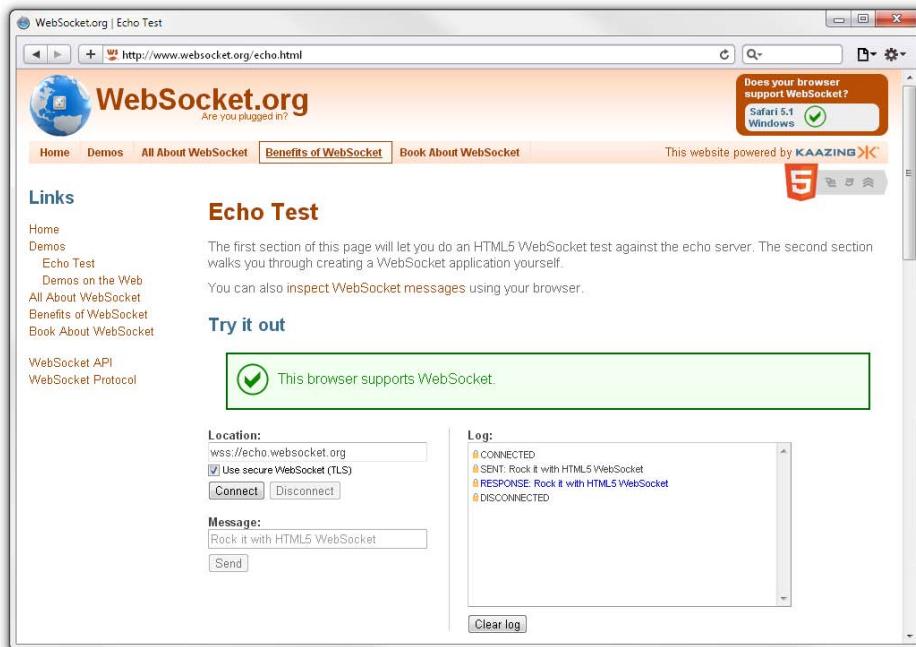


Bild 4.24 WebSocket-Anwendung auf WebSocket.org

WebSocket.org bietet eine eigene Demo-Anwendung an und enthält eine Liste mit weiteren im Web verfügbaren Demos. Die eigene Demo ist ein einfacher *Echo Test*, der Client und Server bereitstellt. Der Client ist auf der HTML-Seite eingebettet und zeigt Ihnen auch an, ob Ihr Browser WebSocket-fähig ist oder nicht. Der Serveranteil der Demo ist auf demselben Rechner installiert, der auch die Website beherbergt. Wie Sie derartige Client-Server-Anwendungen auf Basis des WebSocket-Protokolls und der WebSocket-API implementieren können, ist Gegenstand der nachfolgenden [Kapitel 5, 6](#) und [8](#), wobei wir in [Kapitel 8](#) mit Ihnen deutlich umfangreichere und nützlichere Anwendungen realisieren werden.

Sie können in der Demo von WebSocket.org eine WebSocket-Verbindung vom Webbrowsers zu einem Echo-Server aufbauen. Bevor Sie dies tun, öffnen Sie Wireshark und starten die Protokollaufzeichnung auf der entsprechenden Netzwerkschnittstelle. Danach kehren Sie auf die Webseite des Echo-Test-Clients, die in [Bild 4.24](#) dargestellt ist, zurück. Hier klicken Sie zuerst auf den Button *Connect* unter dem *Location*-Formularfeld mit dem Eintrag *wss://echo.websocket.org*. Konnte die Verbindung zum WebSocket-Server hergestellt werden, können Sie als Nächstes in dem weiter unten platzierten Textfeld eine Nachricht eingeben und diese zum Server schicken. Ein vordefinierter Text ist hier bereits enthalten (*Rock it with HTML5 WebSocket*). Durch das Aktivieren des *Send*-Buttons wird die Nachricht abgeschickt. Im rechten Fenster erscheint daraufhin die Antwort des Servers.

Wenn Sie diese WebSocket-Kommunikation vollzogen haben, kehren Sie zurück in die Wireshark-Anwendung und stoppen zunächst die Protokollaufzeichnung. Anschließend filtern Sie alle relevanten Pakete aus der Liste aus, indem Sie im Filter-Textfeld durch die Angabe *websocket* die WebSocket-Pakete herauspicken. Jetzt werden nur noch die Pa-

kete angezeigt, die auf den Filter passen, was in unserem Kontext alle aufgezeichneten WebSocket-Frames sind (siehe Bild 4.25).

Filter: websocket						Expression...	Clear	Apply	Save
No.	Time	Source	Destination	Protocol	Length	Info			
86	11.6405360	192.168.1.11	174.129.224.73	websocket	88	websocket Text [FIN] [MASKED]			
87	11.7455480	174.129.224.73	192.168.1.11	websocket	84	websocket Text [FIN]			
95	15.5484470	192.168.1.11	174.129.224.73	websocket	60	websocket Connection Close [FIN] [MASKED]			
96	15.6607860	174.129.224.73	192.168.1.11	websocket	60	websocket Connection Close [FIN]			

Bild 4.25 Herausgefilterte WebSocket-Frames mit Wireshark

Klicken Sie nun auf einen dieser WebSocket-Frames, können Sie sich weitere Details dazu anzeigen lassen. Mit dem Dreieckssymbol können Sie bis zu den einzelnen Bits und Bytes vorstoßen. Angenehm ist dabei, dass Wireshark die Bits des Headers gemäß der Festlegungen des RFC 6455 annotiert, was das Lesen der binären WebSocket-Frames deutlich erleichtert.

Der in Bild 4.26 dargestellte WebSocket-Frame zeigt, wie Wireshark die Header-Informationen und die Nutzdaten anzeigen. Sie finden hier einige der beschriebenen Felder aus Abschnitt 4.3 wieder.

WebSocket
1.... = Fin: True
.000 = Reserved: 0x00
.... 0001 = Opcode: Text (1)
0.... = Mask: False
.001 1100 = Payload length: 28
Payload
Text: Rock it with HTML5 websocket

Bild 4.26 WebSocket-Textframe im Detail

Ganz oben sehen Sie, dass das erste Bit im Header gesetzt wurde. Dies bedeutet, dass es sich um einen vollständigen Frame handelt und keine Fragmentierung vorliegt. Eine Zeile darunter werden die reservierten RSV-Flags angezeigt, die alle drei nicht gesetzt sind, was bedeutet, dass keine Extension auf diesen Frame angewendet wurde. Wiederum eine Zeile weiter unten sind die vier Bits des Opcodes auf 0001 (0x1 in Hexadezimal) gesetzt und deuten auf einen Textframe hin. Das Mask-Bit ist nicht gesetzt, somit ist auch kein Masking-Key vorhanden. Hieraus können Sie schließen, dass es sich hierbei um den Frame handelt, den der Server auf die Echo-Anfrage des Clients erwidert hat. Die Nutzdaten sind 28 Bytes groß, was bei einem Textframe 28 UTF8-Zeichen gleichkommt.

In der Hexadezimalanzeige können Sie den WebSocket-Frame auch im Rohformat untersuchen (siehe Bild 4.27).

0000	00 46 22 3b 40 00 28 06 df d8 ae 81 e0 49 c0 a8	08 00 45 20 ..E
0010	01 0b 00 50 1a 4e c9 a2 6e a5 b1 e8 f7 b9 50 18 .F";@. (.I..	
0020	00 89 47 f8 00 00 81 1c 52 6f 63 6b 20 69 74 20 ...P.N.. n.....P.	
0030	77 69 74 68 20 48 54 4d 4c 35 20 57 65 62 53 6f ..G..... Rock it	
0040	63 6b 65 74 with HTM L5 webso	
0050	cket	

Bild 4.27 WebSocket-Textframe als Hexdump

Neben der Analyse eines einzigen WebSocket-Frames erlaubt es Ihnen Wireshark auch, die gesamte Kommunikation, die über einen TCP-Kanal stattgefunden hat, nachzuvollziehen. Hierzu müssen alle TCP-Pakete gesammelt und in der richtigen Reihenfolge zusammengestellt werden. Dies können Sie durch einen Rechtsklick auf eines der Pakete im Paketlisten-Fenster erreichen, die einer TCP-Verbindung angehören, für die Sie sich interessieren. In dem sich öffnenden Kontextmenü wählen Sie den Eintrag *Follow TCP Stream*. Nun sollte die in [Bild 4.28](#) dargestellte Ansicht Ihrer WebSocket-Verbindung angezeigt werden.

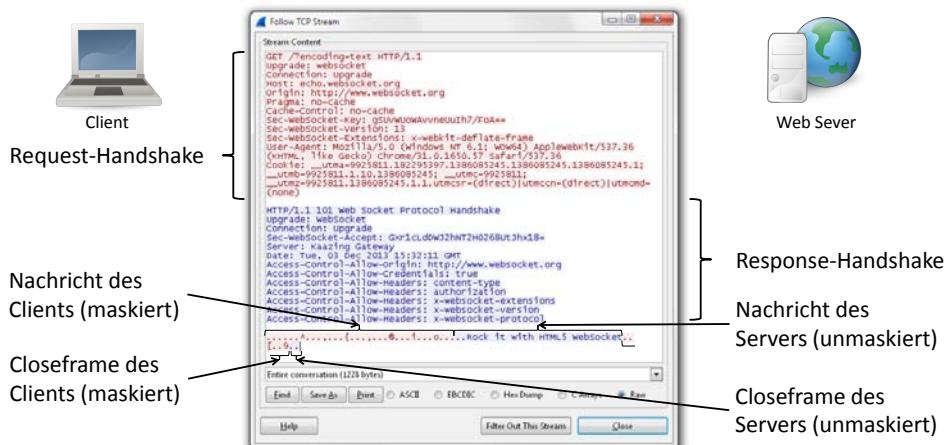


Bild 4.28 Vollständiger Verlauf einer WebSocket-Verbindung

In diesem Fenster können Sie den vollständigen TCP-Stream der WebSocket-Verbindung bzw. Verlauf nachvollziehen (siehe [Bild 4.28](#)). Im oberen Teil können Sie den Opening-Handshake erkennen. Dieser fängt mit einem HTTP-GET-Request an, gefolgt von einer HTTP-Response mit einem 101 Switching Protocols. Daraufhin wird ein Datenframe vom Client abgesendet, der maskiert sein muss. Da es sich hierbei um die Echo-Test-Demo handelt, antwortet der Server mit einer unmaskierten Nachricht, die denselben Inhalt hat.

4.9.2 Fiddler

Fiddler ist ein Web-Proxy-Debugging-Tool. Dieses schaltet sich zwischen Client und Server und kann daher den Datenverkehr protokollieren oder auch manipulieren. Ihr Browser verbindet sich dann über den Proxy-Server von Fiddler ins Internet (siehe [Bild 4.29](#)).

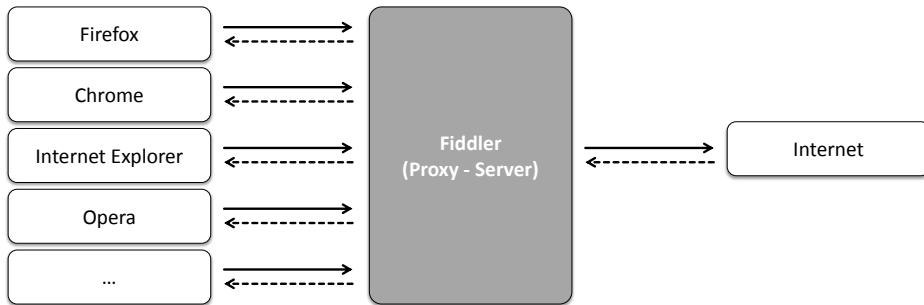


Bild 4.29 Funktionsprinzip von Fiddler

Installation und Einrichtung

Fiddler ist kostenlos und kann von der Telerik-Seite heruntergeladen werden.³ Die Proxy-Anwendung setzt auf das .NET Framework 2.0 und höher auf.

Für Chrome und den Internet Explorer sind keine weiteren Einstellungen unter Windows notwendig, da sie die Proxy-Einstellungen des Systems benutzen. Beim Starten dieser beiden Browser schaltet sich Fiddler also automatisch als Proxy dazwischen.

Möchten Sie den Firefox nutzen, müssen Sie Fiddler als Proxy manuell einstellen. Dazu klicken Sie oben links auf den orangenen Firefox-Button (siehe Bild 4.30).



Bild 4.30 Oranger Firefox-Button



Achten Sie darauf, dass keine Proxy-Add-ons, wie z. B. ProxyMate, in Ihren Browsern aktiviert sind. Diese Erweiterungen erstellen eigene Proxys. Fiddler wird dann nicht mehr dazwischen geschaltet. Um Netzanalysen mit Fiddler durchführen zu können, müssen Sie deshalb evtl. vorhandene Add-ons, die in diese Kategorie fallen, temporärer deaktivieren. Ein Deaktivieren für die Dauer Ihrer Tests reicht dabei vollkommen aus. Add-ons müssen hierfür nicht gelöscht bzw. deinstalliert werden.

³ <http://www.telerik.com/download/fiddler/>

Es erscheint ein Menü. Hier gehen Sie auf den Menüeintrag *Einstellungen*. Im darauffolgenden Menü wählen Sie ebenfalls den Eintrag mit dem Namen *Einstellungen*. Hierauf öffnet sich das Firefox-Einstellungsfenster (siehe Bild 4.31). Öffnen Sie die *Erweitert*-Ansicht (ganz rechts) und wählen Sie anschließend den *Netzwerk*-Reiter aus. Im Bereich *Verbindung* klicken Sie dann auf den Button *Einstellungen...*

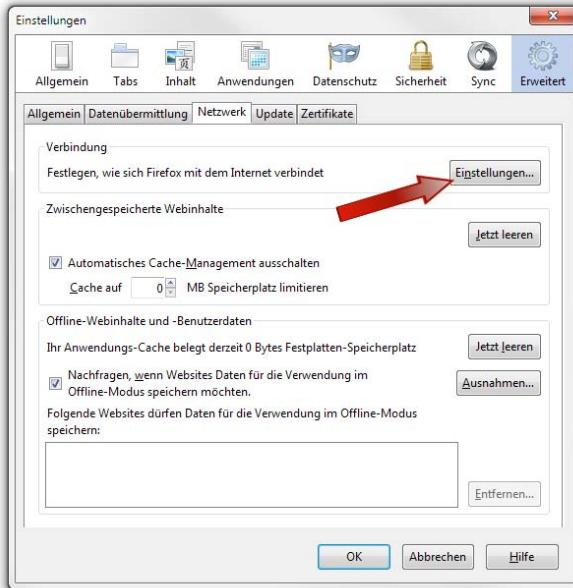


Bild 4.31 Firefox-Netzwerk-Einstellungen

In den Netzwerkeinstellungen können Sie u. a. Proxy-Einstellungen vornehmen (siehe Bild 4.32). Um die Einstellungen für Fiddler vorzunehmen, aktivieren Sie den Radiobutton *Manuelle Proxy-Konfiguration*. In dem dadurch freigeschalteten Formular tragen Sie in das Textfeld *HTTP-Proxy* den Wert `127.0.0.1` und in das Textfeld *Port* den Wert `8888` ein. Bestätigen Sie Ihre Einstellungen durch einen Klick auf den *OK*-Button in beiden geöffneten Fenstern. Nun ist Ihr Firefox auf den Proxy-Server von Fiddler eingestellt.

Im neu geöffneten Fenster können Sie dann unter dem Punkt *Proxyserver* die Checkbox *Proxyserver für LAN verwenden* anwählen oder abwählen (siehe Bild 4.33).

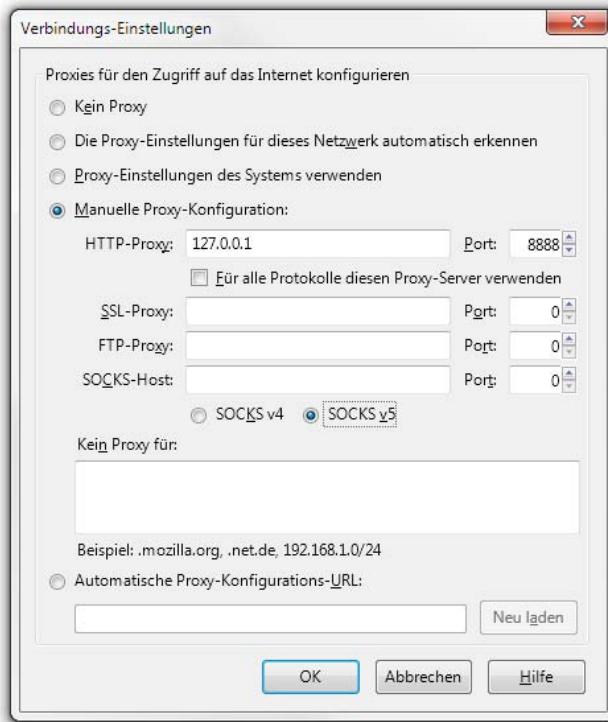


Bild 4.32 Proxy-Einstellungen für einen lokalen Fiddler in Firefox

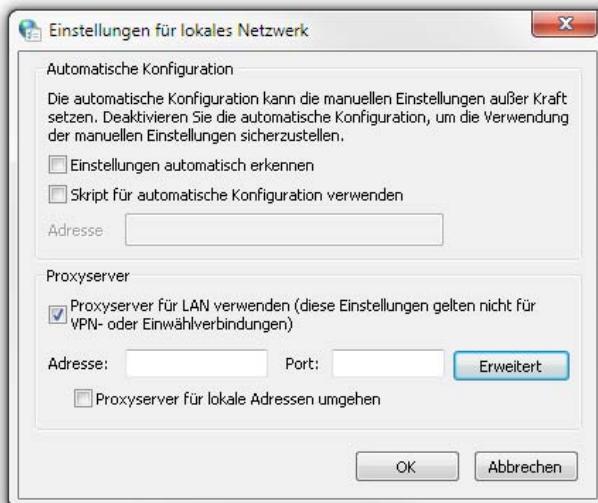


Bild 4.33 Proxeinstellungen in Windows (z. B. für IE und Chrome)



Fiddler fungiert als Proxy-Server zwischen Client und Server. Beim Starten der Anwendung wird der Proxy-Server automatisch gestartet und beim Verlassen von Fiddler auch wieder automatisch angehalten. Ist die Fiddler-Anwendung also nicht geöffnet, ist auch kein Proxy-Server vorhanden und damit keine Verbindung zum Internet verfügbar, falls in Ihren Anwendungen der Proxy-Server noch eingetragen ist. Achten Sie also darauf, dass Sie die Proxy-Konfigurationen wieder umstellen, wenn Sie Fiddler nicht im Betrieb haben. In den Internetoptionen unter Windows können Sie dies einstellen. Die *Internetoptionen* finden Sie in der *Systemsteuerung* (Anzeige: Kategorie) unter *Netzwerk und Internet*. Dort gehen Sie auf den Tab *Verbindungen* und klicken auf *LAN-Einstellungen* (siehe Bild 4.34).



Bild 4.34 Internetoptionen in Windows (z. B. für IE und Chrome)

Starten der Protokollierung

Das Fiddler-Hauptfenster ist in zwei Unterfenster aufgeteilt, die standardmäßig nebeneinander angeordnet sind (siehe Bild 4.35). Links werden alle aufgezeichneten HTTP-Antworten aufgelistet. Im rechten Fenster können verschiedenste Ansichten gruppiert durch Tabs ausgewählt werden.

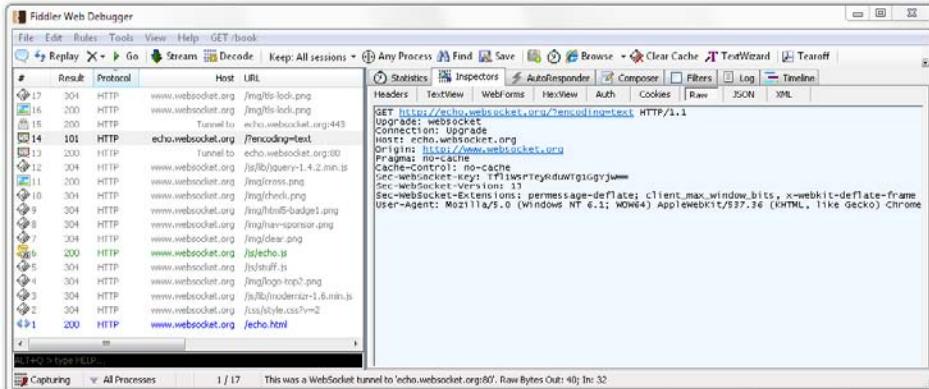


Bild 4.35 Hauptfenster von Fiddler

Der Reiter *Statistics* zeigt – wie der Name schon vermuten lässt – Statistiken über die protokollierte HTTP-Anfrage bzw. HTTP-Antwort an. Dort können Sie z. B. nachlesen, wann sich Ihr Browser mit dem Server verbunden hat, wann er den Request gestartet hat, wie lange die TCP-Verbindung gedauert hat und vieles mehr. Zusätzlich bietet Ihnen Fiddler eine Visualisierung an, die das Verhältnis zwischen Header- und Nutzdaten eines bzw. mehrerer Pakete als Kuchendiagramm darstellt (siehe Bild 4.36). Das Kuchendiagramm wird für die markierten Nachrichten erzeugt, indem Sie auf *Collapse Chart* klicken.

Unter dem Tab *Inspectors* können Sie die Anfragen und Antworten untersuchen. Auch hier bietet Fiddler verschiedene Ansichten wie z. B. die Darstellung des HTTP-Request oder der HTTP-Response in Hex- und Rohform an. Sie können sich auch Cookies und die Nutzdaten samt GET- und POST-Parameter ansehen. Im *Autoresponder*-Reiter stehen Ihnen Funktionen bereit, mit denen Sie z. B. Anfragen des Browsers abfangen und manipulieren können. Mit *Composer* haben Sie die Möglichkeit, eine Anfrage selber zu erstellen. Dieser Reiter teilt sich nochmals in vier Unterreiter auf. Im ersten Tab *Parsed* können Sie Ihren Request mit verschiedenen Formularen konstruieren. Wollen Sie Ihren Request allerdings im Rohformat eingeben, empfehlen wir Ihnen, einen Tab weiter zu klicken. Im *Scratchpad*-Tab können Sie mehrere Requests im Rohformat eintragen, aber jeweils nur einen abschicken. Durch Markieren des Requests und einen Klick auf *Execute* führen Sie den markierten Request aus. Einen Tab weiter können Optionen eingestellt werden, auf die wir erst mal nicht näher eingehen wollen. Sie können unter dem Tab *Filter* nach bestimmten Einträgen suchen. Die Logeinträge von Fiddler sind im *Log*-Tab aufgelistet. In dem *Timeline*-Tab wird die Transferzeit der Einträge grafisch dargestellt. Sie können auch mehrere Einträge auf einmal markieren, um z. B. die Transferzeiten miteinander zu vergleichen. Fiddler passt die Grafik dementsprechend an.

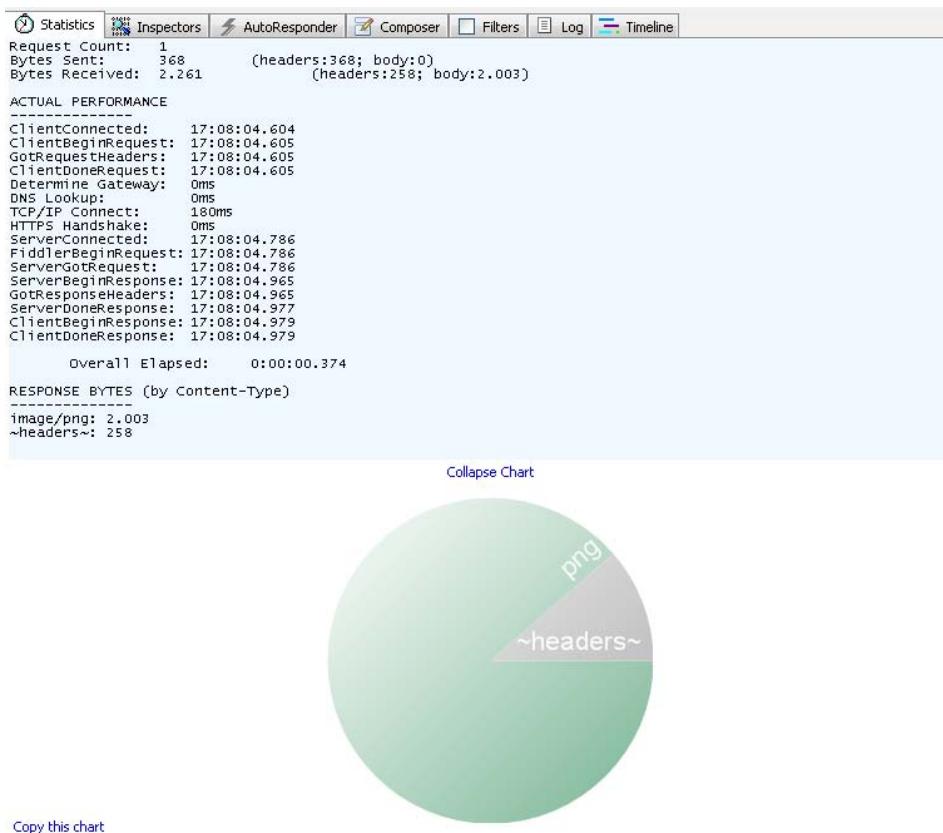


Bild 4.36 Statistiken und Kuchendiagramm eines Frames

Analyse des WebSocket-Datenverkehrs

Fiddler ist leider nur auf das HTTP-Protokoll beschränkt, daher kann nur der WebSocket-Handshake, der aus einem HTTP-Request besteht, vollwertig analysiert werden.

Starten Sie eine WebSocket-Verbindung, indem Sie beispielsweise, wie in [Abschnitt 4.9.1](#) beschrieben, die Webseite

<http://www.websocket.org/echo.html>

in Ihrem Browser aufrufen. Verbinden Sie sich mit dem Server. Dies reicht bereits, da der Verbindungsaufbau hiermit abgeschlossen ist und Fiddler nicht mehr als das wahrnehmen kann.

Suchen Sie nun im linken Fenster der Fiddler-Anwendung nach einem Eintrag, der mit dem Statuscode 101 versehen ist. Da das linke Fenster so organisiert ist, Ihnen die HTTP-Antworten anzuzeigen, müssen Sie nach der HTTP-Antwort des Opening-Handshakes Ausschau halten. Wenn Sie einen solchen in der Liste entdeckt haben und auf die WebSocket-Handshake-Response klicken, wird der Inspector-Tab auf der rechten Seite geöffnet. Dort werden der Request- und die Response-Nachricht angezeigt. Unter der Auswahl

Hexview werden die Handshake-Informationen in der Hexdump-Darstellung, wie in Wireshark (siehe [Abschnitt 4.9.1](#)), dargestellt. Wenn Sie im Tab *Statistics* das Kuchendiagramm öffnen, werden Sie feststellen, dass der WebSocket-Handshake nur aus Overhead-Daten besteht. Es werden dort keine Nutzdaten übertragen (siehe [Bild 4.37](#)).

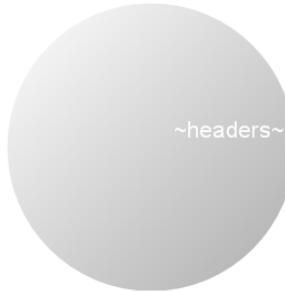


Bild 4.37 Kuchendiagramm eines WebSocket-Handshakes

Da Fiddler wie erwähnt nur HTTP-Nachrichten in seinen Funktionen betrachtet, spielen Control- und Datenframes im Kuchendiagramm wie auch in den anderen Fiddler-Tools keine Rolle.

Konstruktion von WebSocket-Handshakes

Im *Composer*-Reiter können Sie einen Opening-Handshake-Request erstellen, ohne programmieren zu müssen. Um dies für den Anwender so komfortabel wie möglich zu gestalten und den Aufwand für das manuelle Erstellen des Requests weitestgehend zu minimieren, können Sie bereits aufgezeichnete Nachrichtenpaare von WebSocket-Handshakes per Drag-and-Drop ins Composer-Fenster ziehen. Fiddler analysiert den Request und passt die Formularfelder dementsprechend an (siehe [Bild 4.38](#)).

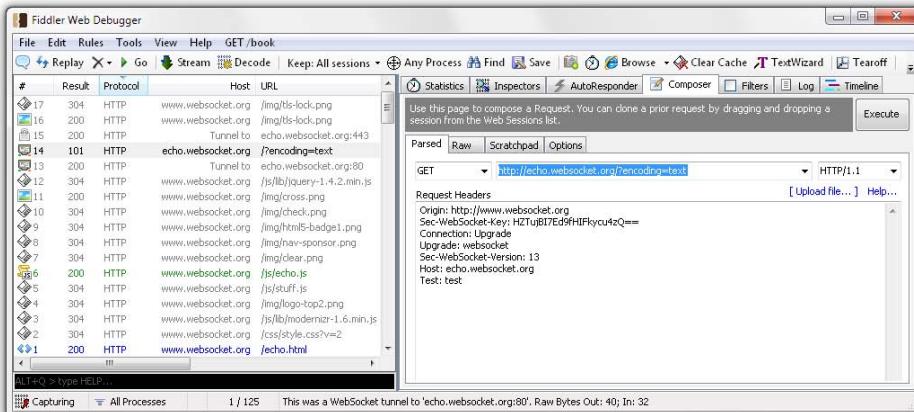


Bild 4.38 Per Drag-and-Drop Einträge in den Composer schieben

Aus der Anfragezeile werden die Methode und die HTTP-Version entnommen. Die URL des WebSocket-Endpunkts wird aus der Anfragezeile und dem Host-Header konstruiert. Die HTTP-Header werden in ein dafür vorgesehenes Textfeld eingefügt. Verwenden Sie bei Ihren ersten Gehversuchen mit Fiddler auch die Echo-Test-Demo-Anwendung von WebSocket.org, so sollten Sie folgende Headereinträge dort wiederfinden:

Listing 4.5 Headereinträge eines Handshake-Requests am Beispiel der Echo-Test-Demo-Anwendung von WebSocket.org im Composer von Fiddler

```
Origin: http://www.websocket.org
Sec-WebSocket-Key: HZTujBI7Ed9fHIFkycu4zQ==
Connection: Upgrade
Upgrade: Websocket
Sec-WebSocket-Version: 13
Host: echo.websocket.org
```

Durch Klicken auf den *Execute*-Button kann der Handshake-Request abgesetzt werden. Hiermit können Sie z. B. prüfen, ob Ihre Serverseite auf Verbindungsanfragen reagiert.

Die Header können Sie zudem beliebig verändern und nach jeder Veränderung beobachten, wie der WebSocket-Server darauf reagiert. Verändern Sie z. B. die Versionsnummer von 13 auf 14. Ein nach dem aktuellen Standard implementierter Server sollte diese Anfrage verweigern. Der WebSocket-Server von WebSocket.org, der die Kaazing-Version benutzt, tut dies auch.

Der Composer ermöglicht Ihnen in Hacker-Manier, alle möglichen Requests zu bauen. Ihrer Fantasie sind dabei keine Grenzen gesetzt. Seien Sie ruhig experimentierfreudig. So können Sie eventuell Fehler oder auch Sicherheitslücken in Ihrer Webanwendung frühzeitig entdecken. Eine ausführlichere Dokumentation über Fiddler finden Sie auf der Internetseite von Telerik⁴.

4.9.3 Chrome Developer Tools

Die Chrome Developer Tools sind integraler Bestandteil des Google Chrome-Webbrowsers und ermöglichen u. a. das Mitlesen der zwischen Chrome und einem WebSocket-Server ausgetauschten Nachrichten (Handshake und WebSocket-Frames).

Installation

Da die Chrome Developer Tools im Google Chrome-Browser integriert sind, müssen Sie einen solchen auf Ihrem System installiert haben. Sie bekommen den Browser auf einer Google-Seite.⁵ Wenn Sie bereits über eine Installation verfügen, achten Sie darauf, dass es sich um die richtige Version handelt. Um WebSockets zu analysieren, wird die Version 20 oder höher benötigt.

⁴ <http://www.telerik.com/support/fiddler/>

⁵ <http://www.google.de/intl/de/chrome/browser/>



Chrome ist für Windows, Mac OS, Linux, Android und für iOS verfügbar. Allerdings sind die Developer Tools nur auf den Desktop-Betriebssystemen Windows, Mac OS und Linux vollständig verfügbar. Für iOS existiert zurzeit keine Möglichkeit, auf die Chrome Developer Tools zurückzugreifen. Unter Android ist dies über die Remote Debugging-Funktionalität möglich (siehe [Anhang B](#)).

Analyse der WebSockets

Die Chrome Developer Tools können entweder über die Chrome-Menüpunkte *Entwickler* und dann *Entwickler-Tools* oder mittels der Tastenkombination *Strg + Umschalt + I* unter Windows/Linux oder *cmd + alt + I* unter Mac OS geöffnet werden. Ein letzter Weg zu den Developer Tools führt über einen Rechtsklick und das Kontextmenü. Wenn Sie in diesem die Option *Element untersuchen* auswählen, gelangen Sie ebenfalls dorthin.



Achten Sie darauf, dass Sie die Entwickler-Tools vor dem Verbindungsauftakt zum WebSocket-Server öffnen. Eine Aufzeichnung des Datenverkehrs kann erst erfolgen, wenn die Tools vorher schon offen sind. Falls Sie dies vergessen haben, lassen Sie einfach die Developer Tools offen und laden Sie die Seite neu.

Für die folgenden Beschreibungen wollen wir erneut auf die durchgängig verwendete Beispianwendung von [WebSocket.org](http://www.websocket.org/echo.html) zugreifen und den dort bereitgestellten Echo-Test verwenden. Wenn Sie mittels der bereitgestellten Webseite mit eingebettetem WebSocket-Client eine Verbindung zum WebSocket-Server aufgebaut haben, klicken Sie in der Developer-Tools-Menüleiste auf den Eintrag *Netzwerk*. Hier sind alle HTTP-Nachrichtenpaare aus Ihrem aktuellen Browser-Tab seit der Aktivierung der Developer Tools aufgelistet. Klicken Sie nun zunächst auf das Filtersymbol und anschließend auf den Punkt *WebSockets*. Hiermit werden alle WebSocket-Verbindungen herausgefiltert und ausschließlich diese angezeigt (siehe [Bild 4.39](#)).

Name	Method	Status	Type	Initiator	Size Content	Time Latency	Timeline
?encoding=text echo.websocket.org	GET	101 Web Socket Protocol Handshake		Other	338 B 0 B	Pending	1.00 s

Bild 4.39 Auflistung der WebSockets in Chrome Developer Tools

Wenn Sie nun auf einen der WebSocket-Einträge klicken, erscheint rechts ein Ausgabebereich, in dem detailliertere Informationen zum ausgewählten Eintrag angezeigt werden (siehe [Bild 4.40](#)).

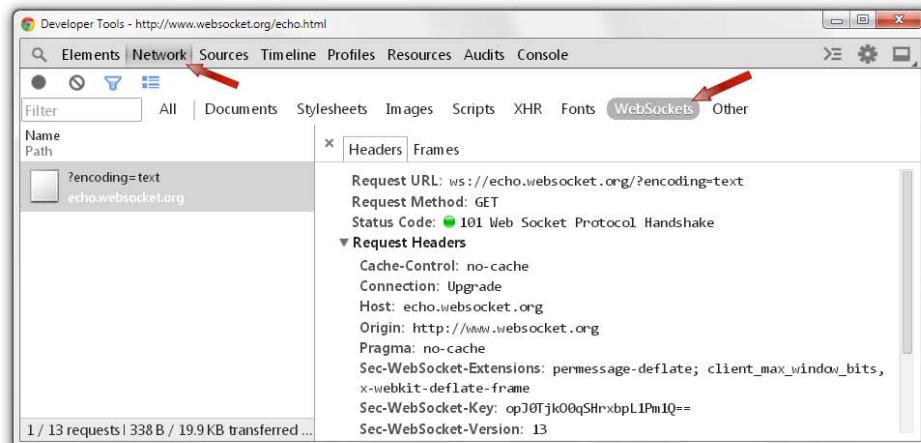


Bild 4.40 Handshake-Header in Chrome Developer Tools

Im Reiter *Headers* sehen Sie den Opening-Handshake. Dieser ist aufgeteilt in einen Request-Header und einen Response-Header. Einen Tab weiter werden die Datenframes angezeigt. Aus den protokollierten Frames können der Inhalt und die Länge der Nachricht entnommen werden. Zudem vermerkt Chrome den Zeitpunkt, an dem die Nachricht eingegangen ist. Achten Sie bitte darauf, dass die Anzeige der WebSocket-Frames nicht automatisch neu lädt, falls weitere Nachrichten verschickt werden oder ankommen. Dies müssen Sie manuell tun, indem Sie einfach wieder auf den Endpoint klicken (siehe Bild 4.41).

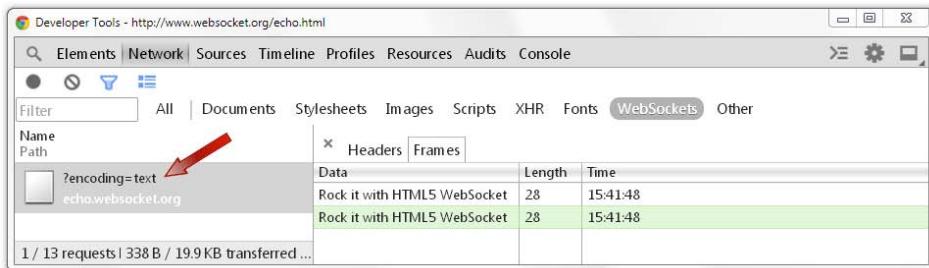


Bild 4.41 Analyse von WebSocket-Frames mit den Chrome Developer Tools

4.9.4 Chrome Network Internals

Die *Chrome Network Internals* bieten Ihnen die Möglichkeit, alle Netzwerkaktivitäten Ihres Google Chrome-Browsers anzeigen zu lassen und diese zu analysieren. Sie erreichen diese Funktion unter der URL

`chrome://net-internals`

die Sie in das URL-Textfeld eingeben müssen (siehe Bild 4.42).

Mit diesem Tool können Sie z. B. DNS-Anfragen, SPDY-Sessions, TCP-Timeouts und eine Menge andere Netzwerkparameter anschauen. Im Drop-down-Menü finden Sie den Eintrag *Sockets* (siehe Bild 4.42).

The screenshot shows the 'Sockets' tab in the Google Chrome Network Internals interface. It displays two tables: one for the transport_socket_pool and one for the ssl_socket_pool. The transport_socket_pool table has columns: Name, Handled Out, Idle, Connecting, Max, Max Per Group, and Generation. It shows entries for 'transport_socket_pool' and 'ssl_socket_pool'. The ssl_socket_pool table has columns: Name, Pending, Top Priority, Active, Idle, Connect Jobs, Backup Timer, and Stalled. It shows an entry for 'www.websocket.org:80'.

Name	Handled Out	Idle	Connecting	Max	Max Per Group	Generation
transport_socket_pool	0	1	0	256	6	0
ssl_socket_pool	0	0	0	256	6	0

transport_socket_pool							
Name	Pending	Top Priority	Active	Idle	Connect Jobs	Backup Timer	Stalled
www.websocket.org:80	0	-	0	1	0	stopped	false

Bild 4.42 Google Chrome Network Internals

Wenn Sie diesen aktivieren, öffnet sich im Contentbereich eine Socketpool-Ansicht, die alle geöffneten TCP-Sockets auflistet. Wenn Sie in dieser Ansicht auf den Link *View Live Sockets* klicken, gelangen Sie auf die *Events*-Seite, die alle offenen TCP-Sockets anzeigt (siehe Bild 4.43).

The screenshot shows the 'Events' tab in the Google Chrome Network Internals interface. It displays a table of open sockets. The table has columns: ID, Source Type, and Description. The descriptions list various hosts and ports, such as 'ssl/www.google.de:443', 'websocket.org:80', and 'ws://echo.websocket.org/?encoding=text'.

ID	Source Type	Description
1030	SOCKET	ssl/www.google.de:443
1076	SOCKET	websocket.org:80
1109	SOCKET	www.websocket.org:80
1136	SOCKET	www.websocket.org:80
1145	SOCKET	www.websocket.org:80
1154	SOCKET	www.websocket.org:80
1195	SOCKET	www.websocket.org:80
1206	SOCKET	www.websocket.org:80
1353	SOCKET_STREAM	ws://echo.websocket.org/?encoding=text
1357	SOCKET	ws://echo.websocket.org/?encoding=text

Bild 4.43 Offene TCP-Sockets des Google Chrome-Browsers

Wenn Sie keine aktive WebSocket-Verbindung offen haben, taucht in der Auflistung entsprechend nichts Derartiges auf. Daher wollen wir diesen Fall nun hervorrufen. Öffnen Sie dazu einen neuen Tab und öffnen Sie z. B. die URL

<http://www.websocket.org/echo.html>

Wenn Sie sich mithilfe des auf der Webseite ausgelieferten Formulars mit dem WebSocket-Echo-Server verbunden haben, gehen Sie zurück in den Chrome Network Internals Tab mit den Live Events. Hier sollten nun zwei neue Einträge mit den Source-Typen *Socket_Stream* und *Socket* in der Auflistung hinzugekommen sein. Um den WebSocket-Verbindungsauflauf nachvollziehen zu können, öffnen Sie den Eintrag mit dem Source-Type *Socket_Stream*, der Ihnen detaillierte Informationen zum Handshake bereitstellt (siehe Bild 4.44).

```
1353: SOCKET_STREAM
ws://echo.websocket.org/?encoding=text
Start Time: 2014-02-14 16:30:09.734

t=1392391009734 [st= 0] +REQUEST_ALIVE [dt=7]
t=1392391009734 [st= 0] +SOCKET_STREAM_CONNECT [dt=203]
    >>> host="echo.websocket.org" /?encoding=text
t=1392391009734 [st= 0] +PROXY_SERVICE [dt=0]
    >>> proxy_service_resolved_proxy_list
        --> proxy_string = "DIRECT"
t=1392391009734 [st= 0] -PROXY_SERVICE
t=1392391009735 [st= 1] +PROXY_SERVICE [dt=0]
t=1392391009735 [st= 1] -PROXY_SERVICE_BESOLVED_PROXY_LIST
        --> proxy_string = "DIRECT"
t=1392391009735 [st= 1] -PROXY_SERVICE
t=1392391009735 [st= 1] HOST_RESOLVED_IMPL [dt=32]
        --> source_dependency = 1354 (HOST_RESOLVER_IMPL_REQUEST)
t=1392391009937 [st=203] -SOCKET_STREAM_CONNECT
t=1392391009938 [st=204] WEB_SOCKET_PROTOCOL_HEADERS
    --> GET /*echo-dictate*/ HTTP/1.1
        Upgrade: websocket
        Connection: Upgrade
        Host: echo.websocket.org
        Origin: http://www.websocket.org
        Pragma: no-cache
        Cache-Control: no-cache
        Sec-WebSocket-Key: H12y3T7UfcNi0uCzt2VjQw=
        Sec-WebSocket-Version: 13
        Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits, x-websocket-deflate-frame
        User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.107 Safari/537.36

t=1392391009938 [st=204] SOCKET_STREAM_SEND
t=1392391010110 [st=376] SOCKET_STREAM_RECEIVED
t=1392391010110 [st=376] WWW_SOCKET_PROTOCOL_HEADERS
    --> HTTP/1.1 101 Web Socket Protocol Handshake
        Upgrade: WebSocket
        Connection: Upgrade
        Sec-WebSocket-Accept: H12y3T7UfcNi0uCzt2VjQw=
        Server: Apache/2.2.21 (Ubuntu)
        Date: Fri, 14 Feb 2014 15:19:24 GMT
        Access-Control-Allow-Origin: http://www.websocket.org
        Access-Control-Allow-Credentials: true
        Access-Control-Allow-Headers: content-type
        Access-Control-Allow-Headers: authorization
        Access-Control-Allow-Headers: x-websocket-extensions
        Access-Control-Allow-Headers: x-websocket-version
        Access-Control-Allow-Headers: x-websocket-protocol
```

Bild 4.44 Analyse des Handshakes mit den Chrome Network Internals

In dieser Ansicht wird der Inhalt des gesendeten Handshake-Requests und der Handshake-Responses gezeigt. Die Aufschlüsselung enthält alle Einträge der ausgetauschten HTTP-Nachrichten.

Wenn Sie den Eintrag mit dem Source-Type *Socket* öffnen, bekommen Sie Informationen zu einem WebSocket auf TCP-Ebene (siehe Bild 4.45).

1357: SOCKET**ws://echo.websocket.org/?encoding=text**

StartTime: 2014-02-14 16:30:09.767

```
t=1392391809767 [st= 0] +SOCKET_ALIVE [dt=?]
--> source_dependency = 1353 (SOCKET_STREAM)
t=1392391809767 [st= 0] +TCP_CONNECT [dt=170]
--> address_list = ["174.129.224.73:80"]
t=1392391809768 [st= 1]   TCP_CONNECT_ATTEMPT [dt=169]
--> address = "174.129.224.73:80"
t=1392391809937 [st=170] -TCP_CONNECT
--> source_address = "192.168.0.5:21076"
t=1392391809938 [st=171] SOCKET_BYTES_SENT
--> byte_count = 465
t=1392391810110 [st=343] SOCKET_BYTES_RECEIVED
--> byte_count = 542
```

Bild 4.45 TCP-Verbindungsaufbau eines WebSockets in den Chrome Network Internals

Den bereitgestellten Informationen können Sie entnehmen, zwischen welchen beiden IP-Endpunkten die TCP-Verbindung auf welchen Ports geöffnet wurde. Das Beispiel in [Bild 4.45](#) zeigt die TCP-Verbindung zwischen dem Chrome-Browser in einem lokalen Netz (192.168.0.5) und dem Echo-WebSocket auf WebSocket.org (174.129.224.73). Die letzten beiden Einträge *Socket_Bytes_Sent* und *Socket_Bytes_Received* zeigen den Handshake-Request und die Handshake-Response, die zwischen Browser und Server zum Verbindungsaufbau ausgetauscht wurden (inkl. der dazu benötigten Bytes).

Wenn Sie nun eine Nachricht verschicken bzw. wenn Sie eine Nachricht empfangen, wird dies umgehend in der LiveView-Ansicht durch einen entsprechenden neuen Eintrag angezeigt (siehe [Bild 4.46](#)).

t=1392392753913 [st=944146]	SOCKET_BYTES_SENT --> byte_count = 34
t=1392392754080 [st=944313]	SOCKET_BYTES_RECEIVED --> byte_count = 30

Bild 4.46 Anzahl der gesendeten und empfangenen Bytes

Das in [Bild 4.46](#) dargestellte Beispiel zeigt die Größe einer an den Echo-WebSocket von WebSocket.org gesendeten Nachricht und die daraufhin vom Server empfangene Antwort. Hieran können Sie noch mal gut nachvollziehen, dass die Größe der gesendeten Nachrichten vom Client wegen des Masking-Keys um 4 Bytes größer als die Nachricht vom Server ist.

5

Der Client: Die W3C WebSocket-API

Damit Sie eine WebSocket-Verbindung zwischen Client und Server aufbauen und damit das Protokoll, das wir uns im vorherigen Kapitel zu Gemüte geführt haben, anstoßen können, stellt Ihnen ein entsprechender W3C-Standard die dafür notwendige JavaScript-API [[Hic12b](#)] bereit. Hieran lässt sich schön die inhaltliche Trennung der Standardisierungsgremien IETF und W3C sehen. Die IETF standardisiert Internet-Protokolle, wozu die Protokollspezifikation der WebSockets gehört (siehe [Kapitel 4](#)). Das W3C hingegen spezifiziert Webtechnologien, die insbesondere den Browser angehen. Hierzu zählt folglich die API-Spezifikation des WebSocket-Standards, die im Browser implementiert sein muss und die wir in diesem Kapitel genauer betrachten möchten.



Fragen, die dieses Kapitel beantwortet:

- Wie wird ein WebSocket-Objekt instanziert?
- In welchen Zuständen kann es sich während seines Lebens befinden?
- Wie kann man darüber Daten senden und empfangen?
- Wie geht man mit Fehlern um?

■ 5.1 Was bisher geschah

Alle Entwürfe vom ersten bis zum letzten Working Draft sowie die Candidate Recommendation und die Editor's Drafts wurden verfasst vom Google-Mitarbeiter Ian Hickson. Ian Hickson verfasste auch den ersten Internet-Draft des WebSocket-Protokolls am 9. Januar 2009.

Der erste Working Draft der WebSocket-API erschien am 23. April 2009. Am 29. Oktober 2009 wurde bereits der zweite Working Draft herausgegeben. In dieser Version wurde der Konstruktor durch einen zweiten optionalen Parameter ergänzt, worin ein Subprotokoll übergeben werden konnte. Des Weiteren wurde die API überarbeitet. Für das Versenden von Nachrichten und das Schließen der Verbindung wurden in der ersten Version noch die Methoden `postMessage()` und `disconnect()` verwendet. Diese Methoden wurden im zweiten Working Draft zu `send()` und `close()`. Es wurde außerdem ein Attribut namens

bufferedAmount für das WebSocket-Objekt hinzugefügt. Damit kann die Anzahl an Bytes ausgelesen werden, die noch nicht versendet wurden.

Danach folgten zwei weitere Working Drafts am 22. Dezember 2009 und am 11. April 2011. Die Version des 11. April 2011 wurde durch den Event-Handler onerror ergänzt. Zusätzlich kann seitdem ein zweiter optionaler Parameter genutzt werden, wodurch mehrere Subprotokolle in Form eines String-Arrays angegeben werden können statt bis dato nur eins. Somit kann ein Server eines der Subprotokolle auswählen, das er unterstützen will. Aus diesem Grund wurde zusätzlich das Attribut protocol eingeführt. Der Client ist dadurch in der Lage zu erkennen, welches dieser Subprotokolle vom Server unterstützt wird. Die Anzahl der Ready States wurde im April 2011 durch CLOSING ebenfalls um eins erhöht. Vorher existierten nur die drei Verbindungszustände CONNECTING, OPEN und CLOSED. In dieser Version konnte auch zum ersten Mal bei einer Beendigung der Verbindung ausgelesen werden, ob die Verbindung sauber geschlossen wurde.

Am 29. September desselben Jahres erschien der nächste Working Draft. Dieser Arbeitsentwurf ermöglichte es, Binärdaten in Form von BLOBs und ArrayBuffern zu versenden. Die bisherigen Versionen hatten nur Strings vorgesehen. Da jetzt zwei Arten von Binärdaten versendet und empfangen werden konnten, hat das W3C in dieser Version das Attribut binaryType eingeführt. Der Client kann dadurch festlegen, in welcher Form er die Binärdaten empfangen will. Bei der close() -Methode kam die Möglichkeit hinzu, optional einen Close-Code als Zahl und einen Grund für die Beendigung als String an den Server mitzugeben. Zusätzlich wurde noch das Attribut extensions hinzugefügt.

Im Dezember 2011 wurde die erste Candidate Recommendation der WebSocket-API veröffentlicht. Danach folgten zwei weitere Working Drafts am 24. Mai 2012 und am 9. August 2012. Die zweite Candidate Recommendation wurde am 20. September 2012 vorgestellt. Darauf folgte am 04. Juni 2014 der letzte Editor's Draft.

■ 5.2 Browserunterstützung

WebSockets sind fester Bestandteil der HTML5-Standardfamilie. Als solcher ist der Support in den aktuellen Versionen der fünf Freunde schon recht breit. Die Website

<http://www.caniuse.com/>

(gesprochen: can i use <dot> com) gibt über die Browserunterstützung vieler neuer Webtechnologien – insbesondere im HTML5-Umfeld – Auskunft. Tabelle 5.1, die einen Teil der Kompatibilitätstabelle von caniuse.com zur WebSockets-Unterstützung zeigt, macht einem Mut. Sie zeigt, dass alle relevanten Browser sowohl im Desktop- als auch im Mobile-Bereich WebSockets unterstützen. Die grauen Versionsnummern bedeuten eine teilweise Unterstützung, wohingegen die weiß markierten Browsersversionen nicht mit WebSockets umgehen können.

Von Hand können Sie auch schnell herausfinden, ob ein Browser WebSockets unterstützt. Hat er das Protokoll und die API implementiert, dann finden Sie im window-Objekt ein WebSocket-Element. Dieses fehlt entsprechend, wenn der Browser keine native Unterstüt-

Tabelle 5.1 Browserunterstützung für WebSockets [Dev14c]

IE	Firefox	Chrome	Safari	Opera	iOS-Safari	Android-Browser	Opera-Mobile	Blackberry-Browser
		4						
		5						
2	6							
3	7							
3.5	8							
3.6	9							
4	10							
5	11							
6	12							
7	13							
8	14							
9	15							
10	16							
11	17							
12	18							
13	19			9.6				
14	20			10.1				
15	21			10.5				
16	22			10.6				
17	23			11				
18	24			11.1				
19	25			11.5				
20	26			11.6				
21	27			12				
22	28			12.1				
23	29			15				
24	30	3.1		16				
25	31	3.2		17		2.1		
26	32	4		18	3.2	2.2		
5.5	27	33	5	29	4.1	2.3		
6	28	34	5.1	20	4.3	3		
7	29	35	6	21	5.1	4	10	
8	30	36	6.1	22	6.1	4.1	11.5	
9	31	37	7	23	7.1	4.3	12	
10	32	38	7.1	24	8	4.4	12.1	7
11	33	39	8	25	8.1	4.4.4	24	10
	34	40		26		37		
	35	41		27				
	36	42						

zung für WebSockets mit an Bord hat. Somit ist der Test mit den JavaScript-Codezeilen aus [Listing 5.1](#) getan (unbedingt auf die Groß- und Kleinschreibung achten!).

Listing 5.1 Test auf WebSocket-Unterstützung in einem Browser

```

1  <html>
2      <body>
3          <script type="text/javascript">
4              if ("WebSocket" in window) {
5                  alert("Super! WebSockets werden unterstuetzt!");
6              }
7              else{
8                  alert("Schade! Wir muessen uns um eine Alternative
9                      kuemmern.");
10             }
11         </script>
12     </body>
13 </html>
```

In den meisten Fällen sollte ein Check auf WebSocket-Support im Browser ausreichen, um loslegen zu können. Es können sich bei erfolgreichem Check dennoch Problemchen einschleichen. Der Teufel steckt im Detail. Dies kann Ihnen passieren, wenn ältere Protokollimplementierungen auf der Client- oder Serverseite zum Einsatz kommen. Welche Version des Protokolls Sie wo antreffen können, haben wir in [Tabelle 5.2](#) zusammengetragen.

Tabelle 5.2 Browserunterstützung in Bezug auf WebSocket-Protokollversionen [[sta14](#), [Aut12](#)]

WebSocket Protokoll- version	Chrome	Firefox	IE	Opera	Opera Mobile	Safari
Hixie-75	4.0, 5.0	–	–	–	–	5.0.0
HyBi-00/ Hixie-76	6.0 - 13.0	4.0 (standard- mäßig deaktiviert)	–	11 (standard- mäßig deaktiviert)	–	5.0.2, 5.1
HyBi-07+	14.0	6.0 (Präfix: MozWeb- Socket)	9.0 (Silverlight Erweiterung)	–	–	–
HyBi-10	14.0, 15.0	7.0 - 10.0 (Präfix: MozWeb- Socket)	10 (Windows 8 Developer Preview)	–	–	–
HyBi-17/ RFC 6455	16	11	10	12.10	12.10	7

Sie sollten darauf achten, dass die Komponenten des Anwendungssystems, die Sie mit Ihren Implementierungen beeinflussen können, auf die letzte Version des Protokolls HyBi-17 bzw. RFC 6455 abstellt. Ihre Tests sollten unbedingt auch frühere Versionen der Browser mit

WebSocket-Support mit einschließen, damit Sie feststellen können, ob die Verwendung Ihrer Anwendung mit älteren Browsern funktioniert. Ergeben die Tests das Gegenteil, müssen für diese – ähnlich wie für Browser ohne WebSocket-Support – alternative Fallbacks vorgesehen und implementiert werden.

Wenn Ihnen die WebSocket-Browserabdeckung so nicht reicht und Sie höhere Anforderungen an die zu unterstützenden Browser haben, dann müssen Sie ebenfalls Fallbacks implementieren. Diese beruhen in der Regel auf den in [Kapitel 3](#) genannten Verfahren wie Long-Polling oder Comet. Tatsächlich nehmen Ihnen viele clientseitige Frameworks für die Entwicklung mit WebSockets diese Arbeit bereits ab. Dazu an gegebener Stelle mehr (siehe [Abschnitt 6.2.3](#)).

■ 5.3 Namensschema

Wie können Sie nun einen WebSocket-Server ansprechen? Wie Sie einen Webserver ansprechen, wissen Sie. Die URL spielt dabei eine wesentliche Rolle. In [Kapitel 2](#) haben wir gesehen, wie eine URL allgemein aufgebaut ist und wie Sie eine solche für konkrete Ausprägungen von Webressourcen konstruieren müssen. Zur Adressierung eines WebSocket-Servers können wir auf diesem Wissen aufbauen und werden einige Déjà-vus erleben. In Kapitel 3 des RFC 6455 finden Sie dazu die folgende Spezifikation [[FM11a](#)]:

```
"ws://" host [ ":" port ] path [ "?" query ]
```

Bei der Notation handelt es sich um die angereicherte Backus-Naur-Form (ABNF), die im RFC 3986 [[CO08](#)] definiert ist. Damit Sie die angegebene URL-Konstruktionsregel lesen und verstehen können, müssen Sie wissen, dass die eckigen Klammern optionale Terme bezeichnen, die in einer URL nicht zwingend auftauchen müssen. Die Terme in Anführungszeichen stellen Zeichenketten dar, die genau in der angegebenen Form in der URL eingefügt werden. Terme ohne Anführungszeichen sind variable Platzhalter, die mit entsprechendem Inhalt zu füllen sind. Sowohl der Platzhalter für den Host, die Portnummer und den Query-String kennen wir schon aus regulären Web-URLs. Eine einfache WebSocket-URL kann also z. B. wie folgt ausgestaltet sein:

ws://chat.example.com/

oder auch

ws://noch-deins-bald-meins.de/auktionen/

Einfach gesprochen, unterscheidet sich eine WebSocket-URL von einer Web-URL damit im Wesentlichen durch den angepassten Schema, der nicht wie im Web üblich "http:", sondern für WebSockets "ws:" ist.

Neben dieser URL-Spezifikation definiert der RFC 6455 noch eine zweite URL, die nach der folgenden ABNF aufgebaut ist:

```
"wss://" host [ ":" port ] path [ "?" query ]
```

Wie Sie sehen, unterscheidet sich diese URL-Konstruktionsregel nur durch einen winzigen Zusatz im Vergleich zur vorherigen. Dieser kleine, aber feine Unterschied befindet sich im Schema der URL und enthält ein zusätzliches „s“. Dieses steht für „secure“ und ist das Pendant zu „`https:`“. Dazu mehr, wenn wir in [Abschnitt 7.2.2](#) sicherheitsrelevante Aspekte beleuchten.

■ 5.4 Objekterzeugung und Verbindungsauftbau

Da die clientseitige Programmierung in der Regel in einem Webbrowser stattfindet, ist die hierfür definierte API eine vom W3C standardisierte JavaScript-API. Dreh- und Angelpunkt ist dabei eine Instanz eines WebSocket-Objekts. Eine solche können Sie mittels der bereitgestellten Konstruktoren erzeugen. Abstrakt geschrieben, stehen Ihnen die folgenden Konstruktoren zur Verfügung:

```
[Constructor(  
    DOMString url, optional (DOMString or DOMString[]) protocols)]
```

Das ist die Schreibweise, die im Standard genutzt wird. Sie basiert auf der WebIDL [[McC12](#)], einer abstrakten Schnittstellenbeschreibungssprache, die Sie in vielen W3C-Standards antreffen. Eine kürzere Notation kann auf Grundlage der angereicherten Backus-Naur-Form (ABNF) [[CO08](#)] wie folgt angegeben werden:

```
WebSocket(url[, protocols])
```

Ob die eine oder die andere Schreibweise, aus beiden können Sie die drei verschiedenen Wege der Objekterzeugung ablesen, die Ihnen bereitstehen. Der minimale Konstruktor erwartet genau *ein* String-Argument, das ihm die URL des WebSocket-Endpunkts angibt. Möchten Sie z. B. eine Verbindung zum WebSocket-Server

```
ws://echo.websocket.org/
```

aufbauen, so müsste Ihr Konstruktorauftruf wie folgt aussehen:

```
var ws = new WebSocket('ws://echo.websocket.org/');
```

Sie können dem WebSocket-Konstruktor aber auch noch ein zweites String-Argument übergeben. Das erste bleibt wie gehabt die zwingend erforderliche URL des adressierten Servers. Das zweite kann optional das Subprotokoll angeben, das über den WebSocket-Kanal zwischen Client und Server verwendet werden soll. Für das folgende Beispiel wollen wir als Subprotokoll SOAP [[GHM⁺07](#)] verwenden:

```
WebSocket ws = new WebSocket('ws://echo.websocket.org/', 'soap');
```

Dies hat den Effekt, dass in die HTTP GET-Anfrage des WebSocket-Handshakes der folgende zusätzliche Headereintrag hinzugefügt wird:

Sec-WebSocket-Protocol: soap

Sie können unter Verwendung eines String-Arrays im zweiten Argument auch mehrere Subprotokolle angeben:

```
WebSocket ws =
    new WebSocket('ws://echo.websocket.org/', ['soap', 'wamp']);
```

Der Headereintrag der GET-Anfrage im WebSocket-Handshake hat dann entsprechend folgendes Aussehen:

Sec-WebSocket-Protocol: soap, wamp

Wie wir bereits in [Abschnitt 4.2](#) besprochen haben, kommt es nur zu einer WebSocket-Verbindung, wenn der Server eines der angegebenen Subprotokolle akzeptiert. Welches Subprotokoll der Server schließlich gewählt hat, können Sie anhand eines Attributs, das wir in [Abschnitt 5.8](#) behandeln, in Erfahrung bringen. Registrierte Subprotokolle sind in der WebSocket Subprotocol Name Registry der IANA nachzusehen [[MML⁺14](#)].

■ 5.5 Zustände

Haben Sie auf diese Weise ein WebSocket-Objekt instanziert, wurde der Handshake zwischen Client und Server durchgeführt. Je nachdem ob der Verbindungsauftbau erfolgreich war oder nicht, befindet sich Ihr Objekt in einem der im Standard dafür spezifizierten Zustände. Genauer gesagt sind es vier Zustände, die Sie in [Tabelle 5.3](#) aufgelistet sehen.

Tabelle 5.3 Zustände und das readyState-Attribut gemäß [[Hic12b](#)]

Zustand	Beschreibung
CONNECTING (readyState: 0)	Das Objekt wurde instanziert und befindet sich aktuell dabei, eine Verbindung zum Server aufzubauen.
OPEN (readyState: 1)	Die Verbindung ist erfolgreich aufgebaut und damit steht ein kommunikationsbereiter Kanal zur Verfügung.
CLOSING (readyState: 2)	Die Verbindung geht durch einen Closing-Handshake und wird damit geschlossen.
CLOSED (readyState: 3)	Die Verbindung ist geschlossen oder konnte gar nicht erst aufgebaut werden.

Zur besseren Veranschaulichung, was genau passiert und wie die Zustände zusammenhängen und ineinander übergehen, haben wir diese in [Bild 5.1](#) in einem UML-Zustandsdiagramm zusammenhängend dargestellt.

Wie eingangs gesagt, befindet sich Ihr WebSocket, nachdem Sie ihn durch den Aufruf des Konstruktors erzeugt haben, zunächst für eine kurze Zeit im Zustand CONNECTING. Das readyState-Attribut Ihres Objekts hat dabei den Wert 0. Im CONNECTING-Zustand versucht Ihr Objekt, eine Verbindung zum im Konstruktorauftruf angegebenen Server herzustellen. Die Objekterzeugung initiiert also den WebSocket-Handshake, wie wir ihn in [Ab-](#)

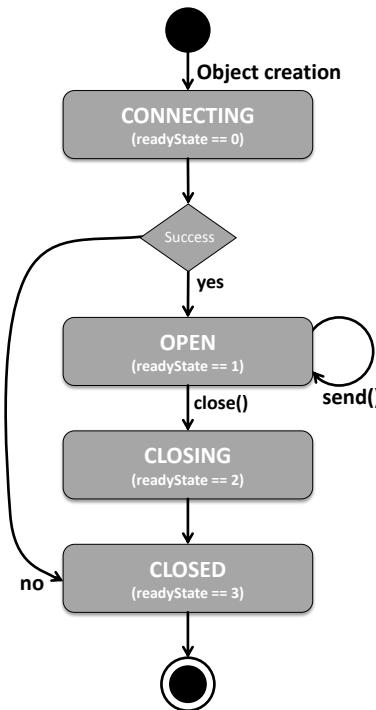


Bild 5.1 UML-Zustandsdiagramm eines WebSocket-Objekts gemäß [Hic12b]

schnitt 4.2 eingeführt und besprochen haben. Gelingt es Ihrem Objekt nicht, eine Verbindung zum Server herzustellen, so geht Ihr Objekt in den Zustand CLOSED über. Das readyState-Attribut bekommt in diesem Zustand den Wert 3 zugewiesen und ist im Grunde zu nichts mehr zu gebrauchen. Andernfalls konnte die Verbindung hergestellt werden, was Ihr Objekt in den Zustand OPEN versetzt.

Im OPEN-Zustand ist Ihr Objekt sende- und empfangsbereit. Erkennen können Sie dies ebenfalls am Wert des readyState-Attributs. Dieses hat dann nämlich den Wert 1 inne. In diesem Zustand verweilt Ihr Objekt typischerweise eine längere Zeit, so lange wie Daten gesendet und/oder empfangen werden sollen. Sofern kein Fehler währenddessen auftritt, bleibt dieser Zustand so lange erhalten, bis einer der beiden Kommunikationspartner nicht mehr möchte und die Verbindung schließt. Auf der Clientseite erfolgt dies durch den Aufruf der einschlägigen Methode `close()`, für die es auf der Serverseite natürlich eine Entsprechung gibt. Kommt es dazu, dass eine bestehende WebSocket-Verbindung geschlossen wird, so geht Ihr Objekt zunächst in den Zustand CLOSING über.

Im CLOSING-Zustand verbleibt Ihr WebSocket-Objekt nur so lange, wie es dauert, den Closing-Handshake zu durchlaufen. Während dieser Zeit hat das readyState-Attribut den Wert 2. Ist der Kanal geschlossen, geht Ihr Objekt in den vierten und letzten Zustand über.

Der CLOSED-Zustand markiert das Lebensende Ihres WebSocket-Objekts. An dieser Stelle können Sie nicht mehr viel damit tun. Wenn Sie eine weitere WebSocket-Verbindung aufbauen und verwenden wollen, müssen Sie dafür eine neue Instanz erzeugen. Dieses Objekt

können Sie also beruhigt durch das Setzen der Referenz auf `null` in die ewigen Jagdgründen verabschieden. Achten Sie beim Erreichen dieses Zustands unbedingt darauf, dass Ihre WebSocket-Objekte nicht mehr referenzierbar sind, damit der belegte Speicher frei gegeben werden kann und keine Speicherlöcher verbleiben.

■ 5.6 Event-Handler

Die soeben besprochenen Zustände stehen im engen Zusammenhang mit den spezifizierten Event Handlern, mit denen auf Zustandsänderungen reagiert werden kann. Ändert sich der Zustand, in dem sich Ihr WebSocket-Objekt befindet, wird ein entsprechender Event ausgelöst, der wiederum den korrespondierenden Event Handler triggert. Überhaupt ist das Programmiermodell stark ereignisgetrieben. [Tabelle 5.4](#) zeigt Ihnen die im Standard definierten Event Handler sowie die korrespondierenden Events.

Tabelle 5.4 Event Handler [[Hic12b](#)]

Event Handler	Event	Beschreibung
<code>onopen</code>	<code>open</code>	Der Event Handler <code>onopen</code> ist mit dem Zustand <code>open</code> assoziiert. Wird ein WebSocket-Objekt in den <code>open</code> -Zustand versetzt, wird der <code>open</code> -Event erzeugt, der von diesem Event Handler verarbeitet wird.
<code>onerror</code>	<code>error</code>	Der Event Handler <code>onerror</code> ist mit dem <code>error</code> -Event gekoppelt. Tritt ein Fehler beim Verbindungsaufbau, während der Datenübertragung oder beim Verbindungsabbau auf, wird dies über diesen Handler signalisiert und die Gründe für den Fehler werden in Ihre Anwendung getragen.
<code>onmessage</code>	<code>message</code>	Trifft eine Nachricht beim Client ein, wird der <code>message</code> -Event ausgelöst. Ihrer Anwendung wird dies mittels des Event Handlers <code>onmessage</code> signalisiert. Gleichzeitig werden die empfangenen Daten zur Weiterverarbeitung an Ihre Anwendung übergeben.
<code>onclose</code>	<code>close</code>	Der Event Handler <code>onclose</code> wird immer dann aktiviert, wenn die WebSocket-Verbindung geschlossen oder gar nicht erst zustande gekommen ist. Auch hier werden über den Event Handler Informationen an Ihre Anwendung übergeben, aus denen Sie die Gründe entnehmen können.

Die Event Handler sind Attribute des WebSocket-Objekts, denen Sie eine Funktion zuweisen können, die dann im Falle des Eintretens des zugehörigen Ereignisses ausgeführt wird. Sie müssen nicht für alle Event Handler eine Funktion angeben. Tatsächlich können Sie sich Anwendungen vorstellen, für die kein einziger der vier Event Handler eine Funktion zugewiesen bekommen muss. Dies trifft auf Anwendungen zu, die über einen WebSocket schlicht Daten vom Client zum Server schicken sollen wobei die umgekehrte Kommunikationsrichtung nicht benötigt wird. Beispiele für derartige Anwendungen lassen sich in den Settings finden, die den Client z.B. als eine Art Sensor nutzen und viele Daten mit

einer geringen Verzögerung kontinuierlich dem Server bereitstellen. Hierunter fallen z. B. Beobachtungen von Probanden im Rahmen von Usability-Tests, deren Interaktion mit einer Webseite über Mausbewegungsbeobachtungen sowie Video- und Audioaufzeichnungen erfasst werden. In der Regel sollte es aber zum guten Programmierstil gehören, für alle Eventualitäten entsprechende Codezeilen vorzusehen, die mit Fehlern umgehen und zum Schluss das Aufräumen übernehmen.

Wenn Ihr Programm direkt nach Etablierung der WebSocket-Verbindung etwas tun soll, dann können Sie das über den onopen-Event-Handler realisieren. Wenn Sie gerade keine Idee haben, was das sein könnte, möchten wir Ihnen mit dem folgenden Beispiel einige Anregungen geben.

Listing 5.2 Beispiel zur Verwendung des onopen-Event-Handler

```
1 ws.onopen = function() {
2     console.log('WebSocket-Verbindung aufgebaut.');
3     turnConnectionIndicatorOn();
4     ws.send('Add: ISIN=DE0008469008');
5 };
```

Sie könnten z. B. darüber Buch führen wollen, was alles mit Ihrem WebSocket-Objekt so passiert ist. Das ist sicherlich am einfachsten durch eine Logausgabe in der Konsole getan. Vielleicht haben Sie aber auch einen visuellen Indikator in die Webseite integriert, der dem Benutzer über den Zustand des Kommunikationskanals informiert. Diesen Indikator sollten Sie dann in der onopen-Funktion so einstellen, dass er fortan eine geöffnete Verbindung anzeigt. Dann kann es sein, dass Sie in Ihrer Anwendung umgehend nachdem die Verbindung aufgebaut ist, ein bestimmtes Datum vom Server abrufen möchten. Sagen wir, Sie entwickeln gerade an einem Frontend für die Darstellung von Finanzinstrumenten in Realzeit. Nachdem Sie die WebSocket-Verbindung aufgebaut haben, wollen Sie immer die Daten für den DAX vom Server gestreamt haben. Daher setzen Sie unmittelbar eine erste Nachricht an den Server ab, die über ein proprietäres Protokoll die Daten des DAX anfragt. Hier soll das Protokoll durch das Add-Kommando ein Finanzinstrument der Beobachtungsliste des Clients hinzufügen. Angegeben wird das Wertpapier durch die *International Securities Identification Number* (ISIN), die über eine zwölfstellige Buchstaben-Zahlen-Kombination eine Identifikation für ein Wertpapier darstellt.

Fehler können immer auftreten. Bei der Programmierung verteilter Anwendungen trifft dies im besonderen Maße zu, da es eine Vielzahl von Fehlerquellen gibt, auf die Sie manchmal nicht einmal mit Ihrem eigenen Code Einfluss nehmen können. Denken Sie z. B. dabei an die Netzwerkverbindung. Hier bleibt Ihnen also keine andere Wahl, als mit diesem Fehler umzugehen.

Listing 5.3 Ausgabe eines Fehlercodes und der Fehlerbegründung in der Konsole

```
1 ws.onerror = function(event) {
2     var reason = event.reason;
3     var code = event.code;
4     console.log('Ein WebSocket-Fehler ist aufgetreten: ' +
5                 reason + '(' + code + ')');
6 };
```

Im Beispiel wird auf den Fehlercode und die formulierte Fehlerbegründung zugegriffen, die zum Fehlerfall geführt haben, und schlicht in der Konsole ausgegeben. Mehr zu den definierten Fehlercodes und Fehlerbegründungen haben wir in [Abschnitt 4.7.3](#) besprochen.

Der Event Handler onmessage wird immer dann aktiviert, wenn eine Nachricht eingeht. Wenn Sie z. B. vorhaben, Ihre auf HTML5 basierenden Präsentationsfolien via WebSockets vor- und zurückzublättern, dann könnten Sie das durch das Absetzen von next- bzw. prev-Kommandos. Sie müssen dann im onmessage-Event-Handler Ihres WebSocket-Objekts eine Funktion registrieren, die eingehende Nachrichten nach den entsprechenden Kommandos absucht (siehe [Listing 5.4](#)).

Listing 5.4 Registrieren einer Funktion im onmessage-Event-Handler

```

1 ws.onmessage = function(message) {
2     var data = message.data;
3     if(data == 'next') {
4         nextSlide();
5     }
6     else if(data == 'prev') {
7         prevSlide();
8     }
9     else {
10        // Ignoriere unbekannte Kommandos
11    }
12 };

```

Ein WebSocket-Objekt kann über zwei Wege in den CLOSED-Zustand gelangen (siehe [Abschnitt 5.4](#) und [Bild 5.1](#)). In beiden Fällen wird der onclose-Event-Handler aktiviert. Haben Sie dem entsprechenden Attribut Ihres WebSocket-Objekts eine Funktion zugewiesen, kommt diese dann zur Ausführung (siehe [Listing 5.5](#)).

Listing 5.5 Registrieren einer Funktion im onclose Event Handler

```

1 ws.onclose = function(event) {
2     if(this.readyState == 2) {
3         console.log('Schliesse die Verbindung...');
4         console.log('Die Verbindung durchlaeuft den Closing-
5             Handshake');
6     }
7     else if(this.readyState == 3) {
8         console.log('Verbindung geschlossen...');
9         console.log('Die Verbindung wurde geschlossen oder konnte
10            nicht aufgebaut werden.');
11    }
12    else {
13        console.log('Verbindung geschlossen...');
14        console.log('Nicht behandelter readyState: ' +
15            this.readyState);
16    }
17 };

```

Alternativ können Event Handler auch mit der `addEventListener()`-Methode definiert werden. In Anlehnung an das zuvor in [Listing 5.2](#) angegebene Beispiel zum `onopen`-Event Handler können Sie das gleiche Resultat mit dem Codeschnipsel aus [Listing 5.6](#) erzielen.

Listing 5.6 Definition eines Event Handlers mit der `addEventListener()`-Methode

```
1 ws.addEventListener("open", function() {
2     console.log('Die WebSocket-Verbindung wurde aufgebaut.');
3     turnConnectionIndicatorOn();
4     ws.send('Add: ISIN=DE0008469008');
5 });
```

Diese alternative Definition von Event Handler-Funktionen hat dann Vorteile, wenn Sie einem Event mehr als *einen* Handler zuweisen wollen. Dies können Sie nur mit der Verwendung der `addEventListener()`-Methode realisieren.

■ 5.7 Erstes vollständiges Programmchen

So, an dieser Stelle wissen Sie schon genug, um einen ersten WebSocket-Client gemäß der W3C-API in JavaScript implementieren zu können. Probieren Sie es doch einfach mal aus! Lesen Sie nicht weiter und versuchen Sie ein vollständiges JavaScript-Programm anzugeben, das eine WebSocket-Verbindung zum Echo-Server aufbaut, der unter der URL

ws://echo.websocket.org/

verfügbar ist. Wenn die Verbindung aufgebaut werden konnte, soll Ihr Programm den String '*'Hallo WebSocket Endpoint!'*' an den Echo-Server schicken. Der Echo-Server würde einer derartigen Kommunikation mit einer einfachen Antwort mit dem gleichen Inhalt begegnen. Ihr Programm soll diese Antwort entgegennehmen und in der JavaScript-Konsole ausgeben. Danach sollen Sie die Verbindung wieder schließen. Falls es zu Fehlern kommt, sollen Sie diese mit Ihrem Programm nicht ignorieren, sondern vielmehr die Gründe dafür ebenfalls in die JavaScript-Konsole ausgeben. Also dann mal los!

Das nachfolgende [Listing 5.7](#) zeigt eine mögliche Implementierung für den beschriebenen Echo-Client. Hierbei werden die Kernbestandteile der W3C WebSocket-API, die wir bisher besprochen haben, an einem Beispiel zusammenhängend dargestellt und verdeutlicht.

Listing 5.7 Vollständiger WebSocket-basierender Echo-Client

```
1 var ws = new WebSocket('ws://echo.websocket.org/');
2
3 ws.onopen = function() {
4     console.log('WebSocket-Verbindung aufgebaut.');
5     ws.send ('Hallo WebSocket Endpoint!');
6     console.log('Uebertragene Nachricht: Hallo WebSocket Endpoint!');
7 };
8
9 ws.onmessage = function(message) {
```

```

10    console.log('Der Server sagt: ' + message.data);
11    ws.close();
12  };
13
14 ws.onclose = function(event) {
15   console.log('Der WebSocket wurde geschlossen oder konnte
16     nicht aufgebaut werden.');
17 };
18 ws.onerror = function(event) {
19   console.log('Mit dem WebSocket ist etwas schiefgelaufen!');
20   console.log('Fehlermeldung: ' + event.reason +
21     ' (' + event.code + ')');
21 };

```

Schauen wir uns an, was im dargestellten Listing genau passiert. Zunächst haben wir ein WebSocket-Objekt instanziert ([Zeile 1](#)), das bei der Objekterzeugung eine Verbindung zum Server aufbauen soll, dessen URL im Argument des Konstruktors angegeben ist. Das passt dann auch, allerdings ohne die weitere Verarbeitung des Programms zu blockieren. Während also der Verbindungsaufbau innerhalb der Objekterzeugung noch im vollen Gange ist, wird der Rest des Programms asynchron (oder parallel) weiterverarbeitet. Dabei werden vier Funktionen den vier spezifizierten Event Handlern zugewiesen; jeweils eine Funktion, die angibt, was im Falle eines erfolgreichen Verbindungsbaus (onopen, [Zeile 3](#)), beim Eintreffen einer Nachricht vom Server (onmessage, [Zeile 9](#)), beim Eintreten eines Fehlers (onerror, [Zeile 18](#)) und bei dem Schließen der Verbindung (onclose, [Zeile 14](#)) ausgeführt werden soll. Wie es jetzt mit der Ausführung weitergeht, hängt davon ab, ob der Verbindungsbaus zum Server erfolgreich ist oder nicht. Wenn Sie sich an das UML-Zustandsdiagramm in [Bild 5.1](#) zurück erinnern, dann stehen wir jetzt genau am Scheideweg, der uns entweder zum OPEN- oder CLOSED-Zustand führt. Es sind also zwei Pfade durch das Programm möglich:

1. Schlägt der Verbindungsbaus fehl, geht das Objekt ws in den CLOSED-Zustand über, wobei dies mit dem Absetzen eines close-Events einhergeht, was wiederum bewirkt, dass der onclose-Event Handler aktiviert wird. Dieser Programmdurchlauf erzeugt folglich nur eine einzige Ausgabe in der Konsole, und zwar die, die im onclose Event Handler angegeben ist: 'Der WebSocket wurde geschlossen oder konnte nicht aufgebaut werden.'
2. Konnte der Verbindungsbaus erfolgreich durchgeführt werden, geht das Objekt ws in den OPEN-Zustand über, wobei dies mit dem Absetzen eines OPEN-Events einhergeht, was wiederum bewirkt, dass der onopen-Event Handler aktiviert wird. Die für diesen Handler registrierte Funktion erzeugt zunächst eine informative Ausgabe in der Konsole über den erfolgreichen Verbindungsbaus. Danach setzt sie eine Nachricht an den Server ab, die 'Hallo WebSocket Endpoint!' zum Inhalt hat. Schließlich wird noch in der Konsole protokolliert, dass diese Nachricht an den Server abgeschickt wurde. Was als Nächstes ausgeführt wird, hängt davon ab, was passiert. Da es sich beim Server um einen Echo-Server handelt, wird dieser auf die an ihn gerichtete Nachricht mit einer Antwort reagieren, die exakt den gleichen Inhalt hat wie die des Clients. Es wird also eine Nachricht beim Client eintreffen, die einen message-Event auslöst, der wiederum den onmessage-Event Handler aktiviert. Die diesem Event Handler zugewiesene Funktion kommt zur Ausführung. Sie erzeugt eine Aussage in der Konsole, die

den Inhalt der Nachricht vom Server mit einem Präfix ausgibt, in diesem Beispiel 'Der Server sagt: Hallo WebSocket Endpoint!'. Danach schließt der Client die Verbindung durch den Aufruf der `close()`-Methode. Dadurch wird ein `close`-Event abgesetzt, der den `onclose`-Event Handler aktiviert, wodurch die Ausgabe 'Der WebSocket wurde geschlossen oder konnte nicht aufgebaut werden.' in der Konsole erscheint.

Da es den im Beispiel angeführten WebSocket-Echo-Server tatsächlich gibt, können Sie dieses Beispiel direkt ausprobieren, ohne sich an dieser Stelle schon Gedanken zur Implementierung der Serverseite machen zu müssen. Mit den in vielen Browsern enthaltenen JavaScript-Konsolen können Sie sich außerdem die beiden beschriebenen Ausführungspfade verdeutlichen. In [Bild 5.2](#) geben wir Ihnen ein Beispiel.

The image shows two vertically stacked screenshots of the Chrome Developer Tools Console tab. Both windows have the title 'Developer Tools - chrome://newtab/'. The top window shows the creation of a WebSocket object with an incorrect URL ('ws://echo.websocket.org/'). The bottom window shows the creation of a WebSocket object with the correct URL ('ws://echo.websocket.org/'), followed by its closure and subsequent state change.

```

Developer Tools - chrome://newtab/
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> var ws;
undefined
> ws = new WebSocket('ws://echo.websocket.org/');
▶ WebSocket {binaryType: "blob", extensions: "", protocol: "", onclose: null, onerror: null...}
> ws.readyState
3
>

Developer Tools - chrome://newtab/
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> var ws;
undefined
> ws = new WebSocket('ws://echo.websocket.org/');
▶ WebSocket {binaryType: "blob", extensions: "", protocol: "", onclose: null, onerror: null...}
> ws.readyState
1
> ws.close();
undefined
> ws.readyState
3
>

```

Bild 5.2 Zustandsübergänge bei der Objekterzeugung (JavaScript-Konsole von Chrome)

Im oberen Chrome-Browserfenster hat sich im Konstruktorgument ein Tippfehler eingeschlichen. Aus diesem Grund kann keine Verbindung zum angegebenen Server aufgebaut werden (dieser existiert so nicht). Daher ist das erzeugte WebSocket-Objekt anschließend im CLOSED-Zustand, was dem `readyState` mit dem Wert 3 entspricht. Im unteren Chrome-Fenster ist die URL im Konstruktorgument korrekt eingegeben. Eine Verbindung konnte folglich hergestellt werden und das WebSocket-Objekt befindet sich im OPEN-Zustand mit dem `readyState=1`. Das Aufrufen der `close()`-Methode beendet die Verbindung und versetzt das Objekt in den `readyState=3`.

■ 5.8 Attribute

Ein WebSocket-Objekt verfügt neben dem readyState und den Event Handlern über weitere Attribute, auf die Sie zurückgreifen können, um in Ihren Programmen noch auf weitere Statusinformationen zur WebSocket-Verbindung zugreifen zu können. Diese haben wir aus dem W3C-Standard in die [Tabelle 5.5](#) überführt.

Tabelle 5.5 Attribute eines WebSockets (ohne readyState und Event Handler) [[Hic12b](#)]

Attribute	Datentyp	Beschreibung
binaryType	DOMString	Gibt den Typ der Binärdaten an. Kann entweder den Wert Blob oder ArrayBuffer annehmen.
bufferedAmount (nur lesender Zugriff)	long	Gibt die Anzahl an Bytes an, die noch in der Warteschlange stehen und nicht versendet wurden. Dieser Wert wird nicht auf 0 gesetzt, wenn die Verbindung geschlossen wird. So können Sie überprüfen, ob alle abgesetzten Daten auch ihre Reise an das Ziel angetreten haben.
extensions (nur lesender Zugriff)	DOMString	Eine Liste mit den Erweiterungen, die vom Server ausgesucht wurden.
protocol (nur lesender Zugriff)	DOMString	Das aktuell benutzte Subprotokoll, das vom Server ausgewählt wurde.
url (nur lesender Zugriff)	DOMString	Die URL der Verbindung.

Das url-Attribut Ihres Objekts gibt die URL des WebSocket-Servers wieder, mit dem die Verbindung etabliert wurde. Wollen Sie sich diese in der Konsole ausgeben lassen, ist dies mit der folgenden Zeile getan:

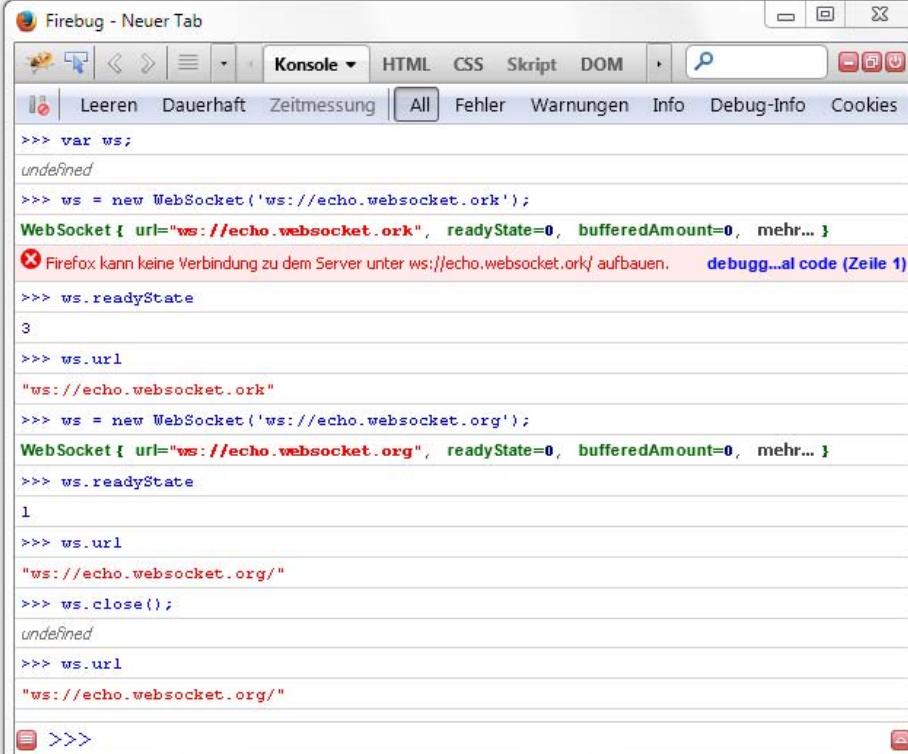
```
console.log(
  'WebSocket-Verbindung aufgebaut mit ' + ws.url + '.');
```

Eine elegantere Lösung würde innerhalb des onopen-Event Handlers die URL des verbundenen Servers in eine in der Webseite dafür vorgesehene Statuszeile einfügen. Das Beispiel in [Listing 5.8](#) zeigt das Prinzip, wobei angenommen wird, dass das HTML-Element mittels der eindeutigen ID 'server-status-line' angesprochen werden kann. Das url-Attribut wird auf den Wert gesetzt, der bei der Objekterzeugung im Konstruktorkonzept angegeben wird. Der Wert verändert sich über die gesamte Lebensdauer des Objekts nicht mehr. Selbst im CLOSED-Zustand ist der Wert noch zugreifbar.

Listing 5.8 Beispiel für die Verwendung des url-Attributs innerhalb eines onopen-Event Handlers

```
1 ws.onopen = function() {
2   // Hier stehen nützliche initiale Einstellungen...
3   document.getElementById('server-status-line').innerHTML=this.url;
4 };
```

Wir haben das in der JavaScript-Konsole des Firebug für Sie nachvollziehbar gemacht (siehe Bild 5.3).



```

Firebug - Neuer Tab
Konsole ▾ HTML CSS Skript DOM ⌂ 🔍
Leeren Dauerhaft Zeitmessung All Fehler Warnungen Info Debug-Info Cookies
>>> var ws;
undefined
>>> ws = new WebSocket('ws://echo.websocket.org');
WebSocket { url="ws://echo.websocket.org", readyState=0, bufferedAmount=0, mehr... }
✖ Firefox kann keine Verbindung zu dem Server unter ws://echo.websocket.org/ aufbauen. debug...al code (Zeile 1)
>>> ws.readyState
3
>>> ws.url
"ws://echo.websocket.org"
>>> ws = new WebSocket('ws://echo.websocket.org');
WebSocket { url="ws://echo.websocket.org", readyState=0, bufferedAmount=0, mehr... }
>>> ws.readyState
1
>>> ws.url
"ws://echo.websocket.org/"
>>> ws.close();
undefined
>>> ws.url
"ws://echo.websocket.org/"

>>>

```

Bild 5.3 Lebensdauer des url-Attributs (JavaScript-Konsole von Firebug)

Zunächst erzeugen wir ein WebSocket-Objekt und damit eine WebSocket-Verbindung zu einem nicht existenten Server mit der falschen URL:

ws://echo.websocket.org/

Daher geht das Objekt in den CLOSED-Zustand (readyState=3). In diesem Zustand ist das url-Attribut noch auf dem Wert der fälschlich eingegebenen URL. Im Anschluss erzeugen wir ein neues Objekt, diesmal mit der korrekten URL:

ws://echo.websocket.org/

Da die Verbindung zu diesem Server aufgebaut werden kann, befindet sich das Objekt im OPEN-Zustand (readyState=1). In diesem Zustand hat das url-Attribut als Wert die URL des verbundenen Servers inne. Schließen wir nun die Verbindung zum Server durch Aufrufen der `close()`-Methode, wechselt das Objekt in den CLOSED-Zustand (readyState=3). Wie schon zuvor beim Vertipper, bleibt der Wert des url-Attributs erhalten und zeigt die URL an, für die das Objekt erzeugt wurde.

Die Beobachtungen zur Lebensdauer von Attributwerten gelten auch für alle anderen Attribute aus [Tabelle 5.5](#), für die wir im Folgenden noch jeweils ein Beispiel mit Ihnen durchgehen möchten.

Sie werden sich erinnern, dass wir in [Abschnitt 4.2](#) darüber gesprochen haben, dass Sie beim Verbindungsauftakt in der Handshake-Anfrage ein oder mehrere Subprotokolle angeben können. Der Server sucht sich dann ein passendes aus. Ist kein passendes darunter, muss er die Verbindungsanfrage unterbinden. In [Abschnitt 5.4](#) sind uns die Subprotokolle dann wieder begegnet. Hier konnten wir diese nämlich als optionale Parameter im Konstruktor angeben. Haben Sie Ihren Client nun so implementiert, dass er mit mehreren Subprotokollen umgehen kann und Sie diese dem Server zur Wahl stellen, müssen Sie wissen, für welches er sich entschieden hat. Das können Sie aus dem `protocol`-Attribut auslesen.

```
console.log(  
    'Der WebSocket-Server nutzt das Subprotokoll ' + ws.protocol + '.');
```

Das Protokoll sieht von Haus aus Erweiterungen vor. Erweiterungen können entweder öffentlich gemacht werden oder sie bleiben eine reine private Angelegenheit. Im ersten Fall werden sie zu RFC-Standards. Im zweiten Fall bleiben sie proprietäre Lösungen. Aktuell gibt es noch keine standardisierten Erweiterungen. In den verschiedenen Vorläufer-Entwürfen zum RFC 6455 war eine Zeit lang eine Erweiterung mit dem Namen *deflate-stream* enthalten. Diese Erweiterung hat die Kompression der Daten zum Gegenstand, hat es aber aufgrund vieler konzeptioneller Schwächen schließlich nicht in den Standard geschafft. Außer einer Nennung in einem Beispiel ist nichts mehr von *deflate-stream* im RFC 6455 übrig geblieben. Zwei andere Erweiterungen befinden sich gerade innerhalb der IETF-Arbeitsgruppe HyBi als Drafts in Bearbeitung. Die eine Erweiterung trägt den Namen *WebSocket Per-frame Compression* [[Yos12](#)] und nähert sich dem Thema Datenkompression von einer anderen Seite. Zunächst legt sich diese Erweiterung nicht auf einen spezifischen Kompressionsalgorithmus fest. Dieser kann ausgehandelt werden. Ein einziger Algorithmus ist in der Erweiterung allerdings als kleinster Nenner vorgeschrieben und das ist der altbekannte Deflate-Algorithmus. Komprimiert werden dann die Nutzdaten in den Datenframes. Die andere Erweiterung mit dem Namen *Multiplexing Extension for WebSockets* [[TY13](#)] soll es ermöglichen, mehrere logische WebSocket-Kanäle über eine tatsächlich physisch vorhandene WebSocket-Verbindung zu schleusen. Mit dieser Erweiterung wird möglichen Skalierungsproblemen entgegengewirkt, die entstehen können, wenn ein Client viele unterschiedliche WebSocket-Verbindungen (z. B. in mehreren Tabs) zum gleichen WebSocket-Server aufgebaut hat. Da sich diese WebSocket-Erweiterungen noch stark im Fluss befinden, werden Sie in der überwiegenden Anzahl an Fällen einen leeren String im `extensions`-Attribut in den Headern der WebSocket-Handshakes vorfinden.

Es kann nicht immer alles prompt gesendet werden, so wie Sie es mit der `send()`-Methode absetzen. Die Gründe dafür können vielfältig sein. Der häufigste ist sicher eine begrenzte Datentransferkapazität. Um dies abzufedern, sieht der Browser einen Buffer vor, in dem noch nicht gesendete Daten eingereiht werden. Um herauszufinden, wie viele Bytes sich aktuell im Buffer befinden, steht das Attribut `bufferedAmount` bereit. Wenn wir beim Beispiel bleiben, dass eine Menge von Daten an den Server gesendet werden und diese nicht umgehend auf den Weg gebracht werden können, könnten Sie wie folgt mit der Situation umgehen, um Probleme durch unkontrollierte Überlastungen zu vermeiden.

In dem Codeschnipsel aus [Listing 5.9](#) senden Sie in regelmäßigen Intervallen Mausbewegungsdaten an einen Server zur Unterstützung einer Usability-Studie. Alle 50 Millisekunden wird eine Funktion aufgerufen, die die Mauspositionsdaten ermittelt und an den Server schickt, dies aber nur tut, wenn keine „älteren“ Daten mehr im Buffer stehen und auf ihre Abreise warten. Mit dieser einfachen Methode passt sich die zeitliche Auflösung der Mausbewegungen automatisch an die Datentransferkapazitäten der Anwendung – in diesem Fall einer Usability-Testumgebung, die wir anhand eines vollständigen Beispiels in [Abschnitt 8.3](#) besprechen – an.

Listing 5.9 Beispiel zur Vermeidung von unkontrollierten Überlastungen durch Datenmengen

```
1 var ws = new WebSocket('ws://usability.example.com/updates');
2 ws.onopen = function() {
3     setInterval(function() {
4         if(ws.bufferedAmount == 0)
5             ws.send(getMousePositions());
6     }, 50);
7 };
```

Wie wir uns im nachfolgenden [Abschnitt 5.9](#) noch näher anschauen werden, sieht die WebSocket-API drei verschiedene Formen vor, wie binäre Daten an die `send()`-Methode übergeben werden können: entweder als Blob (Abkürzung für *Binary Large Object*), als ArrayBuffer oder als ArrayBufferView. Der letztgenannte Datentyp entspricht dabei im Wesentlichen einem ArrayBuffer, mit dem Unterschied, dass bei einem ArrayBufferView nur ein bestimmter Teil der vorliegenden Daten selektiert wird. Durch Auslesen des `binaryType`-Attributs können Sie feststellen, in welcher der beiden Formen die binären Daten eingetroffen sind. Initial steht der Wert auf `Blob`. Werden nur Strings übertragen, ändert sich der Wert des `binaryType`-Attributs nicht und verbleibt auf dem Initialwert (siehe [Bild 5.4](#)).

Werden Binärdaten empfangen, wird vom Browser auch auf Basis des `binaryType`-Attributs entschieden, ob auf die Daten als Blob oder ArrayBuffer zugegriffen werden kann. Genau diese zwei Werte dürfen Sie diesem Attribut zuweisen. Es kommen aber auch noch andere Faktoren bei der Entscheidungsfindung des Browsers zum Tragen. Darunter zählt z. B. die Datenmenge. Da ein ArrayBuffer im Hauptspeicher gehalten wird, sollte dieser nur bei kleineren Datenmengen verwendet werden. Größere Datenmengen sollten unbedingt als Blob temporär auf dem persistenten Speicher ausgelagert werden.

■ 5.9 Datenübertragung

Zum Senden von Daten ist die `send()`-Methode im W3C-Standard definiert [[Hic12b](#)]. Wenn Sie sich den Standard genauer anschauen, werden Sie feststellen, dass sie vierfach überladen ist und die folgenden Argumente akzeptiert:

```
void send(DOMString data);
void send(Blob data);
void send(ArrayBuffer data);
```

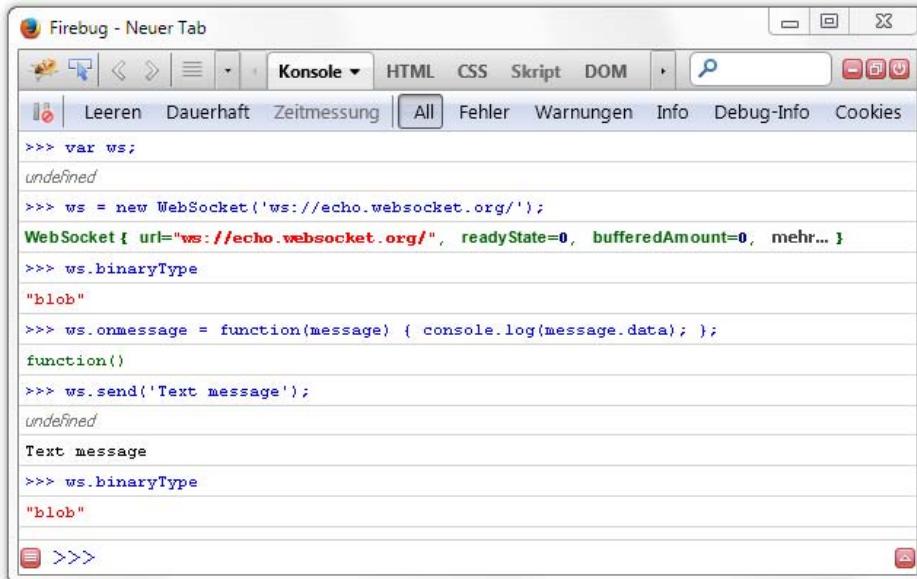


Bild 5.4 Das Attribut `binaryType` wird nur gesetzt, wenn binäre Daten empfangen werden.

```
void send(ArrayBufferView data);
```

Genau eine dieser vier `send()`-Methoden haben wir bereits häufiger in unseren Beispielen verwendet. Bisher haben wir jedes Mal die erstgenannte Alternative aufgerufen. Diese ist mit der textbasierten Übertragung einfacher Strings betraut. Die verbleibenden drei Varianten dienen allesamt der binären Datenübertragung, die wir bisher vernachlässigt haben. In den folgenden zwei Unterkapiteln wollen wir uns nun noch mal genauer mit den beiden bereitstehenden Übertragungsmodi auseinandersetzen.

5.9.1 Übertragung textbasierter Daten

Ist das Argument der `send()`-Methode ein String, wird er als solcher übertragen. Dazu wird der String in seine Einzelteile zerlegt und daraus eine Sequenz von UTF-8-Zeichen gebildet. Anschließend werden die UTF-8-Zeichen in einen WebSocket-Frame mit dem Text-Opcode eingefügt und gesendet. Der Aufruf

```
ws.send('a simple text message');
```

erzeugt auf der Leitung also folgenden WebSocket-Frame (siehe dazu auch [Abschnitt 4.3](#)). Bei der Übertragung dürfen Sie nicht außer Acht lassen, dass Ihre Daten evtl. zunächst in einen browser-internen Buffer gelangen und dadurch verzögert auf die Leitung kommen.



Bild 5.5 WebSocket-Frame mit Textdaten: „a simple text message“ (maskiert, da der Frame vom Client zum Server gesendet wurde)

5.9.2 Übertragung binärer Daten

Mit binären Daten kann auf zwei verschiedene Arten umgegangen werden. Welche der beiden Möglichkeiten Sie bzw. der Browser wann verwenden, hängt über den Daumen gepeilt von der Datenmenge ab. Handelt es sich um viele Daten, ist das Verwenden eines BLOB ratsam. Dies können Sie quasi schon aus dem Akronym ablesen (*Binary Large Object*). Andernfalls stehen Ihnen ArrayBuffer- oder ArrayBufferView-Objekte zur Verfügung. Kleinere Datenstrukturen wie z. B. Array-Variablen, die ohnehin im Hauptspeicher Ihrer Anwendung gehalten werden, können Sie über ArrayBuffer abwickeln. Blob-Objekte sollten Sie immer dann verwenden, wenn die Daten potenziell groß werden können. Beispiele hierfür sind Dateiinhalte.

Im W3C-Standard findet sich eine ähnliche Empfehlung für die Browserhersteller wieder. Auf Basis dieser Empfehlungen sollen von den Browsern entsprechende Strategien entwickelt werden, wie mit den binären Daten umgegangen werden soll. Kleinere Datenmengen können im Hauptspeicher verbleiben. Größere Datenmenge sollten als Ganzes auf die Festplatte ausgelagert werden, um den Hauptspeicher nicht über die Maße zu beanspruchen. Wo genau die Grenzen liegen und wie genau damit umgegangen werden soll, schreibt der Standard nicht vor. Diesen Freiheitsgrad sollten Sie im Hinterkopf behalten, da dieser zu unterschiedlichen Implementierungen und damit auch zu unterschiedlichem Verhalten der Browser führt. Sie sollten daher immer prüfen, in welcher Form Ihnen die Binärdaten bereitgestellt werden, bevor Sie auf die empfangenen Daten zugreifen. Auch wenn Sie zuvor durch das Setzen des `binaryType`-Attributs auf den Wert `ArrayBuffer` den Wunsch ausgedrückt haben, die Daten als `ArrayBuffer` übergeben zu bekommen, heißt es nicht, dass Sie sie tatsächlich auch in dieser Form erhalten. Der Browser kann Ihre Präferenz überstimmen und sich für den Blob-Datentyp entscheiden. Eine Überprüfung auf den Binärdatentyp hat folgenden prinzipiellen Aufbau (siehe [Listing 5.10](#)):

Listing 5.10 Prinzipieller Aufbau einer Überprüfung auf den Binärdatentyp

```
1 ws.onmessage = function(event) {
2     if(event.data instanceof ArrayBuffer) {
```

```

3      // Verarbeitung der ArrayBuffer-Daten
4  }
5  else if(event.data instanceof Blob) {
6      // Verarbeitung der BLOB-Daten
7  }
8  else {
9      onclose.log("Fehler: Es werden nur Binaerdaten
10         unterstuetzt!");
11 }

```

Ist das Argument der `send()`-Methode ein Blob-Objekt, werden die rohen Bytes in einen WebSocket-Frame mit dem Binary Frame Opcode versehen und verschickt. Das folgende Beispiel zeigt, wie Sie eine Datei vom lokalen Dateisystem via WebSockets als Blob an den Server schicken können (siehe [Listing 5.11](#)):

Listing 5.11 Verschicken einer Datei als BLOB-Objekt

```

1 var file = document.querySelector('input[type="file"]').files[0];
2 ws.send(file);

```

Für große Dateien hat ein derartiges Vorgehen wie gesagt den Vorteil, dass die Daten vom Browser nicht vollständig in den Hauptspeicher geladen werden müssen. Stattdessen werden die Daten von der Festplatte häppchenweise eingelesen und an den Empfänger gestreamt. Dieser wiederum kann auch die Tatsache nutzen, dass es sich um einen BLOB handelt, und die darüber bereitgestellten Daten ebenfalls direkt auf die Festplatte schreiben; auch hier wieder aus dem Grund, nicht alle Daten in den Hauptspeicher schreiben zu müssen.

Die verbleibenden zwei Varianten funktionieren äquivalent, mit dem Unterschied, dass geringe Datenmengen als ArrayBuffer- oder ArrayBufferView-Objekt der `send()`-Methode als Argument übergeben werden. Das folgende Beispiel zeigt, wie Sie die Pixelinformationen aus einem canvas entnehmen, daraus ein ArrayBuffer-Objekt erzeugen und dieses dann an den Server schicken können:

Listing 5.12 Verschicken von Pixelinformationen eines Canvas als ArrayBuffer-Objekt

```

1 var img = canvas_context.getImageData(0, 0, 400, 320);
2 var binary = new Uint8Array(img.data.length);
3 for (var i=0; i<img.data.length; i++) {
4     binary[i] = img.data[i];
5 }
6 ws.send(binary.buffer);

```

Die Pixelinformation wird von der `getImageData()`-Methode als `ImageData`-Objekt zurückgegeben. Das `ImageData`-Objekt wiederum verwaltet die Pixel in einem eindimensionalen Array mit dem Namen `data`. Die Reihenfolge der Pixel geht von links nach rechts und von Zeile zu Zeile, beginnend in der linken oberen Ecke. Jedes Pixel wird dabei durch vier Elemente im Array repräsentiert, die die Farbe durch den Anteil an Rot, Grün und Blau sowie die Deckkraft (Alphakanal) bestimmen. Jeder Wert muss dabei im Bereich von 0 bis 255 liegen.

■ 5.10 Verbindungsabbau

Beim Schließen des WebSocket-Kommunikationskanals muss unterschieden werden, wer von beiden Kommunikationspartnern die Verbindung abbaut. Ist es der Client selbst, so steht ihm dafür die `close()`-Methode bereit (siehe [Abschnitt 5.10.1](#)). Wird der Verbindungsabbau von der Serverseite initiiert, bekommt der Client dies durch das Auslösen des `close`-Events angezeigt (siehe [Abschnitt 5.10.2](#)).

5.10.1 Verbindung beenden

Zum Verbindungsabbau stellt die W3C-WebSocket-API dem Webbrowser eine `close()`-Methode bereit. Sie ist wie folgt anhand der WebIDL [[McC12](#)] spezifiziert:

```
void close([Clamp] optional unsigned short code,
          optional DOMString reason);
```

Die `close()`-Methode hat keinen Rückgabewert und kann parameterlos aufgerufen werden. Dies bewirkt, dass der Browser einen Close-Frame an den Server schickt, der den Statuscode 1000 mit der Bedeutung `Normal Closure` enthält (siehe [Tabelle 4.1](#)).

Entstehen in Ihren Programmen im Browser unerwartete Situationen und müssen Sie dadurch die Verbindung abbrechen, können Sie der `close()`-Methode zusätzliche Parameter übergeben, mit denen Sie der Gegenseite signalisieren können, was zum vorzeitigen Abbruch der Verbindung geführt hat. Kombinationsmöglichkeiten, die sich aus der WebIDL-Spezifikation der Methode ergeben, sind die folgenden:

```
void close(unsigned short code);
void close(unsigned short code, DOMString reason);
```

Mit dem ganzzahligen Parameter `code` können Sie einen Statuscode angeben, der sich vom Standardcode 1000 unterscheidet. Da der Parameter in der Spezifikation mit `[Clamp]` markiert ist, sind nicht alle Zahlenwerte des Datentyps als Werte zulässig. Mögliche Werte sind weiterhin die 1000 und die Zahlen von 3000 bis 4999. Genau in diesem Bereich befinden sich gemäß RFC 6455 die Statuscodes, die für Frameworks, Libraries und Anwendungen vorgesehen sind, und zwar die standardisierten Codes im Bereich von 3000 bis 3999 und die Codes von 4000 bis 4999 für proprietär festgelegte Codes (siehe [Abschnitt 4.7.3](#)). Geben Sie einen anderen Zahlenwert als ersten Parameter ein, wird eine `InvalidAccessError`-Exception ausgelöst.

Mit dem String `reason` können Sie zusätzliche Informationen zum Verbindungsabbruch an den Server senden. Der String ist allerdings längenbeschränkt. Im W3C-Standard ist definiert, dass dieser maximal 123 Bytes betragen darf. Wird dieses Längenlimit überschritten, wird eine `SyntaxError`-Exception ausgelöst.

5.10.2 Close-Event

Eine Verbindung kann ordnungsgemäß, d. h. durch das Senden eines Close-Frames vom Server oder Client, geschlossen werden. Wenn der Server die Verbindung trennt, wird dies

der Anwendung im Browser durch einen entsprechenden close-Event signalisiert. Dieser stößt die Funktionen an, die sich am onclose-Event-Handler registriert haben. Dem onclose-Handler wird ein event-Objekt als Parameter mitgegeben, das die in [Tabelle 5.6](#) aufgelisteten Attribute enthält. Diese entsprechen im Großen und Ganzen den Informationen, die im empfangenen Close-Frame enthalten sind.

Tabelle 5.6 Close-Event Attribute [[Moz14](#)]

Attribut	Datentyp	Beschreibung
code	long	Gibt den Close-Code wieder, der durch den Server bestimmt wird.
reason	DOMString	Eine Textnachricht, die den Grund für die Beendigung angibt. Diese Nachricht ist von Servern bzw. Subprotokollen selbst bestimmbar.
wasClean	boolean	Gibt an, ob eine Verbindung ordnungsgemäß geschlossen wurde (wasClean=true) oder nicht (wasClean=false).

Das wasClean-Attribut gibt über einen Boolean-Wert an, ob die Verbindung ordnungsgemäß geschlossen wurde oder nicht. Eine Verbindung wird als ordnungsgemäß geschlossen angesehen, wenn sowohl Client als auch Server jeweils einen Close-Frame gesendet und empfangen haben. Der Grund für die Beendigung der Verbindung kann aus dem code-Attribut ermittelt werden. Bei einer ordnungsgemäßen Trennung beinhaltet es den Statuscode 1000. Alle in RFC 6455 standardisierten Statuscodes sind in [Tabelle 4.1](#) in [Abschnitt 4.7.3](#) aufgelistet.

Das folgende Programmfragment illustriert, wie Sie auf diese Attribute zugreifen und diese in der JavaScript-Konsole Ihres Webbrowsers anzeigen können:

Listing 5.13 Beispiel für die Verwendung der close-Event-Attribute

```

1 ws.onclose = function(event) {
2     if(!event.wasClean) {
3         console.log('WS-Fehlercode: ' + event.code);
4         console.log('WS-Fehlerursache: ' + event.reason);
5     }
6 }
```

■ 5.11 Ausblick: HTTP 2.0/SPDY

Das World Wide Web hat sich seit der letzten Standardisierung der Protokollversion HTTP 1.1 im Jahre 1999 sehr stark verändert. Heutzutage sind Webseiten viel komplexer und enthalten deutlich mehr Inhalte, wie Bilder, Skripte oder CSS-Dateien. Auch die Dateigrößen der Inhalte sind deutlich höher als in den späten Neunzigerjahren des letzten Jahrtausends. Internetseiten werden heutzutage auch sehr oft an mobile Endgeräte wie Smartphones und Tablet-PCs ausgeliefert. Webseiten, die sehr große oder sehr viele Dateien enthalten, erzeugen mehr Traffic, wodurch für Benutzer mobiler Endgeräte höhere Kosten entstehen.

hen können. Dazu kann die Akkulaufzeit von Smartphones und Tablet-PCs durch größere Datenübertragungen verringt werden.

Durch diese gestiegenen Anforderungen werden Nachteile des heutigen HTTP-Protokolls deutlich sichtbar. Pro Anfrage kann nur eine Ressource zurückgeliefert werden. Des Weiteren können, wie wir schon besprochen haben, nur Daten angefordert werden, wenn der Client diese anfragt. Der Server ist nach aktuellem Stand nicht in der Lage, über das HTTP-Protokoll von sich aus Daten an den Client zu schicken.

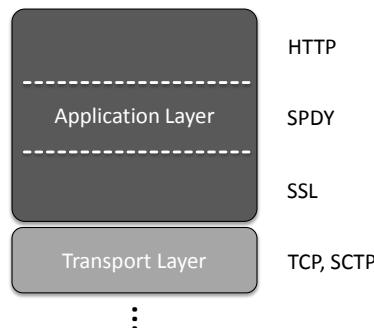


Bild 5.6 Einordnung von SPDY in das OSI-Modell bezogen auf das Internet

Um das Internet zu beschleunigen und das HTTP-Protokoll an den aktuellen Stand der Technik anzupassen, hat Google SPDY (ausgesprochen „Speedy“) [The10] entwickelt. Mit dieser Technologie soll u. a. die Performance der Ladezeiten verbessert und der Traffic der Übertragung verringert werden. Google hat sich das Ziel gesetzt, mit SPDY die Ladezeit von Webseiten um bis zu 50% zu verringern. SPDY benutzt TCP als Transportschicht, somit müssen Entwickler ihre Netzwerkinfrastruktur nicht ändern. Die einzige Voraussetzung besteht darin, dass der Client und der Server das SPDY-Protokoll unterstützen. Unterstützt der Client kein SPDY, so wird wieder auf das normale HTTP zurückgegriffen. Zusätzlich soll SPDY direkt über SSL verschlüsselt werden, um das Mitschneiden von Traffic nutzlos zu machen.

Damit die oben genannten Vorteile umgesetzt werden können, erlaubt das SPDY-Protokoll, mehrere HTTP-Anfragen in einer TCP-Verbindung zu stellen. Um Ladezeiten zu verringern, werden die Header bei der Übertragung zusätzlich komprimiert. SPDY ermöglicht auch die Priorisierung von Anfragen, sodass wichtige Daten früher ankommen können als weniger wichtige Daten. Ein weiteres Feature des Google-Protokolls ermöglicht dem Server, Daten eigenständig zu „pushen“, ohne dass der Client vorher eine Anfragen stellen musste. Der Server hat auch die Möglichkeit, dem Client wichtige Hinweise zu geben, z. B. welche Daten er noch laden muss, von denen er zum aktuellen Zeitpunkt noch nichts wusste.

Google geht sogar einen Schritt weiter und bietet ein neues Protokoll namens *Stream Control Transmission Protocol* (SCTP) [SXM+00] als Transport Layer für SPDY an, das TCP irgendwann ersetzen soll. SCTP bietet multiplexes Streamen und Überlastungskontrollen an.

SPDY ist bereits laut caniuse.com [Dev14b] ab Chrome 4, Firefox 13, Safari 8 und Opera 12.1 implementiert. Der Internet Explorer unterstützt das Protokoll ab der Version 11 aller-

dings nur unter Windows 8. Mit Safari kann diese Technologie noch nicht genutzt werden. Serverseitig ist SPDY schon weit verbreitet. Für die Webserver Apache, nginx und Node.js gibt es bereits Module und auch Jetty unterstützt SPDY seit den Versionen 7.6.2, 8.1.2 und 9.

Am 28. November 2012 hat die IETF den ersten Arbeitsentwurf zu HTTP 2.0 veröffentlicht, der auf Googles SPDY basiert.

Bei Google Chrome besteht die Möglichkeit, WebSockets über SPDY aufzubauen. Diese Implementierung ist aber nur experimentell und daher noch nicht stabil. Mit der Befehlszeile

```
--enable-websocket-over-spdy
```

starten Sie Google Chrome mit SPDY-Unterstützung für WebSockets. Der Startbefehl kann z. B. so aussehen:

```
C:\Users\SomeUser>"AppData\Local\Google\Chrome SxS\Application\  
chrome.exe" --enable-websocket-over-spdy
```


6

Der Server: Sprachliche Vielfalt

Mit der im vorangegangenen [Kapitel 5](#) eingeführten API des W3C sind Sie nun in der Lage, WebSocket-Clientanwendungen in Webbrowsern mittels JavaScript zu implementieren. Was Ihnen noch zum vollständigen Glück fehlt, ist die dazugehörige Serverseite. Für diese ist die Auswahl an verfügbaren Implementierungen mannigfaltig und lässt keine Programmierumgebung unberücksichtigt.



Fragen, die dieses Kapitel beantwortet:

- Womit kann man auf der Serverseite entwickeln?
- Wie funktioniert das Entwickeln genau mit den Kombinationen aus Node.js und Socket.io sowie Node.js und WebSocket.io?
- Wie kann man die im Java-Umfeld gebräuchlichen Frameworks Vert.x, Play und den JSR 356, der Teil der Java Enterprise Edition 7 (JEE 7) ist, für die Entwicklung nutzen? (Um der Vielfalt einigermaßen gerecht zu werden.)

■ 6.1 Übersicht verfügbarer WebSocket-Implementierungen

Eine Vielzahl von Implementierungen ist für ganz verschiedene Programmierumgebungen verfügbar. [Tabelle 6.1](#), die nach Programmiersprachen in alphabetischer Reihenfolge gruppiert ist, gibt einige der wichtigsten Vertreter wieder.

Da wir nicht jede dieser Implementierungen behandeln können, wollen wir uns auf eine Auswahl beschränken, die wir aktuell als repräsentativ und im breiten Einsatz sehen. Mit diesen Frameworks können Sie nicht nur WebSocket-Server implementieren, sondern auch die Clientseite, die nicht zur Ausführung an einen Webbrowser gebunden ist. Aus diesem Grund werden wir in den jeweiligen Kapiteln beide Seiten betrachten.

Tabelle 6.1 WebSocket-Implementierungen, gruppiert nach Programmierumgebungen

Name	Sprache	Webseite	Lizenz
SuperWebSocket	.NET	http://superwebsocket.codeplex.com/	Apache 2.0
libwebsockets	C++	https://github.com/warmcat/libwebsockets/	LPGL 2.1
Vert.x	CoffeScrip, Groovy, Java, JavaScript, Python, Ruby	http://vertx.io/	Apache 2.0
Shirasu	Erlang	https://github.com/michilu/shirasu/	BSD 3-Clause
Jetty	Java	http://www.eclipse.org/jetty/	Apache 2.0 EPL 1.0
jWebSocket	Java, JavaScript	http://jwebsocket.org/	LPGL 3.0
HornetQ	Java	http://www.jboss.org/hornetq/	Apache 2.0
Kaazing WebSocket Gateway	Java	http://kaazing.com/	Kaazing
The WebSockets SDK	Java	https://websocket-sdk.java.net/	GPL
Engine.IO	Node.js	https://github.com/Automattic/engine.io/	MIT
Socket.IO	Node.js	http://socket.io/	MIT
WebSocket-Node	Node.js	https://github.com/Worlize/WebSocket-Node/	Apache 2.0
WebSocket.IO	Node.js	https://github.com/LearnBoost/websocket.io/	MIT
phpwebsocket	PHP	https://code.google.com/p/phpwebsocket/	LPGL 3.0
Autobahn	Python	http://autobahn.ws/	Apache 2.0
pywebsocket	Python	http://code.google.com/p/pywebsocket/	BSD 3-Clause
Tornado	Python	http://www.tornadoweb.org/	Apache 2.0
EM-WebSocket	Ruby	https://github.com/igrigorik/em-websocket/	MIT
EventMachine	Ruby	http://rubyeventmachine.com/	GPL
web-socket-ruby	Ruby	https://github.com/gimite/web-socket-ruby/	BSD 3-Clause

■ 6.2 Node.js

Node.js ist eine frei verfügbare Softwarelaufzeitumgebung. Es basiert auf der Programmiersprache JavaScript, die damit erstmals aus ihren Browserfängen entflieht und daher auch serverseitig ausführbar ist. Node.js wurde mit einem spezifischen Fokus auf skalier-

bare, blockierungsfreie und ereignisorientierte Netzwerkprogramme entwickelt. Als JavaScript Interpreter kommt darin die V8-Engine zum Einsatz, welche eine freie Implementierung der Skriptsprache ECMAScript (JavaScript) nach dem Standard ECMA-262 ist. Sie wird vom Unternehmen Google Inc. unter einer BSD-Lizenz als freie Software veröffentlicht und kommt als Teil des Webbrowsers Google Chrome zum Einsatz. Node.js ist ein Beispiel dafür, dass V8 auch unabhängig von Chrome verwendet werden kann.

Im Gegensatz zu anderen Servern wie z. B. dem Apache HTTP-Server unterteilt Node.js (oder auch einfach nur Node) Instanzen nicht in mehrere Threads. Es kommt mit einem einzigen Thread pro Instanz aus, da es ereignisorientiert aufgebaut ist. Das bedeutet, es wird nicht für jeden Request ein neuer Thread gestartet. Stattdessen arbeitet Node mit Event-Listenern und asynchronen Callback-Funktionen. Das daraus resultierende blockierungsfreie I/O-Handling stellt das Grundprinzip von Node.js dar. Node.js-Anwendungen warten also nicht, bis ein bestimmtes Ergebnis eingetroffen ist, sondern arbeiten in der Zwischenzeit weitere Anfragen ab oder „legen sich schlafen“. Dies wird durch einen Event-loop realisiert, der alle Event-Listener und die dazugehörigen Callback-Funktionen registriert.

Löst z. B. ein Request eine Datenbankabfrage aus, wartet Node.js nicht, bis dieser Prozess fertiggestellt ist, sondern arbeitet umgehend andere Requests ab oder „legt sich schlafen“, bis die Ergebnisse aus der Datenbank vorliegen. Dadurch kann eine Vielzahl von Anfragen nebenläufig bearbeitet werden, ohne Ressourcen zu vergeuden. Stellt die Ausführungsumgebung mehrere Rechenressourcen z. B. in Form eines Mehrkernprozessors zur Verfügung, erfolgt die Verarbeitung sogar parallel.

An dem folgenden Beispiel können Sie den Aufbau von Node.js-Programmen und die Funktionsweise der Laufzeitumgebung nachvollziehen.

Node.js ist ein modulares System. Wichtige Module für die wesentlichen Basisfunktionalitäten sind bereits in der Installation enthalten. Daneben steht Ihnen eine Vielzahl weiterer Module im Internet zur Verfügung, die Sie mit dem Node.js-Modulmanager herunterladen und integrieren können (dazu mehr in den anschließenden Kapiteln). In Ihren Programmen können Sie die vorhandenen Module durch die Methode `require()` importieren. Das HTTP-Modul ist ein Basismodul, das Sie nicht zusätzlich herunterladen müssen. Es erlaubt Ihnen, u. a. einen Webserver zu implementieren.

Listing 6.1 Ein einfacher in Node.js implementierter Webserver

```

1 var http = require('http');
2
3 http.createServer(function (req, res) {
4     res.writeHead(200, {'Content-Type': 'text/plain'});
5     res.end('Hallo Welt!\n');
6 }).listen(4000);
7
8 console.log('Der Server ist erreichbar unter:
  http://127.0.0.1:4000');
```

In dem Programm in Listing 6.1 können Sie bereits viele Prinzipien und Grundkonzepte von Node.js erkennen. Das Programm implementiert einen – in seiner Funktionalität sehr eingeschränkten – Webserver. Dieser ist auf dem Localhost

`http://127.0.0.1/`

über den Port 4000 ansprechbar und gibt jedem HTTP-Request den Antwortstring 'Hallo Welt!\n' zurück. In Zeile 1 wird das Basismodul `http` der gleichnamigen Variablen `http` zugewiesen. Anschließend wird durch den Aufruf der Methode `createServer()` eine Instanz eines HTTP-Servers erzeugt, dem wir im Argument eine anonyme Callback-Funktion übergeben. Diese besitzt zwei Parameter, die, wie Sie vielleicht schon ahnen, den Request und die Response repräsentieren. Innerhalb der Callback-Funktion können Sie auf diese beiden Argumente zugreifen und damit arbeiten. Im Beispiel verwenden wir nur das Response-Objekt, in das wir die Antwort an den anfragenden Client schreiben. Dazu müssen wir zunächst mit der verfügbaren Methode `writeHead()` den Response-Header schreiben. Diese erwartet als Argumente den HTTP-Statuscode und den Content-Type. Wir setzen den Statuscode auf den Wert 200 und den Content-Type auf `text/plain`. Die Response wird anschließend mit einer simplen Textnachricht der Form 'Hallo Welt!\n' beendet. Zu guter Letzt müssen wir noch einen Event-Listener registrieren, der auf dem Port 4000 auf Anfragen von Clients wartet.

Wenn Sie diesen Programmcode nehmen und mit Node.js ausführen, haben Sie mit diesen wenigen Zeilen Code einen ersten simplen Server programmiert. Damit Sie dies nun auch in die Tat umsetzen können, zeigen wir Ihnen in den nachfolgenden Kapiteln, wie Sie Node installieren und ausführen.

6.2.1 Installation von Node.js

Node.js können Sie für ein breites Spektrum an Betriebssystemen herunterladen.¹ Für Windows und Mac OS X finden Sie auf der Download-Seite Installer, mit denen Sie die Installation von Node.js mit wenigen Mausklicks durchführen können. Für viele der gängigen Linux-Distributionen kann Node.js mithilfe des jeweiligen Paketmanagers installiert werden. Weiterführende Informationen zur Installation via Paketmanager finden Sie auf der Wikiseite von Node.² Natürlich können Sie Node auch mit GIT herunterladen und installieren. Eine Anleitung dazu finden Sie ebenfalls online.³

Wenn Sie eine Installation auf einem Linux-System mittels der Node.js-Quelltexte bevorzugen, müssen Sie darauf achten, dass die Pakete `make` bzw. `g++`, `python 2.6` oder `2.7` und `libexecinfo` auf Ihrem System installiert sind. Diese werden zum Übersetzen der Source-Dateien benötigt. Ob Ihnen die Installation geeglückt ist, können Sie leicht prüfen, indem Sie den folgenden Befehl in das Terminal eingeben:

```
node -v
```

Erscheint daraufhin die Versionsnummer von Node.js im Terminal, war die Installation erfolgreich (siehe Bild 6.1).

Node.js enthält zusätzlich zum JavaScript-Interpreter V8 und diversen Basismodulen auch einen eigenen Modulmanager, mit dem externe Module nachgeladen und installiert werden können. Der Modulmanager mit dem Namen `npm` (Node Packaged Modules) wird in

¹ <http://nodejs.org/download/>

² <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager/>

³ <https://github.com/joyent/node/wiki/Installation/>



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The output of the command 'node -v' is 'v0.10.28' and the output of 'npm -v' is '1.4.9'. This indicates a successful installation of both Node.js and npm.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright © 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\User>node -v
v0.10.28

C:\Users\User>npm -v
1.4.9

C:\Users\User>
```

Bild 6.1 Erfolgreiche Node.js-Installation

der Regel bei der Installation von Node.js mitinstalliert. Ob dies für Ihre Installation der Fall ist und der Modulmanager ordnungsgemäß funktioniert, können Sie mit dem Befehl

```
npm -v
```

testen. Nach einer erfolgreichen Installation sollte sich der Modulmanager mit einer Ausgabe im Terminal melden, die Ihnen Versionsinformationen anzeigt (siehe [Bild 6.1](#)).

Da nun sowohl Node.js als auch der Modulmanager installiert sind, steht Ihnen nichts mehr im Wege, um mit Ihren Entwicklungen loszulegen. Mit dem Konsolenbefehl node können Sie Ihre Node.js-Programme ausführen. Wenn Sie z. B. den in [Listing 6.1](#) angegebenen Webserver in einer Datei mit dem Namen *HelloWorldServer.js* gespeichert haben, dann können Sie diesen mit der folgenden Kommandozeile ausführen:

```
node HelloWorldServer.js
```

Node.js-Module, die den Funktionsumfang der Basismodule erweitern, können Sie über den Modulmanager durch den Aufruf `npm install` installieren. Hier ist ein Beispiel für das fiktive Modul mit dem Namen `modulname`:

```
npm install modulname
```

Der Modulmanager `npm` erstellt Ihnen daraufhin im Ordner, in dem Sie sich zur Zeit der Ausführung des Konsolenkommandos zur Paketinstallation befinden, einen Unterordner namens `node_modules`. Dort ist das Modul dann enthalten. Existiert der Ordner `node_modules` bereits, muss dieser nicht erzeugt werden und das angegebene Modul wird dem Ordner schlicht hinzugefügt. Es empfiehlt sich, die für ein Projekt benötigten Module im Wurzelverzeichnis des Projekts zu installieren. Auf diese Weise können die Module von der Node.js-Laufzeitumgebung gefunden werden und der Projektordner ist damit einigenständig und abgeschlossen. Sie können diesen z. B. im Dateibaum verschieben oder packen und auf ein anderes System bringen, ohne dass Abhängigkeiten dabei verloren gehen. Andererseits bringt das den Nachteil mit sich, dass Sie in verschiedenen Projekten unter Umständen viele Kopien des gleichen Moduls vorliegen haben. Möchten Sie Module lieber nur einmal und dafür systemweit installieren – was z. B. für Module, die in vielen Projekten zum Einsatz kommen, sinnvoll sein kann –, können Sie dies mit dem Kommandozeilenparameter `-g` triggern. Der Aufruf nimmt dann die folgende Gestalt an:

```
npm install -g modulname
```

Auf diese Weise werden systemweit installierte Module in einen spezifischen Ordner der Node.js-Installation abgelegt und verwaltet. Dies ist das *prefix/lib/node_modules*-Verzeichnis, wobei *prefix* in der Regel mit */usr/local* korrespondiert.

Um Module für einen bestimmten Zweck ausfindig machen zu können, steht in `npm` eine Suchfunktion bereit. Mit

```
npm search "Suchwörter stehen hier"
```

können Sie die Modulbeschreibungen nach den von Ihnen angegebenen Suchwörtern durchforsten lassen. Wenn Sie mehrere Suchwörter eingeben, müssen Sie diese durch Anführungszeichen einfassen.

Für weiterführende Informationen über Node.js und `npm` möchten wir Sie auf die folgenden zwei Webseiten aufmerksam machen:

- <http://nodejs.org/>
- <https://npmjs.org/>

6.2.2 WebSocket.io

Nun wollen wir das bisher Gelernte anwenden und ein erstes Node.js-Programm schreiben, das einen WebSocket-Service implementiert und auf dem Modul WebSocket.io aufbaut. Wie der Name schon sagt, stellt das Modul eine Implementierung des WebSocket-Protokolls bereit und eine API, um dieses in eigenen Node.js-Programmen verwenden zu können. WebSocket.io wird von der Firma LearnBoost entwickelt und unter der MIT-Lizenz zum Abruf bereitgestellt.⁴

Für unser erstes Projekt erstellen wir zunächst ein Projektverzeichnis. Öffnen Sie dazu als Erstes die Eingabeaufforderung bzw. das Terminal. Erstellen Sie danach einen Ordner und wählen diesen als Arbeitsverzeichnis:

```
mkdir echo_websocket.io
cd echo_websocket.io
```

Als Nächstes rufen wir den Modulmanager `npm` auf und installieren das Modul WebSocket.io mit dem folgenden Konsolenbefehl im Projektverzeichnis:

```
npm install websocket.io
```

Nach der Installation, können wir mit der Implementierung Ihres ersten WebSocket-Servers auf Basis von WebSocket.io beginnen. Erstellen Sie dazu mit einem Editor Ihrer Wahl eine Datei im Projektverzeichnis, die die Endung *.js* trägt. Diese kann z. B. *echoServer.js* heißen. Fügen Sie den Programmcode aus Listing 6.2 in die Datei ein. Das Codebeispiel implementiert einen Webserver, der einen WebSocket-Server integriert.

⁴ <https://github.com/LearnBoost/websocket.io/>

Listing 6.2 WebSocket-Echo-Server mit WebSocket.io und Node.js

```

1 var ws = require("websocket.io"),
2     fs = require('fs'),
3     http = require('http');
4
5 var httpServer = http.createServer(function (request, response) {
6     fs.readFile (__dirname+"/index.html", function(error, data) {
7         if(error) {
8             response.writeHead(500);
9             return response.end('Fehler beim Laden der Datei
10                index.html');
11        }
12        else {
13            response.writeHead(200);
14            response.end(data);
15        }
16    });
17
18 // Vereint Webserver und WebSocket-Server
19 var wsServer = ws.attach(httpServer);
20
21 wsServer.on('connection', function(client) {
22     client.on('message', function(message) {
23         client.send(message);
24     });
25 })
26
27 httpServer.listen(4000);
28 console.log("Der Echo-Server laeuft auf dem Port",
29             httpServer.address().port);

```

Das Programm importiert in [Zeile 1](#) das externe Modul `websocket.io`, welches Sie zuvor installiert haben. Damit eine HTML-Datei vom Dateisystem eingelesen werden kann, wird das Basismodul `fs`, was für „File System“ steht, benötigt und daher in [Zeile 2](#) importiert. Der Code in [Zeile 5](#) bis [16](#) erzeugt den Webserver, der auf alle eingehenden HTTP-Anfragen eine einzige HTML-Datei, in unserem Fall ist das die Datei `index.html`, an die anfragenden Clients zurückliefert. Diese HTML-Datei soll als WebSocket-Client fungieren und eine Verbindung mit dem WebSocket-Server aufbauen. Wenn wir nun das `WebSocket.io`-Modul mit dem Webserver vereinen, wird ein WebSocket-Server instanziert (siehe [Zeile 19](#)). In den Zeilen [21](#) bis [25](#) definieren wir die Funktionalität des WebSocket-Servers, der alle ein-treffenden Nachrichten an den Client als Echo zurückschickt. In [Zeile 27](#) bestimmen wir nur noch, dass der Server auf dem Port 4000 laufen soll.

Widmen wir uns nun der Clientseite. Erstellen Sie im Projektordner eine HTML-Datei mit dem Namen `index.html`, die den Quelltextausschnitt aus [Listing 6.3](#) enthält.

Listing 6.3 Erste clientseitige WebSocket-Anwendung

```

1 ...
2 <h1>WebSocket.io Test</h1>

```

```

3 <script type="text/javascript">
4     var ws = new WebSocket("ws://" + window.location.host);
5     ws.onopen = function(event) {
6         console.log("Verbindung aufgebaut!");
7         ws.send("Hallo!");
8     }
9
10    ws.onmessage = function(event) {
11        console.log("Eingegangene Nachricht: " + event.data);
12        ws.close();
13    }
14
15    ws.onerror = function(event) {
16        console.log(event.reason);
17    }
18
19    ws.onclose = function(event) {
20        if(!event.wasClean) {
21            console.log("Ein Fehler ist aufgetreten! Code: " +
22                        event.code);
23            console.log("Der Grund dafuer war: " + event.reason);
24        }
25        else {
26            console.log("Verbindung getrennt!");
27        }
28    }
29    ...

```

Auf der Clientseite wird durch das Ausführen des enthaltenen JavaScripts eine WebSocket-Verbindung zu Ihrem Echo-Server aufgebaut. Die dazu verwendete WebSocket-API des W3C haben wir in [Kapitel 5](#) eingeführt. Ist der Verbindungsauftbau erfolgreich, wird "Verbindung aufgebaut!" in der JavaScript-Konsole des Browsers ausgegeben ([Zeile 6](#)) und die Textnachricht "Hallo!" an Ihren Server geschickt ([Zeile 7](#)). Wird eine Nachricht vom Server empfangen, wird diese ebenfalls in der Konsole des Browsers ausgegeben ([Zeile 11](#)). Kommt es bei dieser Interaktion zu einem Fehler und die Verbindung wird abgebrochen, gibt der Code mithilfe des event-Objekts, das dem Event-Handler onclose als Parameter übergeben wird, den Code (event.code) und den Grund (event.reason) aus ([Zeile 20](#) bis [23](#)).

Da Sie nun alle Komponenten für ein erstes WebSocket-Programm implementiert haben, können Sie mit dem Erproben und Erkunden beginnen. Starten Sie dazu Ihren Echo-Server, indem Sie den folgenden Kommandozeilenbefehl in ein Terminal eingeben:

```
node echoServer.js
```

Der Echo-Server ist nun gestartet. Wenn Sie alles richtig gemacht haben, bekommen Sie auf der Konsole eine Ausgabe, die der in [Bild 6.2](#) entspricht.

Nehmen Sie sich nun einen WebSocket-fähigen Webbrowser zur Hand und öffnen Sie damit die URL

http://localhost:4000/

```
C:\Windows\system32\cmd.exe - node echoServer.js
C:\Users\User\socket.io>node echoServer.js
Der EchoServer läuft auf dem Port 4000
```

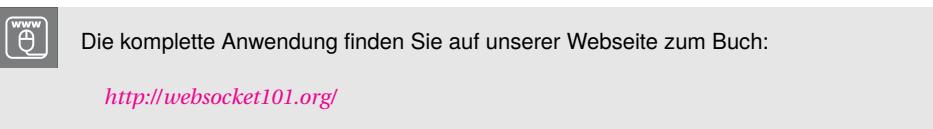
Bild 6.2 Erfolgreich gestarteter Echo-Server

Die Darstellung in [Bild 6.3](#) sollte daraufhin in Ihrem Browser zu sehen sein.



Bild 6.3 Ihre erste WebSocket-Anwendung realisiert mit WebSocket.io und Node.js

Herzlichen Glückwunsch! Sie haben Ihre erste WebSocket-Anwendung geschrieben.



6.2.3 Socket.io

Das Node.js-Modul `WebSocket` bietet eine Implementierung des `WebSocket`-Protokolls und eine API zur Implementierung von `WebSocket`-Servern. Es stellt jedoch keine Ersatzlösungen für Browser zur Verfügung, die keine `WebSockets` unterstützen. Eine solche Ersatzlösung wird auch als *Fallback* bezeichnet. Sollte Ihre Anwendung diese Anforderung haben, müsste dies eigenständig von Ihnen implementiert werden. In [Kapitel 3](#) haben wir Vorgängertechnologien angesprochen, mit denen sich ebenfalls Webanwendungen mit einer höheren Interaktivität und Echtzeitfähigkeit realisieren lassen. Um Ihnen an dieser Stelle die Arbeit abzunehmen, haben die Macher von `WebSocket` das Node.js-Modul `Socket.io` entwickelt.

`Socket.io` ist eine client- und serverseitige JavaScript-Bibliothek für Echtzeitanwendung unter Node.js. Es bietet insbesondere eine clientseitige Abstraktionsschicht für Web-

Sockets an, die für Browser ohne WebSocket-Unterstützung einen Fallback auf die Long-Polling-Technologie realisiert. Durch eine eigene API und ein eigenes Protokoll bietet Socket.io zusätzliche Vorteile, wie das Definieren eigener Events und eine Session-ID für jede WebSocket-Verbindung. Gleichzeitig schickt jeder WebSocket in einem vordefinierten Zeitraum *Heartbeats* an den Server, um zu signalisieren, dass die Verbindung noch nicht beendet wurde. Dies ist mit dem Ping des WebSocket-Protokolls zu vergleichen. Dadurch kann der Socket.io-Server eine Verbindung automatisch trennen, falls diese nicht sauber beendet wurde.

Um die Unterschiede zwischen WebSocket.io und Socket.io besser verstehen zu können, wollen wir im Folgenden eine Implementierung des Echo-Servers aus [Abschnitt 6.2.2](#) mithilfe von Socket.io realisieren. Dazu erstellen Sie zunächst einen neuen Projektordner und installieren das Modul mit dem Paketmanager:

```
mkdir echo_socket.io
cd echo_socket.io
npm install socket.io
```

Erstellen Sie nun eine JavaScript-Datei, in der Sie den Echo-Server implementieren, diesmal allerdings auf Grundlage des Socket.io-Moduls. Als Dateiname können Sie wie im Beispiel zuvor *echoServer.js* wählen.

Der Aufbau und die Struktur der Echo-Anwendung unterscheiden sich kaum vom WebSocket.io-Beispiel in [Listing 6.2](#). Eine wesentliche Erweiterung, die durch Socket.io eingebracht wird, können Sie der Zeile 24 des [Listing 6.4](#) entnehmen. Mit Socket.io können Sie eigene Events definieren. Möchten Sie einen selbst definierten Event auslösen, müssen Sie diesen mit der `emit()`-Methode aktivieren. Mit dem ersten Parameter der `emit()`-Methode geben Sie den Namen des adressierten Events an und als weiteren Parameter die Nachricht, die an diesen gerichtet wird. Wenn Sie keine eigenen Events benötigen, können Sie für den einfachen Nachrichtenaustausch zwischen Client und Server auch die Standardmethode `send()` verwenden, die den Standardevent `message` auslöst.

Listing 6.4 Echo-Server mit Socket.io

```
1 var fs = require('fs'),
2 http = require('http');
3
4 var httpServer = http.createServer(function(request, response){
5     fs.readFile(__dirname+"/index.html", function(error, data){
6         if(error) {
7             response.writeHead(500);
8             return response.end('Fehler beim Laden der Datei
9                 index.html');
10        }
11        else {
12            response.writeHead(200);
13            response.end(data);
14        }
15    });
16
17 var io = require('socket.io').listen(httpServer);
```

```

18
19 io.sockets.on('connection', function(socket){
20     socket.on('message', function(message){
21         socket.send(message);
22     })
23
24     socket.on('myevent', function(message){
25         socket.emit('myevent', "Eine Nachricht von myevent: " +
26             message);
27     })
28 }
29 httpServer.listen(4000);
30 console.log("Der EchoServer läuft auf dem Port",
31             httpServer.address().port);

```

Schauen wir uns nun an, wie die Clientseite der Anwendung, also die Datei *index.html*, zu gestalten ist:

Listing 6.5 Socket.io-Testclient

```

1 ...
2 <h2>Socket.io Test</h2>
3 <script type="text/javascript"
4         src="/socket.io/socket.io.js"></script>
5 <script>
6     var socket = io.connect("http://localhost:4000");
7     socket.on("connect", function(){
8         socket.send("Hallo!");
9         socket.emit("myevent", "Hallo Welt!");
10    });
11    socket.on("message", function(message){
12        console.log(message);
13    });
14    socket.on("myevent", function(message){
15        console.log(message);
16        socket.disconnect();
17    });
18    socket.on("disconnect", function(){
19        console.log("Verbindung getrennt!");
20    });
21    socket.on("error", function(reason){
22        console.log(reason)
23    });
24    socket.on("connect_failed", function(){
25        console.log("Verbindung fehlgeschlagen!");
26    });
27 </script>
28 ...

```

Um die Socket.io-Abstraktionsschicht im Browser benutzen zu können, müssen Sie die clientseitige Socket.io-Bibliothek importieren (siehe erstes script-Element in [Listing 6.5](#)).

Diese JavaScript-Library wird beim Starten eines Socket.io-Serverprozesses per Konvention unter der Adresse

`http://<server>/socket.io/socket.io.js`

bereitgestellt und kann folglich über diese URL in der HTML-Datei referenziert und geladen werden. Wie Sie im Quellcode sehen, hat Socket.io eine etwas abgewandelte API gegenüber der vom W3C standardisierten WebSocket-API. Die Eröffnung der Verbindung wird mit der Methode `io.connect()` realisiert. In diesem Schritt prüft der Socket.io-Server, ob der adressierte Server WebSockets unterstützt. Einen weiteren Unterschied zur W3C-API sieht man an dem URL-Schema. Der Server wird nicht mit dem Protokoll `ws` angeprochen, sondern mit dem `http`-Protokoll. Die Socket.io-Events `connect`, `message`, `error` und `disconnect` sind das Pendant zu den Event-Handlers `onopen`, `onmessage`, `onerror`, `onclose`. Eigene Events können, wie bereits für die Serverseite beschrieben, durch die `emit()`-Methode ausgelöst ([Zeile 8](#)) und durch einen entsprechenden Event-Handler behandelt werden ([Zeile 13](#)). Nachrichten, die mit der `send()`-Methode abgesetzt werden, lösen den Standardevent `message` aus. Starten Sie nun die Anwendung ebenfalls mit dem Befehl:

```
node echoServer.js
```

In Ihrem Terminal sollte daraufhin die im folgenden [Bild 6.4](#) dargestellte Ausgabe erscheinen:

```
C:\Windows\system32\cmd.exe - node echoServer.js
C:\Users\User\socket.io>node echoServer.js
Der EchoServer läuft auf dem Port 4000
```

Bild 6.4 Konsolenausgabe des Socket.io-Echo-Servers

Öffnen Sie nun mit Ihrem Browser die URL

`http://localhost:4000/`

Daraufhin sollte sich das in [Bild 6.5](#) dargestellte Ergebnis einstellen.

Mehr Informationen zu Socket.io können Sie der offiziellen Webseite entnehmen.⁵

www

Diese Beispielanwendung können Sie ebenfalls von der Webseite zum Buch unter der Adresse

<http://websocket101.org/>

herunterladen. Eine komplexere Anwendung auf Basis von Socket.io stellen wir Ihnen im [Kapitel 8](#) vor. Dort werden wir Ihnen näherbringen, wie Sie mit WebSockets Ihr Smartphone oder Tablet zu einer Fernbedienung umfunktionieren können.

⁵ <http://socket.io/>

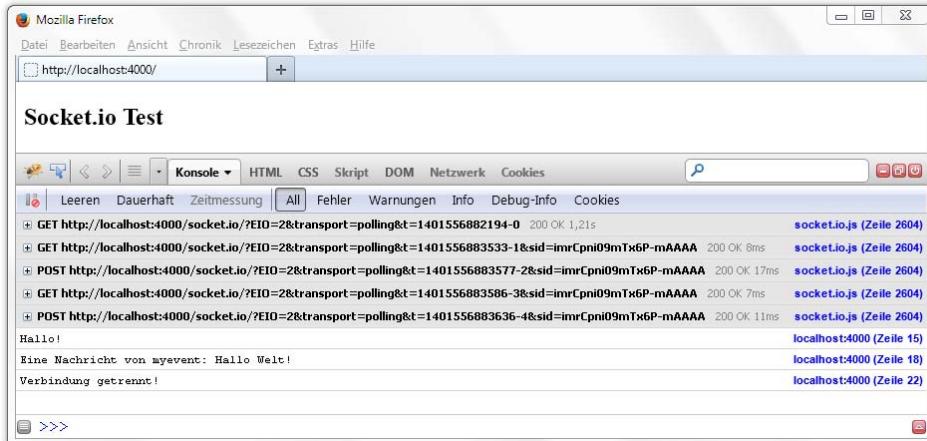


Bild 6.5 WebSocket-Client mit Socket.io

6.2.4 WebSocket-Node

WebSocket-Node ist eine weitere WebSocket-Implementierung für Node.js. Sie bringt einige Vorteile gegenüber WebSocket.io mit. Mit WebSocket-Node können Sie z. B. die maximale Framegröße oder die maximale Nachrichtengröße für eine vom Client gesendete WebSocket-Nachricht bestimmen sowie in welchem Intervall ein WebSocket-Server Pings zur Ermittlung des Verbindungsstatus senden soll. Eine weitere nützliche Funktion ist die Möglichkeit, den Handshake-Request eines Verbindungsaufbaus programmatisch abzufangen und diesen mit eigenen Programmen auszuwerten und gegebenenfalls zu verweigern. Näheres dazu erfahren Sie in [Abschnitt 7.2.3](#), wo wir mit dieser Funktionalität einen Autorisierungsmechanismus realisieren werden.

Wir werden uns nun in diesem Kapitel mithilfe von WebSocket-Node mit Binärdaten beschäftigen, die die Stelle von Textnachrichten einnehmen. Unser Ziel ist, einen Echo-Server zu implementieren, dem wir Binärdaten senden können und der uns dieselben wieder zurückschickt. Für einen besseren visuellen Eindruck des Geschehens wollen wir in unserem Beispiel Bilder zum Server schicken. Das zurückgeschickte Bild vom Server wollen wir dann in unserer HTML-Datei via JavaScript darstellen.

Erzeugen Sie sich einen Projektordner mit dem Namen `echo_websocket-node` und installieren Sie das Modul WebSocket-Node mithilfe von npm:

```
mkdir echo_websocket-node
cd echo_websocket-node
npm install websocket
```

Die Basis unserer Implementierungen ist der Webserver aus [Listing 6.2](#) im [Abschnitt 6.2.2](#), der für jede Anfrage die gleiche statische HTML-Datei zurückliefert. Die dazu im Vergleich vorgenommenen Änderungen sind in fetter Schrift hervorgehoben (siehe [Listing 6.6](#)).

Listing 6.6 WebSocket-Node-Server mit einem Webserver verbinden (*server.js*)

```

1 var fs = require("fs");
2 var http = require("http");
3 var WebSocketServer = require("websocket").server;
4
5 var httpServer = http.createServer(function(req,res) {
6     fs.readFile(__dirname+"/index.html", function(error, data) {
7         if(error) {
8             res.writeHead(500);
9             return res.end('Fehler beim Laden der Datei index.html');
10        }
11        else {
12            res.writeHead(200);
13            res.end(data);
14        }
15    })
16 });
17
18 var webSocketServer = new WebSocketServer({
19     httpServer:httpServer,
20     autoAcceptConnections:true,
21     maxReceivedFrameSize: 64*1024*1024,
22     maxReceivedMessageSize: 64*1024*1024,
23 });
24
25 webSocketServer.on("connect",function(socket){
26     socket.on("message",function (message){
27         if(message.type=="binary"){
28             socket.sendBytes(message.binaryData);
29         }
30     });
31 });
32
33 httpServer.listen(4000);
34 console.log("Der EchoServer läuft auf dem Port ",
            httpServer.address().port);

```

Wir laden das WebSocket-Node-Modul ein ([Zeile 3](#)) und erstellen einen WebSocket-Server ([Zeile 18](#)). Diesem weisen wir den vorher erzeugten Webserver zu. Zusätzlich definieren wir, dass der WebSocket-Server alle WebSocket-Handshakes automatisch akzeptieren soll ([Zeile 20](#)). Die maximale Framegröße ([Zeile 21](#)) und die maximale Nachrichtengröße für eine serverseitig empfangene Nachricht ([Zeile 22](#)) legen wir ebenfalls fest. Da wir Bilder austauschen wollen, die aus einer größeren Datenmenge bestehen können, müssen wir großzügig sein und erlauben jeweils 64 MB.

Danach beschreiben wir, was passieren soll, wenn eine WebSocket-Verbindung aufgebaut ist und eine Nachricht vom Client abgeschickt wurde. Falls eine Nachricht am Server ankommt, überprüfen wir, ob es sich um eine Binärnachricht handelt. Das `message`-Objekt von WebSocket-Node ist folgendermaßen aufgebaut:

Listing 6.7 Ein Message-Objekt in WebSocket-Node

```

1 // Fuer Textframes:
2 {
3     type: "utf8",
4     utf8Data: "Ein String, der die eingegangene Nachricht
5      beinhaltet."
6 }
7 // Fuer Binaer-Frames:
8 {
9     type: "binary",
10    binaryData: binaryDataBuffer
11    // Ein Buffer-Objekt, das die binaere Nachricht im
12    // Payload beinhaltet
13 }
```

Wir können anhand der Eigenschaft `type` zwischen binären Daten und UTF-8-Text unterscheiden. Wenn wir nur Binärdaten versenden wollen, müssen wir dies vorher abfragen und beim Senden der Nachricht die Methode `sendBytes()` aufrufen. Als Argument übergeben wird dann aus dem `message`-Objekt die Eigenschaft `binaryData`, die die Nutzdaten der Nachricht enthält.

Widmen wir uns nun der Clientseite unserer Anwendung. Wir erweitern dazu die HTML-Seite aus [Listing 6.3](#). Die wesentlichen Ergänzungen dazu sind in [Listing 6.8](#) angegeben.

Listing 6.8 Definieren des Dateiformulars und des Sende-Buttons

```

1 <div>
2     <input type="file"/>
3     <button onclick="sendFile()">send</button>
4 </div>
5 <div>
6     <img id="image"/>
7 </div>
```

Wir erstellen ein `<input>`-Tag vom Typ `file`, damit wir Dateien bzw. Bilder in dieses Formularfeld importieren können. Danach wird ein Button definiert. Das Aktivieren dieses Buttons stößt die Ausführung der JavaScript-Methode `sendFile()` an, die unsere Datei abschickt, welche wir in [Listing 6.9](#) annehmen werden.

Listing 6.9 Öffnen eines WebSockets und Implementierung der `sendFile()`-Funktion

```

1 var webSocket = new WebSocket("ws://" + window.location.host);
2
3 function sendFile(){
4     var file = document.querySelector('input[type="file"]').files[0];
5     webSocket.send(file);
6 }
```

Wir instanziiieren einen WebSocket ([Zeile 1](#)) und implementieren die `sendFile()`-Funktion ([Zeile 3](#) bis [Zeile 6](#)). Diese speichert eine Referenz auf die Bilddatei in der Variablen `file`

ab. Über diese Variable kann auf die Bytes des BLOB-Objekts zugegriffen und durch den Aufruf der `send()`-Methode über den WebSocket an den Server geschickt werden.

Das über den WebSocket zurückkommende Bild kann dann z. B. durch das in [Listing 6.10](#) angegebene Codefragment eingelesen und zur Darstellung dynamisch in die HTML-Seite eingebettet werden.

Listing 6.10 Einlesen und Darstellen binärer Bilddaten

```

1 var url;
2 window.URL = window.URL ||
3             window.webkitURL ||
4             window.mozURL ||
5             window.msURL;
6 webSocket.onmessage = function(evt){
7     if(url){
8         window.URL.revokeObjectURL(url);
9     }
10    var img = document.getElementById("image");
11    url = window.URL.createObjectURL(evt.data);
12
13    img.src = url;
14 }
```

Wir erstellen eine globale Variable namens `url`, in der wir die URL des erzeugten Bildes ablegen. Wenn eine Nachricht ankommt, holen wir uns als Erstes eine Referenz auf das ``-Tag und speichern diese in der `img`-Variablen ab. Dann erzeugen wir mit der Methode `window.URL.createObjectURL()` eine URL, mit der wir auf die gerade erhaltenen Binärdaten (diese müssen als BLOB vorliegen) zugreifen können. Die Binärdaten sind nun im Cache gespeichert. Das URL-Objekt weisen wir dann dem `src`-Attribut unserer `img`-Variablen (und damit dem ``-Tag) zu. Davor überprüfen wir in [Zeile 7](#), ob die `url`-Variable noch eine URL von einem vorherigen BLOB beinhaltet. Falls dies der Fall ist, löschen wir diesen BLOB und das dazugehörige URL-Objekt aus dem Cache, da wir das vorherige Bild nicht mehr benötigen. Ohne das Aufrufen der `revokeObjectURL()`-Methode bleiben der eingestellte BLOB und die dazugehörige URL so lange aktiv, bis Sie die Seite verlassen bzw. schließen.

Führen Sie nun die fertige Anwendung aus und sehen Sie sich an, wie das Programm funktioniert. Dazu starten Sie zunächst den Server:

```
node server.js
```

Öffnen Sie mit einem Browser die URL

```
http://localhost:4000/
```

Sie sollten die statische HTML-Seite angezeigt bekommen, die das Formular zum Absetzen eines Bilds an den Echo-Server via WebSocket enthält (siehe [Bild 6.6](#)).

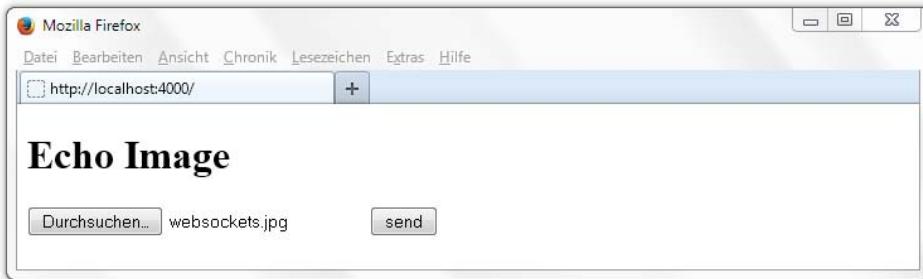


Bild 6.6 Ein WebClient zum Initiieren eines Bilddaten-Echos

Klicken Sie auf *Durchsuchen...* und wählen Sie ein Bild aus, das Sie versenden möchten. Starten Sie die WebSocket-Kommunikation mit einem Klick auf *send*. Nach wenigen Augenblicken (in Abhängigkeit der Dateigröße) sollten Sie das ausgesuchte Bild auf der Webseite angezeigt bekommen (siehe [Bild 6.7](#)).

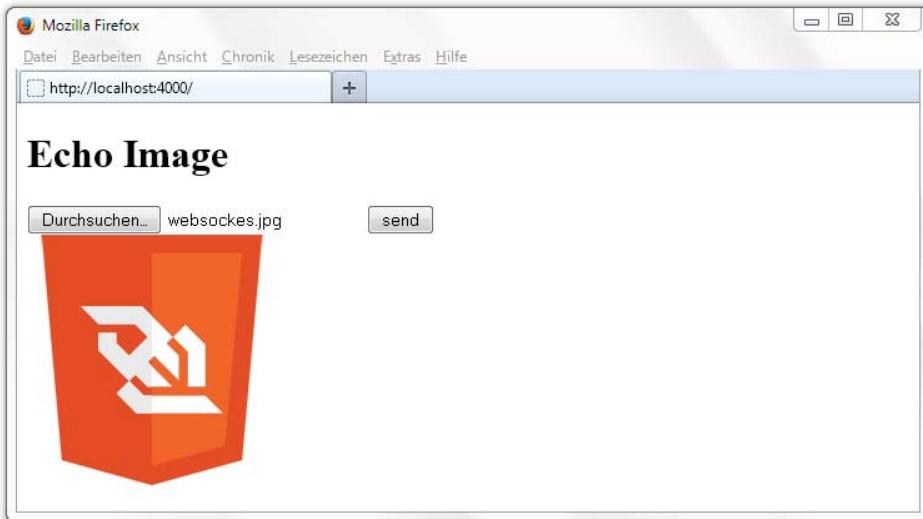


Bild 6.7 Darstellung der über den WebSocket empfangenen Bilddaten

Bevor das Bild in die bereits geladene Webseite eingefügt und dargestellt wurde, ist es anhand des WebSockets einmal zum Server und zurück übertragen worden. Wenn Sie diese HTML-Seite mit den Chrome Developer Tools untersuchen, können Sie folgende Beobachtungen machen (siehe [Bild 6.8](#)):

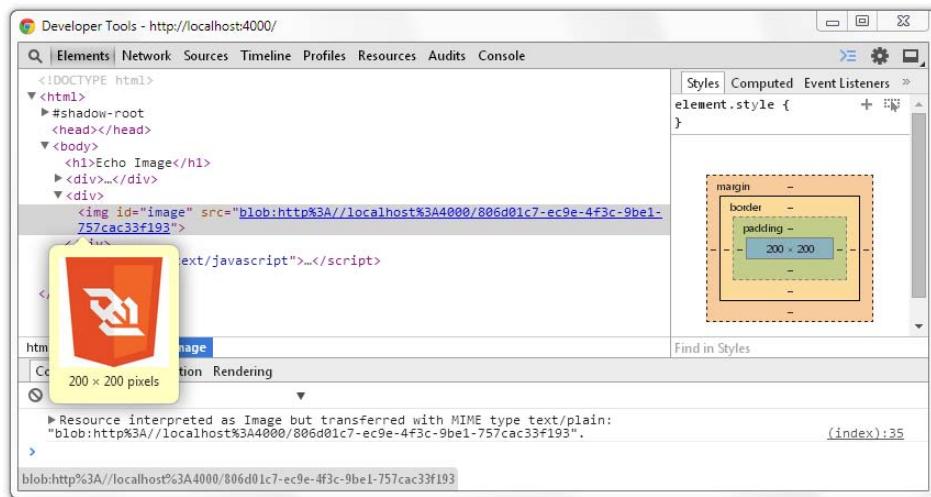


Bild 6.8 Inspeziere die eingebetteten binären Bilddateien

Es wurde für Sie eine spezielle URL generiert, welche zum Bild führt. Unter dem Punkt *Network* (siehe Bild 6.9) können wir auch erkennen, dass diese Datei aus dem Cache kommt.

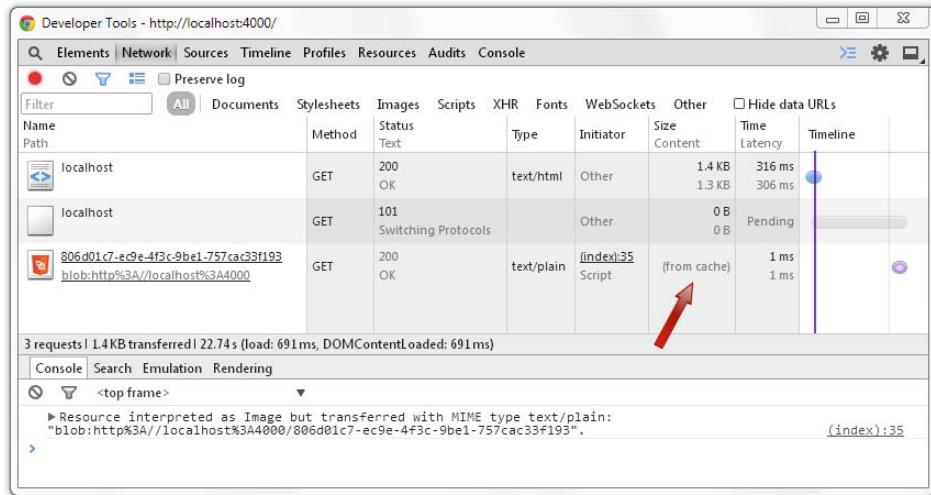


Bild 6.9 Zugriff auf Binärdaten

Noch interessanter ist für uns folgende Information in den WebSocket-Frames (siehe Bild 6.10):

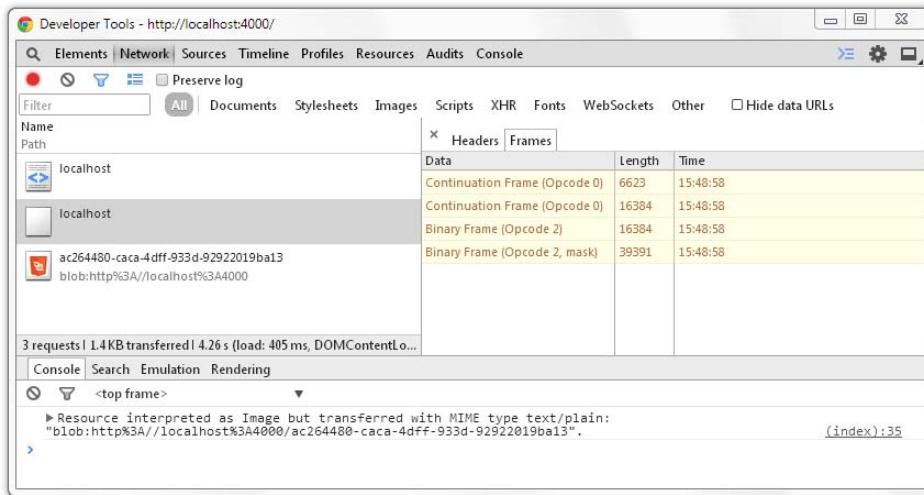


Bild 6.10 Analyse der ausgetauschten WebSocket-Frames

Da wir nicht jede dieser Implementierungen behandeln können, wollen wir uns auf eine Auswahl beschränken, die wir aktuell als repräsentativ und im breiten Einsatz sehen. Mit diesen Frameworks können Sie nicht nur WebSocket-Server implementieren, sondern auch die Clientseite, die nicht zur Ausführung an einen Webbrowser gebunden ist. Aus diesem Grund werden wir in den jeweiligen Kapiteln beide Seiten betrachten.

Wir sehen hier, dass wir einen binären Frame mit der Größe des Bildes zum Server gesendet haben. Der Server antwortet mit einem *Binary Frame* gefolgt von mehreren *Continuation Frames*. Der Grund dafür liegt in einer Standardeinstellung des WebSocket-Node Frameworks, wodurch Nachrichten ab einer Größe von 16 KB fragmentiert werden. Dieser Wert kann beliebig angepasst werden (siehe dazu [MS14]). Wir entnehmen daraus, dass der onmessage-Event-Handler erst aktiv wird, wenn die gesamte Nachricht bestehend aus den *Continuation Frames* vollständig zusammengesetzt ist.

Diese Beispielanwendung haben wir in den gängigen Browsern Firefox, Chrome, Opera, Internet Explorer und Safari getestet. Wir werden WebSocket-Node in [Abschnitt 7.2.3](#) wiederbegegnen, wenn wir, wie schon eingangs erwähnt, ein Zugriffskontrollsysteem für WebSocket-Server implementieren. Ein Wiedersehen gibt es auch in [Abschnitt 8.4](#), wenn wir mit Ihnen eine Beispielanwendung zur Videoüberwachung programmieren wollen.

■ 6.3 Vert.x

Die Antwort von Java auf Node.js ist Vert.x. Anstatt der JavaScript V8-Engine verwendet Vert.x die Java Virtual Machine (JVM) als Interpreter. Um Vert.x zu benutzen, benötigen Sie das JDK 1.7.0 oder spätere Versionen. Vert.x ist wie Node.js ein Kommandozeilenprogramm. Gegenüber Node.js hat Vert.x den Vorteil, nicht auf eine einzige Programmiersprache beschränkt zu sein. WebSocket-Programme können Sie mit Vert.x nicht nur in JavaScript, sondern auch in CoffeeScript, Java, Groovy, Ruby und Python entwickeln.

Derzeit kann Vert.x weder über einen Installer noch über einen Modulmanager installiert werden. Die Installation ist aber auch von Hand in wenigen Schritten getan. Laden Sie zunächst die aktuellste Vert.x-Version herunter⁶ und entpacken Sie das Archiv.

Wenn Sie danach den Unterordner *bin* zu Ihren PATH-Umgebungsvariablen hinzugefügt haben (siehe [Anhang C](#)), können Sie Vert.x über die Kommandozeile benutzen. Um zu testen, ob Sie alles richtig gemacht haben, geben Sie bitte folgenden Befehl in Ihr Terminal bzw. in die Eingabeaufforderung ein:

```
vertx version
```

Erscheint die Versionsnummer Ihrer Vert.x-Installation unterhalb des abgesetzten Kommandozeilenbefehls (siehe [Bild 6.11](#)), ist Ihr Vert.x betriebsbereit.

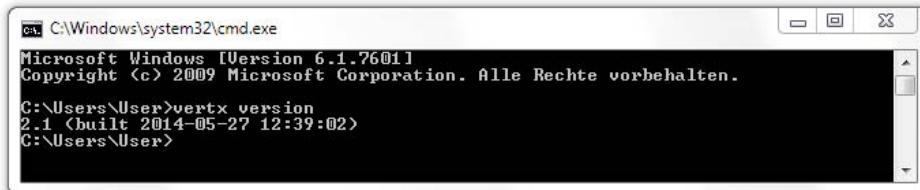


Bild 6.11 Erfolgreiche Installation von Vert.x

Auch Vert.x bringt, ähnlich wie Node.js, ein eigenes Paketmanagement mit. Um weitere Module zu installieren, steht für Sie das Kommando `vertx install` bereit. Für unser folgendes Beispiel muss allerdings kein Paket installiert werden, da Vert.x WebSockets bereits als Basismodul mitbringt.

Da wir in den vorangegangenen Kapiteln die Serverseite in JavaScript implementiert haben, werden wir für den Server in Vert.x dieses Mal auf Java zurückgreifen. Erzeugen Sie erneut einen Projektordner und darin zwei Dateien; eine Java-Datei mit z.B. dem Dateinamen *EchoServer.java* für den Server und eine HTML-Datei für den Testclient (*index.html*). Fügen Sie in die *EchoServer.java*-Datei folgenden Programmcode ein:

Listing 6.11 Echo-Server in Vert.x

```

1 import org.vertx.java.core.Handler;
2 import org.vertx.java.core.buffer.Buffer;
3 import org.vertx.java.core.http.HttpServerRequest;
4 import org.vertx.java.core.http.HttpServer;
5 import org.vertx.java.core.http.ServerWebSocket;
6 import org.vertx.java.platform.Verticle;
7
8 public class EchoServer extends Verticle {
9
10    public void start() {
11        HttpServer webServer = vertx.createHttpServer();
12        webServer.websocketHandler(new Handler<ServerWebSocket>() {
```

⁶ <http://vertx.io/downloads.html>

```

13     public void handle(final ServerWebSocket ws) {
14         ws.dataHandler(new Handler<Buffer>() {
15             public void handle(Buffer data) {
16                 ws.writeTextFrame(data.toString());
17             }
18         });
19     });
20 });
21
22 webServer.requestHandler(new Handler<HttpServerRequest>() {
23     public void handle(HttpServerRequest req) {
24         req.response.sendFile("index.html");
25     }
26 }).listen(4000); {
27
28     System.out.println("Der EchoServer laeuft auf dem Port
29         4000.");
30 }
```

Für den Testclient können Sie den Code aus [Listing 6.3](#) wiederverwenden. Unser Echo-Server hat die gleiche Funktionalität wie der WebSocket.io-Echo-Server aus [Abschnitt 6.2.2](#). Er gibt auf alle HTTP-Anfragen die HTML-Datei *index.html* zurück, welche als Testclient dient ([Zeile 22 bis 26](#)). Wird eine Nachricht an den WebSocket-Handler geschickt, gibt dieser die gleiche Antwort zurück ([Zeile 12 bis 20](#)). Starten Sie nun das Programm mit dem Befehl:

```
vertx run EchoServer.java
```

Es sollte der Eintrag "Der EchoServer laeuft auf dem Port 4000." auf der Konsole erscheinen. Öffnen Sie nun mit Ihrem Webbrowser die URL

```
http://localhost:4000/
```

Voilà! Sie sollten nun das gleiche Ergebnis wie in [Bild 6.5](#) auf [Seite 113](#) sehen. Ihre erste Vert.x-Anwendung haben Sie somit erfolgreich gemeistert.



Die komplette Anwendung können Sie ebenfalls auf unserer Webseite herunterladen. Dort befindet sich auch eine JavaScript-Variante des Echo-Servers.

Vert.x wird Ihnen in [Kapitel 8](#) wiederbegegnen, wenn wir einen einfachen Chat realisieren. Mehr Informationen finden Sie natürlich auf der Internetseite von Vert.x.⁷

⁷ <http://vertx.io/>

■ 6.4 Play Framework

Das Play Framework ist ein freies quelloffenes MVC-Web-Framework. Webapplikationen können darin in Java oder Scala geschrieben werden. Play verfolgt das Ziel, möglichst schnell und einfach Webanwendungen zu entwickeln. Zusätzliche Features bietet Play in puncto „Hot-Code-Reloading“, „Convention over Configuration“ und der Fehleranzeigen im Browser. Play unterstützt WebSockets seit der Version 1.2. Einen eigenen Webserver (Netty) bringt Play auch mit, sodass Sie keinen zusätzlich installieren müssen. Ein weiterer Vorteil von Play ist z. B., dass Cookies zustandslos sind. Diese werden nicht auf dem Server gespeichert, sondern mit einem Secret-Key signiert. Solange Sie im Besitzt dieses geheimen Schlüssels sind, können Sie Ihre Play-Anwendungen auf beliebig viele Server verteilen. Sie brauchen also z. B. keinen Application-Server wie bei JavaEE, der dafür sorgt, dass Sessions umgezogen werden.

Wir möchten Ihnen in diesem Kapitel eine kleine Einführung in das Play Framework geben, damit Sie das Konzept und die Anwendungslogik dieses Frameworks lernen und verstehen. Auf dieser Grundlage wird dann die Entwicklung eines Echo-Servers näher erläutert.

6.4.1 Installation

Play-Anwendungen sind unter Linux, Windows und Mac OS ausführbar.

Da dieses Framework auf Java bzw. Scala basiert, brauchen Sie das JDK 6 oder höher. Das aktuelle JDK können Sie von der Oracle Homepage beziehen.⁸ Wenn Sie ein aktuelles JDK installiert haben, laden Sie sich bitte das Play Framework herunter.⁹ Entpacken Sie anschließend das Zip-File und fügen Sie diesen Ordner zu Ihren Umgebungsvariablen hinzu (siehe [Anhang C](#)). Öffnen Sie danach bitte die Eingabeaufforderung oder das Terminal und geben Sie diesen Befehl ein:

```
activator
```

Folgendes sollte jetzt auf der Konsole erscheinen (siehe [Bild 6.12](#)):

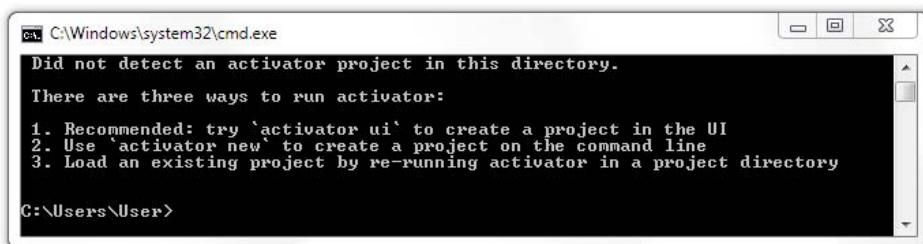


Bild 6.12 Erste Berührungen mit Play

⁸ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

⁹ <http://www.playframework.com/download/>

6.4.2 Anlegen eines neuen Play-Projekts

Wir erzeugen nun durch das Kommando activator new ein neues Projekt mit dem Namen echo_play.

```
C:\Windows\system32\cmd.exe
C:\Users\User>activator new echo_play
Fetching the latest list of templates...
Browse the list of templates: http://typesafe.com/activator/templates
Choose from these featured templates or enter a template name:
  1) minimal-java
  2) minimal-scala
  3) play-java
  4) play-scala
(hit tab to see a list of all templates)
> 3
OK, application "echo_play" is being created using the "play-java" template.
To run "echo_play" from the command line, "cd echo_play" then:
C:\Users\User\echo_play>activator run

To run the test for "echo_play" from the command line, "cd echo_play" then:
C:\Users\User\echo_play>activator test

To run the Activator UI for "echo_play" from the command line, "cd echo_play" then:
C:\Users\User\echo_play>activator ui

C:\Users\User>
```

Bild 6.13 Java oder Scala?

```
activator new echo_play
```

Sie können sich zwischen einem Scala-Projekt und einem Java-Projekt entscheiden. Wir wählen die 3 für ein Java-Projekt. Play erzeugt daraufhin ein Projekt mit einer festgelegten Ordnerstruktur.

6.4.3 Anatomie einer Play-Anwendung

Die Ordnerstruktur, sozusagen die Anatomie einer Play-Anwendung, sehen Sie in [Bild 6.14](#).

Der Ordner *app* beinhaltet die ganzen ausführbaren Quelltexte der Anwendung. Unterteilt ist dieser Ordner in *controllers*, *models* und *views*. Im Ordner *conf* befinden sich zwei Dateien. Mit der Datei *application.conf* können Sie den Server Ihrer Anwendung konfigurieren. Im Ordner *public* können Sie statische Dateien, wie JavaScript-Dateien, Stylesheets oder z. B. Bilder ablegen. Die anderen Ordner *target*, *test* und *projects* sind für den Anfang erst mal nicht relevant. Daher werden wir im weiteren Verlauf nicht näher darauf eingehen. Für die Anwendungen, die wir in diesem Kapitel und im [Abschnitt 7.2](#) implementieren wollen, sind nur die Ordner *app/controllers*, *app/views* und die Datei *routes* im Ordner *conf* von Bedeutung.

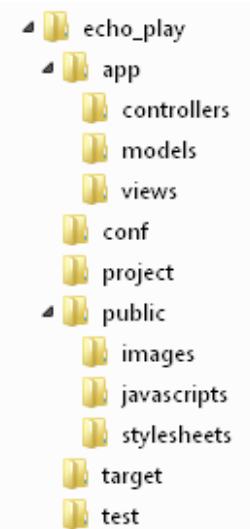


Bild 6.14 Anatomie einer Play-Anwendung

6.4.4 Die Play-Konsole

Play bringt seine eigenen Konsolenbefehle mit. Durch die Eingabe von `activator` in der Eingabeaufforderung oder dem Terminal wird die Konsole gestartet. Anschließend folgen die Befehle. Die wichtigsten stellen wir Ihnen in [Tabelle 6.2](#) vor.

Tabelle 6.2 Konsolenbefehle von Play [Pla14]

Befehl	Beschreibung
<code>new Name</code>	Erstellt ein neues Projekt mit dem gewählten Namen
<code>run</code>	Startet die Play-Anwendung im Entwicklungsmodus, wodurch „Hot-Code-Deployment“ möglich ist
<code>~run</code>	Startet die Play-Anwendung im Entwicklungsmodus und kompiliert den Code direkt nach dem Abspeichern. Der Code wird also nicht erst nach dem Refreshen des Browsers kompiliert.
<code>compile</code>	Kompiliert den gesamten Code der Play-Anwendung

Sie können zuerst die Play-Konsole mit dem Befehl `activator` starten und dann die Befehle eingeben oder den `activator`-Befehl zusammen mit dem Kommando eintippen (z. B. `activator run`).

6.4.5 Controller in Play

Das Play Framework schmückt sich mit den Worten „Convention over Configuration“. Diese Konvention wollen wir nun einhalten und erzeugen alle Controller im Ordner `app/controllers`.

trollers. Zu beachten ist, dass all diese Controller von der Klasse `play.mvc.Controller` erben müssen. Wir wollen uns einmal die Klasse `Application.java`, welche bereits im Ordner `controllers` vorhanden ist, in [Listing 6.12](#) näher ansehen.

Listing 6.12 Controller in Play

```

1 ...
2 public class Application extends Controller {
3
4     public static Result index() {
5         return ok(index.render("Ihre Applikation ist bereit."));
6     }
7 }
```

Controller-Klassen beinhalten statische Methoden, die als *Actions* bezeichnet werden und ein *Result*-Objekt zurückgeben. Die Klasse *Result* repräsentiert eine HTTP-Antwort, welche zum Client zurückgeschickt wird. Sie enthält sehr viele Hilfsmethoden, um z. B. den Statuscode festzulegen. Die Methode `ok()` liefert demnach eine HTTP-Antwort mit dem Statuscode 200. Für einen HTTP-Client bedeutet dies, dass die Anfrage erfolgreich bearbeitet wurde. Andere Statuscodes, wie z. B. (Page Not Found) oder 500 (Internal Server Error), können mithilfe anderer Methoden genutzt werden. Darauf werden wir hier allerdings nicht weiter eingehen. Für uns ist erst mal wichtig, dass wir den Status 200 wiedergeben können.

In der Methode `ok()` können jegliche Arten von Nutzdaten (der Payload) übergeben werden, wie z. B. ein normaler Text, Views oder auch Binärdaten. In diesem Beispiel übergeben wir die View `index`, die sich im Verzeichnis `apps/views` befindet. Jede View beinhaltet eine `render`-Methode, der wir auch zusätzliche Variablen mitgeben können.

6.4.6 Views in Play

Wie zuvor erwähnt, liegen die Views in dem Ordner `apps/views`. In diesen Views kann lediglich Scala-Code eingebettet werden. Zwei davon sind bereits vordefiniert. Die Datei `main.scala.html` dient hier als Layout-Template.

Alle Scala-Anweisungen fangen mit einem @ an. In [Zeile 1 \(Listing 6.13\)](#) werden direkt zwei Variablen deklariert; eine Variable mit dem Namen `title`, die den Datentyp `String` besitzt. Die andere Variable mit dem Namen `content` besitzt den Datentypen `Html`. In dieser Variablen wird der HTML-Content aller Views, die dieses Layout-Template erweitern, eingebettet. Im nächsten Schritt muss nur noch festgelegt werden, wo die Variablen am besten platziert werden sollen. Die `title`-Variable wird in den gleichnamigen `<title>`-Tag gelegt. Da noch kein wirkliches Layout vorhanden ist, wird der `content` erst mal zwischen den `<body>`-Tags platziert.

Listing 6.13 Das Layout-Template

```

1 @(title: String)(content: Html)
2
3 <!DOCTYPE html>
```

```

4
5 <html>
6   <head>
7     <title>@title</title>
8     <link rel="stylesheet" media="screen"
9       href="@routes.Assets.at("stylesheets/main.css")">
10    <link rel="shortcut icon" type="image/png"
11      href="@routes.Assets.at("images/favicon.png")">
12    <script src="@routes.Assets.at ("javascripts/hello.js")"
13      type="text/javascript"></script>
14  </head>
15  <body>
16    @content
17  </body>
18 </html>
```

In [Listing 6.14 \(*index.scala.html*\)](#) wird das Layout-Template erweitert. Die `main`-Methode ruft das Layout-Template `main.scala.html` auf. Als erster Parameter wird der Text "Welcome to Play" übergeben und gleichzeitig der `title`-Variablen des Layout-Templates zugewiesen. Alternativ kann auch `@main("Welcome to Play")` verwendet werden. Da wir im Layout-Template als Erstes die `title`-Variable deklariert und platziert haben, erkennt Play dies automatisch und weist diesen Text der Variablen `title` zu. Nutzen Sie die Variable, welche Ihnen lieber ist. In den geschweiften Klammern der `main`-Methode wird der HTML-Inhalt des Layout-Templates erweitert. Alle Inhalte, die sich in diesen Klammern befinden, werden in die `content`-Variable des Layout-Templates aus [Listing 6.13](#) eingefügt. Die `play20.welcome()`-Methode ist ein schon vorher definiertes Welcome-Template.

Listing 6.14 Erweitern des Layout-Templates

```

1  @(message: String)
2  @main(title = "Welcome to Play") {
3    @play20.welcome(message, style = "Java")
4 }
```

In der ersten Zeile des [Listing 6.14](#) wird noch eine Variable des Datentyps `String` deklariert. Dies muss gemacht werden, da in der `render()`-Methode des Controllers ein `String` übergeben wird. Alle Variablen, die Sie vom Controller in die View übergeben, müssen nochmals in der View deklariert werden.

6.4.7 Routes in Play

Die Datei `routes` im Ordner `conf` fungiert als Router, der die URL jeder Anfrage prüft und den Request an die entsprechende Controller-Methode weiterleitet. In der bereits vorhandenen `routes`-Datei ist bereits ein Pfad festgelegt.

```
GET      /           controllers.Application.index()
```

Hier wird die Methode `index()` aufgerufen, falls das Wurzelverzeichnis angesprochen wird, anders als in der folgenden Beispielzeile:

```
GET      /getUsers    controllers.Application.getUsers()
```

In dieser Zeile wird die Methode `getUsers()` ausgeführt, wenn ein Client über einen GET-Request `/getUsers` aufruft. Die Methode befindet sich in der Controller-Klasse `Application` aus dem Paket `controllers`. Natürlich können Sie diese `routes`-Datei beliebig erweitern.

```
POST /addUser controllers.Application.addUser()
```

Hier z. B. definieren Sie einen zweiten Pfad, der die Methode `addUser()` aufruft, sobald ein HTTP-POST-Request vom Client an die URL `/addUser` geschickt wird. Die Methoden PUT, DELETE und HEAD werden ebenfalls unterstützt.

6.4.8 Vom Controller zur View

Wir wollen nun das Wissen aus den vorherigen Kapiteln verinnerlichen und den ganzen Weg von der Erstellung einer Controller-Action bis hin zur Darstellung der View selbst durchführen. Erstellen Sie in dem Controller `Application` im Paket `app/controller` eine neue Action bzw. Methode. Wir wollen ein typisches „Hallo Welt“-Beispiel nehmen und nennen unsere Action `halloWelt()`.

Listing 6.15 Erste eigene Controller-Action

```
1 public static Result halloWelt() {
2     String message = "Hallo Welt";
3     return ok(halloWelt.render(message));
4 }
```

Wir initialisieren eine Variable `message` mit dem Inhalt "Hallo Welt". Der Action wird mit der Methode `ok()` ein Rückgabewert übergeben, der die `halloWelt`-View rendert und ihr zugleich die Variable `message` übergibt. Achten Sie darauf, dass Sie vorher alle Views mit dem Befehl `import views.html.*` importieren. In dem Controller `Application` sollte dies schon vorhanden sein. Wundern Sie sich außerdem nicht, falls Sie mit Eclipse arbeiten, dass eine Fehlermeldung unter der Render-Methode `halloWelt.render()` ([Zeile 3](#)) erscheint. Das liegt daran, dass die View zu diesem Controller noch nicht erzeugt wurde. Der Fehler verschwindet, sobald wir diese View erstellt haben, den Play-Code kompilieren und das Projekt in Eclipse aktualisieren.

Damit wir etwas sehen, müssen wir noch eine passende View mit demselben Namen erstellen. Wir erzeugen eine neue View im Paket `app/views` und nennen diese `halloWelt.scala.html`. Alle HTML-Views müssen in Play mit `.scala.html` enden. In dieser View fügen wir folgenden Code aus [Listing 6.16](#) hinzu:

Listing 6.16 Erste eigene View

```
1 @(message: String)
2 @main(title="Hallo Welt"){
3     <h1>@message</h1>
4 }
```

Zuerst muss die Variable `message` deklariert werden, da wir diese mit dem Controller übergeben wollen. Erweitert wird diese View vom `main`-Template. Der Titel soll ebenfalls "Hallo Welt" sein. Die `message`-Variable mit dem Inhalt wird in einem `<h1>`-Tag eingefügt.

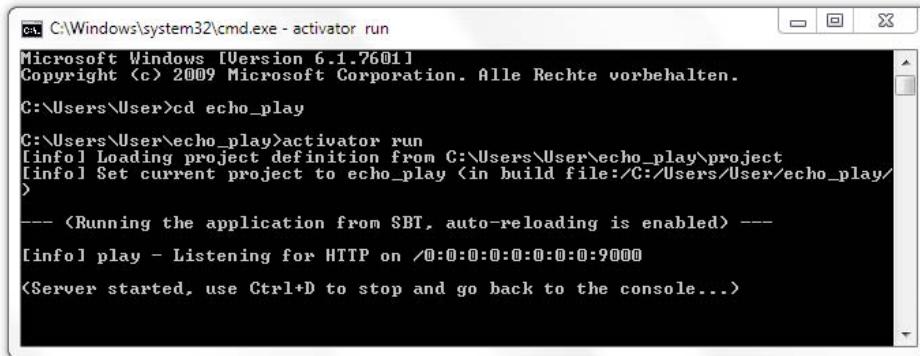
Als letzten Schritt müssen wir noch einen Pfad in der `routes`-Datei für unsere Action definieren:

```
GET      /halloWelt      controllers.Application.halloWelt()
```

Nun ist Ihre erste Play-Action definiert. Um Ihr Ergebnis zu sehen, öffnen Sie bitte die Eingabeaufforderung oder das Terminal, wechseln Sie in das Verzeichnis Ihres Play-Projektes und starten Sie die Anwendung mit dem Befehl:

```
activator run
```

Wenn Sie alle Schritte richtig gemacht haben, sollte die Konsole wie in [Bild 6.15](#) aussehen.



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe - activator run'. The window displays the following text:

```
C:\Windows\system32\cmd.exe - activator run
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\User>cd echo_play
C:\Users\User\echo_play>activator run
[info] Loading project definition from C:\Users\User\echo_play\project
[info] Set current project to echo_play (in build file:/C:/Users/User/echo_play/
)
--- <Running the application from SBT, auto-reloading is enabled> ---
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
<Server started, use Ctrl+D to stop and go back to the console...>
```

Bild 6.15 Starten einer Play-Anwendung

Nun können Sie Ihre erste Play-Anwendung im Browser aufrufen (siehe [Bild 6.16](#)).

Wenn Sie

`http://localhost:9000`

in Ihrem Browser aufrufen, erscheint das `Welcome`-Template von Play. Unter

`http://localhost:9000/halloWelt`,

sollte die vorhin erstellte `halloWelt`-Action dargestellt werden (siehe [Bild 6.17](#)).

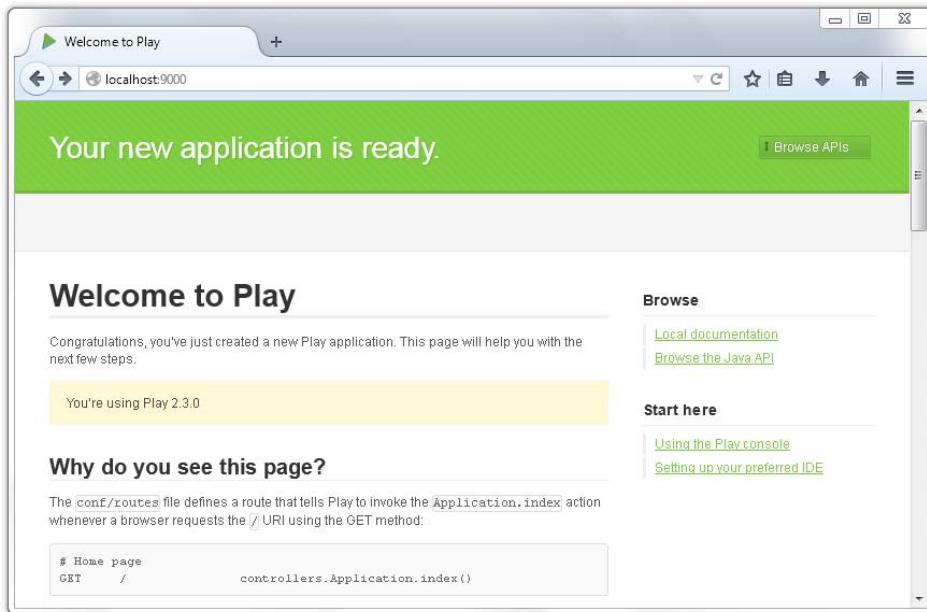


Bild 6.16 Startseite der ersten Play-Anwendung



Bild 6.17 Aufruf unserer ersten Action

6.4.9 Erstellen eines Echo-Servers in Play

Da wir nun die ersten Hürden überwunden haben, wollen wir unsere Anwendung im nächsten Schritt zu einem WebSocket-Server erweitern. Dieser WebSocket-Server soll, wie in den vorherigen Kapiteln mit Node.js und Vert.x, einen Echo-Server repräsentieren. Das Play Framework ist, ähnlich wie Node.js und Vert.x, ein ereignisorientiertes Framework. Dies spiegelt sich deshalb ebenfalls in den WebSockets wider. Es werden Event-Handler für z. B. das Ankommen einer Nachricht registriert, zu denen dann ein Callback definiert werden kann.

Öffnen Sie nun die Datei *echo_play/app/controllers/Application.java* und fügen Sie die folgende Controller-Methode hinzu:

Listing 6.17 Controller-Methode für einen Echo-Server in Play

```

1 import play.libs.F.Callback;
2 import play.libs.F.Callback0;
3 ...
4 public static WebSocket<String> ws() {
5     return new WebSocket<String>() {
6         @Override
7         public void onReady( play.mvc.WebSocket.In<String> in,
8             final play.mvc.WebSocket.Out<String> out) {
9             System.out.println("Handshake abgeschlossen");
10            in.onMessage(new Callback<String>() {
11                @Override
12                public void invoke(String event) {
13                    out.write(event);
14                }
15            });
16            in.onClose(new Callback0() {
17                public void invoke() throws Throwable {
18                    System.out.println("WebSocket geschlossen");
19                }
20            });
21        };
22    }
}

```

Bei Actions, die WebSockets behandeln, muss kein Result-Objekt zurückgegeben werden, sondern ein generisches WebSocket-Objekt. Mit dem Typparameter des WebSocket-Objekts kann bestimmt werden, welche Art der Daten durch die WebSockets verschickt werden. Will man z. B. Texte verschicken, wird der Datentyp String als Typparameter verwendet (WebSocket<String>).

Das WebSocket-Objekt enthält eine abstrakte onReady()-Methode, die wir implementieren müssen. Diese wird aufgerufen, sobald der Opening-Handshake erfolgreich war. Es werden ein WebSocket.In-Objekt und ein WebSocket.Out-Objekt übergeben. Sie können als ein einfließender Kanal und als rausfließender Kanal betrachtet werden. Das WebSocket.In-Objekt enthält zwei Listener-Methoden; eine onMessage()-Methode, mit der wir bestimmen können, was passiert, wenn eine Nachricht am Server ankommt. In der onClose()-Methode kann bestimmt werden, was passiert, wenn der Server einen Close-Frame empfängt. Jedem dieser Listener-Methoden muss ein Callback-Objekt übergeben werden. Das Callback-Objekt ist ein Interface mit einer Methode invoke(), die bestimmt, was im Callback passieren soll. Das Out-Objekt ist unser Kommunikationskanal nach außen. In diesem Objekt existieren zwei Methoden, die write()- und die close()-Methode. Mit write() können wir Daten an den Absender zurückschicken und mit close() eine Verbindung schließen.

Da wir einen Echo-Server programmieren wollen, erzeugen wir im Fall onMessage ein Callback-Objekt, das dieselbe Nachricht an den Client zurückschickt.

Nun haben wir eine WebSocket-Action definiert. Im nächsten Schritt muss bestimmt werden, mit welcher URL diese Action bzw. Methode angesprochen werden soll.

Verändern Sie nun die *routes*-Datei in *echo_play/conf* und fügen Sie folgenden Eintrag hinzu:

```
GET      /ws      controllers.Application.ws()
```

Die GET-Methode wird hier verwendet, da der WebSocket-Handshake mit einem GET-Request beginnt.

Jetzt müssen wir nur noch einen Client definieren, mit dem wir unseren Echo-Server testen können. Das machen wir wieder mit einer ganz normalen HTML-Seite. Dazu erzeugen wir eine neue View (z. B. *echo.scala.html*) im Paket *app/views*. Dieser fügen wir folgenden Code hinzu:

Listing 6.18 View für einen Echo-Server in Play

```
1 @main("WebSocket Server Test -- Play Framework") {
2     <h1> WebSocket Echo-Server mit dem Play Framework</h1>
3     <script>
4         var ws = new WebSocket('ws://'+window.location.host+'/ws');
5         ws.onmessage = function(event) {
6             console.log(event.data);
7         }
8         ws.onopen = function(event) {
9             ws.send('Hallo');
10        }
11    </script>
12 }
```

Jetzt muss nur noch die passende Action dazu implementiert werden. Diese soll einfach nur die dazugehörige View rendern:

Listing 6.19 Rendern der View

```
1 public static Result echo() {
2     return ok(echo.render());
3 }
```

Für diese Action muss noch ein passender Eintrag in der *routes*-Datei gemacht werden:

```
GET      /echo      controllers.Application.echo()
```

Jetzt ist unsere erste WebSocket-Anwendung unter Play betriebsbereit und wir können diese starten:

```
activator run
```

Öffnen Sie die JavaScript-Konsole in Ihrem Browser und geben Sie

```
http://localhost:9000/echo
```

in der Adresszeile ein. Das Ergebnis sollte wie in [Bild 6.18](#) aussehen.



Bild 6.18 Ihr erster WebSocket-Server mit Play

Mehr Informationen zum Play Framework erfahren Sie auf der gleichnamigen Internetseite.¹⁰

Wir begegnen dem Play Framework in [Abschnitt 7.2](#) wieder, wenn wir mit dessen Hilfe eine Zugriffskontrolle auf einen WebSocket-Endpoint definieren.

■ 6.5 JSR 356

Im Java-Umfeld spielt der Einzug von WebSockets insbesondere für Java Enterprise eine wichtige Rolle, da hier alle Technologien zur Entwicklung verteilter (Web-)Anwendungen beherbergt sind. Wie üblich erfolgt die Einführung neuer Technologien über einen festgelegten Prozess. Der 1998 etablierte *Java Community Process* (JCP) [Ora14a] dient der Weiterentwicklung der Programmiersprache Java und ihrer Standardbibliothek.

Jede Erweiterung muss einen bestimmten Prozess durchlaufen. Die Erweiterungen werden *Java Specification Request* (JSR) genannt und einfach durchnummieriert. Die Liste aller JSRs können Sie sich auf der JCP-Webseite anschauen.¹¹ Den Prozess, den solch ein JSR durchlaufen muss, ist in [Bild 6.19](#) dargestellt.

Ein oder mehrere Mitglieder des JCP können einen Vorschlag für eine Erweiterung in Form eines JSR machen (Phase 1). Erfüllt der Inhalt des JSR die Rahmenbedingungen und Voraussetzungen, muss noch das *Executive Committee* (EC), das von den Mitgliedern des JCP im Vorfeld bestimmt wird, den JSR annehmen. Wird der Vorschlag angenommen, wird in Folge eine Expertengruppe gebildet, die den JSR mit Expertenwissen weiterbringen soll (Phase 2). Diese bildet zunächst einen Early Draft. Auf dieser Grundlage arbeitet die Expertengruppe dann weiter bis zu einem Public Draft, den alle Interessierten begutachten und

¹⁰ <http://playframework.com/>

¹¹ <http://jcp.org/ja/jsr/all/>

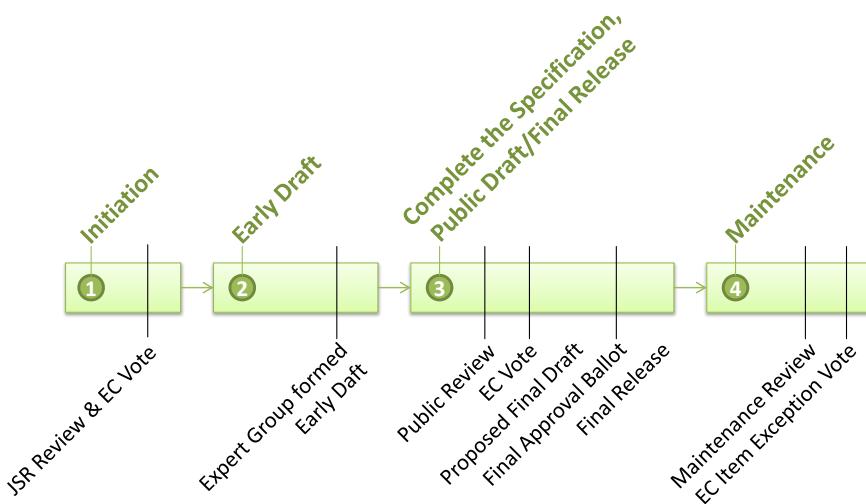


Bild 6.19 JCP-Standardisierungsprozess [Ora14b]

kommentieren können (Phase 3). Aus dem erhaltenen Feedback wird eine abschließende Version des JSR erarbeitet. Wird diese erneut vom EC bestätigt, entwickelt die Expertengruppe eine Referenzimplementierung. Nachdem diese begutachtet wurde, stimmt das EC endgültig über die Annahme des JSR ab. Ist auch diese letzte Abstimmung erfolgreich, wird das JSR offizieller Bestandteil der Sprache bzw. der Standardbibliothek. Um die Relevanz und Gültigkeit eines verabschiedeten JSR zu gewährleisten, wird dieser in regelmäßigen Abständen begutachtet (Phase 4). Dies kann zu einer Bestätigung, einer Überarbeitung oder der Stilllegung des JSR führen.

Wenn Sie in der Liste aller JSRs nach der Zahl 356 suchen, dann stoßen Sie auf den JSR mit dem aussagekräftigen Titel „Java™API for WebSocket“. Der Kurzansicht können Sie entnehmen, dass sich der JSR 356 in Phase 3 befindet und seit dem 22. Mai 2013 den Status *Final Release* erreicht hat. Dies bedeutet, dass es für die JSR 356 in diesem Zustand eine Spezifikation mit Referenzimplementierung gibt und dass die finale Entscheidung zur Aufnahme in die Java-Standardbibliothek (genauer gesagt in die Standard-APIs der Java Enterprise Edition, JEE) gefallen ist.

JEE 7 enthält aus diesem Grund WebSockets gemäß der JSR 356. Die Referenzimplementierung steht unter der CDDL 1.0-Lizenz [Ope14] und kann von der Java-Homepage heruntergeladen werden.¹² Eine weitere Open-Source-Referenzimplementierung namens *Tyrus*, die ebenfalls unter der CDDL 1.0-Lizenz steht, ist ebenfalls verfügbar.¹³ Auf Tyrus basiert auch die WebSocket-Implementierung des GlassFish 4-Applikationsservers.

¹² <http://java.net/projects/websocket-spec/>

¹³ <http://tyrus.java.net/>

Trotz des verabschiedeten *Final Release* des JSR 356 ist noch Vorsicht an den Tag zu legen, wenn Sie nach Codebeispielen im Web suchen. Dort finden Sie viele Blogs und Tutorials, die sich auf frühere Integrationen in Betaversionen gängiger Applikationsserver beziehen. Diese berücksichtigen nicht die aktuellen Gegebenheiten des Final Release, sondern basieren noch auf älteren Versionen wie dem *Early Draft*. Bei den nachfolgenden Be trachtungen halten wir uns natürlich an die Spezifikationen des Final Release. Praktische Beispiele setzen dabei auf den Implementierungen des aktuellen GlassFish 4-Servers auf. Der GlassFish-Applikationsserver ist ab der Java EE-Version NetBeans 7.3.1 enthalten. Die neueste Version können Sie von der NetBeans-Webseite herunterladen.¹⁴ Nach der Installation erzeugen Sie bitte ein neues Projekt unter *File → New Project...*

Wählen Sie dort unter *Categories* die Option *Java Web* und auf der rechten Seite unter *Projects* die Option *Web Application* aus (siehe Bild 6.20).

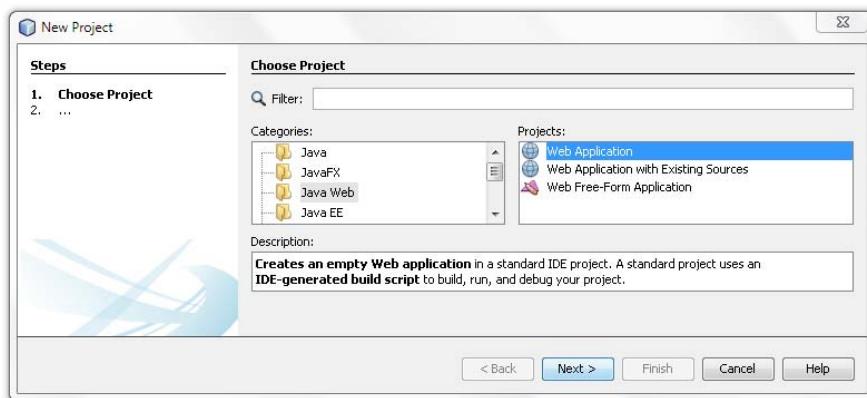


Bild 6.20 Erzeugung einer Webapplikation in NetBeans

Drücken Sie bitte danach auf *Next >*, um einen Projektnamen auszuwählen. Mit einem weiteren Klick auf *Next >* können Sie den Server und die Java EE-Version einstellen (siehe Bild 6.21).

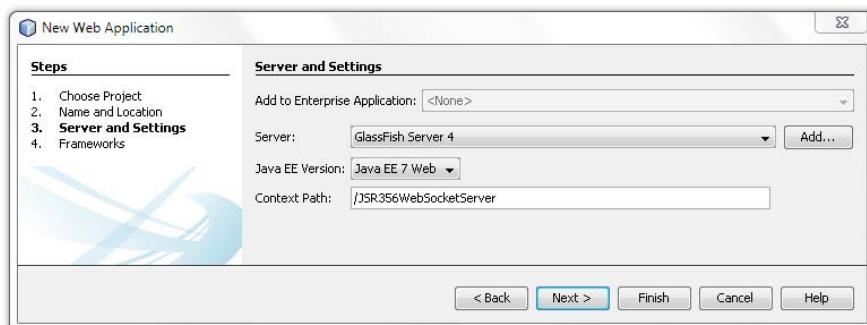


Bild 6.21 Applikationsserver und Java EE-Version einstellen in NetBeans

¹⁴ <https://netbeans.org/downloads/index.html>

Wählen Sie den *GlassFish Server 4.0* und die *Java EE 7 Web* aus. Klicken Sie danach entweder auf *Finish* oder auf *Next >*, um weitere Frameworks einzubinden. Falls Sie eine ältere Version von NetBeans besitzen, können Sie den GlassFish-Server 4.0 alternativ über die Servereinstellungen einbinden. Den GlassFish 4-Applikationsserver können Sie über die Internetseite beziehen.¹⁵ Wenn Sie das heruntergeladene Archiv entpackt haben, müssen Sie den Server zunächst in Ihre Entwicklungsumgebung integrieren. Richten Sie hier einen neuen GlassFish-Server ein und geben Sie dann das Verzeichnis Ihrer GlassFish 4-Installation an. Benennen sollten Sie den Server nach den tatsächlichen Gegebenheiten, also *GlassFish Server 4* (siehe Bild 6.22).

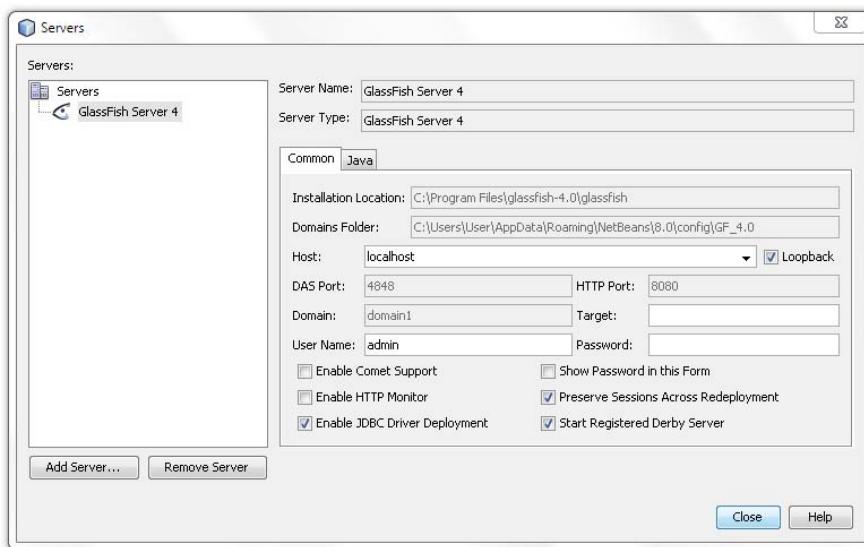


Bild 6.22 Installation des GlassFish 4.0-Servers in der NetBeans-IDE

Die Installation eines GlassFish 4-Servers in andere gängige IDEs wie z. B. Eclipse oder IntelliJ erfolgt äquivalent. Ist der Server der IDE bekannt, kann auf diesen im Rahmen von Webprojekten als Laufzeitumgebung zugegriffen werden. Auch hier ist die prinzipielle Vorgehensweise bei den verschiedenen IDEs ähnlich. Die Bezeichnungen und der Zugang zu den Funktionen variieren wahrscheinlich ein wenig. Bei NetBeans müssen Sie in der Kategorie *Java Web* ein *Web Application* Projekt anlegen. In Eclipse wählen Sie aus der *Web*-Kategorie ein *Dynamic Web Project* aus. Danach müssen Sie zu den grundlegenden Projekteinstellungen immer auch eine Server-Laufzeitumgebung angeben, aus der die APIs entnommen werden, die dann zur Entwicklung der Webanwendung bereitstehen. Wenn Sie dies in der NetBeans-IDE durchgeführt haben, sollte ein leeres Projekt bereitstehen, das die Bibliotheken Ihres GlassFish 4-Servers enthält. Hier können Sie nun schnell überprüfen, ob die Implementierungen der JSR 356 darunter sind. Dazu klappen Sie den Ordner *Libraries* auf. Hier sollte Ihr *GlassFish Server 4.0* enthalten sein. Wenn Sie diesen wieder-

¹⁵ <https://glassfish.java.net/>

um aufzuklappen, erscheinen alle von ihm bereitgestellten Bibliotheken in Form von JAR-Dateien. In der Liste der verfügbaren JARs sollte eines mit dem Namen *javax.websocket-api.jar* auftauchen (siehe Bild 6.23).

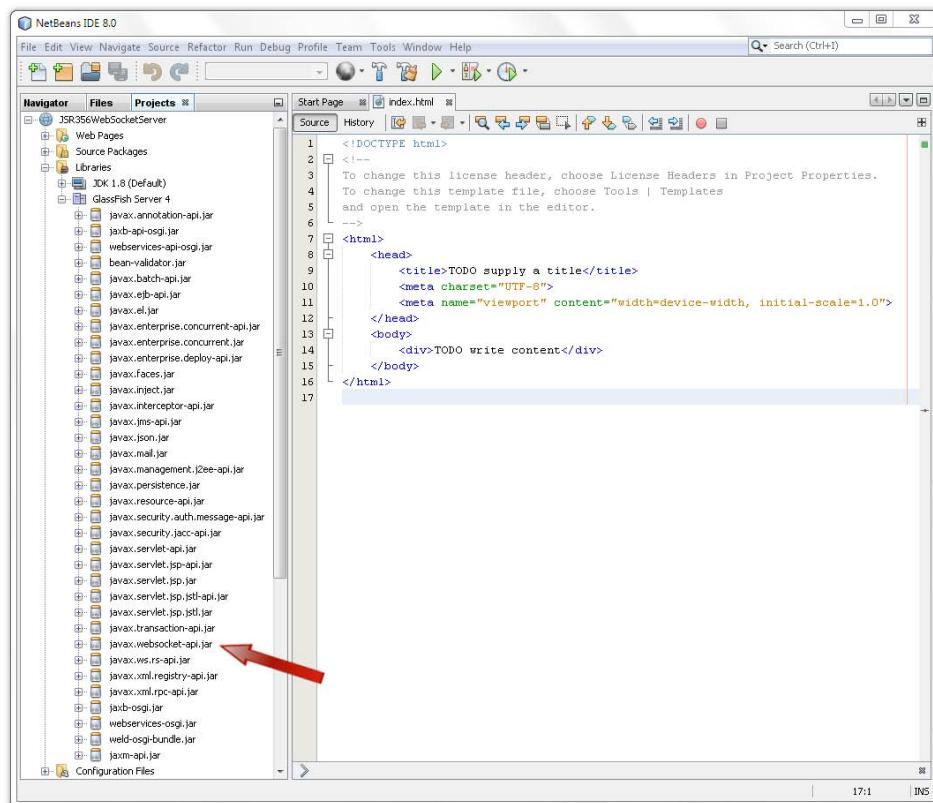


Bild 6.23 NetBeans Java Web Application-Projekt

Wenn Sie ein solches Projekt in der IDE Ihrer Wahl erstellt haben, können Sie mit der Implementierung Ihrer ersten WebSocket-Clients und -Server anhand der JSR 356 beginnen. Der JSR 356 führt ein neues Paket ein, in dem alle Klassen und Interfaces zur Implementierung von WebSocket-Clients und -Servern gebündelt sind. Das Paket trägt den treffenden Namen *javax.websocket*.

6.5.1 JSR 356-basierender WebSocket-Server

Einen WebSocket-Server-Endpoint bei NetBeans erzeugen Sie per Rechtsklick auf das jeweilige Projekt. Anschließend wählen Sie *New → WebSocket Endpoint...* aus. Falls die Option *WebSocket Endpoint...* nicht erscheint, gehen Sie bitte auf *New → Other...* (siehe Bild 6.24).

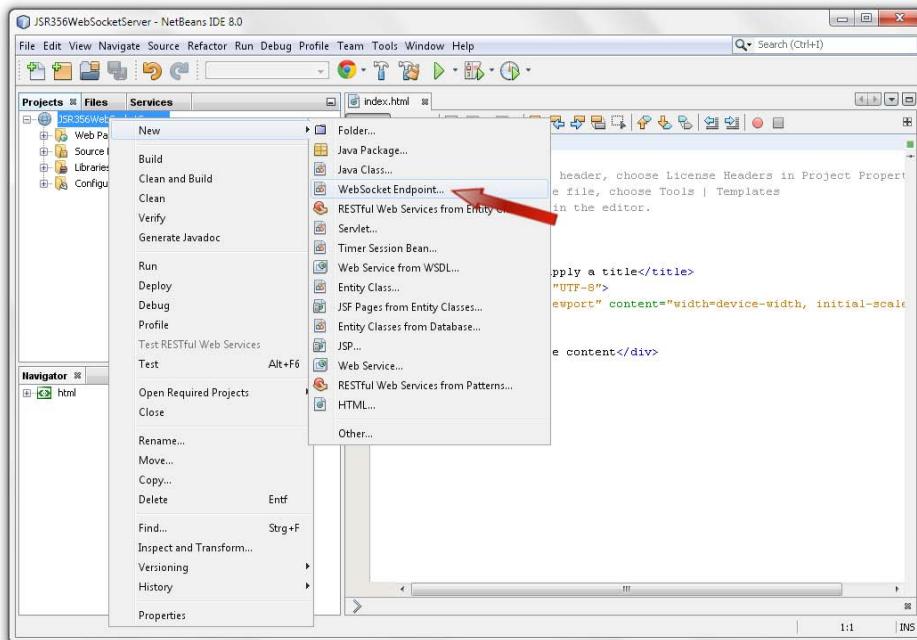


Bild 6.24 Neuen WebSocket-Endpoint erzeugen in NetBeans

Dort wählen Sie die Projektkategorie *Web* und den Dateityp *WebSocket Endpoint* aus und erzeugen so einen neuen Endpoint (siehe Bild 6.25). In Eclipse ist leider keine ähnliche Funktion vorhanden.

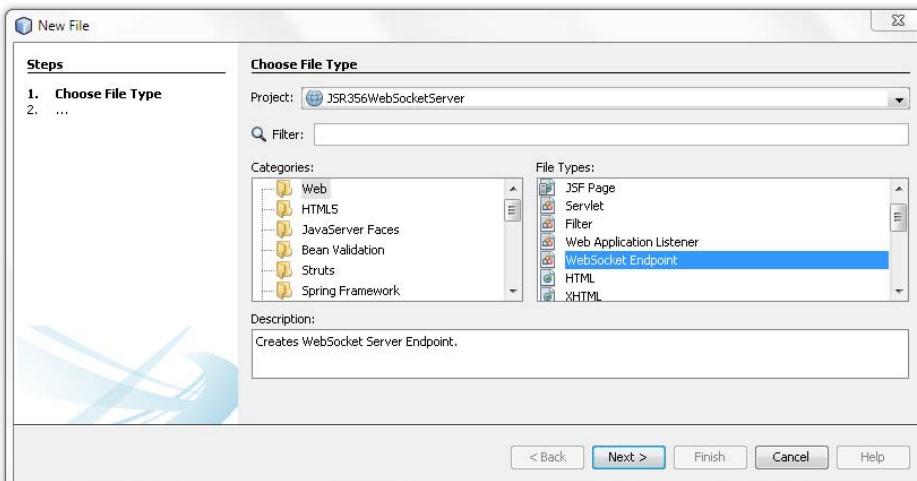


Bild 6.25 Erzeugung eines neuen WebSocket-Endpoints unter *Other...*

Um bei den bisherigen Beispielen zu bleiben, wollen wir uns jetzt anschauen, wie wir mit dem JSR 356 einen WebSocket-Server implementieren, der eine Art Echo-Dienst bereitstellt. Genauer gesagt soll der Server auf einen bereitgestellten Namen eine höfliche Anrede konstruieren und zurückliefern.

Die Verbindung und die Interaktionen zwischen Server und Client werden in der JSR 356-Spezifikation durch eine Session definiert (siehe [Listing 6.20, Zeile 13](#)).

Listing 6.20 WebSocket-Endpoint mit Annotationen

```

1 ...
2 import javax.websocket.CloseReason;
3 import javax.websocket.OnClose;
4 import javax.websocket.OnMessage;
5 import javax.websocket.OnOpen;
6 import javax.websocket.Session;
7 import javax.websocket.server.ServerEndpoint;
8
9 @ServerEndpoint("/polite")
10 public class GreetingsServer_Annotations {
11
12     @OnOpen
13     public void onOpen(Session session) {
14         System.out.println("New session: "+session.getId());
15     }
16
17     @OnMessage
18     public String onMessage(String name) {
19         return "Dear "+name;
20     }
21
22     @OnClose
23     public void onClose(Session session, CloseReason closeReason){
24         System.out.println("Session "+session.getId()+" beendet");
25         System.out.println(closeReason);
26         session = null;
27     }
28 }
```

Das Listing zeigt die Implementierung eines WebSockets unter Verwendung der dafür spezifizierten Annotationen. Der WebSocket-Server ist über den Endpoint /polite erreichbar. Die komplette URL setzt sich dann aus dem Hostnamen, dem Projektnamen und dem Endpoint zusammen. Heißt das Projekt z. B. Greetings, so ist dieser WebSocket durch die URL

ws://hostname/Greetings/polite

erreichbar. Senden Sie dem WebSocket-Server eine Textnachricht, die einen Namen enthält, sagen wir Mr. President, liefert Ihnen der Server eine höfliche Anrede als Textnachricht zurück: Dear Mr. President

Diese Implementierung steht mit den Spezifikationen des Final Release des JSR 356 im Einklang und kann in dieser Form auch mit den verfügbaren Implementierungen umgesetzt

werden. Den gleichen WebSocket können Sie auf eine zweite Weise implementieren, die nicht die Annotationen, sondern direkt die API verwendet.

Listing 6.21 WebSocket-Endpoint per API

```

1 ...
2 import java.io.IOException;
3 import javax.websocket.CloseReason;
4 import javax.websocket.Endpoint;
5 import javax.websocket.EndpointConfig;
6 import javax.websocket.MessageHandler;
7 import javax.websocket.Session;
8
9 public class GreetingsServer_API extends Endpoint {
10
11     @Override
12     public void onOpen(final Session session, EndpointConfig ec) {
13         System.out.println("New session " + session.getId());
14
15         session.addMessageHandler(
16             new MessageHandler.Whole<String>() {
17
18                 @Override
19                 public void onMessage(String name) {
20                     try {
21                         session.getBasicRemote().sendText("Dear " +
22                             name);
23                     } catch (IOException ex) {
24                     }
25                 }
26             });
27     }
28
29     @Override
30     public void onClose(Session session, CloseReason closeReason) {
31         System.out.println("Session " + session.getId() +
32             " beendet");
33         System.out.println(closeReason);
34     }
35 }
```

Das Listing 6.21 ist gemäß der Spezifikationen des Final Release und der dazu verfügbaren Java-Docs erstellt worden. Wenn Sie einen WebSocket-Endpoint über die API implementieren, müssen Sie zusätzlich eine Konfigurationsklasse erstellen, die den Endpoint des Servers festlegt (siehe Listing 6.22).

Listing 6.22 Konfigurationsklasse für den Endpoint

```

1 ...
2 import java.util.Collections;
3 import java.util.HashSet;
4 import java.util.Set;
5 import javax.websocket.Endpoint;
```

```

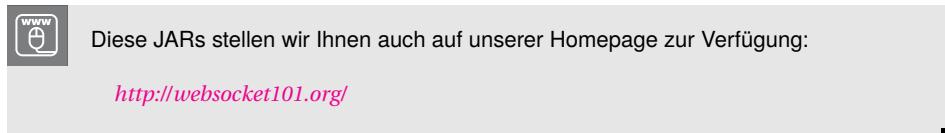
6 import javax.websocket.server.ServerApplicationConfig;
7 import javax.websocket.server.ServerEndpointConfig;
8
9 public class Config implements ServerApplicationConfig {
10
11     @Override
12     public Set<ServerEndpointConfig> getEndpointConfigs(
13         Set<Class<? extends Endpoint>> set) {
14         return new HashSet<ServerEndpointConfig>() {
15             {
16                 add(ServerEndpointConfig.Builder.create(
17                     GreetingsServer_API.class, "/polite").build());
18             }
19         };
20     }
21     @Override
22     public Set<Class<?>> getAnnotatedEndpointClasses(
23         Set<Class<?>> set) {
24         return Collections.emptySet();
25     }
26 }
```

6.5.2 JSR 356-basierender WebSocket-Client

Mit Ihren Kenntnissen zur Programmierung von WebSocket-Clients anhand der W3C-Spezifikation für browser-basierte Clients bzw. mit einem der bis hierhin behandelten WebSocket-Frameworks können Sie natürlich umgehend einen Client für den im vorangegangenen Kapitel implementierten WebSocket-Server erstellen. Möchten Sie den Client weder im Browser noch anhand eines anderen WebSocket-Frameworks realisieren, sondern durchgängig mit dem JSR 356, so können Sie das unter Verwendung der clientseitigen APIs erreichen. In diesem Beispiel werden wir ein einfaches Java-Projekt erzeugen, das einen Namen zum „Greetings-Server“ schickt. Um den WebSocket-Client des JSR 356 zu verwenden, müssen Sie noch folgende JAR-Dateien in Ihr Projekt integrieren:

```

grizzly-framework-2.3.3.jar
grizzly-http-2.3.3.jar
grizzly-http-server-2.3.3.jar
grizzly-rqm-2.3.3.jar
javax.websocket-api-1.0.jar
tyrus-client-1.2.1.jar
tyrus-container-grizzly-1.2.1.jar
tyrus-core-1.2.1.jar
tyrus-server-1.2.1.jar
tyrus-spi-1.2.1.jar
tyrus-websocket-core-1.2.1.jar
```



In NetBeans können Sie JARs hinzufügen, indem Sie einen Rechtsklick auf das jeweilige Projekt machen und dann *Properties* auswählen. Es öffnet sich ein neues Fenster. Unter *Categories → Libraries* können Sie durch den Button *Add JAR/Folder* JAR-Dateien hinzufügen (siehe Bild 6.26).

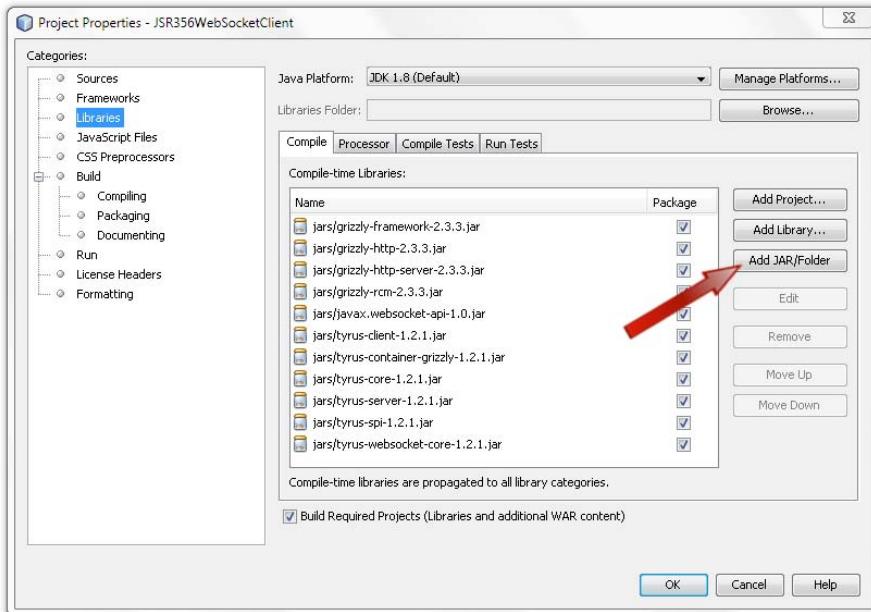


Bild 6.26 JARs hinzufügen in NetBeans

Wenn Sie Eclipse bevorzugen, klicken Sie bitte mit der rechten Maustaste auf das Projekt und wählen *Build Path → Configure Build Path* aus. Im neuen geöffneten Fenster können Sie mit dem Button *Add External JARs...* externe JAR-Dateien integrieren (siehe Bild 6.27).

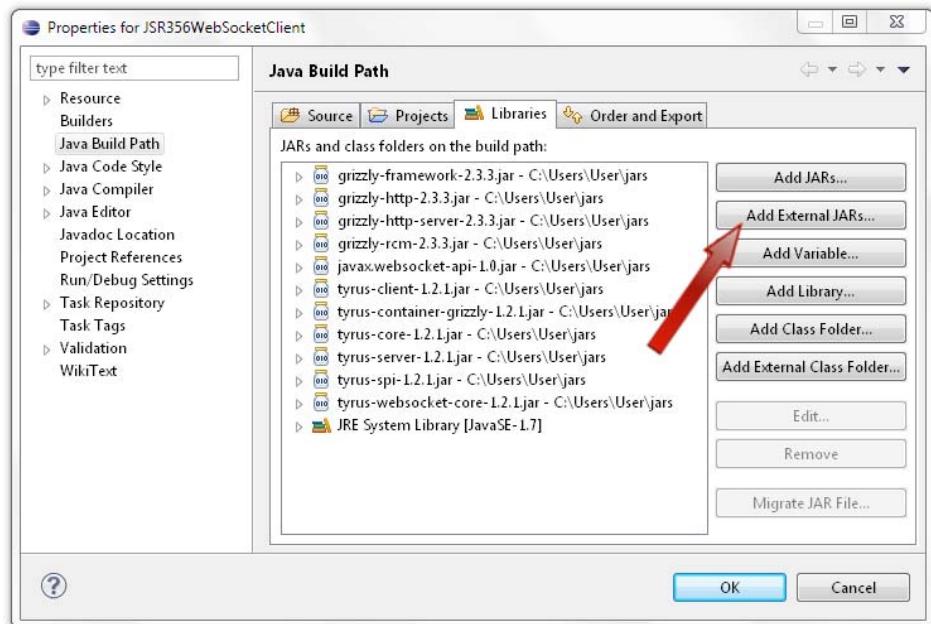


Bild 6.27 JARs hinzufügen in Eclipse

Es existiert noch zusätzlich die Möglichkeit, die JARs über ein Maven-Projekt hinzuzufügen. Erstellen Sie dazu zunächst ein Java-Maven-Projekt (siehe [Bild 6.28](#)).

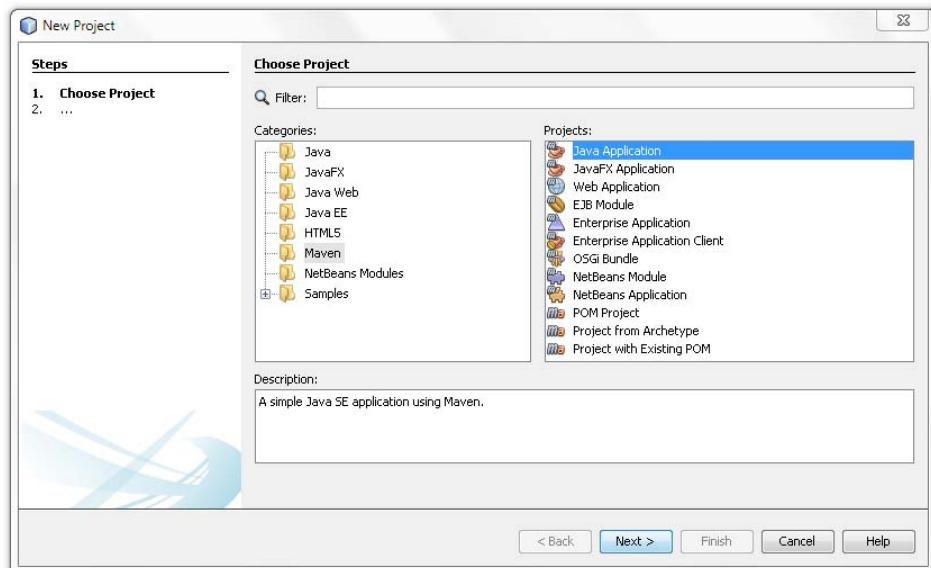


Bild 6.28 Erstellen eines neuen Maven-Projekts

Danach müssen Sie die Datei *pom.xml* mit den entsprechenden Abhängigkeiten anpassen. Die Datei befindet sich in NetBeans unter *Project Files* → *pom.xml* (siehe Bild 6.29).

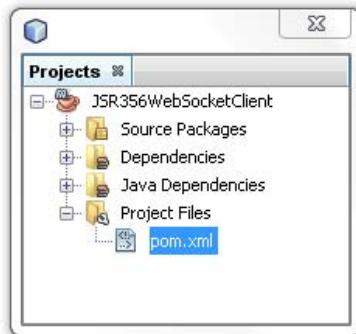


Bild 6.29 *pom.xml*-Datei in NetBeans

Falls Sie mit Eclipse arbeiten, finden Sie die Datei dort direkt im Projektordner (siehe Bild 6.30).

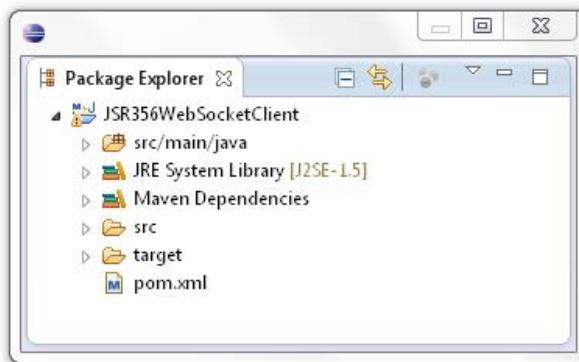


Bild 6.30 *pom.xml* in Eclipse

Ergänzen Sie folgende Abhängigkeiten im Tag `<dependencies>` (siehe Listing 6.23):

Listing 6.23 Maven-Dependencies für JSR-WebSocket-Client

```

1 <dependency>
2   <groupId>javax.websocket</groupId>
3   <artifactId>javax.websocket-api</artifactId>
4   <version>1.0</version>
5 </dependency>
6 <dependency>
7   <groupId>org.glassfish.tyrus</groupId>
8   <artifactId>tyrus-container-grizzly</artifactId>
9   <version>1.2.1</version>
10 </dependency>
11 <dependency>
```

```

12      <groupId>org.glassfish.tyrus</groupId>
13      <artifactId>tyrus-client</artifactId>
14      <version>1.2.1</version>
15  </dependency>
```

Die Datei sollte zum Schluss ungefähr Bild 6.31 entsprechen.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5                               http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <groupId>com.mycompany</groupId>
8   <artifactId>JSR356WebSocketClient</artifactId>
9   <version>1.0-SNAPSHOT</version>
10  <packaging>jar</packaging>
11  <properties>
12      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13      <maven.compiler.source>1.7</maven.compiler.source>
14      <maven.compiler.target>1.7</maven.compiler.target>
15  </properties>
16  <dependencies>
17      <dependency>
18          <groupId>javax.websocket</groupId>
19          <artifactId>javax.websocket-api</artifactId>
20          <version>1.0</version>
21      </dependency>
22      <dependency>
23          <groupId>org.glassfish.tyrus</groupId>
24          <artifactId>tyrus-container-grizzly</artifactId>
25          <version>1.2.1</version>
26      </dependency>
27      <dependency>
28          <groupId>org.glassfish.tyrus</groupId>
29          <artifactId>tyrus-client</artifactId>
30          <version>1.2.1</version>
31      </dependency>
32  </dependencies>
33 </project>
```

Bild 6.31 Abhängigkeiten der Datei *pom.xml*

Der Code aus [Listing 6.24](#) implementiert nun einen WebSocket-Client, der unsere Greetings-Server aus [Abschnitt 6.5.1](#) anspricht. Passen Sie die Kommentare in [Zeile 48](#) bis [50](#) an, je nachdem welchen Server Sie testen möchten.

Listing 6.24 JSR-356 WebSocketClient

```

1 ...
2 import java.io.IOException;
3 import java.net.URI;
4 import java.net.URISyntaxException;
5 import java.io.IOException;
6 import java.net.URI;
7 import java.net.URISyntaxException;
8 import javax.websocket.ClientEndpoint;
9 import javax.websocket.CloseReason;
10 import javax.websocket.ContainerProvider;
11 import javax.websocket.DeploymentException;
12 import javax.websocket.MessageHandler;
13 import javax.websocket.OnClose;
```

```
14 import javax.websocket.OnOpen;
15 import javax.websocket.Session;
16 import javax.websocket.WebSocketContainer;
17
18 @ClientEndpoint
19 public class GreetingsClient {
20     private Session userSession = null;
21     public static String message;
22     public GreetingsClient(URI endpointURI) throws
23         DeploymentException, IOException {
23         WebSocketContainer container =
24             ContainerProvider.getWebSocketContainer();
25         userSession = container.connectToServer(this, endpointURI);
26     }
27
28     @OnOpen
29     public void onOpen(Session session) {
30         System.out.println("SessionId:" + session.getId());
31     }
32
33     public Session getSession() {
34         return this.userSession;
35     }
36
37     @OnClose
38     public void onClose(Session session, CloseReason
39         closeReason) {
40         this.userSession = null;
41         System.out.println(closeReason);
42     }
43
44     public void sendMessage(String message) throws IOException {
45         this.userSession.getBasicRemote().sendText(message);
46     }
47
48     public static void main(String[] args) throws URISyntaxException,
49         IOException, DeploymentException {
50
51         String webSocketServerURI = "ws://echo.websocket.org";
52         //String webSocketServerURI = "ws://localhost:8080/
53             GreetingsServerAnnotations/polite";
54         //String webSocketServerURI = "ws://localhost:8080/
55             GreetingsServerAPI/polite";
56
57         GreetingsClient client = new GreetingsClient(
58             new URI(webSocketServerURI));
59         client.sendMessage("Mr. President");
60         client.getSession().addMessageHandler(
61             new MessageHandler.Whole<String>() {
62
63                 @Override
64                 public void onMessage(String message) {
65                     System.out.println(message);
66                 }
67             });
68     }
69 }
```

```
59         }
60     });
61     while(true);
62 }
63 }
```

Da wir hier einen sequenziellen Programmcode vorfinden, der möglicherweise terminiert, bevor unser Greetings-Server die Begrüßungsnachricht zurückschickt, fügen wir eine while-Schleife ([Listing 6.24, Zeile 61](#)) zum Testen hinzu. Die while-Schleife sorgt dafür, dass unser Programm nicht beendet wird, bevor unser Greetings-Server die Begrüßungsnachricht zurückschickt. Führen Sie nun das Projekt entweder in einem Java-Projekt oder in einem Maven-Projekt aus. Sie haben somit Ihren ersten WebSocket-Client in JSR-356 gestartet.

7

WebSockets in der Praxis

Hat Sie das WebSocket-Fieber gepackt und wollen Sie über die einführenden Aspekte hin-aus eigene Projekte verwirklichen, so ergibt sich eine ganze Reihe von Fragen, die es bei der Umsetzung in der Praxis zu berücksichtigen gilt. Relevante wollen wir im Folgenden mit Ihnen beleuchten.



Fragen, die dieses Kapitel beantwortet:

- Worauf sollten Sie Ihre Aufmerksamkeit bei Entwicklungen im Bezug auf Performance und Skalierbarkeit richten?
- Worauf sollten Sie bei sicherer Kommunikation und Authentifizierung achten?
- Was gibt es derzeit für mögliche Gefährdungen und was kann man dagegen tun?
- Was gibt es aktuell für Haken und Ösen, die Sie kennen sollten, um holprige Projektstarts und langwierige Fehlersuchen vermeiden zu können?

■ 7.1 Performance und Skalierbarkeit

In diesem Kapitel wollen wir Ihnen zeigen, wie Sie die Performance von WebSockets testen können. Wir empfehlen Ihnen einen Test, wenn Sie WebSockets mit einem bestimmten Framework bzw. mit bestimmten Servern nutzen möchten. Durch eine Analyse der Testergebnisse kann anschließend festgestellt werden, ob das jeweilige Framework überhaupt für Ihre geplante Anwendung geeignet ist.

Das Testverfahren besteht aus drei Komponenten. Die erste ist ein Echo-Server, der ankommende Nachrichten annimmt und sie wieder an den Absender zurück-schickt. Die zweite Komponente besteht aus einer variierenden Anzahl von Testclients, die Nachrichten an den Echo-Server schicken. Diese können dazu genutzt werden, die Auslastung des Servers zu prüfen. Zudem werden die eingesetzten Testclients auf mehrere Rechner verteilt.

Um einen aussagekräftigen Wert über die Auslastung bzw. die Performance ermitteln zu können, kommt eine dritte Komponente ins Spiel. Ein einzelner WebSocket-Testclient schickt ebenfalls Nachrichten an den Server, wobei zusätzlich die Round-Trip-Zeit zwischen dem Server und ihm gemessen wird (siehe Bild 7.1).

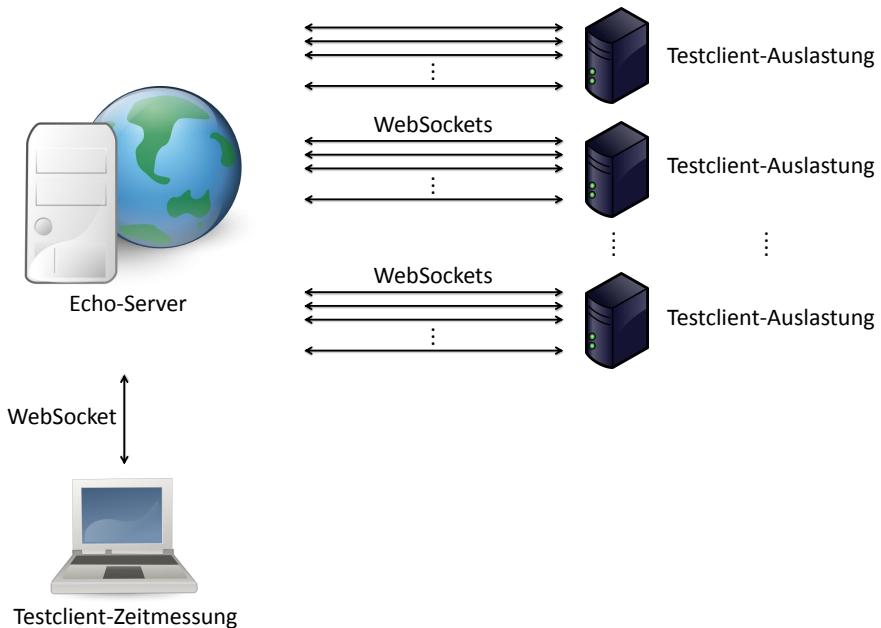


Bild 7.1 Testszenario zur Messung der Auslastung bzw. Performance eines Servers

Das Ziel des Ganzen ist herauszufinden, wie lange ein Server braucht, um eine Nachricht anzunehmen und zum jeweiligen Client zurückzuschicken, wenn er gleichzeitig eine bestimmte Anzahl von anderen Anfragen verwalten muss. Wir werden Ihnen im Folgenden zeigen, wie Sie solch einen Test einrichten und implementieren können.

Für die Clientseite nutzen wir Node.js und für die Serverseite Vert.x. Natürlich können Sie Ihren Client oder Ihren Server auch in einer anderen Programmiersprache oder in einem anderen Framework realisieren. Wir möchten Ihnen in diesem Kapitel nur die grundlegende Architektur der Testumgebung näherbringen.

Um eine wirkungsvolle Testumgebung einzurichten, brauchen wir möglichst viele Rechner, um damit möglichst viele WebSockets parallel öffnen zu können. Webservices, die es Ihnen ermöglichen, virtuelle Rechner dynamisch zu erzeugen und zu verwalten, eignen sich sehr gut für diesen Zweck. Deshalb werden wir zur Realisierung des Testszenarios die Amazon Web Services (AWS) nutzen, die Ihnen, wie auch viele andere Dienstleister, diese nützliche Funktion zur Verfügung stellen.

7.1.1 Test-Echo-Server

Widmen wir uns nun unserem Echo-Server, den wir in Vert.x unter JavaScript entwickeln.



Eine entsprechende Version des Servers in Java und Node.js finden Sie auf unserer Webseite.

Wir erzeugen einen Webserver ([Listing 7.1, Zeile 3](#)) und zwei weitere Variablen ([Zeile 4](#) und [5](#)). Die Variable `sockets` initialisieren wir als Array, in das wir später bestimmte WebSockets speichern. Warum wir dies machen, wird im weiteren Verlauf erklärt. Zusätzlich erzeugen wir die Variable `clients` und initialisieren diese mit dem Wert 0. Darin speichern wir die Anzahl der verbundenen Clients.

Als Nächstes öffnen wir den WebSocket-Handler unseres Webservers ([Zeile 7](#)). Die Callback-Funktion, die wir dem WebSocket-Handler übergeben, enthält den verbundenen WebSocket als Parameter. Hier wollen wir Folgendes realisieren: Wir bestimmen einen Endpunkt `/watch` ([Zeile 8](#)) und einen Endpunkt `/echo` ([Zeile 21](#)). Nur den Clients, die diese beiden Endpunkte ansprechen, erlauben wir eine Verbindung und erhöhen den Wert der Variablen `clients` (siehe [Zeile 9](#) und [22](#)). Allen Clients, die sich zu einem anderen Endpunkt verbinden wollen, verweigern wir den Zugang ([Zeile 35](#)).

Listing 7.1 Echo-Server in Vert.x (*vertx_EchoServer.js*)

```

1 var vertx = require('vertx');
2 var console = require('vertx/console');
3 var webServer = vertx.createHttpServer();
4 var sockets = [];
5 var clients = 0;
6
7 webServer.websocketHandler(function(ws) {
8     if(ws.path() === "/watch") {
9         clients++;
10        sockets.push(ws);
11        for (var i = 0; i < sockets.length; i++)
12            sockets[i].writeTextFrame(clients);
13        ws.closedHandler(function(buffer) {
14            clients--;
15            if(sockets.indexOf(ws) >= 0)
16                sockets.splice(sockets.indexOf(ws),1);
17            for (var i = 0; i < sockets.length; i++)
18                sockets[i].writeTextFrame(clients);
19        });
20    }
21    else if(ws.path() === "/echo") {
22        clients++;
23        for(var i = 0; i < sockets.length; i++)
24            sockets[i].writeTextFrame(clients);
25        ws.dataHandler(function(buffer) {
26            ws.writeTextFrame(buffer.toString());
27        });
28        ws.closedHandler(function(buffer) {
29            clients--;
30            for (var i = 0; i < sockets.length; i++)
31                sockets[i].writeTextFrame(clients);
32        });
33    }
34    else{
35        ws.reject();
36    }
}

```

```

37 });
38
39 webServer.requestHandler(function(req) {
40     req.response.sendFile("index.html");
41 });
42
43 webServer.listen(3000);
44 console.log("Der EchoServer läuft auf dem Port 3000");

```

Im Endpunkt /watch soll keine Echo-Nachricht zurückgeschickt werden. Stattdessen wird der Client über die Anzahl der aktuell verbundenen Clients informiert. Damit jeder Client fortlaufend den aktuellsten Stand erfährt, speichern wir alle Sockets, die sich mit dem Server verbinden, in das WebSocket-Array aus [Zeile 4](#) ab. Wir definieren zusätzlich einen Closed-Handler, der, sobald eine Verbindung geschlossen wird, die Anzahl der Clients verringert und die im selben Endpunkt weiterhin verbundenen WebSockets über diese Änderung benachrichtigt. Diesen Endpunkt können wir dann auch später nutzen, um die Anzahl der Clients zu beobachten.

Verbindet sich ein Client zum Endpunkt /echo, wird die Anzahl der Clients, wie schon erwähnt, in der Variablen clients aus [Zeile 5](#) um eins erhöht. Anschließend werden alle Clients des Endpunkts /watch über diese Änderung informiert. Dieselben Clients sollen auch benachrichtigt werden, wenn sich eine Verbindung vom Endpunkt /echo schließt. Dies realisieren wir wieder mit einem Closed-Handler (siehe [Zeile 28](#)).

Spricht ein Client unseren Server über HTTP an, so wird diesem mithilfe des Request-Handlers mit einer statischen HTML-Seite geantwortet (siehe [Zeile 40](#)). Das [Listing 7.2](#) zeigt den Inhalt der HTML-Datei.

Listing 7.2 HTML-Seite zum Anzeigen der verbundenen WebSockets (*index.html*)

```

1 <!doctype html>
2 <html>
3     <head>
4         <title>WebSocket Echo-Test (Vert.x)</title>
5     </head>
6     <body>
7         <h1>Vert.x</h1>
8         <p> WebSockets: <span id="count"></span></p>
9     </body>
10    <script type="text/javascript">
11        var count = document.getElementById("count");
12        var ws = new WebSocket ("ws://" + window.location.host + "/watch"
13                               );
14        ws.onmessage = function(evt){
15            count.innerHTML = evt.data;
16        }
17    </script>
18 </html>

```

Diese Seite verwenden wir später, um die Anzahl der verbundenen Clients in Echtzeit zu beobachten. Dazu wird ein WebSocket geöffnet, der sich mit dem Endpunkt /watch verbindet.

7.1.2 Testclient für die Auslastung

Der Testclient für die Auslastung hat nur die Aufgabe, kontinuierlich Nachrichten an den Server zu schicken. Für die Realisierung, werden wir Node.js bzw. das Modul WebSocket-Node verwenden. WebSocket-Node können Sie folgendermaßen über die Konsole installieren:

```
npm install websocket
```

Betrachten wir nun die Implementierung des Testclients. Als Erstes wird in dem Quellcode der WebSocket-Client von WebSocket-Node importiert (siehe [Listing 7.3, Zeile 1](#)). Der Testclient soll eine bestimmte Anzahl von WebSockets öffnen, die in der Variablen count festgelegt wird. Die Variable url speichert die Adresse des Echo-Servers.

Listing 7.3 Testclient für den Auslastungstest (*testclient.js*)

```
1 var WebSocketClient = require('websocket').client;
2 var count = 500;
3 var url = "ws://127.0.0.1:3000/echo";
4
5 for(var i = 0; i < count; i++) {
6     var client = new WebSocketClient();
7     client.connect(url);
8     client.on("connect",function(connection) {
9         connection.sendUTF("Hallo Welt");
10        connection.on("message",function(message) {
11            console.log(message);
12            connection.sendUTF("Hallo Welt");
13        });
14        connection.on("close",function(){
15            delete connection;
16        });
17    });
18    client.on("connectFailed",function(err) {
19        client.connect(url);
20        console.log(new Date() + err);
21    });
22 }
```

Im nächsten Schritt wird eine for-Schleife durchlaufen. In dieser Schleife erstellen wir einen WebSocket-Client und speichern diesen in der lokalen Variablen client ab. Wie Sie sehen, ist die API von WebSocket-Node anders als die von der WebSocket-API des W3C. Wenn Sie hier einen WebSocket erstellen, enthält der Konstruktor keine Parameter. Erst danach ([Zeile 7](#)) können Sie mit der Methode connect() Argumente zum Verbindungsauftbau übergeben. Im ersten Parameter der connect()-Methode wird die URL übergeben. Im zweiten Parameter können optional Subprotokolle übergeben werden.

Da unser Echo-Server keine Subprotokolle spricht und wir für unseren Test auch keine benötigen, übergeben wird der connect()-Methode nur die URL, die sich in der Variablen url befindet. Daraufhin registrieren wir für den Event connect eine Callback-Funktion. Diese Funktion enthält den Parameter connection, der die erfolgreiche Verbindung repräsentiert. Wenn sich der Client erfolgreich mit dem Server verbunden hat, wird diese

Funktion aufgerufen. Darin schicken wir eine Nachricht an den Echo-Server. Danach definieren wir eine neue Callback-Funktion für die Verbindung, die nun in der Variablen connection gespeichert ist. Diese Callback-Funktion soll ausgeführt werden, wenn eine Nachricht empfangen wird. Wir loggen diese Nachricht in der Konsole, um zu sehen, ob sie erfolgreich versendet und empfangen wurde. Im Anschluss schicken wir direkt eine neue Nachricht los. In Zeile 18 registrieren wir eine weitere Callback-Funktion, die den Client wieder zum Server verbindet, falls der Verbindungsauftakt nicht erfolgreich war. Für spätere Fehleranalysen geben wir den Fehler für den nicht erfolgreichen Verbindungsauftakt auf der Konsole mit dem aktuellen Datum bzw. der aktuellen Zeit aus.

Solange das Programm läuft, versendet der Client nun permanent Nachrichten. Dies passt laut unserem Programmcode 500 mal pro Instanz. Dadurch können 500 Clients simuliert werden, die ständig Nachrichten an der Server senden und empfangen. Starten wir mehrere von diesen Instanzen, können wir testen, wie es die Performance beeinflusst, wenn der Server mehrere Tausend Clients verwaltet muss.

Beachten Sie, dass die Rechenleistung Ihres Computers begrenzt ist. D. h., Sie können nicht gleichzeitig mehrere Instanzen bzw. Tausende WebSockets starten. Deswegen empfehlen wir Ihnen, pro Rechner nur 500 WebSocket-Verbindungen aufzubauen. Um noch mehr WebSocket-Clients zu öffnen, können Sie mehrere Rechner benutzen, die jeweils eine Instanz mit dem Code aus Listing 7.3 starten. Dieses Szenario ist auch näher an der Realität, da ein Server im Normalfall verschiedene Clients von verschiedenen Rechnern verwalten muss.

7.1.3 Testclient für die Zeitmessung

In diesem Kapitel geht es darum, einen einfachen Client zu implementieren, der im Grunde genommen das Gleiche tut wie der Testclient für den Auslastungstest aus Listing 7.3. Der Unterschied ist, dass wir nur einen einzigen WebSocket öffnen. Außerdem versenden wir statt einer Textnachricht einen Zeitstempel (siehe Listing 7.4, Zeile 7).

Listing 7.4 Testclient für die Zeitmessung (*roundtrip_measure.js*)

```
1 var WebSocketClient = require('websocket').client;
2 var url = "ws://127.0.0.1:3000/echo"
3 var client = new WebSocketClient();
4
5 client.connect(url);
6 client.on("connect",function(connection) {
7     connection.sendUTF(new Date().getTime());
8     connection.on("message",function(message){
9         console.log(new Date().getTime()-message.utf8Data);
10        connection.sendUTF(new Date().getTime());
11    });
12 });
13 client.on("connectFailed",function(err){
14     client.connect(url);
15 });
```

Wenn eine Nachricht empfangen wird, erstellen wir einen neuen Zeitstempel und errechnen aus der Zeitangabe in der Nachricht die Zeitdifferenz. Diese loggen wir in der Konsole und können so die Round-Trip-Zeit feststellen. Direkt im Anschluss verschicken wir wieder eine Nachricht mit einem Zeitstempel. Was wir somit erzeugt haben, ist eine unendliche Schleife, die kontinuierlich Round-Trip-Zeiten berechnet und in der Konsole ausgibt.

7.1.4 Einrichtung der Testumgebung

Nun wollen wir Ihnen zeigen, wie Sie eine Testumgebung für einen Lasttest einrichten können. Wir werden hierzu die Amazon Web Services (AWS) verwenden. Einsetzen werden wir den Service „Amazon Elastic Compute Cloud“ („Amazon EC2“ oder einfach nur kurz „EC2“), mit dem wir virtuelle Server bzw. Rechner erstellen können. Für die einzelnen virtuellen Rechner lassen sich die Rechenleistung und das Betriebssystem einstellen. Sowohl für die Testclients als auch für den Server wählen wir folgende Eigenschaften:

- Betriebssystem: Ubuntu 12.04 64 Bit
- Arbeitsspeicher: 613 MB
- Kerne: 1
- CPUs: bis zu 2 CPUs mit 1,0–1,2 GHz
- Standort: Irland

Diese Rechenleistung entspricht einer Amazon Mikro-Instanz [Ama14]. Der EC2-Dienst bietet Ihnen auch die Möglichkeit, ein virtuelles Image von einem Rechner zu erstellen. Sie können demnach einen Rechner erstellen und dort Programme Ihrer Wahl installieren und konfigurieren. Wenn alles Ihren Wünschen entspricht, können Sie damit anschließend eine exakte Kopie (ein Image) von diesem Rechner erstellen, wodurch Ihnen dann eine oder mehrere identische virtuelle Computer zur Verfügung stehen. Das können wir später nutzen, um ohne großen Aufwand eine beliebige Anzahl von Testclient-Rechnern für unsere Testumgebung zu erstellen. Fangen wir aber zunächst nur mit einem an.

Um nun den Test durchzuführen, brauchen wir also einen Server, einen Rechner, der die Testclients für die Auslastung ausführt, und einen Rechner, der die Round-Trip-Zeiten misst. Suchen Sie sich als Erstes den Server aus, den Sie testen möchten, und richten Sie darauf den Echo-Server aus [Listing 7.1](#) ein. Der virtuell erstellte Rechner in der Amazon Cloud kommt als Testclient zur Auslastung zum Einsatz. Kopieren Sie dazu den Programmcode aus [Listing 7.3](#). Vergessen Sie nicht, die benötigten Programme wie z. B. Node.js zu installieren. Starten Sie nun zuerst den Echo-Server und danach die Testclients auf der virtuellen Maschine. Wenn Sie anschließend mit einem Browser auf die URL des Echo-Servers zugreifen, sollte Ihnen die Anzahl der verbundenen Clients angezeigt werden. Diese sollte identisch mit der Anzahl sein, die Sie in dem Programmcode für die Testclients eingestellt haben (siehe [Bild 7.2](#)). Natürlich kommt noch genau ein Client hinzu, nämlich Ihr Browser, der ebenfalls eine WebSocket-Verbindung zum Server aufgebaut hat.

Wenn bei Ihnen alles einwandfrei funktioniert, empfehlen wir Ihnen, als Nächstes Startskripte für den Testclient-Rechner einzurichten. Damit können Sie festlegen, dass der Programmcode direkt ausgeführt wird, sobald Sie den Rechner gestartet haben. Nach der Einrichtung können Sie dann die bereits erwähnten Images nutzen, um in wenigen Schritten eine Vielzahl von Testclients zu erstellen. Danach müssen Sie, dank der Startskripte, nur



Bild 7.2 Anzahl der verbundenen Clients in Echtzeit

noch die virtuellen Maschinen starten, ohne dass Sie sich auf den einzelnen Rechnern manuell anmelden müssen.

Wir verwenden für das Erstellen und Verwalten von Startskripten die Software Supervisor [Age14]. Diese kann laut der Webseite auf allen Unix-Systemen betrieben werden. Für Windows existiert leider keine Version. Eine Anleitung zur Installation finden Sie auf folgender Internetseite.¹ In einigen Linux-Distributionen wie z. B. unter Ubuntu lässt sich Supervisor über den Paketmanager installieren:

```
sudo apt-get install supervisor
```

Supervisor setzt Python in der Version 2.4 oder höher voraus. Python 3 wird bisher allerdings nicht unterstützt. Denken Sie daran, dass Sie für das Verwalten von Startskripten bzw. für Supervisor Root-Rechte benötigen.

Wenn Sie Supervisor über den Paketmanager installiert haben, können Sie im Ordner */etc/supervisor/conf.d/* Konfigurationsdateien ablegen. Erstellen Sie hier eine Datei z. B. *myProgram.conf* mit dem Inhalt aus [Listing 7.5](#).

Listing 7.5 Konfigurationsdatei für Supervisor (*myProgram.conf*)

```
[program:mytestclient]
command = node testclient.js
directory = /home/user/node-testclient/
stdout_logfile = /home/user/node-testclient/testclient.log
```

Wir definieren hier, dass unser Skript *mytestclient* heißt. Dieses soll den Befehl

```
node testclient.js
```

im Ordnerpfad */home/user/node-testclient/* ausführen. Alle Ausgaben des Programms wollen wir in der Datei *testclient.log* im Ordnerpfad */home/user/node-testclient* speichern. Im Internet finden Sie weitere Informationen zu den Einstellungsmöglichkeiten für Supervisor.²

Nachdem Sie die Konfigurationsdatei gespeichert haben, können Sie mit dem Befehl

¹ <http://supervisord.org/installing.html>

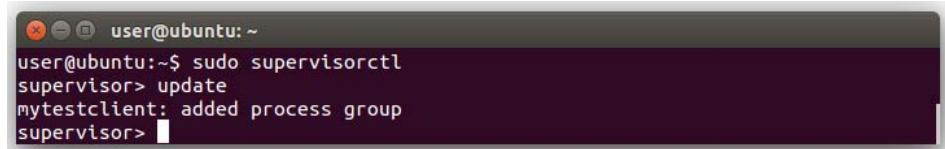
² <http://supervisord.org/configuration.html#program-x-section-settings/>

```
supervisorctl
```

die Verwaltungskonsole von Supervisor öffnen. Geben Sie den Befehl

```
update
```

ein und bestätigen Sie mit *Enter*. Nun sollte Folgendes auf der Konsole erscheinen (siehe Bild 7.3):



```
user@ubuntu: ~
user@ubuntu:~$ sudo supervisorctl
supervisor> update
mytestclient: added process group
supervisor>
```

Bild 7.3 Aktualisieren der Startprogramme von Supervisor

Das Testclient-Programm wurde nun im Hintergrund gestartet und ist zudem als Startprogramm für zukünftige Neustarts des Computers eingerichtet. Als Nächstes können Sie ein Image von diesem Rechner erstellen, um damit weitere Testclients einzurichten.



Erhöhen der Datei-Handler in Unix-Systemen

„Alles ist eine Datei“ ist eine wesentliche Eigenschaft von Unix-Systemen. Auch jeder TCP-Socket, der sich zum Server verbindet, ist eine Datei. Da die Anzahl der sogenannten Datei-Handler (*engl. File descriptors*) geringer sein kann als die von Ihnen gewünschte Anzahl von WebSocket-Verbindungen, erhöhen Sie bitte die Anzahl der Handler, bevor Sie Ihren Test starten. Dies können Sie mit dem Befehl `ulimit -n` erreichen. Um die Anzahl der Datei-Handler z. B. auf 100000 zu erhöhen, lautet der Befehl:

```
ulimit -n 100000
```

Auch für diesen Befehl benötigen Sie Root-Rechte. Um die Einstellung dauerhaft zu speichern, können Sie den Befehl in der Datei `.bashrc` (oder `bash_profile`), die sich versteckt im Homeverzeichnis des Root-Benutzers befindet, hinzufügen. Falls dies nicht funktioniert, können Sie es mit dem folgenden Befehl versuchen:

```
sysctl -w fs.file-max = 100000
```

Diese Konfiguration können Sie dauerhaft einstellen, indem Sie z. B. den Befehl

```
fs.file-max = 100000
```

in die Datei `/etc/sysctl.conf` einfügen. Nachdem Sie die Datei gespeichert und geschlossen haben, führen Sie bitte

```
sysctl -p
```

aus, damit die Änderung wirksam wird. Überprüfen Sie dies, indem Sie

```
cat /proc/sys/fs/file-max
```

oder

```
sysctl fs.file-max
```

in der Konsole ausführen.

Es ist ebenfalls möglich, die Anzahl der Dateideskriptoren in der Datei `/etc/security/limits.conf` für einen bestimmten Benutzer einzustellen. Hier ein Beispiel:

```
user soft nofile 50000
user hard nofile 100000
```

Starten Sie nun zuerst den fertig konfigurierten Server und dann nacheinander Ihre vorbereiteten Computer, um den Performancetest zu beginnen. Als Letztes starten Sie den Testclient für die Zeitmessungen (siehe [Listing 7.4](#)). Dieser kann ebenfalls in einer virtuellen Instanz laufen oder aber auch auf irgendeinem anderen Rechner, je nachdem aus welchem Netzwerk Sie die Round-Trip-Zeiten messen wollen.

Um die Round-Trip-Zeiten, die in der Konsole ausgegeben werden, in einer Datei zu speichern, können Sie auch hier wieder mit Supervisor arbeiten. Wie wir gesehen haben, ist es damit sehr leicht möglich, alle Ausgaben des Programms in eine Datei schreiben zu lassen.

Eine andere Möglichkeit für Node.js-Programme liefert das Modul Forever. Dieses ermöglicht Ihnen, Node.js-Programme im Hintergrund auszuführen. Zudem werden Ausgaben in der Konsole automatisch in Log-Dateien gespeichert. Forever installieren Sie einfach global über npm. Achtung, auch hierzu benötigen Sie Root-Rechte.

```
npm install -g forever
```

Mit dem Befehl `forever start` können Sie Node.js-Programme nach der Installation im Hintergrund starten:

```
forever start roundtrip_measure.js
```

Der Befehl `forever list` listet Ihnen alle Programme und deren Log-Files auf, die mit Forever gestartet wurden (siehe [Bild 7.4](#)).

```
user@ubuntu:~$ forever list
info:  Forever processes running
data:    uid  command   script          forever pid  logfile           uptime
data: [0] UJT4 /usr/bin/nodejs roundtrip-measure.js 30348  30350 /home/user/.forever/UJT4.log 0:0:1:19.699
user@ubuntu:~$
```

Bild 7.4 Auflistung der unter Forever gestarteten Programme

Sie können nun mit einem Browser auf die URL Ihres Echo-Servers zugreifen und die Anzahl der verbundenen WebSockets beobachten. Alternativ zum Browser lässt sich die Anzahl der TCP-Verbindungen für ein Programm oder einen Port mit der Eingabe des folgenden Befehls messen:

```
netstat -an | grep :3000 | grep ESTABLISHED | wc -l
```

Dieser Befehl besteht aus drei Programmen, die miteinander verbunden (*engl. pipe*) werden. Das Programm `netstat` ist ein Diagnose-Werkzeug für Netzwerkschnittstellen. Mit `grep` können Sie Dateien oder Ausgaben nach bestimmten Mustern oder Zeichenketten durchsuchen. Word count (`wc`) ermöglicht Ihnen das Zählen von Wörtern, Zeilen und Zeichen in einer Ausgabe oder einer Datei. Mit dem Befehl `|` können Sie Programme miteinander verbinden bzw. pipen.

Mit dem Befehl `netstat -an` listen Sie alle aktuellen TCP-Verbindungen auf. Die Ausgabe wird dann mit dem Befehl `grep :3000` verbunden, der die Ausgabe nach allen TCP-Verbindungen durchsucht und auflistet, die auf Port 3000 laufen. Diese Liste wird an das Programm `grep` weitergeleitet, welches nach erfolgreich aufgebauten (*engl. established*) Verbindungen sucht. Das Ergebnis wird letztlich noch mal mit dem Befehl `wc -l` verbunden, der alle Zeilen zählt.

Die Zeitmessungen können Sie jetzt z. B. für 1000, 2000, 3000 usw. WebSockets Verbindungen durchführen. Damit Sie am Ende genug Messwerte haben, empfehlen wir Ihnen, ca. fünf bis zehn Minuten pro Schritt zu messen. Erhöhen Sie dann die Anzahl der simulierten Clients z. B. im nächsten Schritt um 1000.

7.1.5 Ergebnisse

Wir haben ebenfalls einen Performancetest durchgeführt und möchten Ihnen die Ergebnisse zum Vergleich vorstellen. Wie zu erwarten war, steigt die Round-Trip-Zeit mit der Anzahl der verbundenen Clients an (siehe [Tabelle 7.1](#) und [Bild 7.5](#)).

Tabelle 7.1 Ergebnisse der Zeitmessungen

Anzahl der Clients	1000	2000	3000	4000
Durchschnittliche Round-Trip-Zeit [ms]	83,14	390,67	1509,18	2793,08
Standardabweichung [ms]	47,59	504,06	1098,72	2026,97

Auch die Standardabweichung steigt, wenn der Server eine Vielzahl von Clients verwalten muss. Bei einer Anzahl von 2000 variieren die Zeiten schon so stark, dass sie nicht mehr vorhersehbar sind. Eine Mikro-Instanz der AWS wäre somit unter den getesteten Bedingungen ab 2000 verbundenen Clients nicht mehr für eine Produktivumgebung einsetzbar.

Wir konnten bei der Durchführung der Messungen ebenfalls beobachten, dass die CPU-Leistung bei ca. 1500 Clients auf über 90% anstieg. Demzufolge war der eingesetzte Computer ab diesem Zeitpunkt fast ausgelastet. Dazu sollte gesagt werden, dass die Amazon Mikro-Instanz nicht für eine CPU-Auslastung über einen langen Zeitraum geeignet ist.³

³ <http://aws.amazon.com/de/ec2/instance-types/#instance-features/>

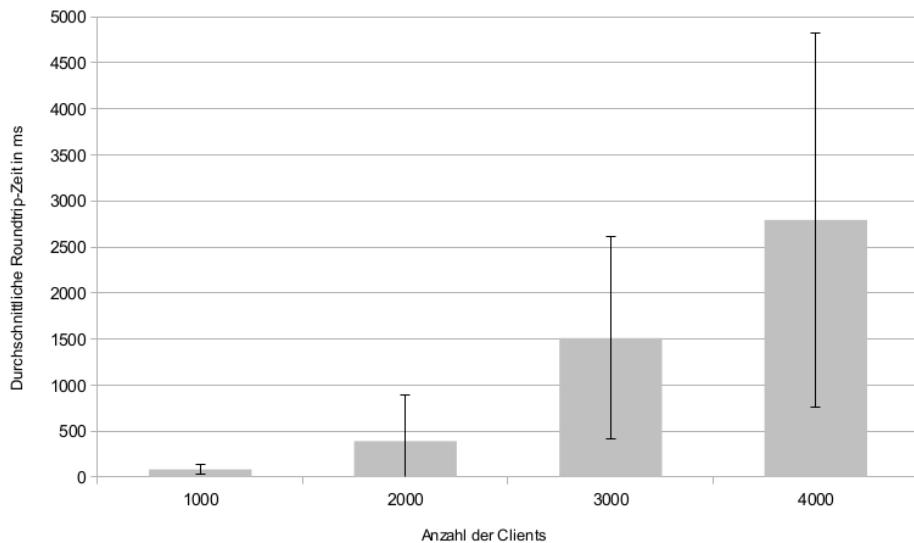


Bild 7.5 Ergebnisse der Zeitmessung

■ 7.2 Sicherheit

Neue Technologien bringen neben neuen Möglichkeiten häufig auch neue Gefährdungspotenziale mit sich, insbesondere dann, wenn es sich bei der Technologie – wie es für Web-Sockets der Fall ist – um ein Zahnrädchen verteilter Anwendungssysteme handelt, die über offene Netze kommunizieren. Daher darf trotz der Euphorie um die Palette an neuen Funktionen nie auf die Berücksichtigung von Sicherheitsaspekten verzichtet werden.

7.2.1 Same Origin Policy

Die *Same Origin Policy* (SOP) hat 1996 Einzug in den Webbrower gefunden, als seiner Zeit Netscape seinen gleichnamigen User Agent mit JavaScript ausstattete. Durch die Einführung aktiver Inhalte zunächst in Form von JavaScript-Programmen im Brower wurde die Notwendigkeit gesehen, diese Inhalte in ihrer Möglichkeit zu beschränken, auf Objekte von anderen Quellen als dem eigenen Server zuzugreifen. Bei der SOP handelt es sich also um einen Sicherheitsmechanismus auf der Clientseite (genauer gesagt im Brower), der sicherstellt, dass aktive Inhalte nur mit dem Server interagieren dürfen, von dem sie ursprünglich ausgeliefert wurden. Unter die SOP fallen heute neben JavaScript- auch z. B. ActionScript- Programme und Layout-Festlegungen mittels *Cascading Style Sheets* (CSS). Angreifer suchen deshalb nach Cross-Site-Scripting Lücken auf oft besuchten Webseiten, um die SOP auszuhebeln und die Nutzer dazu zu bringen, unwissentlich bösartige Skripte von anderen Quellen auszuführen.

Da WebSockets ebenfalls Cross-Origin-Zugriffe ermöglichen und damit Verbindungen zu beliebigen Servern aufbauen können, spezifizierte die WebSocket-Arbeitsgruppe den Origin-Header, um Browser zu schützen [FM11a]:

```
Origin: http://example.com
```

Es wird allerdings nicht nur empfohlen diesen Header im Browser zu überprüfen, sondern auch auf der Serverseite. Bei einem Handshake sollte ein WebSocket-Server die Herkunft der aktiven Inhalte eines Clients prüfen und gegebenenfalls den Verbindungsaufbau ablehnen. Dies gilt jedoch nur für Anfragen von Webbrowsern. Die Spezifikation weist darauf hin, dass das WebSocket-Protokoll nur gegen bösartigen Code aus einem Webbrowser schützen kann. Diese müssen daher beim Handshake den Origin-Header mitsenden. Für eigenständige Clients hingegen ist der Einsatz, von der Spezifikation her, rein optional. Hier ist ein kurzer Auszug aus dem RFC 6455 [FM11a]:

„While this protocol is intended to be used by scripts in web pages, it can also be used directly by hosts. Such hosts are acting on their own behalf and can therefore send fake Origin header fields, misleading the server. Servers should therefore be careful about assuming that they are talking directly to scripts from known origins and must consider that they might be accessed in unexpected ways. In particular, a server should not trust that any input is valid.“

Da das WebSocket-Protokoll auf der Transportschicht arbeitet, ist es in erster Linie dazu da, den Austausch von Bits zwischen Clients und Servern zu ermöglichen. Die Sicherheit des Datenaustauschs liegt demnach in der Verantwortung des Serverbetreibers, was in der Spezifikation zum Ausdruck gebracht wird. Für Sie als Entwickler sollte es keine Lösung sein, darauf zu hoffen, dass ein Client wirklich derjenige ist, für den er sich ausgibt.

Ein Angreifer könnte einen Client implementieren und den Handshake mit einem beliebigen Origin-Header versehen. Es wäre allerdings auch z. B. mit einem Proxy-Server möglich, den Header eines Webbrowsers zu manipulieren [Eil12]. Sie können sich daher nicht ohne Weiteres auf die Authentizität und Unverfälschtheit des Origin-Headers verlassen. Wenn Sie WebSockets verwenden und diese evtl. in sicherheitskritischen Bereichen einsetzen möchten, sollten Sie also berücksichtigen, dass die Herkunft eines Clients gefälscht sein kann.

Generell ist es empfehlenswert, wie bei allen Ihren Anwendungen, ankommende Daten zu validieren und zu escapen, um mögliche *Injection* oder auch *Cross-Site-Scripting*-*(XSS)*-Angriffe zu verhindern. Den Client schützt die Prüfung des Origin-Headers innerhalb eines Browsers vor *Cross-Site-Request-Forgery*-*(CSRF)*-Angriffen.

7.2.2 Vertrauliche Kommunikation

Uns ist nach 24 Jahren kommerziellen Internets und Webs nunmehr bewusst geworden, dass die Kommunikation unverschlüsselt ist. Sensitive Daten sollten Sie daher unbedingt davon fernhalten. Da dies zunehmend schwerer fällt, bekommen Zusätze eine immer größere Bedeutung, die die Vertraulichkeit in die Kommunikationsplattformen zurückbringen. Im Web hat sich hierfür ein Platzhirsch etabliert. Sie werden in einer überwältigenden Vielzahl an Fällen den Standard *Secure Socket Layer* (SSL) [FKK96] bzw. seinen Nachfolger

Transport Layer Security (TLS) [DR08] vorfinden. Da das standardisierte TLS den Platz des proprietären SSL eingenommen hat, werden wir im weiteren Verlauf nur noch auf TLS zu sprechen kommen.

Aus Sicht des Clients wird eine TLS-Verbindung durch das URL-Schema (*engl. Scheme*) initiiert. Im Web ist dafür das Schema https standardisiert worden. Rufen Sie eine Webressource via http auf, so wird diese über das einfache und ohne Sicherheitsmerkmale versehene HTTP angefragt und übermittelt. Bietet der Webserver auch eine TLS-gesicherte Kommunikation an, so können Sie die gleiche Ressource auch entsprechend über das URL-Schema https anfragen, das den Aufbau eines TLS-Kanals bewirkt, über den das HTTP-Nachrichtenpaar ausgetauscht wird.

In [Abschnitt 5.3](#) haben wir gesehen, dass für WebSockets das URL-Schema ws standardisiert wurde. In Analogie zu den http- und https-Schemata steht zur Etablierung eines TLS-gesicherten WebSockets das Schema wss bereit. Wenn Sie einen WebSocket mit einer URL der Form

```
wss://<host>/<pathToServer>
```

öffnen, so wird zunächst ein TLS-gesicherter Kanal aufgebaut. Darüber werden dann der WebSocket-Handshake sowie anschließend die WebSocket-Frames ausgetauscht (siehe [Bild 7.6](#)). Wie es in [Abschnitt 2.2](#) beschrieben ist, wird dabei zunächst eine TCP-Verbindung zwischen Client und Server aufgebaut. Nach dem Drei-Wege-TCP-Handshake setzt dann der TLS-Handshake an, der auf Basis kryptografischer Verfahren einen gemeinsamen Sicherheitskontext zwischen Client und Server etabliert. Damit sind beide Seiten in der Lage, Nachrichtenpakete zu ver- und entschlüsseln. Im Bezug auf WebSockets bedeutet das, dass die erste TLS-gesicherte Nachricht der WebSocket-Opening-Handshake ist. Alle über den WebSocket ausgetauschten Frames inklusive der Close Frames des Websocket-Closing Handshakes werden ebenfalls verschlüsselt übertragen.

Damit Sie TLS-gesicherte Verbindungen mit Ihrem WebSocket-Server anbieten können, müssen Sie entsprechende Konfigurationen an Ihrer Server- bzw. Framework-Umgebung vornehmen. Vom Prinzip her laufen diese immer ähnlich ab. Als Grundvoraussetzung benötigen Sie ein Paar kryptografische Schlüssel, das aus einem privaten und dem dazugehörigen öffentlichen Schlüssel besteht. Den privaten Schlüssel müssen Sie unbedingt geheim halten. Dieser wird auf der Serverseite integriert und verwendet. Der öffentliche Schlüssel hingegen muss nicht geheim gehalten werden. Im Gegenteil, damit seine Zugehörigkeit zum entsprechenden Server zweifelsfrei feststellbar ist, müssen Sie sich hierfür ein TLS-Zertifikat von einer Zertifizierungsstelle ausstellen lassen.



Besonderheit von Safari

Es ist unserer Meinung nach erwähnenswert, dass sich der Safari-Browser im Kontext der https / wss-Schemata besonders verhält. Bei diesem Browser spielt es keine Rolle, was für ein Protokoll vor der Etablierung einer ungesicherten ws-Verbindung vorgelegen hat. Chrome, IE, Opera und Firefox verhalten sich in einem solchen Fall anders. Sie reagieren mit einer Fehlermeldung bzw. Warnung, falls Sie versuchen, aus einer gesicherten https-Verbindung heraus eine ungesicherte ws-Verbindung aufzubauen.

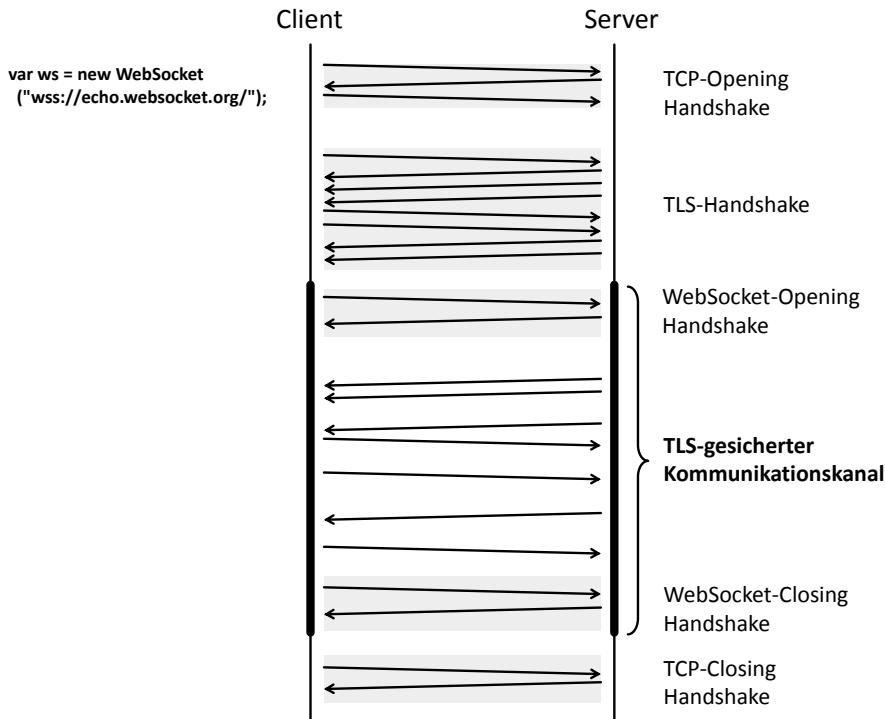


Bild 7.6 TLS-gesicherte WebSocket-Kommunikation

Unter Umständen ist derzeit eine WebSocket-Kommunikation ohne TLS-Verschlüsselung gar nicht erst möglich. Viele Mobilfunkprovider inspizieren den HTTP-Datenverkehr sehr genau. Wenn sie dabei feststellen, dass in einer HTTP-Anfrage der Upgrade-Header enthalten ist, um eine WebSocket-Verbindung zu öffnen, unterbinden sie dies. Sie werden also häufig beobachten, dass Ihre auf WebSocket basierenden Programme in mobilen Netzen auf die von Ihnen vorgesehene WebSocket-Alternative zurückfallen. Wenn Sie in Mobilfunknetzen dennoch WebSockets einsetzen wollen, müssen Sie diese mithilfe von TLS verschlüsseln. Die HTTP-Anfrage mit dem WebSocket-Handshake ist dann nicht mehr einsehbar. Der Mobilfunkprovider kann folglich einen verschlüsselt abgesetzten WebSocket-Verbindungsaufbau nicht mehr von anderem HTTP-Datenverkehr unterscheiden. In diesem Fall könnte der Verbindungsaufbau nicht unterbunden werden, ohne TLS-verschlüsselte Verbindungen generell zu unterbinden.

Wir empfehlen Ihnen allgemein, produktive WebSocket-Server nur über TLS-Verbindungen zu betreiben und zugänglich zu machen. Die Erfahrung zeigt, dass diese stets in irgendeiner Form mit Daten und Funktionen zu tun haben, die schützenswert sind. Ohne einen derartigen Grundschutz stünden sonst Tür und Tor für alle erdenklichen Missbräuche offen, die Sie im schlimmsten Fall nicht einmal als solche erkennen könnten. Außerdem ist eine TLS-Verbindung häufig die Grundlage für aufgesetzte Sicherheitsmechanismen, welche wir in den anschließenden Unterkapiteln angehen wollen.

7.2.3 Authentifizierung

Wenn Ihre WebSocket-Endpunkte nicht für die Öffentlichkeit bestimmt sind, sondern nur für einen bestimmten Benutzerkreis zugänglich sein sollen, stellt sich die Frage nach einem geeigneten Verfahren zur Kontrolle des Zugriffs. Daraus wiederum folgt die Frage, welche Verfahren prinzipiell für die Authentifizierung verwendet werden können und was der Standard dafür vorsieht. Eine Antwort darauf gibt der RFC 6455 [FM11a]:

„This protocol doesn't prescribe any particular way that servers can authenticate clients during the WebSocket handshake. The WebSocket server can use any client authentication mechanism available to a generic HTTP server, such as cookies, HTTP authentication, or TLS authentication.“

Dies bedeutet, dass es Ihnen grundsätzlich freisteht, sich für eine geeignete Authentifizierungsmethode zu entscheiden. Das Spektrum an Möglichkeiten orientiert sich – wie das Zitat aus dem Standard deutlich macht – an den im Web verbreiteten Verfahren. Die am meisten genutzten sind:

- HTTP Basic Authentication
- HTTP Digest Authentication
- Form-based Authentication
- Certificate-based Authentication

HTTP-Authentifizierung

Mit der im HTTP-Standard verankerten HTTP-Authentifizierung können Sie eine Benutzeroauthentifizierung direkt über das HTTP-Protokoll realisieren, um damit bestimmte Ressourcen auf Ihrem Server zu schützen [FHBH⁺⁹⁹]. Wenn ein Benutzer auf eine geschützte Ressource zugreifen möchte, sich allerdings noch nicht authentifiziert hat, erscheint ein Anmeldedialog, wie er in Bild 7.7 exemplarisch dargestellt ist.

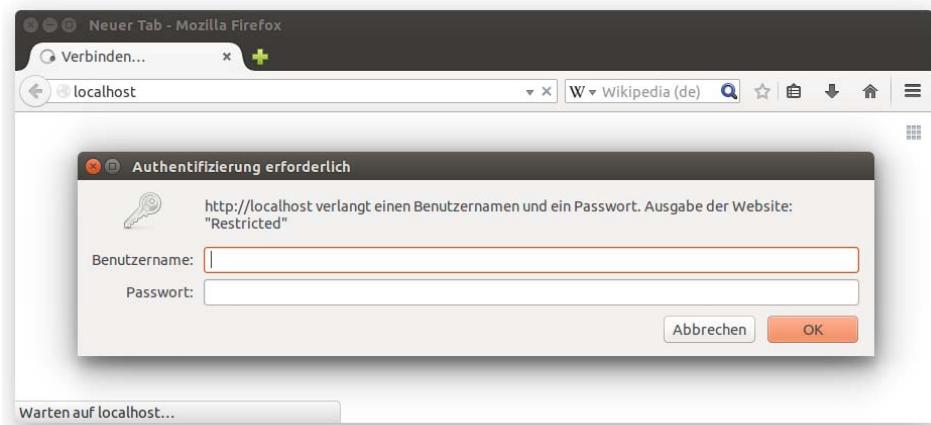


Bild 7.7 Browser-Authentifizierungsdialog im Rahmen der HTTP-Authentifizierung

Da es sich hierbei, wie gesagt, um einen Bestandteil des HTTP-Standards handelt, ist diese Art der Authentifizierung in den meisten Webbrowsern sowie HTTP-Servern und Web-

frameworks implementiert. Für die Nutzung genügt daher meist eine Aktivierung und Konfiguration in den Server-Einstellungen. Dabei werden Sie auf zwei Varianten der HTTP-Authentifizierung stoßen, die im RFC 2617 unterschieden werden.

Bei der sogenannten *HTTP Basic Authentication* sendet der HTTP-Server im Falle einer nicht authentifizierten Anfrage eine Antwort mit dem Statuscode 401 Unauthorized an den Browser, die keine Daten außer dem Response-Header enthält (siehe Bild 7.8).

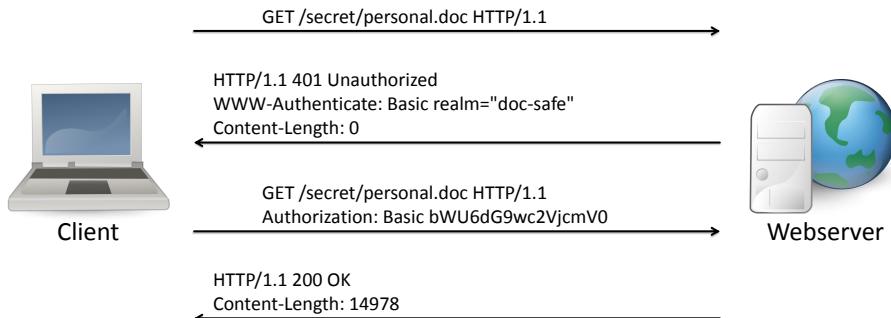


Bild 7.8 HTTP Basic Authentication

Der Statuscode 401 veranlasst den Browser, einen Dialog zur Authentifizierung mittels Benutzernamen und Passwort, wie er in Bild 7.7 dargestellt ist, anzuzeigen. Nach der Eingabe des Benutzernamens und Passworts erzeugt der Browser aus den beiden Angaben einen Token, indem er an den Benutzernamen einen Doppelpunkt und dann das Passwort anhängt. Der so konstruierte String der Form <username>:<password> wird schließlich Base64-codiert, um mögliche Schwierigkeiten mit Sonderzeichen im Benutzernamen und/oder im Passwort zu vermeiden. Dieser Token wird nun vom Browser in den Authorization-Header gesetzt und in Verbindung mit dem ursprünglichen Request an den HTTP-Server gesendet. Der Server kann nun auf den darin enthaltenen Benutzernamen sowie auf das Passwort zugreifen und auf der Grundlage einer lokalen Zugriffsrechtedatenbasis entscheiden, ob der Zugriff gewährt wird oder nicht. Hat der anfragende Client eine ausreichende Berechtigung nachgewiesen, bekommt er die angefragte Ressource mit einem 200 OK zugestellt. Eine Abkürzung des Ablaufs kann durch eine entsprechende Konstruktion der Anfrage-URL hervorgerufen werden:

`http://<username>:<password>@host/<pathToResource>`

Der Browser erkennt diesen besonderen Aufbau der URL und erstellt aus dem Benutzernamen und dem Passwort direkt einen Authorization-Header. Der an den Server abgesetzte Request enthält somit bereits die Zugriffslegitimation, wodurch das erste Nachrichtenpaar der HTTP Basic Authentication entfällt.

Da HTTP ein zustandsloses Protokoll ist, müsste der dargestellte Ablauf bei jedem weiteren Zugriff auf eine geschützte Ressource erneut durchlaufen werden. Insbesondere müsste jeder der Zugriffe mit der Eingabe des Benutzernamens und des Passworts legitimiert werden. Um dies zu vermeiden, cache der Browser Benutzernamen und Passwörter mit

der dazugehörigen URL. Wird später eine Anfrage an eine URL gerichtet, für die der Browser einen Benutzernamen und ein Passwort im Cache vorliegen hat, dann fügt dieser ohne weitere Benutzerinteraktion einen entsprechenden Authorization-Header ein.



Hinweis

Eine HTTP Basic Authentication beim Öffnen eines WebSockets können Sie folgendermaßen realisieren:

```
var ws = new WebSocket("wss://username:password@server.net/");
```

Dass es sich bei dem zunächst sehr unverständlich anmutenden Code tatsächlich um einen Benutzernamen und ein Passwort im Klartext handelt, können Sie mit einem Base64-Decoder nachvollziehen. Am einfachsten Sie benutzen hierfür einen der vielen Online-Decoder, die Sie mit einer entsprechenden Suchmaschinenanfrage angezeigt bekommen. Wenn Sie in einem solchen Base64-Decoder den letzten Teil des folgenden Beispielheaders eingeben

```
Authorization: Basic YWRtaW46c2VjcmV0
```

wird Ihnen der String `admin:secret` angezeigt. Der Benutzername ist in diesem Fall also `admin` und das Passwort entsprechend `secret`. Versuchen Sie es doch nochmals mit dem Wert des Authorization-Headers aus [Bild 7.8](#).

Die potenzielle Gefahr beim HTTP-Basic-Authentication-Verfahren ist also, wie Sie gerade selber gesehen haben, dass ein Angreifer, der Ihren Datenverkehr mitschneidet, ebenfalls problemlos die Zeichenfolge Ihrer Anmeldedaten decodieren kann.

Ein etwas höheres Maß an Sicherheit bietet die zweite standardisierte HTTP-Authentifizierungsmethode, die sogenannte *HTTP Digest Authentication*. Diese funktioniert vom Ansatz her wie die HTTP Basic Authentication, was Sie leicht an der Gestalt des in [Bild 7.9](#) illustrierten Protokolls erkennen können.

Unterschiede zur HTTP Basic Authentication beginnen mit der ersten Antwort des Servers nach einer nicht autorisierten Anfrage auf eine geschützte Ressource. Bei der HTTP Digest Authentication sendet der Server im `WWW-Authenticate`-Header zusätzlich eine Zufallszahl mit, die als `nonce` bezeichnet wird. Diese verwendet der Client, um nachzuweisen, dass er in Kenntnis des Passworts ist, ohne dieses direkt über die Leitung an den Server schicken zu müssen. Dazu berechnet der Browser einen Code anhand des Passworts, das der Benutzer im Authentifizierungsdialog eingegeben hat. Genauer gesagt, wird dieser Code unter Zuhilfenahme einer kryptografischen Hashfunktion erzeugt, bei der es sich häufig um MD5 handelt. Insgesamt werden mehrere Daten verknüpft und anschließend gehasht. In [Bild 7.9](#) ist zu sehen, wie genau der `response`-Code erzeugt wird. Zum Schluss wird er im Authorization-Header mit der Kennung `response` an den Server übermittelt.

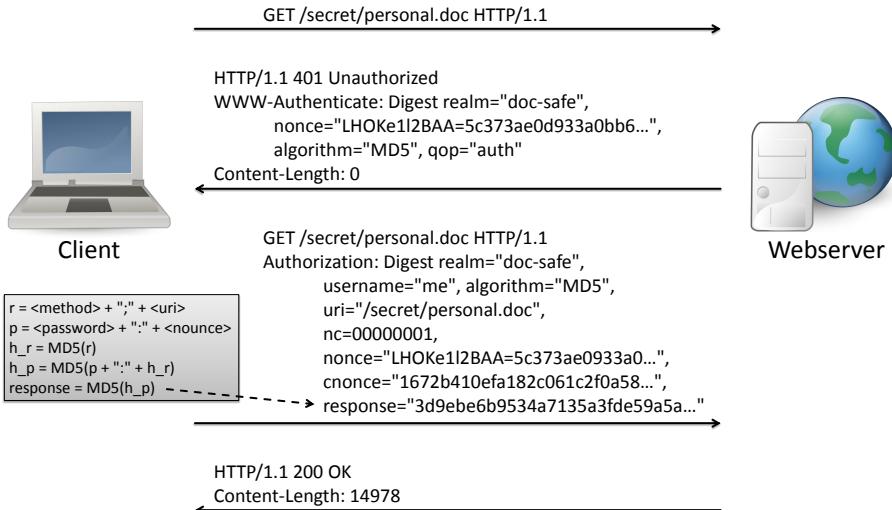


Bild 7.9 HTTP Digest Authentication

Bemerkenswert an dieser Stelle ist, dass der Code keine direkten Rückschlüsse auf das Passwort zulässt und dieses nicht – wie es bei der HTTP Basic Authentication der Fall ist – im Klartext im `Authorization`-Header enthalten ist.

In der Theorie ist es also möglich, jede Art von Authentifizierung, die bisher mit HTTP möglich war, auch in die WebSocket-Technologie zu integrieren. In der Praxis sieht es allerdings anders aus. Da WebSockets noch sehr jung sind, sind zugehörige Authentifizierungsverfahren in vielen Clients und Servern bzw. serverseitigen Frameworks noch nicht optimal implementiert. Dadurch haben Sie es mit zwei potenziellen Problemquellen zu tun, der Server- und der Clientseite. Wenn Sie z. B. einen Reverse Proxy für Ihren WebSocket-Server einrichten wollen, kann es sein, dass Ihr HTTP-Proxy-Server nicht mit WebSockets umgehen kann (Näheres dazu in [Abschnitt 7.2.4](#)).

Gehen wir mal davon aus, der von Ihnen eingerichtete Proxy baut problemlos eine WebSocket-Verbindung auf und Sie möchten zusätzlich eine HTTP-Authentifizierung nutzen, um Ihre Ressourcen zu schützen. Leider kann das unter Umständen nicht möglich sein, weil der Server die HTTP-Authentifizierung nicht für den WebSocket-Handshake unterstützt.

Ein Proxy-Server, der bereits WebSockets auch in Verbindung mit der HTTP Basic Authentication beherrscht, ist z. B. nginx. Probleme kann hier allerdings noch die HTTP Digest Authentication verursachen, da dieses Verfahren nicht standardmäßig in nginx implementiert ist. Es ist zwar ein Plug-in-Modul [[Swi11](#)] verfügbar, das Sie zusätzlich installieren müssen, dieses unterstützt nur leider nicht das WebSocket-Protokoll. In diesem Fall funktioniert die Authentifizierung nicht, auch wenn in Ihrem Browser bzw. Client die HTTP Digest Authentication für WebSockets implementiert ist. Falls Sie diesen Server also als Proxy-Server einrichten wollen, sollten Sie dieses Problem beachten.

Dies war nur ein konkretes Beispiel anhand von nginx, um die Problematik zu verdeutlichen. Solche Schwierigkeiten können natürlich auch bei anderen Servern oder auch

serverseitigen Frameworks auftreten. Aus diesem Grund empfehlen wir Ihnen, HTTP-Authentifizierungen für WebSockets gründlich zu testen. Falls Sie serverseitige Frameworks wie Node.js oder das Play Framework verwenden, haben Sie natürlich leichter die Möglichkeit, fehlende Implementierungen selbst zu programmieren.

Betrachten wir nun ein bestehendes Problem auf der Clientseite. Nachdem Sie Ihren Benutzernamen und Ihr Passwort für die HTTP-Authentication in einem Dialog eingegeben haben, generiert der Browser daraus ein Token und schickt dieses im Header an den Server. Wie bereits beschrieben, ist der Server nun durch das Token im Header in der Lage, die Authentifizierung zu überprüfen. Einige Browser haben allerdings die Eigenschaft, dass sie den Authorization-Header beim WebSocket-Handshake-Request nicht mitschicken. Da die HTTP-GET-Anfrage für den Handshake keinen Token enthält, ist diese Anfrage für den Server ungültig und dieser verweigert die Anfrage. Er gibt daraufhin den Statuscode 401 Unauthorized (siehe [Abschnitt 2.2.4](#)) zurück und nicht den erhofften Statuscode 101. Diese Eigenschaft weist z. B. die aktuelle Version von Safari (7.1) auf. Die aktuelle Version von Firefox (34), Internet Explorer (11), Chrome (Version 39) und Opera (26) hingegen schicken den Authorization-Header beim Handshake-Request mit. Mit diesen Browsern ist es momentan also möglich, eine HTTP-Authentifizierung mit WebSockets zu realisieren.

Wie Sie sehen, ist eine Zugriffskontrolle über das HTTP-Protokoll nur bedingt möglich. Clientseitig und serverseitig ist noch nicht alles im grünen Bereich. Grundsätzlich erlaubt Ihnen das WebSocket-Protokoll eine HTTP-Authentifizierung, allerdings wurde dieses Verfahren noch nicht in allen Clients und Servern für WebSockets implementiert.

Es gibt außerdem noch andere grundsätzliche Nachteile bei den HTTP-Authentifizierungsmechanismen. Beide realisieren nur eine einseitige Authentifizierung, nämlich die eines Benutzers gegenüber einem Server. Der Server authentifiziert sich nicht gegenüber dem Benutzer, was problematisch ist, da dem Server sensitive Daten wie z. B. Passwörter übermittelt werden müssen. Daher ist es für Sie unverzichtbar zu wissen, mit welchem Server Sie tatsächlich kommunizieren. Des Weiteren bieten die Verfahren keine Vertraulichkeit, sodass die initiale und sich wiederholende Legitimation in Form des Zugangsdatenaustausches von Dritten mitgeschnitten werden kann. Aus diesem Grund können wir Ihnen nur sehr empfehlen, beide Authentifizierungsverfahren über einen TLS-gesicherten Kanal zu implementieren, der auch nach der Authentifizierung aufrechterhalten bleibt, damit die Authorization-Header nachfolgender Anfragen ebenfalls geschützt sind.

Formularbasierte Authentifizierung

Beim überwiegenden Anteil von Webanwendungen ist die Authentifizierung auf Basis eines (HTML-)Formulars realisiert (*engl. form-based Authentication*). Benutzer können also z. B. nur auf bestimmte Teile der Webanwendung zugreifen, wenn sie sich über eine HTML-Seite mit einem Formular authentifiziert haben (siehe [Bild 7.10](#)).

Da das HTTP-Protokoll zustandslos ist und daher alle Anfragen voneinander unabhängig sind, müsste sich ein Client bei jeder Anfrage aufs Neue authentifizieren. Um dies zu umgehen, erzeugt der Webserver nach einer erfolgreichen Authentifizierung eine Session-ID für den Client und schickt ihm diese in der Antwort zu. Dies wird in der Regel durch Cookies realisiert, die ebenfalls serverseitig erzeugt werden. Die Session-ID wird in diesem Cookie gespeichert und im Header der Antwort eingebettet. Dieses Cookie wird nun bei jeder

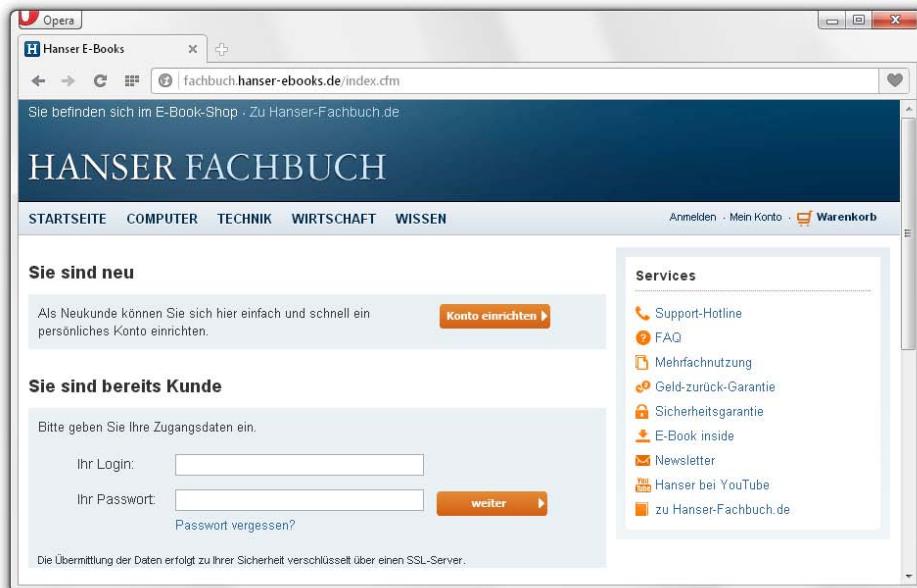


Bild 7.10 Login-Formular auf der Seite <http://fachbuch.hanser-ebooks.de/>

folgenden Anfrage vom Client mitgeschickt, wodurch der Webserver überprüfen kann, ob sich der Benutzer zuvor schon angemeldet hat.

Eine Session-ID sollte aus einer langen kryptografisch geeigneten Zufallszahl oder Zeichenkette bestehen, damit diese nicht leicht erraten werden kann. In vielen Web-Frameworks können und werden Cookies noch mit einer zusätzlichen zufällig erzeugten Zeichenkette signiert. Das Sicherheitslevel wird hierdurch erhöht, da der Server ein manipuliertes Cookie direkt erkennen kann.

Cookies und deren Session-IDs werden nun bei jedem Request auf ihre Gültigkeit überprüft. Eine gültige Session-ID ist erst vorhanden, wenn sich der Benutzer erfolgreich authentifiziert hat und danach eingeloggt ist. Falls bei einem Request auf eine geschützte URL keine gültige Session-ID vorliegt, wird der Client z. B. auf die Seite mit der Login-Maske weitergeleitet.

Cookies werden ebenfalls bei jedem WebSocket-Handshake-Request mitgeschickt. Einige frei verfügbare Frameworks wie Socket.io, WebSocket-Node und das Play Framework bieten Ihnen eine Funktion an, mit der Sie auf die Cookies des Handshake-Requests zugreifen können. Dadurch können Sie die darin enthaltenen Daten auswerten und entscheiden, ob der Request angenommen wird oder nicht.

In den folgenden Unterkapiteln möchten wir Ihnen zeigen, wie Sie den WebSocket-Handshake selber auswerten können, um dadurch eine Zugriffskontrolle für WebSockets zu ermöglichen. Die Implementierungen wollen wir Ihnen anhand des Node.js Webframeworks Express.js sowie der beiden Module Socket.io und WebSocket-Node zeigen. Wir werden außerdem mit Ihnen durchgehen, wie Sie bei dem Play Framework, das Ihnen keinen Zugriff auf den Handshake erlaubt, trotzdem eine Zugriffskontrolle realisieren können.

Beide Frameworks bieten Ihnen Möglichkeiten zur Erzeugung und Verwaltung von Cookies sowie Session-IDs. Eine Einführung in das Play Framework können Sie ggf. noch mal in [Abschnitt 6.4](#) nachlesen. Eine Einführung in Express.js finden Sie im [Anhang D](#).



Die kompletten Quelltexte der gezeigten Anwendungen finden Sie natürlich auf unserer Homepage:

<http://websocket101.org/>

Formularbasierte Authentifizierung mit Express.js

Als Erstes möchten wir Ihnen anhand von Express.js zeigen, wie Sie allgemein bestimmte Pfade Ihrer Webanwendung mit einer Zugriffskontrolle schützen können. Dazu gehört auch, wie Sie Sessions erzeugen und signieren können, um Zustandsinformationen des Clients zu speichern. Wir werden an dieser Stelle noch nicht auf die Zugriffskontrolle für WebSockets eingehen, sondern zunächst das Prinzip und die Implementierung des Verfahrens erklären. Wenn Sie das verstanden haben, können Sie dies problemlos auf WebSockets übertragen. Natürlich werden wir das Thema in den nächsten Unterkapiteln ebenfalls angehen.

Erstellen Sie zunächst mit der Eingabeaufforderung oder einem Terminal eine neue Express.js-Anwendung:

```
express session_app  
cd session_app  
npm install
```

Als Nächstes bearbeiten wir den Quelltext der Datei *app.js*. Importieren Sie vier zusätzliche Module; die zwei Module `session` und `store`, damit wir die Session-IDs der Cookies in einem `MemoryStore` ablegen können, sowie zwei Module für einen HTTP-Server (siehe [Listing 7.6](#)).

Listing 7.6 Zusätzliche Module für das Parsen der Cookies und einen `MemoryStore`

```
1 ...  
2 var session = require('express-session');  
3 var store = new session.MemoryStore();  
4 var httpServer = require("http").Server(app);  
5 ...
```

Nun müssen wir ein paar Konfigurationen an unserer Anwendung vornehmen (siehe [Listing 7.7](#)).

Listing 7.7 Konfiguration der Express.js-Anwendung

```
1 // Dies sollte ein zufällig generierter Schlüssel sein.  
2 var signKey = "RHDwSKMa&vcxnkj"  
3  
4 app.use(bodyParser.json());
```

```

5 app.use(bodyParser.urlencoded({ extended: false }));
6 app.use(cookieParser(signKey));
7 app.use(express.static(path.join(__dirname, 'public')));
8
9 app.set('views', path.join(__dirname, 'views'));
10 app.set('view engine', 'jade');
11 app.use(session({store:store}));
12 ...
13 httpServer.listen(3000, function(){
14   console.log("Express-Server laeuft auf dem Port 3000");
15 });

```

Der Schlüssel `signKey` wird zum Signieren der Cookies benötigt ([Zeile 6](#)). Damit kann verhindert werden, dass Angreifer gültige Cookies selber erzeugen oder manipulieren können. Als Nächstes definieren wir den Pfad sowie die Engine der Views und legen den `MemoryStore`, welchen wir in [Listing 7.6](#) erzeugt haben, für die Verwaltung unserer Session-ID fest. Schließlich muss noch ein Port für den Server festgelegt werden.

Kommen wir nun zum Auslesen der Session-ID des Clients. Diese können wir dem Argument `req (request)` der Callback-Funktion eines `app.get()`-Route-Handlers entnehmen.

Der Code in [Listing 7.8](#) soll verdeutlichen, wie z. B. überprüft werden kann, ob die Session über einen Benutzernamen verfügt. Falls keiner vorhanden ist, wissen wir, dass der Client bisher noch keine Autorisierung hat. Diese Abfrage könnten wir jetzt bei jedem Pfad bzw. jedem `app.get()`-Route-Handler, den wir schützen wollen, anbringen. Da das allerdings keine elegante und flexible Lösung ist, definieren wir für diesen Zweck eine Funktion.

Listing 7.8 Prinzip der Abfrage einer Session aus einem Request

```

1 app.get('/', function(req, res) {
2   if(!req.session.username) {
3     console.log("Der Benutzer ist nicht autorisiert");
4   }
5   else{
6     console.log("Der Benutzer ist autorisiert");
7   }
8 });

```

Die Funktion `requiresLogin()` in [Listing 7.9](#) überprüft, ob für den Request eine gültige Session vorliegt. Ist das nicht der Fall, wird die Anfrage ignoriert und stattdessen auf eine Login-Seite weitergeleitet. Der geschützte Pfad wird außerdem in der Session abgespeichert. Warum, erklären wir zwei Absätze später. Falls bereits eine gültige Session vorliegt, wird der Client zu der geschützten Seite weitergeleitet.

Listing 7.9 Funktion zur Abfrage einer gültigen Session

```

1 function requiresLogin(req, res, next) {
2   if(!req.session.username) {
3     req.session.path = req.url;
4     res.redirect('/login');
5   }
6   else{

```

```

7         next();
8     }
9 }
10 ...
11 app.get('/', requiresLogin, function(req, res) {
12     res.render('index', {title:'Home'});
13 });

```

Die Funktion können wir jetzt an jedem `app.get()`-Route-Handler, der geschützt werden soll, verwenden (siehe [Zeile 11](#)). Im nächsten Schritt legen wir zwei Route-Handler für die Login-Seite fest.

Listing 7.10 Festlegen der Login-View und die Auswertung des Login-Formulars

```

1 app.get('/login', function (req, res) {
2     res.render('login', {title:'Login'});
3 });
4
5 app.post('/login', function (req, res) {
6     if(req.body.username == "admin" && req.body.password == "123") {
7         var path = req.session.path;
8         req.session.regenerate(function(err) {
9             var hour = 60*60*1000;
10            req.session.username = req.body.username;
11            req.session.cookie.expires = new Date(Date.now()+hour);
12            if(path) {
13                res.redirect(path);
14            }
15            else{
16                res.redirect("/");
17            }
18        });
19    }
20    else {
21        res.redirect('back');
22    }
23 });

```

Mit dem Route-Handler `app.get('/login')` definieren wir zunächst eine Login-Seite mit dem Titel `login`. Als Nächstes erstellen wir den Route-Handler `app.post('/login')`, der das Login-Formular auswertet. Für dieses Beispiel machen wir es uns einfach und fragen nur ab, ob der Benutzername mit `admin` und das Passwort mit `123` übereinstimmt. In der Produktion findet an dieser Stelle in der Regel eine Datenbankabfrage statt. Falls sich der Benutzer nun erfolgreich eingeloggt hat, weisen wir der Session seinen Benutzernamen zu und legen ihre Gültigkeitsdauer fest. Danach leiten wir den Client an den Pfad weiter, den wir vorher in der Session gespeichert haben. Ist kein Pfad vorhanden, leiten wir den Benutzer auf die Startseite weiter.

Im letzten Schritt müssen wir noch die View-Dateien für den Login und die Startseite festlegen (siehe [Listing 7.11](#) und [Listing 7.12](#)). Letztere ist bereits nach der Erzeugung des Projekts vorhanden und muss nicht neu erstellt werden.

Listing 7.11 View-Template der Login-Seite (*login.jade*)

```

1  extends layout
2
3  block content
4    h1 Login
5    form(action="/login",method="POST")
6      input(type="text",placeholder="Benutzername", name="username")
7      br
8      input(type="password", placeholder="Passwort", name="password")
9      br
10     input(type="submit", value="Login")

```

Listing 7.12 View-Template der Startseite (*index.jade*)

```

1  extends layout
2
3  block content
4    h1= title
5    p Willkommen auf der Seite #{title}

```

Nun können wir die Anwendung mit folgendem Befehl starten:

```
node app.js
```

Rufen Sie in einem Browser die URL

```
http://localhost:3000/
```

auf. Da dieser Pfad von Ihnen geschützt wurde, werden Sie zunächst auf die Login-Seite umgeleitet (siehe [Bild 7.11](#)).



Bild 7.11 Login-Seite unserer Express.js-Webanwendung

Wenn Sie sich mit dem Benutzernamen admin und dem Passwort 123 angemeldet haben, erlaubt Ihnen die programmierte Anwendung, auf die Startseite zuzugreifen (siehe [Bild 7.12](#)).

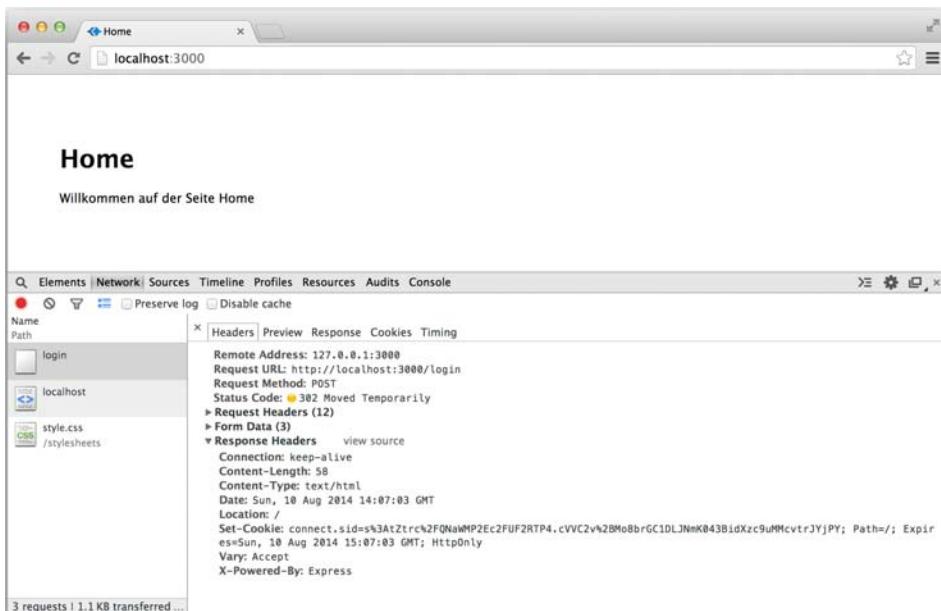


Bild 7.12 Startseite unserer Express.js-Webanwendung

Benutzen Sie Firebug oder die Chrome Developer Tools, um sich das generierte Cookie im Response-Header der Login-Seite genauer anzusehen (siehe Bild 7.12). Es beinhaltet die signierte Session-ID, den Zeitpunkt, an dem es erzeugt wurde, und einen Ablaufzeitpunkt. Der Eintrag *HttpOnly* ist ein zusätzlicher Schutz, der verhindert, dass das Cookie über JavaScript abrufbar ist und so z. B. die Session-ID von Angreifern gestohlen werden kann. Bei Express.js handelt es sich hierbei um eine standardmäßige Einstellung. In anderen Frameworks, wie z. B. in Play, ist dies allerdings nicht der Fall. Daher empfehlen wir Ihnen dringend, die *HttpOnly*-Einstellung grundsätzlich zu überprüfen.

Sie haben somit eine formularbasierte Authentifizierung für Webanwendungen in Express.js realisiert. Im folgenden Unterkapitel widmen wir uns nun der nächsten Problemstellung, der Authentifizierung einer WebSocket-Verbindung.

Formularbasierte Authentifizierung mit Express.js und Socket.io

Der Handshake wird bei Socket.io durch einen Ajax-Request bzw. einen JSONP-Request ausgelöst, für den eine Authentifizierung festgelegt werden kann. Dabei besteht die Möglichkeit, die Cookies, die im Handshake mitgeschickt werden, zu überprüfen [Rau14]. Das können Sie nutzen, um ausschließlich eingeloggten Clients, die über eine gültige Session verfügen, den Aufbau einer WebSocket-Verbindung zu erlauben.

Wir werden also in diesem Kapitel die Webanwendung aus dem vorherigen Kapitel mit einer Zugriffskontrolle für WebSockets durch Socket.io erweitern. Vergessen Sie nicht, vorher das Modul Socket.io über den Paketmanager zu installieren. Zusätzlich brauchen wir auch noch drei weitere Module.

```
npm install socket.io
```

```
npm install cookie
npm install express-session
npm install cookie-parser@1.3
```

In [Listing 7.13](#) finden Sie die Hilfsmittel, um in der Datei *app.js* eine Zugriffskontrolle zu realisieren.

Listing 7.13 Hilfsmittel zur Realisierung einer Authentifizierung mit Socket.io

```
1 var cookie = require('cookie');
2 var io = require('socket.io').listen(httpServer);
3 io.use(function (socket, next) {
4     if(socket.handshake.headers.cookie) return next();
5     next(new Error("Es wurde kein Cookie uebertragen"));
6 });
```

Der Methode `io.use` wird eine Funktion übergeben, die zwei Argumente besitzt; die `socket.io`-Verbindung (`socket`) sowie die Methode `next()`. Falls im Header des Handshakes ein Cookie zu finden ist, wird die Methode `next()` ohne Argument zurückgegeben. Im nächsten Schritt kann das Programm eine WebSocket-Verbindung aufbauen. Dazu kommt es nicht, wenn das Cookies fehlt, denn in diesem Fall wird in der Methode ein neues `Error`-Objekt erzeugt ([Zeile 5](#)). Als Fehlermeldung können wir hier z. B. "Es wurde kein Cookie uebertragen" festlegen.

Da wir allerdings für die Zugriffskontrolle nicht nur wissen müssen, ob ein Cookie vorhanden ist, sondern auch ob dieses gültig ist, werten wir zusätzlich dessen Daten aus.

Listing 7.14 Authentifizierung mit Socket.io

```
1 io.use(function (socket, next) {
2     if(socket.handshake.headers.cookie) {
3         socket.cookie = cookie.parse(socket.handshake.headers.
4             cookie);
5         sessionID = cookieParser.signedCookie(socket.cookie[
6             'connect.sid'], signKey);
7         store.get(sessionID, function(err, session) {
8             if(err){
9                 next(new Error("Fehler aufgetreten"));
10            }
11            else if(typeof session === "undefined") {
12                next(new Error("Ungueltige Session"));
13            }
14            else if(session.hasOwnProperty("username")) {
15                next();
16            }
17        });
18    }
19    next(new Error("Es wurde kein Cookie uebertragen"));
20});
```

Da die Methode `io.use()` keine Methode von Express.js ist, sondern von `Socket.io`, wird das Cookie im Objekt `socket.handshake.headers.cookie` nicht direkt als JSON geparsst,

sondern liegt zunächst als String vor. Mit der Methode `cookie.parse()` (Zeile 3) wandeln wir es deshalb zunächst in ein JSON-Objekt um, was uns erleichtert, die signierte Session-ID herauszulesen. Das Objekt speichern wir in einer Variablen ab. Nun decodieren wir die signierte Session-ID mit der Methode `cookieParser.signedCookie()` und dem Signierungsschlüssel aus der `signKey`-Variablen (siehe Listing 7.7, Seite 168). Die decodierte Session-ID speichern wir ebenfalls in einer Variablen ab. Im nächsten Schritt rufen wir mit der Methode `store.get()` unseren Session-Store auf, um zu prüfen, ob diese Session-ID gültig ist und außerdem über einen Benutzernamen verfügt. Ist dies nicht der Fall, verweigern wir den Handshake. Verfügt die gültige Session-ID über einen Benutzernamen, gewähren wir den Zugriff.

Um dies zu testen, bauen wir eine Echo-Funktion in unsere Anwendung ein (siehe Listing 7.15).

Listing 7.15 Echo-Funktion für die Express.js- und Socket.io-Anwendung

```
1 io.sockets.on("connection", function(socket) {
2     socket.on("message", function(message) {
3         socket.send(message);
4     });
5 });


```

Unsere Startseite mit dem Socket.io-Testclient passen wir ebenfalls an (siehe Listing 7.16).

Listing 7.16 View-Template der Startseite mit Testclient (*index.jade*)

```
1 extends layout
2
3 block content
4     h1= title
5     p Willkommen auf der Seite #{title}
6     script.
7         var socket = io.connect();
8         socket.on("connect",function(){
9             socket.send("WebSocket aufgebaut");
10        })
11        socket.on("message",function(message){
12            console.log(message);
13        });
14        socket.on("error", function(message){
15            console.log(message);
16        })


```

Zum Schluss fügen wir noch die Zeile aus Listing 7.17 in das Layout-Template (*layout.jade*) ein, die den Zugriff auf den Socket.io Client ermöglicht. Die Datei befindet sich ebenfalls im Ordner *views*.

Listing 7.17 Zusätzliche Codezeile für den Zugriff auf den Socket.io Client (*layout.jade*)

```
1 script(src='/socket.io/socket.io.js')
```

Starten Sie nun abschließend die programmierte Anwendung mit dem Befehl

```
node app.js
```

und rufen Sie die Adresse

`http://localhost:3000`

im Browser auf. Sie werden wieder augenblicklich auf die Login-Seite geleitet. Auf dieser Seite ist es nicht möglich, sich ohne Anmeldung mit dem Socket.io-Server zu verbinden (siehe Bild 7.13).



Bild 7.13 Verweigerung des Handshakes

Wir haben also mit der formularbasierten Authentifizierung mit Express.js und Socket.io erreicht, dass Sie erst eine WebSocket-Verbindung aufbauen können, nachdem Sie sich erfolgreich eingeloggt haben (siehe Bild 7.14).

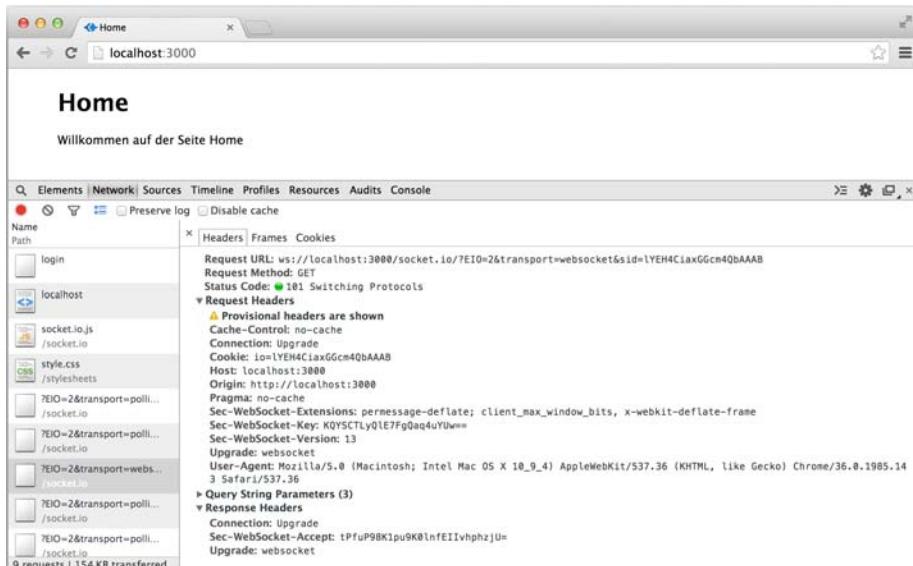


Bild 7.14 Erfolgreiche Authentifizierung des WebSocket-Clients

Formularbasierte Authentifizierung mit Express.js und WebSocket-Node

In diesem Unterkapitel möchten wir die formularbasierte Authentifizierung mit WebSocket-Node, einer reinen Implementierung des WebSocket-Protokolls, umsetzen. Die Webanwendung wird weiterhin auf unser erstes Beispiel mit Express.js aufbauen. Installieren Sie zuerst WebSocket-Node mit dem Befehl:

```
npm install websocket
```

Wir beginnen wieder damit, den Quelltext der Datei *app.js* anzupassen (siehe Listing 7.18).

Listing 7.18 Importieren und Konfigurieren von WebSocket-Node

```
1 ...
2 var WebSocketServer = require("websocket").server;
3 var webSocketServer = new WebSocketServer({
4     httpServer:httpServer,
5     autoAcceptConnections:false
6 });
7 ...
```

Dort stellen wir ein, dass der WebSocket-Node-Server auf denselben Port hören soll wie unser HTTP-Server. Außerdem möchten wir nicht, dass der Server alle Verbindungen akzeptiert, denn darum wollen wir uns schließlich selber kümmern.

Listing 7.19 Auswertung des Handshake-Requests unter WebSocket-Node

```
1 webSocketServer.on('request', function(request) {
2     var sid = "";
3     for (var i = 0; i < request.cookies.length; i++) {
4         if(request.cookies[i].name === "connect.sid") {
5             sid = request.cookies[i].value;
6         }
7     };
8
9     var sessionID = cookieParser.signedCookie(sid, signKey);
10    store.get(sessionID, function(err, session) {
11        if(!session || !session.username) {
12            request.reject();
13            return;
14        }
15        else {
16            request.accept();
17        }
18    });
19});
```

Der Event-Listener `webSocketServer.on('request')` ruft in der ersten Zeile die Callback-Funktion auf, wenn der Client einen WebSocket-Handshake-Request verschickt hat. Der Callback-Funktion wird ein `request`-Objekt als Argument übergeben, über das wir an die signierte Session-ID im Cookie herankommen. Zunächst suchen wir also mithilfe einer for-Schleife die `connect.sid` aus dem Cookie heraus und speichern sie in der Variablen `sid` ab.

Im nächsten Schritt parsen wir die Session-ID wieder mit der Methode `connect.utils.-parseSignedCookie()`, welche dazu natürlich den Signierungsschlüssel `signKey` aus [Listing 7.7 \(Seite 168\)](#) benötigt. Im letzten Schritt fragen wir im Session-Store nach, ob diese Session-ID vorhanden bzw. gültig ist. Falls nicht oder wenn die Session keinen Benutzernamen hat, lehnen wir den Request mit `request.reject()` ab. Bei einer gültigen Session akzeptieren wir den Handshake mit `request.accept()`.

Nun ergänzen wir unseren WebSocket-Node-Server mit der Echo-Funktion:

Listing 7.20 Echo-Funktion mit WebSocket-Node

```

1  webSocketServer.on("connect", function(socket) {
2      socket.on("message", function(message) {
3          if(message.type === 'utf8') {
4              socket.send(message.utf8Data);
5          }
6      });
7 });

```

Als Letztes fehlt noch die Startseite mit einem Testclient.

Listing 7.21 View-Template für die Startseite der Express.js- und WebSocket-Node-Anwendung

```

1 extends layout
2
3 block content
4     h1= title
5     p Willkommen auf der Seite #{title}
6     script(charset="utf-8")
7         var ws = new WebSocket("ws://" + window.location.host);
8         ws.onopen = function(){
9             ws.send("Hallo");
10        }
11        ws.onmessage = function(evt){
12            console.log(evt.data);
13        }

```

Die Login-Seite ist identisch mit dem Quelltext aus [Listing 7.11 \(Seite 171\)](#). Nun können Sie das Programm wieder mit

`node app.js`

starten und die URL

`http://localhost:3000`

in einem Browser aufrufen.

Sie landen erneut auf der Login-Seite. Wenn Sie jetzt versuchen, eine WebSocket-Verbindung aufzubauen, wird diese ordnungsgemäß verweigert (siehe [Bild 7.15](#)).

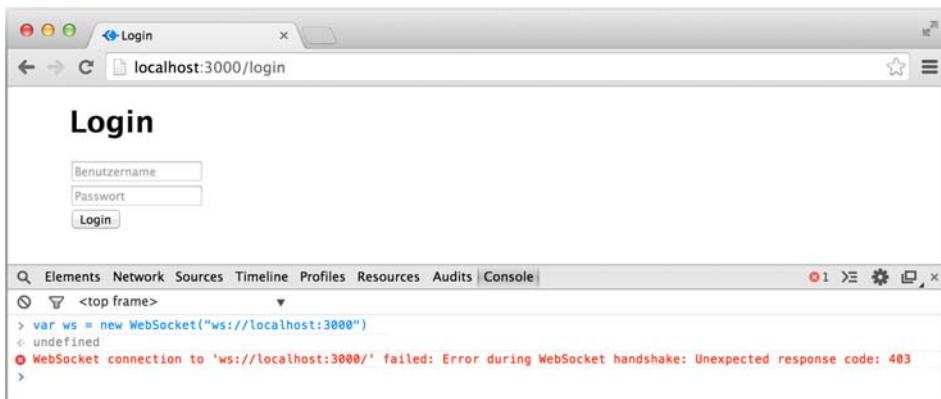


Bild 7.15 WebSocket-Node verhindert den Handshake.

Erst nach dem Login wird dies von der Anwendung erlaubt (siehe Bild 7.16).

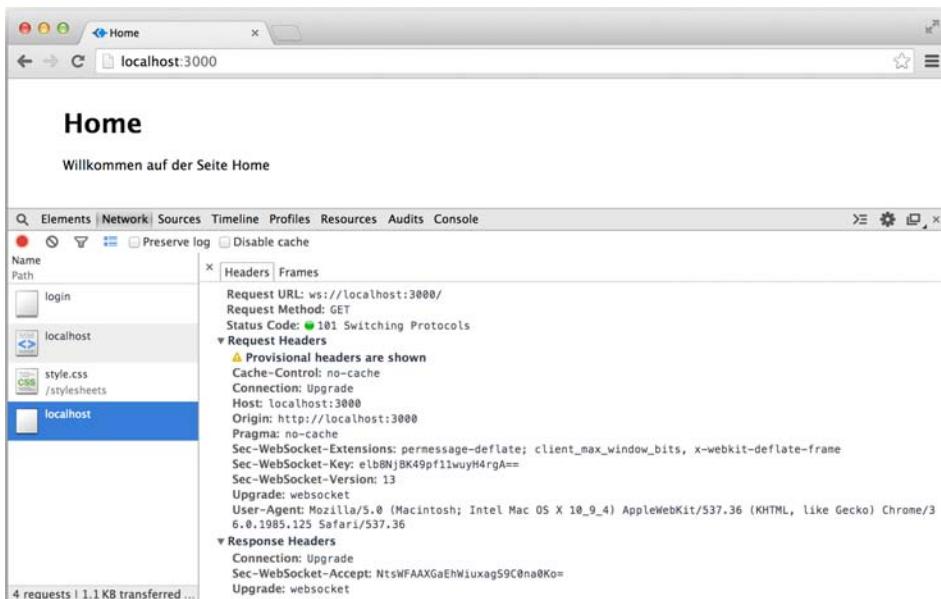


Bild 7.16 WebSocket-Verbindung wird vom WebSocket-Node-Server akzeptiert.

Formularbasierte Authentifizierung mit dem Play Framework

Play erlaubt es Entwicklern, selber zu bestimmen, auf welche Controller oder auch Controller-Methoden ein eingeloggter bzw. nicht eingeloggter Benutzer zugreifen darf. Dies gilt auch für WebSocket-Endpunkte. Wie das funktioniert, möchten wir Ihnen in diesem Unterkapitel demonstrieren.

Zunächst zeigen wir Ihnen, wie Sie Controller-Klassen bzw. Actions, die eine HTTP-Antwort wiedergeben, durch eine Zugriffskontrolle schützen können. Ausgehen wollen

wir von dem Echo-Server aus [Abschnitt 6.4.9](#). Zusätzlich benötigen wir noch zwei weitere Klassen; einen Controller für die Authentifizierung und eine Klasse, die prüft, ob der Client über eine gültige Session verfügt oder nicht.

Zunächst erstellen Sie eine Datei mit dem Namen *Credential.java* im Ordner *app/controllers*. Dabei handelt es sich um den Controller für die Authentifizierung. Im ersten Schritt definieren wir eine Controller-Action, die die View der Login-Maske rendert.

Listing 7.22 Login-Action

```
1 public class Credential extends Controller {
2     public static Result login() {
3         return ok(login.render());
4     }
}
```

Wir definieren dazu ebenfalls eine URL in der *routes*-Datei (siehe [Listing 7.23](#)).

Listing 7.23 Route-Pfad zur Login-Maske

```
1 GET      /login           controllers.Credential.login()
```

Anschließend erstellen wir die View und definieren den Inhalt (siehe [Listing 7.24](#)).

Listing 7.24 View der Login-Maske

```
1 @main("Login") {
2     <h2>Login</h2>
3     <form method="POST" action="/login">
4         <input type="text" name="username" placeholder=
5             "Benutzername"/>
6         <br/>
7         <input type="text" name="password" placeholder="Passwort"/>
8         <br/>
9         <input type="submit" value="login" />
10    </form>
11 }
```

Um ein abgeschicktes Formular in einer Play-Action auszuwerten, gibt es die Helper-Klasse `play.data.Form`. Dadurch brauchen wir nicht mehr jeden POST-Parameter einzeln auszulesen, sondern definieren eine Klasse mit diesen Parametern als Instanzvariablen. In unserem Fall erstellen wir eine statische Klasse `Login` in der Datei *Credential.java* (siehe [Listing 7.25](#)).

Listing 7.25 Statische Login-Klasse für die Helper-Klasse Form

```
1 public static class Login {
2     @Required
3     public String password;
4     @Required
5     public String username;
6
7     public String validate() {
8         if(!username.equals("admin") || !password.equals("123")) {
```

```

9         return "Ungültiger Benutzer oder ungültiges Passwort";
10    }
11    return null;
12  }
13 }
```

Hier definieren wir die zwei Instanzvariablen `password` und `username`. Beide müssen `public` sein, damit die Helper-Klasse darauf zugreifen kann, in der die Methode `validate()` implementiert ist. Diese Methode teilt dem Form-Helper mit, auf welche Weise die Instanzvariablen validiert werden sollen. Da dies ein Beispiel ist, um Ihnen die Funktionsweise von Formularbehandlungen in Play näherzubringen, legen wir die Login-Daten direkt fest. Der Benutzer `admin` muss sich mit dem Passwort `123` anmelden. Auf diese Weise sollten Sie, abgesehen für dieses einfache Beispiel, nie die eingegebenen Daten überprüfen. In einer Produktivumgebung wird bei der if-Abfrage z. B. eine Datenbankanfrage wie `if(User.find("username", "password"))...` abgeschickt. Die `validate()`-Methode ist für den Form-Helper so definiert, dass im Erfolgsfall `null` und im Fehlerfall ein String mit der Fehlermeldung zurückgegeben wird.

Im nächsten Schritt definieren wir eine Controller-Action, um das Formular der Login-Maske auszuwerten bzw. den Login zu authentifizieren (siehe [Listing 7.26](#)).

Listing 7.26 Controller-Action für die Authentifizierung

```

1 public static Result authenticate() {
2   Form<Login> loginForm = Form.form(Login.class).bindFromRequest();
3   if(loginForm.hasErrors()) {
4     return unauthorized(login.render());
5   }
6   else {
7     String url = session().get("url");
8     session().clear();
9     session("username",loginForm.get().username);
10    if(url == null || url.isEmpty()){
11      return redirect(routes.Application.index());
12    }
13    else{
14      return redirect(url);
15    }
16  }
17 }
```

In [Zeile 2](#) definieren wir einen Form-Helper mit dem Typparameter `Login`. Diesem Helper weisen wir mit dem Befehl `bindFromRequest()` die Formulareinträge des POST-Requests der Login-Maske zu. Nun kann mit der Methode `hasErrors()` geprüft werden, ob ungültige Werte übergeben wurden oder ob sich der Benutzer erfolgreich authentifiziert hat. Der Form-Helper von Play erkennt dies automatisch, da wir vorher die beiden Instanzvariablen und eine `validate()`-Methode erstellt haben.

Konnte sich der Benutzer nicht authentifizieren oder ist irgendein anderer Fehler bei der Einreichung des Formulars aufgetreten, fangen wir dies ab und geben mit `return unauthorized()` einen entsprechenden Statuscode zurück. Der Benutzer wird zu der

Login-Seite zurückgeschickt. Bei einer erfolgreichen Anmeldung löschen wir die vorherige Session und erzeugen eine neue, die den Benutzernamen als Cookie-Parameter beinhaltet. Danach leiten wir den Benutzer zu seiner gewünschten Seite weiter.

Für diese Klasse müssen wir noch eine HTTP-Methode und eine passende URL in der Datei *routes* eintragen:

Listing 7.27 Route-Pfad zur Authentifizierung

```
1 POST      /login           controllers.Credential.authenticate()
```

Als Nächstes erstellen wir die Klasse Secured, die überprüfen soll, ob der Benutzer über eine gültige Session verfügt (siehe [Listing 7.28](#)).

Listing 7.28 Überprüfen einer Session

```
1 public class Secured extends Security.Authenticator {
2     @Override
3     public String getUsername(HttpContext ctx) {
4         return ctx.session().get("username");
5     }
6     @Override
7     public Result onUnauthorized(HttpContext ctx) {
8         ctx.session().put("url", "GET".equals(ctx.request().method())
9             ? ctx.request().uri() : Play.application() + "/");
10    return redirect(routes.Credential.login());
11 }
```

Secured erbt von der Klasse `Security.Authenticator` und überprüft bei jedem Request, ob der Benutzer eine gültige Session vorweisen kann. Falls nicht, wird er zur Login-Seite weitergeleitet.

Nun können wir mit der Annotation `@Security.Authenticated()` bestimmen, welche Controller-Klasse mit einer Zugriffskontrolle geschützt werden soll (siehe [Listing 7.29](#)).

Listing 7.29 Setzen der Security-Annotation

```
1 @Security.Authenticated(Secured.class)
2 public class Application extends Controller{
3     ...
4 }
```

Die bearbeiteten Teile des Quelltextes unseres Echo-Servers aus [Abschnitt 6.4.9](#) sehen anschließend folgendermaßen aus (siehe [Listing 7.30](#)):

Listing 7.30 Durch Zugriffskontrolle geschützte Controller-Klasse

```
1 @Security.Authenticated(Secured.class)
2 public class Application extends Controller {
3     ...
4     public static WebSocket<String> ws() {
5         return new WebSocket<String>()
```

```
6      @Override
7      public void onReady(play.mvc.WebSocket.In<String> in,
8          final play.mvc.WebSocket.Out<String> out) {
9          in.onMessage(new Callback<String>() {
10             @Override
11             public void invoke(String event) {
12                 out.write(event);
13             }
14         });
15         in.onClose(new Callback0() {
16             public void invoke() throws Throwable {
17                 System.out.println("WebSocket geschlossen");
18             }
19         });
20     };
21 }
22 }
```

Dieser Controller ist nun durch eine Zugriffskontrolle geschützt. Ausschließlich angemeldete Benutzer sollten jetzt auf diesen Controller zugreifen können. Für einen Test öffnen Sie die Adresse

<http://localhost:9000/echo>

in einem Browser. Sie werden zur Login-Seite geleitet, um sich zu authentifizieren. Versuchen Sie, eine WebSocket-Verbindung über die JavaScript-Konsole aufzubauen.

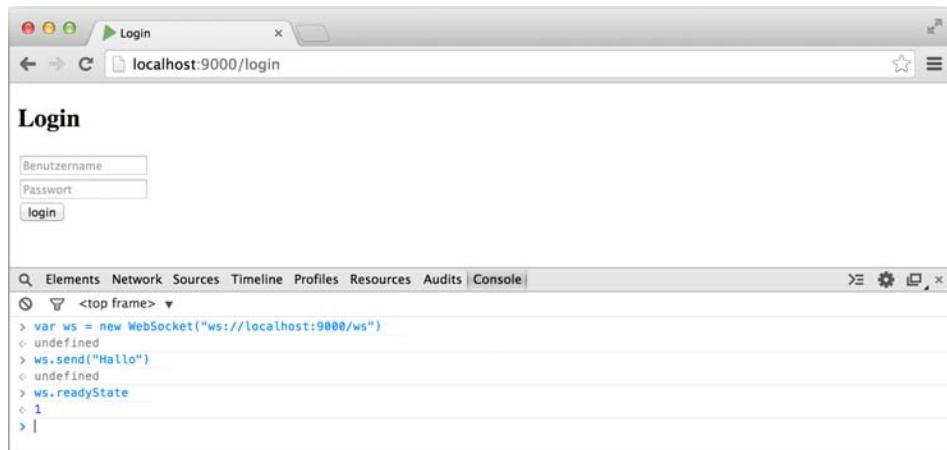


Bild 7.17 Erfolgreicher Handshake trotz Zugriffskontrolle

Wie Sie in Bild 7.17 am readyState erkennen können, ist es uns trotzdem möglich, eine Verbindung zum WebSocket-Endpunkt aufzubauen, obwohl wir den gesamten Controller mit der Security-Annotation geschützt haben (siehe Listing 7.30). Die Klasse Secured in Listing 7.28 schützt demnach keine HTTP-Upgrade-Requests auf das WebSocket-Protokoll, sondern nur gewöhnliche HTTP-Anfragen.

Um die WebSocket-Verbindung trotzdem verweigern zu können, müssen Sie die Methode `WebSocket.reject` verwenden. Dieser müssen Sie ein Result-Objekt übergeben. In diesem Fall soll der Server mitteilen, dass die Anfrage abgelehnt wurde. Genau dafür ist der Statuscode 403 (Forbidden) gedacht, den wir mit der gleichnamigen Methode `forbidden()` zurückgeben (siehe Listing 7.31).

Listing 7.31 Zugriffskontrolle für WebSocket-Endpunkte im Play Framework

```

1 public static WebSocket<String> ws() {
2     if(session().get("username")==null){
3         return WebSocket.reject(forbidden());
4     }
5     return new WebSocket<String>() {
6         ...
7     };
8 }
```

Schauen Sie sich das Ergebnis wieder im Browser an. Jetzt wird der Aufbau der WebSocket-Verbindung verweigert, falls Sie sich noch nicht eingeloggt haben (siehe Bild 7.18).

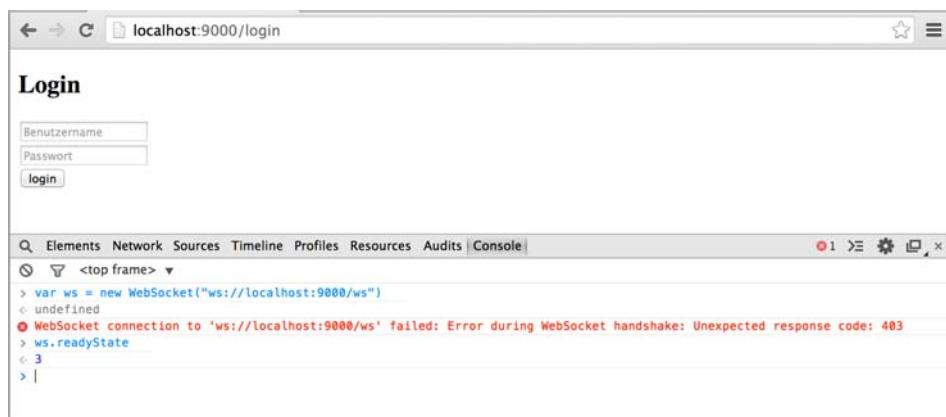


Bild 7.18 WebSocket-Verbindung wird geschlossen wegen ungültiger Session

7.2.4 Firewalls und Proxys

WebSockets verwenden üblicherweise die Standardports des HTTP-Protokolls (80 und 443), die im Allgemeinen als firewall-freundlich gelten. Dies bedeutet, dass die Konfigurationen der Paketfilter, die den Datenverkehr u. a. nach Portnummern filtern, nicht verändert werden müssen.

Mit Proxy-Servern verhält es sich leider etwas schwieriger. Diese fungieren als Vermittler zwischen Clients und Servern. Typische Funktionen umfassen z. B. das Cachen häufig angefragter Webressourcen, das Anonymisieren von Clients und das gezielte Filtern bestimmter Inhalte. Proxy-Server werden zudem zur Überwachung von Datenverbindungen eingesetzt, um z. B. bei Bedarf Verbindungen zu schließen, die schon längere Zeit inaktiv sind.

All dies sollten Sie im Hinterkopf behalten, da Sie in der Regel nicht wissen, über welche Wege bzw. über welche Proxys sich ein Client zu Ihrem Server verbindet. Aus diesem Grund wollen wir uns in diesem Kapitel näher mit diesem Thema im Zusammenhang mit WebSockets befassen und Ihnen Tipps geben, wie Sie potenziellen Problemen mit Proxy-Servern begegnen können.

Zunächst muss man die Sichtbarkeit von Proxy-Servern etwas differenzierter betrachten. *Sichtbare Proxys* stellen Sie explizit über den Browser oder über die Netzwerkeinstellungen des Betriebssystems ein. Das haben wir zum Beispiel in [Abschnitt 4.9.2](#) gemacht, wo wir den Proxy-Server von Fiddler im Browser eingerichtet haben, um den Datenverkehr analysieren zu können. *Transparente Proxys* werden hingegen nicht explizit eingestellt. Der Client bzw. Browser bekommt sozusagen nichts davon mit, dass z. B. eine angeforderte Webseite von einem Proxy-Server geliefert wird. Damit ist dieser unsichtbar für den Client.

Die 2011 durch den Artikel „*Talking to Yourself for Fun and Profit*“ [HCB⁺11] aufgedeckten Schwachstellen des WebSocket-Protokolls, bezüglich *IP-Hijacking-* und *Caching-Poisoning*-Angriffen gegen transparente Proxy-Server, führten zu einer Maskierung der WebSocket-Frames, die vom Client zum Server geschickt werden (siehe [Abschnitt 4.5](#)). Absicht dieser Maskierung der Nutzdaten ist nicht die Vertraulichkeit, also diese für Dritte unleserlich zu machen, denn der Demaskierungsschlüssel wird in der Nachricht mitgeschickt, sondern diese für Proxys unleserlich zu machen. Angreifer sind somit nicht mehr imstande, den Payload so zu manipulieren, dass damit Angriffe gegen Proxy-Server durchgeführt werden können.

Diese Maskierung macht den Payload jedoch auch für andere Filtersysteme unbrauchbar. So schränkt dies ebenfalls Firewalls, *Intrusion Detection Systems* (IDS) und *Intrusion Prevention Systems* (IPS), welche auf der „guten Seite“ stehen und versuchen, böswillige Muster innerhalb der Nutzdaten ausfindig zu machen, ein. Für solche Filtertools wird der Datenverkehr unleserlich, weshalb sie diesen ohne einzugreifen passieren lassen. Ebenso fehlen dadurch Hilfsinformationen innerhalb des Datenverkehrs, was eine Analyse verkompliziert [SST12]. Ein möglicher Lösungsansatz, um Firewalls und Router in die Lage zu versetzen, passende Sicherheitsrichtlinien anzuwenden, könnte sein, den optionalen Headereintrag Sec-WebSocket-Protocol des WebSocket-Protokolls zu nutzen, um anzugeben, welches Protokoll auf Applikationsebene innerhalb des Payloads des WebSocket-Frames genutzt wird [Zim12].

Bei Proxy-Servern im Web handelt es sich in der Regel um HTTP-Proxys. Viele davon können nicht mit dem WebSocket-Protokoll umgehen, wodurch Probleme entstehen können. Wenn Sie einen Browser explizit auf einen Proxy-Server einstellen, können Sie problemlos eine WebSocket-Verbindung aufbauen, da der Browser automatisch einen Tunnel mit der Methode `HTTP-CONNECT` aufbaut. Bei transparenten Proxys können genau an dieser Stelle Komplikationen auftreten, da Sie hier keine Möglichkeit haben einzugreifen. Dadurch kann kein Tunnelaufbau mit der Methode `HTTP-CONNECT` forciert werden. Wenn ein HTTP-Proxy-Server in einem solchen Fall das WebSocket-Protokoll nicht versteht, kommt es im Endeffekt zu keiner WebSocket-Verbindung.

Im unangenehmsten Fall ist ein Proxy-Server sogar so konfiguriert, dass er beim Weiterleiten eines Upgrade-Requests bestimmte Header-Felder, wie z. B. den Connection-Header, wegschneidet. Dadurch kann ebenfalls keine WebSocket-Verbindung aufgebaut werden [Lub10]. In dieser Situation sind Sie auf transparente Proxy-Server angewiesen, die das WebSocket-Protokoll unterstützen und Anfragen entsprechend weiterleiten können. Falls

Sie also in Ihrem Firmen- oder Heimnetz einen solchen Server verwenden möchten, achten Sie auf diese Eigenschaft. Der Web- und Proxy-Server nginx z. B. unterstützt WebSockets seit der Version 1.4.0 [Ngi14] und auch der Apache-Webserver enthält ab der Version 2.4.5 ein Modul, um WebSockets zu tunneln.

In der [Tabelle 7.2](#) haben wir für Sie verfügbare WebSocket-Proxys aufgelistet.

Tabelle 7.2 WebSocket-Proxys

Name	Homepage
Hipache	https://github.com/hipache/hipache/
nginx	http://nginx.org/
node-http-proxy (bis Node v0.8.6)	https://github.com/nodejitsu/node-http-proxy/
EventMachine WebSocket-Proxy	https://github.com/mcoleyer/em-websocket-proxy/
HAProxy	http://www.haproxy.org/
Apache Module mod_proxy_wstunnel	https://httpd.apache.org/docs/2.4/mod/mod_proxy_wstunnel.html
Sencha Ext.ux.data.proxy.WebSocket	https://market.sencha.com/extensions/ext-ux-data-proxy-websocket/

Eine Alternative, um WebSockets durch transparente Proxys zu schleusen, ist die Nutzung einer TLS-verschlüsselten Verbindung. Wird eine verschlüsselte WebSocket-Verbindung mit dem URL-Schema *wss* etabliert, wird die Verbindung grundsätzlich immer mit TLS getunnelt, wodurch eine viel größere Chance besteht, Ihre WebSockets durch alle Proxys durchzubringen. Die Verwendung von verschlüsselten WebSocket-Verbindungen kann, wie schon in [Abschnitt 7.2.2](#) beschrieben, Vorteile in Mobilfunknetzen mit sich bringen. Generell raten wir Ihnen, Verbindungen immer mithilfe von TLS zu verschlüsseln. Das gilt natürlich vor allem, wenn Sie sensible Daten übertragen wollen und Sie verhindern sollten, dass ein Angreifer Ihren Datenverkehr im Klartext mitschneiden kann.

Wenn Sie TLS nutzen möchten, müssen Sie zu Beginn eine kleine Hürde überwinden. Sie benötigen ein Zertifikat, das Sie sich relativ einfach und schnell mit XCA oder OpenSSL erstellen können. Wenn dieses allerdings nicht von einer Zertifizierungsstelle (CA) (*engl. Certification Authority*) unterschrieben bzw. signiert wird, erscheint beim Aufruf der URL eine Warnmeldung im Browser (siehe [Bild 7.19](#)).

Falls Sie also verschlüsselte WebSocket-Verbindungen in einer Produktivumgebung einsetzen möchten, empfehlen wir Ihnen, Ihr Zertifikat von einer CA signieren zu lassen oder sich direkt ein gültiges Zertifikat von einer solchen Stelle zu beschaffen. Damit verhindern Sie Verwirrungen und Misstrauen, da ein Warnhinweis natürlich nicht gerade einen vertrauenswürdigen Eindruck erweckt, besonders wenn es um eine Webanwendung mit vielen Benutzern geht. Ist Ihre Applikation hingegen nur für wenige Benutzer gedacht oder sie kommt nur in Ihrem Firmennetz zum Einsatz, brauchen Sie unserer Meinung nach nicht unbedingt ein gültiges Zertifikat. Hier kann es ausreichen, wenn Sie den Nutzerkreis auf die Zertifikatswarnung hinweisen.

Ein anderes Problem kann entstehen, wenn aufgebaute WebSocket-Verbindungen lange ungenutzt bleiben, also keine Daten darüber fließen. Ein Proxy-Server oder ein Router kann solche über einen bestimmten Zeitraum inaktive Verbindungen schließen. Wenn Sie dieses Verhalten verhindern möchten, empfehlen wir Ihnen, in regelmäßigen Abständen

Dieser Verbindung wird nicht vertraut

Sie haben Firefox angewiesen, eine gesicherte Verbindung zu **localhost** aufzubauen, es kann aber nicht überprüft werden, ob die Verbindung sicher ist.

Wenn Sie normalerweise eine gesicherte Verbindung aufzubauen, weist sich die Website mit einer vertrauenswürdigen Identifikation aus, um zu garantieren, dass Sie die richtige Website besuchen. Die Identifikation dieser Website dagegen kann nicht bestätigt werden.

Was sollte ich tun?

Falls Sie für gewöhnlich keine Probleme mit dieser Website haben, könnte dieser Fehler bedeuten, dass jemand die Website fälscht. Sie sollten in dem Fall nicht fortfahren.

Diese Seite verlassen

► **Technische Details**

► **Ich kenne das Risiko**

Bild 7.19 Warnmeldung wegen eines Zertifikats, das nicht von einer CA signiert wurde

Pings und Pongs zu versenden, um die Verbindung „am Leben zu halten“. Allerdings können Pings und Pongs momentan leider nicht mit der JavaScript-WebSocket-API von einem Client zu einem Server versendet werden. Diese Aufgabe sollte der Browser für Sie übernehmen. Mehr zu dem Thema Pings und Pongs in der Praxis erfahren Sie in [Abschnitt 7.3.1](#).

7.2.5 Mögliche Gefährdungen

Wenn Sie Anwendungen für das Web entwickeln, insbesondere wenn diese aus verteilten Komponenten zusammengesetzt sind, sollten Sie Sicherheitsaspekte niemals außer Acht lassen, denn bereits gefundene Angriffsvektoren sind vielseitig und der Einfallsreichtum findiger Angreifer ist nicht zu unterschätzen. Deshalb möchten wir Sie in diesem Unterkapitel für mögliche Gefährdungen im Bezug auf die noch junge WebSocket-Technologie sensibilisieren. Wir stellen Ihnen bekannte Schwachstellen bzw. Angriffsszenarien vor und, falls bekannt, auch mögliche Gegenmaßnahmen, damit Sie bei Ihren Entwicklungen an empfindlichen Stellen eine Alarmglocke klingen hören und entsprechend reagieren können. [Bild 7.20](#) ermöglicht Ihnen zunächst einen kurzen Überblick über die in den nächsten Abschnitten besprochenen Themen. Wir möchten unsere Ausführungen explizit nicht als Anleitung zum wissentlichen Brechen von WebSocket-Anwendungen verstanden wissen.

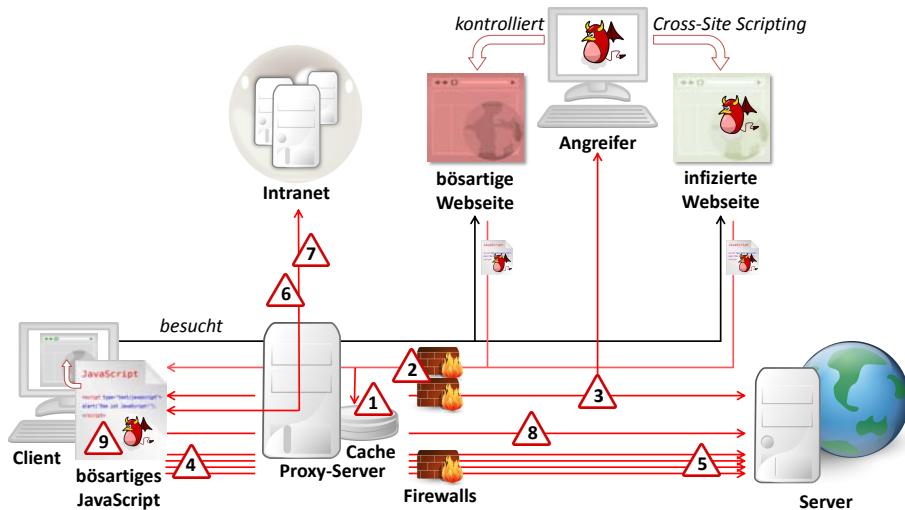


Bild 7.20 Ein Überblick über mögliche Gefährdungen durch Angriffe mit WebSockets

Auf Bedenken im Zusammenhang mit der Same Origin Policy sind wir bereits in [Abschnitt 7.2.1](#) eingegangen. Welche Probleme durch das Unterbinden von *IP-Hijacking*- und *Caching-Poisoning*-Angriffen (1) im Bezug auf Firewalls (2) entstanden sind, haben wir bereits in [Abschnitt 7.2.4](#) beschrieben. Wie Sie sehen, fehlt allerdings noch einiges. Wir möchten mit Ihnen der Reihe nach die Angriffe *Man in the Middle* (3), *Denial of Service* (4, 5), *Remote Shell* (6), *Port Scanner* (7) und ein Angriffsszenario gegen authentifizierte Benutzer (8) durchgehen. Zudem möchten wir diskutieren, wie grundsätzlich verhindert werden könnte, dass ein bösertiger Code in den Browser eines Benutzers gelangt (9).

Man-in-the-Middle-Angriffe

2012 veröffentlichte BlackHat im Rahmen eines Briefings Ergebnisse einer Untersuchung zur Verbreitung von WebSockets auf Seiten [SST12]. Die Zahlen wurden mithilfe eines Crawlers ermittelt, der auf den 600.000 am häufigsten aufgerufenen Internetseiten (*Alexa Top 600.000*⁴) nach WebSockets suchte. Sie fanden dadurch heraus, dass nur 0,15% der Seiten die neue Technologie auf ihrer Landing-Page benutzen. Ganze 95% der gefundenen WebSocket-Verbindungen hatten ihren Ursprung in einem Chat-Programm für Kundensupport, das von einem einzelnen Anbieter vertrieben wird und verschlüsselte Verbindungen einsetzt. Von den verbleibenden 5%, von denen angenommen wird, dass die WebSocket-Applikationen von verschiedenen Entwicklern stammten, verschlüsselten weniger als 1% die Verbindungen. Im Umkehrschluss bedeutet dies, dass die versendeten Nachrichten auf der Transportschicht von 99% der Entwickler der gefundenen Applikationen ungesichert gelassen wurden, was Tür und Tor für *Man-in-the-Middle-Angriffe* (MITM-Angriffe) öffnet.

⁴ <http://www.alexa.com/topsites/>

Bei diesem Angriff lockt der Angreifer sein Opfer auf eine durch *Cross-Site-Scripting* (XSS) infizierte oder selbst veröffentlichte böswillige Internetseite und initiiert oder übernimmt dadurch die WebSocket-Verbindung. Dies funktioniert auf dieselbe Weise wie bei HTTP (siehe Bild 7.21).

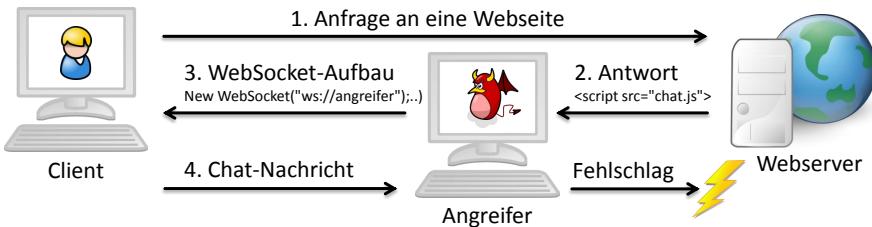


Bild 7.21 Man-in-the-Middle-Angriff

1. Der Client ruft eine Zielseite auf, z. B. um mit jemandem zu chatten.
2. Der angefragte Webserver antwortet mit einem Skript (chat.js), das vom Angreifer abgefangen wird.
3. Der Client erhält unwissentlich ein manipuliertes Skript, das eine WebSocket-Verbindung zum Server des Angreifers etabliert.
4. Unter der falschen Annahme, er wäre mit dem angefragten Webserver verbunden, kommuniziert der Client nun in Wahrheit mit dem Server des Angreifers.

Die Arbeitsgruppe um das WebSocket-Protokoll versuchte dieses Problem zumindest etwas einzudämmen, indem sie festlegten, dass aus einer gesicherten HTTPS-Verbindung heraus keine ungesicherte WS-Verbindung aufgebaut werden darf. Wie schon in Abschnitt 7.2.2 erwähnt, hält sich der Apple-Browser Safari nicht an diese Vorgabe.

Um diese Art der Angriffe zu vermeiden, ist es notwendig, eine verschlüsselte WebSocket-Verbindung, wie in Abschnitt 7.2.2 beschrieben, zu benutzen. Auch Anwender auf der Clientseite sollten sich dieses Problems bewusst sein und darauf achten, dass eine verschlüsselte Verbindung vorliegt, bevor sie sensible Daten übermitteln. Zusätzlich sollten Sie im Hinterkopf behalten, dass TLS nur den Transportkanal sichert und keine Sicherheit auf der Nachrichtenebene gewährleistet.

Denial-of-Service-Angriffe

Ein Denial-of-Service-Angriff hat zum Ziel, einen Server oder Client gezielt außer Gefecht zu setzen. Auch hierfür können WebSockets missbraucht werden.

In Abschnitt 4.6 haben wir Ihnen gezeigt, dass eine WebSocket-Nachricht aus mehreren Frames bestehen kann. In der Spezifikation wurde die Anzahl dieser zusammengehörigen Frames nicht limitiert, weshalb eine Nachricht bei einer fragmentierten Übertragung theoretisch aus einer unbegrenzten Anzahl von Frames bestehen kann. In einem einzelnen WebSocket-Frame können dabei durch den `Extended payload length` Header theoretisch $2^{64} - 1$ Bytes an Daten übertragen werden. Das ist eine beachtliche Größe, wenn man bedenkt, dass man sich bereits bei 2^{40} im Bereich der Terabytes bewegt. Dies macht es demnach möglich, eine beliebig große WebSocket-Nachricht zu senden.

Webbrowser begrenzen das Limit simultaner HTTP-Requests an dieselbe Domain gewöhnlich auf vier bis sechs Verbindungen [SST12]. Die Zahlen für gleichzeitige WebSocket-Verbindungen heben sich im Vergleich deutlich ab. **Tabelle 7.3** zeigt die Ergebnisse eines Experiments, das die maximal mögliche Anzahl simultaner WebSocket-Verbindungen verschiedener Browser mit derselben Domain aufzeigt.

Tabelle 7.3 Maximale WebSocket-Verbindungen bei Browsern [SST12]

Browser	Maximale WebSocket-Verbindungen
Chromium	924
Chrome	3237
Safari	2970
Firefox	200
Opera	900

Daraus geht hervor, dass Safari und Chrome die Anzahl der Verbindungen nicht begrenzen. Schafft es ein Angreifer, seinen bösartigen JavaScript-Code im Browser seines Opfers zu starten, könnte er versuchen, Tausende WebSocket-Verbindungen aufzubauen. Dem Browser würde der ihm zur Verfügung gestellte Speicher ausgehen und sich im Endeffekt aufhängen. Aber auch das Begrenzen der erlaubten simultanen WebSockets, wie es z. B. der Firefox macht, birgt seine Tücken. Wenn durch einen Angriff 200 WebSocket-Verbindungen etabliert werden, wäre damit das Limit erreicht. Keine weitere vom Benutzer geöffnete Applikation könnte dadurch eine WebSocket-Verbindung aufbauen.

Diese Art der Angriffe stellt jedoch nicht nur ein Risiko für Clients, sondern auch für Server dar. Ein Angreifer könnten mit dem schädlichen Code innerhalb des befallenen Browsers ebenfalls versuchen, eine möglichst große Anzahl von WebSocket-Verbindungen zu einem Server herzustellen. Das Ziel dabei ist, den jeweiligen Server zu überlasten und somit unerreichbar für andere Clients zu machen.

Durch die Kompromittierung Tausender PCs ist ein Denial-of-Service-Angriff in solchem Ausmaß auch über das HTTP möglich. WebSockets reduzieren jedoch den Aufwand erheblich. Damit lässt sich die Hypothese aufstellen, dass es einfacher ist, DoS-Angriffe mit WebSockets durchzuführen als mit HTTP. In [Bild 7.22](#) betrachten wir das Szenario, in dem ein Client eine Webseite besucht, auf der Schadcode platziert wurde.

- Der Client klickt unachtsam auf einen Link und wird zu der vom Angreifer kontrollierten Internetseite geleitet.
- Der Browser führt ein platziertes bösartiges JavaScript aus.
- Ohne Wissen des Clients öffnet dieses Skript eine große Anzahl von WebSocket-Verbindungen und greift damit einen bestimmten Server an.

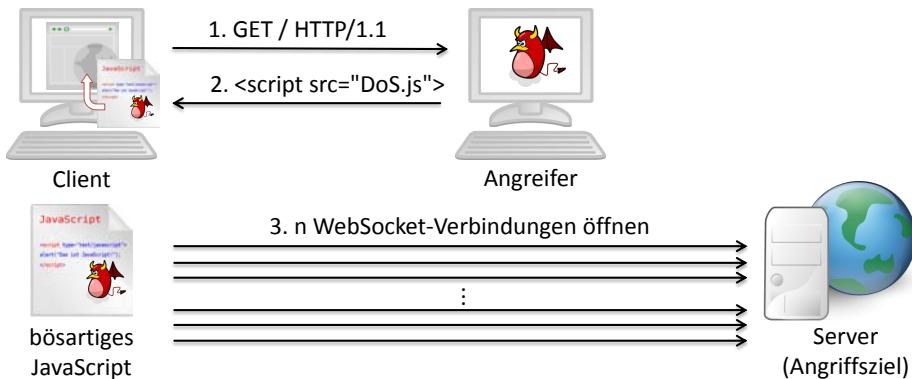


Bild 7.22 Denial-of-Service-Angriff

Dieser Angriff wird mit jedem weiteren befallenen Client effektiver. Jemand könnte deshalb zunächst versuchen, eine möglichst große Anzahl von Browsern zu kompromittieren, um sich ein WebSocket-basiertes Botnetz aufzustellen (siehe Abbildung 7.23). Dazu müsste der Angreifer es entweder schaffen, z. B. mithilfe von Phishing-Mails, viele Anwender auf seine Webseite zu locken oder eine beliebte Internetseite durch Cross-Site-Scripting zu infizieren.

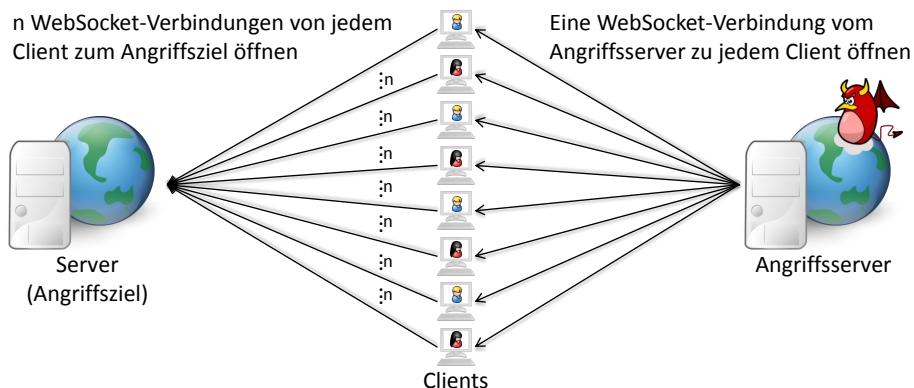


Bild 7.23 Ein WebSocket-Botnetz

Für die betroffenen Serverbetreiber ist es sehr schwierig, die tatsächliche Herkunft des Angriffs herauszubekommen, da die Clients als Bauernopfer vorgeschnickt werden. Zur Abwehr von DoS-Angriffen kann der in Abschnitt 7.2.1 beschriebene Origin-Header dazu verwendet werden, unerwünschte Verbindungsanfragen herauszufiltern [WSM13]. In letzter Instanz könnten auffällige Quellen in einer schwarzen Liste geführt werden. Trotzdem müssen Sie bedenken, dass Header manipulierbar sind. Browseranbieter wären hier schon eher in der Lage, eine größere Hürde aufzubauen, indem sie die erlaubte Anzahl simultaner WebSocket-Verbindungen reduzieren. Auch das Begrenzen der Datenmenge einer einzelnen WebSocket-Nachricht könnte die Situation etwas entschärfen.

Remote-Shell-Angriffe

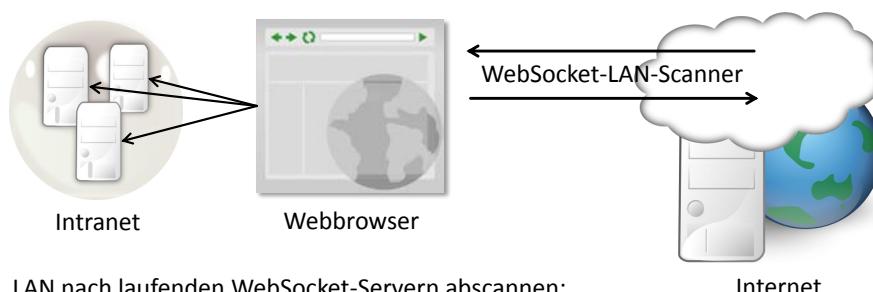
Über den bidirektionalen Kanal eines WebSockets kann eine Echtzeit-Remote-Shell aufgebaut werden. Damit kann ein Server Kommandos oder Programme in einem Browser ausführen. Auch in diesem Szenario muss es dem Angreifer gelingen, sein JavaScript-Programm in dem Browser eines Opfers auszuführen. Dann ist er allerdings in der Lage, solange ein WebSocket-Kanal aufgebaut ist, den Browser eines Clients innerhalb der Funktionalität von JavaScript zu kontrollieren, und dies bedeutet im Klartext, er kann sehr viel Schaden anrichten. Besonders kritisch ist, dass er damit Zugriff auf alle Daten innerhalb der laufenden Domain hat, wie zum Beispiel Inhalte eines Intranets. Auch die nachträgliche Installation von Malware im Browser oder eine Weiterleitung auf beliebige Webseiten, um dort beispielsweise Spammnachrichten zu verbreiten, sind im Rahmen des Machbaren [Sch11].

Die Remote Shell ist auch ein bekanntes Problem anderer Protokolle, die für die Kommunikation im Internet genutzt werden, und kann nicht so ohne Weiteres durch serverseitige Implementierungen verhindert werden [Sch11]. Die dynamische Natur von JavaScript macht eine automatisierte Analyse und Erkennung von infiziertem Code in der Praxis enorm schwierig [Erk12]. Bisher kann dieser Angriff nur durch das manuelle Deaktivieren von WebSockets innerhalb des Browsers vollständig unterbunden werden.

Network Scan

Ein vorstellbares neues Angriffsszenario zum Scannen von privaten Netzen könnte wie folgt realisiert werden. Der Browser im Rechner des Clients wird über einen geöffneten WebSocket zu einem Server umfunktioniert, um Anfragen aus dem Netz zu bedienen. Mit einer solchen technischen Kommunikationsgrundlage könnten in Zukunft Trojaner geschrieben werden, die den Browser infizieren und über einen WebSocket-Kanal Befehle von einer Kommandozentrale entgegennehmen.

Aus diesem Grund erscheint uns ein Szenario wahrscheinlich, in dem WebSockets als Eingangspforte für Netzwerkanalysen von LANs verwendet werden (siehe Bild 7.24).



LAN nach laufenden WebSocket-Servern abscannen:

```
ws[1] = new WebSocket("192.168.1.1");
ws[2] = new WebSocket("192.168.1.2");
...

```

Bild 7.24 Scan eines LAN nach WebSocket-Endpunkten

Als JavaScript-Programm gelangt der in Bild 7.24 als WebSocket-LAN-Scanner bezeichnete Virus per HTTP-Response in den Webbrower.

Dort ermittelt das Skript zunächst den IP-Adressraum des LAN (z. B. 192.168.1.0/255) und beginnt anschließend, die verfügbaren Adressen nach laufenden WebSocket-Servern zu scannen. Ob solche Informationen relevant genug für einen Angriff sein werden, wird die Zukunft zeigen.

Auf dieser Basis kann aufbauend mit dem nachfolgend beschriebenen Angriff die Analyse auf einen Zielhost konzentriert werden.

Port Scanner

Durch einen Cross-Origin-Angriff kann ein Angreifer über einen geenterten Browser beliebig und unerkannt testen, ob Domains existieren, oder nicht. Die WebSocket-API wird dabei eingesetzt, um Requests an willkürliche URLs abzusetzen und anschließend die Antwortzeiten auszuwerten und Rückschlüsse auf die Existenz der Domains zu ziehen. Dieses Verfahren ermöglicht es ebenfalls, nach verfügbaren Ports zu scannen, deren Dienste als mögliches Einfallstor dienen könnten [Sch11].

Eine Anfrage an eine URL erhält eine Antwort nach unterschiedlich langer Zeit, abhängig davon, ob die Domain existiert, nicht existiert, ein Statuscode zurückgesendet oder der Zugriff durch das Verifizieren des Acces-Conroll-Allow-Origin-Headers gefiltert wird. Basierend auf den gemessenen Antwortzeiten ist es möglich, eine Aussage darüber zu treffen, ob ein Port offen ist, geschlossen ist oder gefiltert wurde [Kup10]. Tabelle 7.4 zeigt die Zeitunterschiede.

Tabelle 7.4 Auf Port-Status basierende Antwortzeiten [Kup10]

Port-Status	Antwortzeit
offen	< 100 ms
geschlossen	~ 1000 ms
gefiltert	> 30000 ms

Ein solcher Port-Scanner könnte über das Internet dazu missbraucht werden, ein Firmen-Intranet zu scannen. Wird damit ein offener Port 80 gefunden, könnte das ein Startpunkt für weitere Angriffe sein. Schafft der Angreifer es, z. B. erfolgreich eine Remote Shell zum Brower eines Firmenmitarbeiters aufzubauen, kann er unter Umständen auf diesem Weg die Firewall des Firmennetzes umgehen und auf interne Inhalte zugreifen [Sha12].

Vor diesem Szenario kann man sich gegenwärtig ebenfalls nur mit Gewissheit durch das manuelle Deaktivieren der WebSocket-Unterstützung im Webbrower schützen. Ein Lösungsvorschlag, um die Analyse der Antwortzeiten unbrauchbar zu machen, ist das künstliche Angleichen der Verarbeitungsdauer. Durch eine kurze Verzögerung wäre zumindest eine Differenzierung offener oder geschlossener Ports nicht mehr möglich. Der Preis der höheren Sicherheit wäre in diesem Fall eine geringere Performance.

Entführung der Client-Authentifizierung

In Abschnitt 7.2.3 haben wir Ihnen unterschiedliche Authentifizierungsmechanismen vorgestellt. Christian Schneider veröffentlichte 2013 auf seinem „Web Application Security Blog“ einen Man-in-the-Middle-Angriff, der sich gegen authentifizierte WebSocket-Applikationen richtet [Sch13]. In diesem Szenario stiehlt der Angreifer die Authentifizierungsinformationen eines Opfers, um damit eine neue WebSocket-Verbindung zum betreffenden Service aufzubauen. Dieses Angriffszenario ist einem Cross-Site-Request-Forgery-(CSRF-)Angriff sehr ähnlich, unterscheidet sich jedoch darin, dass der Angreifer eine zweite Verbindung aufbaut, welche ihm vollen Lese- und Schreibzugriff gewährt. In diesem Zusammenhang taufte Christian Schneider den Angriff *Cross-Site WebSocket Hijacking* (CSWSH).

Wir gehen in diesem Szenario davon aus, dass die Entwickler den sensiblen Bereich einer Applikation entweder mit der HTTP-Authentifizierung oder durch ein Verfahren, das Cookies nutzt, schützen. Als Beispielanwendung nehmen wir ein WebSocket-basiertes Aktien-Portfolio an, um immer auf dem aktuellen Stand der Handelsinformationen zu sein. Das Szenario könnte wie in Bild 7.25 ablaufen.



Bild 7.25 Cross-Site WebSocket Hijacking

- Der Nutzer authentifiziert sich gegenüber der Aktien-App.
- Der Client initiiert die WebSocket-Verbindung mit einem HTTP- oder HTTPS GET-Request des Opening-Handshakes an den Endpunkt des Aktien-Portfolio-Management-Systems. Dabei wird auch der Cookie-Header übertragen.
- Der Server antwortet mit dem Statuscode 101, um den Handshake abzuschließen und zum WebSocket-Protokoll zu wechseln.
- Während der Benutzer in der Applikation eingeloggt ist, schafft es der Angreifer, ihn auf eine böswillige Webseite zu locken.
- Ein Skript dieser Webseite greift ebenfalls auf die Aktien-Applikation zu und etabliert mit den Authentifizierungsinformationen des Clients (Cookie oder HTTP-Authentifizierung) eine zweite WebSocket-Verbindung.

Dies ist möglich, weil WebSockets nicht durch die Same Origin Policy eingeschränkt werden. Die Authentifizierungsinformationen des Clients werden beim Opening-Handshake website-übergreifend mitgesendet. Damit wäre der Angreifer imstande, das Portfolio des Opfers zu lesen und zu manipulieren.

Um CSWSH-Angriffen vorzubeugen, sollte der Server den Origin-Header beim Handshake validieren und website-übergreifend Anfragen ablehnen. Zusätzliche Sicherheit würde ein auf der Serverseite generierter individueller Session-Token (CSRF-Token) bieten, der ebenfalls ein Teil der Validierung beim Opening-Handshake wäre.

Bösartige Services

Alle vorgestellten Angriffsszenarien setzen voraus, dass ein Angreifer imstande ist, seinen JavaScript-Code im Browser eines Opfers auszuführen. Ein normaler Benutzer kann nicht davor geschützt werden, eine böswillige oder eine vertraute, durch XSS infizierte Webseite zu besuchen, und hat noch weniger die Möglichkeit zu erkennen, ob eine WebSocket-Verbindung aufgebaut wurde. Eine Lösung konnte bisher weder durch clientseitige noch durch serverseitige Implementierungen erzielt werden. Ideen zur Lösung dieses Problems richten sich jedoch eher an die Anbieter der Browser als an Webentwickler.

Ein möglicher Ansatz zum Schutz vor ungewollten WebSocket-Verbindungen ist, den Benutzer grundsätzlich vorher um Erlaubnis zu fragen. Kritisch zu hinterfragen ist allerdings, wie zumutbar ständige Pop-up-Fenster wären, gesetzt den Fall, die Technologie verbreitet sich und wird oft eingesetzt. Usability-Studien sollten zudem klären, wie irritierend diese Dialoge eventuell auf normale Nutzer wirken und ob ihre Relevanz richtig eingeschätzt wird.

Ein etwas zurückhaltenderer Vorschlag ist, WebSocket-Verbindungen passiv im Browser anzuzeigen. Auch hier sollten Usability-Studien herausfinden, inwiefern sich dadurch die Sicherheit der Anwender erhöhen lässt. Ein Vorschlag von Jussi-Pekka Erkkilä ist, die Metadaten von Internetseiten und deren WebSocket-Verbindungen zu analysieren [Erk12]. In diesem Rahmen definierte er einige Metriken, um eine Webseite als legitim oder nicht einzustufen. Mechanismen im Webbrower könnten damit versuchen, Anomalien innerhalb einer WebSocket-Kommunikation, wie z. B. eine ungewöhnlich hohe Anzahl von Nachrichten oder Frames, ausfindig zu machen. Zusätzlich sollte die Anzahl von aufgebauten Verbindungen zu verdächtigen IP-Adressen, Ports, dem Localhost oder anderen internen Netzwerkadressen überwacht werden. Dies sollte geschehen mithilfe solcher Metriken und dem zusätzlichen Einpflegen eingestufter Origin-Hosts in Blacklists oder besser noch Whitelists, denn diese sind von Natur aus fehlerunanfälliger. Auf dieser Basis könnten Webbrower entscheiden, ob eine WebSocket-Verbindung aufgebaut oder unterbunden werden sollte. Um falsche Einstufungen zu reduzieren und damit die Filter zu optimieren, sollten die Metriken kontinuierlich verbessert werden.

Es wäre auch möglich, die verschiedenen Ansätze miteinander zu verknüpfen. Eine Webseite, die beispielsweise aufgrund der Metriken nicht in der Whitelist aufgeführt ist, gilt damit als verdächtig, weshalb der Benutzer an dieser Stelle vom Brower um Erlaubnis gefragt wird. Ein Symbol könnte zudem Hinweise im Bezug auf offene WebSocket-Verbindungen geben. Bisher finden die Vorschläge noch in keinem Brower Berücksichtigung.

7.2.6 Bewertung der Sicherheitslage

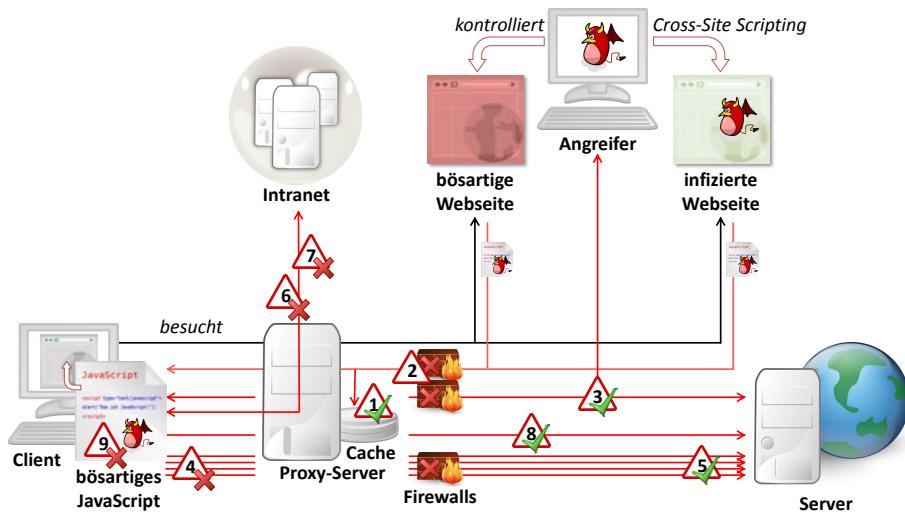


Bild 7.26 Offene und eingedämmte Sicherheitslücken

Als Gesamtfazit lässt sich sagen, dass WebSockets unter Berücksichtigung der derzeit bekannten Angriffsvektoren und deren Gegenmaßnahmen sicher eingesetzt werden können. Wirksame Schutzmaßnahmen gegen *Man-in-the-middle*-Angriffe (3), Angriffe gegen authentifizierte Benutzer(8), *Denial-of-Service*-Angriffe (5) und gegen den Port-Scanner (7) sind derzeit bekannt.

Probleme bestehen aktuell darin, bösartige Web-Services zu detektieren (9), *Remote-Shell*-Angriffe zu unterbinden (6), zusätzliche Sicherheitsmechanismen zu unterstützen (2) und im Einschränken von WebSocket-Verbindungen (3). Hier sind insbesondere die Entwickler von Browsern gefragt, Lösungen zu finden.

Bei der Entwicklung Ihrer Anwendungen sollten Sie sich immer die ganz grundsätzliche Frage stellen, wie hoch der durch einen Angriff entstehende Schaden sein kann. Mit der Antwort können Sie ein geeignetes Verhältnis zwischen Sicherheit und Performance Ihrer Anwendung finden. Sicherlich ist es fraglich, WebSockets zu nutzen, wenn durch eine evtl. notwendige großflächige Absicherung die Vorteile der Technologie verloren gehen. Wenn es um einen sehr sicherheitskritischen Einsatz geht, ist es unter Umständen sinnvoll zu warten, bis die Entwicklung und Erprobung von WebSockets weiter vorangeschritten ist.



Wir verfolgen die Sicherheitslage der WebSockets weiter, tragen aktuelle Erkenntnisse in diesem Kontext zusammen und stellen diese auf der Webseite des Buches für Sie bereit. Wir benötigen dazu Ihre Unterstützung in jeglicher Form und freuen uns besonders über nützliche Hinweise und Anregungen.

<http://websocket101.org/>

■ 7.3 Mit Haken und Ösen

In diesem Kapitel haben wir Widrigkeiten gesammelt, die uns bei der Entwicklung von WebSocket-Anwendungen aufgefallen sind. Es handelt sich hierbei also um einen bunten Blumenstrauß an Informationen rund um Besonderheiten der verschiedenen Implementierungen des Protokolls.



Wir möchten diese Sammlung auf der Internetseite zum Buch fortführen und aktuell halten.

<http://websocket101.org/>

Schauen Sie doch auch dort mal vorbei. Wir sind zudem dankbar, falls Sie Hinweise oder Anregungen für uns haben, die wir dort ergänzen können.

7.3.1 Pings und Pongs

Einige Router und Proxys schließen Verbindungen, die zu lange ungenutzt bestehen. Dies kann verhindert werden, indem zwischenzeitlich kurze Signalmeldungen ausgetauscht werden, die nur bewirken sollen, dass die Verbindung als aktiv eingestuft und folglich netzwerkseitig nicht geschlossen wird. Hierfür sieht das WebSocket-Protokoll die Ping- und Pong-Frames vor (siehe [Abschnitt 4.7](#)). Pings und Pongs können sowohl serverseitig als auch clientseitig abgesetzt werden. Zu jedem versendeten Ping wird mit einem Pong geantwortet. Es besteht auch die Möglichkeit, Pongs zu senden, ohne davor ein Ping empfangen zu haben. Dabei handelt es sich um eine ressourcenschonendere Art und Weise, dem Kommunikationspartner zu signalisieren, dass der Absender noch da ist.

Es ist nicht möglich, über die W3C WebSocket-API vom Browser aus Ping- und Pong-Frames zu senden. Eine entsprechende Schnittstelle ist willentlich weggelassen worden, da die Aufgabe des Sendens von Pings und Pongs dem Browser direkt zugesprochen wurde. Dies ist allerdings noch nicht von allen Browser-Entwicklern entsprechend umgesetzt worden. Folglich können Sie sich zurzeit nicht darauf verlassen, dass Webbrowser-Clients Pings oder Pongs zur Aufrechterhaltung der Verbindung absetzen. Aus diesem Grund ist es notwendig, dies in Ihren Implementierungen von WebSocket-Servern zu berücksichtigen, wie der Pseudocode in [Listing 7.32](#) verdeutlichen soll.

Listing 7.32 Pseudocode für serverseitige Ping- oder Pong-Frames

```
1 setInterval(function() {  
2     für alle Clients  
3         sende Ping- oder Pong-Frame  
4 }, 10000);
```

WebSocket-Node versendet standardmäßig in einem bestimmten Intervall Pings zum Client. Die Zeitabstände können Sie über `keepaliveInterval` einstellen (siehe [Listing 7.33](#)).

Listing 7.33 Ping-Intervall für WebSocket-Node einstellen

```

1 var WebSocketServer = require("websocket").server;
2 var webSocketServer = new WebSocketServer({
3     httpServer:httpServer,
4     autoAcceptConnections:false,
5     keepaliveInterval:10000
6 });

```

Wenn Sie die Implementierung Ihres Clients anpassen können, lässt sich die Verwaltung zur Aufrechterhaltung der Verbindung mit Pings bzw. Pongs auch auf den Client übertragen.

Generell ist es ratsam, Ping- und Pong-Frames in regelmäßigen Abständen zu versenden. Allerdings sollten Sie ein geeignetes Intervall wählen. Zu lange Intervalle können dazu führen, dass die Verbindung unterbrochen wird. Zu kurze Intervalle erhöhen das Transfervolumen und schlagen deshalb mit erhöhter Datenverarbeitung wegen des zusätzlichen Datenverkehrs zu Buche.

7.3.2 Das Cache-Problem im Internet Explorer 10

Der Internet Explorer unterstützt seit der Version 10 das WebSocket-Protokoll. In einigen Fällen kann es mit dieser Browerversion zu Schwierigkeiten beim Aufbau einer WebSocket-Verbindung mit einem Server kommen. Diese können Sie immer dann beobachten, wenn Sie in Ihrer Webanwendung einen WebSocket-Endpunkt implementiert haben, der bis auf das URL-Schema dieselbe Adresse aufweist wie Ihr HTTP-Endpunkt. Zum Beispiel wenn die URL des HTTP-Endpunkts so aussieht:

http://localhost/

Und wenn Sie den WebSocket-Endpunkt so gewählt haben wie in diesem Beispiel:

ws://localhost/

In so einer Konstellation kann der IE 10 keine WebSocket-Verbindung zu Ihrem WebSocket-Server aufbauen. Das Problem kann durchaus noch relevant sein, denn auch ältere Browerversionen sind noch im Einsatz.

Um besser zu verstehen, wie sich dieses Problem bemerkbar macht, betrachten wir es an einem konkreten Beispiel. Angenommen Sie haben die Webanwendung aus [Abschnitt 6.2.2](#) implementiert, die einen Echo-Dienst über einen WebSocket bereitstellt. Falls Sie nun die URL

http://localhost:4000/

mit dem IE 10 aufrufen, liefert Ihnen der HTTP-Server ein statisches HTML-Dokument. Darin befindet sich ein JavaScript-Programm mit dem Codefragment, das in [Listing 7.34](#) zu sehen ist.

Listing 7.34 Eröffnung eines WebSockets

```

1 var ws = new WebSocket("ws://localhost:4000/");
2 ws.onopen = function(e){...};
3 ws.onmessage = function(e){...};
4 ws.onclose = function(e){console.log(e.code)};

```

Wenn Sie sich beim Laden der HTML-Seite die JavaScript-Konsole des IE-Entwicklertools anschauen, sehen Sie, dass dort eine Fehlermeldung angezeigt wird (siehe Bild 7.27). Die Entwicklertools des IE öffnen Sie über das Menü oder mit der [F12]-Taste.



Bild 7.27 Fehlermeldung beim WebSocket-Verbindungsaufbau im Internet Explorer 10

Der Browser meldet, dass der Statuscode des Handshake-Requests 200 ist und nicht wie erforderlich 101. Nun stellt sich die Frage, wie es zu dem falschen Responsecode gekommen ist. Der Grund für den Fehler liegt im lokalen Browser-Cache. Darin werden die Antworten der zuvor gestellten Anfragen für einen bestimmten Zeitraum zwischengespeichert. Der Browser kann bei wiederholter angefragten Ressourcen darauf zurückgreifen und muss nicht erneut einen Request absetzen. Benötigte Ressourcen werden erkannt, wenn sie im Cache des Browsers vorliegen, und die Antwort aus dem Zwischenspeicher geladen. Dieses Vorgehen spart Datenverkehr ein und Anfragen können schneller verarbeitet werden, da die Nachrichten nicht über die Weiten des Internets transportiert werden müssen.

Im IE 10 ist die Verwendung von zwischengespeicherten Ressourcen so umgesetzt, dass diese in bestimmten Situationen zu einem unerwünschten Verhalten führt.

In unserem Beispiel wird die Antwort auf die Anfrage der URL

http://localhost:4000/

im Cache gespeichert. Wenn Sie nun mit Ihrem Client eine WebSocket-Verbindung zu

ws://localhost:4000/

öffnen möchten, wird im ersten Schritt des Handshakes ein HTTP GET-Request an die entsprechende HTTP URL abgesetzt. An dieser Stelle passiert nun der Fehler, denn der Browser stellt keine neue Anfrage an den Server, sondern generiert die Antwort aus den bereits

vorliegenden Daten im Cache. Da diese zwischengespeicherte Response den Statuscode 200 aufweist, bricht die WebSocket-API die Verbindung ab.

Diesen Fehler können Sie serverseitig beheben, indem Sie den Cache-Control-Header der Response auf den Wert no-cache setzen und dadurch das Cachen unterbinden. [Listing 7.35](#) zeigt Ihnen eine Umsetzung mit Node.js.

Listing 7.35 Cache-Control auf no-cache setzen

```
1 var httpServer = http.createServer(function (req, res) {  
2     fs.readFile(__dirname+"/index.html", function (err, data) {  
3         if(err) {  
4             res.writeHead(500);  
5             return res.end('Error loading index.html');  
6         }  
7         res.writeHead(200, {'Cache-Control': 'no-cache'});  
8         res.end(data);  
9     })  
10    })  
11});
```

Wenn Sie anschließend Ihren Server neu starten und die Seite neu laden, wird die WebSocket-Verbindung erfolgreich aufgebaut.

Bereits im Internet Explorer 11 wurde dieser Fehler behoben. Andere Browser, die das WebSocket-Protokoll unterstützen, scheinen nicht betroffen zu sein. Beachten Sie diesen Bug ggf. aus Gründen der Abwärtskompatibilität, wenn bei Ihrer Anwendung ein HTTP-Endpunkt mit einem WebSocket-Endpunkt übereinstimmt.

8

Beispielanwendungen

Genug geredet und an kleinen Erläuterungsbeispielen gebastelt. Jetzt geht es ans Eingemachte. Wir entführen Sie jetzt in Entwicklungsprojekte, die tiefsschürfendere Betrachtungen eröffnen und sich von den gängigen Beispielen unterscheiden, um damit die Tragweite und Innovationskraft von WebSockets zu verdeutlichen.



Die vollständigen Quellen stehen für Sie auf der Webseite zum Buch bereit. Lassen Sie es uns jedoch zunächst lieber gemeinsam Schritt für Schritt probieren, ohne direkt auf die fertigen Anwendungen zurückzugreifen.



Fragen, die dieses Kapitel beantwortet:

- **Abschnitt 8.1:** Wie kann man Webanwendungen mit einer Smartphone-/Tablet-Fernbedienung steuern?
Wir entwickeln eine generische Fernbedienung für Smart Devices, die zur Steuerung von Webanwendungen in entkoppelten Geräten wie z. B. Smart-TVs verwendet werden kann.
WebSocket-Stack: Node.js, Socket.io, Express.js
- **Abschnitt 8.2:** Wie kann man ein Chat-System mit WebSockets implementieren?
Wir programmieren eine einfache textbasierte Chat-Anwendung.
WebSocket-Stack: Vert.x
- **Abschnitt 8.3:** Wie kann man eine Heatmap für Usability-Tests realisieren?
Wir zeigen Ihnen eine Anwendung zur Unterstützung von Web-Usability-Tests. Via WebSockets werden kontinuierlich Mauspositionen zur Auswertung an einen Server übertragen, womit sich in einem nachgelagerten Verarbeitungsschritt Heatmaps erstellen lassen.
WebSocket-Stack: Node.js, Socket.io
- **Abschnitt 8.4:** Wie kann man mithilfe von WebSockets eine Webcam als Überwachungskamera einsetzen?
Wir realisieren eine Videoüberwachung per Webcam, wobei in diesem Beispiel die Übertragung von binären Daten im Vordergrund steht.
WebSocket-Stack: Express.js, WebSocket-Node

■ 8.1 Fernbedienung von Webanwendungen mit einer Smartphone-/Tablet-Fernbedienung

Um die Vielseitigkeit von WebSockets hervorzuheben, wollen wir Ihnen in diesem Beispiel die Implementierung einer Anwendung präsentieren, die es ermöglicht, eine Webanwendung mit einer anderen Webanwendung zu steuern. Die Steuer-Webanwendung soll von einem Smartphone oder Tablet-PC bedient werden. Die zu steuernde Webanwendung wird z. B. von einem Smart-TV oder einem PC ausgeführt und dargestellt.

Wie Sie in [Bild 8.1](#) erkennen können, besteht die Architektur insgesamt aus drei Komponenten; einem Server, einem Fernseher oder PC und einem Smart Device als Steuergerät. Die Kommunikation zwischen Smart Device und Fernseher findet nicht auf direktem Weg zwischen den beiden Geräten statt. Das mobile Endgerät schickt zunächst Steuernachrichten an den Server (1), welche dann an den Fernseher weitergeleitet werden (2). Für Benutzer hat es allerdings trotzdem den Anschein, als würden Smart Device und Fernseher direkt miteinander sprechen (3).

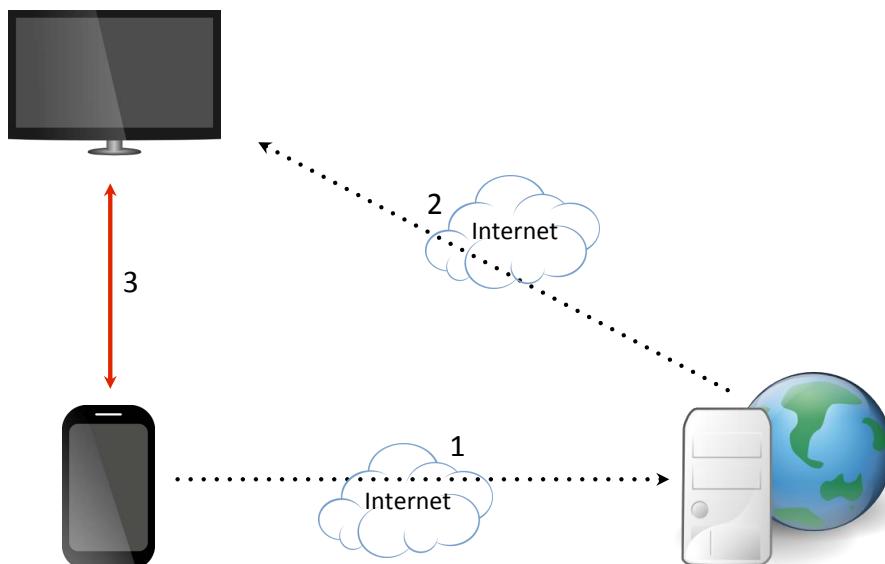


Bild 8.1 Grobes Architekturmodell der Anwendung

Im Detail besteht der Server aus zwei Teilen (siehe [Bild 8.2](#)), einem Webserver und einem Event-Server. Der Webserver ist für die Erzeugung und Auslieferung von Webinhalten zuständig und der Event-Server für die Echtzeit-Kommunikation. Als Erstes muss die zu steuernde Anwendung vom Smart-TV beim Webserver angefordert werden (1-2). Dann

wird die Steueranwendung von einem Smart Device ebenfalls beim Webserver abgerufen (3–4). Wird nun ein bestimmtes Steuerkommando durch die Interaktion mit der Bedienschnittstelle ausgelöst, aktiviert dies einen Event-Listener (5). Dieser wandelt die Eingabe in eine entsprechende Befehlsnachricht um und schickt diese an den Event-Server (6). Der Eventserver leitet den Befehl an den entsprechenden Fernseher weiter (7). Der Webserver und der Event-Server können auf demselben physikalischen Rechner oder bei Bedarf auch getrennt voneinander betrieben werden. Die Beispielimplementierung setzt erstgenannte Variante um.

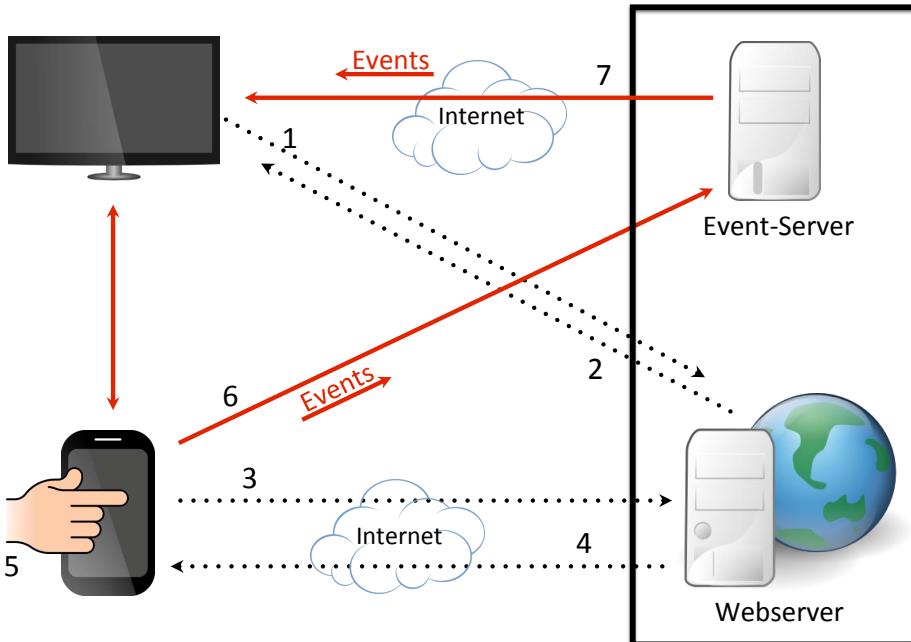


Bild 8.2 Das Architekturmodell im Detail

Die Architektur kann grundsätzlich sowohl mit Server-Sent Events (SSE, [Abschnitt 3.5](#)) als auch mit WebSockets realisiert werden. SSE sind z. B. *eine* Möglichkeit, um in Schritt 7 die Nachrichten vom Event-Server zur gesteuerten Anwendung zu schicken. Vorausgehend würde die Steueranwendung in Schritt 6 für jeden einzelnen Befehl einen HTTP-Request an den Server senden. Dieser Implementierungsansatz hätte den Nachteil, dass mit jedem dieser Requests ein Overhead an Daten entstünde, der in der Summe nicht zu vernachlässigen wäre. Ressourcenschonender wäre hier ein einzelner WebSocket-Kanal für den Datenaustausch. Auch eine Mischkonstellation ist als Lösungsansatz denkbar, bei dem die Kommandos in SSE umgewandelt und an die betreffende Anwendung weitergeleitet werden. Die verschiedenen Technologien müssten dann allerdings aneinander angepasst und integriert werden. Ein Anwendungsfall, der allerdings auch bei dieser Überlegung nicht berücksichtigt wird, ist der Nachrichtenverkehr in die andere Richtung, wodurch sich neue Entwicklungsmöglichkeiten ergeben würden. Das Smart-TV könnte der Steueranwendung

z. B. mitteilen, dass das Interface zur Steuerung geändert werden muss. Damit wäre eine individuelle Anpassung der Bedienoberfläche an die gerade genutzte Anwendung möglich. Auch mehrere verschiedene Interfaces könnten dem Benutzer dann zur Auswahl angeboten werden. Für eine solche Anforderung kommt ausschließlich eine Implementierung mit WebSockets in Betracht.

In diesem Beispiel soll die zu steuernde Anwendung einen roten Kreis auf einem Fernseher oder PC-Monitor darstellen, dessen Position durch den Benutzer verändert werden kann. Gleichzeitig soll ein QR-Code generiert werden, der einen Link zu der Steueranwendung beinhaltet. Dieser QR-Code kann dann mit einem Smart Device fotografiert werden, um die enthaltene URL mit einem Webbrowser zu öffnen und auf die Steueranwendung zuzugreifen. Diese Anwendung besteht in unserem Beispiel aus vier Buttons. Betätigt der Benutzer einen dieser Knöpfe, kann er darüber den roten Kreis steuern.

Der Web- und Event-Server wird mit Node.js [Joy14] implementiert. Für die WebSockets verwenden wir die Abstraktionsschicht Socket.io [Rau14], da dieses Node.js-Modul für jede WebSocket-Verbindung eine eigene Session-ID verwaltet. Diese wird für die Zuordnung der Webanwendung im Steuergerät und im Smart-TV benötigt. Einen weiteren Vorteil bietet Socket.io hinsichtlich des Fallbacks auf Long Polling, falls der Browser keine WebSockets unterstützt.

Die Webanwendung wird mit dem Node.js-Modul Express [Hol14a] umgesetzt, das uns bei der Erstellung der Webanwendung hilft. Wie in Abschnitt D.3 erklärt wird, können verschiedene Template-Engines zur Spezifikation des Aufbaus von Webseiten verwendet werden. Wir haben uns in diesem Fall für Jade [Hol14b] entschieden. Die Vorlagen liegen in einem festgelegten Ordner und werden abhängig von der URL geladen und gerendert.

Wie Sie vielleicht schon in der Architektur (Bild 8.2) erkennen konnten, gliedern sich auch die Quelltexte in vier zusammenhängende Teile. Die eingesetzten Module nehmen uns einen Großteil der Arbeit ab, sodass *eine* Source-Datei für die jeweilige Komponente ausreicht (siehe Bild 8.3).

Erstellen Sie nun ein neues Express-Projekt und installieren Sie Socket.io mit diesen Befehlen:

```
express remoteControl
cd remoteControl
npm install
npm install socket.io
```

Damit können Sie mit der Implementierung beginnen. Um die Übersicht zu behalten, benennen Sie die Datei *app.js* am besten in *webApp.js* um.

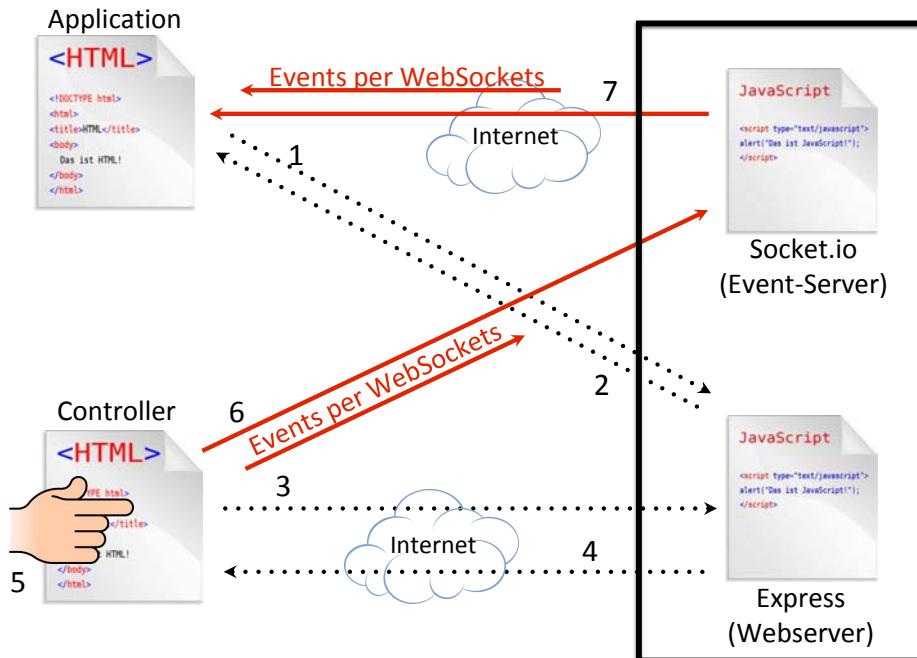


Bild 8.3 Architekturmodell der Quelltexte

In Listing 8.1 wird zunächst der Webserver implementiert. Dabei werden insbesondere die URL-Endpunkte angegeben, die der Webserver bedienen soll. In unserem Beispiel sind es zwei Endpunkte, einer für die Webanwendung, die gesteuert werden soll (/), und einer für die Steueranwendung (/controllers/:sid).

Listing 8.1 Webserver mit den URL-Endpunkten für die Anwendung und Steuerkomponente (*webApp.js*)

```

1 ...
2 var app = express();
3 var httpServer = require("http").Server(app);
4 var evtServer = require('./evtServer.js').listen(httpServer);
5 ...
6 app.get('/', function(req, res) {
7     res.render('index');
8 });
9
10 app.get('/controller/:sid', function(req, res) {
11     res.render('controller', {sid:req.params.sid});
12 });
13
14 httpServer.listen(3000, function() {
15     console.log("Express-Server laeuft auf dem Port 3000");
16 });

```

Listing 8.2 zeigt das Jade-Template der Webanwendung, die im Smart-TV oder PC geöffnet und später durch das Smart Device gesteuert wird. Wir definieren erst ein <canvas>-Element, in dem später die Anwendung gezeichnet wird. Anschließend erstellen wir ein -Tag. Darin generieren wir später unseren QR-Code. Das -Tag ist von einem <a>-Tag umgeben, in das wir den Link zum Controller laden, um durch einen Klick auf die Steueranwendung zu kommen. Das ermöglicht Ihnen, die Anwendung zu testen, falls Sie kein Smartphone oder Tablet-PC mit QR-Code-Scanner zur Hand haben.

Listing 8.2 Definieren einer Zeichenfläche und Erzeugung des QR-Codes (*index.jade*)

```

1 extends layout
2 block content
3   h1 Remote Control
4   canvas#game(width="500",height="400")
5   h2 QR-Code
6   a(target="_blank")#link
7     img#qr
8   script.
9     var socket = io.connect();
10    var host = "http://"+window.location.host;
11    var qrUrl = 'http://chart.apis.google.com/chart?chs=150x150&cht=
12      qr&chl=';
13    socket.on("connect",function(){
14      var sid = socket.io.engine.id;
15      $('#qr').attr('src', qrUrl+host+'/controller/'+sid);
16      $('#link').attr('href', host+'/controller/'+sid);
17    })

```

Durch die Socket.io-Anweisung `io.connect()` wird ein WebSocket zum Server aufgebaut, von dem die geladene Webseite stammt. Ist der Kanal etabliert, wird der `socket.on("connect")`-Event ausgelöst, der in diesem konkreten Fall eine Funktion ausführt, die die Session-ID des Sockets ausliest. Wir werden hier noch zusätzlich jQuery benutzen, um uns die Implementierung etwas leichter zu machen. Mit der Session-ID wird die URL des Controllers konstruiert und in einen QR-Code umgewandelt. Das Bild des QR-Codes wird dann in das `src`-Attribut unseres -Tags geladen. Danach definieren wir noch den Link zum Controller, den wir in das <a>-Tag einbetten.

Anschließend wird der rote Kreis in das <canvas>-Element gezeichnet und die dazugehörige Steuerfunktionen definiert (siehe [Listing 8.3](#)). Der Event-Listener `socket.on("message")` wird immer dann aktiviert, wenn eine Nachricht vom Server eingegangen ist. Die an den Listener gebundene Funktion liest die Nachricht ein und ermittelt daraus, welche Steuerfunktion auszuführen ist, um den Kreis in die befehligte Richtung zu bewegen.

Listing 8.3 Zeichnen der Anwendung sowie Definition der Steuerbefehle und Bewegungen (*index.jade*)

```

1 var canvas = document.getElementById('game');
2 var ctx = canvas.getContext('2d');
3 var speed = 5;
4 ctx.beginPath();
5 ctx.fillStyle = "Red";

```

```

6 var x=200;
7 var y=200;
8 drawCircle();
9 function drawCircle() {
10   ctx.beginPath();
11   ctx.arc(x,y,40,0,Math.PI*2,true);
12   ctx.fill();
13 }
14
15 function clear(){ctx.clearRect(0,0,canvas.width,canvas.height)}
16 function moveRight(){x+=speed;}
17 function moveLeft(){x-=speed;}
18 function moveDown(){y+=speed;}
19 function moveUp(){y-=speed;}
20
21 socket.on("message",function(message){
22   clear();
23   if(message=="left"){moveLeft();}
24   else if(message=="right"){moveRight();}
25   else if(message=="down"){moveDown();}
26   else if(message=="up"){moveUp();}
27   drawCircle();
28 });

```

Ist die *index.jade*-Webseite gerendert, werden im Smart-TV der rote Kreis und daneben der QR-Code angezeigt. Der QR-Code muss nun mit einem Smart Device abfotografiert und die darin enthaltene URL im Browser geöffnet werden. Der GET-Request beinhaltet die Session-ID, die vom Webserver ausgelesen und dem Template *controller.jade* als Parameter übergeben wird (siehe [Listing 8.2, Zeile 12 bis 16](#)).

Im Jade-Template der Steueranwendung (siehe [Listing 8.4](#)) werden vier <div>-Elemente definiert, die als Buttons fungieren. Jedem dieser Divs wird eine ID zugewiesen.

Listing 8.4 Definieren der Buttons (*controller.jade*)

```

1 extends layout
2 block content
3   div.button#left left
4   div.button#right right
5   div.button#up up
6   div.button#down down

```

Bevor wir mit der Implementierung der Steueranwendung beginnen, müssen wir noch herausfinden, ob der Browser, mit dem später die Anwendung aufgerufen werden soll, Touch-Events unterstützt. Falls nicht, müssen wir dies abfangen und auf „normale“ Klick-Events zurückgreifen.

Listing 8.5 Prüfung, ob der Browser Touch-Events erkennt (*controller.jade*)

```

1 ...
2 script.
3 var is_touch_device = 'ontouchstart' in document.documentElement;

```

```

4 var socket = io.connect();
5
6 if(is_touch_device) {
7   touch();
8 }
9 else {
10   click();
11 }
```

Danach definieren wir die Funktion, die ausgeführt wird, falls Touch-Events unterstützt werden.

Listing 8.6 Definieren der Touch-Events (*controller.jade*)

```

1 function touch() {
2   $('#left').bind('touchstart', function(e) {
3     e.preventDefault();
4     socket.json.send({sid:'#{sid}', command:'left'});
5   });
6
7   $('#right').bind('touchstart', function(e) {
8     e.preventDefault();
9     socket.json.send({sid:'#{sid}', command:'right'});
10  });
11
12  $('#up').bind('touchstart', function(e) {
13    e.preventDefault();
14    socket.json.send({sid:'#{sid}', command:'up'});
15  });
16
17  $('#down').bind('touchstart', function(e) {
18    e.preventDefault();
19    socket.json.send({sid:'#{sid}', command:'down'});
20  });
21 }
```

Und natürlich implementieren wir auch die Fallback-Funktion, falls der Browser keine Touch-Befehle versteht.

Listing 8.7 Definieren des Fallbacks für Klick-Events (*controller.jade*)

```

1 function click() {
2   $('#left').click(function() {
3     socket.json.send({sid:'#{sid}', command:'left'});
4   });
5
6   $('#right').click(function() {
7     socket.json.send({sid:'#{sid}', command:'right'});
8   });
9
10  $('#down').click(function() {
11    socket.json.send({sid:'#{sid}', command:'down'});
12  });
13}
```

```

12   });
13
14   $('#up').click(function() {
15     socket.json.send({sid:'#{sid}', command:'up'});
16   });
17 }

```

Diese Funktionen enthalten Event-Listener, die wiederum andere Funktionen ausführen, falls einer der Divs betätigt wird. Die jeweiligen Funktionen verschicken eine Nachricht im JSON-Format, die zwei Parameter beinhaltet. Der erste Parameter sid trägt die Session-ID der zu steuernden Anwendung und der zweite Parameter command den Befehl.

Das jeweilige JSON-Objekt wird vom Event-Server über einen WebSocket empfangen und verarbeitet. Die Parameter werden dabei herausgelesen und der Befehl über den WebSocket, der durch die Session-ID identifiziert ist, an den betreffenden Fernseher oder PC weitergeleitet.

Listing 8.8 Event-Server auf Grundlage von Socket.io (*evtServer.js*)

```

1 module.exports.listen = function(httpServer) {
2   var io = require('socket.io').listen(httpServer);
3   io.sockets.on('connection', function(socket) {
4
5     socket.on("message",function(message){
6       if(io.sockets.connected[message.sid])
7         io.sockets.connected[message.sid].emit("message",
8           message.command)
9     });
10   });

```

In der Datei *layout.jade* fehlen jetzt noch zwei Quellen für unseren JavaScript-Code.

Listing 8.9 Externe absolute JavaScript-Quellen (*layout.jade*)

```

1 script(src="https://code.jquery.com/jquery-1.11.1.min.js")
2 script(src="/socket.io/socket.io.js")

```

Zu guter Letzt definieren wir einige kleinere CSS-Regeln, um das Aussehen der Anwendung etwas ansehnlicher zu gestalten (siehe [Listing 8.10](#)).

Listing 8.10 Stylesheet der Anwendung (*style.css*)

```

1 body {
2   padding: 20px;
3   font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
4 }
5
6 a {
7   color: #00B7FF;
8 }
9
10 button {

```

```
11     width: 200px;  
12     height: 200px;  
13     margin: 5px;  
14     float: left;  
15     font-weight: bold;  
16 }  
17  
18 #game {  
19     float: left;  
20     border: 1px solid #ddd;  
21     display: block;  
22 }
```

Starten Sie nun die Anwendung, indem Sie den Webserver mit dem folgenden Kommandozeilenaufruf starten:

```
node webApp.js
```

Öffnen Sie danach die URL

```
http://localhost:3000/
```

in Ihrem Browser (siehe Bild 8.4).

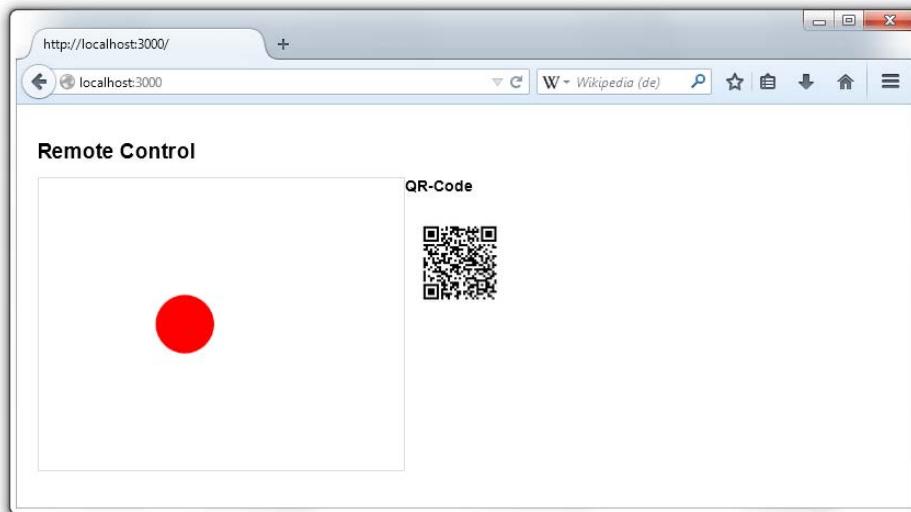


Bild 8.4 Anwendung, die ferngesteuert werden soll

Falls Sie kein Smartphone oder keinen Tablet-PC zur Verfügung haben, klicken Sie einfach auf den QR-Code. Die Steueranwendung öffnet sich dann in einem neuen Tab (siehe Bild 8.5). Positionieren Sie die beiden Fenster am besten so, dass Sie beide Anwendungen gleichzeitig sehen können.

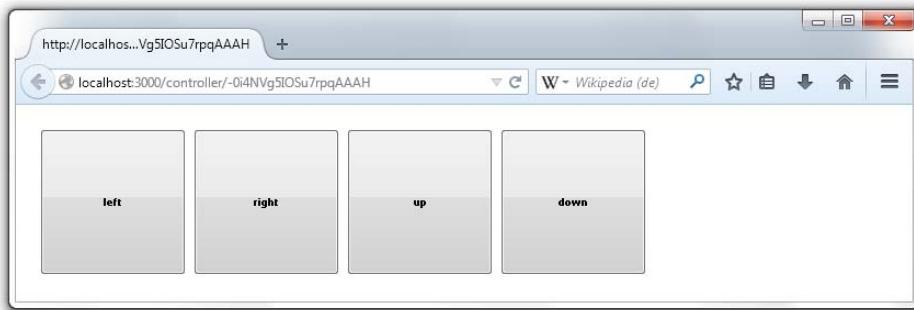


Bild 8.5 Anwendung zur Steuerung

Haben Sie ein Smartphone oder Tablet zur Hand, müssen Sie die Anwendung zunächst über die URL mit der entsprechenden IP-Adresse oder dem Domainnamen Ihres Servers aufrufen, z. B.

http://192.168.1.10:3000/

Der *localhost* muss hier also durch die jeweilige IP-Adresse ersetzt werden.

Lesen Sie den QR-Code mit einer entsprechenden App ein und öffnen Sie den Link, der im QR-Code eingebettet ist, mit einem Browser. Achten Sie darauf, dass der Browser das WebSocket-Protokoll unterstützt (siehe [Abschnitt 5.2](#)). Natürlich können Sie auch einen Browser ohne WebSocket-Support wählen, auf die Performance und Vorteile von WebSockets müssen Sie dann allerdings verzichten. Klicken Sie oder berühren Sie eines der Divs und der rote Kreis bewegt sich über den Bildschirm.

■
Die Quellcodes zu dieser Beispielanwendung können Sie von unserer Webseite herunterladen:

<http://websocket101.org/>

Für den Fall, dass Sie mit Node.js nicht vertraut sind, haben wir Ihnen dort ebenfalls eine Java-Variante, die mit dem Play Framework implementiert ist, zum Abruf bereitgelegt.

■ 8.2 Chat-System

In diesem Kapitel zeigen wir Ihnen, wie Sie schnell und einfach einen Chat mithilfe des Frameworks Vert.x realisieren können. Für Chats bieten WebSockets eine optimale Lösung, da permanent Nachrichten zwischen Chat-Teilnehmern und dem Server in beide Richtungen verschickt werden müssen.

Dieses Beispiel möchten wir mit Java realisieren, da wir uns damit bereits in [Abschnitt 6.3](#) vertraut gemacht haben. Erstellen Sie zunächst die Datei für den Server. Darin implemen-

tieren wir die Java-Klasse mit dem Namen *ChatServer.java*, welche die Klasse *Verticle* erweitert (siehe Listing 8.11).

Listing 8.11 Der Route-Matcher (*ChatServer.java*)

```

1 ...
2 public class ChatServer extends Verticle {
3     public void start() {
4         RouteMatcher rm = new RouteMatcher();
5
6         rm.get("/", new Handler<HttpServerRequest>() {
7             public void handle(HttpServerRequest req) {
8                 req.response().sendFile("index.html");
9             }
10        });
11
12        rm.get("/jquery.js", new Handler<HttpServerRequest>() {
13            public void handle(HttpServerRequest req) {
14                req.response().sendFile("jquery.js");
15            }
16        });
17    ...

```

Wir definieren zuerst einen Route-Matcher, der eine ähnliche Funktionalität wie der des Request-Handlers von Express.js hat. Er bietet uns ebenfalls verschiedene Handler für alle HTTP-Request-Methoden, also GET, POST, PUT, DELETE etc. Wir legen fest, dass der Server bei einem GET an die Document-Root (/) eine statische HTML-Seite an den Client schickt, welche die Chat-Anwendung beinhaltet. Bei einem GET auf /jquery.js liefern wir die jQuery-Bibliothek zurück. Im nächsten Schritt müssen wir unseren Webserver erzeugen und die WebSocket-Handler definieren (siehe Listing 8.12).

Listing 8.12 WebSocket-ChatServer (*ChatServer.java*)

```

1 HttpServer webServer = vertx.createHttpServer();
2 final List<ServerWebSocket> sockets =
3     new ArrayList<ServerWebSocket>();
4
5 webServer.websocketHandler(new Handler<ServerWebSocket>() {
6     public void handle(final ServerWebSocket ws) {
7         sockets.add(ws);
8         ws.dataHandler(new Handler<Buffer>() {
9             public void handle(Buffer data) {
10                 for (ServerWebSocket socket : sockets) {
11                     socket.writeTextFrame(data.toString());
12                 }
13             }
14         });
15         ws.endHandler(new Handler<Void>() {
16             public void handle(Void arg0) {
17                 sockets.remove(ws);
18             }
19         });
20     }
21 });

```

```

19     }
20 });
21 webServer.requestHandler(rm).listen(3000);
22 System.out.print("Der ChatServer läuft auf dem Port 3000");

```

Dazu erstellen wir eine Arrayliste des Datentyps ServerWebSocket. Darin speichern wir alle WebSockets, die sich erfolgreich verbunden haben ([Zeile 6](#)). Wenn eine Nachricht beim Server ankommt, senden wir diese an alle WebSockets in der Arrayliste ([Zeile 9](#)). Falls ein WebSocket geschlossen wird, entfernen wir diesen wieder. Als Letztes weisen wir dem Request-Handler unseres Webservers den Route-Matcher zu und definieren einen Port. Nun wird der Webserver auf die Request-Handler des Route-Matchers reagieren.

In [Listing 8.13](#) erstellen wir jetzt eine HTML-Datei für den Client, in der wir die Chat-Anwendung implementieren. Ein <div>-Tag definiert, wo später der Verlauf des Chats angezeigt wird. Zusätzlich wird noch ein Textfeld zur Eingabe von Nachrichten und ein Button gebraucht, mit dem wir unsere Mitteilungen abschicken können.

Listing 8.13 Definition des Chat-Fensters und der Formularfelder (*index.html*)

```

1 <h2>Chat</h2>
2 <div id="chat"></div>
3 <input type="text" id="message">
4 <button id="send">Nachricht senden</button>

```

In dem nun folgenden <script>-Tag implementieren wir die Chat-Anwendung. Als Erstes wird ein JavaScript-Objekt initialisiert, das zwei Eigenschaften besitzt, den Namen des Absenders und die Nachricht. Nachdem sich jemand mit dem WebSocket-Server verbunden hat, wird ein Fenster aufgerufen, das nach dem Namen des Chat-Teilnehmers fragt. Diesen speichern wir in der Eigenschaft name.

Listing 8.14 Chat-Anwendung fragt nach dem Namen des Teilnehmers (*index.html*).

```

1 var msg = {
2   name: "",
3   message: ""
4 }
5 var ws = new WebSocket('ws://' + window.location.host);
6
7 (function setName(){
8   msg.name = prompt("Willkommen im Chat! Wie ist dein Name?", "");
9   if(msg.name == "") {
10     setName();
11   }
12 })();
13 ...

```

Als Nächstes kümmern wir uns um das Verschicken der Nachrichten.

Listing 8.15 Der Versand der Nachrichten (*index.html*)

```

1 $('#message').keydown(function(evt) {
2   if(evt.which == 13) {

```

```

3     msg.message = $('#message').val();
4     ws.send(JSON.stringify(msg));
5     msg.message = "";
6     $('#message').val("");
7   }
8 });
9
10 $('#send').click(function() {
11   msg.message = $('#message').val();
12   ws.send(JSON.stringify(msg));
13   msg.message = "";
14   $('#message').val("");
15 });

```

Dazu benötigen wir zwei Event-Listener; einen, wenn die Eingabetaste gedrückt wird, und einen weiteren, wenn auf den Button geklickt wird. Falls einer dieser beiden Events eintrifft, lesen wir die Nachricht aus dem Textfeld heraus und speichern sie in der Eigenschaft `message` unseres JavaScript-Objektes ab. Als Nächstes wandeln wir dieses in ein JSON-Objekt um und senden die Nachricht an den Server. Anschließend leeren wir das Textfeld und die Eigenschaft `message` wieder.

Jetzt bestimmen wir noch, was passieren soll, wenn Nachrichten ankommen.

Listing 8.16 Empfang der Nachrichten (*index.html*)

```

1 ws.onmessage = function(evt) {
2   try {
3     var message = JSON.parse(evt.data);
4     $('#chat').append('<p><b>' + message.name + '</b>: ' +
5       + message.message + '</p>');
6   }
7   catch(e) {
8     $('#chat').append('<p class="error">invalid message</p>');
9   }
10 }

```

Falls eine Nachricht ankommt, wandeln wir diese in ein JSON-Objekt um, lesen daraus den Absender sowie den Inhalt der Nachricht und fügen beides zum Chat-Verlauf hinzu. Falls es nicht möglich ist, die Nachricht als JSON-Objekt zu parsen, fangen wir diesen Fehler ab. Zum Schluss verschönern wir unsere Anwendung etwas mit entsprechenden Stylesheets.

Listing 8.17 Stylesheet der Chat-Anwendung (*index.html*)

```

1 <style>
2   #chat {
3     width: 700px;
4     height: 400px;
5     border: 1px solid #ddd;
6     overflow-y: scroll;
7     padding: 5px;
8   }

```

```

9      #message {
10         width: 400px;
11     }
12     #chat p {
13         margin: 0 0 0 5px;
14     }
15     .error {
16         color:red;
17     }
18 </style>
```

Um das Chat-Programm zu starten, geben Sie bitte in der Konsole folgenden Befehl ein:

```
vertx run chatServer.js
```

Wenn Sie daraufhin die URL

http://localhost:3000/

in Ihrem Browser öffnen, sollten Sie den in [Bild 8.6](#) dargestellten Screen sehen.

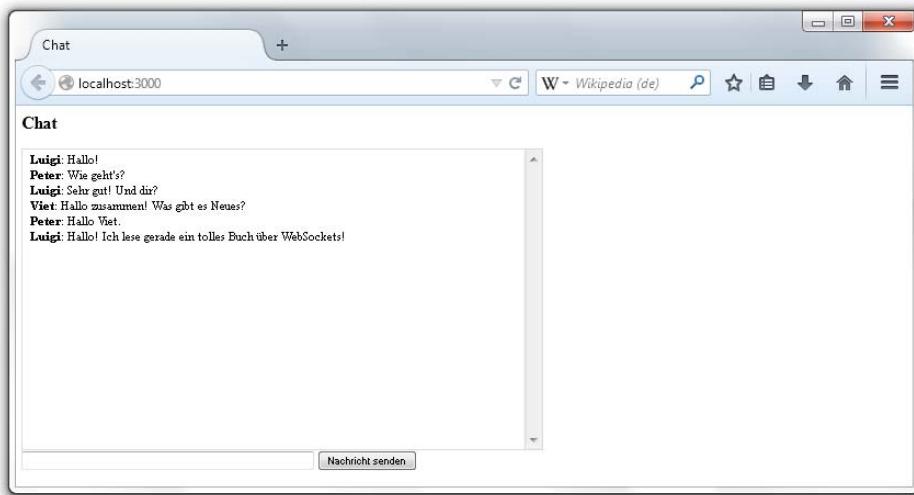


Bild 8.6 Chat-Anwendung

Beachten Sie bitte, dass die Nachrichten bei dieser Anwendung noch nicht validiert sind! Ein Angreifer, der am Chat teilnimmt, könnte demnach JavaScript-Code als Nachrichten verschicken. Falls Sie diese Anwendung in der Produktion erweitern wollen, raten wir Ihnen, alle Nachricht auf dem Server zu validieren und zu escapen.



Auch diese Anwendung können Sie von unserer Homepage beziehen. Dort ist auch noch eine weitere Vert.x-Variante des Chat-Servers in JavaScript vorhanden. Und falls Ihnen die Implementierungen mit Vert.x nicht zusagen, bieten wir Ihnen dort ebenfalls eine Node.js-Implementierung mit Express.js an.

■ 8.3 Heatmap für Usability-Tests

Wir wollen in diesem Kapitel eine Mouse-Tracking-Anwendung realisieren, die die Mausposition von Benutzern aufzeichnet und über WebSockets an einen Client schickt. Die Mauspositionen werden beim Client ausgewertet und in Form einer Heatmap visualisiert. Hierfür bietet sich eine sehr gute JavaScript-Bibliothek namens *heatmap.js v1.0* von Patrick Wied an, die unter der MIT- und der bei vielen beliebten Beerware-Lizenz steht.¹

Wir werden die Anwendung in diesem Kapitel mit Node.js in Kombination mit Express.js und Socket.io implementieren.

Erstellen Sie ein neues Express.js-Projekt mit den folgenden Befehlen:

```
express mouse_tracking
cd mouse_tracking
npm install
```

Installieren Sie zusätzlich auch noch Socket.io:

```
npm install socket.io
```

Wir beginnen damit, in einer JavaScript-Datei einen Webserver und einen Socket.io-Server zu erzeugen und beide miteinander zu verbinden (siehe Listing 8.18).

Listing 8.18 Erzeugen eines Web- und Socket.io-Servers (*app.js*)

```
1 var app = express();
2 var httpServer = require('http').Server(app);
3 var io = require('socket.io').listen(httpServer);
4 ...
5 app.get('/', function (req, res){
6     res.sendFile(__dirname + "/static/bootstrap.html");
7 });
8
9 app.get('/admin', function (req, res) {
10     res.sendFile(__dirname + "/static/admin.html");
11 })
```

In Zeile 5 legen wir für den Root-Pfad fest, dass der Server eine statische HTML-Seite zurückgibt. Dabei handelt es sich um die Webseite, für die wir die Mausinteraktionen von Benutzern auswerten möchten. Die Mauspositionen auf der Seite werden dort aufgezeichnet und zum Server via WebSockets geschickt. Der Server leitet dann die Informationen zu einer Administrationsseite weiter.

Die Oberfläche der Administrationsseite, die in unserem Beispiel wieder eine statische HTML-Seite ist, soll die Mauspositionsdaten, die vom Server kommen, zu einer Heatmap verarbeiten. Der Pfad zu dieser Seite muss natürlich auch festgelegt werden, was wir in Zeile 9 machen.

Zur Erfassung der Mauspositionen nutzen wir das Beispiel auf der Homepage von heatmap.js.² In dieser Implementierung werden die Mauspositionen des Benutzers in be-

¹ <https://github.com/pa7/heatmap.js/releases/tag/v1.0/>

² <http://www.patrick-wied.at/static/heatmapjs/example-mousemove-heatmap.html>

stimmten Abständen aufgezeichnet und direkt auf der Seite des Benutzers dargestellt. Da wir die Mauspositionen allerdings an unseren WebSocket-Server schicken und zu einer Admin-Seite weiterleiten möchten, passen wir das Beispiel entsprechend an. Nur derjenige, der später Zugriff auf die Admin-Seite hat, kann dann anhand der dargestellten Heatmap erkennen, wo sich die Maus am häufigsten befindet.

Schauen wir uns nun an, wie wir die Implementierung von Patrick Wied in ein Zusammenspiel mit WebSockets bringen können.

Listing 8.19 Zustandsinformationen und Mauspositionen (*Bootstrap.html*)

```

1 var socket = io.connect();
2 var element = document.getElementById("trackingArea");
3 var capture = true, idle = false, over = false,
4     x = 0,
5     y = 0;
6 ...
7 function add(e){
8     x = e.layerX;
9     y = e.layerY;
10    socket.json.send({x:x,y:y,count:1});
11 }

```

Zuerst einmal verbinden wir uns mit dem Server und speichern den <div>-Tag, in dem wir die Mauspositionen erfassen wollen, in einer Variablen ab. Danach initialisieren wir drei boolesche Variablen, die uns Zustandsinformationen liefern. Die Variable `capture` soll verhindern, dass bei einer Mausbewegung ständig Daten an den Server gesendet werden. Mit der `idle`-Variablen werden wir prüfen können, ob die Maus noch bewegt wird, und mithilfe der `over`-Variablen können wir abfragen, ob sich die Maus noch innerhalb des Bereichs befindet, in dem wir aufzeichnen wollen.

Anschließend bestimmen wir die Koordinaten der Mausposition. Dazu definieren wir eine Funktion, der als Argument ein Mausevent übergeben werden muss. Daraus lassen sich die Werte der beiden Positionsvariablen `x` und `y` entnehmen. Die Koordinaten senden wir danach als JSON an den Server.

Listing 8.20 Reaktion auf Mausbewegungen (*Bootstrap.html*)

```

1 setInterval(function(){
2     capture = true;
3 }, 80);
4
5 // pruefen, ob die Maus ungenutzt ist
6 var idlechecker = setInterval(function(){
7     if(over && !idle){
8         // wenn sie ungenutzt ist, wird die Simulation gestartet
9         // und die letzten x/y-Koordinaten werden hinzugefuegt
10        idle = setInterval(function(){
11            socket.json.send({x:x,y:y,count:1});
12        }, 1000);
13    }
14 }, 150);

```

```

15
16 element.onmousemove = function(e){
17     over = true;
18     if(idle){
19         clearInterval(idle);
20         idle = false;
21     }
22
23     else if(capture){
24         add(e);
25         capture = false;
26     }
27 };

```

Als Nächstes legen wir ein Intervall fest, das die `capture`-Variable alle 80 Millisekunden auf `true` setzt. Warum dies gemacht wird, erklären wir im nächsten Absatz. Wir definieren ein zweites Intervall, das alle 150 Millisekunden prüft, ob die Maus nicht mehr bewegt wird und ob sie sich in dem Bereich der Aufzeichnung befindet. Wenn das der Fall ist, erhält der Server die Positionsdaten nur noch einmal in der Sekunde. Die Intervalldauer speichern wir in der Variablen `idle`.

Wenn die Maus im Aufzeichnungsbereich bewegt wird, setzen wir als Erstes die Variable `over` auf `true`. Danach prüfen wir, ob das Intervall `idle` noch aktiv ist. Falls ja, wird das alte Intervall gelöscht und der Variablen `false` zugewiesen. Die Koordinaten werden jetzt, sobald sich die Maus auf einer anderen Position befindet, zum Server geschickt. Da dies mehr Daten sind, als für die Visualisierung benötigt werden, nutzen wir an dieser Stelle das Intervall aus [Zeile 1](#). Wir verschicken nur dann die Daten einer neuen Position, wenn die Variable `capture` den Wert `true` besitzt, also alle 80 Millisekunden. Danach müssen wir die Variable wieder auf `false` setzen.

Jetzt fehlen noch die Reaktionen auf zwei weitere Mausevents; das ist zum einen der Mausklick und zum anderen die Situation, wenn sich die Maus außerhalb des Aufnahmebereichs befindet.

Listing 8.21 Weitere Mausevents (*Bootstrap.html*)

```

1 element.onclick = function(e){
2     add(e);
3 };
4 element.onmouseout = function(){
5     over = false;
6 };

```

Bei jedem Mausklick des Benutzers sollen auch die zugehörigen Koordinaten an den Server geschickt werden. Wenn die Maus außerhalb des von uns definierten Bereichs der Aufnahme bewegt wird, sollen dementsprechend auch keine Positionsdaten bestimmt werden. Darum erhält `over` den Wert `false`. Nun geht es weiter mit der Datenverarbeitung auf der Serverseite.

Listing 8.22 Validierung der ankommenden Daten (*app.js*)

```

1 io.sockets.on('connection', function(socket) {
2     socket.on('message', function(point) {
3         if(!isNaN(point.x) && !isNaN(point.y)) {
4             io.sockets.emit('data', point);
5         }
6     })
7 });

```

Bevor wir die Daten im Event `data` zur Auswertung weiterleiten, wird im ersten Schritt zur Sicherheit validiert, ob es sich dabei noch um Zahlen handelt. Wir wissen schließlich nicht, ob die JSON-Objekte manipuliert wurden. Weiter geht es mit der Admin-Seite.

Listing 8.23 Administrationsseite (*admin.html*)

```

1 var socket = io.connect();
2
3 // Konfiguration der Heatmap
4 var config = {
5     element: document.getElementById("heatmapArea"),
6     radius: 30,
7     opacity: 50,
8 };
9
10 //erzeugt und initialisiert die Heatmap
11 var heatmap = h337.create(config);
12
13 socket.on("data", function(data) {
14     heatmap.store.addDataPoint(data.x, data.y, data.count);
15 })

```

Wir konfigurieren zunächst die Darstellung der Heatmap nach unseren Wünschen. Nun speichern wir darin bei jedem eintreffenden `data`-Event die dazugehörigen Positionen. Damit die Änderungen sichtbar werden, müssen wir die Darstellung immer wieder neu zeichnen lassen.

Für das Beispieldesign der HTML-Seite haben wir das Bootstrap-Framework verwendet.³ Der HTML-Code der Administrationsseite ist bis auf einen kleinen Unterschied mit der Startseite identisch. Wir benötigen ein zusätzliches Canvas-Element, um die Heatmap darzustellen. Jetzt können Sie die Applikation mit dem Befehl

```
node app.js
```

starten und sich das Ergebnis in einem Browser ansehen. Am besten Sie öffnen zwei Fenster und betrachten die Seite

```
http://localhost:3000
```

und

³ <http://getbootstrap.com/>

<http://localhost:3000/admin>

nebeneinander (siehe Bild 8.7 und Bild 8.8). Dies sollte Ihnen noch einmal einen sehr guten Eindruck von der Echtzeitfähigkeit vermitteln, die die WebSocket-Technologie ermöglicht.

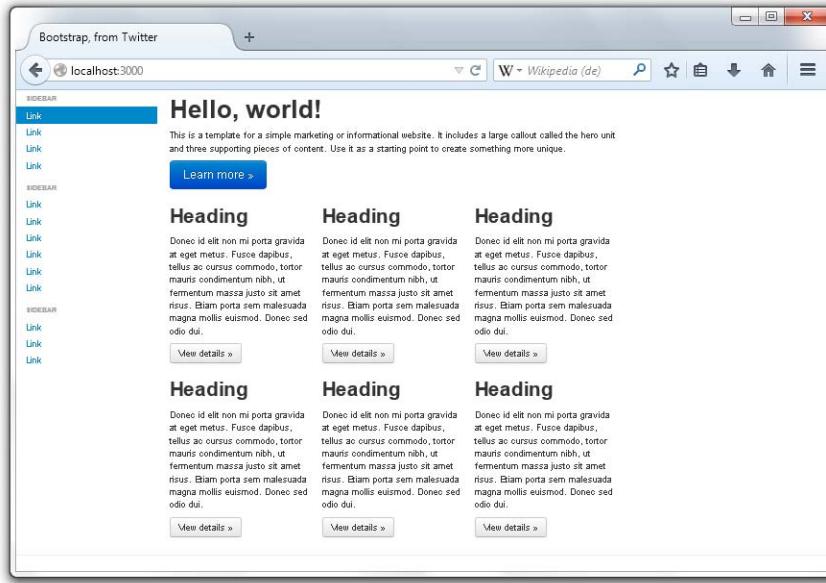


Bild 8.7 Startseite für den Benutzer

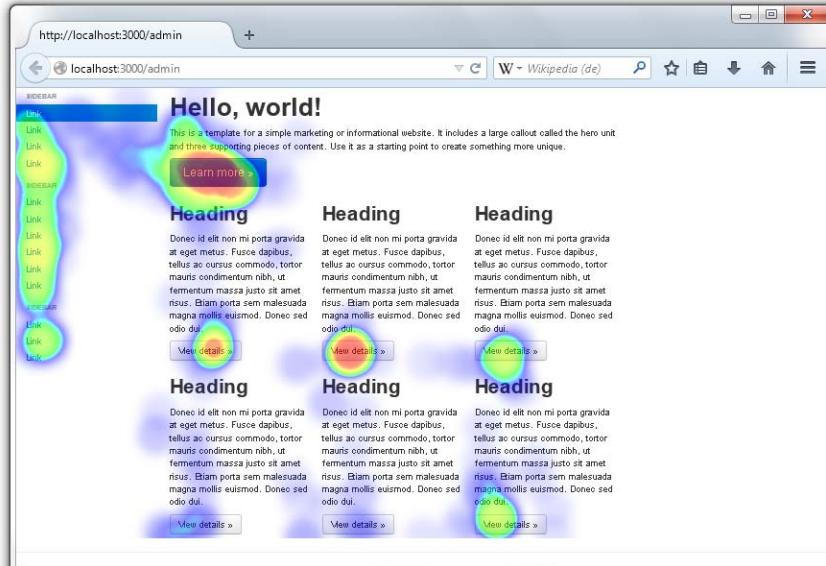


Bild 8.8 Heatmap für den Administrator

Diesen Prototypen können Sie z. B. verwenden, um Usability-Tests durchzuführen. Sie erfahren durch die Heatmap, an welchen Stellen der Webseite sich die Maus am meisten befunden hat, und können damit Antworten auf verschiedene Fragestellungen finden, wie z. B. ob die Nutzer wichtige Elemente zur Navigation finden oder welche Inhalte von besonderem Interesse sind.

Die Daten der Mauspositionen stehen in dieser Anwendung noch nicht dauerhaft auf der Serverseite zur Verfügung. Wenn also die Admin-Seite neu geladen wird, wird die Heatmap zurückgesetzt, da die Daten nur clientseitig gespeichert werden. Die Datenstruktur des JSON-Objekts, in dem die Koordinaten gespeichert werden, eignet sich jedoch optimal für eine dokumentorientierte Datenbank wie z. B. MongoDB.⁴

Diese Anwendung steht ebenfalls für Sie auf
<http://websocket101.org/>
zur Verfügung.

■ 8.4 Überwachungskamera per Webcam

1991 wurden an der Universität von Cambridge die ersten Bilder einer Webcam aufgezeichnet. Diese 128 x 128 Pixel großen Graustufenbilder wurden an Webbrowser im lokalen Netzwerk geschickt und zeigten, wie viel Kaffee noch in der berühmten „Trojan-Room-Kaffeemaschine“ vorhanden war.⁵ Die Webcam entstand aus dem Wunsch heraus, nicht zu einer leeren Kaffeemaschine laufen zu müssen. Heute, über zwei Jahrzehnte später, können Sie mit Ihrem gelernten Know-how über WebSockets ebenfalls eine sehr elegante Lösung für dieses Problem realisieren.

Die Hardware, die Sie dafür benötigen, haben Sie mit großer Wahrscheinlichkeit schon zur Hand. Ihr Smartphone, Tablet-PC oder Laptop mit integrierter Kamera bzw. Webcam wird die Bilder aufzeichnen. Zum Kontrollieren der Kaffeemaschine reicht ein Gerät mit einem HTML5-fähigen Browser aus.

Auf der Softwareseite werden wir die Anwendung aus [Abschnitt 7.2.3](#), die wir mit Express.js und WebSocket-Node programmiert haben, als Grundlage nehmen. Dadurch steht uns bereits eine Zugriffskontrolle zur Verfügung. Die Anwendung erweitern wir mit HTML5-Technologien, um auf die Kamera zugreifen zu können und die Bilder per WebSockets binnen an den Server zu schicken. Der Server verteilt diese an eingeloggte Benutzer.

Die Anwendung möchten wir ohne Plug-ins, wie Flash oder Java-Applets, realisieren. Dazu bedienen wir uns der *WebRTC*-Technologie⁶. WebRTC steht für *Real Time Communication*. Die in der Standardisierung befindliche Technologie ermöglicht es uns, per JavaScript auf

⁴ <http://www.mongodb.org/>

⁵ <http://de.wikipedia.org/wiki/Trojan-Room-Kaffeemaschine>

⁶ <http://www.webrtc.org/home/>

die Webcam und auf angeschlossene Mikrofone zuzugreifen. Zusätzlich bietet uns Web-RTC an, diese Multimediadaten per Peer to Peer (P2P) zu anderen Browsern zu schicken. Wir wollen uns in diesem Kapitel jedoch nicht näher mit P2P beschäftigen.

Da diese Anwendung eine simple Überwachungskamera darstellen soll, interessieren wir uns nur für die Bildübertragung und vernachlässigen den Ton. Die Aufnahmen sollen zur Vereinfachung nicht als Videostream, sondern als Sequenz einzelner Videoschnappschüsse an den Server geschickt werden. Außerdem möchten wir noch zusätzlich die bereits erwähnte Zugriffskontrolle nutzen, sodass sich nur berechtigte Benutzer, die sich eingeloggt haben, mit dem WebSocket-Server verbinden können. Das Starten der Webcam und das Anschauen der Überwachungsbilder soll ebenfalls nur berechtigten Personen gestattet werden.

Wir beginnen damit, die Anwendung aus [Abschnitt 7.2.3](#) zu erweitern, und stellen zunächst die maximale Frame- und Nachrichtengröße unseres WebSocket-Servers auf 64 MB, um auch ggf. hohen Bildauflösungen von Webcams gerecht zu werden (siehe [Listing 8.24](#)).

Listing 8.24 Anpassen der Frame- und Nachrichtengröße (*app.js*)

```

1 ...
2 var app = express();
3
4 var httpServer = require("http").Server(app);
5 var WebSocketServer = require("websocket").server;
6 var webSocketServer = new WebSocketServer({
7     httpServer:httpServer,
8     autoAcceptConnections:false,
9     maxReceivedFrameSize: 64*1024*1024,
10    maxReceivedMessageSize: 64*1024*1024,
11 });

```

Danach schmücken wir die Zugriffskontrolle des [Listing 7.19 \(Seite 176\)](#) aus. Als Voraussetzung für den Zugriff hatten wir dort festgelegt, dass der Client über eine gültige Session verfügen muss. Nun fügen wir die Forderung hinzu, dass sich ein Client nur mit zwei bestimmten Endpunkten verbinden kann. Für den Zugriff auf die Bilder der Überwachungskamera legen wir den Endpunkt /watch fest ([Listing 8.25, Zeile 16](#)) und für den Client, der die Kamerabilder liefern soll, den Endpunkt /cam ([Zeile 24](#)).

Listing 8.25 Zugriffskontrolle (*app.js*)

```

1 var sockets = [];
2 var camsocket;
3 webSocketServer.on('request', function(request) {
4     var sid = "";
5     for (var i = 0; i < request.cookies.length; i++) {
6         if(request.cookies[i].name === "connect.sid") {
7             sid = request.cookies[i].value;
8         }
9     };
10    var sessionID = cookieParser.signedCookie(sid, signKey);
11
12

```

```

13     store.get(sessionID, function(err, session) {
14
15         //WebSocket-Endpunkt fuer den Beobachter
16         if(request.resource == "/watch") {
17             var connection = request.accept();
18             sockets.push(connection);
19             connection.on("close", function() {
20                 delete connection;
21             });
22         }
23         //WebSocket Endpunkt fuer die Kamera
24         else if(request.resource=="/cam") {
25             //Schliessen der WebSocket-Verbindung einer evtl. bereits
26             //verbundenen Kamera
27             if(camsocket) {
28                 camsocket.close();
29             }
30             camsocket = request.accept();
31             camsocket.on("message", function(message) {
32                 if(message.type=="binary") {
33                     for(var i = 0 ; i < sockets.length;i++) {
34                         sockets[i].sendBytes(message.binaryData);
35                     }
36                 });
37                 camsocket.on("close", function() {
38                     for(var i = 0 ; i < sockets.length;i++) {
39                         sockets[i].sendUTF("Webcam closed");
40                     }
41                 });
42             }
43             else {
44                 request.reject();
45             }
46         });
47     });

```

Verfügt ein Client über eine gültige Session und möchte sich mit einem der beiden Endpunkte verbinden, akzeptieren wir die Anfrage mit der Methode `request.accept()`, die als Rückgabewert den gerade aufgebauten WebSocket hat. Bei einer Verbindung mit `/watch` speichern wir den WebSocket in einer lokalen Variablen ab ([Zeile 17](#)) und fügen diese einem dynamischen Array hinzu. Damit haben wir eine Liste aller aufgebauten Verbindungen zu diesem Endpunkt. Falls ein Client die WebSocket-Verbindung schließt, wird diese dementsprechend mit dem Befehl `delete` wieder aus dem Array gelöscht ([Zeile 19](#) bis [Zeile 21](#)). Verbindet sich ein Client mit `/cam`, speichern wir diesen WebSocket in der globalen Variablen `camsocket` ([Zeile 29](#)), denn schließlich möchten wir nur das Signal einer einzelnen Webcam verteilen. In dem Fall, dass bereits ein Client als Überwachungskamera verbunden ist, lösen wir diesen ab und schließen dazu seine WebSocket-Verbindung ([Zeile 26](#) bis [28](#)).

Das Kamerabild kann nun mithilfe einer einfachen for-Schleife an alle gelisteten Teilnehmer des sockets-Array gesendet werden ([Zeile 32](#) bis [Zeile 34](#)). Falls die Verbindung zum Webcam-Client abbricht, werden alle Betrachter mit der Textnachricht "Webcam closed" darüber benachrichtigt ([Zeile 37](#) bis [41](#)).

Nun müssen wir noch die URLs von zwei HTML-Seiten definieren, die wir benötigen, um zum einen die Bilder der Kamera aufzuzeichnen und zum Server zu schicken, und zum anderen, um diese Bilder den Betrachtern zugänglich zu machen. Da wir keine dynamischen Webinhalte erzeugen müssen, übermitteln wir zwei statische HTML-Seiten (siehe [Listing 8.26](#)).

Listing 8.26 Definieren der URLs der HTML-Seiten (*app.js*)

```

1 app.get('/cam', requiresLogin, function (req, res) {
2     res.sendFile(__dirname + "/static/cam.html")
3 });
4
5 app.get("/watch", requiresLogin, function (req, res) {
6     res.sendFile(__dirname + "/static/watch.html")
7 });

```

In der Datei *cam.html* definieren wir die clientseitige Anwendung, die die Webcam-Bilder zum Server sendet (siehe [Listing 8.27](#)). Dazu importieren wir die externe JavaScript-Bibliothek *canvas-toBlob.js*. Die Verwendung dieser JavaScript-Klasse erklären wir im weiteren Verlauf. Wir erstellen außerdem ein <video>-Tag und weisen diesem das Attribut *id* mit dem Wert *cam* sowie die Eigenschaft *autoplay* zu.

Listing 8.27 Der Webcam-Client (*cam.html*)

```

1 <html>
2     <head>
3         <title>Security Cam</title>
4         <script type="text/javascript"
5             src="/javascripts/canvas-toBlob.js"></script>
6     </head>
7     <body>
8         <h1>Security Cam</h1>
9         <video id="cam" autoplay></video>
10        <script>...</script>
11    </html>

```

Als Nächstes widmen wir uns dem JavaScript-Teil der HTML-Seite (siehe [Listing 8.28](#)). Wir öffnen einen WebSocket mit dem Endpunkt "/cam". Danach speichern wir das <video>-Tag in der Variablen *video* ab und erzeugen ein *canvas*-Element. Daraus ziehen wir den Kontext, um später darin zeichnen zu können.

Listing 8.28 JavaScript des Webcam-Clients (1) (*cam.html*)

```

1 var ws = new WebSocket("ws://" + window.location.host + "/cam");
2 var video = document.getElementById("cam");
3 var canvas = document.createElement("canvas");

```

```

4 var ctx = canvas.getContext("2d");
5
6 navigator.getUserMedia = navigator.getUserMedia ||
7             navigator.webkitGetUserMedia ||
8             navigator.mozGetUserMedia ||
9             navigator.msGetUserMedia
10
11 window.URL = window.URL ||
12             window.webkitURL ||
13             window.mozURL ||
14             window.msURL;

```

Mit `navigator.getUserMedia` können wir auf die Medienquellen, wie z. B. eine Webcam oder ein Mikrofon, zugreifen. Die Fallunterscheidungen sind derzeit noch notwendig, da das `getUserMedia`-Objekt nicht einheitlich in den Browsern implementiert ist. Das Gleiche gilt für das `window.URL`-Objekt, mit dem wir den Mediendatenstrom aus dem `navigator.getUserMedia`-Objekt in einen `<video>`-Tag einbinden können.

Weiter geht es mit der Funktion `navigator.getUserMedia()`, der wir drei Argumente übergeben (siehe [Listing 8.29](#)). Mit dem ersten Argument können Sie die Medien bestimmen, auf die Sie zugreifen möchten (Audio, Video). Dem zweiten Argument wird eine Callback-Funktion übergeben, die aufgerufen wird, wenn der Benutzer dem Zugriff auf seine Webcam bzw. auf sein Mikrofon zustimmt. Die Funktion im letzten Argument ist für den Fall, dass der Zugriff abgelehnt wird.

Listing 8.29 JavaScript des Webcam-Clients (2) (*cam.html*)

```

1 navigator.getUserMedia({video:true, audio:false}, function(stream) {
2
3     if(navigator.webkit GetUserMedia || navigator.mozGetUserMedia)
4         video.src = window.URL.createObjectURL(stream) || stream;
5     else
6         video.src = stream;
7
8     setInterval(takeSnapshot, 500);
9
10 }, function() {
11     ws.close();
12     console.log("Zugriff auf getUserMedia verweigert ");
13 });
14
15 function takeSnapshot() {
16     canvas.width = video.videoWidth;
17     canvas.height = video.videoHeight;
18     ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
19     canvas.toBlob(function(blob) {
20         ws.send(blob)
21     });
22 }

```

Falls der Benutzer den Zugriff auf seine Medienquellen erlaubt, übergeben wir ihm eine Funktion, die als Argument den Mediendatenstrom `stream` enthält. Eine darauf verweisende URL können wir Hilfe der Methode `window.URL.createObjectURL()` erzeugen und dann dem `src`-Attribut des `<video>`-Tags zuweisen. Leider existiert die Methode nicht in allen Browsern, weshalb wir eine Fallunterscheidung machen. In einigen Browsern wird keine URL benötigt, sondern `stream` kann direkt dem `src`-Attribut des `<video>`-Tags übergeben werden. Auf der Internetseite des Mozilla Developer Networks können Sie eine aktuelle Kompatibilitätsliste finden.⁷

Nun definieren wir ein Intervall, das die Methode `takeSnapshot()` alle 500 Millisekunden aufruft. Diese nimmt sich das aktuelle Bild aus dem `<video>`-Tag und zeichnet es in das `canvas`-Element, dessen Größe wir vorher festlegen und das danach in einen BLOB umgewandelt zum Server geschickt wird.

Die `toBlob()`-Methode ist momentan nur in der Firefox-Version 19 und höher implementiert.⁸ Das ist der Grund, warum wir in [Listing 8.27](#) die JavaScript-Bibliothek `canvas-toBlob.js` importiert haben, denn das ermöglicht es uns, die Methode trotzdem in anderen Browsern zu verwenden. Diese JavaScript-Klasse steht unter der MIT/X11-Lizenz und kann bei GitHub heruntergeladen werden.⁹

Wir definieren jetzt noch in der Callback-Funktion, die aufgerufen wird, falls der Benutzer den Zugriff auf seine Medienquellen ablehnt, dass der WebSocket geschlossen wird, und geben eine dazu passende Nachricht in der Konsole aus. Damit sind wir nun unserem Ziel sehr nahe. Das Einzige, was wir jetzt noch programmieren müssen, ist die Seite zum Betrachten des von der Kamera aufgezeichneten Bildes.

Für die HTML-Seite des Überwachungsclients kommt im HTML-Teil lediglich ein ``-Tag dazu, dem wir zunächst ein Standardbild übergeben, und statthen die Seite noch mit einer Überschrift aus (siehe [Listing 8.30](#)).

Listing 8.30 Der Überwachungsclient (`watch.html`)

```
1 <h1>Watch Security Cam</h1>
2 
```

Nun folgt der letzte kurze JavaScript-Teil. Hier öffnen wir eine WebSocket-Verbindung zum Endpunkt `"/watch"` und speichern das ``-Tag mit der Id `"watch"` in der Variablen `img` (siehe [Listing 8.31](#)).

Listing 8.31 JavaScript des Überwachungsclients (`watch.html`)

```
1 var ws = new WebSocket("ws://" + window.location.host + "/watch");
2 var img = document.getElementById("watch");
3
4 window.URL = window.URL ||
5             window.webkitURL ||
6             window.mozURL ||
7             window.msURL;
```

⁷ <https://developer.mozilla.org/en-US/docs/Web/API/URL.createObjectURL/>

⁸ <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement.toBlob>

⁹ <https://github.com/eligrey/canvas-toBlob.js/>

```

8
9 ws.onmessage = function(evt) {
10   if(evt.data == "close") {
11     img.src="/images/cam.png";
12     return;
13   }
14
15   window.URL.revokeObjectURL(img.src);
16   img.src = window.URL.createObjectURL(evt.data);
17 }
```

Falls eine Textnachricht mit dem Inhalt `close` empfangen wird, wissen wir, dass der WebSocket zur Webcam geschlossen wurde, und zeigen wieder das Standardbild an. Wenn es sich allerdings um ein Bild in Binärform handelt, erzeugen wir mit der Methode `createObjectURL()` eine URL aus dem BLOB, die uns zum Kamerabild führt. Da die erzeugten Daten im Cache gespeichert werden und wir diesen nicht unnötig belasten wollen, rufen wir direkt danach die Methode `revokeObjectURL()` zum Löschen auf. Damit ist unsere Überwachungskamera-Anwendung bereit für den Einsatz. Starten Sie sie mit dem Befehl:

```
node app.js
```

Nachdem Sie sich mit dem Benutzernamen „admin“ und dem Passwort „123“ auf der Seite

```
http://localhost:3000/cam
```

eingeloggt haben, werden Sie vom Browser um die Zugriffserlaubnis auf Ihre Medienquellen gebeten (siehe [Bild 8.9](#)).



Bild 8.9 Frage nach der Zugriffserlaubnis auf Medienquellen im Browser

Wenn Sie zustimmen, startet die Aufzeichnung der Webcam und Sie bekommen das Bild angezeigt (siehe [Bild 8.10](#)).

Richten Sie die Webcam z. B. auf Ihre Kaffeemaschine. Sie können nun entspannt in Ihr Büro zurückgehen und einen Browser öffnen. Die URL mit der IP Ihres Device kann z. B. so aussehen:

```
http://192.168.2.104:3000/watch
```

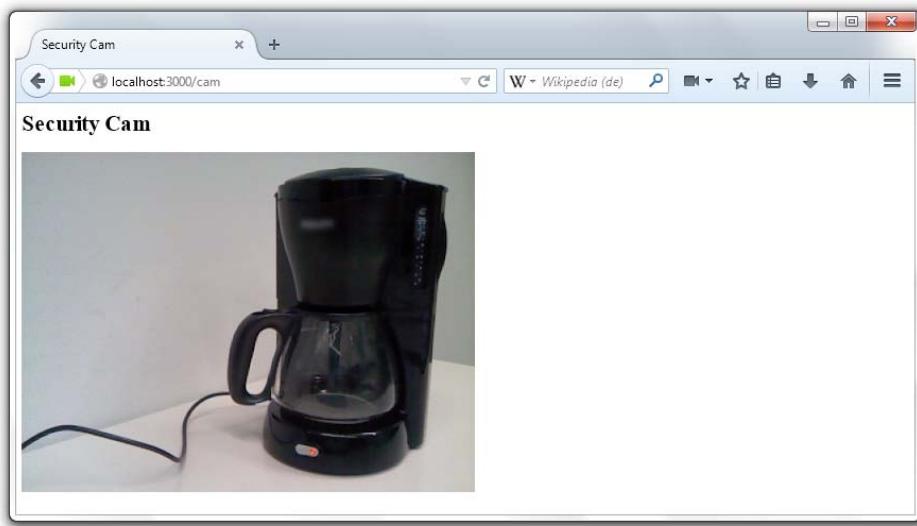


Bild 8.10 Überwachungskamera

Loggen Sie sich ein und voilà! Sie können sich den Kaffeestand von dort aus ansehen (siehe [Bild 8.11](#)).

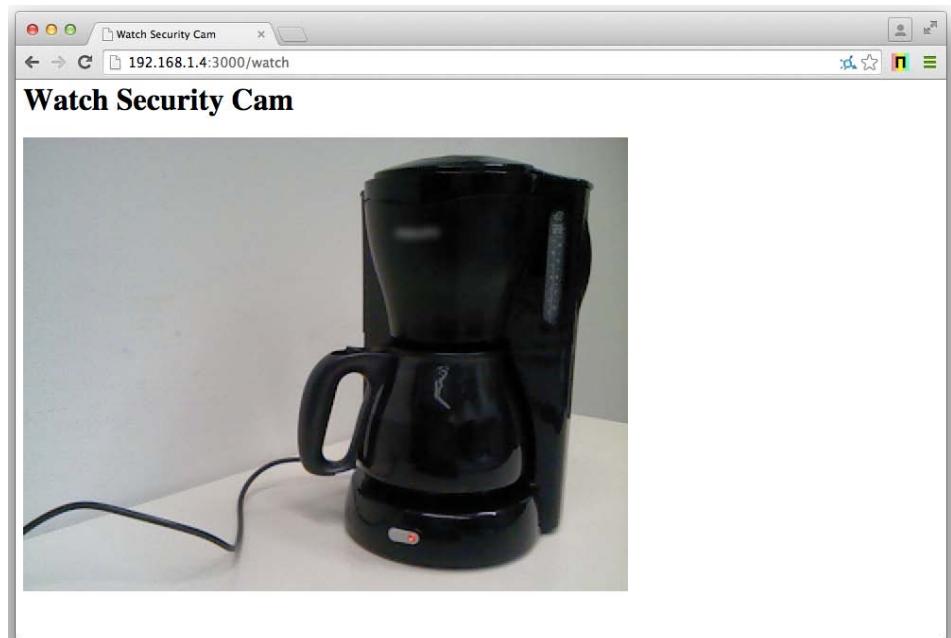


Bild 8.11 Betrachten der Bilder der Überwachungskamera

Die Aufzeichnung der Bilder über die Webcam ist derzeit unter den Browsern möglich, die die getUserMedia-API unterstützen, welche momentan nur Chrome, Firefox und Opera sind. Ansehen können Sie sich die Bilder mit allen aktuellen Versionen von Safari, Internet Explorer, Chrome und Opera.



Die komplette Anwendung, wie auch alle anderen gezeigten Beispiele, haben wir für Sie auf unserer Homepage bereitgestellt:

<http://websocket101.org/>

Schlusswort

Gratulation! Sie haben das Ende dieses Buches erreicht. An dieser Stelle angekommen, sollte das für Sie der Anfang für erfolgreich realisierte WebSocket-Projekte bedeuten. Wir hoffen, dass wir Ihnen das dazu nötige Rüstzeug vermitteln konnten, und Sie hilfreiche Einblicke sowie erforderliche Impulse durch das Lesen des Buches erhalten haben. Bei der Ausgestaltung des Buches haben wir zudem versucht, es als nützliches Nachschlagewerk anzulegen, um eventuell in Vergessenheit geratenes schnell wieder auffinden und auffrischen zu können. Wir hoffen, dass uns das insgesamt gelungen ist. Sagen Sie uns gerne, wie Sie das sehen.

Wir wollen an dieser Stelle auch nochmal unser Angebot aus [Kapitel 1](#) wiederholen. Sie sind herzlich eingeladen, sich bei Fragen, Problemen, Diskussionsbedarf und so weiter bei uns zu melden. Häufig lassen sich so auftuende Widrigkeiten schnell umschiffen. Ein Rat noch: Lassen Sie sich nicht von ersten Schwierigkeiten bei eigenen Implementierungen entmutigen. Aus unserer Erfahrung können wir sagen, dass der Teufel im Detail steckt und man sich nicht selten bei den ersten Gehversuchen im Detail-Dickicht verrennt. Ihre Ausdauer und Ihr Fleiß werden sich aus unserer Sicht aber langfristig bezahlt machen. WebSockets, als Mitglied der HTML5-Standardfamilie, werden an Relevanz zunehmen und wir werden interessante Einsatzbereiche – sicher auch aus Ihrer Feder – zu sehen bekommen.

In diesem Sinne möchten wir mit einem Zitat des griechischen Philosophen Aristoteles abschließen:

„Was man lernen muss, um es zu tun, das lernt man, indem man es tut.“

Aristoteles (384 v. Chr.–322 v. Chr.)

Oder um es mit einfacheren Worten zu formulieren: Ärmel hoch und ran!



Feedback willkommen!

Wir freuen uns von Ihnen zu hören! Ob Lob oder Kritik, ob Frage oder Anregung, ob Hinweis oder Verbesserung, wir glauben, dass das Buch von all dem profitieren kann, wenn es konstruktiv eingebracht wird. Dazu wollen wir unser Nötigstes tun und freuen uns auf Ihr Mitwirken. Aktuelles wird dabei zeitnah auf der buchbegleitenden Webseite bereitgestellt. Schauen Sie doch häufiger mal auf

<http://websocket101.org/>

vorbei. Dort finden Sie auch unsere Kontaktinformationen.



XHR-Objekt

Der durch das W3C spezifizierte XMLHttpRequest (XHR) verfügt über die folgende API [KASS14]:

Konstruktor

```
XMLHttpRequest([XMLHttpRequestOptions options])
```

Attribute

unsigned short readyState	(read only)
unsigned long timeout	
boolean withCredentials	
XMLHttpRequestUpload upload	(read only)
unsigned short status	(read only)
ByteString statusText	(read only)
XMLHttpRequestResponseType responseType	
any response	(read only)
DOMString responseText	(read only)
Document? responseXML	(read only)

Methoden

request

```
void open(ByteString method,  
          DOMString url,  
          optional boolean async = true,  
          optional DOMString? user = null,  
          optional DOMString? password = null);
```

```
void setRequestHeader(ByteString header, ByteString value);
    attribute unsigned long timeout;
    attribute boolean withCredentials;
    readonly attribute XMLHttpRequestUpload upload;

void send(optional (ArrayBufferView or Blob or Document or DOMString or
FormaData)? data = null);

void abort();
```

response

```
readonly attribute unsigned short status;
readonly attribute ByteString statusText;
ByteString? getResponseHeader(ByteString header);
ByteString getAllResponseHeaders();
void overrideMimeType(DOMString mime);
```

Event Handler

```
onreadystatechange;
onloadstart;
onprogress;
onabort;
onerror;
onload;
ontimeout;
onloadend;
```



Chrome Developer Tools auf Android

In diesem Kapitel wollen wir Ihnen eine kleine Anleitung geben, mit der Sie die Chrome Developer Tools, die es noch nicht für Android gibt, per Remote-Debugging mit Ihrem Android-Gerät nutzen können. Eine Analyse einer auf Ihrem Android-Gerät geöffneten Webseite ist mithilfe der Chrome Developer Tools des Google Chrome-Browsers auf Ihrem Rechner möglich. Dazu müssen Sie die beiden Geräte per USB verbinden. Sie brauchen also nur ein USB-Kabel und Chrome auf Ihrem Computer sowie Ihrem mobilen Endgerät.

Als Erstes müssen Sie das *USB-Debugging* auf dem Android-Gerät unter *Einstellungen → Entwickleroptionen* einschalten (siehe [Bild B.1](#)).



Bild B.1 USB-Debugging einschalten

**Hinweis**

Bedauerlicherweise wird diese Funktion nur ab der Android-Version 4.0 unterstützt.
Frühere Versionen müssen leider auf dieses Feature verzichten.

Jetzt können Sie Ihr Smartphone oder Tablet per USB-Kabel mit dem PC verbinden. Öffnen Sie danach Google Chrome auf dem Android-Gerät und rufen Sie eine beliebige Seite auf. Nun öffnen Sie ebenfalls Google Chrome auf dem Rechner und rufen die URL

chrome://inspect/#devices

auf. Es erscheint eine Seite, mit der Sie die geöffneten Tabs auf Ihrem Mobile Device untersuchen können (siehe Bild B.2).

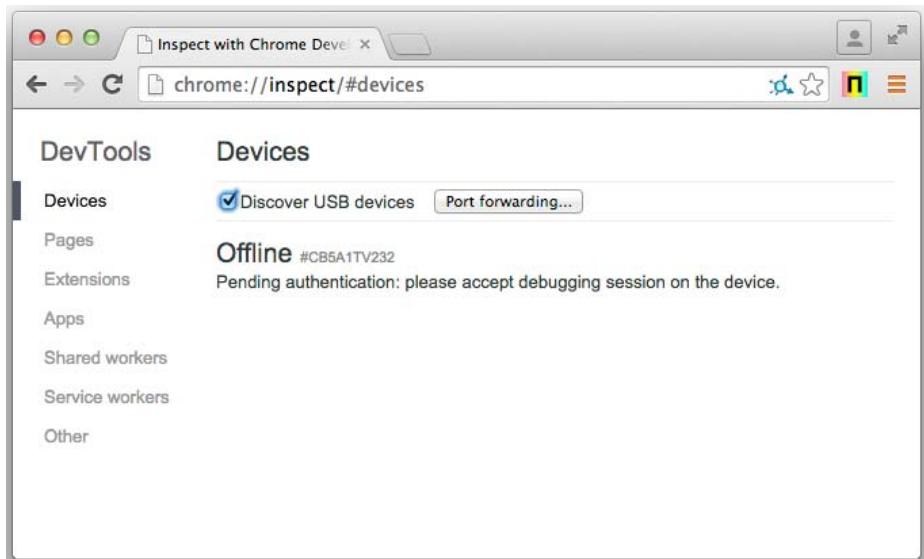


Bild B.2 Remote-Debugging-Inspect-Seite der Chrome Developer Tools

Gleichzeitig erscheint ein Dialog auf dem Android-Gerät, wo Sie bestätigen müssen, dass die Google Chrome-Anwendung Ihres Rechners die geöffneten Tabs Ihres mobilen Gerätes untersuchen darf (siehe Bild B.3).

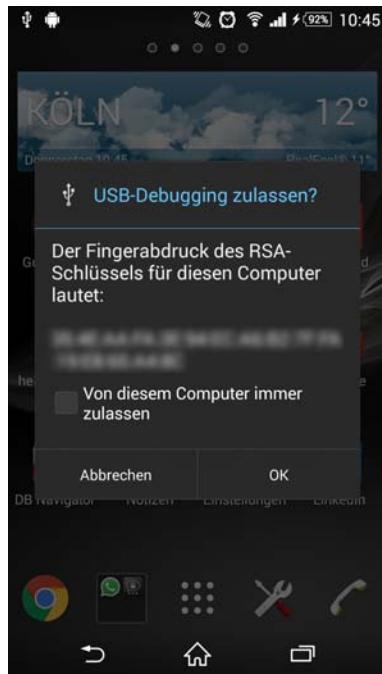


Bild B.3 Bestätigungsdialog USB-Debugging

Wenn Sie auf **OK** drücken, können Sie auf Ihrem Rechner alle geöffneten Tabs Ihres Android-Gerätes einsehen (siehe Bild B.4).

A screenshot of the "Inspect with Chrome DevTools" browser window. The address bar shows "chrome://inspect/#devices". The left sidebar lists "DevTools" with options like Devices, Pages, Extensions, Apps, Shared workers, Service workers, and Other. The main content area is titled "Devices" and shows a list with one item: "C6603 #CB5A1TV232". Underneath it, there's a card for "Chrome (38.0.2125.114)" with links for "Open tab with url", "Open", and "Impressum | Carl Hanser Verlag http://www.hanser.de/impressum/ inspect focus tab reload close".

Bild B.4 Remote-Debugging-Inspect-Seite der Chrome Developer Tools nach der Bestätigung

Wenn Sie nun unterhalb des Tabs, den Sie genauer betrachten möchten, auf *inspect* klicken, öffnet sich ein zusätzliches Fenster (siehe Bild B.5).

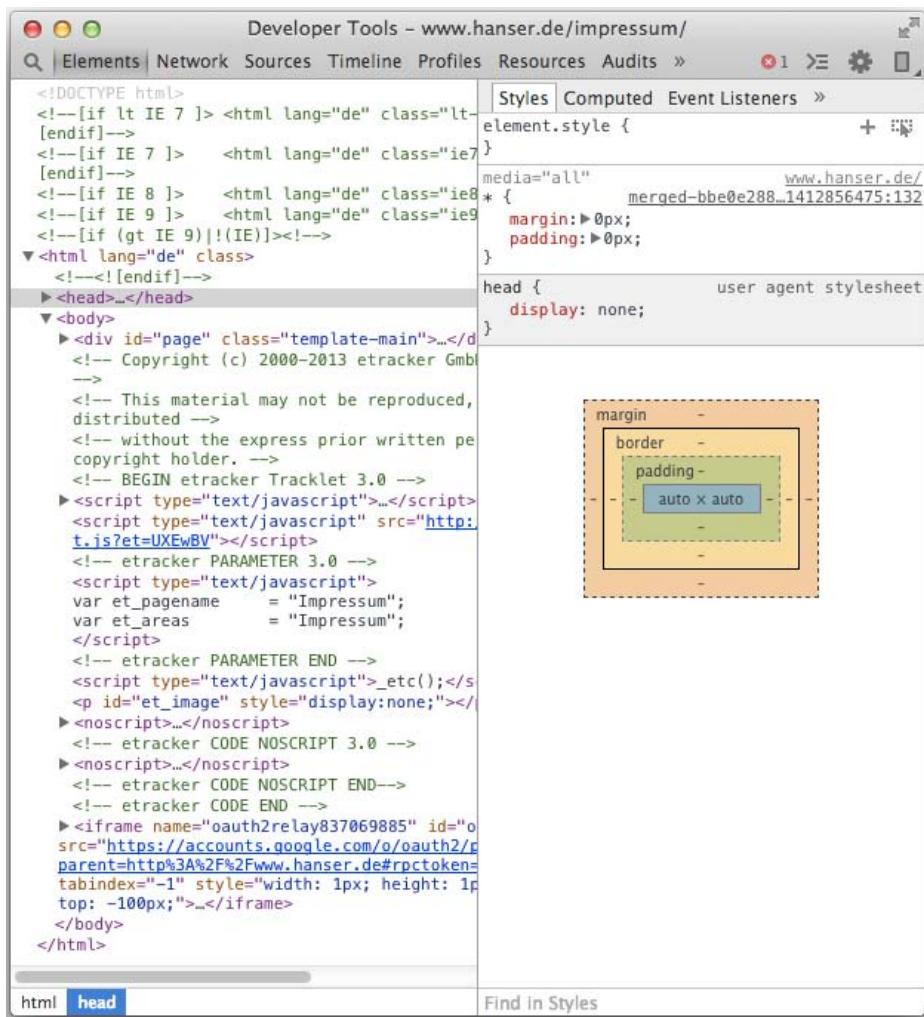


Bild B.5 Quelltextansicht einer clientseitigen Webanwendung auf einem Android-Gerät mit Chrome Developer Tools und Remote-Debugging

Mit den bekannten Developer-Tools sind Sie jetzt in der Lage, die geöffnete Webseite zu analysieren (siehe Bild B.6).



Bild B.6 Seitenansicht einer clientseitigen Webanwendung auf einem Android-Gerät mit Chrome Developer Tools und Remote-Debugging

C

Umgebungsvariablen definieren

Umgebungsvariablen sind einstellbare Variablen in einem Betriebssystem. Sie enthalten meistens Pfade und Einstellungen, die von Programmen genutzt werden können. Durch definierte Pfade kann z.B. das Terminal in Unix-Systemen oder die Windows-Eingabeaufforderung Programme direkt aufrufen, ohne dass Sie vor dem Befehl zu den Speicherorten navigieren oder den absoluten Pfad beim Aufruf angeben müssen.

Anstatt z.B. jedes Mal folgenden Befehl aufrufen zu müssen, um ein Vert.x-Programm auszuführen:

```
/home/ubuntu/vert.x/bin/vertx run MyProgram.java
```

können Sie, wenn Sie die Umgebungsvariablen richtig einstellen, folgenden Befehl aufrufen:

```
vertx run MyProgram.java
```

Da Sie damit wesentlich angenehmer arbeiten können, möchten wir in diesem Kapitel zeigen, wie Sie die Umgebungsvariablen für Windows, Mac OS und Linux definieren können. Als Beispiel nehmen wir das Programm Vert.x. Für das Play Framework oder andere Programme funktioniert dies analog.

■ C.1 Mac OS/Linux mit Bash

Um in Mac OS Umgebungsvariablen einzustellen, fügen Sie bitte in der `.bash_profile`-Datei, die sich in Ihrem *Home*-Verzeichnis befindet, folgende Zeile hinzu:

```
export PATH=$PATH:/zu/vert.x/bin
```

Hier müssen Sie natürlich den Platzhalter durch den tatsächlichen Pfad ersetzen. Speichern Sie die Datei anschließend ab. Sie können hier einen grafischen oder auch einen Texteditor aus dem Terminal wählen (z.B. vi oder nano). Alternativ ist die Eingabe auch mit folgendem Befehl möglich:

```
echo "export PATH=$PATH:/zu/vert.x/bin" >> ~/.bash_profile
```

Der Befehl echo gibt eine bestimmte Zeichenkette in der Konsole aus, die mit dem Befehl >> der Datei hinzugefügt werden kann. Achten Sie bitte darauf, dass Sie auch wirklich >> und nicht ein Zeichen zu wenig eingeben, denn mit > wird der komplette Inhalt der Datei mit der Zeichenkette des Befehls echo ersetzt.

Die Umgebungsvariable PATH wird beim nächsten Start des Terminals erneuert. Damit Sie das Terminal nicht schließen und wieder öffnen müssen, können Sie den Befehl

```
source ~/bash_profile
```

nutzen, mit dem die Umgebungsvariablen aktualisiert werden. Nun probieren Sie, vert.x auszuführen.

In den Linux-Derivaten Ubuntu und Debian funktioniert dies analog mit dem Unterschied, dass anstelle der *.bash_profile*-Datei die Datei *.bashrc* geändert werden muss. Also z. B.:

```
echo "export PATH=$PATH:/zu/vert.x/bin" >> ~/bashrc
source ~/bashrc
```

Beachten Sie, dass diese Konfiguration nur für Bash in Mac OS, Ubuntu und Debian funktioniert. Wenn die oben genannten Einstellungen fehlschlagen, schauen Sie bitte nach, wie Sie in Ihrem eingesetzten Shell-Programm Umgebungsvariablen einstellen können.

C.2 Windows

In Windows können Sie Umgebungsvariablen über die Systemeinstellungen hinzufügen. Dorthin gelangen Sie, indem Sie die *Systemsteuerung* öffnen. Klicken Sie nun auf *System und Sicherheit* (siehe Bild C.1).

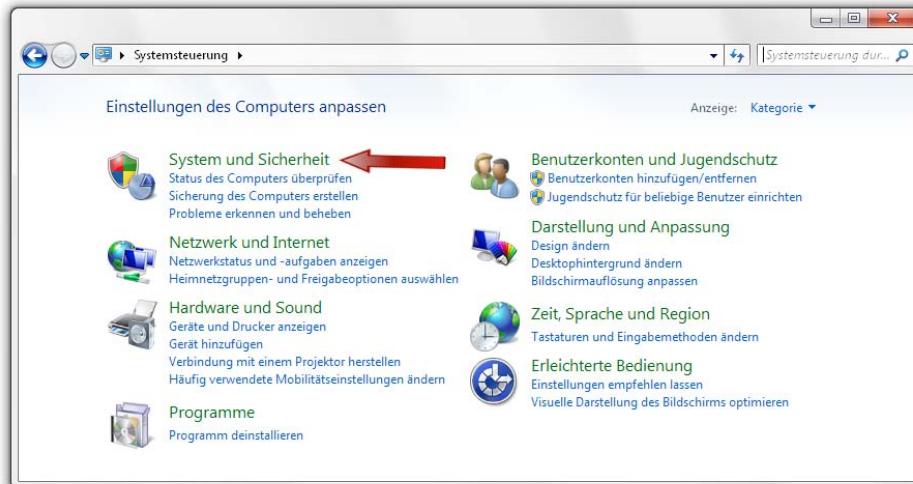


Bild C.1 Systemsteuerung

Nun klicken Sie auf das Element *System* (siehe Bild C.2).

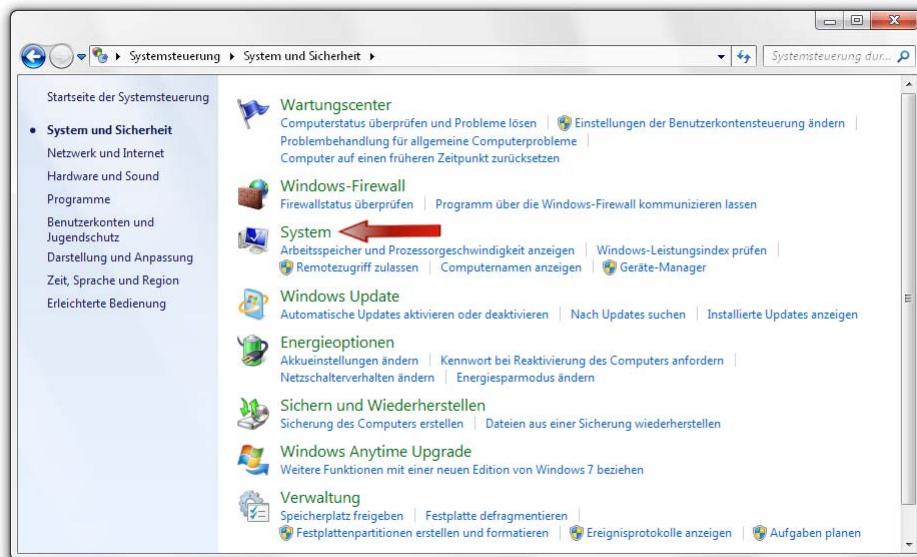


Bild C.2 System und Sicherheit

Danach klicken Sie auf *Erweiterte Systemeinstellungen* (siehe Bild C.3).

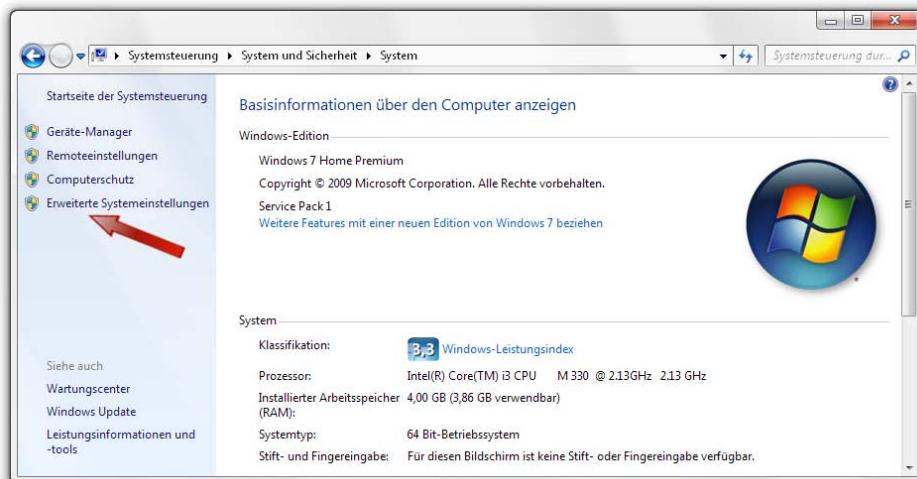


Bild C.3 System

Über dieses Fenster gelangen Sie zu den *Umgebungsvariablen*... (siehe Bild C.4).

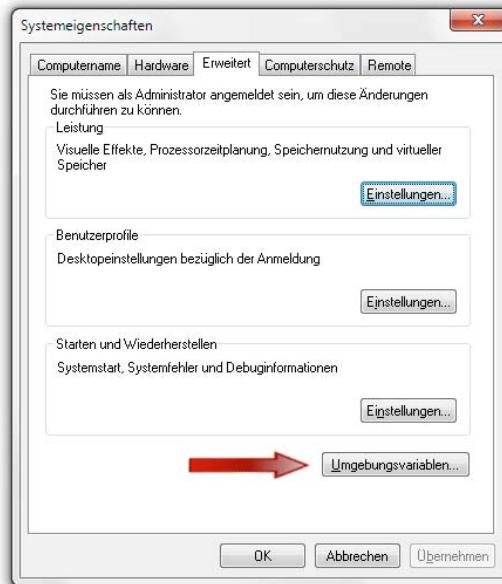


Bild C.4 Systemeigenschaften – Erweitert

Suchen Sie im Bereich *Systemvariablen* nach dem Eintrag *PATH*. Klicken Sie diesen an und betätigen Sie den *Bearbeiten...*-Button (siehe Bild C.5).

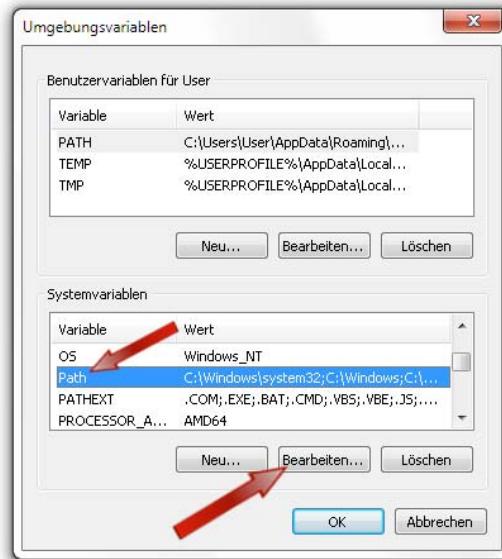


Bild C.5 Umgebungsvariablen

Fügen Sie nun am Ende des Formularfeldes mit dem Label *Wert der Variablen* zuerst ein Semikolon und ohne ein Leerzeichen dazwischen den Pfad zu Ihrer entsprechenden Anwendung ein (siehe [Bild C.6](#)).

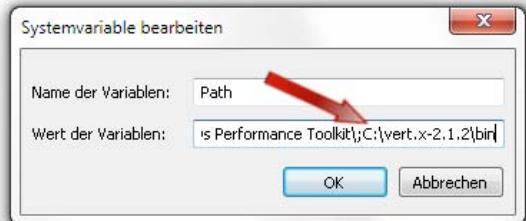


Bild C.6 Systemvariable bearbeiten

Bestätigen Sie dieses Fenster und das Fenster *Umgebungsvariablen* mit *OK*. Ihr Programm ist nun zu den Umgebungsvariablen hinzugefügt worden. Wenn Sie jetzt die Eingabeaufforderung aufrufen, kann Ihr Programm dort ausgeführt werden. Die Eingabeaufforderung rufen Sie auf, indem Sie die *Windows*-Taste drücken, im erscheinenden Suchfeld `cmd` eingeben und anschließend die *Eingabe*-Taste drücken.



Express.js

Express oder auch Express.js ist ein Web-Framework für Node.js, das unter der MIT-Lizenz¹ steht und Ihnen beim Erstellen von Webanwendungen bzw. dynamischen Webseiten unter die Arme greift. Wir werden Ihnen in diesem Kapitel zeigen, wie Sie mit Express.js Webanwendungen entwickeln und diese dann mit verschiedenen WebSocket-Modulen für Node.js, wie WebSocket.io, Socket.io oder WebSocket-Node verbinden können.

Da die Entwickler derzeit rege an der Software arbeiten, sollten Sie auf der Internetseite nachsehen, ob vielleicht schon eine neue Version veröffentlicht wurde.² Es kann leider sein, dass sich dadurch bereits die hier genutzte API der Version 4.10 geändert hat. Falls Sie Node.js und npm noch nicht installiert haben, finden Sie in [Abschnitt 6.2.1](#) eine Anleitung dazu.

Installieren Sie zuerst global den Express-Generator, damit Sie Ihre Express-Projekte später mit dem Konsolenbefehl express erstellen können:

```
npm install -g express-generator
```

Nach der Installation sollte Ihnen der Konsolenbefehl express zur Verfügung stehen. Ein schneller Test bringt Gewissheit:

```
express -V
```

Erscheint die Versionsnummer von Express.js in der Konsole, war Ihre Installation erfolgreich und der Entwicklung von Express.js-Webanwendungen steht nichts mehr im Wege. Ein neues Express.js-Projekt erstellen Sie z. B. mit dem Befehl:

```
express myapp
```

Es wird Ihnen ein vorgefertigtes Express.js-Projekt erstellt. Jetzt müssen noch die erforderlichen Module für dieses Projekt nachinstalliert werden.

```
cd myapp  
npm install
```

Ihre erste Express.js-Webanwendung kann nun mit

¹ <http://opensource.org/licenses/mit-license.php>

² <http://expressjs.com/>

```
npm start
```

gestartet werden. Öffnen Sie danach die URL

`http://localhost:3000`

in Ihrem Browser. Ihnen wird daraufhin die erste Seite angezeigt (siehe Bild D.1).

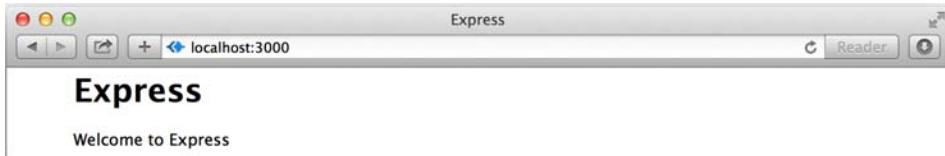


Bild D.1 Erste Express.js-Anwendung

D.1 Anatomie einer Express.js-Anwendung

In diesem Abschnitt möchten wir Ihnen die eines Express.js-Projektes, wie sie in Bild D.2 abgebildet ist, erklären.

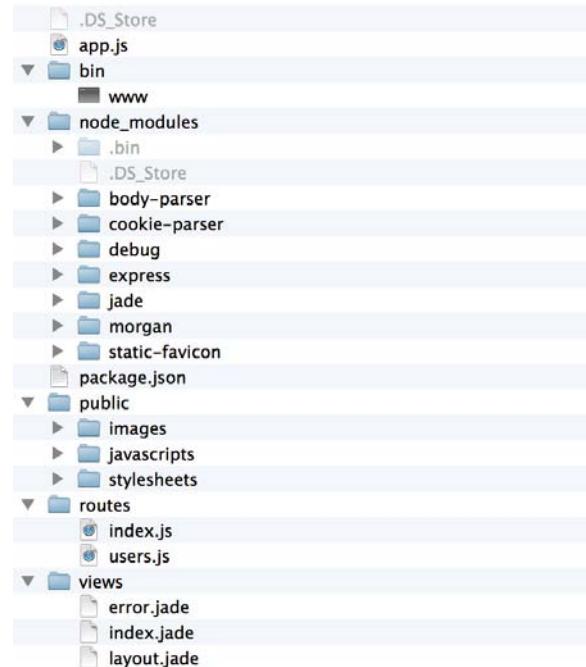


Bild D.2 Anatomie einer Express.js-Anwendung

In der Datei *app.js* befindet sich die Anwendungslogik. Dort können wir einen Webserver erstellen, den Webserver mit einem WebSocket-Server verbinden und auf Benutzereingaben reagieren.

Im Ordner *node_modules* sind die dazu erforderlichen Module abgelegt, die wir im letzten Abschnitt bei der Einrichtung eines Projekts mit dem Befehl `npm install` installiert haben. Dieser Ordner enthält auch die Datei *package.json*, in der wichtige Verwaltungsinformation für npm eingetragen sind. [Listing D.1](#) gibt einen exemplarischen Eindruck.

Listing D.1 *package.json*

```

1  {
2    "name": "Mein_Express-Projekt",
3    "version": "1.0.0",
4    "private": true,
5    "scripts": {
6      "start": "node app.js"
7    },
8    "dependencies": {
9      "express": "~4.10.0",
10     "body-parser": "~1.8.1",
11     "cookie-parser": "~1.3.3",
12     "morgan": "~1.3.0",
13     "serve-favicon": "~2.1.3",
14     "debug": "~2.0.0",
15     "jade": "~1.6.0"
16   }
17 }
```

Nun zur Erklärung des JSON-Objektes, das Sie dort sehen. In den ersten Name-Wert-Paaren können Sie dem Projekt einen Namen oder eine Versionsnummer geben. Zudem können Sie einstellen, ob die Anwendung privat oder öffentlich sein soll, indem Sie die Eigenschaft `private` sinngemäß auf `true` oder `false` setzen. Falls Sie z. B. vorhaben, Ihr Projekt nach der Fertigstellung als Node.js-Modul zu veröffentlichen, um es anderen Entwicklern zur Verfügung zu stellen, ist der Wert `false` die richtige Wahl. Diese Einstellung ermöglicht es Ihnen, dann Ihr Projekt mit dem Befehl

```
npm publish
```

auf den npm-Server hochzuladen und zu publizieren. Haben Sie den Wert auf `true` gesetzt, wird der Upload verweigert.



Hinweis

Falls Sie Ihr Projekt als eigenständiges Node.js-Modul veröffentlichen möchten, müssen Sie sich vorher auf der Internetseite

<https://www.npmjs.org/>

registrieren. Dort finden Sie auch Informationen zu weiteren Konfigurationsmöglichkeiten, die *package.json* anbietet.

Unter `scripts` sind – wie der Name schon sagt – Skripte wie z. B. `start` definierbar, womit Sie bestimmen können, welche Datei zum Starten Ihrer Anwendung ausgeführt werden soll. Wenn Sie Ihr Projekt an andere weitergeben, müssen diese dann nicht mehr nach einem bestimmten Skript suchen, sondern können das Programm direkt mit dem Befehl `npm start` ausführen. Es lassen sich aber auch Test- oder Stopp-Skripte definieren. Mehr dazu finden Sie online.³

Mit `dependencies` können Sie angeben, welche Node.js-Module von Ihrem Projekt benötigt werden. Das hat den Vorteil, dass Sie die Module im Ordner `node_modules` bei einer Veröffentlichung oder Weitergabe nicht mitliefern müssen. Alle angegebenen Module können schnell und einfach mit dem Befehl `npm install` im jeweiligen Projektordner nachträglich installiert werden.

Der nächste Ordner in der Liste ist `public`. Darin befinden sich alle statischen Dateien, wie z. B. Bilder, JavaScript- oder CSS-Dateien. Im Ordner `routes` liegen einige durch den Express-Generator erstellte Dateien, die die Anwendungslogik für bestimmte Pfade definieren. Falls Sie mit dem MVC-Entwurfsmuster vertraut sind, ist Ihnen der Begriff Controller bekannt. Da Express.js eigentlich kein MVC-Framework ist, wird dieses Konzept sozusagen nachgeahmt. Die Skripte, die Sie in diesem Ordner vorfinden, sind dementsprechend die Controller-Klassen einer Express.js-Anwendung. Wir werden allerdings zur Vereinfachung bei den Express.js-Anwendungen im Rahmen dieses Buches keinen Gebrauch davon machen und definieren stattdessen die ganze Anwendungslogik in der Datei `app.js`. Auch das voreingestellte Startskript für Express-Projekte, das sich in dem Ordner `bin` befindet, werden wir nicht benutzen. Stattdessen bestimmen wir direkt die Datei `app.js` als Startskript.

Der letzte Ordner in der Reihe ist `views`. Hier befinden sich alle Dateien, die zur visuellen Darstellung Ihrer Anwendung wichtig sind, also sozusagen die „Views“ des Projekts.

D.2 Die Anwendungslogik in der Datei `app.js`

Bevor wir mit der Implementierung unserer ersten Express.js-Anwendung beginnen, werfen wir noch einen kurzen Blick in die Datei `app.js`.

Listing D.2 Importieren der benötigten Module und erzeugen eines Express-Objektes

```

1 var express = require('express');
2 var path = require('path');
3 var favicon = require('serve-favicon');
4 var logger = require('morgan');
5 var cookieParser = require('cookie-parser');
6 var bodyParser = require('body-parser');
7
8 var routes = require('./routes/index');
```

³ <https://www.npmjs.org/doc/misc/npm-scripts.html>

```

9 var users = require('../routes/users');
10
11 var app = express();

```

In [Zeile 1](#) bis [6](#) werden Module importiert, die u. a. für die Verarbeitung von Requests wichtig sind. Benötigt wird z. B. eine Möglichkeit zum Einlesen von Cookies oder zum Parsen von bestimmten Datenformaten. Da wir die Skripte im Ordner *routes* nicht benötigen, nehmen wir die [Zeile 8](#) und [9](#) heraus. In [Zeile 11](#) wird ein Express-Objekt initialisiert, mit dem die Anwendung konfiguriert und die Verarbeitung von Requests verwaltet werden kann.

Standardmäßig enthält ein vom Express-Generator erstelltes Projekt folgende Konfiguration:

Listing D.3 Express.js-Standardkonfiguration

```

1 app.set('views', path.join(__dirname, 'views'));
2 app.set('view engine', 'jade');
3
4 app.use(logger('dev'));
5 app.use(bodyParser.json());
6 app.use(bodyParser.urlencoded({ extended: false }));
7 app.use(cookieParser());
8 app.use(express.static(path.join(__dirname, 'public')));

```

Diese Einstellungen definieren u. a. den Ordner, in dem sich die Views befinden sollen. Viele Express.js-Anwendungen nutzen *Jade* [[Hol14b](#)] als Template-Engine der Views. Das Framework hält Ihnen aber auch die Möglichkeit offen, auf andere Engines wie z. B. *ejs*⁴ oder *swig*⁵ zurückzugreifen.

Des Weiteren kann eingestellt werden, wie ausführlich die Anwendung loggen soll. Während der Entwicklung kann es sehr hilfreich sein, den Modus *dev*, welcher für *development* steht, zu nutzen, um detaillierte Informationen in den Logs zu erhalten.

Zusätzlich werden die Benutzung eines JSON-Parsers, eines Parsers für URL-Queries, eines Cookie-Parsers und ein Ordner für statische Dateien festgelegt.

Um auf Anfragen eines Clients zu reagieren, werden in Express.js sogenannte Request-Handler verwendet. Request-Handler sind Methoden des express-Objekts, das in [Listing D.2](#) erzeugt und in der Variablen *app* gespeichert wird. Da wir die Variablen *routes* und *users* entfernt haben, können wir die zwei folgenden Zeilen ebenfalls löschen:

```

app.get('/', routes.index);
app.get('/users', user.list);

```

Für unsere Beispiele ist es einfacher, eigene Request-Handler zu definieren.

Listing D.4 Definieren von Request-Handlern

```

1 app.get('/', function(req, res){
2     var message = "Hallo Welt!";

```

⁴ <http://www.embeddedjs.com/>

⁵ <https://paularmstrong.github.io/swig/>

```

3   res.render("index",{title:"Home",message:message});
4 });
5
6 app.post("/",function(req, res){
7   console.log(req.body);
8   res.redirect("/");
9 });

```

Hier werden zwei Handler implementiert, um auf einen GET- und POST-Request zu reagieren. Mit der Methode `res.render()` können Sie ein View-Template auswählen, das gerendert werden soll, und diesem Variablen übergeben.

Jetzt können Sie die Anwendung eigentlich schon starten. Eine kleine Änderung nehmen wir allerdings noch vor, die später für das Einbinden eines WebSocket-Servers wesentlich ist. Wir importieren ergänzend das `http`-Modul und erzeugen einen HTTP- bzw. Webserver. Dabei wird diesem das `express`-Objekt, das sich in der Variablen `app` befindet, übergeben. Abschließend muss lediglich noch der Port, auf dem die Anwendung ausgeführt werden soll, festgelegt werden.

Listing D.5 Erzeugung eines Servers und Zuweisung der Express-Anwendung

```

1 var app = express();
2 var httpServer = require('http').Server(app);
3 ...
4 httpServer.listen(3000, function(){
5   console.log("Der Express-Server laeuft auf dem Port 3000");
6 })

```

■ D.3 Die Jade-Template-Engine

In diesem Kapitel geben wir Ihnen einen kurzen Einblick in die Funktionsweise von Jade, einer Template-Engine von Express.js.

Jade bietet Ihnen eine vereinfachte Schreibweise, um HTML-Tags zu definieren. Auf schließende Tags wird dabei verzichtet. Stattdessen haben die Einrückungen eine entsprechende Bedeutung. Selbstverständlich können Sie auch Variablen, die aus der Anwendungslogik (`app.js`) übergeben werden, darstellen. In [Listing D.4 \(Zeile 3\)](#) übergeben wir der View zwei Variablen. Diese können dann z. B. wie in [Listing D.6 \(Zeile 4 bis 8\)](#) platziert werden.

Listing D.6 Einfaches Beispiel einer Jade-Datei

```

1 doctype html
2 html
3   head
4     title= title
5     link(rel='stylesheet', href='/stylesheets/style.css')
6   body
7     div#container

```

```

8      p Das ist meine Nachricht: #{message}
9      script.
10     console.log("Hallo Welt!");

```

Das anschließend gerenderte HTML sieht dann folgendermaßen aus:

Listing D.7 Das Ergebnis eines Jade-Codes in HTML

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Home</title>
5    <link rel="stylesheets" href="/stylesheets/style.css" />
6  </head>
7  <body>
8    <div id="container">
9      <p>Das ist meine Nachricht: Hallo Welt!</p>
10   </div>
11   <script>
12     console.log("Hallo Welt!");
13   </script>
14 </body>
15 </html>

```

Sie haben bei Jade auch die Möglichkeit, Layouts zu erstellen und diese zu erweitern. Im Ordner *views* ist z. B. schon eine Layout-Datei mit dem Namen *layout.jade* vorhanden.

Listing D.8 Jade-Layout-Datei

```

1 doctype html
2 html
3   head
4     title= title
5     link(rel='stylesheet', href='/stylesheets/style.css')
6   body
7     block content

```

Mit dem Schlüsselwort **block** können Sie Stellen definieren, die später erweitert werden. Die Datei kann dann z. B. wie folgt in einer anderen *jade*-Datei erweitert werden:

Listing D.9 Erweiterte Jade-Layout-Datei

```

1 extends layout
2
3 block content
4   h1= title
5   p Willkommen bei #{title}

```

Die Erweiterung der Datei *layout.jade* lässt sich also mit dem Befehl `extends layout` realisieren. Mit `block content` bestimmen Sie, welcher Bereich bzw. Block erweitert werden soll.

■ D.4 Express.js mit WebSocket-Servern verbinden

Wir zeigen Ihnen nun, wie Sie einen WebSocket-Server erzeugen, der in einer gemeinsamen Instanz mit dem erzeugten Webserver und der Express-Anwendung ausgeführt wird. Dazu müssen Sie zunächst ein WebSocket-Modul installieren, z. B. Socket.io:

```
npm install socket.io
```

Speichern Sie den erzeugten Webserver in eine Variable ab, damit Sie ihn dem WebSocket-Server anschließend zuweisen können.

Listing D.10 Vereinigung von Webserver und Socket.io-Server

```
1 var app = express();
2 ...
3 var httpServer = require("http").Server(app);
4 var io = require("socket.io").listen(httpServer);
5
6 httpServer.listen(3000, function(){
7   console.log("Der Express-Server laeuft auf dem Port 3000");
8 });
```

Damit sich die beiden Server denselben Port teilen, wird der Webserver an die `listen()`-Methode des WebSocket-Moduls übergeben. Ähnliches haben wir in den Beispielen aus [Abschnitt 6.2](#) durchgeführt, nur ohne Express.js. Wenn Sie z. B. den WebSocket-Server von WebSocket.io einbinden möchten, sieht das so aus:

Listing D.11 Vereinigung von Webserver und WebSocket.io-Server

```
1 var app = express();
2 ...
3 var httpServer = http.createServer(app);
4 var ws = require("websocket.io");
5 var webSocketServer = ws.attach(httpServer);
6
7 httpServer.listen(3000, function(){
8   console.log("Der Express-Server laeuft auf dem Port 3000");
9 });
```

Die Konfiguration ist auch bei dem WebSocket-Node-Modul ähnlich:

Listing D.12 Vereinigung von Webserver und WebSocket-Node-Server

```
1 var app = express();
2 ...
3 var httpServer = require('http').Server(app);
4 var WebSocketServer = require('websocket').server;
5
6 var wsServer = new WebSocketServer({
```

```
7     httpServer:httpServer,
8     autoAcceptConnections:false
9 });
10
11 httpServer.listen(3000, function(){
12     console.log("Der Express-Server laeuft auf dem Port 3000");
13 })
```

Nun können Sie die Anwendung starten, indem Sie den Befehl

```
node app.js
```

eingeben. Wenn Sie die Anwendung lieber mit npm starten möchten, müssen Sie die Eigenschaft `start` in der Datei `package.json`, wie in [Listing D.1](#) gezeigt, anpassen.

Literatur

- [Age14] AGENDALESS CONSULTING: *Supervisor: A Process Control System*. Webseite. <http://supvisord.org/>. Version: Oktober 2014
- [Ama14] AMAZON: *Amazon Web Services*. Webseite. <https://aws.amazon.com/>. Version: 2014
- [Aut12] AUTOBAHN: *WebSocket Implementation Test Report*. Webseite. <http://autobahn.ws/testsuite/reports/clients/index.html>. Version: Oktober 2012
- [CO08] CROCKER, D. (Hrsg.); OVERELL, P.: *Augmented BNF for Syntax Specifications: ABNF*. IETF Network Working Group RFC 5234. Version: Januar 2008. <https://tools.ietf.org/html/rfc5234/>
- [Dev14a] DEVERIA, A.: *Can I use Server-sent events?* Webseite. <http://caniuse.com/#feat=eventsource>. Version: 2014
- [Dev14b] DEVERIA, A.: *Can I use SPDY networking protocol?* Webseite. <http://caniuse.com/#feat=spdy>. Version: 2014
- [Dev14c] DEVERIA, A.: *Can I use Web Sockets?* Webseite. <http://caniuse.com/#feat=websockets>. Version: 2014
- [DR08] DIERKS, T.; RESCORLA, E.: *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF Network Working Group RFC 5246. Version: August 2008. <https://tools.ietf.org/html/rfc5246/>
- [DS13] DUSSEAU, L.; SNELL, J.: *PATCH Method for HTTP*. IETF RFC 5789. Version: März 2013. <https://tools.ietf.org/html/rfc5789/>
- [Eil12] EILERS, C.: *HTML5 Security*. EPUB. <http://entwickler.de/press/HTML5-Security/>. Version: Mai 2012
- [Erk12] ERKKILÄ, J.-P.: *WebSocket Security Analysis*. Aalto University School of Science. <http://juerkkil.iki.fi/files/writings/websocket2012.pdf>. Version: 2012
- [FGM⁺99] FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1*. IETF Network Working Group RFC 2616. <https://tools.ietf.org/html/rfc2616/>. Version: Juni 1999
- [FHBH⁺99] FRANKS, J.; HALLAM-BAKER, P.; HOSTETLER, J.; LAWRENCE, S.; LEACH, P.; LUOTONEN, A.; STEWART, L.: *HTTP Authentication: Basic and Digest Access*

- Authentication.* IETF Network Working Group RFC 2617. Version: Juni 1999.
<https://tools.ietf.org/html/rfc2617/>
- [Fie00] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine.
<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Version: 2000
- [FKK96] FREIER, A. O.; KARLTON, P. ; KOCHER, P. C.: *The SSL Protocol: Version 3.0*. IETF Transport Layer Security Working Group Internet-Draft. Version: November 1996. [https://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00/](https://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00)
- [FM11a] FETTE, I.; MELNIKOV, A.: *The WebSocket Protocol*. IETF RFC 6455. Version: December 2011. <https://tools.ietf.org/html/rfc6455/>
- [FM11b] FETTE, I.; MELNIKOV, A.: *The WebSocket protocol:*
draft-ietf-hybi-thewebsOCKETprotocol-17. IETF HyBi Working Group Internet-Draft. Version: September 2011.
[https://tools.ietf.org/html/draft-ietf-hybi-thewebsOCKETprotocol-17/](https://tools.ietf.org/html/draft-ietf-hybi-thewebsOCKETprotocol-17)
- [GHM⁺07] GUDGIN, M. (Hrsg.); HADLEY, M. (Hrsg.); MENDELSONN, N. (Hrsg.); MOREAU, J.-J. (Hrsg.); NIELSEN, H. (Hrsg.) E; KARMARKAR, A. (Hrsg.) ; LAFON, Y. (Hrsg.): *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. Version: April 2007. <https://www.w3.org/TR/soap12-part1/>
- [Gro03] GROSSMAN, J.: *Cross-Site Tracing (XST) - The new techniques and emerging threats to bypass current web security measures using TRACE and XSS*. E-Book. http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf. Version: Januar 2003
- [HCB⁺11] HUANG, L.-S.; CHEN, E. Y.; BARTH, A.; RESCORLA, E. ; JACKSON, C.: *Talking to Yourself for Fun and Profit*. Web 2.0 Security and Privacy 2011, W2SP 2011. Oakland, California, USA. <http://w2spconf.com/2011/papers/websocket.pdf>. Version: Mai 2011
- [Hic09] HICKSON, I. (Hrsg.): *The Web Socket protocol:*
draft-hixie-thewebsOCKETprotocol-00. IETF Network Working Group Internet-Draft. Version: Januar 2009.
<https://tools.ietf.org/html/draft-hixie-thewebsOCKETprotocol-00/>
- [Hic10] HICKSON, I. (Hrsg.): *The Web Socket protocol:*
draft-hixie-thewebsOCKETprotocol-76. IETF Internet-Draft. Version: Mai 2010.
<https://tools.ietf.org/search/draft-hixie-thewebsOCKETprotocol-76>
- [Hic12a] HICKSON, I. (Hrsg.): *Server-Sent Events*. W3C Candidate Recommendation. Version: Dezember 2012. <https://www.w3.org/TR/eventsource/>
- [Hic12b] HICKSON, I. (Hrsg.): *The WebSocket API*. W3C Candidate Recommendation. Version: September 2012. <https://www.w3.org/TR/websockets/>
- [Hol14a] HOLOWAYCHUK, TJ: *Express - Fast, unopinionated, minimalist web framework for Node.js*. Webseite. <http://expressjs.com/>. Version: 2014
- [Hol14b] HOLOWAYCHUK, TJ: *jade - Node Template Engine*. Webseite. <http://jade-lang.com/>. Version: 2014
- [IEE12a] IEEE: *Standard for Ethernet*. Standard.
<https://standards.ieee.org/about/get/802/802.3.html>. Version: Dezember 2012

- [IEE12b] IEEE: *Standard for Information technology – Telecommunications and information exchange between systems Local and metropolitan area networks – Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. Standard. <https://standards.ieee.org/about/get/802/802.11.html>. Version: März 2012
- [ISO94] ISO/IEC: *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. International Standard 7498-1. Version: November 1994. [http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip)
- [Jos06] JOSEFSSON, S.: *The Base16, Base32, and Base64 Data Encodings*. IETF Network Working Group RFC 4648. Version: Oktober 2006. <https://tools.ietf.org/html/rfc4648/>
- [Joy14] JOYENT, INC: *Node.js*. Webseite. <https://nodejs.org/>. Version: 2014
- [KASS14] KESTEREN, A. v. (Hrsg.); AUBOURG, J. (Hrsg.); SONG, J. (Hrsg.) ; STEEN, H. R. M. (Hrsg.): *XMLHttpRequest Level1*. W3C Working Draft. Version: Januar 2014. <https://www.w3.org/TR/XMLHttpRequest>
- [Kes14] KESTEREN, A. v. (Hrsg.): *Cross-Origin Resource Sharing*. W3C Recommendation. <https://www.w3.org/TR/cors/>. Version: Januar 2014
- [Kup10] KUPPAN, L.: *Attacking with HTML5*. Journal, Attack & Defense Labs. <http://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-wp.pdf>. Version: Oktober 2010
- [Lub10] LUBBERS, P.: *How HTML5 Web Sockets Interact With Proxy Servers*. InfoQ Artikel. <http://www.infoq.com/articles/Web-Sockets-Proxy-Servers/>. Version: März 2010
- [McC12] McCORMACK, C.: *WEB IDL (Second Edition)*. W3C Candidate Recommendation. Version: April 2012. <https://www.w3.org/TR/WebIDL/>
- [MML⁺14] MELNIKOV, A.; MESNIL, J.; LUNDSTEDT, M.; HAPNER, M.; GANSTERER, P.; RADEMAKERS, P.; BATUM, P.; COVER, R.; OBERSTEIN, T.; RICHARDSON, T.; RAUSCHENBACH, U.; IESG HYBI WG ; OMNA: *WebSocket Protocol Registries*. Webseite. <https://www.iana.org/assignments/websocket/websocket.xhtml>. Version: Oktober 2014
- [Moz14] MOZILLA DEVELOPER NETWORK: *WebSockets*. Webseite. <https://developer.mozilla.org/en-US/docs/WebSockets/>. Version: 2014
- [MS14] MCKELVEY, B.; SANSOY, S.: *WebSocket-Node*. GitHub. <https://github.com/Worlize/ WebSocket-Node/>. Version: Oktober 2014
- [Ngi14] NGINX: *WebSocket proxying*. Webseite. <http://nginx.org/en/docs/http/websocket.html>. Version: 2014
- [Ope14] OPEN SOURCE INITIATIVE: *Open Source Initiative: Common Development and Distribution License (CDDL-1.0)*. Webseite. <http://opensource.org/licenses/cddl1.php>. Version: 2014
- [Ora14a] ORACLE CORPORATION: *Java Community Process – Community Development of Java Technology Specifications*. Webseite. <https://www.jcp.org/>. Version: 2014

- [Ora14b] ORACLE CORPORATION: *Java Community Process: Introduction Timeline*. Webseite. <https://jcp.org/en/introduction/timeline>. Version: 2014
- [Pla14] PLAY: *Play Documentation: Using the Play 2.0 console*. Webseite. <https://www.playframework.com/documentation/2.0/PlayConsole>. Version: 2014
- [Rau14] RAUCH, G.: *Introducing Socket.IO 1.0*. Webseite. <http://socket.io/blog/introducing-socket-io-1-0/>. Version: Mai 2014
- [Rus06] RUSSELL, A.: *Infrequently Noted – Comet: Low Latency Data for the Browser*. Blog-Post. <https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>. Version: März 2006
- [RWDN07] RUSSELL, A.; WILKINS, G.; DAVIS, D. ; NESBIT, M.: *The Bayeux Specification: Bayeux Protocol – Bayeux 1.0.0*. The Dojo Foundation, Request for Comments. <http://svn.cometd.org/trunk/bayeux/bayeux.html>. Version: 2007
- [Sch11] SCHMIDT, M.: *HTML5 web security*. Artikel, Compass Security AG. http://www.csnc.ch/en/modules/news/news_0066.html_533560828.html. Version: Dezember 2011
- [Sch13] SCHNEIDER, Christian: *Cross-Site WebSocket Hijacking (CSWSH)*. Blögeintrag. <http://www.christian-schneider.net/CrossSiteWebSocketHijacking.html>. Version: August 2013
- [Sha12] SHAH, S.: *HTML5 Top 10 Threats Stealth Attacks and Silent Exploits*. Artikel, blackhatUSA2012. https://media.blackhat.com/bh-us-12/Briefings/Shah/BH_US_12_Shah_Silent_Exploits_WP.pdf. Version: März 2012
- [Sim10] SIMPSON, K.: *Defining Safer JSON-P*. Webseite. <http://www.json-p.org/>. Version: 2010
- [SST12] SHEMA, M.; SHEKYAN, S. ; TOUKHARIAN, V.: *Hacking with WebSockets*. Präsentation, blackhatUSA2012. http://media.blackhat.com/bh-us-12/Briefings/Shekyan/BH_US_12_Shekyan_Toukharian_Hacking_Websocket_Slides.pdf. Version: 2012
- [sta14] STACK OVERFLOW: *What browsers support HTML5 WebSocket API?* Forenbeitrag. <https://stackoverflow.com/questions/1253683/what-browsers-support-html5-websocket-api>. Version: April 2014
- [Swi11] SWINEHART, C.: *Nginx Digest Authentication module*. GitHub. <https://github.com/samizdatco/nginx-http-auth-digest/blob/master/readme.rst>. Version: November 2011
- [SXM⁺00] STEWART, R.; XIE, Q.; MORNEAULT, K.; SHARP, C.; SCHWARZBAUER, H.; TAYLOR, T.; RYTINA, I.; KALLA, M.; ZHANG, L. ; PAXSON, V.: *Stream Control Transmission Protocol*. IETF Network Working Group RFC 2960. Version: Oktober 2000. <https://tools.ietf.org/html/rfc2960>
- [The10] THE CHROMIUM PROJECTS: *SPDY*. Webseite. <https://dev.chromium.org/spdy/>. Version: 2010
- [The14] THE DOJO FOUNDATION: *Welcome to CometD Project @ The Dojo Foundation*. Webseite. <http://cometd.org/>. Version: 2014

- [TLM⁺14] TOUCH, J.; LEAR, E.; MANKIN, A.; KOJO, M.; ONO, K.; STIEMERLING, M.; EGGERT, L.; MELNIKOV, A.; EDDY, W.; KOHLER, E.; MANKIN, A.: *Service Name and Transport Protocol Port Number Registry*. Webseite.
<http://www.iana.org/assignments/service-names-port-numbers/>. Version: Oktober 2014
- [TY13] TAMPLIN, J.; YOSHINO, T.: *A Multiplexing Extension for WebSockets: draft-ietf-hybi-websocket-multiplexing-11*. IETF HyBi Working Group Internet-Draft. Version: Juli 2013.
<https://tools.ietf.org/html/draft-ietf-hybi-websocket-multiplexing-11>
- [W3C14] W3C: *HTML5: Take Control – Your Web, Your Logo*. Webseite.
<http://www.w3.org/html/logo/>. Version: 2014
- [WSM13] WANG, V.; SALIM, F.; MOSKOVITS, P.: *The Definitive Guide to HTML5 WebSocket – Build real-time applications with HTML5*. Apress, 2013
- [Yer03] YERGEAU, F.: *UTF-8, a transformation format of ISO 10646*. IETF Network Working Group RFC 3629. Version: November 2003.
[https://tools.ietf.org/html/rfc3629/](https://tools.ietf.org/html/rfc3629)
- [Yos12] YOSHINO, T.: *WebSocket Per-frame Compression: draft-ietf-hybi-websocket-perframe-compression-04*. IETF HyBi Working Group Internet-Draft. Version: Mai 2012. [https://tools.ietf.org/html/draft-ietf-hybi-websocket-perframe-compression-04/](https://tools.ietf.org/html/draft-ietf-hybi-websocket-perframe-compression-04)
- [Yos14] YOSHINO, T.: *Compression Extensions for WebSocket draft-ietf-hybi-permessage-compression-18*. IETF HyBi Working Group Internet-Draft. Version: Mai 2014.
[https://tools.ietf.org/html/draft-ietf-hybi-permessage-compression-04/](https://tools.ietf.org/html/draft-ietf-hybi-permessage-compression-04)
- [Zim12] ZIMMERMANN, Robin: *HTML5 WebSocket Security is Strong*. kaazing Blogbeitrag.
<http://blog.kaazing.com/2012/02/28/html5-websocket-security-is-strong/>. Version: Februar 2012

Stichwortverzeichnis

Symbole

100Base-TX [11](#)

802.11 [11](#)

802.3 [11](#)

A

ABNF → angereichert Backus-Naur-Form
 Add-on [2](#)
 Administrationsseite [219](#)
 Ajax [26](#), [172](#)
 aktive Inhalte [158](#)
 Alexa Top [187](#)
 Amazon Web Services [148](#), [153](#)
 Android [235](#)
 angereichert Backus-Naur-Form [79f.](#)
 Anmeldedialog [162](#)
 Annotation [181](#)
 Anwendungsschicht [6f.](#), [12](#)
 Apache-Webserver [185](#)
 Application Layer ⇒ Anwendungsschicht
 Applikationsserver [134](#)
 ArrayBuffer [92](#), [94f.](#)
 ArrayBufferView [92](#), [94](#)
 ASCII-Zeichen [57](#)
 Auslastung [147f.](#), [151](#)
 Authentifizierung [162](#)
 Authentizität [159](#)
 AWS → Amazon Web Services

B

Base64-Codierung [16](#), [37](#), [163](#)
 Bash [241](#)
 Bayeux-Protokoll [30](#)
 Betaversion [134](#)
 BiDirectional or Server-Initiated HTTP [36](#), [78](#),
 [91](#)
 bidirektional [35](#)
 Bild [15](#), [116](#)
 Binärdaten [16](#), [47](#), [76](#), [94](#), [113](#)

binäre Codierung [14](#)

Binary Frame [119](#)

Binary Large Object → BLOB

Bitübertragungsschicht [6](#)

BLOB [92](#), [94](#), [116](#), [226](#)

Bösartige Services [194](#)

Bootstrap [219](#)

Botnetz [190](#)

Browserunterstützung [76](#)

Buffer [91](#), [93](#)

BufferedReader [15](#)

Button [115](#), [207](#), [213](#)

C

CA → Zertifizierungsstelle
 Cache [22](#), [163](#), [198](#)
 Caching-Poisoning [184](#), [187](#)
 Callback-Funktion [103](#), [149](#), [151f.](#), [176](#)
 canvas [95](#)
 <canvas>-Tag [206](#)
 Cascading Style Sheets [158](#), [209](#)
 Chat [211](#)
 Chrome Developer Tools [68](#), [117](#), [235](#)
 – Analyse [69](#)
 – Installation [68](#)
 Chrome Network Internals [70](#)
 Clamp [96](#)
 Client [13](#), [75](#)
 Close-Event [96](#)
 Close-Frame [41](#), [49](#), [53](#), [97](#)
 – Statuscode [51](#), [96](#)
 – Statuscodes [51](#)
 Closing-Handshake [53](#)
 CoffeeScript [119](#)
 Comet [29](#)
 Connectivity-Gruppe [2](#), [30](#)
 Continuation Frames [119](#)
 Control-Frames [48](#)
 Cookie [122](#), [166f.](#), [172](#), [176](#), [193](#)

- CORS → Cross-Origin Resource Sharing
- Cross-Origin-Angriff [192](#)
- Cross-Origin Resource Sharing [29](#)
- Cross-Origin-Zugriffe [159](#)
- Cross-Site-Request-Forgery [159, 193](#)
- Cross-Site-Scripting [158 f., 188, 190, 194](#)
- Cross-Site-Tracing [19](#)
- Cross-Site WebSocket Hijacking [193](#)
- CSRF → Cross-Site-Request-Forgery
- CSS → Cascading Style Sheets
- D**
 - Darstellungsschicht [6 f.](#)
 - Data Link Layer ⇒ Sicherungsschicht
 - Datei-Handler [155](#)
 - Datenübertragung [92](#)
 - Datenframes [40, 45](#)
 - Datenkompression [91](#)
 - Datenvolumen-Overhead [16](#)
 - Deflate-Algorithmus [91](#)
 - Demaskierung [44](#)
 - Denial of Service [187, 188, 195](#)
 - Dienst [9](#)
 - DNS [11](#)
 - DNS-Anfragen [71](#)
 - Domain [16](#)
- E**
 - Early Draft [132, 134](#)
 - EC2 [153](#)
 - echo [242](#)
 - Echo-Server [58, 147](#)
 - Echtzeit-Kommunikation [202](#)
 - Echtzeit-Remote-Shell [191](#)
 - Echtzeitfähigkeit [25](#)
 - Eclipse [127](#)
 - ECMAScript → JavaScript
 - Endpunkt [149 f., 205](#)
 - Entführung der Client-Authentifizierung [193](#)
 - Erweiterungen → Extensions
 - Escape-Sequenz [15](#)
 - Ethernet [11](#)
 - Event-Server [202](#)
 - Executive Committee [132](#)
 - Express.js [216, 247](#)
 - app.js [250](#)
 - Formularbasierte Authentifizierung [168, 172, 176](#)
 - Instanz [254](#)
 - Layout [253](#)
 - Methode
 - cookie.parse() [174](#)
- F**
 - Fallback [33, 79, 109, 204](#)
 - Fehlerbegründung [85](#)
 - Fehlercode [85](#)
 - Fernbedienung [202](#)
 - Fiddler [60](#)
 - Analyse [66](#)
 - Einrichtung [61](#)
 - Installation [61](#)
 - Protokollierung [65](#)
 - FIN-Flag [40, 42](#)
 - Final Release [133](#)
 - Firewall [183 f.](#)
 - Formularbasierte Authentifizierung [166](#)
 - Fragment [17, 40](#)
 - Fragmentierung [42](#)
 - Framegröße [113 f.](#)
 - FTP [11](#)
- G**
 - GlassFish [133, 135](#)
 - Globally Unique Identifier [38](#)
 - grep [157](#)
 - Groovy [119](#)
 - GUID → Globally Unique Identifier
 - gzip [15](#)
- H**
 - Hashfunktion [164](#)
 - Header [59](#)
 - Heartbeats [110](#)
 - Heatmap [216](#)
 - Hexdump [57](#)
 - HTML-Content [125](#)
 - HTML-Datei [107](#)
 - HTML-Element [89](#)
 - HTML-Formular [166](#)
 - HTML-Seite [150](#)
 - HTML5 [2, 76](#)

- HTTP → Hypertext Transfer Protocol
 HTTP 2.0 97
 HttpOnly 172
 HyBi → BiDirectional or Server-Initiated HTTP
 Hypertext Transfer Protocol 1, 5, 13, 97
 - Anfrage 1, 12 ff., 22, 25
 - Anfragezeile 16
 - Antwort 1, 12, 24, 60
 - Authentifizierung 162
 - Basic Authentication 163
 - Content-Type 32, 104
 - Digest Authentication 164
 - Header
 - Acces-Conroll-Allow-Origin 192
 - Allow 18
 - Authorization 163
 - Cache-Control 199
 - Connection 29, 36
 - Origin 36, 159, 190, 194
 - Sec-WebSocket-Accept 37f.
 - Sec-WebSocket-Key 36f.
 - Sec-WebSocket-Protocol 37, 81, 184
 - Sec-WebSocket-Version 36
 - Upgrade 21, 36, 161
 - WWW-Authenticate-Header 164
 - Methode
 - CONNECT 184
 - DELETE 18
 - GET 17, 25, 32
 - HEAD 18
 - OPTIONS 18
 - PATCH 19
 - POST 17
 - PUT 18
 - TRACE 19
 - Reason 20
 - Status 20
 - Status-Line 20
 - Statuscode 20, 104, 163
 - Streaming 30
 - Version 17, 20f.
- I**
- IANA 81
 ICMP 11
 IDS → Intrusion Detection Systems
 IETF → Internet Engineering Task Force
 ImageData-Objekt 95
 IMAP 11
 -Tag 206
 Injection 159
 Instanz 103
- Integer Codierung 50
 Interaktivität 25
 International Organization for Standardization 6
 Internet Engineering Task Force 35, 75
 Interpreter 119
 Intranet 191
 Intrusion Detection Systems 184
 Intrusion Prevention Systems 184
 IP 9, 11
 IP-Hijacking 184, 187
 IPS → Intrusion Prevention Systems
 IPSec 11
 ISO → International Organization for Standardization
 ISO/OSI-Layers 6, 57
 ISO/OSI-Referenzmodell 6
- J**
- Jade 204, 251
 Jade-Template 206
 Jade-Template-Engine 251f.
 JAR-Datei 136
 Java 119f., 122f.
 Java API for WebSocket 133
 Java Community Process 132
 Java Enterprise 132
 Java Specification Request 132
 Java Virtual Machine 119
 JavaScript 25, 29, 75, 103, 109, 158
 - canvas-toBlob.js 224
 - getImageData() 95
 - Konsole 86 ff., 182
 - Objekt 213
 - sendFile() 115
 - takeSnapshot() 226
 JavaScript-API → W3C WebSocket-API
 JCP → Java Community Process
 JDK 122
 jQuery 206, 212
 JSON 209
 JSON-Objekt 214, 249
 JSON-Parser 251
 JSONP 29, 172
 JSR → Java Specification Request
 JSR 356 132, 136
 - Annotation 138
 - API 139
 - Konfigurationsklasse 139
 - Session 138
 - WebSocket-Client 140
 - WebSocket-Server 136

JVM → Java Virtual Machine

K

Kommunikation 10, 13
 Kommunikationsendpunkt 9
 Kommunikationsfluss 11
 Kommunikationskanal 10
 Komprimierungsverfahren 15
 kryptografische Schlüssel 160

L

Ladezeit 98
 Landing-Page 187
 Lasttest 153
 Latenz 48
 Login-Formular 167, 170
 Login-Maske 179
 Login-Seite 170
 Long-Polling 28, 110

M

MAC 7
 Malware 191
 Man in the Middle 187, 195
 Mask-Bit 59
 MASK-Flag 41
 Maskierung 43, 184
 Maskierungsschlüssel → Masking-Key
 Masking-Key 39, 41, 43
 Mausevents 218
 Mausposition 92
 Medienquellen 225
 Methode 16
 Mikro-Instanz 153
 MIME 11
 MIME-Typ 32
 Mobilfunknetze 161
 Modulo-Operator 44
 Mouse-Tracking 216
 Multiplexing Extension for WebSockets 91

N

Nachrichtengröße 113 f., 222
 Namensschema 79
 NetBeans 134 f.
 netstat 157
 Netty 122
 Network Layer → Vermittlungsschicht
 Network Scan 191
 Netzwerkanalysen 191
 Netzwerkschicht 12
 Netzwerkschnittstelle 55

nginx 165, 185

Node.js 102, 148, 151, 204
 – Echo-Server 107
 – Installation 104
 – Modul
 – Express 204
 – Forever 156
 – Modulmanager 103, 105
 – Webserver 103
 Non-Control-Frames 40
 nonce 164

O

öffentlicher Schlüssel 160
 Opcode 40, 42, 48, 59
 Open Systems Interconnection 6
 Opening-Handshake 36, 60, 70
 – Request 36, 67, 72, 167
 – Response 37, 72
 Origin-Host 194
 OSI → Open Systems Interconnection
 OSI-Modell 6, 8, 11
 Overhead 27, 42

P

Paket 56
 Paketfilter 183
 Path 17
 Payload 19, 125
 Payload len 41
 Performance 98, 147
 Performancetest 157
 Persistent Connection 29
 Phishing-Mails 190
 Physical Layer → Bitübertragungsschicht
 Ping 113, 186, 196
 Ping-Frame 41, 48
 Ping-Intervall 197
 Play Framework 122
 – Action 125, 130, 179
 – activator 122, 124
 – Controller 124, 127, 178
 – Echo-Server 129
 – Event-Handler 129
 – Formularbasierte Authentifizierung 178
 – Helper-Klasse 179
 – Installation 122
 – Konsole 124
 – Layout-Template 125
 – Methode
 – close() 130
 – invoke() 130

- onClose() 130
- onMessage() 130
- onReady() 130
- render() 125 f.
- validate() 180
- WebSocket.reject 183
- write() 130
- Ordnerstruktur 123
- Projekt 123
- readyState 182
- Result 125
- Route 126, 128, 181
- Typparameter 130
- View 125, 127, 131, 179
- WebSocket.In 130
- WebSocket.Out 130
- Plug-In 2
- Polling 27
- Pong 186, 196
- Pong-Frame 41, 49
- POP3 11
- Port 7, 17, 183, 192
- Port Scanner 187, 192, 195
- PPPoE 11
- Presentation Layer ⇒ Darstellungsschicht
- PrintWriter 15
- privater Schlüssel 160
- Protokoll 1, 9 f.
- Protokollanalyse 54
- Proxy-Server 60, 159, 165, 183
- Public Draft 132
- Python 119

- Q**
- QR-Code 204, 206 f.
- Query 17, 79

- R**
- Referenzmodell 6, 8
- Remote-Debugging 235
- Remote Shell 187, 191, 195
- Request → Hypertext Transfer Protocol
 - Anfrage
- Request For Comments 35
- Response → Hypertext Transfer Protocol
 - Antwort
- Response-Code 164
- Response-Header 24
- Ressource 16, 22
- REST-Konzept 19
- RFC → Request For Comments
- RFC 6455 78, 91, 159, 162
- Round-Trip-Zeit 147, 153, 157
- Route-Matcher 212
- RS-232 11
- RSV-Flag 40, 42, 59
- Ruby 119

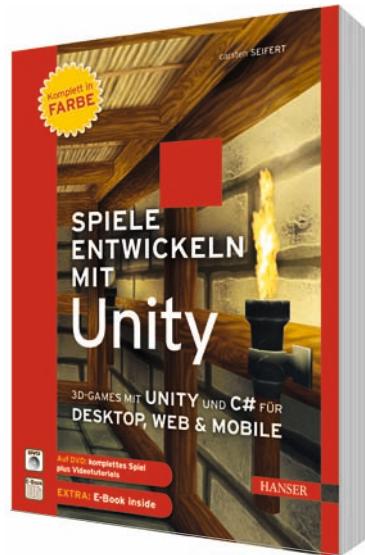
- S**
- Same Origin Policy 29, 158, 193
- Scala 122, 125
- Script Tag Long Polling 29
- SCTP → Stream Control Transmission Protocol
- Secure Socket Layer 11, 159
- Server 101
- Server-Push 29, 31
- Server-Sent Events 2, 31, 203
- EventSource-Objekt 31
- Handler
 - onerror 31
 - onmessage 31
 - onopen 31
- Kanal 31
- Serverbenachrichtigung 33
- Serverfehler 23
- Session 168
- Session-ID 166 f., 174, 176, 204, 206
- Session Layer ⇒ Sitzungsschicht
- Session-Token 194
- SFTP 11
- SHA-1 37
- Sicherheit 158
- Sicherungsschicht 6
- Sichtbare Proxys 184
- Signierungsschlüssel 174
- Sitzungsschicht 6 f.
- Skalierbarkeit 147
- Smart-TV 202
- SMTP 11
- SOAP 80
- Sockets 14
- Socket.io 109, 204, 216
 - Echo-Server 110
 - Event
 - connect 112
 - disconnect 112
 - error 112
 - message 112
 - Formularbasierte Authentifizierung 172
 - Methode
 - emit() 112
 - io.connect() 112, 206
 - send() 112
- SOP → Same Origin Policy

- SPDY 97
 SPDY-Session 71
 SSE → Server-Sent Events
 SSH 11
 SSL → Secure Socket Layer
 Startskript 153
 Stream Control Transmission Protocol 98
 String 93
 Subprotocol Name Registry 81
 Subprotokoll 39, 76, 80, 91
 Supervisor 154
 Switching Protocols 21, 37, 60
- T**
 TCP 9, 11f., 14, 53, 60, 73, 160
 TCP-Timeout 71
 Technology Classes 30
 Telnet 11, 14
 Testclient 151
 Testszenario 148
 Testumgebung 153
 textbasiert 14
 textbasierte Daten 93
 Textdaten 45
 Textframe 59
 Thread 103
 Timeout 54
 TLS → Transport Layer Security
 Token 166
 Touch-Events 207f.
 Transfervolumen 15, 197
 Transparente Proxys 184
 Transport Layer → Transportschicht
 Transport Layer Security 160f., 185
 – Handshake 160
 – Zertifikat 160
 Transportschicht 6, 12, 187
 Tyrus 133
- U**
 UDP 9, 11
 Umgebungsvariablen 241
 Umleitung 22
 unidirektional 33
 Unverfälschtheit 159
 URL 1, 16f., 79f., 116
 URL-Schema 16, 79
 – https 160
 – ws 79
 – wss 80, 160, 185
 Usability-Test 216
 User 16
- UTF-8 93
- V**
 V8-Engine 103
 Validierung 219
 Verbindungsabbau 10, 13, 96
 Verbindungsauflaufbau 10, 13
 Vereinfachtes OSI-Modell 8
 Vermittlungsschicht 6f.
 Vert.x 119, 148, 211
 – Closed-Handler 150
 – Echo-Server 120, 149
 – Installation 120
 – Paketmanager 120
 – WebSocket-Handler 121, 149
 Vertrauliche Kommunikation 159
 Vertraulichkeit 184
 virtueller Rechner 148, 153
- W**
 W3C 30, 75
 W3C WebSocket-API 75, 151, 196
 – Attribut 89
 – binaryType 89, 92, 94
 – bufferedAmount 89, 91
 – code 96f.
 – extensions 89, 91
 – protocol 89, 91
 – readyState 81, 88
 – reason 96f.
 – url 89
 – wasClean 97
 – Candidate Recommendation 76
 – Editor's Draft 76
 – Event-Handler 83
 – onclose 83, 85, 87f., 97, 108
 – onerror 83f., 87
 – onmessage 83, 85, 87
 – onopen 83f., 86f., 89
 – Event-Objekt 97
 – Instanz 80, 87
 – Konstruktor 80, 87
 – Methode
 – addEventListener() 86
 – close() 82, 88, 96
 – send() 82, 92, 95
 – Objekt 80
 – Objekterzeugung 80, 88
 – Protokollversion 78
 – Verbindungsauflaufbau 80
 – Working Draft 75
 – Zustand

- CLOSED [81f., 85, 87](#)
 - CLOSING [81f.](#)
 - CONNECTING [81](#)
 - OPEN [81f., 87](#)
 - Webcam [221](#)
 - WebIDL [80, 96](#)
 - WebRTC [222](#)
 - WebSocket-Client [75, 86](#)
 - WebSocket-Datenverkehr [66](#)
 - WebSocket-Element [76](#)
 - WebSocket-Endpoint [139, 178](#)
 - WebSocket-Frame [39, 42, 59, 70, 93, 118, 188](#)
 - WebSocket-Handshake [36, 67, 159](#)
 - WebSocket-Header [39f.](#)
 - WebSocket-Node [113, 151, 196](#)
 - Event
 - connect [151](#)
 - onmessage [119](#)
 - Formularbasierte Authentifizierung [176](#)
 - message-Objekt [114](#)
 - Methode
 - connect() [151](#)
 - send() [116](#)
 - sendBytes() [115](#)
 - Server [114](#)
 - WebSocket-Payload [39](#)
 - WebSocket Per-frame Compression [91](#)
 - WebSocket-Protokoll [35](#)
 - WebSocket-Proxys [185](#)
 - WebSocket.io [106](#)
 - Echo-Server [107](#)
 - window-Objekt [76](#)
 - Wireshark [54](#)
 - WLAN [11](#)
 - WLAN-Router [12](#)
 - word count [157](#)
- X**
- XHR → XMLHttpRequest
 - XHR-Objekt [233](#)
 - XMLHttpRequest [25, 233](#)
 - Methode
 - send() [26](#)
 - setTimeout() [27](#)
 - Objekt [26](#)
 - XOR [43](#)
 - XSS → Cross-Site-Scripting
 - XST → Cross-Site-Tracing
- Z**
- Zeitmessung [148, 152, 157](#)
 - Zeitstempel [152](#)
 - Zertifikat [185](#)
 - Zertifikatswarnung [185](#)
 - Zertifizierungsstelle [160, 185](#)
 - Zufallszahl [43, 167](#)
 - Zugriffskontrolle [167f.](#)
 - Zustandsdiagramm [82](#)
 - Zustandsinformation [168](#)
 - zustandslos [24](#)

HANSER

Spiele entwickeln leicht gemacht



Carsten Seifert

Spiele entwickeln mit Unity

3D-Games mit Unity und C# für Desktop,

Web & Mobile

496 Seiten. Mit DVD. Komplett in Farbe

€ 34,99. ISBN 978-3-446-43939-9

Auch als E-Book erhältlich

€ 27,99. E-Book-ISBN 978-3-446-44129-3

- Für alle, die ihr eigenes Spiel entwickeln wollen; Vorkenntnisse sind nicht erforderlich
- Auf DVD: das Game aus dem Buch mit allen Ressourcen; weitere Beispiele; Videotutorials (Gesamtdauer: 65 Minuten)
- Im Internet: Zusatzmaterialien und Aktualisierungen
- Extra: E-Book inside

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

HANSER

Richtig gute Apps entwickeln



Thomas Sillmann

Apps für iOS 8 professionell entwickeln

Sauberen Code schreiben, stabile Apps programmieren. Techniken und Methoden von Grund auf verstehen

426 Seiten

€ 39,99. ISBN 978-3-446-44018-0

Auch als E-Book erhältlich

€ 31,99. E-Book-ISBN 978-3-446-44130-9

- Lernen Sie mit einem effizienten Arbeitsstil richtig gut funktionierende und stabile Apps speziell für iOS zu programmieren
- Profitieren Sie von zahlreichen Tipps und Tricks
- Für fortgeschrittene iOS-App-Entwickler und für Entwickler, die in die Programmierung für iOS einsteigen wollen
- Auf der Website des Autors finden Sie einige Basisklassen, aufgebaut aus den Code-Beispielen aus dem Buch

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

HANSER

Fundierter Einstieg in die Spieleentwicklung



Jonas Freiknecht
SPIELE ENTWICKELN MIT GAMESTUDIO
Virtuelle 3D-Welten mit Gamestudio A8 und Lite-C
480 Seiten. Vierfarbig. Mit DVD.
E-Book-inside
€ 39,99. ISBN 978-3-446-43119-5

Auch einzeln als E-Book erhältlich
€ 31,99. E-Book-ISBN 978-3-446-43267-3

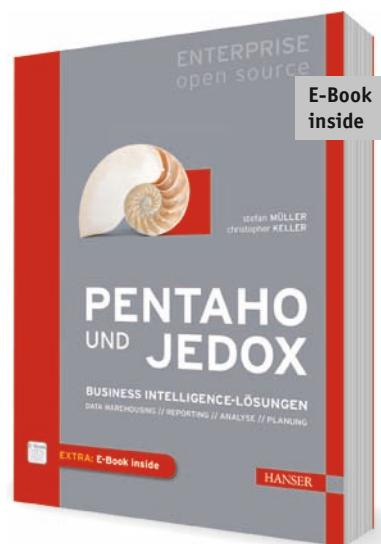
Dieses umfassende Handbuch zeigt Ihnen, wie Sie die Autorensoftware Gamestudio bedienen, um ein eigenes Spiel zu programmieren. Sie werden sich mit den Editoren vertraut machen, um 3D-Welten zu entwerfen und ihnen mit selbst gemachten Spielfiguren und Gegenständen Leben einzuhauchen. Sie werden lernen, wie man ein Projekt so plant, dass es realisierbar ist, gut aussieht und: Spaß macht!

Schritt für Schritt wird Ihnen auf spielerische Art und Weise erklärt, wie Sie die Programmiersprache Lite-C und die Funktionen der Gamestudio-API einsetzen. Sie werden 3D-Welten laden, deren Bewohner mit Funktionen versehen und diese auf Aktionen von außen reagieren lassen. Sie werden eine realistische Naturlandschaft mit Pflanzen und Gräsern versehen, zwischen verschiedenen Levels hin und her wechseln, dynamische Ladebildschirme erstellen und und...

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

HANSER

Daten in Informationen verwandeln



Müller, Keller

Pentaho und Jedox

Business Intelligence-Lösungen: Data Ware-

housing, Reporting, Analyse, Planung

444 Seiten. E-Book-inside

€ 59,99. ISBN 978-3-446-43897-2

Auch einzeln als E-Book erhältlich

€ 47,99. E-Book-ISBN 978-3-446-44125-5

- Für Manager und IT-Führungskräfte, Entwickler und Berater
- Lernen Sie Pentaho und Jedox im Detail und im Vergleich zueinander kennen
- Mit vielen Anwendungsbeispielen
- Die Beispiele aus dem Buch finden Sie im Downloadbereich des Verlages

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

TOM DEMARCO



Als auf der Welt
DAS LICHT
ausging

Ein Wissenschafts-Thriller

HANSER

**»Der Weltuntergang steht bevor,
aber nicht so, wie Sie denken.
Dieser Krieg jagt nicht alles in die Luft,
sondern schaltet alles ab.«**

Tom DeMarco
Als auf der Welt das Licht ausging

ca. 560 Seiten. Hardcover
ca. € 19,99 [D]/€ 20,60 [A]/sFr 28,90
ISBN 978-3-446-43960-3 · WG 121
Erscheint im November 2014



Sie möchten mehr über Tom DeMarco und
seine Bücher erfahren.
Einfach Code scannen oder reinklicken unter
www.hanser-fachbuch.de/special/demarco

Lizenziert für luxiliph@gmail.com.

© 2015 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

TOM DEMARCO

**Als auf der Welt
DAS LICHT
ausging**

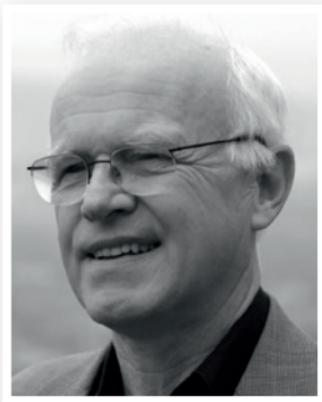
Aus dem Amerikanischen von Andreas Brandhorst

Leseprobe

Das Buch erscheint im November 2014.

Tom DeMarco

ist Projektmanagement-Experte, vielgefragter Berater und Autor zahlreicher im Carl Hanser Verlag erschienener Bestseller wie »Der Termin« oder zuletzt »Wien wartet auf Dich«. Er ist Partner der Atlantic Systems Guild, einer Beratergruppe, die sich auf die komplexen Prozesse der Systementwicklung spezialisiert hat, mit besonderem Augenmerk auf die menschliche Dimension.



Andreas Brandhorst

Andreas Brandhorst (geboren 1956 in Norddeutschland) hat mit dem Schreiben sehr früh angefangen und wenige Jahre später wurde aus dem Hobby ein Beruf, zu dem nicht nur das Schreiben eigener Texte gehörte, sondern auch das Übersetzen (u.a. der Werke von Terry Pratchett). Er ist

Autor der bekannten Kantaki-Romane (2 Trilogien), der Science-Fiction-Romane »Kinder der Ewigkeit«, »Das Artefakt« (ausgezeichnet mit dem Deutschen Science-Fiction-Preis als bester deutscher SF-Roman 2013), »Der letzte Regent« und »Das Kosmotop« (erscheint Juni 2014). Außerdem gehen die Thriller »Äon«, »Die Stadt« und »Seelenfänger« auf sein Konto.



Was vorher geschah:

Fanatiker in der US-Regierung glauben sich hinter ihrem Raketenabwehrschild sicher und wollen Amerikas Gegner überall auf der Welt mit einem nuklearen Erstschlag auslöschen. Doch eine kleine Gruppe von Forschern hat einen Apparat entwickelt, der weltweit Atomexplosionen verhindern kann. Die Aktivierung dieses Apparats würde zwar einen Atomkrieg verhindern, aber der Menschheit auch die Elektrizität nehmen

...

DER MANN, DER EINGRIFF

Homer hatte sie alle zu Bett geschickt. Während der vergangenen Nacht waren sie auf den Beinen gewesen, sagte er, und sie mussten frisch und ausgeruht sein für das, was geschehen würde. Doch zumindest für Loren und Edward war an Schlaf nicht zu denken. Sie saßen am Fenster von Edwards Zimmer und blickten über die Stadt. Ed hatte das Licht ausgeschaltet, dann wieder ein und noch einmal aus. Eine ganz einfache Sache, über die man gar nicht nachdachte, aber würde sie am kommenden Tag noch möglich sein? Er durchquerte das dunkle Zimmer und setzte sich auf den leeren Stuhl Loren gegenüber.

»Ich glaube, ich weiß, was passieren wird«, sagte er.
»Kelly hat es bereits gesagt. Es scheint vorherbestimmt zu sein.«

Loren nickte betrübt.

»Die Offshore-Kabaner werden sich genau so verhalten, wie es Simula-7 vorhergesagt hat. Sie sind vollkommen berechenbar. Sie werden St. Louis angreifen, wie in der Simulation. Sie werden es sich nicht anders überlegen. Bestimmt gehen sie davon aus, dass unsere Behörden eine Evakuierung der Stadt veranlassen. Und bestimmt reden sie sich ein, dass wir eine Eskalation vermeiden wollen. Es wäre dumm von uns, alles auf die Spitze zu treiben – nach dem, was wir auf Kuba angerichtet haben, ist die Zerstörung einer leeren Stadt durch

eine einzelne Rakete nicht mehr als eine kleine Verwarnung.«

Loren nickte erneut. »Dumm«, wiederholte er.

»Sie werden glauben, dass wir nicht zurückschlagen, aber da irren sie sich. Wir wissen es besser. Stell dir vor, was passiert, wenn sie ihre eine Rakete auf St. Louis abfeuern. Das modifizierte Revelation 13 erledigt sie vielleicht, oder auch nicht. Möglicherweise ist das System nicht in der Lage, eine einzelne Rakete abzufangen. Angenommen, es ist dazu imstande. Was machen wir dann?«

Loren dachte darüber nach. »Die logische Sache wäre, nichts weiter zu tun. Die Welt könnte glauben, dass Amerikas Raketenabwehr schild funktioniert, dass wir unangreifbar sind. Es wäre eine sehr starke Position für uns.«

»Aber wir gehen nicht logisch vor. Wir sind Fanatiker.«

»Ja. Also regen wir uns mächtig auf. Man hat uns angegriffen; wir müssen es den Angreifern zeigen. Es ist eine Frage verletzten Männerstolzes.«

»Also schicken wir unsere Raketen los. Wir radieren Iran, Nordkorea, Pakistan und alle anderen Länder aus, die uns ein Dorn im Auge sind. Genau das passiert, wenn der Schild hält. Wenn nicht ... Dann sitzen wir in den Trümmern von St. Louis. Es wird viele Opfer geben, weil wir die Stadt nicht evakuiert haben. Was machen wir?«

»Vergeltung. Wir suchen einen Sündenbock und starten unsere Raketen.«

»Wir starten sie auf jeden Fall.«

»Ja, auf jeden Fall.«

Sie schauten in die Nacht hinaus und beobachteten die Lichter der Stadt. Nach einer Weile fuhr Edward fort: »In der griechischen Tragödie gibt es einen Moment des Übergangs, direkt nach dem Höhepunkt. Vorher haben Menschen die Ereignisse kontrolliert und nachher kontrollieren die Ereignisse die Menschen. Ich habe den Eindruck, dass dieser Moment heute Morgen verstrichen ist. Jetzt erwartet uns das düstere Ende

der griechischen Tragödie; die Akteure werden zu Zuschauern.«

»Das gilt nicht für uns«, sagte Loren. »Bei uns sieht es anders aus. Wir können eingreifen. Wir können handeln, den Effektor einschalten.«

»Aber können wir frei entscheiden? Haben wir eine Wahl? Eigentlich nicht. Wir müssen den Effektor einschalten. Die Zahlen diktieren es, denn selbst ein begrenzter nuklearer Schlagabtausch würde weitaus mehr Opfer fordern. Wir müssten selbst dann eingreifen, wenn wir wüssten, dass weitere Eskalationen ausbleiben. Das ist der zweite Teil von dem, was meinem Gefühl nach heute Nacht geschehen wird: Wir schalten den Effektor ein. Wodurch die Welt zum Stillstand kommt. Die Menschen, die von den Atomraketen getötet worden wären, leben noch – wir haben sie gerettet. Aber jetzt funktioniert nichts mehr. Die abgefeuerten Raketen fallen zu Boden, ohne zu explodieren, doch das ist nur der Anfang. Motoren lassen sich nicht mehr starten, es gibt keinen elektrischen Strom, Flugzeuge stürzen ab ... Der wahre Albtraum beginnt.«

»Vielleicht können wir den Effektor später abschalten. Wenn die Krise überstanden ist.«

»Nein, nie. Es wird immer mehr Waffen geben, die auf ihre Chance warten.«

»Wir können den Effektor nach Belieben ein- und ausschalten.«

»Loren. Denk gründlich darüber nach. Wenn wir uns einen Spaß daraus machen, den Permanenten Effektor zu aktivieren und zu deaktivieren ... Wie lange dauert es dann wohl, bis die Leute merken, dass wir dahinterstecken?«

»Ich verstehe nicht ganz ...«

»Rupert Paule wendet sich an Armitage, einen Physiker von Weltklasse, und sagt: »Was zum Teufel ist hier los? Welche Kraft neutralisiert unsere Raketen, Motoren und Generatoren?« Armitage nimmt einige Untersuchungen vor, während der Effekt wirkt. Was kann er herausfinden?«

Loren überlegte. »Die potenzielle Energie in jeder brennbaren Substanz erscheint reduziert, wenn sich der Effekt auswirkt, und kehrt ohne den Effekt auf das normale Niveau zurück. Das böte einen Hinweis auf die ganze Theorie von T-Prime. Wozu wir Jahre gebraucht haben, um es zu verstehen ...«

»Armitage könnte viel schneller die richtigen Schlüsse ziehen, vielleicht in Wochen.«

»In Tagen«, sagte Loren.

»Wahrscheinlich. Und dann würde Paule fragen: »Wer tut uns dies an, Dr. Armitage?« Und für Lamar wäre es schon nach einer Sekunde klar: Homer Layton und sein Team. Homers Artikel in Science über die Pekuliarbewegung bietet einen eindeutigen Hinweis.«

»Sie würden über uns herfallen, uns den Effektor wegnehmen und ihn ausschalten.«

»Und manchmal würden sie ihn wieder einschalten.«

Loren begriff, was Edward meinte. »Oh.«

»Ja. Sie schalten ihn ein, wenn sie gegen sie gerichtete strategische Aktionen entdecken. Und sie schalten ihn aus, wenn sie selbst zuschlagen wollen. Einen besseren Abweherschild gibt es nicht.«

»Vielleicht wäre es gar nicht so schlecht.«

»Es wäre furchtbar. Denn ihnen müsste klar sein, dass auch andere Länder Physiker haben. Mit dem Hinweis, den wir ihnen gegeben haben, kommen sie schnell hinter das Geheimnis von T-Prime. Was bedeutet, dass sie nach einigen Wochen ihren eigenen Effektor bauen können. Doch das würde den Eiferern und Fanatikern ganz und gar nicht gefallen. Sie müssten damit rechnen, dass ihr Vorteil nach wenigen Wochen dahin ist. Was würden sie tun?«

»Sie müssten handeln, um ihren Vorteil abzusichern. Sie würden vielleicht ...«

»Genau. Sie würden angreifen, solange sie die Gewiss-

heit haben, dass sich der Gegner nicht wirkungsvoll zur Wehr setzen kann. Davon müssen wir ausgehen.«

Es dauerte eine Weile, bis sich die Erkenntnis festsetzte. Die Entscheidung, den Effektor einzuschalten, war auch die Entscheidung, ihn für immer an zu lassen. »Vielleicht funktioniert der Effektor gar nicht«, sagte Loren.

»Das ist unsere optimistischste Hoffnung.« Edward lächelte bitter. »Es würde bedeuten, dass wir zusammen mit dem Rest der Welt in nuklearer Glut gebraten oder vom Fallout vergiftet werden. Wir wären tot, was wir eines Tages ohnehin sein werden. Aber zumindest müssten wir uns nicht vorwerfen, dass alles unsere Schuld ist.«

.

.

.

... Eine Atomrakete vom Typ SS-24 startete von einer Insel vor der Küste Ecuadors, gerichtet auf St. Louis, Missouri. Der Startzeitpunkt war so gewählt, dass die Rakete ihr Ziel genau um Mitternacht St. Louis-Zeit erreicht. Ein zweihundert Meilen westlich von San Diego stationierter Zerstörer der amerikanischen Marine ortete die Rakete und einige Sekunden später wurde ein Alarm ausgelöst. Da der Zerstörer auf eine solche Sichtung vorbereitet gewesen war, ging man gleich auf Alarmstufe Rot und schickte eine Nachricht ins StratCom-Netzwerk.

Albert döste mit dem Ohr am Empfänger. Die Mitteilung hätte Teil seines Traums sein können, denn in letzter Zeit träumte er oft von solchen Dingen. Er hob den Kopf und starrte auf das Gerät in seiner Hand, das den Alarm wiederholte. Er blickte zu Homer, der wach im Sessel neben ihm saß. Homer hatte alles gehört. Die Worte der Ankündigung schienen keine nennenswerte Wirkung auf ihn zu haben; sie waren mehr wie ein morgens klingelnder Wecker. In diesem Fall lautete die Botschaft des Klingelns: Es geht los.

Albert hielt das kleine Gerät wieder ans Ohr und sein

Blick kehrte zu Homer zurück. »Neunzehn Minuten, glauben sie«, sagte er und sah auf die Uhr. »Um ein Uhr unserer Zeit.«

Homer stand mühsam auf. Alte Leute sollten nicht in tiefen Sesseln sitzen, dachte er. Loren, der auf dem Boden neben ihm geschlafen hatte, war schon auf den Beinen. »Ich hole die anderen«, sagte er.

Edward hatte seine Tür einen Spaltbreit offen gelassen. Loren sah ins Zimmer und sagte: »Es ist so weit, Ed.« Es hätte das frühe Wecken für den Beginn eines Campingausflugs sein können. Er hörte Edwards Antwort aus dem dunklen Zimmer und ging weiter, zu Sonia gleich nebenan.

Loren klopfte an und wartete. Er hörte Bewegung im Zimmer, die Tür öffnete sich und Sonia blinzelte im Licht des Flurs. »Sonia.« Er wollte sie in die Arme nehmen, sie trösten, doch sie behielt die Tür zwischen ihnen.

»Ich bin im Schlafanzug«, sagte sie.

»Komm so schnell du kannst zu Homer.«

»Gib mir ein paar Minuten fürs Anziehen.« Sie schloss die Tür.

Weiter zu Kellys Zimmer. Loren klopfte an und hörte Geräusche, bevor sich die Tür öffnete. Kelly war hellwach. Sie trug ein weißes Nachthemd mit Rüschen an den Ärmeln. Hinter ihr brannte eine kleine Lampe.

»Es ist geschehen«, sagte Loren.

Kelly zog ihn herein. »Sieh nach Curtis«, sagte sie. »Ich ziehe mir schnell was über.«

Loren ging ins Nebenzimmer und spähte in die Dunkelheit. Er hörte das gleichmäßige Atmen des Kindes. Die Gestalt im Bett wirkte friedlich im Schlaf. Er kehrte in Kellys Raum zurück. Sie stand vor der Kommode, mit dem Rücken zu ihm, und zog eine Jeans unter ihrem Nachthemd hoch. Ihr Hintern zeigte sich kurz, als sie die Hose zurechtrückte. Das Nachthemd warf sie achtlos beiseite. Loren sah ihren langen, schmalen Rücken. Sie war größer als seine Schwestern, dach-

te er, ein bisschen größer. Kelly zog sich ein T-Shirt über den Kopf und drehte sich zu ihm um. »Fertig«, sagte sie und stand barfuß vor ihm. Keine Schuhe, keine Unterwäsche. Sie trafen noch vor Edward in Homers Suite ein.

Homer hatte Maria geweckt. Sie trat aus dem Schlafzimmer und zog den Gürtel eines Morgenmantels zu. Claymore kam von der anderen Seite herein. Sonia und Edward erschienen gleichzeitig. Noch elf Minuten bis zum Einschlag. Homer schloss die Tür, verriegelte sie und drehte sich ernst zu ihnen um.

»Gloria Verde hat eine Rakete auf St. Louis abgefeuert. Albert hat den Alarm vor einigen Minuten mit seinem StratCom-Apparat gehört. Die Rakete wird ihr Ziel um ein Uhr unserer Zeit erreichen. Uns bleiben nur wenige Minuten, um genau zu überlegen. Darauf kommt es jetzt an, dass wir genau nachdenken.

Es gibt einige Dinge, die wir Albert, Maria und Claymore erklären müssen, über unsere Vereinbarung in Bezug auf den Effektor, falls wir entscheiden, ihn einzuschalten. Hörst du zu, Clay?«

»Oh, klar.« Claymore hatte als einziger Platz genommen. Er saß auf der Couch, in einem pfirsichfarbenen Schlafanzug. Auf dem Tisch lag eine Hochglanzbroschüre über das Nachtleben von Fort Lauderdale. Er schlug sie auf. »Klar«, sagte er.

Homer wandte sich an Albert und Maria. »Ihr wisst, was es mit dem Effektor auf sich hat. Ich habe es euch erklärt. Ihr wisst auch, was wir heute Nacht tun könnten, was wir in Erwägung ziehen. Aber was auch immer hier geschieht, ihr seid dafür nicht verantwortlich. Das ist wichtig. Die Verantwortung tragen wir fünf.« Er sah die Mitglieder der Gruppe an. »Ich selbst, Edward, Sonia, Loren und Kelly. Nur wir fünf. Wir stimmen ab, bevor wir etwas unternehmen. Zuvor sind wir übereinkommen, dass die Entscheidung, den Effektor einzuschalten,

die Zustimmung von uns allen verlangt. Eine Nein-Stimme läuft auf ein Veto hinaus. Offenbar müssen wir heute Nacht abstimmen. Bald.

Noch hat eine Abstimmung darüber, ob wir den Effektor verwenden sollen, keinen Sinn, denn ich würde mit Nein stimmen. Wir können nicht einschreiten, um St. Louis zu retten. Es gibt noch immer die Möglichkeit, dass damit alles vorbei ist. Wenn Washington entscheidet, den Angriff auf St. Louis ohne Vergeltungsmaßnahmen hinzunehmen, brauchen wir den Effektor nicht einzuschalten. Das wäre eine große Erleichterung für uns alle. Auf diese Weise müssen wir es sehen. Wir warten bis nach der Explosion der Rakete. Wir warten und warten. Wenn Amerika protestiert, ohne einen Gegenangriff zu starten, brauchen wir nicht abzustimmen. In dem Fall muss niemand sagen, wie er oder sie gestimmt hätte. Dann können wir den Rest unseres Lebens mit ruhigem Gewissen verbringen, weil wir die Macht, die in unsere Hände fiel, unangetastet ließen, eine Macht, die die Welt in Dunkelheit stürzen kann. Dann werden wir uns immer fragen, was geschehen wäre, wenn wir ein paar Leben in einer Stadt des Mittelwestens gerettet, dafür aber die ganze Welt grundlegend verändert hätten. Wir könnten bei einem Bier in Cornell darüber reden.«

Ihm gingen die Worte aus. Er hätte überhaupt nichts sagen müssen, das wussten sie alle.

Für einen langen Moment herrschte Stille und dann raschelte es, als Claymore umblätterte.

Homer fiel noch etwas ein. »Wenn wir abstimmen müssen, und ich hoffe, das ist nicht der Fall, aber wenn uns die Umstände zu einer Abstimmung zwingen, so möchte ich fragen ...«

Albert hob die Hand. Er hatte das Ohr am Empfänger und sein Blick ging ins Leere. »Sie starten«, sagte er.

»Was?«, fragte Loren fassungslos. »Wer startet? Wir?«

»Der Präsident hat den Befehl gegeben. Amerika schlägt zu.«

»Aber das kann doch nicht sein! Sie müssen warten, bis die Rakete St. Louis trifft. Vielleicht hält der Abwehrschild. Oder die Kubaner überlegen es sich im letzten Moment anders und lassen die Rakete ins Meer stürzen. Oder sie explodiert überhaupt nicht. Es ist zu früh für eine Reaktion.«

Albert zuckte die Schultern.

Homer sah auf die Uhr. »Wir stimmen jetzt ab«, sagte er. »Es bleiben noch neun Minuten. Wenn wir alle mit Ja stimmen, können wir handeln, noch bevor die Rakete St. Louis erreicht. Dann retten wir auch das Leben der dortigen Menschen, was alles leichter macht.«

»Es wird gestartet«, sagte Albert. »StratCom bestätigt, dass sich die erste Rakete auf den Weg macht ... und jetzt die zweite, von einem U-Boot aus. Es hat begonnen. Weitere Starts werden gemeldet ...«

»Wir stimmen ab.« Homer und seine Gruppe wichen beiseite, weg von Albert und Maria. Eine symbolische Trennung. »Ja bedeutet, dass wir den Effektor einschalten. Nein bedeutet, dass wir nichts unternehmen. Ich stimme ...«

»Warte!«, sagte Loren. Er erinnerte sich an die letzte Abstimmung. Alle hatten sofort ihre Stimme abgegeben, mit Ausnahme von Sonia; letztendlich war es also ihre Stimme gewesen, die den Ausschlag gegeben hatte. Loren wollte nicht, dass sich so etwas wiederholte. »Kleine Zettel«, sagte er. »Wir schreiben unsere Stimme auf. Damit niemand der Letzte ist und den ganzen Druck fühlen muss.«

Auf dem Tisch lag ein Block mit gelben Haftzetteln. Loren riss einen für jeden von ihnen ab. Es gab Stifte und jeder nahm einen. Sonia holte einen aus ihrer Handtasche. Loren schrieb »Ja« auf seinen Zettel und sammelte dann die anderen ein. Er klebte sie an seinen Ärmel, in einer Reihe: alles Ja-Stimmen. Sonias Ja war so klein geschrieben, dass man genau hinsuchen musste, um es zu erkennen: zwei winzige Buchstaben, kaum einen halben Zentimeter groß.

»Alle haben mit Ja gestimmt«, sagte er.

Homer nickte. »Ich schalte den Effektor selbst ein.«

»Noch sieben Minuten«, sagte Albert.

Edward hatte den verzierten Eichenholzkasten mitgebracht. Er stellte ihn auf den Tisch, öffnete ihn und trat zurück. Stille herrschte. Homer ging allein zu dem Kasten und blickte darauf hinab.

»Es befindet sich ein Schiebeschalter an der Seite«, sagte Loren.

»Ich weiß, ich weiß.«

Alberts Stimme kam wie aus weiter Ferne. »Noch sechs Minuten«, sagte er. »Was nicht heißt, dass ich drängen möchte.«

»Ich weiß«, erwiderte Homer.

Es wäre Loren lieber gewesen, wenn Maria jetzt neben Homer gestanden hätte; er sollte jetzt nicht so allein sein. Doch Maria war tief in den weißen Sessel gesunken und hatte den Kopf zur Seite gedreht.

Kelly trat vor, griff mit beiden Händen nach Homers linker Hand und drückte ihre Wange an seine. Loren glaubte zu sehen, dass sie ihm etwas zuflüsterte, aber er hörte nichts. Homer nickte und streckte die rechte Hand nach dem Schalter aus. Loren reckte den Hals. Hatte er den Effektor eingeschaltet? Homer wirkte wie erstarrt.

»Wie viele Menschen leben in St. Louis?«, fragte Edward. »Drei Millionen? Homer, in den nächsten Minuten rettest du genug Menschen, um die Entscheidung zu rechtfertigen. Innerhalb der nächsten Stunde wirst du Dutzende von Millionen Leben gerettet haben, weitaus mehr, als durch den Effekt verlorengehen.«

»Ich weiß«, sagte Homer. »Also tue ich es.« Er betätigte den Schiebeschalter und trat zurück. Die anderen beugten sich vor. Der Schalter leitete Strom in den kleinen, einem Maser ähnelnden Generator und löste die mechanische Arretierung,

die das freie Schweben der Karte verhinderte. In der Mitte des Apparats glühte es rosarot. Die Karte begann sich zu drehen und suchte nach dem magnetischen Nordpol. Sie drehte sich über den Norden hinaus, kehrte dann quälend langsam zu ihm zurück und verharrte schließlich. Loren blickte aus dem Fenster. Nichts war geschehen.

»Vielleicht ist der Magnet ...«, begann er.

Das Licht im Zimmer wurde schwächer. Es ging nicht einfach aus, wie bei einem plötzlichen Stromausfall; es wirkte eher, als würde jemand einen Dimmer herunter drehen. Als es im Zimmer ganz dunkel geworden war, sahen sie aus dem Fenster. Auch in der Stadt breitete sich Dunkelheit aus – nach einigen Sekunden waren überhaupt keine elektrischen Lichter mehr zu sehen. Eine Zeit lang blieb es still, bis Albert das Schweigen brach. »Drei Minuten bis zum Einschlag der Rakete in St. Louis.« Er hielt sich noch immer den StratCom-Apparat ans Ohr. Das Gerät lief mit Batterie, war also nicht vom Effektor betroffen. Der StratCom-Sender befand sich in einem Satelliten, außerhalb des irdischen Magnetfelds.

Sie wandten sich alle dem Fenster zu. Claymore stand auf und kam näher. »Sieh nur«, sagte er und winkte Homer nach vorn. »Ich hab's dir ja gesagt. Es ist eine andere Farbe.«

Der Nachthimmel hatte einen Hauch von Rosarot. Es sah wie die Nordlichter aus, die Aurora Borealis, aber das schwache Leuchten zeigte sich im Süden.

»Es ist eine andere Farbe«, wiederholte Claymore.
»Pink.«

»Ja, stimmt«, sagte Homer.

Loren holte tief Luft. »Es ist ein Uhr. Wird etwas durchgegeben?«

Alle Blicke richteten sich auf Albert. Er drückte sich den Empfänger noch etwas fester ans Ohr und schüttelte den Kopf. Dann starre er wieder ins Nichts. »Moment ... Es heißt, der Schild habe gehalten. Ja, der Schild habe gehalten und St.

Louis sei nicht zerstört. Es gibt Beobachter unweit der Stadt und sie melden keine Explosion.« Albert sah die anderen an. »Sie glauben, es liegt am Raketenabwehrschild.«

»Oh«, sagte Homer. »Ihnen dürfte bald klarwerden, was geschehen ist.« Er setzte sich auf die Armlehne von Marias Sessel. Sie sah noch immer zur Seite.

»Es werden die Namen der Personen genannt, die angeblich St. Louis gerettet haben«, sagte Albert. »Armitage und seine Leute ... und Curly Burlingame. Curly Burlingame?«

»Ein wahrer amerikanischer Held«, sagte Edward.

»Jetzt werden einige Stromausfälle in den Vereinigten Staaten gemeldet«, fuhr Albert fort. »Keine große Sache, heißt es. Die Rede ist von mutmaßlicher Sabotage, aber nur Einzelfälle.«

Homer lächelte grimmig. »Sabotage, ja. Einzelfälle, nein.«

»Stromausfälle auch in Europa. Sie wissen noch nicht, was sie davon halten sollen.«

Homer winkte geistesabwesend. »Schalt aus, Albert. Worauf es jetzt ankommt, passiert nicht dort draußen, sondern hier drinnen.«

Albert legte den StratCom-Empfänger auf den Couchtisch und sah wieder aus dem Fenster. Es gab überhaupt kein künstliches Licht mehr, nur Sterne und das fahle rosarote Leuchten, wie das schwache Licht etwa eine Stunde vor Sonnenaufgang. Aber es ließ sich in allen Richtungen beobachten und war am südlichen Horizont ein wenig stärker.

»Meine Güte«, sagte Albert. »Was haben wir getan?«

Homer saß in der Dunkelheit. »Was haben wir getan? Was habe ich getan? Wir haben etwa acht Millionen Menschen zum Tod verurteilt – sie werden im Lauf der nächsten Monate sterben. Acht Millionen.« Er sprach leise, schwieg einige Sekunden und fügte dann noch leiser hinzu: »Im Vergleich mit uns war Hitler ein Dilettant.«

Loren hielt den Atem an. Kelly beugte sich zu Homer

hinab, streckte die Hände nach seinen Seiten aus und ... kitzelte ihn. Homer war unglaublich kitzlig. Er zuckte heftig zusammen. »Dummer alter Kerl«, sagte Kelly. »Du hast gerade St. Louis gerettet und sechzig Millionen Menschen überall auf der Welt. Das geht aus unseren Berechnungen hervor. Du hast die Atmosphäre der Erde vor radioaktiver Verseuchung bewahrt. Vielleicht hast du sogar das ganze Leben auf diesem Planeten gerettet.«

»Es stimmt, Homer«, sagte Loren. »Du bist der größte Held aller Zeiten.«

»Aber all das Sterben, das jetzt beginnt ...«, wandte er ein.

»Daran ist jemand anderer schuld.« Edward legte Homer den Arm um die Schulter. »Rupert Paule. Er und General Simpson und all die anderen. Es ist ihre Schuld, Homer.«

Homer nickte, wirkte aber nicht sonderlich überzeugt.

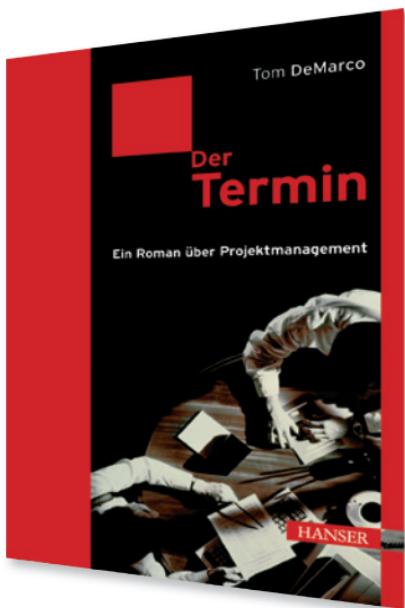
Loren löste die Batterie vom Effektor und sah seine Annahmen bestätigt, als das winzige rosarote Licht in der Kartenmitte blieb - es bezog seine Energie vom irdischen Magnetfeld. Der kleine Apparat auf der Karte war nötig für die Übertragung der Störung, die den Effekt erhielt. Solange er aktiv und ausgerichtet blieb, dauerte der Effekt an. Loren entfernte auch die Arretierung, damit sie nicht unabsichtlich ausgelöst werden konnte, schloss den Kasten und schloss ihn ab.

Edward verteilte Taschenlampen aus einer Box mit Vorräten, die sie Stunden zuvor hochgetragen hatten. Außerdem gab er jedem eine Liste mit detaillierten Anweisungen für die nächsten Schritte.

»Es wartet viel Arbeit auf uns, Leute, und wir haben nur ein paar Stunden Zeit, alles zu erledigen. Packen wir's an.«

(Ende 15. Kapitel)

Weitere Bücher von Tom DeMarco



Tom DeMarco
Der Termin
ISBN 978-3-446-41439-6



Tom DeMarco, Timothy Lister
Wien wartet auf Dich!
ISBN 978-3-446-43895-8

Tom DeMarco, Tim Lister
Bärentango
ISBN 978-3-446-22333-2

Tom DeMarco
Spielräume
ISBN 978-3-446-21665-5

»Der Weltuntergang steht bevor, aber nicht so, wie Sie denken. Dieser Krieg jagt nicht alles in die Luft, sondern schaltet alles ab.«

Im obersten Stock der Cornell University's Clark Hall stehen der Physiker Homer Layton und seine drei jungen Assistenten vor einem Durchbruch, der es ermöglicht, die Zeit etwas langsamer ablaufen zu lassen. Sie vermuten, dass der sogenannte Layton-Effekt keinen praktischen Nutzen haben wird, rechnen aber damit, dass die von ihnen geplante Abhandlung einem Paukenschlag in der Welt der theoretischen Physik gleichkommen wird. Doch dann bemerkt Loren Martine, jüngstes Mitglied von Homers Team, etwas Seltsames: Wird die Zeit verlangsamt, reicht die in Brennstoffen gespeicherte Energie nicht mehr für ein plötzliches Feuer. Dinge können noch immer brennen, wenn auch langsamer, aber nichts kann mehr explodieren. Die Wissenschaftler stellen sich eine Art Layton-Effekt-Taschenlampe vor, die das Abfeuern einer Waffe verhindert. Ihnen wird klar, dass man auch die Explosion einer Bombe oder gar einen ganzen Krieg verhindern könnte.



Sie möchten mehr über Tom DeMarco und
seine Bücher erfahren.
Einfach Code scannen oder reinklicken unter
www.hanser-fachbuch.de/special/demarco

WEBSOCKETS //

- Für Webentwickler mit Grundkenntnissen, Studenten der Informatik, Softwarearchitekten und IT-Konzepte
- Umfassender Einstieg in die WebSocket-Technik: Grundlagen, Performance, Sicherheit, Test-Verfahren
- Mit zahlreichen Beispielen auf Basis ganz unterschiedlicher Werkzeuge, Technologien, Sprachen und Frameworks
- Die Beispiele aus dem Buch sowie Infos zu aktuellen Entwicklungen unter:
<http://websocket101.org/>

Dieses Buch führt Sie umfassend in die WebSocket-Technik und die damit einhergehenden neuen Entwicklungsmöglichkeiten ein. Unter den zahlreichen exemplarischen Anwendungen finden sich Beispiele auf Basis von Node.js, Vert.x, und JSR 356, als Programmiersprachen werden Java und JavaScript eingesetzt.

Nach einer Einführung in die notwendigen Grundlagen von HTTP lernen Sie zunächst die Mechanismen für höhere Interaktivität und Echtzeitfähigkeit bei Webanwendungen kennen. Weiter geht es mit dem WebSocket-Protokoll und der WebSocket-API. An dieser Stelle werden Sie mit JavaScript erste Beispiele für WebSocket-Clientanwendungen in Webbrowsersn programmieren. Es folgen WebSocket-Implementierungen auf der Serverseite auf Basis gebräuchlicher Frameworks.

Weitere Themen sind das Testen von verteilten WebSocket-basierten Applikationen, Performance-Eigenschaften und – ganz wichtig – Sicherheitsaspekte, insbesondere wenn die Anwendung aus verteilten Komponenten zusammengesetzt ist, die über offene Netze miteinander gekoppelt sind.

Schließlich werden Sie verschiedene größere und vollständige Anwendungen implementieren: eine generische Fernsteuerung für Webanwendungen, ein Chatsystem, eine Heatmap für Usability-Tests und eine Überwachungskamera per Webcam.

Prof. Dr. Luigi LO IACONO lehrt an der Fakultät für Informations-, Medien- und Elektrotechnik an der FH Köln. **Peter Leo GORSKI** und **Hoai Viet NGUYEN** sind wissenschaftliche Mitarbeiter an derselben Fakultät.



Die Forschungs- und Entwicklungsinteressen des Autorenteams liegen im Bereich verteilter Anwendungen auf Basis von Webtechnologien und deren Sicherheit.

AUS DEM INHALT //

- HTTP-Grundlagen
- Techniken für höhere Interaktivität und Echtzeitfähigkeit
- Die Leitung: Das IETF WebSocket-Protokoll
- Der Client: Die W3C WebSocket-API
- Der Server: Sprachliche Vielfalt
- WebSockets in der Praxis: Performance & Skalierbarkeit, Sicherheit, Gefahren & Probleme
- Vier komplette Beispielanwendungen

UNSER BUCHTIPP FÜR SIE //



Ziebler, Web Hacking
2014, 217 Seiten, FlexCover, € 29,99.
ISBN 978-3-446-44017-3

€ 34,99 [D] | € 36,00 [A]
ISBN 978-3-446-44371-6



9 783446 443716

HANSER