

Sebastian Springer



Node.js

Das umfassende Handbuch

- ▶ Professionelle, echtzeitfähige Anwendungen entwickeln
- ▶ Installation, Grundlagen, Best Practices
- ▶ Debugging, Skalierung, Qualitätssicherung

Sebastian Springer

Node.js

Das umfassende Handbuch

Liebe Leserin, lieber Leser,

mit Node.js wird bisher Unvorstellbares möglich: Sie können serverseitige Webapplikationen mit JavaScript entwickeln. Die Vorteile liegen auf der Hand, denn Sie können nun JavaScript einheitlich für Client und Server verwenden. Einfache, hochperformante und echtzeitfähige Webanwendung sind für Sie nun leicht zu realisieren. Und aufgrund der schnellen Umsetzung sparen Sie auch noch wertvolle Zeit.

Wie das geht, zeigt Ihnen unser Autor Sebastian Springer. Aufgrund seiner Erfahrungen, gibt er Ihnen einen leichten Einstieg in das JavaScript-Framework. Viele Hinweise und verständliche Beispiele zeigen Ihnen, wie und wo Sie von Node.js profitieren können. Durch den klaren Aufbau eignet sich das Buch sowohl zum Erlernen von Node.js als auch zum Nachschlagen im täglichen Gebrauch. Ich wünsche Ihnen bei der Programmierung und Entwicklung mit Node.js viel Spaß und Erfolg!

Um die Qualität unserer Bücher zu gewährleisten, stellen wir stets hohe Ansprüche an Autoren und Lektorat. Falls Sie dennoch Anmerkungen und Vorschläge zu diesem Buch formulieren möchten, so freue ich mich über Ihre Rückmeldung.

Ihr Stephan Mattescheck

Lektorat Galileo Computing

stephan.mattescheck@galileo-press.de

www.galileocomputing.de

Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

1	Grundlagen	19
2	Installation	39
3	Ein erstes Beispiel	57
4	Anpassung und Erweiterung	71
5	Arbeiten mit Dateien	161
6	Kommunikation	187
7	Asynchrone Programmierung	223
8	Anbindung von Datenbanken	257
9	Qualitätssicherung	279
10	Skalierbarkeit und Deployment	311
11	Sicherheitsaspekte	335
12	HTTP-Server	349
13	Socket-Server	365
14	Multi-Page Webapplikationen	385
15	Single-Page Webapplikationen	409
16	Echtzeit-Webapplikationen	433

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit. Eine E-Mail mit Ihrem Lob oder Tadel senden Sie direkt an den Lektor des Buches: stephan.mattescheck@galileo-press.de. Im Falle einer Reklamation steht Ihnen gerne unser Leserservice zur Verfügung: service@galileo-press.de. Informationen über Rezensionen- und Schulungsexemplare erhalten Sie von: britta.behrens@galileo-press.de.

Informationen zum Verlag und weitere Kontaktmöglichkeiten finden Sie auf unserer Verlagswebsite www.galileo-press.de. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher versandkostenfrei bestellen.

An diesem Buch haben viele mitgewirkt, insbesondere:

Lektorat Stephan Mattescheck, Erik Lipperts

Korrektur Angelika Glock

Fachgutachten Andreas Wenk

Herstellung Martin Pätzold

Layout Vera Brauner

Einbandgestaltung Barbara Thoben, Köln

Satz Ill-satz, Husby

Druck Beltz Druckpartner, Bad Langensalza

Dieses Buch wurde gesetzt aus der TheAntiquaB (9,35/13,7 pt) in FrameMaker.

Gedruckt wurde es auf chlorfrei gebleichtem Offsetpapier (90 g/m²).

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch *Eppur si muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

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.

ISBN 978-3-8362-2119-1

1. Auflage 2013

© Galileo Press, Bonn 2013

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien.

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Inhalt

Geleitwort des Fachgutachters	13
Vorwort	15
1 Grundlagen	19
1.1 Die Geschichte von Node.js	19
1.2 Die Vorteile von Node.js	22
1.3 Einsatzgebiete von Node.js	23
1.4 Das Herzstück – die V8-Engine	24
1.4.1 Das Speichermodell	26
1.4.2 Zugriff auf Eigenschaften	26
1.4.3 Maschinencodgenerierung	28
1.4.4 Garbage Collection	30
1.5 Bibliotheken um die Engine	31
1.5.1 Eventloop	32
1.5.2 Eingabe und Ausgabe	33
1.5.3 libuv	34
1.5.4 DNS	35
1.5.5 Crypto	36
1.5.6 Zlib	36
1.5.7 HTTP-Parser	36
1.6 Zusammenfassung	37
2 Installation	39
2.1 Installation von Paketen	40
2.1.1 Linux	41
2.1.2 Windows	44
2.1.3 Mac OS X	48
2.2 Kompilieren und installieren	54
2.3 Zusammenfassung	56

3	Ein erstes Beispiel	57
3.1	Der interaktive Modus	57
3.2	Die erste Applikation	61
3.3	Zusammenfassung	70
4	Anpassung und Erweiterung	71
4.1	Node.js-Module	71
4.1.1	Modularer Ansatz	71
4.1.2	Stabilitätsindex	73
4.1.3	Verfügbare Module	75
4.2	Basismodule	94
4.2.1	Globale Objekte	94
4.2.2	Utility	96
4.2.3	Events	100
4.2.4	OS	101
4.2.5	Process	103
4.2.6	Buffer	107
4.2.7	Path	108
4.3	Eigene Klassen erstellen und einbinden	110
4.3.1	Eigene Module in Node.js	115
4.3.2	Eigene Node.js-Module	115
4.3.3	Das Modules-Modul	117
4.3.4	Der Modulloader	118
4.3.5	Die require-Funktionalität	121
4.3.6	Die Time-Tracker-Applikation	122
4.4	NPM	136
4.4.1	Pakete suchen	137
4.4.2	Pakete installieren	138
4.4.3	Installierte Pakete anzeigen	142
4.4.4	Pakete verwenden	142
4.4.5	Pakete aktualisieren	149
4.4.6	Pakete entfernen	151
4.4.7	Die wichtigsten Kommandos im Überblick	151
4.4.8	Der Aufbau eines Moduls	152
4.4.9	Eigene Pakete erstellen	155
4.5	Zusammenfassung	159

5	Arbeiten mit Dateien	161
5.1	Dateien lesen	163
5.2	Dateien schreiben	169
5.3	Verzeichnisoperationen	175
5.4	Weiterführende Operationen	180
5.4.1	watch	182
5.4.2	Zugriffsberechtigungen	183
5.5	Zusammenfassung	185
6	Kommunikation	187
6.1	Der Webserver	188
6.1.1	Das Server-Objekt	188
6.1.2	Server-Events	190
6.1.3	Das Request-Objekt	193
6.1.4	Das Response-Objekt	197
6.2	HTTP-Client mit Node.js	201
6.2.1	Der http.Agent	203
6.2.2	Die Anfrage-Optionen	203
6.2.3	Die Klasse ClientRequest	205
6.2.4	Die Antwort des Servers	208
6.3	Umgang mit URLs	210
6.4	Streams in Node.js	212
6.4.1	Readable Stream	213
6.4.2	Writable Stream	214
6.5	Sockets	216
6.5.1	TCP	216
6.5.2	UNIX Domain Sockets	219
6.6	Zusammenfassung	221
7	Asynchrone Programmierung	223
7.1	Grundlagen asynchroner Programmierung	223
7.1.1	Das child_process-Modul	225

7.2	Externe Kommandos asynchron ausführen	227
7.2.1	Die exec-Methode	227
7.2.2	Die spawn-Methode	230
7.3	Fork	232
7.4	Das cluster-Modul	237
7.4.1	Der Masterprozess	237
7.4.2	Die Workerprozesse	241
7.5	Die Grundlagen von Promises	244
7.6	Promises in CommonJS	247
7.7	Libraries	248
7.8	Q	248
7.8.1	Deferred	249
7.8.2	Node.js und Promises	250
7.9	PromisedIO	253
7.10	Zusammenfassung	255

8 Anbindung von Datenbanken 257

8.1	Node.js und relationale Datenbanken	258
8.1.1	MySQL	258
8.1.2	SQLite	264
8.2	Node.js und nicht-relationale Datenbanken	269
8.2.1	Redis	270
8.2.2	MongoDB	274
8.3	Zusammenfassung	277

9 Qualitätssicherung 279

9.1	Assertion Testing	279
9.2	jasmine-node	284
9.3	nodeunit	289
9.4	Praktisches Beispiel von Unittests mit nodeunit	294

9.5	Statische Codeanalyse	298
9.5.1	JSLint	298
9.5.2	PMD CPD	300
9.6	Node.js Debugger	303
9.6.1	Navigation im Debugger	304
9.6.2	Informationen im Debugger	305
9.6.3	Breakpoints	307
9.7	Debugging in der Entwicklungsumgebung	309
9.8	Zusammenfassung	309
10	Skalierbarkeit und Deployment	311
10.1	Deployment	312
10.1.1	Einfaches Deployment	312
10.1.2	Dateisynchronisierung mit rsync	314
10.1.3	Die Applikation als Dienst	315
10.1.4	node_modules beim Deployment	317
10.1.5	Applikationen mit dem Node Package Manager installieren	318
10.1.6	Pakete lokal installieren	319
10.1.7	Toolunterstützung mit Grunt	320
10.2	Skalierung	325
10.2.1	Kindprozesse	326
10.2.2	Loadbalancer	329
10.2.3	Node in der Cloud	332
10.3	Zusammenfassung	334
11	Sicherheitsaspekte	335
11.1	Filter Input und Escape Output	335
11.2	Absicherung des Servers	336
11.2.1	Benutzerberechtigungen	336
11.2.2	Single-Threaded-Ansatz	337
11.2.3	Denial of Service	340
11.2.4	SQL-Injections	341
11.2.5	Eval	343
11.2.6	Method Invocation	344

11.3	Schutz des Clients	346
11.3.1	Cross-Site-Scripting	346
11.4	Zusammenfassung	348
12	HTTP-Server	349
12.1	GET – lesender Zugriff	351
12.2	POST – Anlegen neuer Ressourcen	354
12.3	PUT – Aktualisierung bestehender Daten	357
12.4	DELETE – Löschen vorhandener Daten	359
12.5	Accept-Header	360
12.6	Zusammenfassung	363
13	Socket-Server	365
13.1	UNIX-Sockets	366
13.1.1	Zugriff auf den Socket	367
13.1.2	Bidirektionale Kommunikation	369
13.2	TCP-Sockets	371
13.2.1	Datenübertragung	373
13.2.2	Dateiübertragung	374
13.2.3	Flusssteuerung	376
13.2.4	Duplex	377
13.2.5	Pipe	378
13.3	UDP-Sockets	379
13.3.1	Grundlagen eines UDP-Servers	379
13.3.2	Beispiel zum UDP-Server	381
13.4	Zusammenfassung	383
14	Multi-Page Webapplikationen	385
14.1	Das Web Application-Framework Express	385
14.1.1	Installation	386
14.1.2	Setup und Initialisierung der Applikation	386

14.1.3	Routing	387
14.1.4	Middleware	392
14.2	Templates mit Jade	394
14.2.1	Installation	395
14.2.2	Ein einfaches Beispiel	396
14.2.3	Verwendung von Jade	398
14.2.4	Integration in express.js	402
14.3	Auslieferung von statischen Inhalten	404
14.4	Zusammenfassung	407
15	Single-Page Webapplikationen	409
15.1	Die Aufgabenstellung	409
15.2	Setup	410
15.2.1	Ordnerstruktur	410
15.2.2	Die Datenbank	411
15.2.3	Abhängigkeiten	411
15.2.4	Clientbibliotheken	412
15.3	Die Applikation	415
15.3.1	Login	415
15.3.2	Liste der vorhandenen Datensätze	420
15.3.3	Neue Datensätze anlegen	427
15.4	Zusammenfassung	431
16	Echtzeit-Webapplikationen	433
16.1	Die Beispielapplikation	434
16.2	Setup	434
16.3	Websockets	439
16.3.1	Die Serverseite	440
16.3.2	Die Clientseite	443
16.3.3	Userliste	445
16.3.4	Logout	449

16.4 Socket.IO 450

 16.4.1 Installation und Einbindung 451

 16.4.2 Socket.IO-API 452

16.5 Zusammenfassung 456

Index 457

Geleitwort des Fachgutachters

Als ich 2009 die JSConfEU in Berlin besuchte, war mir noch nicht klar, dass ich einem bahnbrechenden Ereignis beiwohnen würde. Ich rede von Ryan Dahls Talk über eine neue Software, die er programmiert hat: Node.js. Der wohl wichtigste Satz, den er in diesem Vortrag gesagt hat, ist: »I/O needs to be done differently«. Die Rede ist von »Nonblocking I/O«. Und genau das tut Node.js mit seiner eventgetriebenen Implementierung.

Node.js ist mittlerweile der De-facto-Standard für serverseitiges JavaScript und stellt die Grundlage für viele Tausend Websites und viele sehr erfolgreiche Projekte dar. Sie, als Leser dieses Buches, haben mit großer Wahrscheinlichkeit bereits Erfahrung mit der einen oder anderen Implementierung sammeln können.

Warum eigentlich JavaScript? Das ist doch irgendwie... komisch? Ja – irgendwie schon. Aber bedenken Sie dabei, dass JavaScript eine der am weitesten verbreiteten Programmiersprachen überhaupt ist. Und wie Douglas Crockford bereits festgestellt hat, gibt es auch die »Good Parts« in JavaScript. Es spricht also nichts dagegen, tiefer in die Materie einzusteigen.

Und genau dies bietet Ihnen Sebastian Springer mit dem vorliegenden Buch über Node.js – einen hervorragenden Einstieg in die serverseitige Implementierung von Applikationen mit Node.js. Aber natürlich kommt auch das Frontend nicht zu kurz. Und der Clou dabei? Alles ist in derselben Programmiersprache geschrieben.

Sebastian Springer führt Sie Schritt für Schritt in alle Themen von und mit Node.js ein. Er zeigt Ihnen auf überaus anschauliche Art und Weise, wie ein Webserver, ein TCP-Server oder sogar ein UDP-Server mit Node.js gebaut werden, um nur wenige der Themen aus dem Buch anzusprechen. Dabei erklärt er nicht nur, was Sie tun müssen, sondern geht auch auf die Hintergründe ein. Er zeigt Ihnen Ressourcen für das weiterführende Studium. Überzeugen Sie sich selbst, und werfen Sie einen Blick in die Inhaltsangabe. Es erwartet Sie ein Buch mit vielen Beispielen aus der Praxis. Es gilt also die Devise, die Tastatur zu malträtieren und zu programmieren. Alle Beispiele sind so gut, dass es für Sie ein Leichtes sein wird, daraus komplette Applikationen zu erstellen.

Es hat mir sehr viel Spaß gemacht, dieses Buch zu begutachten. Sebastian Springer ist ein exzellenter »WebWorker«, mit sehr umfangreichem Wissen über Node.js, JavaScript und das Web als solches. Freuen Sie sich auf viel Stoff zum Lernen. Ich jedenfalls habe es getan.

Hamburg,
Andreas Wenk

Senior Software Developer – SinnerSchrader Group Hamburg

Vorwort

Node.js ist eine serverseitige JavaScript-Plattform, die aktuell in aller Munde ist. Doch was genau ist Node.js? Welche Probleme lassen sich mit dieser Plattform lösen? Kann man Node.js auch im professionellen Umfeld einsetzen? Diesen und vielen weiteren Fragen widmet sich dieses Buch.

Mit Node.js hält JavaScript als Programmiersprache auf der Serverseite Einzug und damit auch zahlreiche Features und Vorgehensweisen der Sprache, die Sie aus der clientseitigen Programmierung kennen. So verfolgt Node.js beispielsweise einen event-getriebenen Ansatz. Das bedeutet, dass Sie an Ereignisse, die innerhalb Ihrer Applikation auftreten, Callback-Funktionen binden können. Dieses Konzept zieht sich wie ein roter Faden durch die gesamte Plattform. Auch das Programmierparadigma von Node.js unterscheidet sich grundlegend von dem anderer Sprachen, die im Webumfeld verwendet werden. Im Gegensatz zu Java oder PHP, die einen klassenbasierten Ansatz verfolgen, existiert dieses Konzept in JavaScript nicht. Stattdessen kommen hier Prototypen und Objekte zum Einsatz. Mit diesem Ansatz steigt auch die Dynamik innerhalb Ihrer Applikation. Und genau diese Dynamik ist eines der wesentlichen Merkmale von Node.js. Die Node.js-Plattform bietet ein sehr flexibles, mehrschichtiges Modulsystem. Im Verlauf dieses Buches lernen Sie, wie Sie die Vorteile ausnutzen können, die Ihnen diese Plattform bietet.

Ich selbst beschäftige mich mittlerweile seit mehreren Jahren mit der Entwicklung von Applikationen auf Basis von Node.js. Begonnen habe ich mit der Webentwicklung unter PHP. Und wie die meisten Webentwickler konnte auch ich mich dem Einfluss von JavaScript nicht entziehen. Neben dem Aspekt der Umsetzung von Applikationslogik im Frontend habe ich mich im Bereich JavaScript vor allem auch mit der Qualitätssicherung beschäftigt. Die Node.js-Plattform und damit die Möglichkeit, JavaScript auch auf dem Server einzusetzen, hat recht früh mein Interesse geweckt. Mir stellte sich, wie so vielen anderen Entwicklern auch, die Frage: Kann man Node.js bereits in Applikationen produktiv nutzen? Dieser Frage näherte ich mich in der Vergangenheit schrittweise an. Zu Beginn standen Implementierungen verschiedener Beispielapplikationen in Node.js und das Austesten des Toolsets, das für Node.js zur Verfügung steht, im Vordergrund. Die Anforderungen an die Werkzeuge für Node.js reichen von der Verfügbarkeit einer Entwicklungsumgebung über Werkzeuge wie Debugger bis hin zu Plattformen für Continuous Integration. In einem ersten Projekt kam Node.js bei mir im Verbund mit anderen Programmiersprachen zum Einsatz. Node.js übernahm in diesem Fall die Realtime-Kommunikation. Nachdem sich über einen längeren Zeitraum gezeigt hat, dass Node.js in einem solchen Umfeld stabil eingesetzt werden kann, stand einer weiteren Verwendung

auch bei anderen Projekten nichts mehr im Wege. Und so setze ich Node.js mittlerweile bei einer ganzen Reihe von Projekten ein.

Dieses Buch soll Ihnen dabei helfen, Node.js verstehen zu lernen, und Ihnen wertvolle Hinweise geben, wann und wo Sie die Plattform verwenden können. Außerdem soll es Ihnen den Einstieg in die Welt von Node.js erleichtern und als Nachschlagewerk bei der täglichen Arbeit dienen. Zu diesem Zweck ist das Buch in verschiedene Kapitel unterteilt. Sie haben die Möglichkeit, das Buch Kapitel für Kapitel durchzuarbeiten oder sich bestimmte Kapitel herauszugreifen und lediglich diese zu bearbeiten. Jedes Kapitel verfügt über ein oder mehrere unabhängige Beispiele, die den dort jeweils behandelten Stoff verdeutlichen.

Aufbau des Buches

Eingeleitet wird das Buch mit einem Kapitel über die Entstehungsgeschichte, potenzielle Einsatzgebiete und technische Grundlagen der Node.js-Plattform.

Nach der eher theoretischen Einführung in die Node.js-Plattform sehen Sie im zweiten Kapitel, wie Sie Node.js auf den verschiedenen Betriebssystemen installieren können. Dieses Kapitel richtet sich vor allem an diejenigen Leser, die noch keine Installation von Node.js auf ihrem System durchgeführt haben.

Im dritten Kapitel sehen Sie, wie Sie Node.js grundsätzlich benutzen können. Hier wird Ihnen zum einen der REPL, der interaktive Modus von Node.js, vorgestellt und zum anderen die Ausführung von JavaScript-Code, der in Dateiform vorliegt.

Das vierte Kapitel beschäftigt sich mit dem Modulsystem der Node.js-Plattform. Wie Sie bereits erfahren haben, ist dieses Modulsystem in mehrere Stufen aufgeteilt. Einen ähnlichen Aufbau weist auch dieses Kapitel auf. Im ersten Teil erhalten Sie einen Überblick über die verschiedenen internen Node.js-Module. Im Anschluss daran sehen Sie, wie Sie auf dieser Basis Ihre eigene Applikation aufbauen können. Den Abschluss dieses Kapitels bildet die Betrachtung des Node Package Managers.

Mit Node.js bewegen Sie sich mit JavaScript nicht mehr im Browser, sondern direkt auf einem Betriebssystem. Damit nutzen Sie nicht mehr das DOM zur Interaktion, sondern greifen direkt auf das Dateisystem zu. Das fünfte Kapitel beschäftigt sich mit der Schnittstelle, die Ihnen Node.js für den Zugriff auf das Dateisystem zur Verfügung stellt.

Das Modulsystem und der Zugriff auf das Dateisystem helfen Ihnen bei der Implementierung von Webapplikationen nur bedingt weiter. Im Web spielt die Kommunikation zwischen Applikationen und Systemen, aber auch mit einem Benutzer eine zentrale Rolle. Das sechste Kapitel stellt Ihnen die Kommunikation über Datenströme im Allgemeinen und das `http`-Modul von Node.js im Speziellen vor.

Node.js verfolgt den Ansatz, dass alle Operationen parallel ausgeführt werden, bis auf den eigentlichen Quellcode der Applikation. Das bedeutet, dass Sie sich selbst um die Parallelisierung kümmern müssen, wenn Sie eine solche beim Aufbau Ihrer Applikation benötigen. Node.js unterstützt Sie bei dieser Aufgabe mit dem `child_process`- und dem `cluster`-Modul. Außerdem existiert mit Promises ein Ansatz, geschachtelte Callback-Funktionen aufzulösen. Auch dieses Konzept wird Ihnen im Verlauf dieses Kapitels vorgestellt.

Mit dem Dateisystemmodul haben Sie bereits eine Möglichkeit kennengelernt, Daten zu speichern und auszulesen. Mit Datenbanken wird Ihnen im achten Kapitel eine weitere Variante vorgestellt, mit der Sie unter Node.js mit Daten arbeiten können. In diesem Kapitel erfahren Sie, wie Sie mit Node.js MySQL, SQLite, Redis und MongoDB anbinden können.

Implementieren Sie Ihre gesamte Applikation oder bestimmte Teile davon in Node.js, müssen Sie auch auf die Qualität Ihrer Applikation achten. Das neunte Kapitel stellt Ihnen neben den Möglichkeiten der statischen und dynamischen Codeanalyse Ihres Quellcodes auch den in Node.js integrierten Debugger vor. Im Verlauf dieses Kapitels lernen Sie, wie Sie Unittests schreiben und mithilfe des Debuggers auftretende Fehler in Ihrer Applikation aufspüren können.

Das zehnte Kapitel widmet sich dem Thema der Skalierbarkeit und dem Deployment von Node.js-Applikationen. Da dieses Thema sehr umfangreich ist, wird hier lediglich ein Überblick über verschiedene Themen wie das Rollout einer Applikation oder den Einsatz von Loadbalancer gegeben. Zudem finden Sie hier mehrere Verweise zu weiterführender Literatur.

Node.js stammt aus dem Webumfeld. Applikationen aus diesem Bereich sind häufig über das Internet erreichbar und damit potenzielle Opfer von Attacken. Im elften Kapitel dieses Buches erfahren Sie mehr über die Sicherheitsaspekte unter Node.js. Hier gilt vor allem: Filter Input und Escape Output.

Ab dem zwölften Kapitel widmet sich dieses Buch vor allem der Entwicklung von Webapplikationen. So sehen Sie im zwölften Kapitel, wie Sie einen REST-Server implementieren und auf diese Weise korrekt auf Anfragen mit den verschiedenen HTTP-Methoden antworten können.

Neben dem HTTP-Server können Sie auch auf weitere Implementierungen von Kommunikationsservern zugreifen. Dies sind zum einen dateibasierte Server auf UNIX Domain Sockets und zum anderen netzwerkbasierte Server mit TCP oder UDP. Das dreizehnte Kapitel widmet sich dieser Thematik und zeigt Ihnen, wie Sie bei der Umsetzung vorgehen müssen.

Die klassische Art der Implementierung von Webapplikationen besteht darin, dass Sie pro URL eine Seite ausliefern. Das vierzehnte Kapitel stellt Ihnen vor, wie Sie mit

dem Web Application-Framework Express und der Template-Engine Jade derartige Webapplikationen erstellen können. Hier lernen Sie die Grundlagen dieser beiden Hilfsmittel kennen, auf denen die folgenden Kapitel aufbauen.

Im fünfzehnten Kapitel entwickeln Sie auf Basis von Express und Jade eine Single-Page Webapplikation. Im Frontend kommt das JavaScript-Framework Backbone.js zum Einsatz.

Im sechzehnten und letzten Kapitel sehen Sie, wie Sie mit Node.js echtzeitfähige Webapplikationen mit Websockets umsetzen können. Sollten Sie eine Anwendung implementieren, die in der Lage sein muss, auch ältere Browser zu unterstützen, können Sie auf die Bibliothek Socket.IO zurückgreifen, die Ihnen einen Fallback-Mechanismus auf andere Technologien zur Verfügung stellt.

Danksagung

Ich möchte hier abschließend noch allen an diesem Buch beteiligten Personen herzlich danken. Allen voran meinem Fachgutachter Andreas Wenk für seine Anmerkungen, Tipps und Tests.

Außerdem gilt mein Dank Angelika Glock für das Gegenlesen meines Buches.

Auch dem ganzen Team von Galileo Press möchte ich danken und hier vor allem Stephan Mattescheck und Erik Lipperts.

Schließlich möchte ich auch noch meiner Frau Alexandra meinen herzlichen Dank für ihre Geduld und Unterstützung sagen.

Kapitel 1

Grundlagen

Man darf nicht das, was uns unwahrscheinlich und unnatürlich erscheint, mit dem verwechseln, was absolut unmöglich ist.
– Carl Friedrich Gauß

JavaScript ist als Programmiersprache mittlerweile allgegenwärtig. Der Siegeszug dieser Scriptsprache begann mit der Integration im Webbrowser von Netscape im Jahre 1995. Brendan Eich, der Entwickler der damals noch als LiveScript benannten Sprache, entwickelte sie, um kleinere Aufgaben wie Formularvalidierungen direkt im Browser durchzuführen. Seit dieser Zeit verbreitete sich die Sprache erst über den Internet Explorer und ist mittlerweile auf sämtlichen grafischen Browsern verfügbar. Der nächste Schritt bestand darin, dass die JavaScript-Engines aus den Browsern herausgelöst und in anderer Form eingesetzt wurden, wie es konkret mit der V8-Engine aus dem Chrome-Browser in Node.js der Fall ist. Mit diesem Ansatz bringt Node.js JavaScript auf den Server. Die Plattform ist dabei keine radikale Neuentwicklung, sondern vielmehr eine Sammlung verschiedener Bibliotheken, die sich bereits in der Praxis bewährt haben.

Wenn Sie sich auch schon einmal die Frage gestellt haben, was hinter Node.js steckt und warum so viele Unternehmen mittlerweile JavaScript sogar serverseitig einsetzen, dann sind Sie hier genau richtig. Dieses Kapitel soll Ihnen einen Einblick in die Entstehung von Node.js und die zugrunde liegenden Konzepte geben. Dieses Kapitel ist allerdings nicht zwingend erforderlich für das Verständnis der folgenden Kapitel. Sollten Sie also nicht interessiert daran sein, welche Optimierungen der JavaScript-Engine dazu führen, dass Node.js performant ist, oder welche Bibliotheken neben der V8-Engine noch zum Einsatz kommen, können Sie dieses Kapitel auch überspringen und direkt mit der Installation der Node.js-Plattform in Kapitel 2, »Installation«, beginnen.

1.1 Die Geschichte von Node.js

Damit Sie besser verstehen, was Node.js ist und auch besser nachvollziehen können, wie es zu manchen Entscheidungen bei der Entwicklung gekommen ist, erfahren Sie hier etwas mehr über die Geschichte der Plattform. Die noch relativ junge Entwick-

lung von Node.js ist direkt mit seinem Entwickler, Ryan Dahl, verbunden. Bevor er sich intensiv mit Informatik und der Entwicklung von Node.js auseinandergesetzt hat, war Ryan Dahl Doktorand der Mathematik. Doch irgendwann stellte sich heraus, dass Mathematik nicht das Richtige für ihn war, er brach seine Bemühungen ab und ging nach Südamerika. Mit einem One-Way-Ticket und nur sehr wenig Geld in der Tasche versuchte er, sich mit Englischunterricht durchzuschlagen. In dieser Zeit lernte er einige Webentwickler kennen, die mit einer frühen Version von PHP dynamische Webseiten erstellten. Er erkannte, dass sein Interesse in der Programmierung von Webseiten lag, und über PHP gelangte er schließlich zu Ruby als Programmiersprache. Ryan Dahl sagt über Ruby, dass es die Sprache mit der schönsten Syntax sei, aber doch einige entscheidende Nachteile mit sich bringt. Mit dem auf Ruby aufbauenden Rails-Framework machte er seine ersten größeren Schritte in der Webentwicklung. Das Ergebnis seiner Versuche war allerdings nur eine einzige produktive Webseite. Ryan erkannte schnell das größte Problem von Rails und dem darunterliegenden Ruby: Die Webseiten waren zu langsam, und die CPU seines Rechners war ständig voll ausgelastet. Rails war nicht in der Lage, ohne Workarounds mit konkurrierenden Anfragen umzugehen, da der darunterliegende Kern mit Ruby einen Single-Threaded-Ansatz verfolgte, also nur die Ausführung eines bestimmten Teils der Programmlogik zu einem Zeitpunkt erlaubt und nicht wie im Multi-Threaded-Ansatz mehrere Teile parallel abarbeitet.

Eine wirkliche Inspiration stellte Mongrel dar, ein Webserver für Applikationen, die auf Ruby basieren. Im Gegensatz zu klassischen Webservern reagiert Mongrel auf Anfragen von Nutzern und generiert die Antworten dynamisch, wo sonst lediglich statische HTML-Seiten ausgeliefert werden.

Die Aufgabe, die eigentlich zur Entstehung von Node.js führte, ist vom heutigen Standpunkt aus betrachtet recht trivial. Im Jahr 2005 suchte Ryan Dahl nach einer eleganten Möglichkeit, einen Fortschrittsbalken für Dateiuploads zu implementieren. Mit den damals verfügbaren Technologien waren nur unbefriedigende Lösungen möglich. Zur Übertragung der Dateien wurde für relativ kleine Dateien das HTTP-Protokoll und für größere Dateien das FTP-Protokoll genutzt. Der Status des Uploads wurde mithilfe von Long Polling abgefragt. Das ist eine Technik, bei der der Client langlebige Requests an den Server sendet und dieser den offenen Kanal für Rückantworten nutzt. Ein erster Versuch von Ryan Dahl zur Umsetzung einer Progressbar fand in Mongrel statt. Nach dem Absenden der Datei an den Server prüfte er mithilfe einer Vielzahl von Ajax-Requests den Status des Uploads und stellte diesen in einer Progressbar grafisch dar. Störend an dieser Umsetzung waren allerdings der Single-Threaded-Ansatz von Ruby und die große Anzahl an Requests, die benötigt wurden.

Weitere Versuche zur Lösung des Progressbar-Problems folgten, diesmal jedoch in anderen Programmiersprachen. Einen vielversprechenden Ansatz bot eine Umsetzung in C. Hier war Ryan Dahl nicht auf einen Thread begrenzt. C als Programmier-

sprache für das Web hat allerdings einen entscheidenden Nachteil: Es lassen sich recht wenige Entwickler für dieses Einsatzgebiet begeistern. Mit diesem Problem sah sich auch Ryan Dahl konfrontiert und verwarf auch diesen Ansatz nach kurzer Zeit wieder.

Die Suche nach einer geeigneten Programmiersprache zur Lösung seines Problems ging weiter und führte ihn zu funktionalen Programmiersprachen wie Haskell. Der Ansatz von Haskell baut auf Nonblocking I/O auf, das heißt also, dass sämtliche Schreib- und Leseoperationen asynchron stattfinden und die Programmausführung nicht blockieren. Dadurch kann die Sprache im Kern single-threaded bleiben, und es ergeben sich nicht die Probleme, die durch parallele Programmierung entstehen. Es müssen unter anderem keine Ressourcen synchronisiert werden, und es ergeben sich auch keine Problemstellungen, die durch die Laufzeit paralleler Threads verursacht werden. Ryan Dahl war aber auch mit dieser Lösung noch nicht vollends zufrieden und suchte nach weiteren Optionen.

Die endgültige Lösung fand Ryan Dahl dann schließlich im Januar 2009 mit JavaScript. Hier wurde ihm klar, dass diese Scriptsprache sämtliche seiner Anforderungen erfüllen könnte. JavaScript war bereits seit Jahren im Web etabliert, es gab leistungsstarke Engines und eine große Zahl von Programmierern. Und so begann er Anfang 2009 mit der Arbeit an seiner Umsetzung für serverseitiges JavaScript, die Geburtsstunde von Node.js. Ein weiterer Grund, der für die Umsetzung der Lösung in JavaScript sprach, war nach Meinung von Ryan Dahl die Tatsache, dass die Entwickler von JavaScript dieses Einsatzgebiet nicht vorsahen. Es existierte zu dieser Zeit noch kein nativer Webserver in JavaScript, es konnte nicht mit Dateien in einem Dateisystem umgegangen werden, und es gab keine Implementierung von Sockets zur Kommunikation mit anderen Anwendungen oder Systemen. All diese Punkte sprechen für JavaScript als Grundlage für eine Plattform für interaktive Webapplikationen, da noch keine Festlegungen in diesem Bereich getroffen und demzufolge auch noch keine Fehler begangen wurden. Auch die Architektur von JavaScript spricht für eine derartige Umsetzung. Der Ansatz der Top-Level-Functions, also Funktionen, die mit keinem Objekt verknüpft und daher frei verfügbar sind und zudem Variablen zugeordnet werden können, bietet eine hohe Flexibilität in der Entwicklung.

Ryan Dahl wählte also neben der JavaScript-Engine, die für die Interpretation des JavaScript-Quellcodes verantwortlich ist, noch weitere Bibliotheken aus und fügte sie in einer Plattform zusammen. Nachdem sämtliche Komponenten integriert und erste lauffähige Beispiele auf der neuen Node.js-Plattform erstellt waren, benötigte Ryan Dahl eine Möglichkeit, Node.js der Öffentlichkeit vorzustellen. Dies wurde auch nötig, da seine finanziellen Mittel durch die Entwicklung an Node.js beträchtlich schrumpften und er, falls er keine Sponsoren finden sollte, die Arbeit an Node.js hätte einstellen müssen. Als Präsentationsplattform wählte er die JavaScript-Konferenz jsconf.eu im November 2009 in Berlin. Ryan Dahl setzte alles auf eine Karte.

Würde die Präsentation ein Erfolg und fände er dadurch Sponsoren, die seine Arbeit an Node.js unterstützten, könnte er sein Engagement fortsetzen, falls nicht, wäre die Arbeit von fast einem Jahr umsonst. In einem mitreißenden Vortrag stellte er Node.js dem Publikum vor und demonstrierte, wie man mit nur wenigen Zeilen JavaScript-Code einen voll funktionsfähigen Webserver erstellen kann. Als weiteres Beispiel brachte er eine Implementierung eines IRC Chat-Servers mit. Der Quellcode dieser Demonstration umfasste etwa 400 Zeilen. Anhand dieses Beispiels demonstrierte er die Architektur und damit die Stärken von Node.js und machte es gleichzeitig für die Zuschauer greifbar. Als Reaktion auf seine überzeugende Präsentation fand sich in Joyent ein Sponsor für Node.js. Joyent ist ein Anbieter für Software und Service mit Sitz in San Francisco und bietet Hosting-Lösungen und Cloud-Infrastruktur. Mit dem Engagement nahm Joyent die Open-Source-Software Node.js in sein Produktportfolio auf und stellte Node.js im Rahmen seiner Hosting-Angebote seinen Kunden zur Verfügung. Ryan Dahl wurde von Joyent angestellt und ab diesem Zeitpunkt als Maintainer in Vollzeit für Node.js eingesetzt.

Überraschend war Anfang 2012 die Ankündigung Ryan Dahls, sich nach drei Jahren der Arbeit an Node.js schließlich aus der aktiven Weiterentwicklung zurückzuziehen. Er übergab die Leitung der Entwicklung an Isaac Schlueter. Dieser ist wie auch Ryan Dahl Angestellter bei Joyent und an der Entwicklung des Kerns von Node.js beteiligt und unter anderem für die Entwicklung des Node Package Managers verantwortlich. Ihm obliegt nun die Aufgabe, Node.js weiter zu stabilisieren, zu verbreiten und neue Features zu integrieren.

1.2 Die Vorteile von Node.js

Die Entwicklungsgeschichte von Node.js zeigt eine Sache sehr deutlich: Die Entwicklung von Node.js ist direkt mit dem Internet verbunden. Mit JavaScript als Basis haben Sie mit Applikationen, die in Node.js umgesetzt sind, die Möglichkeit, sehr schnell sichtbare Ergebnisse zu erzielen. Neben der schnellen initialen Umsetzung erhalten Sie auch während der Entwicklung von Webapplikationen die Möglichkeit, sehr schnell auf sich ändernde Anforderungen zu reagieren. Da der Kern von JavaScript durch ECMAScript größtenteils standardisiert ist, ist JavaScript eine verlässliche Basis, mit der auch umfangreichere Applikationen umgesetzt werden können. Die verfügbaren Sprachfeatures sind sowohl online als auch in Form von Fachbüchern gut und umfangreich dokumentiert. Außerdem sind viele Entwickler verfügbar, die JavaScript beherrschen und in der Lage sind, auch größere Applikationen mit dieser Sprache umzusetzen. Da bei Node.js mit der V8-Engine die gleiche JavaScript-Engine wie auch bei Google Chrome zum Einsatz kommt, stehen Ihnen auch hier sämtliche Sprachfeatures zur Verfügung, und Entwickler, die im Umgang mit JavaScript geübt sind, können sich relativ schnell in die neue Plattform einarbeiten.

Die lange Entwicklungsgeschichte von JavaScript hat eine Reihe hochperformanter Engines hervorgebracht. Eine Ursache für diese Entwicklung liegt darin, dass die verschiedenen Hersteller von Browsern ihre eigenen Implementierungen von JavaScript-Engines stets weiterentwickelten und es so eine gesunde Konkurrenz auf dem Markt gab, wenn es um die Ausführung von JavaScript im Browser ging. Diese Konkurrenz führte einerseits dazu, dass JavaScript mittlerweile sehr schnell interpretiert wird, und andererseits, dass sich die Hersteller auf gewisse Standards einigten. Node.js als Plattform für serverseitiges JavaScript war seit dem Beginn seiner Entwicklung als Open-Source-Projekt konzipiert. Aus diesem Grund entwickelte sich rasch eine aktive Community um den Kern der Plattform. Diese beschäftigt sich vor allem mit dem Einsatz von Node.js in der Praxis, aber auch mit der Weiterentwicklung und Stabilisierung der Plattform. Die Ressourcen zum Thema Node.js reichen von Tutorials, die Ihnen den Einstieg in die Thematik erleichtern, bis hin zu Artikeln über fortgeschrittene Themen wie Qualitätssicherung, Debugging oder Skalierung. Der größte Vorteil eines Open-Source-Projekts wie Node.js ist, dass Ihnen die Informationen kostenlos zur Verfügung stehen und dass Fragen und Problemstellungen recht schnell und kompetent über verschiedenste Kommunikationskanäle beziehungsweise die Community gelöst werden können.

1.3 Einsatzgebiete von Node.js

Falls Sie noch nicht viele Erfahrungen im Einsatz von Node.js sammeln konnten, haben Sie sich bestimmt auch schon das Öfteren die Frage gestellt: In welchen Situationen greife ich eigentlich auf Node.js zurück? Diese Frage lässt sich nur selten klar und eindeutig beantworten. Grundsätzlich hängt der Einsatz bestimmter Technologien natürlich von der Art der Problemstellung und von den persönlichen Präferenzen und dem Wissensstand der Entwickler ab.

»Hat man einen Hammer in der Hand, sieht alles aus wie ein Nagel«, so lautet ein altes Sprichwort, und es beschreibt sehr gut einen der größten Fehler, den Sie in der Softwareentwicklung begehen können. Sie legen sich eine Lösungsstrategie zurecht und suchen dann erst nach einem Problem. Beginnen Sie auf diese Art mit der Entwicklung einer Applikation, ist es möglich, dass Sie auf die falsche Technologie oder Architektur setzen, was zu einem späteren Zeitpunkt sogar zum Scheitern des Projekts führen kann. Die eigentliche Schwierigkeit ist also die Wahl der richtigen Werkzeuge zu Beginn des Projekts. Ob Sie auf das richtige Pferd gesetzt haben, zeigt sich meist erst, wenn die Applikation wächst und neue Anforderungen von den Nutzern gestellt werden. Geht man von dieser Annahme aus, sollte jede Anwendung mit Java umgesetzt werden, und zwar ohne dass man dabei auf Probleme stößt, da mit dieser Sprache nahezu sämtliche Problemstellungen gelöst werden können. Auch diese Annahme ist irreführend, da die Entwicklung einer Webapplikation mit Java nicht

ganz trivial ist und es auch nicht so viele Java-Entwickler auf dem Markt gibt, um sämtliche Webapplikationen mit ihnen umsetzen zu können.

Die korrekte Vorgehensweise beim Projektstart ist also eine Analyse der Anforderung und ihre Priorisierung. Worum genau handelt es sich bei dem Projekt? Wie viel Budget steht dafür zur Verfügung? Wie schnell muss eine erste Version an die Benutzer ausgeliefert werden können? Diese und noch viele andere Fragen müssen Sie sich zu Beginn stellen und mit diesem Fragenkatalog dann die verfügbaren Technologien hinterfragen und prüfen, ob diese die wichtigsten Anforderungen erfüllen. Für eine derartige Technologieevaluierung sollten Sie auf jeden Fall mindestens zwei potenzielle Lösungen, besser allerdings mehr, in Betracht ziehen, um wirklich die beste Lösungsstrategie zu verwenden. Sie werden jetzt sicher fragen, was das alles denn mit Node.js zu tun hat. Node.js ist keinesfalls die »Silver Bullet«, mit der Sie alle Probleme der Welt lösen können. Die Plattform ist kein Ersatz für einen vollwertigen Webserver, und es ist auch fraglich, ob man bei der Umsetzung einer umfangreichen und geschäftskritischen Anwendung allein auf Node.js setzen will. Durch seine Architektur bietet Node.js eine Reihe von Möglichkeiten, die es von anderen Technologien abhebt. Node.js entstand als Plattform für dynamische Webapplikationen. Hier liegt auch das größte Einsatzgebiet von Node.js, und zwar meist im Verbund mit anderen Technologien, die von Datenbanken bis hin zu großen Java-Enterprise-Applikationen reichen. Eine der großen Stärken von Node.js liegt in der Umsetzung leichtgewichtiger Webservices. Node.js bietet ein eigenes Modul zur Kommunikation über HTTP. So lassen sich sehr schnell vollwertige REST-Schnittstellen schaffen. Noch einfacher gestaltet sich die Kommunikation mit Webservices. Auch hier können Sie auf das HTTP-Modul zurückgreifen. Eine weitere Stärke von Node.js liegt in der Fähigkeit, bidirektional mit anderen Systemen zu kommunizieren. Diese Art der Verbindung ist allerdings nicht nur auf Serversysteme beschränkt, sondern kann über die Websocket-Technologie auch mit Client-Systemen wie Webbrowsern oder Mobilapplikationen stattfinden.

Sie können Node.js allerdings nicht nur im Web einsetzen. Mit der JavaScript-Engine und einer Vielzahl von Modulen kann Node.js auch dazu verwendet werden, Kommandozeilenwerkzeuge umzusetzen. Diese lassen sich über den in Node.js integrierten Paketmanager veröffentlichen und so auch anderen Personen zur Verfügung zu stellen. Die denkbaren Einsatzmöglichkeiten von Node.js sind kaum beschränkt.

1.4 Das Herzstück – die V8-Engine

Damit Sie als Entwickler beurteilen können, ob eine Technologie in einem Projekt eingesetzt werden kann, sollten Sie mit den Spezifikationen dieser Technologie ausreichend vertraut sein. Die nun folgenden Abschnitte gehen auf die Interna von

Node.js ein und sollen Ihnen zeigen, aus welchen Komponenten die Plattform aufgebaut ist und wie Sie diese zum Vorteil einer Applikation verwenden können.

Der zentrale und damit wichtigste Bestandteil der Node.js-Plattform ist die JavaScript-Engine V8, die von Google entwickelt wird. Weitere Informationen finden Sie auf der Seite des V8-Projekts unter <https://code.google.com/p/v8/>. Die JavaScript-Engine ist dafür verantwortlich, den JavaScript-Quellcode zu interpretieren und auszuführen. Für JavaScript gibt es nicht nur eine Engine, stattdessen setzen die verschiedenen Browserhersteller auf ihre eigene Implementierung. Eines der Probleme von JavaScript ist, dass sich die einzelnen Engines etwas unterschiedlich verhalten. Durch die Standardisierung nach ECMAScript wird versucht, einen gemeinsamen verlässlichen Nenner zu finden, sodass Sie als Entwickler von JavaScript-Applikationen weniger Unsicherheiten zu befürchten haben. Die Konkurrenz der JavaScript-Engines führte zu einer Reihe optimierter Engines, die allesamt das Ziel verfolgen, den JavaScript-Code möglichst schnell zu interpretieren. Im Verlauf der Zeit haben sich einige Engines auf dem Markt etabliert. Hierzu gehören unter anderem Chakra von Microsoft, JägerMonkey von Mozilla, Nitro von Apple und die V8-Engine von Google.

In Node.js kommt die V8-Engine von Google zum Einsatz. Diese Engine wird seit 2006 von Google hauptsächlich in Dänemark in Zusammenarbeit mit der Universität in Aarhus entwickelt. Das primäre Einsatzgebiet der Engine ist der Chrome-Browser von Google, in dem sie für die Interpretation und Ausführung von JavaScript-Code verantwortlich ist. Das Ziel der Entwicklung einer neuen JavaScript-Engine war es, die Performance bei der Interpretation von JavaScript erheblich zu verbessern. Die Engine setzt dabei den ECMAScript-Standard ECMA-262 in der fünften Version um. Die V8-Engine selbst ist in C++ geschrieben, läuft auf verschiedenen Plattformen und ist unter der BSD-Lizenz als Open-Source-Software für jeden Entwickler zur eigenen Verwendung und Verbesserung verfügbar. So können Sie die Engine beispielsweise in jede beliebige C++-Anwendung integrieren.

Wie in JavaScript üblich, wird der Quellcode vor der Ausführung nicht kompiliert, sondern die Dateien mit dem Quellcode werden beim Start der Applikation direkt eingelesen. Durch den Start der Applikation wird ein neuer Node.js-Prozess gestartet. Hier erfolgt dann die erste Optimierung durch die V8-Engine. Der Quellcode wird nicht direkt interpretiert, sondern zuerst in Maschinencode übersetzt, der dann ausgeführt wird. Diese Technologie wird als Just-in-time-Kompilierung, kurz JIT, bezeichnet und dient zur Steigerung der Ausführungsgeschwindigkeit der JavaScript-Applikation. Auf Basis des kompilierten Maschinencodes wird dann die eigentliche Applikation ausgeführt. Die V8-Engine nimmt neben der Just-in-time-Kompilierung weitere Optimierungen vor. Unter anderem sind das eine verbesserte Garbage Collection und eine Verbesserung im Rahmen des Zugriffs auf Eigenschaften von Objekten. Bei allen Optimierungen, die die JavaScript-Engine vornimmt, sollten Sie beachten, dass der Quellcode beim Prozessstart eingelesen wird und so die

Änderungen an den Dateien keine Wirkung auf die laufende Applikation haben. Damit Ihre Änderungen wirksam werden, müssen Sie Ihre Applikation beenden und neu starten, damit die angepassten Quellcodedateien erneut eingelesen werden.

1.4.1 Das Speichermodell

Das Ziel der Entwicklung der V8-Engine war es, eine möglichst hohe Geschwindigkeit bei der Ausführung von JavaScript-Quellcode zu erreichen. Aus diesem Grund wurde auch das Speichermodell optimiert. In der V8-Engine kommen sogenannte Tagged Pointers zum Einsatz. Das sind Verweise im Speicher, die auf eine besondere Art als solche gekennzeichnet sind. Alle Objekte sind 4-Byte aligned, das bedeutet, dass 2 Bit zur Kennzeichnung von Zeigern zur Verfügung stehen. Ein Zeiger endet im Speichermodell der V8-Engine stets auf »01«, ein normaler Integerwert auf »0«. Durch diese Maßnahme können Integerwerte sehr schnell von Verweisen im Speicher unterschieden werden, was einen sehr großen Performancevorteil mit sich bringt. Die Objektrepräsentationen der V8-Engine im Speicher bestehen jeweils aus drei Datenworten. Das erste Datenwort besteht aus einem Verweis auf die Hidden Class des Objekts, über die Sie im Folgenden noch mehr erfahren werden. Das zweite Datenwort ist ein Zeiger auf die Attribute, also die Eigenschaften des Objekts. Das dritte Datenwort verweist schließlich auf die Elemente des Objekts. Das sind die Eigenschaften mit einem numerischen Schlüssel. Dieser Aufbau unterstützt die JavaScript-Engine in ihrer Arbeit und ist dahingehend optimiert, dass ein sehr schneller Zugriff auf die Elemente im Speicher erfolgen kann und hier wenig Wartezeiten durch das Suchen von Objekten entstehen.

1.4.2 Zugriff auf Eigenschaften

Wie Sie wahrscheinlich wissen, kennt JavaScript keine Klassen, das Objektmodell von JavaScript basiert auf Prototypen. In klassenbasierten Sprachen wie Java oder PHP stellen Klassen den Bauplan von Objekten dar. Diese Klassen können zur Laufzeit nicht verändert werden. Die Prototypen in JavaScript hingegen sind dynamisch. Das bedeutet, dass Eigenschaften und Methoden zur Laufzeit hinzugefügt und entfernt werden können. Wie bei allen anderen Sprachen, die das objektorientierte Programmierparadigma umsetzen, werden Objekte durch ihre Eigenschaften und Methoden repräsentiert, wobei die Eigenschaften den Status eines Objekts repräsentieren und die Methoden zur Interaktion mit dem Objekt verwendet werden. In einer Applikation greifen Sie in der Regel sehr häufig auf die Eigenschaften der verschiedenen Objekte zu. Hinzu kommt, dass in JavaScript Methoden ebenfalls Eigenschaften von Objekten sind, die mit einer Funktion hinterlegt sind. In JavaScript arbeiten Sie fast ausschließlich mit Eigenschaften und Methoden. Daher muss der Zugriff auf diese sehr schnell erfolgen.

Prototypen in JavaScript

JavaScript unterscheidet sich von Sprachen wie C, Java oder PHP dadurch, dass es keinen klassenbasierten Ansatz verfolgt, sondern auf Prototypen setzt, wie die Sprache Self. In JavaScript besitzt normalerweise jedes Objekt eine Eigenschaft `prototype` und damit einen Prototyp. In JavaScript können Sie wie in anderen Sprachen auch Objekte erzeugen. Zu diesem Zweck nutzen Sie allerdings keine Klassen in Verbindung mit dem `new`-Operator. Stattdessen können Sie auf verschiedene Arten neue Objekte erzeugen. Unter anderem können Sie auch Konstruktor-Funktionen oder die Methode `Object.create` nutzen. Diese Methoden haben gemein, dass Sie ein Objekt erstellen und den Prototyp zuweisen. Der Prototyp ist ein Objekt, von dem ein anderes Objekt seine Eigenschaften erbt. Ein weiteres Merkmal von Prototypen ist, dass sie zur Laufzeit der Applikation modifiziert werden können und Sie so neue Eigenschaften und Methoden hinzufügen können. Durch die Verwendung von Prototypen können Sie in JavaScript eine Vererbungshierarchie aufbauen. In Abschnitt 4.2.2, »Utility«, erfahren Sie mehr dazu, wie Sie von Node.js bei der prototypenbasierten Vererbung unterstützt werden.

Im Normalfall geschieht der Zugriff auf Eigenschaften in einer JavaScript-Engine über ein Verzeichnis im Arbeitsspeicher. Greifen Sie also auf eine Eigenschaft zu, wird in diesem Verzeichnis nach der Speicherstelle der jeweiligen Eigenschaft gesucht, danach kann dann auf den Wert zugegriffen werden. Stellen Sie sich nun eine große Applikation vor, die auf der Clientseite ihre Geschäftslogik in JavaScript abbildet und in der parallel eine Vielzahl von Objekten im Speicher gehalten werden, die ständig miteinander kommunizieren, wird diese Art des Zugriffs auf Eigenschaften schnell zu einem Problem. Die Entwickler der V8-Engine haben diese Schwachstelle erkannt und mit den sogenannten Hidden Classes eine Lösung dafür entwickelt. Das eigentliche Problem bei JavaScript besteht darin, dass der Aufbau von Objekten erst zur Laufzeit bekannt ist und nicht schon während des Kompiliervorgangs, da dieser bei JavaScript nicht existiert. Erschwerend kommt hinzu, dass es im Aufbau von Objekten nicht nur einen Prototyp gibt, sondern diese in einer Kette vorliegen können. In klassischen Sprachen verändert sich die Objektstruktur zur Laufzeit der Applikation nicht, die Eigenschaften von Objekten liegen immer an der gleichen Stelle, was den Zugriff erheblich beschleunigt.

Eine Hidden Class ist nichts weiter als eine Beschreibung, wo die einzelnen Eigenschaften eines Objekts im Speicher zu finden sind. Zu diesem Zweck wird jedem Objekt eine Hidden Class zugewiesen. Diese enthält den Offset zu der Speicherstelle innerhalb des Objekts, an der die jeweilige Eigenschaft gespeichert ist. Sobald Sie auf eine Eigenschaft eines Objekts zugreifen, wird eine Hidden Class für diese Eigenschaft erstellt und bei jedem weiteren Zugriff wiederverwendet. Für ein Objekt gibt es also potenziell für jede Eigenschaft eine separate Hidden Class.

In Listing 1.1 sehen Sie ein Beispiel, das die Funktionsweise von Hidden Classes verdeutlicht.

```
function Person(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
}
var johnDoe = new Person("John", "Doe");
```

Listing 1.1 Funktionsweise von Hidden Classes

Im Beispiel erstellen Sie eine neue Konstruktor-Funktion für die Gruppe der Person-Objekte. Dieser Konstruktor besitzt zwei Parameter, den Vor- und Nachnamen der Person. Diese beiden Werte sollen in den Eigenschaften `firstname` beziehungsweise `lastname` des Objekts gespeichert werden. Wird ein neues Objekt mit diesem Konstruktor mithilfe des New Operators erzeugt, wird zuerst eine initiale Hidden Class, Class 0, erstellt. Diese enthält noch keinerlei Zeiger auf Eigenschaften. Wird die erste Zuweisung, also das Setzen des Vornamens, durchgeführt, wird eine neue Hidden Class, Class 1, auf Basis von Class 0 erstellt. Diese enthält nun einen Verweis zur Speicherstelle der Eigenschaft `firstname`, und zwar relativ zum Beginn des Namensraums des Objekts. Außerdem wird in Class 0 eine sogenannte Class Transition hinzugefügt, die aussagt, dass Class 1 statt Class 0 verwendet werden soll, falls die Eigenschaft `firstname` hinzugefügt wird. Der gleiche Vorgang findet statt, wenn die zweite Zuweisung für den Nachnamen ausgeführt wird. Es wird eine weitere Hidden Class, Class 2, auf Basis von Class 1 erzeugt, die dann sowohl den Offset für die Eigenschaft `firstname` als auch für `lastname` enthält, und eine Transition mit dem Hinweis eingefügt, dass Class 2 verwendet werden soll, wenn die Eigenschaft `lastname` verwendet wird. Werden Eigenschaften abseits des Konstruktors hinzugefügt und erfolgt dies in unterschiedlicher Reihenfolge, werden jeweils neue Hidden Classes erzeugt.

Beim initialen Zugriff auf Eigenschaften eines Objekts entsteht durch die Verwendung von Hidden Classes noch kein Geschwindigkeitsvorteil. Alle späteren Zugriffe auf die Eigenschaft des Objekts geschehen dann allerdings um ein Vielfaches schneller, da die Engine direkt die Hidden Class des Objekts verwenden kann und diese den Hinweis auf die Speicherstelle der Eigenschaft enthält.

1.4.3 Maschinencodgenerierung

Wie Sie bereits wissen, interpretiert die V8-Engine den Quellcode der JavaScript-Applikation nicht direkt, sondern führt eine Just-in-time-Kompilierung in nativen Maschinencode durch, um die Ausführungsgeschwindigkeit zu steigern. Während dieser Kompilierung werden keinerlei Optimierungen am Quellcode durchgeführt.

Der vom Entwickler verfasste Quellcode wird also 1:1 gewandelt. Die V8-Engine besitzt neben diesem Just-in-time-Compiler noch einen weiteren Compiler, der in der Lage ist, den Maschinencode zu optimieren. Zur Entscheidung, welche Codefragmente zu optimieren sind, führt die Engine eine interne Statistik über die Anzahl der Funktionsaufrufe und wie lange die jeweilige Funktion ausgeführt wird. Aufgrund dieser Daten wird die Entscheidung getroffen, ob der Maschinencode einer Funktion optimiert werden muss oder nicht.

Nun stellen Sie sich bestimmt die Frage, warum denn nicht der gesamte Quellcode der Applikation mit dem zweiten, viel besseren Compiler kompiliert wird. Das hat einen ganz einfachen Grund: Der Compiler, der keine Optimierungen vornimmt, ist wesentlich schneller. Da die Kompilierung des Quellcodes just in time stattfindet, ist dieser Vorgang sehr zeitkritisch, weil sich eventuelle Wartezeiten durch einen zu lange dauernden Kompilierungsvorgang direkt auf den Nutzer auswirken können. Aus diesem Grund werden nur Codestellen optimiert, die diesen Mehraufwand rechtfertigen. Diese Maschinencodoptimierung wirkt sich vor allem positiv auf größere und länger laufende Applikationen aus und auf solche, bei denen Funktionen öfter als nur einmal aufgerufen werden.

Eine weitere Optimierung, die die V8-Engine vornimmt, hat mit den bereits beschriebenen Hidden Classes und dem internen Caching zu tun. Nachdem die Applikation gestartet und der Maschinencode generiert ist, sucht beziehungsweise erstellt die V8-Engine bei jedem Zugriff auf eine Eigenschaft die zugehörige Hidden Class. Als weitere Optimierung geht die Engine davon aus, dass in Zukunft die Objekte, die an dieser Stelle verwendet werden, die gleiche Hidden Class aufweisen und modifiziert den Maschinencode entsprechend. Wird die Codestelle beim nächsten Mal durchlaufen, kann direkt auf die Eigenschaft zugegriffen werden, und es muss nicht erst nach der zugehörigen Hidden Class gesucht werden. Falls das verwendete Objekt nicht die gleiche Hidden Class aufweist, stellt die Engine dies fest, entfernt den zuvor generierten Maschinencode und ersetzt ihn durch die korrigierte Version. Diese Vorgehensweise weist ein entscheidendes Problem auf: Stellen Sie sich vor, Sie haben eine Codestelle, an der im Wechsel immer zwei verschiedene Objekte mit unterschiedlichen Hidden Classes verwendet werden. In diesem Fall würde die Optimierung mit der Vorhersage der Hidden Class bei der nächsten Ausführung niemals greifen. Für diesen Fall kommen verschiedene Codefragmente zum Einsatz, anhand derer der Speicherort einer Eigenschaft zwar nicht so schnell wie mit nur einer Hidden Class gefunden werden kann, allerdings ist der Code in diesem Fall um ein Vielfaches schneller als ohne die Optimierung, da hier meist aus einem sehr kleinen Satz von Hidden Classes ausgewählt werden kann. Mit der Generierung von Maschinencode und den Hidden Classes in Kombination mit den Caching-Mechanismen werden Möglichkeiten geschaffen, wie man sie aus klassenbasierten Sprachen kennt.

1.4.4 Garbage Collection

Die bisher beschriebenen Optimierungen wirken sich hauptsächlich auf die Geschwindigkeit einer Applikation aus. Ein weiteres, sehr wichtiges Feature ist der Garbage Collector der V8-Engine. Garbage Collection bezeichnet den Vorgang des Aufräumens des Speicherbereichs der Applikation im Arbeitsspeicher. Dabei werden nicht mehr verwendete Elemente aus dem Speicher entfernt, damit der frei werdende Platz der Applikation wieder zur Verfügung steht.

Sollten Sie sich jetzt die Frage stellen, wozu man in JavaScript einen Garbage Collector benötigt, lässt sich dies ganz einfach beantworten. Ursprünglich war JavaScript für kleine Aufgaben auf Webseiten gedacht. Diese Webseiten und somit auch das JavaScript auf dieser Seite hatten eine recht kurze Lebensspanne, bis die Seite neu geladen und damit der Speicher, der die JavaScript-Objekte enthält, komplett geleert wurde. Je mehr JavaScript auf einer Seite ausgeführt wird und je komplexer die zu erledigenden Aufgaben werden, desto größer wird auch die Gefahr, dass der Speicher mit nicht mehr benötigten Objekten gefüllt wird. Gehen Sie nun von einer Applikation in Node.js aus, die mehrere Tage, Wochen oder gar Monate ohne Neustart des Prozesses laufen muss, wird die Problematik klar. Der Garbage Collector der V8-Engine verfügt über eine Reihe von Features, die es ihm ermöglichen, seine Aufgaben sehr schnell und effizient auszuführen. Grundsätzlich hält die Engine bei einem Lauf des Garbage Collectors die Ausführung der Applikation komplett an und setzt sie fort, sobald der Lauf beendet ist. Diese Pausen der Applikation bewegen sich im einstelligen Millisekundenbereich, sodass der Nutzer im Normalfall durch den Garbage Collector keine negativen Auswirkungen zu spüren bekommt. Um die Unterbrechung durch den Garbage Collector möglichst kurz zu halten, wird nicht der komplette Speicher aufgeräumt, sondern stets nur Teile davon. Außerdem weiß die V8-Engine zu jeder Zeit, wo im Speicher sich welche Objekte und Zeiger befinden.

Die V8-Engine teilt den ihr zur Verfügung stehenden Arbeitsspeicher in zwei Bereiche auf, einen zur Speicherung von Objekten und einen anderen Bereich, in dem die Informationen über die Hidden Classes und den ausführbaren Maschinencode vorgehalten werden. Der Vorgang der Garbage Collection ist relativ einfach. Wird eine Applikation ausgeführt, werden Objekte und Zeiger im kurzlebigen Bereich des Arbeitsspeichers der V8-Engine erzeugt. Ist dieser Speicherbereich voll, wird er bereinigt. Dabei werden nicht mehr verwendete Objekte gelöscht und Objekte, die weiterhin benötigt werden, in den langlebigen Bereich verschoben. Bei dieser Verschiebung wird zum einen das Objekt selbst verschoben, zum anderen werden die Zeiger auf die Speicherstelle des Objekts korrigiert. Durch die Aufteilung der Speicherbereiche werden verschiedene Arten der Garbage Collection erforderlich. Die schnellste Variante besteht aus dem sogenannten Scavenge Collector. Dieser ist sehr schnell und effizient und beschäftigt sich lediglich mit dem kurzlebigen Bereich. Für den langlebigen Speicherbereich existieren zwei verschiedene Garbage-Collection-Algorithmen, die

beide auf Mark-and-Sweep basieren. Dabei wird der gesamte Speicher durchsucht, und nicht mehr benötigte Elemente werden markiert und später gelöscht. Das eigentliche Problem dieses Algorithmus besteht darin, dass Lücken im Speicher entstehen, was über längere Laufzeit einer Applikation zu Problemen führt. Aus diesem Grund existiert ein zweiter Algorithmus, der ebenfalls die Elemente des Speichers nach solchen durchsucht, die nicht mehr benötigt werden, und diese markiert und löscht. Der wichtigste Unterschied zwischen beiden ist, dass der zweite Algorithmus den Speicher defragmentiert, also die verbleibenden Objekte im Speicher so umordnet, dass der Speicher danach möglichst wenige Lücken aufweist. Diese Defragmentierung kann nur stattfinden, weil V8 sämtliche Objekte und Pointer kennt. Der Prozess der Garbage Collection hat bei allen Vorteilen auch einen Nachteil: Er kostet Zeit. Am schnellsten läuft die Scavenge Collection mit etwa 2 Millisekunden. Danach folgt der Mark-and-Sweep ohne Optimierungen mit 50 Millisekunden und schließlich der Mark-and-Sweep mit Defragmentierung mit durchschnittlich 100 Millisekunden. In den nächsten Abschnitten erfahren Sie mehr über die Elemente, die neben der V8-Engine in der Node.js-Plattform eingesetzt werden.

1.5 Bibliotheken um die Engine

Die JavaScript-Engine allein macht noch keine Plattform aus. Damit Node.js alle Anforderungen wie beispielsweise die Behandlung von Events, Ein- und Ausgabe oder Unterstützungsfunktionen wie DNS-Auflösung oder Verschlüsselung behandeln kann, sind weitere Funktionalitäten erforderlich. Diese werden mithilfe zusätzlicher Bibliotheken umgesetzt. Für viele Aufgaben, mit denen sich eine Plattform wie Node.js konfrontiert sieht, existieren bereits fertige und etablierte Lösungsansätze. Also entschied sich Ryan Dahl dazu, die Node.js-Plattform auf einer Reihe von externen Bibliotheken aufzubauen und die Lücken, die seiner Meinung nach von keiner vorhandenen Lösung ausreichend abgedeckt werden, mit eigenen Implementierungen zu füllen. Der Vorteil dieser Strategie besteht darin, dass Sie die Lösungen für Standardprobleme nicht neu erfinden müssen, sondern auf erprobte Bibliotheken zurückgreifen können. Ein prominenter Vertreter, der ebenfalls auf diese Strategie setzt, ist das Betriebssystem UNIX. Hier gilt auch für Entwickler: Konzentrieren Sie sich nur auf das eigentliche Problem, lösen Sie es möglichst gut, und nutzen Sie für alles andere bereits existierende Bibliotheken. Bei den meisten Kommandozeilenprogrammen im UNIX-Bereich wird diese Philosophie umgesetzt. Hat sich eine Lösung bewährt, wird diese auch in anderen Anwendungen für ähnliche Probleme eingesetzt. Das bringt wiederum den Vorteil, dass Verbesserungen im Algorithmus nur an einer zentralen Stelle durchgeführt werden müssen. Gleiches gilt für Fehlerbehebungen. Tritt ein Fehler in der DNS-Auflösung auf, wird dieser einmal behoben, und die Lösung wirkt an allen Stellen, an denen die Bibliothek eingesetzt wird. Das

führt gleich auch noch zur Schattenseite der Medaille. Die Bibliotheken, auf denen die Plattform aufbaut, müssen vorhanden sein. Node.js löst dieses Problem, indem es lediglich auf einen kleinen Satz von Bibliotheken aufbaut, die vom Betriebssystem zur Verfügung gestellt werden müssen. Diese Abhängigkeiten bestehen allerdings eher aus grundlegenden Funktionen wie beispielsweise der GCC Runtime Library oder der Standard C Bibliothek. Die übrigen Abhängigkeiten wie beispielsweise »zlib« oder »http_parser« werden im Quellcode mit ausgeliefert.

1.5.1 Eventloop

Clientseitiges JavaScript weist viele Elemente einer eventgetriebenen Architektur auf. Die meisten Interaktionen des Nutzers verursachen Events, auf die mit entsprechenden Funktionsaufrufen reagiert wird. Durch den Einsatz verschiedener Features wie First-Class-Funktionen und anonymen Funktionen in JavaScript können Sie ganze Applikationen auf Basis einer eventgetriebenen Architektur umsetzen. Eventgetrieben bedeutet, dass Objekte nicht direkt über Funktionsaufrufe miteinander kommunizieren, sondern für diese Kommunikation Events zum Einsatz kommen. Die eventgetriebene Programmierung dient also in erster Linie der Steuerung des Programmablaufs. Im Gegensatz zum klassischen Ansatz, bei dem der Quellcode linear durchlaufen wird, werden hier Funktionen ausgeführt, wenn bestimmte Ereignisse auftreten. Ein kleines Beispiel in Listing 1.2 verdeutlicht Ihnen diesen Ansatz.

```
myObj.on('myEvent', function (data) {
    console.log(data);
});
myObj.emit('myEvent', 'Hello World');
```

Listing 1.2 Eventgetriebene Entwicklung in Node.js

Mit der `on`-Methode eines Objekts, das Sie von `events.EventEmitter` ableiten, können Sie definieren, mit welcher Funktion Sie auf das jeweilige Event reagieren möchten. Hierbei handelt es sich um ein sogenanntes Publish-Subscribe Pattern. Objekte können sich so bei einem Event-Emitter registrieren und werden dann benachrichtigt, wenn das Ereignis eintritt. Das erste Argument der `on`-Methode ist der Name des Events als Zeichenkette, auf das reagiert werden soll. Das zweite Argument besteht aus einer Callback-Funktion, die ausgeführt wird, sobald das Ereignis eintritt. Der Funktionsaufruf der `on`-Methode bewirkt also bei der ersten Ausführung nichts weiter als die Registrierung der Callback-Funktion. Im späteren Verlauf des Scripts wird auf `myObj` die `emit`-Methode aufgerufen. Diese sorgt dafür, dass sämtliche durch die `on`-Methode registrierten Callback-Funktionen ausgeführt werden.

Was in diesem Beispiel mit einem selbst erstellten Objekt funktioniert, verwendet Node.js, um eine Vielzahl asynchroner Aufgaben zu erledigen. Die Callback-Funktio-

nen werden allerdings nicht parallel ausgeführt, sondern sequenziell. Durch den Single-Threaded-Ansatz von Node.js entsteht das Problem, dass nur eine Operation zu einem Zeitpunkt ausgeführt werden kann. Vor allem zeitintensive Lese- oder Schreiboperationen würden dafür sorgen, dass die gesamte Ausführung der Anwendung blockiert würde. Aus diesem Grund werden sämtliche Lese- und Schreiboperationen mithilfe des Eventloops ausgelagert. So kann der verfügbare Thread durch den Code der Applikation ausgenutzt werden. Sobald eine Anfrage an eine externe Ressource im Quellcode gestellt wird, wird diese an den Eventloop weitergegeben. Für die Anfrage wird ein Callback registriert, der die Anfrage an das Betriebssystem weiterleitet, Node.js erhält daraufhin wieder die Kontrolle und kann mit der Ausführung der Applikation fortfahren. Sobald die externe Operation beendet ist, wird das Ergebnis an den Eventloop zurückübermittelt. Es tritt ein Event auf, und der Eventloop sorgt dafür, dass die zugehörigen Callback-Funktionen ausgeführt werden.

Der ursprüngliche Eventloop, der bei Node.js zum Einsatz kommt, basiert auf libev, eine Bibliothek, die in C geschrieben ist und für eine hohe Performance und einen großen Umfang an Features steht. libev baut auf den Ansätzen von libevent auf, verfügt allerdings über eine höhere Leistungsfähigkeit wie verschiedene Benchmarks belegen. Auch eine verbesserte Version von libevent, libevent2, reicht nicht an die Performance von libev heran. Aus Kompatibilitätsgründen wurde der Eventloop allerdings abstrahiert und damit eine bessere Portierbarkeit auf andere Plattformen erreicht.

1.5.2 Eingabe und Ausgabe

Der Eventloop allein in Kombination mit der V8-Engine erlaubt zwar die Ausführung von JavaScript, es existiert hier allerdings noch keine Möglichkeit, mit dem Betriebssystem direkt in Form von Lese- oder Schreiboperationen auf das Dateisystem zu interagieren. Bei der Implementierung serverseitiger Anwendungen spielen Zugriffe auf das Dateisystem eine herausragende Rolle, so wird beispielsweise die Konfiguration einer Anwendung häufig in eine separate Konfigurationsdatei ausgelagert. Diese Konfiguration muss von der Applikation vom Dateisystem eingelesen werden. Aber auch die Verwendung von Templates, die dynamisch mit Werten befüllt und dann zum Client geschickt werden, liegen meist als separate Dateien vor. Nicht nur das Auslesen, sondern auch das Schreiben von Informationen in Dateien ist häufig eine Anforderung, die an eine serverseitige JavaScript-Applikation gestellt wird. Die Protokollierung innerhalb einer Applikation ist ebenfalls ein häufiges Einsatzgebiet von schreibenden Zugriffen auf das Dateisystem. Hier werden verschiedene Arten von Ereignissen innerhalb der Applikation in eine Logdatei protokolliert. Je nachdem, wo die Anwendung ausgeführt wird, werden nur schwerwiegende Fehler, Warnungen oder auch Laufzeitinformationen geschrieben. Auch beim Persistieren von Informationen kommen schreibende Zugriffe zum Einsatz. Zur Laufzeit einer

Anwendung werden, meist durch die Interaktion von Nutzern und verschiedenen Berechnungen, Informationen generiert, die zur späteren Weiterverwendung festgehalten werden müssen.

In Node.js kommt für diese Aufgaben die C-Bibliothek `libeio` zum Einsatz. Sie sorgt dafür, dass die Schreib- und Leseoperationen asynchron stattfinden können und arbeitet so sehr eng mit dem Eventloop zusammen. Die Features von `libeio` beschränken sich jedoch nicht nur auf den schreibenden und lesenden Zugriff auf das Dateisystem, sondern bieten erheblich mehr Möglichkeiten, mit dem Dateisystem zu interagieren. Diese Optionen reichen vom Auslesen von Dateiinformatoren wie Größe, Erstellungsdatum oder Zugriffsdatum über die Verwaltung von Verzeichnissen, also Erstellen oder Entfernen, bis hin zur Modifizierung von Zugriffsrechten. Auch für diese Bibliothek gilt, wie auch schon beim Eventloop, dass sie im Laufe der Entwicklung durch eine Abstraktionsschicht von der eigentlichen Applikation getrennt wurde.

Für den Zugriff auf das Dateisystem stellt Node.js ein eigenes Modul zur Verfügung, das `Filesystem`-Modul. Über dieses lassen sich die Schnittstellen von `libeio` ansprechen, es stellt damit einen sehr leichtgewichtigen Wrapper um `libeio` dar.

1.5.3 libuv

Die beiden Bibliotheken, die Sie bislang kennengelernt haben, gelten für Linux. Node.js sollte allerdings eine vom Betriebssystem unabhängige Plattform werden. Aus diesem Grund wurde in der Version 0.6 von Node.js die Bibliothek `libuv` eingeführt. Sie dient primär zur Abstraktion von Unterschieden zwischen verschiedenen Betriebssystemen. Der Einsatz von `libuv` macht es also möglich, dass Node.js auch auf Windowssystemen lauffähig ist. Der Aufbau ohne `libuv`, wie er bis zur Version 0.6 für Node.js gültig war, sieht folgendermaßen aus: Den Kern bildet die V8-Engine, dieser wird durch `libev` und `libeio` um den Eventloop und asynchronen Dateisystemzugriff ergänzt. Mit `libuv` sind diese beiden Bibliotheken nicht mehr direkt in die Plattform eingebunden, sondern werden abstrahiert.

Damit Node.js auch auf Windows funktionieren kann, ist es erforderlich, die Kernkomponenten für Windows-Plattformen zur Verfügung zu stellen. Die V8-Engine stellt hier kein Problem dar, sie funktioniert im Chrome-Browser bereits seit mehreren Jahren ohne Probleme unter Windows. Schwieriger wird die Situation beim Eventloop und bei den asynchronen Dateisystemoperationen. Einige Komponenten von `libev` müssten beim Einsatz unter Windows umgeschrieben werden. Außerdem basiert `libev` auf nativen Implementierungen des Betriebssystems der `select`-Funktion, unter Windows steht allerdings mit IOCP eine für das Betriebssystem optimierte Variante zur Verfügung. Um nicht verschiedene Versionen von Node.js für die unterschiedlichen Betriebssysteme erstellen zu müssen, entschieden sich die Ent-

wickler, mit libuv eine Abstraktionsschicht einzufügen, die es erlaubt, für Linux-Systeme libev und für Windows IOCP zu verwenden. Mit libuv wurden einige Kernkonzepte von Node.js angepasst. Es wird beispielsweise nicht mehr von Events, sondern von Operationen gesprochen. Eine Operation wird an die libuv-Komponente weitergegeben, innerhalb von libuv wird die Operation an die darunterliegende Infrastruktur, also libev beziehungsweise IOCP, weitergereicht. So bleibt die Schnittstelle von Node.js unverändert, unabhängig davon, welches Betriebssystem verwendet wird.

libuv ist dafür zuständig, alle asynchronen I/O-Operationen zu verwalten. Das bedeutet, dass sämtliche Zugriffe auf das Dateisystem, egal, ob lesend oder schreibend, über die Schnittstellen von libuv durchgeführt werden. Zu diesem Zweck stellt libuv die `uv_fs_`-Funktionen zur Verfügung. Aber auch Timer, also zeitabhängige Aufrufe sowie asynchrone TCP- und UDP-Verbindungen, laufen über libuv. Neben diesen grundlegenden Funktionalitäten verwaltet libuv auch komplexe Features wie das Erstellen, das Spawnen, von Kindprozessen und das Thread Pool Scheduling, eine Abstraktion, die es erlaubt, Aufgaben in separaten Threads zu erledigen und Callbacks daran zu binden. Der Einsatz einer Abstraktionsschicht wie libuv ist ein wichtiger Baustein für die weitere Verbreitung von Node.js und macht die Plattform ein Stück weniger abhängig vom System.

1.5.4 DNS

Die Wurzeln von Node.js liegen im Internet, wie seine Entstehungsgeschichte zeigt. Bewegen Sie sich im Internet, stoßen Sie recht schnell auf die Problematik der Namensauflösung. Eigentlich werden sämtliche Server im Internet über ihre IP-Adresse angesprochen. In der Version 4 des Internet Protocols ist die Adresse eine 32-Bit-Zahl, die in vier Blöcken mit je 8 Bits dargestellt wird. In der sechsten Version des Protokolls haben die Adressen eine Größe von 128 Bits und werden in acht Blöcke mit Hexadezimalzahlen aufgeteilt. Mit diesen kryptischen Adressen will man in den seltensten Fällen direkt arbeiten, vor allem, wenn eine dynamische Vergabe über DHCP hinzukommt. Die Lösung hierfür besteht im Domain Name System, kurz DNS. Das DNS ist ein Dienst zur Namensauflösung im Netz. Es sorgt dafür, dass Domainnamen in IP-Adressen gewandelt werden. Außerdem gibt es die Möglichkeit der Reverse-Auflösung, bei der eine IP-Adresse in einen Domainnamen übersetzt wird. Falls Sie in Ihrer Node.js-Applikation einen Webservice anbinden oder eine Webseite auslesen möchten, kommt auch hier das DNS zum Einsatz.

Intern übernimmt nicht Node.js selbst die Namensauflösung, sondern übergibt die jeweiligen Anfragen an die C-Ares-Bibliothek. Dies gilt für sämtliche Methoden des `dns`-Moduls bis auf `dns.lookup`, das auf die betriebssystemeigene `getaddrinfo`-Funktion setzt. Diese Ausnahme ist darin begründet, dass `getaddrinfo` konstanter in sei-

nen Antworten ist als die C-Ares-Bibliothek, die ihrerseits um einiges performanter ist als `getaddrinfo`.

1.5.5 Crypto

Die Crypto-Komponente der Node.js-Plattform stellt Ihnen für die Entwicklung verschiedene Möglichkeiten der Verschlüsselung zur Verfügung. Diese Komponente basiert auf OpenSSL. Das bedeutet, dass diese Software auf Ihrem System installiert sein muss, um Daten verschlüsseln zu können. Mit dem `crypto`-Modul sind Sie in der Lage, sowohl Daten mit verschiedenen Algorithmen zu verschlüsseln als auch digitale Signaturen innerhalb Ihrer Applikation zu erstellen. Das gesamte System basiert auf privaten und öffentlichen Schlüsseln. Der private Schlüssel ist, wie der Name andeutet, nur für Sie und Ihre Applikation gedacht. Der öffentliche Schlüssel steht Ihren Kommunikationspartnern zur Verfügung. Sollen nun Inhalte verschlüsselt werden, geschieht dies mit dem öffentlichen Schlüssel. Die Daten können dann nur noch mit Ihrem privaten Schlüssel entschlüsselt werden. Ähnliches gilt für die digitale Signatur von Daten. Hier wird Ihr privater Schlüssel verwendet, um eine derartige Signatur zu erzeugen. Der Empfänger einer Nachricht kann dann mit der Signatur und Ihrem öffentlichen Schlüssel feststellen, ob die Nachricht von Ihnen stammt und unverändert ist.

1.5.6 Zlib

Bei der Erstellung von Webapplikationen müssen Sie als Entwickler stets an die Ressourcen Ihrer Benutzer und Ihrer eigenen Serverumgebung denken. So kann beispielsweise die zur Verfügung stehende Bandbreite oder der freie Speicher für Daten eine Limitation bedeuten. Für diesen Fall existiert innerhalb der Node.js-Plattform die Zlib-Komponente. Mit ihrer Hilfe lassen sich Daten komprimieren und wieder dekomprimieren, wenn Sie sie verarbeiten möchten. Zur Datenkompression stehen Ihnen die beiden Algorithmen Deflate und Gzip zur Verfügung. Die Daten, die als Eingabe für die Algorithmen dienen, werden von Node.js als Streams behandelt.

Node.js implementiert die Komprimierungsalgorithmen nicht selbst, sondern setzt stattdessen auf die etablierte Zlib und reicht die Anfragen jeweils weiter. Das `zlib`-Modul von Node.js stellt lediglich einen leichtgewichtigen Wrapper zur Zlib dar und sorgt dafür, dass die Ein- und Ausgabestreams korrekt behandelt werden.

1.5.7 HTTP-Parser

Als Plattform für Webapplikationen muss Node.js nicht nur mit Streams, komprimierten Daten und Verschlüsselung, sondern auch mit dem HTTP-Protokoll umgehen können. Da das Parsen des HTTP-Protokolls eine recht aufwendige Prozedur

ist, wurde der HTTP-Parser, der diese Aufgabe übernimmt, in ein eigenes Projekt ausgelagert und wird nun von der Node.js-Plattform eingebunden. Wie die übrigen externen Bibliotheken ist auch der HTTP-Parser in C geschrieben und dient als performantes Werkzeug, um sowohl Anfragen als auch Antworten des HTTP-Protokolls auszulesen. Das bedeutet für Sie als Entwickler konkret, dass Sie mit dem HTTP-Parser beispielsweise die verschiedenen Informationen des HTTP-Headers oder den Text der Nachricht selbst auslesen können.

Das primäre Entwicklungsziel von Node.js ist es, eine performante Plattform für Webapplikationen zur Verfügung zu stellen. Um diese Anforderung zu erfüllen, baut Node.js auf einem modularen Ansatz auf. Dieser erlaubt die Einbindung externer Bibliotheken wie beispielsweise der bereits beschriebenen `libuv` oder dem HTTP-Parser. Der modulare Ansatz wird durch die internen Module der Node.js-Plattform weitergeführt und reicht bis zu den Erweiterungen, die Sie für Ihre eigene Applikation erstellen. Im Laufe dieses Buches werden Sie die verschiedenen Möglichkeiten und Technologien kennenlernen, die Ihnen die Node.js-Plattform zur Entwicklung eigener Applikationen zur Verfügung stellt. Den Anfang macht eine Einführung in das Modulsystem von Node.js.

1.6 Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie Node.js grundsätzlich aufgebaut ist und welche Vorteile Ihnen aus diesem Aufbau entstehen. Außerdem haben Sie einige Fakten über die Entwicklungsgeschichte der Plattform kennengelernt. Im nächsten Kapitel lernen Sie, wie Sie Node.js auf Ihrem System installieren und wie Sie die Funktionsfähigkeit dieser Installation testen können.

Kapitel 2

Installation

*Der Mensch erweitert den Weg und nicht der Weg
erweitert den Menschen.
– Konfuzius*

Die Komponenten von Node.js wurden in C++ entwickelt. Das bedeutet, dass die gesamte Plattform grundsätzlich relativ unabhängig vom Betriebssystem ist, auf dem sie ausgeführt wird. Diese Plattformunabhängigkeit sorgt dafür, dass Sie Node.js sowohl auf UNIX-basierten Systemen wie Linux oder Mac OS X als auch auf Windows installieren können. Ursprünglich war Node.js entwickelt worden, um auf UNIX-basierten Systemen installiert und verwendet zu werden. Diese Herkunft können Sie immer noch feststellen, da einige Funktionalitäten nicht unter Windows zur Verfügung stehen. So sind beispielsweise verschiedene Funktionen des `process`-Moduls nicht verfügbar. Einen weiteren Unterschied können Sie im `fs`-Modul, das die Dateisystemfunktionalität für Node.js zur Verfügung stellt, beobachten. Die aktuell noch instabile `watch`-Funktionalität baut auf den vom Betriebssystem zur Verfügung gestellten Schnittstellen auf und verhält sich aus diesem Grund auf den verschiedenen Systemen etwas unterschiedlich. Zusammenfassend lässt sich feststellen, dass Node.js auf allen drei genannten Plattformen stabil und zuverlässig funktioniert. Bei der Verwendung der verschiedenen Features sollten Sie als Entwickler allerdings stets die API-Dokumentation unter <http://nodejs.org/api/> konsultieren, um sicherzustellen, ob die jeweilige Funktionalität auch auf Ihrem Zielsystem in der erwarteten Art und Weise zur Verfügung steht.

Bei der Versionierung folgt Node.js dem klassischen Ansatz. Das bedeutet, dass eine Versionsnummer von Node.js aus drei Teilen besteht: dem Major Release, dem Minor Release und dem Patch Level. Der Name eines Node.js-Pakets, wie es auf der Webseite von Node.js verfügbar ist, setzt sich wie folgt zusammen: `node-v<major>.<minor>.<patch>`, danach folgen meist noch Informationen zum Betriebssystem und der Zielarchitektur. Das führt beispielsweise für ein Linux-System mit einem 64-Bit-Prozessor zu folgendem Namen: `node-v0.10.9-linux-x64.tar.gz`. Eine Erhöhung des Major-Release bedeutet, dass sich in unserem Fall bei Node.js gravierende Änderungen ergeben haben und nicht sichergestellt werden kann, dass bestehende Applikationen ohne umfangreiche Anpassungen lauffähig sind. Die Angabe des Minor Release hat zwei Bedeutungen: Zum einen sagt eine Erhöhung dieser Versionsnummer aus, dass es bei Node.js zusätzliche Erweiterungen und Verbes-

serungen gibt, zum anderen bedeuten gerade Versionsnummern, dass es sich um die stabile Version von Node.js handelt. Ein ungerades Minor Release bedeutet, dass Sie den Entwicklerzweig von Node.js verwenden. Das Patch Level wird recht häufig erhöht, es sagt lediglich aus, dass die verwendete Version kleinste Aktualisierungen und Bugfixes enthält.

Nach den bisherigen, noch recht theoretischen Ausführungen beschäftigen sich die folgenden Abschnitte mit der Installation der Node.js Plattform. Am Ende dieses Kapitels sollten Sie eine funktionsfähige Installation von Node.js auf Ihrem System installiert haben, egal, ob Sie Linux, Mac OS X oder Windows einsetzen. Im Verlauf dieses Kapitels werden Sie schrittweise durch den Installationsprozess geführt. Dabei wird in erster Linie die Installation der Pakete für Ihr jeweiliges Betriebssystem behandelt. Sie erfahren außerdem, wie Sie den Quellcode kompilieren und auf Ihrem System installieren können.

2.1 Installation von Paketen

Die einfachste Art und Weise Node.js auf Ihrem System zu installieren, ist, wenn Sie auf bestehende Pakete für Ihr System zurückgreifen. Der Vorteil hierbei liegt darin, dass Sie den Paketmanager Ihres Systems verwenden können, der Ihnen die meiste Arbeit während der Installation abnimmt. Sie müssen sich hier nicht um die Auflö- sung von Abhängigkeiten kümmern und können deshalb Node.js mit nur wenigen Zeilen auf der Kommandozeile beziehungsweise ein paar Klicks, je nachdem, welches Betriebssystem Sie verwenden, auf Ihrem System installieren. Node.js ist ein Open- Source-Projekt und aus diesem Grund auch kostenlos auf der Webseite [http:// nodejs.org/download/](http://nodejs.org/download/) verfügbar. Hier können Sie über die jeweiligen Links die Pakete für verschiedene Betriebssysteme und Rechnerarchitekturen auf Ihren Rech- ner herunterladen. Abbildung 2.1 zeigt die Tabelle der verschiedenen Versionen, die als Download verfügbar sind.

		
Windows Installer node-v0.8.14-x86.msi	Macintosh Installer node-v0.8.14.pkg	Source Code node-v0.8.14.tar.gz
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.8.14.tar.gz	

Abbildung 2.1 Downloads von Node.js

Die folgenden Abschnitte behandeln jeweils die Installation der Pakete für ein bestimmtes Betriebssystem. Normalerweise ist es ausreichend, wenn Sie den entsprechenden Abschnitt für Ihr System durchlesen und dann zu Kapitel 3, »Ein erstes Beispiel«, übergehen. Sollten Sie Node.js aus dem Quellcode kompilieren und installieren wollen, können Sie direkt mit Abschnitt 2.2, »Kompilieren und installieren«, fortfahren, in dem dieser Prozess im Detail beschrieben wird.

2.1.1 Linux

Als Erstes lesen Sie in den nun folgenden Abschnitten, wie Sie Node.js auf einem Linux-System installieren können. Hier stehen Ihnen zwei Varianten der Installation von Node.js zur Verfügung. Die erste besteht aus der Installation der Linux Binaries, die Sie von <http://nodejs.org> herunterladen können. Der Einsatz des Paketmanagers des Betriebssystems stellt die zweite Variante dar. In der ersten Variante der Installation unterscheiden sich die verschiedenen Distributionen nicht, da das Linux-Binaries-Paket bereits die ausführbaren Dateien enthält und das Paket lediglich entpackt und korrekt in das System eingebunden werden muss. Laut dem Filesystem Hierarchy Standard für UNIX-Systeme dient das `/opt`-Verzeichnis als Container für zusätzliche Software, also eignet sich dieses Verzeichnis als Speicherort für die Node.js Linux Binaries. Die Installation der Pakete ist in Listing 2.1 beschrieben.

```
$ cd /home/<username>/Downloads
$ wget http://nodejs.org/dist/v0.10.9/node-v0.10.9-linux-x64.tar.gz
--2012-11-03 01:21:34-- http://nodejs.org/dist/v0.10.9/node-v0.10.9-linux-
x64.tar.gz
Resolving nodejs.org (nodejs.org)... 165.225.133.150
...
$ cd /opt
$ sudo tar xvzf /home/<username>/Downloads/node-v0.10.9-linux-x64.tar.gz
[sudo] password for <username>:
node-v0.10.9-linux-x64/
...
$ sudo ln -s node-v0.10.9-linux-x64 nodejs
$
```

Listing 2.1 Installation der Node.js Linux Binaries

Listing 2.1 zeigt Ihnen, wie Sie ohne grafische Oberfläche das Paket mit den Node.js Linux Binaries herunterladen können. Dabei kommt das Kommandozeilenwerkzeug `wget` zum Einsatz. Es lädt das Paket vom Webserver <http://nodejs.org> herunter. Danach wird es im Verzeichnis `/opt` entpackt. Um Software in diesem Verzeichnis installieren zu können, benötigen Sie root-Berechtigung, was Sie in diesem Fall mit dem Kommando `sudo` erreichen. Im letzten Schritt wird ein symbolischer Link mit

dem Namen *nodejs* auf das Verzeichnis erstellt, damit ersparen Sie sich im weiteren Verlauf Schreibarbeit und vermeiden Fehler beim Schreiben des kompletten Verzeichnisnamens. Zu diesem Zeitpunkt können Sie Node.js bereits auf der Kommandozeile verwenden, wenn Sie die ausführbare *node*-Datei mit ihrer absoluten beziehungsweise relativen Pfadangabe ansprechen. Das bedeutet allerdings wiederum, dass Sie bei jedem Aufruf je nach Ablageort einen unverhältnismäßig langen Befehl eingeben müssen, bei dem sich auch leicht Fehler einschleichen. Das Ziel ist es nun, Node.js durch Eingabe des Kommandos *node* direkt aufzurufen, was Sie einfach erreichen können, indem Sie das *bin*-Verzeichnis innerhalb der Node.js-Installation in den Suchpfad des Systems aufnehmen. Listing 2.2 zeigt Ihnen die Befehlszeilen, die erforderlich sind, um den Suchpfad entsprechend zu erweitern.

```
PATH=$PATH:/opt/nodejs/bin  
export PATH
```

Listing 2.2 Erweiterung des Suchpfades unter Linux

Nachdem Sie die beiden Zeilen aus Listing 2.2 auf Ihrer Kommandozeile eingegeben haben, können Sie Node.js verwenden. Das einzige Problem besteht nun lediglich darin, dass Sie diese Befehle für jede Shell-Session erneut eingeben müssen, da die Erweiterung des Suchpfades nicht persistent ist. Abhilfe schafft die Aufnahme dieser Zeilen in eine der systemweiten Konfigurationsdateien. Da Sie in diesem Fall Anpassungen an einer Umgebungsvariablen vornehmen, empfiehlt sich hier die Datei */etc/profile*. Hier müssen lediglich die beiden Zeilen ans Ende der Datei angefügt werden, und Node.js ist bei jeder Anmeldung am System auf der Kommandozeile verfügbar.

Die zweite Variante der Installation von Node.js unter Linux-Systemen ist abhängig vom Paketmanager der jeweiligen Distribution. Im Folgenden sehen Sie, wie Sie mit nur wenigen Schritten Node.js auf einem Ubuntu-System installieren können.

```
sudo apt-get install nodejs
```

Listing 2.3 Installation von Node.js über den Ubuntu-Paketmanager

Durch den Befehl in Listing 2.3 wird die Version, die in den Standardrepositorys liegt, verwendet. Installieren Sie Node.js auf diese Weise, haben Sie den Vorteil, dass die Software automatisch aktualisiert wird, sobald eine neue Version im Repository verfügbar ist. Der Nachteil dieser Variante der Installation ist, dass die Versionen recht langsam aktualisiert werden und so in den seltensten Fällen die aktuelle Version zur Verfügung steht. Aber auch für die Nachteile dieser Art der Installation existiert eine Lösung. Hierfür müssen Sie lediglich ein zusätzliches Repository einbinden. Über dieses steht jeweils die aktuelle Version von Node.js als Paket zur Verfügung und kann automatisch aktualisiert werden, falls eine neue Version verfügbar ist. Listing 2.4 zeigt Ihnen, welche einzelnen Schritte Sie durchführen müssen, um das Repository einzubinden und Node.js davon zu installieren.

```

sudo apt-get install python-software-properties
sudo add-apt-repository ppa.chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs npm

```

Listing 2.4 Einbinden des Node.js Repository für Ubuntu

Wie bei den meisten Operationen, die die Paketverwaltung betreffen, benötigen Sie auch hier wieder root-Berechtigungen, die Sie am einfachsten über das `sudo`-Kommando erhalten.

Einen Nachteil bringt die Installation von Node.js über den systeminternen Paketmanager noch mit sich: Ihnen steht lediglich eine Version von Node.js zur Verfügung. Falls Sie Ihre Applikation unter verschiedenen Versionen von Node.js testen möchten, bleibt Ihnen nur der Weg über die Verwendung von verschiedenen Versionen der Linux Binaries von Node.js, oder Sie kompilieren verschiedene Versionen von Node.js aus den Quelldateien selbst. Wie Sie hierbei vorgehen müssen, erfahren Sie in Abschnitt 2.2, »Kompilieren und installieren«.

Haben Sie diese Schritte erfolgreich durchgeführt, steht Ihnen eine voll funktionsfähige Installation von Node.js auf Ihrem System zur Verfügung, und Sie können mit der Entwicklung eigener Applikationen beginnen. Wie aber erfahren Sie, ob die Installation funktioniert hat und ob Sie auch die aktuellste Version von Node.js ausführen? Die einfachste Variante, dies herauszufinden, besteht darin, dass Sie sich die Version Ihrer Node.js-Installation anzeigen lassen. In Listing 2.5 sehen Sie, wie Sie Node.js aufrufen müssen, um die Versionsinformationen anzuzeigen.

```

$ node -v
v0.10.9

```

Listing 2.5 Versionsinformationen von Node.js unter Linux anzeigen lassen

Wird Ihnen die korrekte Versionsnummer angezeigt, bedeutet das, dass die Installation erfolgreich war und Sie Node.js nun verwenden können. Die nächsten Abschnitte beschäftigen sich mit der Installation der Node.js-Plattform auf anderen Systemen. Sie können diese auch überspringen und direkt mit Kapitel 3, »Ein erstes Beispiel«, und hier dargestellten den ersten praktischen Beispielen unter Node.js fortfahren. Falls Sie die Node.js-Plattform wieder von Ihrem System entfernen möchten, können Sie dies, je nachdem, wie Sie Node.js installiert haben, auf einfache Art durchführen. Bei den Linux Binaries gestaltet sich das Entfernen am leichtesten. Im Laufe der Installation haben Sie ein gezipptes Tar-Archiv entpackt. Sobald Sie dieses entpackte Verzeichnis löschen, ist Node.js von Ihrem System wieder entfernt. Auch das Entfernen einer Node.js-Installation, die mithilfe des Paketmanagers durchgeführt wurde, gestaltet sich recht einfach, wie in Listing 2.6 zu sehen ist.


```
$ apt-get remove --purge nodejs  
$ apt-get clean
```

Listing 2.6 Deinstallation von Node.js unter Linux

Das Kommando `apt-get` mit der Option `remove` sorgt dafür, dass die Node.js-Installation auf Ihrem System gelöscht wird. Mit `--purge` werden auch eventuelle Konfigurationsdateien, die während der Installation erstellt wurden, vom System entfernt. Das nächste Kommando, `apt-get` mit der Option `clean`, ist dafür verantwortlich, dass der Paketcache geleert wird und so der belegte Speicherplatz wieder freigegeben wird. Im Paketcache liegen die Pakete, die zur Installation heruntergeladen wurden. So kann Node.js komplett aus dem System entfernt werden.

2.1.2 Windows

Ursprünglich war Node.js nur unter Linux verfügbar. Wollten Sie es unter Windows installieren, blieb Ihnen nichts anderes übrig, als es mithilfe von Cygwin zu betreiben. Dieses Projekt finden Sie unter der URL <http://www.cygwin.com/>. Cygwin stellt eine zu Linux ähnliche Umgebung unter Windows zur Verfügung, um hier Linux-Software zu betreiben. Mit der Version 0.6 tat sich viel hinsichtlich der Unterstützung unterschiedlicher Plattformen. Durch die Einführung der Bibliothek `libuv`, die Sie in Abschnitt 1.5.3 kennengelernt haben, und damit der Abstraktion von `libev` und `libeio`, wurde es möglich, Node.js nativ unter Windows zu betreiben, und das ganz ohne weitere Workarounds oder zusätzliche Software. Mittlerweile können Sie auf der Download-Seite von Node.js sowohl ein lauffähiges Binärpaket von Node.js als auch ein `.msi`-Paket zur Installation von Node.js herunterladen. Node.js kann auf fast jeder Windows-Plattform betrieben werden. Die Unterstützung reicht von Windows 2000 bis hin zu Windows 8 bei den Desktop-Systemen. Node.js kann aber auch auf den Serversystemen von Windows installiert werden. Außerdem ist Node.js auf der Microsoft Cloud-Plattform Azure verfügbar, sodass auch Cloud-Applikationen mit Node.js möglich werden.

Die einfachste Variante, wie Sie Node.js unter Windows betreiben können, ist der Einsatz des Binärpakets. Dieses müssen Sie lediglich herunterladen und können es dann ohne weitere Installation einsetzen. Gestartet werden kann Node.js anschließend auf zwei verschiedene Arten, entweder mit einem Doppelklick auf die Datei `node.exe` oder über die Kommandozeile. In Listing 2.7 sehen Sie, wie Sie prüfen können, ob Node.js auf Ihrem System funktioniert. Hier wird davon ausgegangen, dass die Datei `node.exe` direkt auf Ihrer Festplatte liegt. Sie können die Datei aber auch in einem beliebigen Verzeichnis ablegen.

```
C:\>node.exe -v  
v0.10.9
```

Listing 2.7 Versionsinformationen von Node.js unter Windows anzeigen lassen

Die Verwendung der Binärversion von Node.js unter Windows bringt einige entscheidende Nachteile mit sich. So können Sie Node.js auf der Kommandozeile nicht überall nur mit dem Befehl `node` starten. Um das zu erreichen, müssen Sie von Hand den Suchpfad des Systems erweitern beziehungsweise die Datei in einem Verzeichnis platzieren, das bereits im Suchpfad enthalten ist. Ein weiterer Nachteil dieser Variante besteht darin, dass Sie mit der Binärdatei lediglich die Node.js-Plattform einsetzen können, nicht aber den Node Package Manager, kurz: NPM. Diesen finden Sie unter <https://npmjs.org/>. Außerdem erfahren Sie in Abschnitt 4.4, »NPM«, mehr über den Paketmanager von Node.js. Diese Nachteile umgehen Sie durch die zweite Art, Node.js auf Windows zu betreiben. Sie haben neben der Verwendung der Binärdatei `node.exe` auch noch die Möglichkeit, Node.js mit einem .msi-Paket auf Ihrem System zu installieren. In nur wenigen Schritten werden Sie durch die Installation geleitet und können dann Node.js auf Ihrem System nutzen.

Nach einem Doppelklick auf das Installer-Paket von Node.js wird der interaktive Installer gestartet, und Sie befinden sich im Willkommensfenster, wie in Abbildung 2.2 zu sehen ist.



Abbildung 2.2 Erster Schritt bei der Installation von Node.js auf Windows

Über den Button **NEXT** gelangen Sie zum zweiten Schritt der Installation. Hier werden Ihnen die Lizenzinformationen für Node.js präsentiert, die Sie durch das Setzen eines Häkchens bei **I ACCEPT THE TERMS IN THE LICENSE AGREEMENT** akzeptieren müssen.

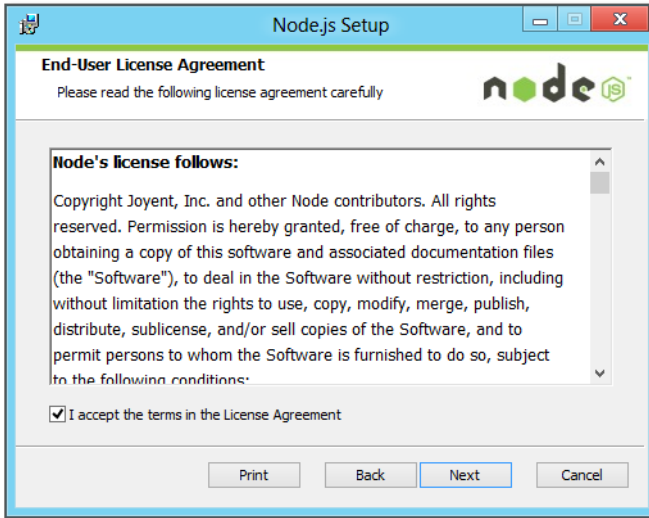


Abbildung 2.3 Lizenzinformationen von Node.js unter Windows

Sie haben hier die Möglichkeit, wie Abbildung 2.3 zeigt, die Lizenzinformationen auszudrucken, zurück zum ersten Schritt zu gehen oder aber die Lizenzbedingungen zu akzeptieren. Den Prozess der Installation können Sie im Übrigen jederzeit über den Button CANCEL abbrechen. Sobald Sie das Häkchen gesetzt haben, können Sie über den Button NEXT mit der eigentlichen Installation von Node.js beginnen. Hier werden die notwendigen Dateien an die richtigen Stellen im System kopiert, und die Software wird so registriert, dass Sie danach Node.js auf Ihrem System verwenden können.

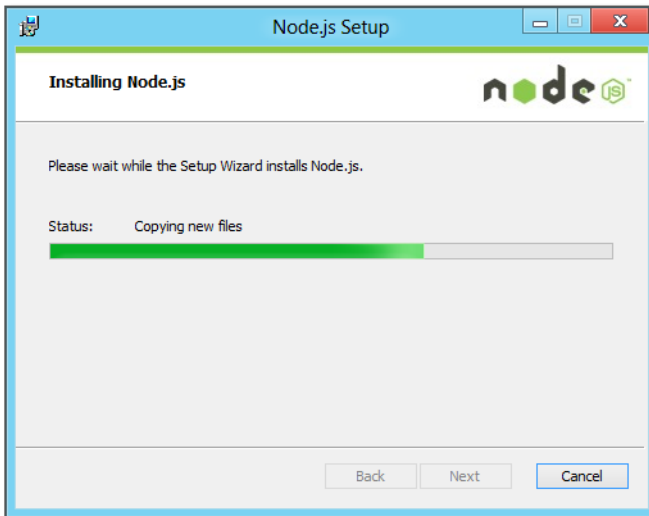


Abbildung 2.4 Installation von Node.js unter Windows

Sobald sämtliche Aufgaben durchgeführt sind, die erforderlich sind, damit Sie Node.js unter Windows betreiben können, wechselt die Anzeige auf den letzten Schritt der Installation. Hier können Sie über den Button **FINISH**, wie in Abbildung 2.5 zu sehen ist, die Installation von Node.js abschließen.

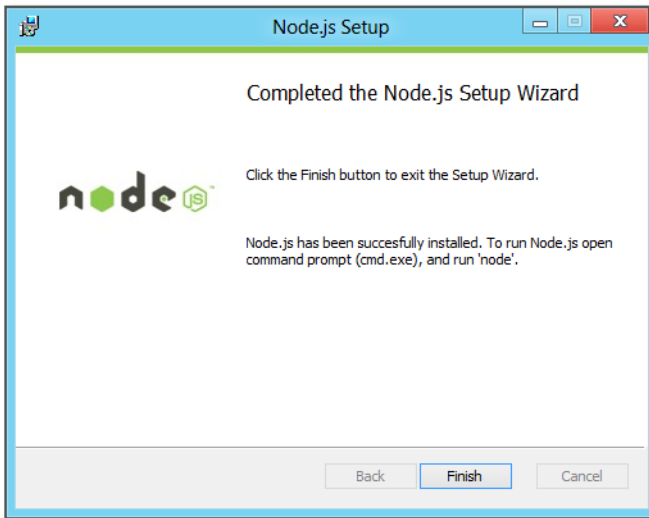


Abbildung 2.5 Installation unter Windows abschließen

Nachdem das Setup-Fenster geschlossen ist, können Sie das Eingabefenster (*cmd.exe*) öffnen und prüfen, ob die Installation von Node.js auf Ihrem System erfolgreich war. Zu diesem Zweck führen Sie, wie in Listing 2.8 beschrieben, den Befehl `node` mit der Option `-v` aus, um sich die Versionsinformation anzeigen zu lassen.

```
C:\>node -v
v0.10.9
```

Listing 2.8 Versionsinformationen von Node.js unter Windows anzeigen lassen

Bei der Installation mit dem `.msi`-Paket ist zu beachten, dass Ihnen Node.js jetzt systemweit auf der Kommandozeile zur Verfügung steht und Sie keinerlei Umgebungsvariablen anpassen müssen. Sie können jetzt damit beginnen, Ihre erste Anwendung mit Node.js umzusetzen.

Sollten Sie zu irgendeinem Zeitpunkt Node.js wieder von Ihrem System entfernen wollen, ist das auch kein Problem. Je nachdem, wie Sie bei der Installation vorgegangen sind, können Sie eine von zwei Möglichkeiten wählen, um Node.js wieder von Ihrem Windows-System zu entfernen. Setzen Sie die Windows Binaries von Node.js ein, müssen Sie lediglich die `node.exe`-Datei von Ihrem System entfernen. Haben Sie Node.js durch das `.msi`-Paket auf Ihrem System installiert, können Sie die Software-

verwaltung von Windows verwenden, um Node.js zu entfernen. Sie gelangen in Windows 8 über den Windows Explorer zur Systemsteuerung, indem Sie links COMPUTER und in der Menüleiste den Punkt COMPUTER auswählen und dann auf SYSTEMSTEUERUNG klicken. Dort gelangen Sie über den Link PROGRAMM DEINSTALLIEREN unter dem Punkt PROGRAMME zu einer Übersicht, in der Sie den Punkt PROGRAMM DEINSTALLIEREN unter PROGRAMME UND FUNKTIONEN wählen müssen, um eine Liste aller installierten Softwarekomponenten auf Ihrem System zu erhalten. In dieser Liste gibt es einen Eintrag mit dem Namen NODE.JS. Wählen Sie diesen aus und klicken dann auf DEINSTALLIEREN, entfernen Sie Node.js von Ihrem System.

Das übrige Kapitel beschäftigt sich mit der Installation von Node.js unter Mac OS X und mit dem Kompilieren von Node.js aus den Quelldateien. Falls Sie direkt in die Entwicklung mit Node.js einsteigen möchten, können Sie auch gleich mit Kapitel 3, »Ein erstes Beispiel«, fortfahren.

2.1.3 Mac OS X

Für Mac OS X steht Ihnen, wie auch schon für Windows, ein Binary-Paket und ein Installer-Paket zur Installation als Download zur Verfügung. Die einfachste Variante, Node.js unter Mac OS X zu betreiben, besteht in der Verwendung des Binärpakets. Das Paket liegt als gezipptes Tar-Archiv vor, und Sie müssen es lediglich, wie Sie auch in Listing 2.9 sehen können, auf der Kommandozeile entpacken.

```
$ tar xvzf node-v0.10.9-darwin-x64.tar.gz
$ cd node-v0.10.9-darwin-x64
$ bin/node -v
v0.10.9
```

Listing 2.9 Installation des Binärpakets von Node.js unter Mac OS X

Wie Listing 2.9 zeigt, kann Node.js nach dem Entpacken direkt verwendet werden. Mit der Option `-v` können Sie sich die Versionsinformation anzeigen lassen und so prüfen, ob es sich um die korrekte Version handelt und ob Node.js prinzipiell funktioniert. Die Installation des Binärpakets hat den Nachteil, dass Node.js nicht systemweit auf der Kommandozeile zur Verfügung steht. Abhilfe für dieses Problem schafft die Verwendung des Installers.

Die Installation wird durch einen Doppelklick auf die heruntergeladene *pkg*-Datei gestartet. Danach werden Sie durch einen mehrstufigen interaktiven Prozess geleitet, an dessen Ende eine funktionsfähige Installation von Node.js steht. Abbildung 2.6 zeigt den ersten Schritt der Installation, in dem Sie darauf hingewiesen werden, dass Node.js und NPM in das Verzeichnis `/usr/local/bin` installiert werden. Durch einen Klick auf CONTINUE setzen Sie die Installation fort.

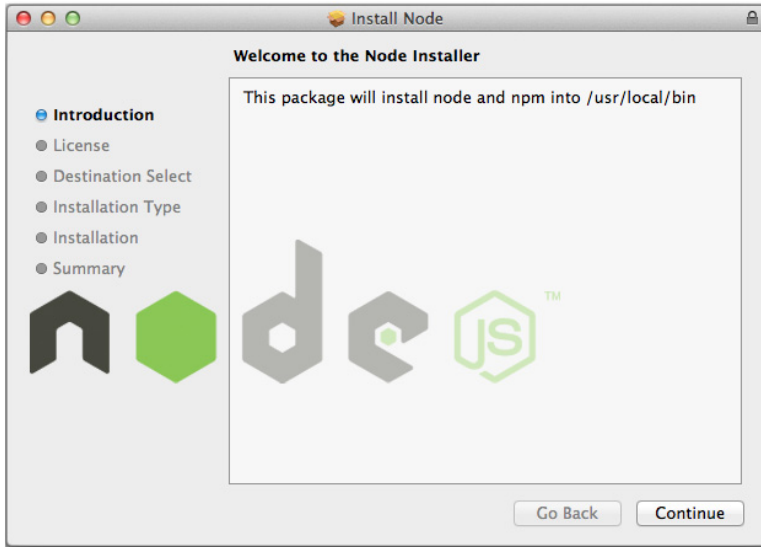


Abbildung 2.6 Installation von Node.js unter Mac OS

Im nächsten Schritt der Installation werden Ihnen die Lizenzbedingungen von Node.js präsentiert. Dabei haben Sie, wie in Abbildung 2.7 zu sehen ist, die Möglichkeit, verschiedene Sprachen zu wählen, in denen Ihnen die entsprechenden Informationen angezeigt werden. Außerdem können Sie die Lizenzbedingungen ausdrucken oder auf Ihrem System speichern.

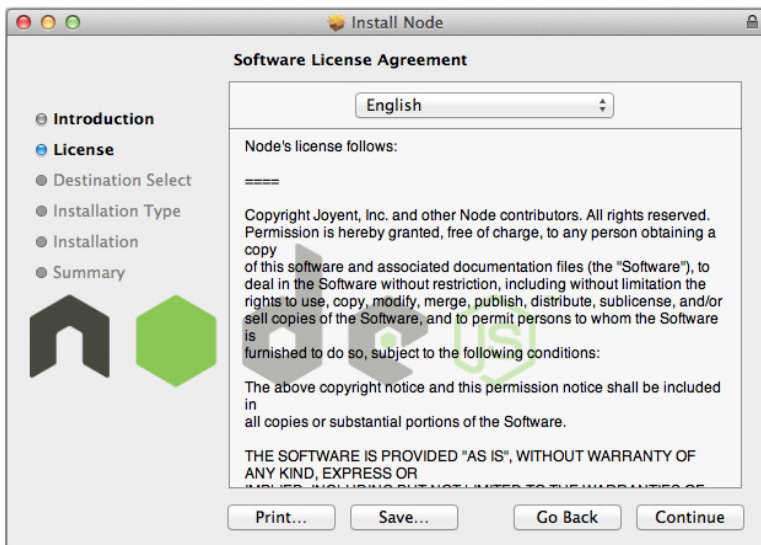


Abbildung 2.7 Lizenzbedingungen von Node.js für Mac OS X

Im weiteren Verlauf der Installation können Sie über den Button GO BACK jeweils zum vorherigen Installationsschritt zurückkehren. Nach einem Klick auf CONTINUE werden Sie in einem separaten Popup-Fenster gefragt, ob Sie den Lizenzbedingungen zustimmen und mit der Installation fortfahren möchten. Sie haben hier die Auswahl, entweder zuzustimmen, dies geschieht mit einem Klick auf den Button AGREE, und den Installationsprozess damit fortzusetzen oder aber nicht zuzustimmen und die Installation damit abubrechen.

Haben Sie die Lizenzbedingungen akzeptiert, wird Ihnen als Nächstes die Frage gestellt, wie Sie Node.js auf Ihrem Rechner installieren möchten. Klicken Sie hierzu INSTALL FOR ALL USERS OF THIS COMPUTER an, damit Node.js systemweit zur Verfügung gestellt wird. Abbildung 2.8 zeigt diesen Schritt der Installation.

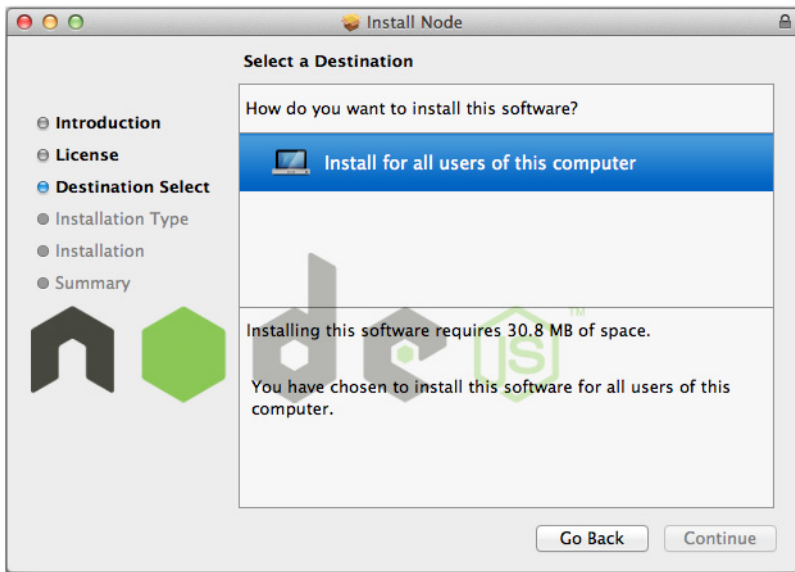


Abbildung 2.8 Art der Installation von Node.js unter Mac OS X auswählen

Hier erhalten Sie außerdem den Hinweis, wie viel Speicherplatz nach der Installation von Node.js zusätzlich belegt sein wird. Mit einem Klick auf CONTINUE gelangen Sie zum nächsten Schritt, in dem Sie den Speicherort von Node.js anpassen können, wie in Abbildung 2.9 zu sehen ist.

Bestätigen Sie über einen Klick auf INSTALL, dass Sie mit der Platzierung von Node.js auf Ihrem System einverstanden sind. Nun müssen Sie noch Ihren Benutzernamen und das zugehörige Passwort angeben, damit die Software korrekt auf Ihrem System installiert werden kann. Sobald Sie diese Angaben gemacht und bestätigt haben, beginnt die eigentliche Installation von Node.js, und die Dateien werden aus dem Installer-Paket heraus ins System kopiert. Ab diesem Zeitpunkt können Sie nicht mehr über Go Back zurückspringen.

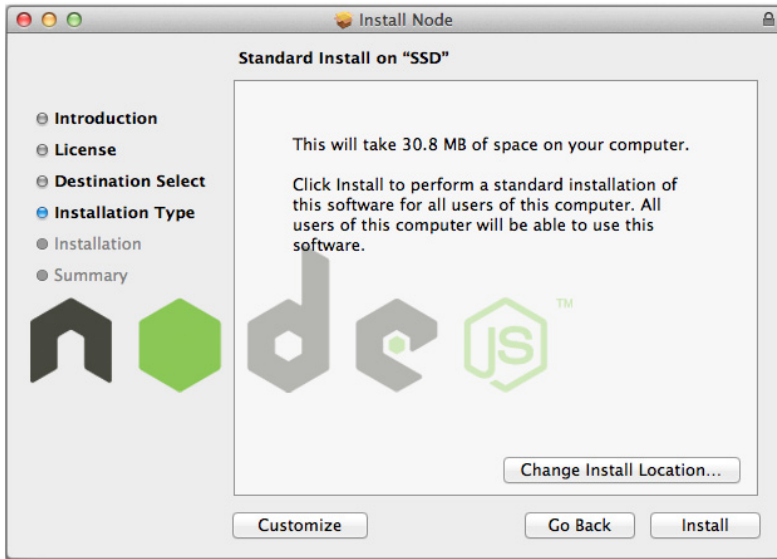


Abbildung 2.9 Anpassen des Speicherorts von Node.js unter Mac OS X

Nachdem sämtliche Dateien kopiert sind, ist die Installation abgeschlossen, und Sie erhalten eine kurze Zusammenfassung, wie sie in Abbildung 2.10 zu sehen ist.

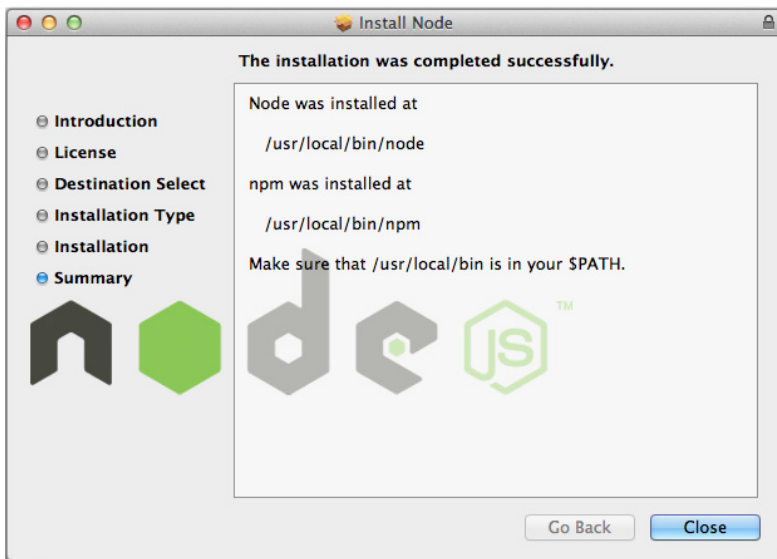


Abbildung 2.10 Abschluss der Installation von Node.js unter Mac OS X

Nun erfahren Sie noch, wohin Node.js und NPM installiert wurden und welches Verzeichnis Sie in Ihre PATH-Variable aufnehmen müssen, damit Sie Node.js auf der Kom-

mandozeile mit dem Befehl `node` verwenden können, ohne den gesamten Pfad mit angeben zu müssen. Standardmäßig befindet sich das Verzeichnis `/usr/local/bin` im Suchpfad des Systems, wie Sie in Listing 2.10 sehen.

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
```

Listing 2.10 Abfrage der `$PATH`-Variablen unter Mac OS X

Haben Sie die Schritte der Installation von Node.js wie beschrieben durchgeführt, können Sie die Plattform nun verwenden. Um sicherzustellen, dass wirklich die aktuelle Version installiert wurde und ob Node.js auch korrekt funktioniert, können Sie sich die Versionsinformationen auf der Kommandozeile ausgeben lassen. Listing 2.11 zeigt, wie Sie hier vorgehen müssen.

```
$ node -v
v0.10.9
```

Listing 2.11 Abfrage der Versionsinformationen von Node.js unter Mac OS X

Für den Fall, dass Sie Node.js von Ihrem System wieder entfernen möchten, haben Sie, wenn Sie das Binärpaket einsetzen, die Möglichkeit, das zuvor entpackte Verzeichnis wieder zu löschen.

Haben Sie Node.js über den Installer installiert, steht Ihnen ebenfalls die Möglichkeit zur Verfügung, Node.js wieder von Ihrem System zu entfernen. In diesem Fall nutzen Sie aus, dass der *pkg*-Installer die Dateien protokolliert, die auf Ihr System kopiert werden. Die Informationen sind unter `/var/db/receipts/` zu finden und liegen in den Dateien *org.nodejs.node.npm.pkg.bom* und *org.nodejs.pkg.bom*. Die Dateiendung *.bom* steht für »Bill of Material«, was übersetzt Stückliste bedeutet. In diesen Dateien ist der Speicherort der Dateien relativ zum Ort der Installation festgehalten. Da sämtliche Informationen, wie die Dateiliste und auch der Ort der Installation, im Binärformat vorliegen, steht Ihnen mit *pkgutil* ein Tool zur Verfügung, mit dem Sie die Informationen ausgeben können. *pkgutil* arbeitet mit Paketnamen. Der Name für Ihr Node.js-Paket lautet *org.nodejs.pkg*. Die Metainformationen lassen sich, wie in Listing 2.12 gezeigt, durch die Option `--pkg-info` anzeigen.

```
$ pkgutil --pkg-info org.nodejs.pkg
package-id: org.nodejs.pkg
version: 1.0
volume: /
location: usr/local
install-time: 1352148814
```

Listing 2.12 Metainformationen zum Node.js-Paket unter Mac OS X

Wie Listing 2.12 zeigt, haben Sie Node.js auf dem Volume / unter dem Pfad *usr/local* installiert. Dieser Pfad stellt das Präfix für die Dateiliste dar, die Sie aus der *.bom*-Datei gewinnen können. Listing 2.13 zeigt, wie Sie hier vorzugehen haben.

```
$ pkgutil --lsbom org.nodejs.pkg
.
./bin
./bin/node
./bin/node-waf
./bin/npm
./include
./include/node
```

Listing 2.13 Liste der installierten Dateien unter Mac OS X

In früheren Versionen stand für *pkgutil* die Option *--unlink* zur Verfügung, um ein Paket komplett zu löschen. In neueren Versionen wurde diese Option entfernt, da das Entfernen eines Pakets Auswirkungen auf andere installierte Software auf Ihrem System haben kann. Aus diesem Grund ist auch äußerste Vorsicht geboten, wenn Sie Software, die mit dem *pkg*-Installer installiert wurde, wieder entfernen möchten. Im einfachsten Fall entfernen Sie die installierten Dateien manuell oder erstellen ein Shellscript, das über die Dateiliste iteriert und die Dateien entfernt. Haben Sie die Dateien des Pakets entfernt, können Sie die Paketinformationen mit *pkgutil* wieder entfernen. Listing 2.14 zeigt Ihnen die dafür notwendige Kommandozeile.

```
$ pkgutil --forget org.nodejs.pkg
```

Listing 2.14 Entfernen der Paketinformationen unter Mac OS X

Die Installation von Node.js über ein vorgefertigtes Paket bietet viele Vorteile. Allen voran ist die Installation problemlos, und Sie müssen keine Abhängigkeiten beachten. Außerdem ist es möglich, diese Pakete wieder vom System zu entfernen. Neben dieser Variante der Installation steht noch eine weitere Möglichkeit zur Verfügung. Sie können sich den Quellcode von Node.js herunterladen und diesen selbst kompilieren und auf dem System installieren. Die nächsten Abschnitte beschreiben, wie Sie von den Quelldateien bis zu einer lauffähigen Version von Node.js gelangen.

Nachdem Sie die einzelnen Schritte zur Installation nachvollzogen haben, verfügen Sie jetzt über eine lauffähige Version von Node.js, mit der Sie Ihre eigenen Applikationen entwickeln können. In Kapitel 3, »Ein erstes Beispiel«, werden Sie auf der Basis Ihrer Installation Schritt für Schritt angeleitet, Ihre erste Anwendung unter Node.js zu programmieren. Der Rest dieses Kapitels beschäftigt sich mit dem Thema, wie Sie Node.js von Hand aus dem Quelltext kompilieren und installieren. Sollte das für Sie keine Option sein, können Sie direkt mit Kapitel 3 fortfahren.

2.2 Kompilieren und installieren

In diesem Abschnitt erfahren Sie, wie Sie Node.js nicht mithilfe von Binärpaketen oder durch eine Installation mit einem Installer für Ihr System betreiben können, sondern wie Sie eine maßgeschneiderte Installation aus den Quelldateien für Ihr System erstellen. Im Folgenden lernen Sie, wie Sie Node.js auf einem Linux-System kompilieren. Zu diesem Zweck wird von einer Standardinstallation eines Ubuntu-Desktops in der Version 12.04 LTS ausgegangen. Der erste Schritt der Installation besteht darin, dass Sie sich den Sourcecode von Node.js herunterladen müssen. Dieser ist auf nodejs.org im Download-Bereich als gezipptes Tar-Archiv verfügbar. Bevor Sie mit dem Kompilieren von Node.js beginnen können, sollten Sie sicherstellen, dass Sie über einen Compiler verfügen. Dies ist standardmäßig nicht der Fall bei der Desktop-Variante von Ubuntu. Sie können den Compiler jedoch über den Paketmanager des Systems nachinstallieren. In Listing 2.15 sehen Sie die dafür notwendige Kommandozeile.

```
$ sudo apt-get install g++
```

Listing 2.15 Installation der GNU Compiler Collection unter Ubuntu

Nachdem diese Voraussetzung erfüllt ist, können Sie, wie Sie Listing 2.16 entnehmen können, das Quellcodepaket entpacken und in das entstandene Verzeichnis wechseln.

```
$ tar xvzf node-v0.10.9.tar.gz
$ cd node-v0.10.9
```

Listing 2.16 Entpacken des Node.js-Quellcodes

Der erste Schritt des eigentlichen Kompilierungsvorgangs besteht in der Erstellung eines *Makefiles*. Diese Datei dient später dem `make`-Kommando als Grundlage für die Übersetzung von Node.js. Das *Makefile* müssen Sie allerdings nicht von Hand erstellen, hierfür existiert das Programm `configure`. Diesem Kommando können zusätzliche Informationen übergeben werden, die für die Installation relevant sind. Optionen sind beispielsweise `--without-npm`, um den Paketmanager von Node.js nicht mit zu installieren, oder `--without-ssl`, um Node.js ohne SSL zu bauen. Außerdem gibt es die Möglichkeit, verschiedene Bibliotheken wie V8, OpenSSL oder zlib als Shared Libraries einzubinden, statt sie statisch zu verlinken. Mit der Option `--dest-cpu=` können Sie die Zielarchitektur wählen, mögliche Werte sind hier `arm`, `ia32` und `x64`. Sie können außerdem das Betriebssystem auswählen, für das Node.js gebaut werden soll. In diesem Fall können Sie der Option `--dest-os=` unter anderem die Werte `win`, `mac` oder `linux` übergeben.

Eine wichtige Option für `configure` ist `--prefix`. Diese gibt an, an welchem Ort Node.js installiert werden soll. Verwenden Sie diese Option nicht, wird der Standard `/usr/local` verwendet. Relativ zu diesem Pfad werden sämtliche Dateien von Node.js auf das Sys-

tem kopiert. Diese Option können Sie nutzen, um sich die Möglichkeit offenzuhalten, mehrere Versionen von Node.js auf Ihrem System zu installieren. Mit dem Kommando `./configure --help` erhalten Sie eine Liste aller verfügbaren Konfigurationsoptionen inklusive einer kurzen Erklärung.

Listing 2.17 zeigt die nächsten Schritte im Prozess.

```
$ ./configure --prefix=/opt/node/node-v0.10.9
{ 'target_defaults': ...
$ make
python tools/gyp_node -f make
...
$ sudo make install
make -C out BUILDTYPE=Release V=1
...
```

Listing 2.17 Kompilieren und Installieren von Node.js unter Ubuntu

Das erste Kommando in Listing 2.17, `configure`, dient wie bereits erwähnt zur Erstellung des *Makefiles* für Node.js. Die Option `--prefix=/opt/node/node-v0.10.9` verwenden Sie dazu, um Node.js in einem separaten Verzeichnis zu installieren. Diese Information wird im *Makefile* festgehalten. Das nächste Kommando, `make`, baut auf den Informationen im *Makefile* auf, sucht darin nach dem ersten Target und baut nach den dort vorliegenden Informationen das Projekt, also in Ihrem Fall Node.js. Ziel dieses Prozessschrittes ist es, eine ausführbare Version von Node.js zu erhalten, die sich allerdings noch in dem Verzeichnis befindet, in dem Sie sich aktuell befinden, also im entpackten Quellcode. Das letzte Kommando, `make install`, sorgt schließlich dafür, dass die kompilierten Dateien an die richtige Stelle im System kopiert werden. Aus diesem Grund müssen Sie `make install` auch mit root-Berechtigungen ausführen, da ein Nutzer im Normalfall keine Schreibberechtigung auf `/opt`, Ihr aktuelles Installationsziel, haben sollte.

Nach diesen Schritten sollten Sie auf Ihrem System im Verzeichnis `/opt` ein neues Verzeichnis `node` sehen können. In diesem liegt nun Ihr Node.js. Sie können dies bereits testen, indem Sie im Unterverzeichnis `node-v0.10.9/bin` das Kommando `node` mit der Option `-v` ausführen. Als Ausgabe erhalten Sie die Versionsnummer Ihrer Node.js-Installation. Hat dieser Test funktioniert, können Sie das Verzeichnis mitsamt dem entpackten Quellcode von Ihrem System entfernen, da Sie beides nicht mehr benötigen. Wie bei den Binärpaketen unter Linux haben Sie bei dieser Installationsvariante ebenfalls das Problem, dass Sie Node.js nicht systemweit von der Kommandozeile ausführen können, sondern stets den Pfad zur ausführbaren Datei vollständig angeben müssen. Diesem Problem widmet sich der nächste Abschnitt dieses Kapitels.

Mit der Installation im `/opt`-Verzeichnis haben Sie die Möglichkeit, mehrere Versionen von Node.js parallel auf Ihrem System zu betreiben. Falls Sie eine Applikation ent-

wickeln möchten, die auf verschiedenen Versionen von Node.js funktionieren soll, und Sie dies testen möchten, bietet sich eine derartige Vorgehensweise an. So können Sie beispielsweise eine Version von 0.6, eine von 0.8 und eine von 0.10 betreiben. Im Normalfall möchten Sie allerdings wahrscheinlich die aktuellste Version von Node.js global auf Ihrem System verfügbar haben. Die einfachste und schnellste Variante, dies zu erreichen, ist ein symbolischer Link auf die Version von Node.js, die Sie global verfügbar haben möchten, und die Aufnahme dieses symbolischen Links in den Suchpfad Ihres Systems. Diese Vorgehensweise ermöglicht es Ihnen, durch die Anpassung dieses Links die Version von Node.js umzuschalten. Listing 2.18 demonstriert Ihnen die Erstellung des besagten Links.

```
$ ln -s /opt/node/node-v0.10.9 /opt/node/node
```

Listing 2.18 Erstellung eines symbolischen Links

Als letzten Schritt müssen Sie den Suchpfad mit einem Eintrag in der */etc/profile*-Datei, wie Sie in Listing 2.19 sehen können, erweitern.

```
PATH=$PATH:/opt/node/node/bin  
export PATH
```

Listing 2.19 Erweiterung des Suchpfades unter Linux

Sie verfügen nun über eine funktionsfähige Installation von Node.js und können diese ganz einfach durch die Eingabe des Kommandos `node` mit der Option `-v` auf der Kommandozeile testen. Die Ausgabe ist die Versionsnummer der Node.js-Installation. Sie können jetzt damit beginnen, eigene Applikationen mit Node.js zu entwickeln. Kapitel 3, »Ein erstes Beispiel«, bietet Ihnen einen ersten Einstieg in die Entwicklung mit Node.js und zeigt Schritt für Schritt die Implementierung einer ersten Anwendung.

2.3 Zusammenfassung

Im Laufe dieses Kapitels haben Sie gesehen, wie einfach es ist, Node.js auf Ihrem System zu installieren. Dabei ist es egal, ob Sie ein Windows-, Linux- oder Mac OS X-System betreiben. Sie können Node.js auf verschiedene Arten auf Ihrem System installieren. Dabei reichen die Varianten von der einfachen Installation der Binärpakete, die Sie lediglich auf Ihrem System entpacken müssen, bis hin zum Kompilieren und Installieren des Quellcodes von Node.js. Je nachdem, welche Variante der Installation Sie gewählt haben, können Sie die installierte Software auch wieder von Ihrem System entfernen. Im nächsten Kapitel sehen Sie, wie Sie ausgehend von Ihrer Installation eine erste Applikation mit Node.js entwickeln können.

Kapitel 3

Ein erstes Beispiel

*Wenn wir die Unbekannte zu suchen streben, dann werden wir tapfer,
besser und aktiver als diejenigen, die denken, dass die Unbekannte
nicht möglich zu finden ist.*
– Platon

Wenn Sie hier angekommen sind, sollten Sie bereits über eine funktionsfähige Installation der Node.js-Plattform auf Ihrem System verfügen. Für die folgenden Beispiele wird Node.js auf der Kommandozeile eines Linux-Systems verwendet. Die Beispiele funktionieren allerdings analog auch unter Windows und Mac OS X. Sie können Node.js auf zwei verschiedene Arten verwenden. Zum einen gibt es eine interaktive Shell und zum anderen einen Modus, bei dem statt einer Nutzerinteraktion eine Datei mit JavaScript-Quellcode übergeben wird, der ausgeführt werden soll.

3.1 Der interaktive Modus

Zunächst erfahren Sie mehr über den interaktiven Modus von Node.js. Diesen erreichen Sie, wie Sie Listing 3.1 entnehmen können, über die Eingabe des Kommandos `node` auf der Kommandozeile.

```
$ node  
>
```

Listing 3.1 Der interaktive Modus von Node.js

Im interaktiven Modus können Sie direkt JavaScript-Code auf der Kommandozeile eingeben und diesen ausführen. Diese Art der Benutzerschnittstelle wird REPL genannt. Das Akronym steht für Read-Eval-Print Loop und bedeutet, dass Kommandos auf der Kommandozeile eingelesen und danach evaluiert werden und im Anschluss daran das Ergebnis auf der Kommandozeile ausgegeben wird. Dieser Modus ist nicht dazu gedacht, komplette Applikationen zu implementieren und auszuführen. Stattdessen dient diese Schnittstelle von Node.js dazu, das Verhalten und die Funktionalität einzelner Codeschnipsel zu testen. Listing 3.2 zeigt Ihnen, wie Sie Kommandos im Node.js REPL absetzen können.

```
$ node
> console.log("Hallo Welt!");
Hallo Welt!
undefined
>
```

Listing 3.2 Kommandos im Node.js REPL ausführen

Die JavaScript-Kommandos werden auch im Node.js REPL durch ein Semikolon abgeschlossen. Ein Zeilenumbruch beendet die Eingabe des aktuellen Kommandos und sendet den Befehl an die JavaScript-Engine. `console.log` mit dem Argument `Hallo Welt!` wird evaluiert, und das Ergebnis `Hallo Welt!` wird auf der Kommandozeile ausgegeben. Wie Sie in Listing 3.2 sehen, wird neben dem erwarteten `Hallo Welt!` außerdem noch der Wert `undefined` ausgegeben. Das liegt daran, dass der Rückgabewert der Funktion `console.log` dargestellt wird, und zwar in diesem Fall `undefined`. Neben der Ausführung von Ausgaben können Sie im REPL sämtliche JavaScript-Kommandos, die unter Node.js zur Verfügung stehen, ausführen. Das Beispiel in Listing 3.3 verdeutlicht Ihnen dies.

```
> function greet(name) {
  console.log("Hallo " + name);
}; greet("Welt");
Hallo Welt
undefined
>
```

Listing 3.3 Definition einer einfachen Node.js REPL-Funktion

In Listing 3.3 sehen Sie, wie man eine Funktion im Node.js REPL definieren kann, die als Parameter einen Namen erhält, und wie man diesen im Verlauf der Funktion auf der Konsole ausgibt. Nach der Definition wird die Funktion `greet` mit dem Argument `Welt` als Name aufgerufen. Die Ausgabe besteht aus `Hallo Welt` und `undefined`. Wenn Sie im Node.js REPL verschiedene Befehle ausführen möchten, die aufeinander aufbauen, können Sie dies ohne Weiteres durchführen. Definieren Sie beispielsweise Funktionen oder führen Sie eine Variablenzuweisung durch, bleiben diese auch nach der Evaluierung der Kommandozeile erhalten. Wie sich diese Tatsache konkret auswirkt, sehen Sie in Listing 3.4.

```
$ node
> var myVar = "Hallo Welt!";
undefined
> console.log(myVar);
```

```
Hallo Welt!  
undefined  
>
```

Listing 3.4 Voneinander abhängige Befehlszeilen

Im Beispiel in Listing 3.4 weisen Sie der Variablen `myVar` einen Wert zu. Diese Variable wird in der nächsten Anweisung wieder in der Ausgabe verwendet. Zu den wichtigsten Features des Node.js REPL zählt die Autovervollständigung. Sie können dieses Feature nutzen, indem Sie beliebige Buchstaben auf der Kommandozeile eingeben und die Tab-Taste drücken. Node.js zeigt die verfügbaren Ergänzungen an. Ein weiteres erwähnenswertes Feature sind die mehrzeiligen Befehle. Wie Sie in Listing 3.3 sehen können, wird der Funktionscode nicht auf einer einzigen Zeile eingegeben, sondern auf mehreren. Ein nicht abgeschlossenes Kommando wird durch drei Punkte statt des Größerals-Zeichens als Kommandoprompt angezeigt. Zu erreichen sind mehrzeilige Kommandos beispielsweise durch nicht abgeschlossene Codeblöcke oder einen Plus-Operator zur Berechnung oder Stringkonkatenation am Zeilenende.

Zu den Besonderheiten des Node.js REPL zählt, dass er Ihnen neben dem JavaScript-Befehlssatz noch einige weitere Befehle zur Steuerung des REPL bietet. Die Kommandos beginnen stets mit einem Punkt und müssen nicht durch ein Semikolon beendet werden. Tabelle 3.1 zeigt Ihnen diese Befehle in der Übersicht.

Kommando	Bedeutung
<code>.break</code>	Beendet die aktuelle Eingabe. <code>.break</code> ist vor allem bei mehrzeiligen Kommandos hilfreich.
<code>.clear</code>	Alias auf <code>.break</code>
<code>.exit</code>	Beendet den Node.js REPL.
<code>.help</code>	Gibt eine Liste der verfügbaren Kommandos aus.
<code>.load</code>	Lädt eine gespeicherte Session in den aktuellen REPL.
<code>.save</code>	Speichert die Kommandos der aktuellen Session in eine Datei.

Tabelle 3.1 Verfügbare REPL-Befehle

Um den REPL wieder zu verlassen, haben Sie zwei Möglichkeiten. Sie können das Kommando `.exit` verwenden, um den Node.js REPL zu schließen. Alternativ können Sie die Tastenkombination `[Strg] + [D]` benutzen, was den Prozess sofort beendet. Die zweite Variante besteht in der Tastenkombination `[Strg] + [C]`, die Sie zweimal betätigen müssen, was das Signal SIGINT an den Node.js-Prozess sendet und diesen

beendet. Diese Art der Signale wird verwendet, um Prozesse von außen zu steuern, um sie beispielsweise zu beenden.

Die Befehle `.break` und `.clear` kommen zum Einsatz, wenn die Kommandozeile durch Falscheingaben blockiert ist. Listing 3.5 zeigt einen Anwendungsfall für diese Kommandos.

```
> console.log("Hallo " + name);  
... .break  
>
```

Listing 3.5 Einsatz von `.break`

Im Beispiel in Listing 3.5 folgt der Variablen `name` fälschlicherweise ein doppeltes Anführungszeichen, was dazu führt, dass nach dem Absenden des Befehls dieser nicht evaluiert wird, sondern eine weitere Eingabezeile zur Verfügung gestellt wird. Stellen Sie fest, dass Sie einen ähnlichen Fehler bei der Eingabe gemacht haben, können Sie die aktuelle Eingabe durch den Befehl `.break` beenden und Ihr Kommando erneut eingeben. Der Node.js REPL bietet Ihnen hier die Möglichkeit, durch die Historie der letzten Befehle zu navigieren. Mit diesem Feature müssen Sie Ihren Befehl nicht erneut eingeben, sondern können mit der `↑`- und `↓`-Taste die entsprechende Kommandozeile zurückholen, diese korrigieren und erneut absenden.

Falls Sie umfangreichere Versuche im REPL durchführen möchten oder die Ergebnisse festhalten wollen, stehen Ihnen mit `.save` und `.load` zwei Kommandos zur Verfügung, über die Sie die bisher abgesetzten Kommandos in eine Datei speichern können beziehungsweise eine Datei mit JavaScript- Anweisungen in den aktuellen REPL laden können.

```
> console.log("Hallo Welt!");  
Hallo Welt!  
undefined  
> .save myShell.js  
Session saved to:myShell.js  
> .load myShell.js  
> console.log("Hallo Welt!");  
Hallo Welt!  
undefined  
>
```

Listing 3.6 Einsatz von `.load` und `.save`

In Listing 3.6 sehen Sie, wie Sie `.load` und `.save` verwenden können. Um den Einsatz zu veranschaulichen, setzen Sie zu Beginn den Befehl `console.log` ab. Danach wird die aktuelle Session in die Datei `myShell.js` gespeichert. Diese Datei enthält dann die

Befehlszeile, wie Sie sie im REPL eingegeben haben, allerdings nicht die zugehörigen Ausgaben. Im nächsten Schritt laden Sie diese Datei mit der `.load`-Anweisung wieder in die Session. Die Datei wird dabei zeilenweise ausgelesen, jedes Kommando ausgeführt, und die zugehörigen Ausgaben werden jeweils angezeigt. Sie können den Befehl `.load` auch dazu verwenden, um eine bestimmte Ausgangssituation für einen Versuch vorzubereiten. Zu diesem Zweck können Sie eine Reihe von Befehlen in einer Datei formulieren, um beispielsweise Variablen oder Funktionen zu definieren, die Sie im Verlauf einer Session im Node.js REPL benötigen.

Wie in JavaScript üblich, bietet auch der Node.js REPL einen globalen Kontext, in dem zu Beginn bereits einige Variablen registriert sind, die Ihnen als Entwickler die Arbeit erleichtern. So steht Ihnen das Objekt `process` zur Verfügung, mit dem Sie Metainformationen über den laufenden Node.js-Prozess, wie beispielsweise die Versionsnummer, auslesen können. Außerdem können Sie mithilfe von `require` zusätzliche Funktionalitäten laden. Auf die genaue Funktionsweise dieser Features wird im Verlauf dieses Buches noch detaillierter eingegangen. Im globalen Scope des Node.js REPL steht Ihnen mit der Variablen `_` eine weitere Besonderheit zur Verfügung. Diese Variable enthält stets den Wert des letzten Kommandos. Setzen Sie beispielsweise das Kommando `console.log` ab, enthält `_` den Wert `undefined`. Rufen Sie eine Methode oder Methode mit einem Rückgabewert wie zum Beispiel `process.uptime()` auf, enthält `_` den jeweiligen Wert. Mit dem Node.js REPL als Werkzeug können Sie auf einfache Art und Weise Quellcode testen und sehen interaktiv, wie sich die Node.js-Plattform in bestimmten Situationen verhält. Für umfangreiche Anwendungen eignet sich der Node.js REPL nicht. In diesem Fall kommt eine andere Art der Ausführung von Node.js zum Einsatz. Über diese erfahren Sie in den folgenden Abschnitten noch mehr.

3.2 Die erste Applikation

Wenn Sie Applikationen mit Node.js erstellen, liegt der Quellcode dieser Applikation in einer oder mehreren Dateien vor. Diese werden Node.js beim Start der Applikation als Option auf der Kommandozeile übergeben und durch den Prozess eingelesen und ausgeführt. Die Eingabedatei enthält nur gültigen JavaScript-Quellcode. Listing 3.7 zeigt Ihnen, wie Sie eine Applikation mit Node.js ausführen.

```
$ node server.js
Hallo Welt!
$
```

Listing 3.7 Ausführung von Node.js-Applikationen

Die Datei `server.js` enthält lediglich die Zeile `console.log("Hallo Welt!")`. Der Vorteil dieser Art der Ausführung besteht darin, dass Sie die Applikation beliebig oft starten

können und Sie dafür nur die Kommandozeile aus Listing 3.7 ausführen müssen. Außerdem können Sie den Quellcode ohne Weiteres auch auf anderen System ausführen oder die Applikation als Open-Source-Software anderen Nutzern zur Verfügung stellen.

Die Ausgabe von `Hallo Welt!` stellt allerdings noch keine Applikation dar. Mit Node.js haben Sie jedoch die Möglichkeit, dynamische Webapplikationen zu erstellen und benötigen hierzu keinen separaten Webserver, denn Node.js bietet Ihnen mit dem `http`-Modul die Möglichkeit, einen eigenen Webserver zu erstellen. In den nachfolgenden Abschnitten werden Sie Schritt für Schritt das klassische Beispiel einer Node.js-Anwendung durchgehen: einen sehr leichtgewichtigen Webserver. Dieser soll Anfragen von Browsern annehmen können und darauf mit einer korrekten HTTP-Antwort und der Ausgabe der Zeichenkette `Hallo, ich bin ein Node.js Webserver` reagieren. Sie beginnen mit einem Grundgerüst und erweitern dieses, bis die Applikation die Anforderungen erfüllt. Listing 3.8 Ihnen zeigt den grundsätzlichen Aufbau der Anwendung. Diesen Code sollten Sie in einer separaten Datei mit dem Namen `webserver.js` speichern.

```
var http = require('http');
http.createServer().listen(8080, '127.0.0.1');
console.log('Webserver wird ausgeführt.');
```

Listing 3.8 Aufbau eines Webservers in Node.js

Eigentlich verfügt eine JavaScript-Engine wie V8 nicht über die Möglichkeiten, auf einfache Art einen Webserver zur Verfügung zu stellen. Aus diesem Grund existieren für Node.js Module, die die Funktionalität der Node.js-Plattform erweitern. In diesem Beispiel benötigen Sie die Funktionalität eines Webservers. Für Node.js existiert ein eigenes Modul zu diesem Zweck und um weitere Aufgaben im Zusammenhang mit HTTP zu lösen. Die Funktionalität aus den verschiedenen Modulen von Node.js steht Ihnen automatisch zur Verfügung. Als Entwickler müssen Sie die Module, die Sie für Ihre Applikation benötigen, vor ihrer Verwendung lediglich laden. Das Laden von Modulen und anderen Dateien geschieht in Node.js über die Funktion `require`. Der Rückgabewert besteht in diesem Fall aus dem `http`-Objekt, das Sie im weiteren Verlauf benötigen, um den Webserver zu betreiben. Dieses Objekt speichern Sie für die weitere Verwendung in einer Variablen.

Mit dem `http`-Modul können Sie neben einem Server auch einen Client zur Abfrage anderer Webserver erstellen. Für Ihre Webserverapplikation benötigen Sie jedoch nur den Server. Dieser wird durch die Methode `createServer` des `http`-Objekts erstellt. Der Rückgabewert dieser Methode ist der HTTP-Server, der Ihnen im weiteren Verlauf als Objekt zur Verfügung steht. Das neu erstellte Serverobjekt hat aktuell

weder Funktionalität, noch wird eine Verbindung nach außen geöffnet. Der nächste Schritt besteht darin, dass Sie eben diese Verbindung in Richtung der Clients öffnen, damit sich diese mit dem Server verbinden und Daten abrufen können. Der Server bietet Ihnen mit der `listen`-Methode eine Möglichkeit, einen Port und eine IP-Adresse anzugeben, über die sich Ihre Nutzer mit dem Server verbinden können. Die Argumente der `listen`-Methode bestehen aus der Angabe der zu verwendenden TCP-Portnummer und der IP-Adresse, auf die sich der Server binden soll. Die Portnummer wird als Ganzzahl angegeben, die zwischen 0 und 65535 liegen muss. Bei der Wahl des Ports für Ihren Webserver müssen Sie zwei Dinge beachten: Der verwendete Port darf nicht bereits durch eine andere Applikation belegt sein, und er sollte nicht im Bereich der Systemports zwischen 0 und 1023 liegen. Verwenden Sie einen Port ab 1024, können Sie Ihr Node.js-Script als normaler Nutzer ausführen, für Ports, die darunterliegen, benötigen Sie Administratorberechtigungen. Tests und Beispiele sollten Sie stets als normaler Nutzer ausführen, da Sie als Administrator über erheblich mehr Berechtigungen verfügen und dadurch das System ernsthaft beschädigen können. Neben der Angabe der Portnummer müssen Sie die IP-Adresse festlegen, auf die der Webserver gebunden werden soll. Ihnen stehen hier sämtliche Adressen des Systems zur Verfügung. Im Falle Ihres Beispiels sollten Sie die Adresse der lokalen Loopback-Schnittstelle `127.0.0.1` als JavaScript-Zeichenkette verwenden. Sie ist auf jedem System verfügbar und öffnet keine Verbindung nach außen, zudem ist keine Netzwerkverbindung erforderlich. Es entstehen also durch Versuche mit Node.js auf Ihrem System keine Sicherheitslücken nach außen, und Sie können sich trotzdem mit dem Browser auf Ihrem System mit dem Webserver verbinden, um die Funktionalität zu testen.

Der Aufruf der Methode `listen` des HTTP-Servers öffnet neben der Festlegung von Adresse und Port die Verbindung und sorgt dafür, dass der Server auf eingehende Anfragen wartet. Die letzte Zeile des Scripts dient lediglich der Information und soll Ihnen auf der Kommandozeile zeigen, dass der Webserver korrekt gestartet wurde, da Node.js ohne diese Zeile nichts ausgibt. Speichern Sie das Script, im Falle dieses Beispiels unter dem Namen *webserver.js*, können Sie den Webserver auf Ihrem System ausführen und das Ergebnis bereits testen. Listing 3.9 zeigt Ihnen, wie das Ergebnis des Tests aussieht.

```
$ node webserver.js
Webserver wird ausgeführt.
```

Listing 3.9 Webserver unter Node.js ausführen

Das Kommando `node` mit der Datei, die den Quellcode des Webserverns enthält, als Option führt dazu, dass ein Node.js-Prozess gestartet wird, der sich auf die angegebene

Kombination aus Adresse und Port verbindet und auf eingehende Verbindungen wartet, die er bedienen kann. Durch die Ausführung des Scripts ist die Kommandozeile blockiert, und Sie können hier keine weiteren Eingaben tätigen. Durch die Architektur von Node.js, die auf dem eventgetriebenen Prinzip aufbaut, entsteht durch das Webserverscript sehr wenig Last, da Node.js nicht blockiert, wenn es nichts zu tun hat. Wollen Sie das Script abbrechen, können Sie dies mit der Tastenkombination `Strg` + `C` erreichen und erhalten dadurch wieder einen Kommandoprompt.

Sie können nun mit Ihrem Browser den Webserver testen, indem Sie in Ihrem Browser in der Adresszeile `http://localhost:8080` eingeben. Abbildung 3.1 zeigt das Ergebnis dieses Tests.

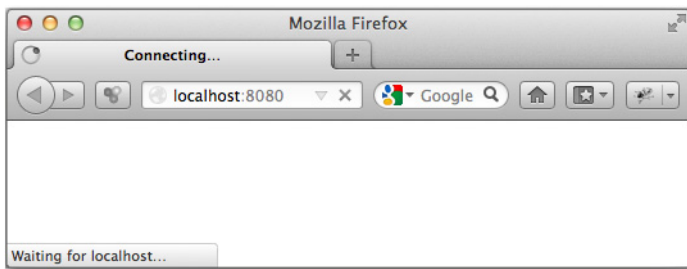


Abbildung 3.1 Test des Webservers im Browser

Es besteht nun das Problem, dass Ihr Webserver zwar auf die korrekte Adresse und den korrekten Port gebunden ist, allerdings keinerlei Logik besitzt, um mit eingehenden Anfragen umzugehen. Das führt in erster Linie dazu, dass Sie beim Test keine Ausgabe erhalten und, wenn Sie Ihr Browserfenster lange genug geöffnet lassen, einen Timeoutfehler erhalten. Aus diesem Grund werden Sie im nächsten Schritt Quellcode einfügen, der dafür sorgt, dass eingehende Anfragen auch sinnvoll bedient werden.

Der Webserver, den Sie hier in Node.js erstellen, unterscheidet sich in einigen Merkmalen gravierend von anderen Umsetzungen in dynamischen Scriptsprachen wie beispielsweise PHP. Hier wird jede Anfrage separat bedient und dabei der notwendige Quellcode eingelesen. Bei Node.js wird der Code der Anwendung einmal eingelesen und verbleibt dann im Arbeitsspeicher. Die Anwendung läuft permanent. Hier kommt ein wichtiger Aspekt von Node.js ins Spiel: die asynchrone Abarbeitung von Aufgaben. Der Webserver reagiert auf Ereignisse, in diesem Fall auf die Anfragen der Clients. Zu diesem Zweck wird eine Funktion definiert, ein Callback, der ausgeführt wird, sobald eine Anfrage eingeht. Dieser Code wird zwar wie auch in anderen Sprachen in einem Block definiert, der Quellcode, der für die Bearbeitung der Anfragen verantwortlich ist, wird jedoch erst ausgeführt, wenn eine solche Anfrage eingeht. In Listing 3.10 sehen Sie den erweiterten Quellcode aus Listing 3.8. Diese Version des Webservers kann dann auch korrekt mit Anfragen umgehen.

```

var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200,
    {'content-type': 'text/plain; charset=utf-8'});
  response.write('Hello ');
  response.end('World\n');
}).listen(8080, '127.0.0.1');
console.log('Webserver wird ausgeführt.');
```

Listing 3.10 Webserver mit Callback

Die einzige Anpassung des Beispiels aus Listing 3.8 findet im Aufruf der Methode `createServer` statt. Hier erhält Node.js den Callback, der angibt, was geschehen soll, wenn eine Anfrage eines Clients den Server erreicht. In diesem einfachen Beispiel geben Sie zuerst lediglich die Zeichenkette `Hello World` auf dem Browser des Clients aus.

Um diese Ausgabe zu erreichen, sehen Sie sich zuerst den Aufbau der Callback-Funktion an. Diese besitzt zwei Parameter, ein Request- und ein Response-Objekt. Wie Sie bereits an den Namen erkennen können, repräsentieren sie jeweils die Anfrage des Clients und die Antwort des Servers. In diesem Beispiel lassen Sie zunächst die Anfrage des Clients außer Acht und konzentrieren sich auf die Antwort des Servers. Im ersten Schritt werden die HTTP-Header-Informationen vorbereitet, die später zum Client zurückgesendet werden. Dies geschieht über die Methode `writeHead`. Das erste Argument dieser Funktion besteht aus einer Zahl, die den HTTP-Statuscode darstellt. Das zweite ist ein Objekt, das den eigentlichen HTTP-Header enthält. Die Schlüsselwerte des Objekts wie im Beispiel der Wert `content-type` sollten laut Konvention in Kleinbuchstaben geschrieben werden. Im Beispiel sehen Sie hier die Angabe des Content-Types, der in diesem Fall auf `text/plain` gesetzt wird, um dem Client anzuzeigen, dass die Antwort des Servers nur Text enthält. Da einige Browser wie beispielsweise Firefox Fehlermeldungen ausgeben, wenn die Zeichencodierung nicht korrekt angegeben wird, wird der Content-Type um die Information `charset=utf-8` erweitert, um dem Browser zu signalisieren, dass der HTTP-Body in UTF-8 codiert ist.

Die für den Benutzer sichtbare Antwort, den HTTP-Body, erstellen Sie mit der Methode `write`. Mit dem Aufruf dieser Methode werden Bruchstücke der Antwort, die sogenannten Chunks, gesendet. Diese Methode können Sie mehrmals hintereinander aufrufen, was dazu führt, dass die einzelnen Teile zusammengefügt werden. Dabei sollten Sie allerdings darauf achten, dass vor dem Aufruf von `write` stets die korrekten HTTP-Header mit `writeHead` gesendet werden. Falls Sie die Methode `writeHead` nicht aufrufen, sendet der Node.js-HTTP-Server implizit einen HTTP-Header mit dem Statuscode 200 und einem Content-Type `text/plain`, sodass auch eine Antwort ohne die explizite Angabe von Header-Informationen gültig ist. Der Aufruf von `write`

sorgt dafür, dass Teile der Antwort zum Client geschickt werden. Der Client erfährt in diesem Fall jedoch nicht, wann der Server mit dem Senden der Antwort fertig ist. Dies müssen Sie als Entwickler durch die Verwendung der `end`-Methode des `Response`-Objekts tun. Als Argument können Sie optional eine Zeichenkette mitgeben. In diesem Fall verhält sich `end` wie `write`, sendet den angegebenen Chunk zum Client und beendet im Anschluss die Antwort.

Die Methode `write` weist zwei weitere Merkmale auf, die erwähnenswert sind. Zum einen können Sie nicht nur Zeichenketten als Argument übergeben, sondern auch Buffer-Objekte. Ein Buffer-Objekt besteht aus Binärdaten, die das Streaming von Daten erheblich erleichtern. Diese Klasse von Objekten kommt vor allem bei der Verwendung von Streams zum Einsatz. Das zweite Merkmal besteht aus der Angabe der Codierung der Zeichenkette. Diese Angabe können Sie über den zweiten Parameter der `write`-Methode machen. Dieser ist optional, und Node.js verwendet, wenn er weggelassen wird, `utf-8` als Standardcodierung. Mögliche andere Werte sind hier `utf16le`, `ascii` oder `hex`. UTF-8 als Codierung ist in diesem Beispiel zulässig, aus diesem Grund ist es auch nicht erforderlich, dass Sie eine Zeichencodierung angeben.

Damit das Beispiel korrekt funktioniert, müssen Sie sicherstellen, dass Sie den Webserver neu starten, damit der angepasste Quellcode von Node.js richtig eingelesen wird. Zu diesem Zweck beenden Sie am besten die eventuell noch laufende Instanz des ersten Beispiels mit der Tastenkombination `[Strg] + [C]` und starten den Webserver durch einen erneuten Aufruf des Kommandos `node` mit dem Namen der Datei Ihres Quellcodes neu. Das Ergebnis des Beispiels sehen Sie in Abbildung 3.2.

In diesem Beispiel haben Sie gesehen, wie es mit nur wenigen Zeilen JavaScript-Code möglich ist, einen funktionsfähigen Webserver zu erstellen, der auf die Anfrage eines Clients mit einer korrekten HTTP-Antwort reagiert. In der Realität haben Sie jedoch in den seltensten Fällen mit Antworten von Webservern in Klartext zu tun. Sie erweitern nun das Beispiel dahingehend, dass der Server mit einer Antwort in HTML reagiert, wie es ein normaler Webserver auch tun würde. Listing 3.11 zeigt Ihnen die Anpassungen, die hierfür durchzuführen sind.

```
var http = require('http');
http.createServer(function (request, response) {

    response.writeHead(200,
        {'content-type': 'text/html; charset=utf-8'});

    var body = '<!DOCTYPE html>' +
        '<html>' +
        '<head>' +
        '<meta charset="utf-8">' +
```

```

        '<title>Node.js Demo</title>' +
        '</head>' +
        '<body>' +
        '  <h1 style="color:green">Hello World</h1>' +
        '</body>' +
        '</html>';

    response.end(body);
  }).listen(8080, '127.0.0.1');
  console.log('Webserver wird ausgeführt.');
```

Listing 3.11 Der Webserver antwortet mit HTML

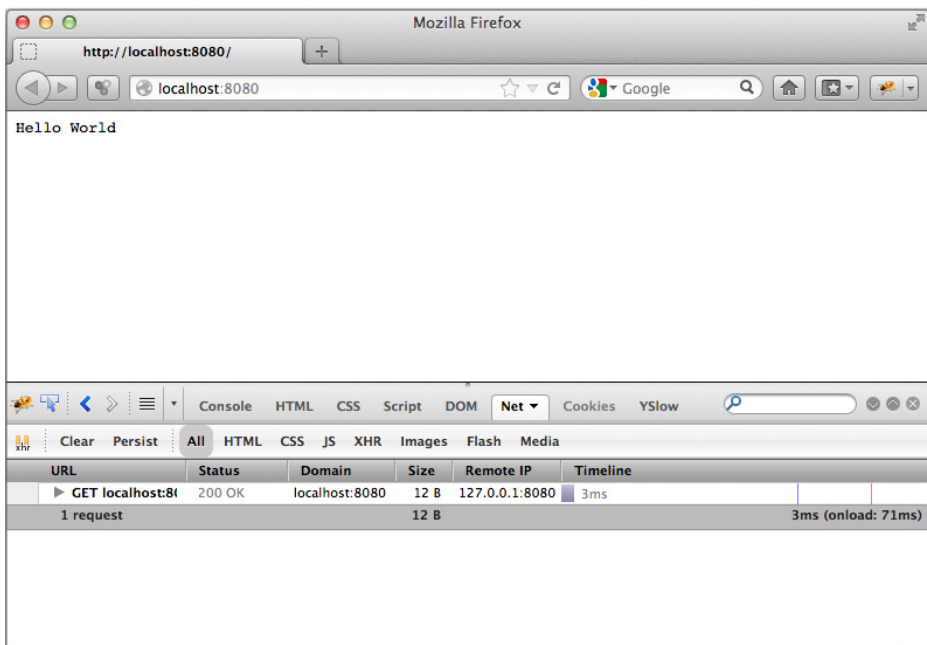


Abbildung 3.2 Antwort des Webservers

Die einzigen Änderungen, die Sie am Quellcode des Beispiels durchführen müssen, ist die Anpassung des Content-Types, der nun nicht mehr `text/plain`, sondern `text/html` lautet, außerdem fällt der Aufruf von `write` weg, und der HTTP-Body wird komplett mithilfe der `end`-Methode verschickt. Der Wert, der an die `end`-Methode übergeben wird, enthält eine HTML-Zeichenkette, die den Aufbau der anzuzeigenden Webseite widerspiegelt. Da diese Zeichenkette recht umfangreich ist, lagern Sie diese am besten in die separate Variable `body` aus und übergeben diese Variable beim Aufruf der `end`-Methode. Haben Sie diese Änderungen durchgeführt, müssen Sie ledig-

lich den Node.js-Prozess, in dem Ihr Webserver ausgeführt wird, neu starten, und die Änderungen werden wirksam. Wenn Sie jetzt die Seite in Ihrem Browser neu laden, sollten Sie ein ähnliches Ergebnis wie in Abbildung 3.3 sehen können.

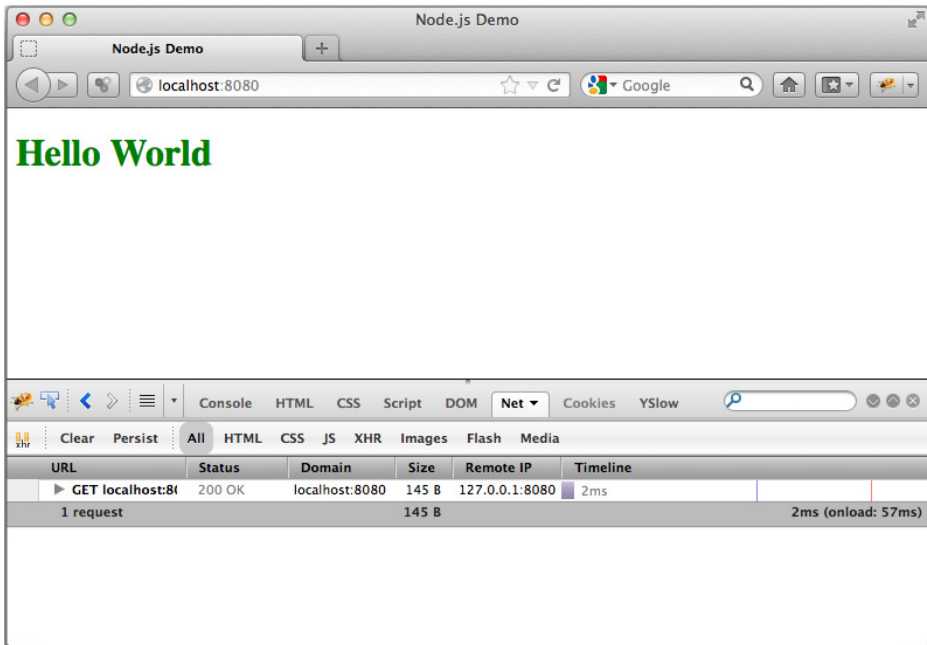


Abbildung 3.3 Auslieferung einer HTML-Seite

Bis jetzt haben Sie sich sehr viel mit dem Response-Objekt, also der Antwort an den Client, beschäftigt. Jetzt ist es an der Zeit, das Request-Objekt, also die eigentliche Anfrage, näher zu betrachten. Dieses Objekt bietet eine Möglichkeit, mit der der Client mit dem Server kommunizieren kann. In klassischen Webapplikationen werden vom Browser aus Informationen über die HTTP-Methoden GET und POST geschickt. Meist geschieht dies über Formulare oder Parameter, die in der URL codiert sind. Sie werden nun das Beispiel abermals erweitern und zeigen eine vom Benutzer in der URL angegebene Zeichenkette auf der ausgelieferten Seite an.

Damit das Beispiel nicht zu unübersichtlich wird, verwenden Sie als Basis den Quellcode aus Listing 3.8 und liefern die Antwort in Klartext statt in HTML aus. Den angepassten Quellcode finden Sie in Listing 3.12 und im Anschluss daran die entsprechenden Erklärungen zu den Änderungen.

```
var http = require('http');
var url = require('url');
http.createServer(function (request, response) {
    response.writeHead(200,
```

```

    {'content-type': 'text/plain; charset=utf-8'}));

    var urlString = url.parse(request.url, true);
    var body = 'Hello ' + urlString.query.name;

    response.end(body);
  }).listen(8080, '127.0.0.1');
  console.log('Webserver wird ausgeführt.');
```

Listing 3.12 Webseiten über Parameter beeinflussen

Die wichtigste Anpassung des Quellcodes besteht darin, dass Sie die URL, die der Client angefragt hat, im Quellcode auslesen und Teile daraus in die Antwort schreiben. Im Request-Objekt liegt die Information, welche URL der Benutzer in seinem Browser angegeben hat, in der Eigenschaft `url` vor. Gehen Sie davon aus, dass der Benutzer die URL `http://localhost:8080/?name=Besucher` in die Adresszeile in seinem Browser eingegeben hat, enthält die Eigenschaft `url` des Request-Objekts den Wert `/?name=Besucher`. Ihr Ziel ist es jetzt, die Zeichenkette `Hallo Besucher` auszugeben. Dazu muss die Zeichenkette, in diesem Fall `Besucher`, aus der `url`-Eigenschaft extrahiert werden. Das können Sie einerseits erreichen, indem Sie die Zeichenkette mit der `split`-Funktion am Gleichheitszeichen zerlegen und das zweite Element des entstandenen Arrays verwenden. Diese Variante funktioniert allerdings nur solange, wie der Benutzer nur einen Parameter in der URL übergibt oder dieser bei mehreren Parametern an erster Stelle steht. Eine bessere Art, mit URLs umzugehen, besteht in der Verwendung des Node.js-Moduls `url`. Dieses stellt Ihnen unter anderem die Möglichkeit zur Verfügung, URLs zu parsen und sie so in ihre einzelnen Bestandteile zu zerlegen. Sie müssen hierfür lediglich das Modul `url` einbinden und auf dem zurückgegebenen Objekt die `parse`-Methode mit der `url`-Eigenschaft des Request-Objekts als Argument aufrufen. Das führt zu folgender Codezeile: `var url = require('url').parse(request.url, true)`. In der Variablen `url` befindet sich dann ein Objekt, das die einzelnen Teile der angefragten URL enthält. Unter anderem findet sich hier eine Eigenschaft `query`, die die einzelnen GET-Parameter enthält. In diesem Fall spielen auch die Anzahl und die Reihenfolge der Parameter in der URL keine Rolle mehr, da Sie über den Namen des Parameters, `name` in diesem Fall, auf den Wert zugreifen können. Mit `url.query.name` können Sie dann also auf die Zeichenkette zugreifen, die der Benutzer in der Adresszeile seines Browsers angegeben hat. Damit haben Sie jetzt sämtliche Komponenten, die Sie benötigen, um Ihr Ziel zu erreichen. Nachdem Sie die Anpassungen am Quellcode vorgenommen und den Webserver neu gestartet haben, können Sie das Ergebnis testen, indem Sie die Seite erneut aufrufen. Abbildung 3.4 zeigt das Ergebnis, das Sie erhalten, wenn Sie die Seite `http://localhost:8080/?name=Benutzer` aufrufen.

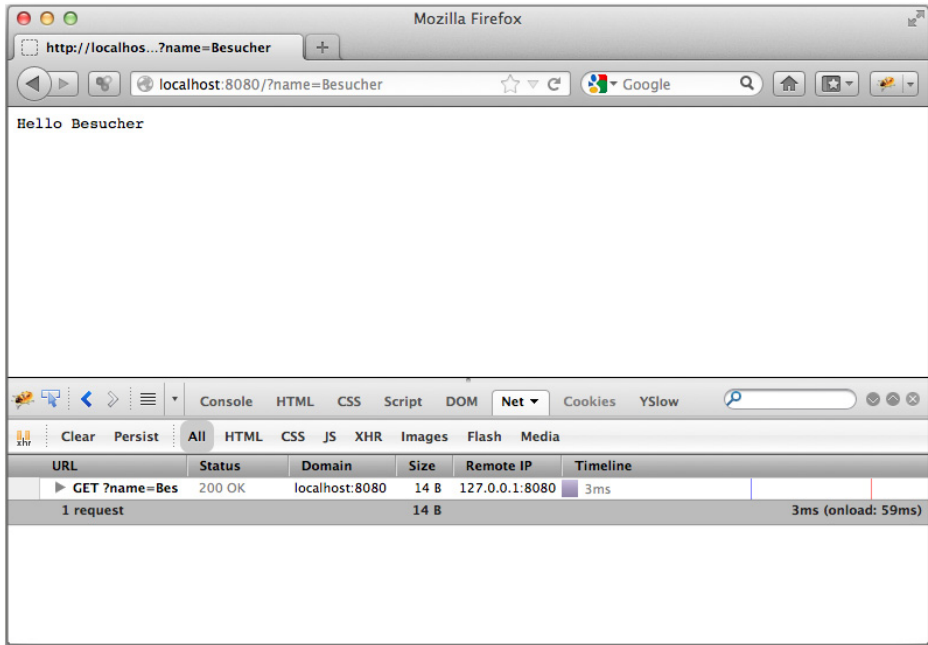


Abbildung 3.4 Die erste dynamische Webseite unter Node.js

3.3 Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie Sie mit wenigen Zeilen Quellcode bereits einfache Webapplikationen mit Node.js erstellen und dabei sogar auf Eingaben des Nutzers reagieren können. In den nächsten Kapiteln erhalten Sie einen tieferen Einblick in den Aufbau von Node.js, das Modulkonzept und den Node.js-Paketmanager.

Kapitel 4

Anpassung und Erweiterung

*Nichts auf der Welt ist so gerecht verteilt wie der Verstand.
Denn jedermann ist überzeugt, dass er genug davon habe.*
– René Descartes

Nachdem Sie im vorangegangenen Kapitel einen kleinen Ausschnitt der Möglichkeiten von Node.js in Aktion erleben konnten, geht dieses Kapitel tiefer ins Detail und zeigt Ihnen, wie Sie die Node.js-Plattform verwenden können, um eigene Applikationen zu erstellen. Sie erfahren außerdem mehr über die Hintergründe des Node.js-Modulsystems und wie der Funktionsumfang von Node.js durch externe Pakete erweitert wird.

4.1 Node.js-Module

Im vorangegangenen Kapitel haben Sie bereits zwei verschiedene Module verwendet, das `http`-Modul und das `URL`-Modul. Diese Module stellen zusätzliche und häufig gebrauchte Funktionalitäten oder Schnittstellen zu den Kernbibliotheken von Node.js zur Verfügung. Die nachfolgenden Abschnitte beschäftigen sich eingehend mit dem Modulsystem von Node.js und den Möglichkeiten, die daraus entstehen.

4.1.1 Modularer Ansatz

Der Kern der Node.js-Plattform besteht aus einer Ansammlung verschiedener Bibliotheken, die getrennt voneinander gepflegt und schließlich in Node.js zusammengefügt werden. Einen ähnlichen Ansatz verfolgt Node.js selbst auch in seinem Modulsystem. Die einzelnen Module werden größtenteils unabhängig voneinander entwickelt, stehen Ihnen jedoch in der Plattform zur Verfügung und können dort beliebig miteinander kombiniert werden. Die Module sind ein fester Bestandteil der Node.js-Plattform und als solcher direkt in die Binärdatei kompiliert. Sie stehen Ihnen also zu jeder Zeit zur Verfügung. Sie haben aber auch keine Möglichkeit, bestimmte Module beim Kompiliervorgang auszuschließen. Entwickelt werden die Node.js-Module in JavaScript selbst. Das bedeutet, dass der Kern der Plattform zwar in C vorliegt, die übrigen Komponenten allerdings in der Sprache der Plattform, also in JavaScript, entwickelt werden. In der Architektur von Node.js, wie sie in Abbildung 4.1

dargestellt ist, bewegen Sie sich in diesem Kapitel im oberen Bereich, also in der Standard Library von Node.js. Das bedeutet, dass die Node.js-Module im Quellcode vorliegen und Sie diese auch bei Bedarf für Ihre Zwecke anpassen können und dazu nichts weiter benötigen als ein solides Wissen über JavaScript. Die Dateien der Module liegen im *lib*-Verzeichnis im Quellcodepaket von Node.js.

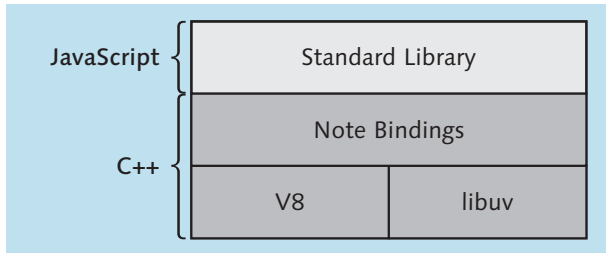


Abbildung 4.1 Die Architektur von Node.js

Wie Sie bereits in Kapitel 3, »Ein erstes Beispiel«, gesehen haben, lassen sich die Node.js-Module sehr einfach mithilfe der Funktion `require` einbinden. Die Funktionalität wird über den Rückgabewert zur Verfügung gestellt. Dieser ist in den meisten Fällen ein Objekt, dessen Methoden direkt verwendet werden können. Listing 4.1 zeigt Ihnen die Verwendung des Dateisystemmoduls zum Auslesen der Datei `/tmp/nodejs.txt`.

```

var fs = require('fs');
fs.readFile("/tmp/nodejs.txt", function (err, content) {
    if (err) {
        throw err;
    }
    console.log(content.toString());
});

```

Listing 4.1 Verwendung von Node.js Modulen

Im Beispiel in Listing 4.1 laden Sie in der ersten Zeile das Dateisystemmodul über das Kommando `require('fs')`. Der Rückgabewert dieser Funktion ist ein Objekt, das Ihnen eine Schnittstelle für verschiedenste Dateisystemoperationen zur Verfügung stellt. Unter anderem existiert die Methode `readFile`, die Sie in diesem Fall verwenden, um eine Datei auszulesen und ihren Inhalt auf der Kommandozeile auszugeben.

Woher aber sollen Sie wissen, welche Module verfügbar sind, wozu sie dienen und unter welchem Namen Sie die Module laden können? Die Module von Node.js sind sehr gut in der API-Dokumentation der Node.js-Plattform dokumentiert. Diese erreichen Sie unter <http://nodejs.org/api/>. Eine Alternative hierzu steht Ihnen zur Verfügung, wenn Sie den Quellcode von Node.js heruntergeladen haben. Im Verzeichnis

doc existiert ein Unterverzeichnis *api*, in dem die gesamte API-Dokumentation von Node.js offline verfügbar ist. Sie können diese Dokumentation über die *index.html*-Datei in diesem Verzeichnis anzeigen lassen. Die Inhalte entsprechen denen auf der Node.js-Webseite, sind allerdings im Quellcodepaket versionsspezifisch. Das bedeutet, dass Sie auf diese Dokumentation zurückgreifen sollten, falls Sie für eine ältere Version von Node.js entwickeln oder Informationen über die API einer älteren Node.js benötigen.

4.1.2 Stabilitätsindex

Eine häufig gestellte Frage ist, ob man Node.js in Unternehmen einsetzen kann. Diese Frage lässt sich nicht klar und eindeutig beantworten. Durch die Entwicklung der Plattform und die Versuche, Node.js stabiler und performanter zu gestalten, geht die Tendenz dennoch in die Richtung die Frage mit einem Ja zu beantworten. Ob Sie eine Technologie einsetzen sollten oder nicht, hängt von einer Vielzahl von Faktoren ab. Darunter befinden sich unter anderem die Art der Problemstellung, die Umgebung, in der die Lösung eingesetzt werden soll, und nicht zuletzt der Entwicklungsstand der verwendeten Technologien. Einen ähnlichen Sachverhalt wie bei der Frage nach der Einsatztauglichkeit der gesamten Node.js-Plattform ergibt sich auch bei den Modulen von Node.js. Da die Module unabhängig voneinander entwickelt werden, weisen sie jeweils unterschiedliche Entwicklungsstände auf, was sich auch in der Stabilität der Module niederschlägt. Module, die über mehrere Jahre hinweg entwickelt werden und sich bereits vielfach im praktischen Einsatz bewährt haben, sind natürlich stabiler und zuverlässiger als Module, die erst in einer der letzten Versionen Einzug in die Plattform gehalten haben. Die Entwickler von Node.js sind sich dieser Problemstellung bewusst und haben hierfür den sogenannten Stabilitätsindex entwickelt. Das ist eine Zahl, anhand derer Sie bestimmen können, ob ein Modul so ausgereift ist, dass Sie es ohne größere Probleme in Ihrer Applikation einsetzen können, oder ob Sie mit dem Einsatz lieber noch warten und stattdessen auf andere Alternativen ausweichen sollten. Tabelle 4.1 stellt Ihnen eine Liste der bestehenden Stabilitätsindex-Zahlen zur Verfügung.

Index	Name	Übersetzung
0	deprecated	veraltet
1	experimental	experimentell
2	unstable	unstabil
3	stable	stabil

Tabelle 4.1 Node.js-Stabilitätsindex

Index	Name	Übersetzung
4	API Frozen	API eingefroren
5	locked	gesperrt

Tabelle 4.1 Node.js-Stabilitätsindex (Forts.)

Der Stabilitätsindex »veraltet« bedeutet, dass das Feature, das mit diesem Stabilitätsindex markiert ist, nicht weiter verwendet werden sollte. Ein Beispiel für ein veraltetes Feature ist die `pump`-Methode des `util`-Moduls. In der Dokumentation von Node.js wird bei jeder als »veraltet« markierten Funktionalität eine Alternative genannt, die Sie stattdessen verwenden sollten. Experimentelle Module auf der anderen Seite bedeuten für Sie als Entwickler, dass Sie das Feature zwar verwenden können, die Funktionalität allerdings umfangreich testen sollten. Im weiteren Entwicklungsprozess von Node.js kann nicht garantiert werden, dass sich keine Änderungen der Schnittstellen ergeben. Das heißt, dass Sie bei Versionsupdates von Node.js Ihre Aufmerksamkeit auf die Stellen richten sollten, an denen Sie derartige Features nutzen. Experimentelle Features sind beispielsweise das `Domain`- oder das `Cluster`-Modul.

Node.js-Module, die als »unstabil« markiert sind, können in Ihrer Applikation eingesetzt werden, ohne dass Sie sich Sorgen machen müssen, dass diese in einer späteren Version nicht mehr zur Verfügung stehen könnten. Sie sollten bei der Verwendung dieser Module bei Updates dennoch Vorsicht walten lassen und die entsprechenden Stellen ausgiebig testen. Beispiele für instabile Module sind unter anderem `Stream` oder `Crypto`.

Die meisten der Kernmodule von Node.js sind mit dem Stabilitätsindex 3, also »stabil«, markiert. Dieser Index bedeutet, dass Sie diese Module in Anwendungen produktiv einsetzen können. Die Schnittstellen bleiben auch nach Updates stabil. Sie können sich darauf verlassen, dass die Stelle in Ihrer Applikation auch nach einem Update weiterhin funktionieren wird. Stabile Module sind beispielsweise `File System`, `Net` oder `HTTP`.

Der Stabilitätsindex »API eingefroren« sagt aus, dass es unwahrscheinlich ist, dass sich die Benutzerschnittstelle des Moduls beziehungsweise die Signatur der verschiedenen Methoden des Moduls in Zukunft ändern werden. Beispiele für derartig markierte Module sind `OS` oder `Events`.

Der letzte Stabilitätsindex, »gesperrt«, bedeutet, dass die Module nicht mehr verändert werden. Beispiele hierfür sind `Timers` oder `Util`.

Der Stabilitätsindex zeigt Ihnen als Entwickler, ob Sie mit Änderungen in den jeweiligen Modulen rechnen müssen. Änderungen bedeuten in diesem Fall vor allem, dass sich die Signatur der Methoden ändern kann, also vor allem die Anzahl und die

Benennung der Parameter. Aber auch der Rückgabewert kann angepasst werden. Nutzen Sie ein Modul, das derartigen Veränderungen unterworfen ist, ergeben sich zwangsläufig Probleme in Ihrer Applikation, die Sie durch Anpassungen beheben müssen, um die Anwendung weiterhin fehlerfrei nutzen zu können.

4.1.3 Verfügbare Module

Node.js wird mit einer Reihe von Basismodulen ausgeliefert. In diesem Abschnitt erfahren Sie mehr über die einzelnen Module und deren Einsatzgebiete. Tabelle 4.2 gibt Ihnen einen ersten Überblick über die vorhandenen Module und wo sie jeweils zum Einsatz kommen.

Modulname	Beschreibung
assert	integriertes Testframework
buffer	Modul für den Umgang mit binären Rohdaten
child_process	Verwaltung von Kindprozessen
cluster	Prozessverwaltung für Mehrkernprozessoren
console	console-Funktionalität in JavaScript
crypto	sichere Verbindungen und Verschlüsselung
debugger	Mittel zur Laufzeitanalyse von Applikationen
dgram	Kommunikation über UDP
dns	Namensauflösung unter Node.js
domain	Sammlung von Exceptions und Fehlern einer Gruppe von Operationen
events	Basismodul für Objekte, die Events erzeugen
fs	Dateisystemoperationen
http	HTTP-Client und -Server
https	HTTPS-Client und -Server
module	Modulloader von Node.js
net	Client- und Serverkomponenten für Netzwerkstreams
os	Betriebssysteminformationen auslesen

Tabelle 4.2 Node.js-Kernmodule

Modulname	Beschreibung
path	Umgang mit Pfadinformationen
punycode	Codierung und Decodierung von Punycode-Zeichenketten
querystring	Erstellung und Parsing von URL Query Strings
readline	Einlesen von Informationen über einen Stream wie die Standardeingabe
repl	interaktive Node.js Shell
stream	schreib- und lesbare Datenstreams
string_decoder	Umwandlung von Buffer-Objekten in Zeichenketten
timers	zeitabhängige Funktionen
tls	verschlüsselte Kommunikation über Datenstreams
tty	Zwischenschicht zum Ansprechen von Standardein- und -ausgabe
url	Funktionalität zum Umgang mit URLs
util	Hilfsfunktionen
vm	Ausführungsumgebung für JavaScript-Code
zlib	Möglichkeit zum Komprimieren und Dekomprimieren von Daten

Tabelle 4.2 Node.js-Kernmodule (Forts.)

Nachfolgend lernen Sie die einzelnen Module näher kennen. Kleine Codebeispiele sollen Ihnen beim Verständnis der konkreten Anwendung helfen.

Assertion Testing

Für JavaScript existiert eine Vielzahl von Frameworks, mit deren Hilfe Sie Unittests für Ihre Applikation erstellen können. Fast jedes JavaScript-Framework bringt seine eigene Interpretation eines Testframeworks mit. Diese Vielfalt existiert jedoch nicht nur clientseitig im Webbrowser, auch für Node.js gibt es mehrere Ansätze, mit denen Unittests erstellt werden können. Die einfachste Variante besteht jedoch in der Verwendung des `assert`-Moduls von Node.js, mit dem Sie vollwertige Unittests erstellen können, wie Sie in Listing 4.2 anhand eines konkreten Beispiels sehen können.

```

var assert = require('assert');
var greaterThan = function(a, b) {
    return a > b;
}
assert.ok(greaterThan(2,3), '2 is not greater than 3');

```

Listing 4.2 Testing mit Node.js

Im Beispiel definieren Sie eine Funktion, die den Wert `true` zurückgibt, wenn das erste Argument größer ist als das zweite, ansonsten ist das Ergebnis `false`. Mit der Methode `ok` des `assert`-Objekts kann dieser Rückgabewert geprüft werden, indem der Funktionsaufruf beziehungsweise der Rückgabewert als erstes Argument übergeben wird. Im Falle des Beispiels erwarten Sie, dass der Test fehlschlägt, da 2 nicht größer als 3 ist. Schlägt ein Test fehl, wird eine Exception vom Typ `AssertionError` geworfen und die Meldung, die Sie der `ok`-Methode als zweites Argument übergeben haben, angezeigt.

Buffer

Im Internet sind Octet Streams ein etabliertes Mittel zur Übertragung von Daten. Dies spiegelt sich bereits im MIME-Type `application/octet-stream` wider, der verwendet wird, wenn Dateien übermittelt werden, deren Format unbekannt sind. In Node.js werden Octet-Streams vor allem im Bereich der Netzkommunikation und bei Dateisystemoperationen verwendet. Die `Buffer`-Klasse steht in Node.js als globale Klasse zur Verfügung und muss nicht erst über `require` eingebunden werden. In Listing 4.3 sehen Sie die Repräsentation eines `Buffer`-Objekts.

```
<Buffer 48 61 6c 6c 6f 20 57 65 6c 74 0a>
```

Listing 4.3 Buffer-Objekt in Node.js

Diese Art der Anzeige können Sie generieren, indem Sie beispielsweise das Resultat einer Dateisystemoperation auf der Konsole ausgeben, wie Sie es in Listing 4.4 sehen können.

```

var fs = require('fs');
fs.readFile("/tmp/nodejs.txt", function (err, content) {
    if(Buffer.isBuffer(content)) {
        console.log(content);
    }
});

```

Listing 4.4 Buffer-Objekt bei Dateisystemoperationen

In diesem Beispiel sehen Sie mit der Methode `isBuffer` ein weiteres Feature der `Buffer`-Klasse. Durch sie kann geprüft werden, ob es sich bei einem bestimmten Objekt tatsächlich um ein `Buffer`-Objekt handelt.

Child Process

Ein zentrales Merkmal von Node.js ist der Single-Threaded-Ansatz, der dafür verantwortlich ist, dass der gesamte Code der Applikation in einem Node.js-Thread ausgeführt wird. In Node.js haben Sie allerdings die Möglichkeit, unabhängige Kindprozesse zu starten. Das können einerseits normale Kommandos sein, deren Ausgabe später zur Verfügung steht, und andererseits separate Node.js-Prozesse, durch die rechenintensive Operationen ausgelagert werden können. Auf Clientseite existiert in HTML5 mit dem `Webworker`-Feature eine Möglichkeit, rechenintensive Operationen auszulagern. In Kapitel 7, »Asynchrone Programmierung«, erfahren Sie im Detail, wann und wie Sie das `Child Process`-Modul in Ihrer Applikation einsetzen können.

Cluster

Das `Cluster`-Modul ist ein noch recht junges Node.js-Modul. Es wurde in der Version 0.6 von Node.js integriert und ist seit dieser Zeit vielen Änderungen und Optimierungen unterworfen. Diese Entwicklung spiegelt auch der Stabilitätsindex »experimentell« dieses Moduls wider. Das `Cluster`-Modul trägt der Tatsache Rechnung, dass moderne Computer nicht nur einen Prozessorkern besitzen, sondern mehrere. Diese Infrastruktur können Sie jedoch nicht ohne Weiteres mit Node.js ausnutzen, da der Code der Applikation bekanntlich in einem einzigen Thread abläuft. Das `Cluster`-Modul sorgt dafür, dass eine Reihe von Workerprozessen gestartet wird, auf die die Arbeitslast der Applikation verteilt wird. Dieses Modul baut direkt auf dem `Child Process`-Modul auf, um Kindprozesse zu starten und ihnen die Kommunikation untereinander zu ermöglichen. Das Loadbalancing der Prozesse wird durch das Betriebssystem selbst übernommen, sodass Sie sich hierüber keine Gedanken zu machen brauchen. Die Verwendung dieses Moduls beeinflusst allerdings die Architektur Ihrer Anwendung, da Sie entscheiden müssen, wann welche Bestandteile der Applikation aus dem Hauptprozess ausgelagert werden sollen. Außerdem müssen Sie selbst für die Verwaltung der Kindprozesse sorgen, dies wird nicht durch Node.js übernommen. Für einen detaillierten Einblick in dieses Modul sollten Sie sich Kapitel 7, »Asynchrone Programmierung«, ansehen, wo Sie anhand eines Beispiels mehr über den Einsatz und die Funktionsweise des `Cluster`-Moduls erfahren.

Console

Teile der Funktionalität dieses Moduls haben Sie im Verlauf dieses Buches bereits häufiger verwendet. Das `Console`-Modul stellt das globale `console`-Objekt zur Verfü-

gung, mit dessen Hilfe Sie beispielsweise Informationen ausgeben können. Zur Ausgabe stehen Ihnen sowohl die Standardausgabe über `console.log` oder `console.info` als auch die Standardfehlerausgabe über `console.error` oder `console.warn` zur Verfügung. Die API der Konsolenbefehle entspricht derjenigen, die auch in den Standardwebbrowsern zum Einsatz kommt.

Während der Entwicklung recht hilfreich, um Durchlaufzeiten zu messen, sind die Funktionen `console.time` und `console.timeEnd`, die Ihnen eine Art Stoppuhr zur Verfügung stellen, mit der Sie eine bestimmte Zeitspanne innerhalb einer Applikation messen können.

```
console.time('example');
var result = 0;
for (var i = 1; i <= 1000; i++) {
    result += i;
}
console.log('Sum: ' + result);
console.timeEnd('example');
```

Listing 4.5 Verwendung des console-Moduls

In Listing 4.5 wird die Zeit gemessen, die benötigt wird, um die Zahlen von 1 bis 1000 aufzusummieren und das Ergebnis auf der Standardausgabe auszugeben. Im Anschluss daran wird die dafür benötigte Zeitspanne mit dem Label `example` auf der Standardausgabe angezeigt.

Crypto

In Webapplikationen haben Sie es als Entwickler auch immer wieder mit vertraulichen Daten zu tun, für deren sichere Übermittlung und Speicherung Sie verantwortlich sind. Auch unter Node.js ist ein sicherer Umgang mit Daten möglich, die nicht allen Personen zugänglich gemacht werden sollen, die Zugriff auf die Applikation beziehungsweise den Server haben. Basierend auf OpenSSL, stellt das `Crypto`-Modul Schnittstellen zur Verfügung, mit denen Sie Daten verschlüsseln und Signaturen oder Hashes erstellen können. Im Beispiel in Listing 4.6 sehen Sie, wie Sie auf einfache Art und Weise den MD5-Hash einer Zeichenkette erstellen können.

```
var crypto = require('crypto');
var md5 = crypto.createHash('md5');
md5.update('Hallo Welt');
console.log(md5.digest('hex'));
```

Listing 4.6 Erstellung eines MD5-Hashes

In Listing 4.6 erstellen Sie mithilfe des `Crypto`-Moduls und der `createHash`-Methode ein neues Hash-Objekt. Dieser Methode übergeben Sie als Argument den gewünschten Algorithmus. Mögliche Werte sind hier unter anderem `sha1` und `md5`. Letzteres verwenden Sie in diesem Beispiel, um einen Hash über die Zeichenkette `Hallo Welt` zu bilden. Im nächsten Schritt werden die Daten über die `update`-Methode in das Hash-Objekt geschrieben. Die `digest`-Methode sorgt schließlich für die Berechnung des Hashwerts und gibt diesen zurück. Rufen Sie diese Methode ohne Argumente auf, wird der Hash binärcodiert. Es gibt allerdings die Möglichkeit, die Art der Ausgabe-codierung durch Argumente zu beeinflussen. Im Beispiel haben Sie den Wert `hex` verwendet, um einen hexadezimalen Hash zu erzeugen. Die Ausgabe entspricht dem Wert, den Sie standardmäßig durch das Kommandozeilenwerkzeug `md5` erhalten. Mehr zum Thema Sicherheit erfahren Sie in Kapitel 11, »Sicherheitsaspekte«.

Debugger

Node.js verfügt über eine umfangreiche Funktionalität, um Applikationen zu debuggen. Die Besonderheit dieses Debuggers liegt darin, dass Sie den Debugger sowohl lokal auf dem Rechner verwenden können, auf dem Sie Ihre Node.js-Anwendung ausführen, als auch über eine TCP-Verbindung von einem entfernten Rechner. Durch diese Funktionalität wird es möglich, den Debugger über die Kommandozeile durch eine Reihe von Befehlen zu bedienen. Eine weitere Variante, wie Sie den Debugger nutzen können, ist, ihn über Ihre Entwicklungsumgebung zu steuern. Für den Node.js Debugger existieren Plugins für Eclipse, Webstorm und weitere Entwicklungsumgebungen. In Kapitel 9, »Qualitätssicherung«, lernen Sie den Node.js Debugger und seine Funktionsweise im Detail kennen.

Datagram

Neben der Kommunikation über TCP Sockets oder Socket-Dateien, die vom `Net`-Modul übernommen wird, kann die Kommunikation auch über das UDP-Protokoll erfolgen. Der Unterschied ist, dass UDP ein sehr leichtgewichtiges, nicht verbindungsorientiertes Protokoll ist. Mit diesem Protokoll können Sie nicht sicherstellen, dass es auf der Verbindung zwischen Server und Client zu keinem Datenverlust kommt. Eine UDP-Verbindung wird vor allem für unkritische Echtzeitdaten verwendet, bei denen der Verlust einiger Datenpakete keine negativen Auswirkungen auf die Applikation hat. Diese Art der Kommunikation wird in Kapitel 6 im Detail erklärt.

DNS

Bei der Entwicklung von Webapplikationen entsteht oft die Notwendigkeit, mit anderen Systemen zu kommunizieren. Die Kommunikation auf Protokollebene findet mit einer eindeutigen Adressierung über eine Kombination von IP-Adresse und

Portnummer statt. Innerhalb einer Applikation ist es in den meisten Fällen jedoch sinnvoller, nicht die Adressen, sondern die Hostnamen zu verwenden, da sich dynamisch zugewiesene Adressen häufig ändern können, der Hostname in den meisten Fällen jedoch derselbe bleibt. Das Domain Name System, kurz DNS, übernimmt die Auflösung der Hostnamen in IP-Adressen. Node.js verfügt über ein Modul, das darauf spezialisiert ist, die Namensauflösung zu übernehmen. Dieses Modul übernimmt nicht nur die Auflösung von Hostnamen, sondern kann verschiedene Typen von Datensätzen einer Domain, wie beispielsweise die MX-Records für die Mailserveradresse, auslesen. In Listing 4.7 sehen Sie, wie Sie die Adresse einer Domain mithilfe von Node.js auflösen können.

```
var dns = require('dns');
dns.resolve('www.google.de', function (err, addrs) {
  addrs.forEach(function (record) {
    console.log('www.google.de is ' + record);
  });
});
```

Listing 4.7 DNS-Auflösung mit Node.js

Sämtliche Funktionalitäten des DNS-Moduls funktionieren asynchron. Das bedeutet, dass Sie jeder Methode eine Callback-Funktion als Argument übergeben, die aufgerufen wird, sobald das Ergebnis der Abfrage des DNS-Servers vorliegt. Alle Methoden, bis auf `dns.lookup`, basieren auf der C-Ares-Bibliothek. Diese ist vor allem bei einer größeren Anzahl von DNS-Auflösungen erheblich schneller als das systeminterne `getaddrinfo`. Im Beispiel in Listing 4.7 soll der Name `www.google.de` mit der `resolve`-Methode aufgelöst werden. Liegt die Antwort des DNS-Servers vor, wird die Callback-Funktion mit zwei Argumenten aufgerufen. Das erste Argument ist, wie bei Node.js üblich, ein Objekt, das verwendet wird, falls bei der Auflösung ein Fehler aufgetreten ist. Das zweite Argument ist ein Array von IP-Adressen, die der DNS-Server zum angefragten Namen gefunden hat. Die `resolve`-Methode erlaubt ein optionales zweites Argument vor der Callback-Funktion, über das Sie angeben können, welche Typen von Records Sie erhalten möchten. Der Standardwert ist hier `A` für IPv4-Adressen.

Domain

Das `Domain`-Modul dient der Zusammenfassung von Fehlern, die bei der Verwendung von `EventEmitttern` auftreten. Dieses Feature ist erst seit der Version 0.8 von Node.js Bestandteil der Plattform und wird aktuell noch als experimentelles Feature geführt. Die Funktionalität dieses Moduls hilft dabei, sämtliche Fehler innerhalb eines bestimmten Abschnitts in einer Applikation abzufangen und damit zu verhindern, dass die Anwendung beendet wird. Listing 4.8 verdeutlicht diesen Sachverhalt.

```

var fs = require('fs'),
    domain = require('domain'),
    emitter = new (require('events').EventEmitter)();
var fsDomain = domain.create();
fsDomain.on('error', function(err) {
    console.log('An error occurred in Domain');
    console.log(err);
});
fsDomain.run(function () {
    fs.readFile('/tmp/doesNotExist', function (err, data) {
        if (err) {
            throw err;
        }
        console.log(data);
    });
});
fsDomain.run(function() {
    emitter.emit('error', {message: 'EventEmitter error Event emitted'});
});

```

Listing 4.8 Das domain-Modul in Aktion

Beim Einsatz des Domain-Moduls wird zuerst eine Domain in Form eines Objekts erzeugt. Dieses Objekt bildet später dann den Kontext für die Ausführung von Quellcode, von dem potenzielle Fehler abgefangen werden sollen. Die Erzeugung dieser Domain geschieht über die `create`-Methode des Domain-Moduls. Der Rückgabewert, das Domain-Objekt, stellt dabei quasi den Ausführungskontext dar. Im Domain-Objekt können Sie nun mithilfe der `on`-Methode Callback-Funktionen registrieren, die ausgeführt werden, sobald ein Fehler auftritt. Solche Fehler können sowohl `error`-Events eines `EventEmitter`s sein als auch `Exceptions`. Voraussetzung für das Funktionieren der Domain ist, dass weder `error`-Events noch `Exceptions` gefangen und behandelt werden, die im Normalfall zum Abbruch der Anwendung führen würden. Schließlich müssen Sie den Codeblock, der ausgeführt werden soll, in einer Funktion kapseln und diese als Argument an die `run`-Methode der Domain übergeben, die Sie eben erstellt haben. Dieser Codeblock wird durch den Aufruf der `run`-Methode evaluiert und ausgeführt.

Events

Node.js basiert auf einer Architektur, die eine eventgetriebene Entwicklung unterstützt. Dafür sorgt schon allein die Verwendung des Eventloops, der dafür sorgt, dass I/O-Operationen aus dem zentralen Applikations-Thread ausgelagert werden. Das Events-Modul liefert eine Klasse `EventEmitter`, die als Basis für sämtliche Objekte die-

nen sollte, die Events auslösen können. Der Server des `http`-Moduls erbt beispielsweise von der Klasse `EventEmitter`, um den korrekten Umgang mit Events sicherstellen zu können. So wird bei einer eingehenden Verbindung das Event `request` ausgelöst, auf das die `RequestListener`-Funktion gebunden ist, die bei der Erstellung des Servers definiert wurde. Ein sehr einfaches Beispiel, das die Verwendung der `EventEmitter`-Klasse demonstriert, sehen Sie in Listing 4.9.

```
var myEvent = new (require('events').EventEmitter)();
myEvent.on('event', function(data) {
    console.log(data);
});
myEvent.emit('event', 'Hello World');
```

Listing 4.9 Der Node.js `EventEmitter`

Im Beispiel sehen Sie, dass ein neues Objekt der Klasse `EventEmitter` erzeugt wird und auf dieses ein Listener für das Event mit dem Namen `event` gebunden wird. Mit der `emit`-Methode wird das Event ausgelöst, was dazu führt, dass die zuvor definierte Callback-Funktion ausgeführt wird und die Eventdaten, in diesem Fall die Zeichenkette `Hello World`, ausgegeben werden. Später in diesem Kapitel werden Sie noch mehr über den `EventEmitter` und seine Features erfahren.

Filesystem

Ein sehr wichtiges Modul der Node.js-Plattform stellt das `File System`-Modul dar. Es regelt sämtliche Zugriffe, sowohl lesende als auch schreibende, auf das Dateisystem. Den Einsatz dieses Moduls haben Sie bereits zu Beginn dieses Kapitels in Listing 4.1 kurz gesehen. Im Gegensatz zum `DNS`-Modul, bei dem sämtliche Methoden nur asynchron, also mit Callback funktionieren, existiert im `File System`-Modul für jede asynchrone Operation ein synchrones Gegenstück. Im Falle einer synchronen Operation ist die Methode lediglich ein Aufruf mit einem bestimmten Rückgabewert. Hier müssen Sie keine Callback-Funktion definieren, der Kontext der aktuellen Funktion wird nicht verlassen, und das Script läuft synchron, Anweisung für Anweisung, durch. Zu beachten ist dabei allerdings, dass die Dateisystemoperationen in diesem Fall die Ausführung des Scripts blockieren. Das bedeutet, dass die nächste Anweisung erst ausgeführt werden kann, wenn die Dateisystemoperation beendet ist. Weitere Details zur Verwendung des `File System`-Moduls erhalten Sie in Kapitel 5, »Arbeiten mit Dateien«, das sich ausschließlich mit diesem Thema beschäftigt.

HTTP

Der klassische Webserver in Node.js wird vom `http`-Modul zur Verfügung gestellt. Dieses Modul bietet jedoch nicht nur die Serverkomponente, sondern Sie können

damit auch einen HTTP-Client erstellen und Webserver abfragen. Im einleitenden Beispiel in Kapitel 3, »Ein erstes Beispiel«, wurde bereits die Serverkomponente detailliert betrachtet. Kapitel 6, »Kommunikation«, beschäftigt sich eingehend mit den Kommunikationsmöglichkeiten von Node.js und hier unter anderem auch mit den Funktionalitäten aus dem `http`-Modul.

HTTPS

In eine ähnliche Richtung wie das `Crypto`-Modul geht auch das `https`-Modul. Es sorgt dafür, dass vertrauliche Daten verschlüsselt zwischen Client und Server hin- und herbewegt werden. Die sichere Implementierung von HTTP in Node.js ist ein separates Modul, das von Teilen des TLS-Moduls ableitet und sehr ähnliche Schnittstellen wie das `http`-Modul aufweist. Das `https`-Modul wird ebenfalls in Kapitel 6, »Kommunikation«, näher betrachtet und seine Einsatzmöglichkeiten anhand konkreter Beispiele verdeutlicht.

Module

Auch das Node.js-Modulsystem selbst ist als Modul umgesetzt. Es dient dazu, Module zu definieren und diese zu laden. Außerdem stehen Ihnen einige zusätzliche Hilfsfunktionen zur Verfügung. Die Modulfunktionalität besitzt eine große Bedeutung in Node.js-Applikationen, da sie es erst ermöglicht, den Quellcode einer Applikation sinnvoll in kleinere Bestandteile zu unterteilen und eine Applikation so wartbarer zu gestalten. Im Verlauf dieses Buches haben Sie das Modulsystem schon etliche Male verwendet. Jeder Aufruf der `require`-Funktion stellt eine Verwendung des `Module`-Moduls dar. Im Laufe dieses Kapitels erfahren Sie noch mehr über dieses Modul, seine Schnittstellen und wie Sie dieses Feature für Ihre eigene Applikation nutzen können.

Net

Kommunikation ist ein wichtiges Thema in modernen Webapplikationen. Diese bestehen häufig aus mehreren Systemen, die Informationen miteinander austauschen müssen. Die Palette an Arten und Protokollen der Kommunikation ist breit gefächert. Mit dem `Net`-Modul erhalten Sie eine Reihe von Schnittstellen, mit denen Sie sowohl über TCP-Verbindungen als auch über UNIX-Sockets kommunizieren können, und das in der Rolle des Servers und des Clients. In Kapitel 6, »Kommunikation«, erfahren Sie, wie Sie dieses Modul dazu einsetzen können, Daten über das Netzwerk oder lokal zwischen verschiedenen Prozessen zu versenden.

Os

Was der Browser für clientseitiges JavaScript ist, ist das Betriebssystem für serverseitiges JavaScript. In erster Linie eine Umgebung, die eine Reihe von Schnittstellen bie-

tet, mit denen Sie interagieren können. Wie im Browser existieren auch zwischen den verschiedenen Betriebssystemen und Rechnerarchitekturen gewisse Unterschiede, die die Entwicklung von Applikationen beeinflussen können. Das `OS`-Modul ermöglicht es Ihnen, gewisse Rahmenparameter des Betriebssystems, wie beispielsweise den Hostnamen oder die Art und Version des Systems, abzufragen und daraufhin die Ausführung der Applikation zu beeinflussen. Im Verlauf dieses Kapitels werden Sie dieses Modul noch näher kennenlernen und die Wirkungsweise der verschiedenen Features anhand praktischer Beispiele sehen.

Path

Node.js ist relativ unabhängig vom darunterliegenden Betriebssystem. Dies gilt jedoch nur bis zu bestimmten Grenzen. Vor allem im Bereich des Zugriffs auf das Dateisystem machen sich die Unterschiede zwischen UNIX-basierten Systemen und Windows bemerkbar. Um den Umgang mit diesem und anderen Problemen bei der Formulierung und dem Parsing von Pfadnamen zu erleichtern, steht Ihnen das `Path`-Modul zur Verfügung. Es erleichtert Ihnen außerdem den Umgang mit absoluten und relativen Pfaden. In Abschnitt 4.2, »Basismodule«, lernen Sie anhand verschiedener Beispiele den Umgang mit Pfaden in Node.js-Applikationen kennen.

Punycode

Punycode dient zur Umwandlung von Unicode-Zeichenketten ins ASCII-Format. Punycode kann dabei in beide Richtungen, also sowohl von Unicode nach ASCII als auch von ASCII in Unicode, verwendet werden. Punycode entstand aus der Notwendigkeit heraus, Domainnamen mit Sonderzeichen zu erlauben, die im Gegensatz dazu das DNS nicht erlaubt. Mit Punycode wurde ein Workaround geschaffen, der dies ermöglicht. Node.js setzt dabei auf einen eigentlich unabhängigen Punycode Converter, der seit Version 0.6.2 fester Bestandteil des Node.js-Modulsystems ist. Die wichtigsten Features dieses Moduls sind die verschiedenen Möglichkeiten der Codierung und Decodierung von Zeichenketten. Listing 4.10 zeigt den Einsatz beider Varianten.

```
var punycode = require('punycode'),
    decoded = 'ҢЄЅЉ ђ0RЅq',
    encoded = '-mfb6xb1226bstb4a3n9oaf';
console.log(punycode.encode(decoded));
console.log(punycode.decode(encoded));
```

Listing 4.10 Punycode in Node.js

Query String

Webseiten werden über ihre URL angesprochen. HTTP ermöglicht neben verschiedenen Angaben wie dem Hostnamen und der Portnummer auch die Spezifizierung

eines sogenannten Query Strings. Dieser wird normalerweise dazu genutzt, Parameter an den Server zu übermitteln. Der Query String folgt im Anschluss an den URL-Pfad und wird von diesem durch ein Fragezeichen getrennt. Der Query String besteht aus Key-Value-Paaren, die untereinander durch &-Zeichen separiert werden. Die Schlüssel und Werte sind durch =-Zeichen voneinander getrennt. Der Umgang mit Query Strings, sei es die Codierung oder die Decodierung, ist eine häufig wiederkehrende Aufgabe, die durch das Query String-Modul erheblich vereinfacht wird.

```
var querystring = require('querystring'),
    string = 'search=nodejs&page=1',
    obj = {search: "nodejs", page: "1"};
console.log(querystring.stringify(obj));
console.log(querystring.parse(string));
```

Listing 4.11 Umgang mit Query Strings

In Listing 4.11 sehen Sie den Einsatz der `stringify`-Methode, mit deren Hilfe ein Objekt in einen Query String gewandelt werden kann. Dabei werden Key-Value-Paare gebildet und mit &-Zeichen verbunden. Die `parse`-Methode dient dazu, einen Query String in eine Objektstruktur zu übersetzen, die Sie innerhalb Ihrer Applikation weiterverwenden können und innerhalb derer Sie auf die verschiedenen Komponenten des Query Strings durch die Schlüsselwerte zugreifen können. Beiden Methoden können Sie sowohl den Separator innerhalb der Key-Value-Paare als auch den Trenner zwischen den einzelnen Key-Value-Paaren übergeben und die Standards somit überschreiben.

Eine Problemstellung existiert noch im Zusammenhang mit dem Parsen von Query Strings. Ein Query String ist in der Regel ein Teil einer gesamten URL, Sie müssen diesen als solchen erst extrahieren, bevor Sie ihn mithilfe des Query String-Moduls weiterverarbeiten können. Node.js bietet Ihnen hierfür eine Kombination aus dem URL-Modul und dem Query String-Modul an. Die `parse`-Methode des URL-Moduls akzeptiert ein zweites, optionales Argument, das dafür sorgt, dass die `parse`-Methode des Query String-Moduls aufgerufen wird, um den Query String einer URL im selben Funktionsaufruf bereits zu parsen und als Array innerhalb des Rückgabewerts zurückzugeben.

Readline

Mit Node.js können Sie jedoch nicht nur Webapplikationen umsetzen, sondern auch Anwendungen, die unabhängig von Browserclients auf einem Server ihre Arbeit verrichten. In solchen Fällen ist es allerdings teilweise erforderlich, eine Schnittstelle zu bieten, mit der ein Benutzer auch diese Art von Anwendungen steuern kann. Auf der Kommandozeile stehen Ihnen hauptsächlich zwei Kommunikationskanäle zur Ver-

fügung: die Standardein- und -ausgabe. Diese können Sie dazu nutzen, um Eingaben des Nutzers anzunehmen beziehungsweise dem Nutzer Informationen darzustellen. Auch im Bereich von Webapplikationen können Kommandozeilenwerkzeuge, die in Node.js geschrieben wurden, zum Einsatz kommen. Dies ist vor allem bei Installation und Konfiguration der Fall.

```
var readline = require('readline'),
    rl = readline.createInterface({
      input: process.stdin,
      output: process.stdout
    });
rl.write('*** Node.js Kommandozeile ***\n');
rl.question('$> ', function (command) {
  rl.write(command + '\n');
  rl.close();
});
```

Listing 4.12 Kommandozeile mit Node.js

Das Beispiel in Listing 4.12 zeigt, wie Sie mit dem `Readline`-Modul Eingaben von Nutzern Ihrer Applikation auf der Kommandozeile abfragen können, um diese in Ihrer Applikation weiterzuverwenden. Die Initialisierung ist wenig problematisch. Sie müssen lediglich ein neues Interface erstellen und die Input- und Output-Kanäle definieren. Im Falle des Beispiels sind dies die Standardein- und -ausgabe. Nach der Initialisierung können Sie die Schnittstelle verwenden, um Informationen auf der Kommandozeile über die `write`-Methode auszugeben oder Eingaben mithilfe der `question`-Methode abzufragen. Die `question`-Methode funktioniert asynchron. Das bedeutet, dass das erste Argument beim Aufruf der Methode auf der Kommandozeile ausgegeben wird. Danach wartet das Script auf die Eingabe des Benutzers. Hat dieser die Eingabe beendet, wird die Callback-Funktion, die im zweiten Argument definiert wurde, mit dem Eingabewert aufgerufen. Bei der Verwendung des `Readline`-Moduls sollten Sie darauf achten, die erstellte Schnittstelle mit `close` wieder zu schließen, damit das Script ordnungsgemäß beendet werden kann.

REPL

REPL steht für Read-Eval-Print Loop und stellt die interaktive Node.js-Shell dar. In Kapitel 3, »Ein erstes Beispiel«, haben Sie bereits einen detaillierten Einblick in die Features des REPL-Moduls erhalten. Neben den dort vorgestellten interaktiven Features können Sie die interaktive Shell auch innerhalb eines normalen Node.js-Scripts starten. Zu diesem Zweck existiert die Methode `start`. Das nachfolgende Beispiel in Listing 4.13 zeigt, wie eine Shell mit einem benutzerdefinierten Prompt gestartet werden kann.

```
var repl = require('repl');
repl.start({
  prompt: 'NODE> ',
  input: process.stdin,
  output: process.stdout
});
```

Listing 4.13 REPL aus einem Script heraus starten

Die `start`-Methode akzeptiert ein Konfigurationsobjekt, mit dessen Hilfe sich das Verhalten der Shell beeinflussen lässt. Die Optionen reichen von den Schlüsseln `input` und `output`, mit denen Sie jeweils spezifizieren, wovon die Eingaben gelesen und wohin die Ausgaben geschrieben werden sollen, über `prompt`, mit der Sie angeben, wie der Prompt aussehen soll, bis hin zu `eval`, mit der Sie eine benutzerdefinierte Funktion zur Ausführung der eingegebenen Befehle angeben können.

Stream

Das `Stream`-Modul repräsentiert eine abstrakte Schnittstelle für les- und schreibbare Datenströme in Node.js und dient damit beispielsweise dem `Net`-Modul als Basis. Mithilfe des `Stream`-Moduls lassen sich sowohl für TCP-Verbindungen als auch für UNIX Domain Sockets Client- wie Serverkomponenten beschreiben, mit der Sie dann zwischen Applikationen, aber auch zwischen verschiedenen Systemen kommunizieren können. Beispiele, die die Funktionalität des `Stream`-Moduls veranschaulichen und weitere Hintergrundinformationen zur Kommunikation über Streams finden Sie in Kapitel 6.

String Decoder

Das `String Decoder`-Modul dient dazu, ein `Buffer`-Objekt in einen `String` zu wandeln. Zu diesem Zweck stellt Ihnen dieses Modul die Klasse `StringDecoder` zur Verfügung. Bei der Erstellung eines Objekts dieser Klasse können Sie dem Konstruktor das Encoding der Zeichenkette übergeben. Wird hier kein Wert übergeben, ist das Encoding standardmäßig UTF-8. Mithilfe der `write`-Methode können Sie dann ein `Buffer`-Objekt in eine Zeichenkette umwandeln, die im Anschluss zurückgegeben wird. Das Beispiel in Listing 4.14 verdeutlicht Ihnen diesen Workflow.

```
var Decoder = require('string_decoder').StringDecoder,
    Buffer = require('buffer').Buffer,
    decoder = new Decoder('utf-8'),
    buffer = new Buffer('Hello World');
console.log(decoder.write(buffer));
```

Listing 4.14 String Decoder-Funktionalität

Timers

Wie Sie schon bei der `console`-Funktionalität gesehen haben, wurden einige bereits aus der Browserumgebung bekannte Features auf Node.js portiert. Dies gilt auch für die zeitabhängigen Funktionen, die Timeouts und Intervalle steuern. Zunächst lernen Sie nun die Timeout-Funktionalität anhand eines kurzen Beispiels kennen. Listing 4.15 zeigt, wie Sie mit den entsprechenden Funktionen umgehen können.

```
var callback = function(string) {
    console.log('Hello ' + string);
}
var timeout = setTimeout(callback, 5000, 'World');
//clearTimeout(timeout);
```

Listing 4.15 `setTimeout` in Node.js

Im Beispiel sehen Sie, dass Sie mit der Funktion `setTimeout` eine Funktion nach Ablauf einer bestimmten Zeitspanne ausführen können. Das erste Argument, das Sie dieser Funktion übergeben, besteht aus der Callback-Funktion, die ausgeführt werden soll. Mithilfe des zweiten Arguments geben Sie die Zeitspanne in Millisekunden an, die gewartet werden soll, bevor die Funktion ausgeführt wird. Sie haben außerdem die Möglichkeit, der `setTimeout`-Funktion beliebig viele weitere Argumente mitzugeben. Diese werden beim Aufruf an die Callback-Funktion weitergegeben, wie es im Beispiel in Listing 4.15 mit der Zeichenkette `World` der Fall ist. Laufende Timeouts können Sie mit der Funktion `clearTimeout` abbrechen. Das führt dazu, dass die Callback-Funktion nicht ausgeführt wird. Um einen Timeout abbrechen zu können, benötigen Sie dessen ID. Diesen erhalten Sie als Rückgabewert der `setTimeout`-Funktion. Wenn Sie im Beispiel die beiden Schrägstriche vor der letzten Zeile entfernen, wird der Timeout abgebrochen, und Sie erhalten keine Ausgabe.

Es gibt Situationen, in denen Ihnen ein Timeout allein nicht ausreicht, sondern Sie die gleiche Funktion mehrmals hintereinander in gleichen Zeitabständen wieder aufrufen müssen. Statt mehrere Timeouts hintereinanderschalten, können Sie in diesen Fällen die `setInterval`-Funktion verwenden. Sie verhält sich wie die `setTimeout`-Funktion, auch die Signaturen der Funktionen gleichen sich. Der einzige Unterschied besteht darin, dass die Callback-Funktion nicht nur einmal, sondern so oft aufgerufen wird, bis die Ausführung mit `clearInterval` unterbrochen wird.

TLS

Mit dem TLS-Modul steht Ihnen ein Werkzeug zur Verfügung, mit dessen Hilfe Sie verschlüsselte Verbindungen zum Datentransfer verwenden können. Das TLS-Modul baut dabei auf einigen anderen Modulen auf, die Sie bereits kurz kennengelernt haben. Zum einen wird das `Crypto`-Modul für Verschlüsselungsaufgaben verwendet,

das `Stream`-Modul stellt die Schnittstellen für die Datenstreams zur Verfügung, und das `Net`-Modul bildet die Basis für eine sichere Serverstruktur.

TTY

Den Einsatz des `TTY`-Moduls konnten Sie im Verlauf dieses Kapitels schon anhand von Beispielen beobachten. Dieses Modul stellt les- und schreibbare Streams für Terminals zur Verfügung. Das bedeutet, dass Sie bei der Verwendung des REPL indirekt mit diesem Modul in Kontakt kommen. Direkt werden Sie dieses Modul eher selten verwenden. In der Praxis bedeutet dies, dass sowohl `process.stdin`, die Standardeingabe, als auch `process.stdout`, die Standardausgabe, Streams aus dem `TTY`-Modul sind. Sie können sehr einfach herausfinden, ob beide Variablen wirklich aus diesem Modul stammen, indem Sie den Code aus Listing 4.16 ausführen.

```
$ node
> process.stdout.isTTY
true
> process.stdin.isTTY
true
>
```

Listing 4.16 Prüfung, ob der REPL TTY-Streams verwendet

Die Eigenschaft `isTTY` besitzt den Wert `true`, wenn die Klassen aus dem `TTY`-Modul verwendet werden.

URL

Das `URL`-Modul wurde bereits angesprochen und fand auch in einigen Beispielen bereits Anwendung. Im einfachsten Fall verwenden Sie die `parse`-Methode, um eine URL in ein Objekt zu übersetzen, bei dem Sie über die verschiedenen Eigenschaften auf die einzelnen Bestandteile der URL zugreifen können.

```
var url = require('url'),
    string = 'http://www.google.de/search?q=node.js';
console.log(url.parse(string));
```

Listing 4.17 Parsing von URLs

Das Ergebnis aus Listing 4.17 ist ein Objekt, in dem Sie unter anderem auf das Protokoll `http`, den Host `www.google.de`, den Pfadnamen `/search` oder den Query String `q=node.js` zugreifen können.

Die zweite Funktionalität des `URL`-Moduls, die `format`-Methode, dient dazu, Ihnen den Weg zurück zu ermöglichen, also aus einem Objekt eine gültige URL zu generieren.

```
var url = require('url'),
    obj = {protocol: 'http',
          host: 'www.google.de',
          pathname: '/search',
          search: 'q=node.js'}
console.log(url.format(obj));
```

Listing 4.18 Erstellen einer URL

Listing 4.18 zeigt Ihnen einige der verfügbaren Eigenschaften, über die Sie eine gültige URL erzeugen können.

Die dritte und letzte Funktionalität des URL-Moduls besteht aus der `resolve`-Methode. Mit dieser lässt sich eine URL zusammensetzen. Zu diesem Zweck erhält die Methode zwei Argumente, eine Basis-URL und einen relativen Pfad. Der Rückgabewert besteht aus der zusammengesetzten URL. Bei der Zusammensetzung werden nicht nur Zeichenketten aneinandergefügt, sondern auch bestimmte Zeichen evaluiert. So bedeuten beispielsweise zwei Punkte, dass eine Ebene im Pfad zurückgegangen werden soll. Listing 4.19 verdeutlicht Ihnen diesen Sachverhalt.

```
var url = require('url')
    from = 'http://de.eurosport.yahoo.com/fussball/',
    to = '../eishockey/';
var resolved = url.resolve(from, to);
console.log(resolved);
```

Listing 4.19 Zusammensetzen einer URL

Die Ausgabe von Listing 4.19 ist die Zeichenkette *`http://de.eurosport.yahoo.com/eishockey/`*.

Utilities

Das Utilities-Modul fasst verschiedene Funktionalitäten zusammen, die im Entwicklungsprozess häufig gebraucht werden. Hierunter fallen Methoden zur Formatierung und Ausgabe, aber auch solche, die Ihnen beispielsweise die Vererbung innerhalb von Node.js-Applikationen erleichtern. Diese Methoden und weitere aus dem Utilities-Modul werden später in diesem Kapitel näher betrachtet und ihre Funktionalität an konkreten Beispielen erläutert.

VM

Das VM-Modul bietet Ihnen die Möglichkeit, JavaScript-Code zu evaluieren und ihn auszuführen. Im Gegensatz zur `eval`-Funktion, die Ihnen JavaScript direkt zur Verfügung stellt, existiert hier eine Abgrenzung des Kontexts, sodass die Ausführung

weniger Schaden anrichten kann. Das VM-Modul enthält verschiedene Methoden, mit denen der Code evaluiert und ausgeführt werden kann. Sie können allerdings den Code auch nur kompilieren, das Ergebnis speichern und zu einem späteren Zeitpunkt ausführen. Insgesamt existieren drei verschiedene Methoden, die sich im jeweiligen Ausführungskontext unterscheiden. `runInThisContext` führt den JavaScript-Code aus, der lokale Kontext außerhalb der VM kann nicht verändert, aber auch nicht gelesen werden. Mit der Methode `runInNewContext` können Sie einen Ausführungskontext definieren, in dem der Quellcode ausgeführt werden soll. Das bedeutet, dass Sie eine Umgebung für die Ausführung mit bestimmten Variablen initialisieren können. Die dritte Methode, `runInContext`, schließlich erlaubt es Ihnen, einen Kontext mitzugeben, innerhalb dessen das Script ausgeführt werden soll. Diesen Kontext können Sie mithilfe der `createContext`-Methode des VM-Moduls erstellen.

Alle drei VM-Methoden akzeptieren ein optionales `filename`-Argument. Dieses dient jedoch nicht dazu, den kompilierten Quellcode oder das Ergebnis in eine Datei zu speichern, sondern wird lediglich in Stacktraces verwendet, um zu signalisieren, dass der Quellcode, der in einer VM ausgeführt wird, eigentlich nicht zur Datei gehört, in der die VM definiert wird.

```
var vm = require('vm'),
    context = {
      firstname: 'John',
      lastname: 'Doe'
    },
    myContext = vm.createContext(context),
    firstname = 'Chuck',
    lastname = 'Norris',
    script = 'firstname = "Mickey"';
vm.runInContext(script, myContext);
console.log('Local: ', firstname, lastname);
console.log('VM: ', myContext.firstname, myContext.lastname);
```

Listing 4.20 Das Node.js VM-Modul

Im Beispiel in Listing 4.20 wird JavaScript-Code in einem neu definierten Kontext ausgeführt. In diesem Kontext sind die Variablen `firstname` und `lastname` bereits mit den Werten `John` und `Doe` vorbelegt. Der auszuführende Quellcode weist `firstname` einen neuen Wert zu. Um zu verdeutlichen, dass innerhalb einer VM kein Zugriff nach außen stattfinden kann, wurden außerdem die Variablen `firstname` und `lastname` außerhalb der VM zusätzlich definiert. Mit der Methode `runInContext` führen Sie den Quellcode aus. Die Ergebnisse des Scripts sehen Sie schließlich, wenn Sie die Variablen mithilfe von `console.log` ausgeben lassen. Sie erhalten Zugriff auf den Kontext der VM über die Variable `myContext`, den Rückgabewert der Methode `createContext`.

ZLIB

Bei der Entwicklung von Software sind Sie immer an bestimmte Beschränkungen gebunden. Bei Server- und Desktop-Anwendungen sind das vor allem Arbeitsspeicher, Rechenleistung und benötigter Festplattenspeicher. Bei Webanwendungen kommen auch noch die verfügbare Bandbreite und die Anzahl der unterstützten Clients hinzu. Was die Bandbreite, aber auch den erforderlichen Speicherplatz von Dateien betrifft, können Sie das ZLIB-Modul verwenden, um Daten zu komprimieren und sie auch wieder zu entpacken. Zu diesem Zweck stehen Ihnen verschiedene Komprimierungsalgorithmen zur Verfügung. Das ZLIB-Modul funktioniert mit schreib- und lesbaren Streams.

```
var zlib = require('zlib'),
    string = 'Hello World'
    compressed = new Buffer('eJzzSM3JyVcIzy/KSQEAGAsEHQ==', 'base64');
zlib.deflate(string, function(err, buffer) {
    if (err) throw err;
    console.log(buffer.toString('base64'));
});
zlib.inflate(compressed, function(err, buffer) {
    if (err) throw err;
    console.log(buffer.toString());
});
```

Listing 4.21 Das ZLIB-Modul im Einsatz

In Listing 4.21 sehen Sie den Einsatz der Deflate-Komprimierung. Im ersten Teil des Beispiels übergeben Sie die Zeichenkette `Hello World` als erstes Argument an die `deflate`-Methode. Die Komprimierung dieser Daten findet asynchron statt. Aus diesem Grund ist das zweite Argument eine Callback-Funktion, die ausgeführt wird, sobald die Komprimierung abgeschlossen ist. Die Callback-Funktion erhält zwei Argumente. Das erste ist ein Objekt, das eventuelle Fehler enthält, die bei der Komprimierung aufgetreten sind. Das zweite Argument ist ein Buffer-Objekt, das die komprimierte Zeichenkette enthält. Der zweite Teil des Beispiels beschäftigt sich mit der Dekomprimierung der Zeichenkette. Dazu wird aus einer komprimierten Zeichenkette ein Buffer-Objekt erstellt, das als Argument der `inflate`-Methode übergeben wird. Auch das Dekomprimieren der Zeichenkette findet asynchron statt. Wie schon bei `deflate` ist auch hier das zweite Argument eine Callback-Funktion, die ausgeführt wird, sobald die Zeichenkette entpackt ist. Das erste Argument ist hier auch wieder das Fehler-Objekt, das zweite die entpackte Zeichenkette. Innerhalb des Callbacks können Sie dann mit der entpackten Zeichenkette weiterarbeiten.

In den vorangegangenen Abschnitten haben Sie viel über die verschiedenen Module der Node.js-Plattform erfahren. Diese stellen neben der V8-Engine und den C-Biblio-

theiken den Kern der Plattform dar und erweitern deren Funktionalität. Die nächsten Abschnitte gehen auf verschiedene Module detaillierter ein.

4.2 Basismodule

Sie haben bereits erfahren, dass die verschiedenen Module der Node.js-Plattform mehr oder weniger unabhängig voneinander sind. Es existieren jedoch einige Module, auf denen andere Module aufbauen. Weitere Module stellen lediglich spezielle Funktionalität zur Verfügung. Im Folgenden werden Sie diese Basismodule näher kennenlernen.

4.2.1 Globale Objekte

Möchten Sie zusätzliche Funktionalität in Form von Node.js-Modulen in Ihrer Applikation einsetzen, müssen Sie normalerweise das Modul mit der `require`-Funktion einbinden. Über den Rückgabewert dieser Funktion können Sie auf die jeweilige Funktionalität zugreifen. Einige Objekte und Funktionen werden so häufig benötigt, dass sich die Entwickler von Node.js entschieden haben, diese direkt im globalen Kontext zur Verfügung zu stellen. Sie müssen in diesem Fall kein Modul vor der Verwendung einbinden. Die globalen Funktionen und Objekte stehen im globalen Kontext zur Verfügung. Dieser Kontext ist mit dem `window`-Kontext des Browsers vergleichbar. Zu beachten ist dabei, dass jedes Modul seinen eigenen globalen Kontext besitzt. In der folgenden Auflistung finden Sie alle globalen Funktionen und Objekte in Node.js.

▶ <code>__filename</code>	▶ <code>exports</code>
▶ <code>__dirname</code>	▶ <code>process</code>
▶ <code>require()</code>	▶ <code>require</code>
▶ <code>Buffer</code>	▶ <code>setTimeout/clearTimeout</code>
▶ <code>Console</code>	▶ <code>setInterval/clearInterval</code>

Datei- und Verzeichnisname

Zwei Variablen, die bei der Entwicklung von Applikationen sehr hilfreich sind, sind `__filename` und `__dirname`. Wie die Namen dieser Variablen bereits vermuten lassen, enthalten beide Informationen über den Speicherort der Datei, in der der Quellcode des aktuell ausgeführten Scripts gespeichert ist. Die Zeichenkette in `__filename` ist der absolute Pfad und der Dateiname des Scripts. `__dirname` ist lediglich der Pfad. Beachten Sie bei der Verwendung der interaktiven Shell, dass weder `__filename` noch `__dirname` definiert ist, da es in diesem Fall kein Script gibt. Neben diesen beiden Variablen existiert im globalen Kontext eine weitere Funktion, die Sie im Verlauf dieses Buches bereits einige Male verwendet haben, die `require`-Funktion. Sie dient zum Laden von Modulen.

Buffer

Auch die Buffer-Klasse ist global verfügbar, da Sie in Node.js sehr häufig mit dieser Art von Objekten zu tun haben. Den Einsatz dieser Funktionalität können Sie beispielsweise im Abschnitt über das `ZLIB`-Modul in diesem Kapitel sehen, in dem der `inflate`-Funktion eine komprimierte Zeichenkette als Buffer-Objekt zum Entpacken übergeben wird. Ein Objekt dieser Klasse kann einfach erstellt werden und es ist nicht erforderlich das `Buffer`-Modul zuvor zu laden.

console

Eine weitere Funktionalität, die Sie schon aus dem clientseitigen JavaScript kennen und über die Sie bereits im Verlauf dieses Kapitels Näheres erfahren haben, ist auch als globales Objekt verfügbar. Die Rede ist vom `console`-Objekt, das vor allem zur Ausgabe auf der Kommandozeile und zum Messen von Ausführungszeiten während der Entwicklung verwendet wird.

exports

Auch das Erstellen eigener Module geschieht bei der Arbeit mit Node.js sehr häufig. Hierfür stellt Ihnen Node.js das Objekt `exports` zur Verfügung. Dieses Objekt ist allerdings nicht im globalen Kontext vorhanden. Für jedes Modul, das Sie erstellen, existiert ein eigenes `exports`-Objekt. Es wird verwendet, um die Funktionalität eines Moduls nach außen verfügbar zu machen. Das `exports`-Objekt ist ein Bestandteil des `Modules`-Moduls und lautet vollständig `module.exports`, wobei beide Schreibweisen die gleiche Funktionalität aufweisen. In Listing 4.22 sehen Sie die Verwendung des `exports`-Objekt in Form eines E-Mail-Validators, den Sie beispielsweise in der Datei `validator.js` speichern. Sie können diese Funktion in Ihrer Applikation verwenden, indem Sie die Zeile `require('validator.js').email('test@test.de');` in Ihre Applikation integrieren. Dabei wird geprüft, ob die Zeichenkette `test@test.de` eine gültige E-Mail-Adresse enthält.

```
exports.email = function (email) {
    var regex = /^[a-z0-9_\. -]+@([\da-z\.-]+\.[a-z\.-]{2,6})$/;
    result = regex.exec(email);

    if (result) {
        return true;
    }

    return false;
};
```

Listing 4.22 E-Mail-Validator in einem eigenen Modul

Im Verlauf dieses Kapitels werden Sie die Erstellung eigener Module und die Verwendung des `Modules`-Moduls noch im Detail kennenlernen.

Das `exports`-Objekt ist jedoch nicht das einzige Feature, das aus dem `Modules`-Modul global zur Verfügung gestellt wird. Auch das `module`-Objekt selbst ist ohne einen vorhergehenden Aufruf von `require` verfügbar. Dabei handelt es sich wie auch schon beim `exports`-Objekt um ein Objekt, das für jedes Modul separat existiert und so nicht für den gesamten Quellcode einer Applikation global verfügbar ist.

Process

Bei der Entwicklung von Applikationen wird häufig auf die Informationen des aktuellen Node.js-Prozesses zugegriffen. Aus diesem Grund ist auch das `Process`-Modul, das Ihnen eben diese Informationen bietet, global verfügbar. Genauso kann über die Methoden dieses Moduls steuernd in den Prozess eingegriffen werden. Nähere Informationen zur Verwendung der Funktionalität dieses Moduls erfahren Sie im weiteren Verlauf dieses Kapitels.

Den Node.js-Modulloader können Sie vor allem durch den Einsatz der `require`-Funktionalität verwenden. Diese ist ebenfalls global verfügbar. In den nächsten Abschnitten erfahren Sie mehr darüber, wie Sie Module laden können, aber auch wie Sie das Verhalten des Modulloaders beeinflussen können.

Die bereits behandelten `Timeout`- und `Interval`-Funktionen kommen recht häufig zum Einsatz. Aus diesem Grund sind sie, wie auch im Browser, als globale Funktionen verfügbar.

4.2.2 Utility

Das `Utility`-Modul enthält zahlreiche hilfreiche Methoden, die Sie während der Entwicklung Ihrer Applikation einsetzen können.

Eine Funktion, die viele Entwickler vermissen, die aus anderen Programmiersprachen kommen, ist die aus C bekannte Funktion `printf`. Das `Utility`-Modul bietet Ihnen mit der `format`-Funktion eine ähnliche Funktionalität. Ihren Einsatz demonstriert Ihnen Listing 4.23.

```
var util = require('util'),
    string = util.format('Hello %s', 'World');
util.log(string);
```

Listing 4.23 Die `format`-Funktion des `Utility`-Moduls

Die `format`-Funktion arbeitet wie die `printf`-Funktion in C mit einer Zeichenkette, die einen oder mehrere Platzhalter enthält, die nach dem Funktionsaufruf durch die wei-

teren Argumente ersetzt werden. Die Platzhalter weisen immer das folgende Format auf: %, also ein Prozentzeichen, gefolgt von einem Buchstaben. Im Gegensatz zu C können Sie in JavaScript allerdings nur auf drei Varianten von Platzhaltern zurückgreifen: `s` steht dabei für einen String, also eine Zeichenkette, `d` für Zahlen und `j` für Daten im JSON-Format. Das `Utility`-Modul unterstützt Sie neben der `format`-Funktion mit weiteren Funktionen zur Ausgabe von Informationen.

Ausgabe auf Stdout	Ausgabe auf Stderr
Util.log	Util.debug
Util.puts	Util.error
Util.print	

Tabelle 4.3 Ausgabemethoden des `Utility`-Moduls

Im Beispiel in Listing 4.23 sehen Sie einen Vertreter der Gruppe von Methoden im Einsatz, mit deren Hilfe Sie Daten ausgeben können. Wie Sie in Tabelle 4.3 sehen können, lassen sich die Ausgabemethoden danach untergliedern, in welche Kanäle sie die ihnen übergebenen Daten schreiben. Die bereits im Beispiel vorgestellte `util.log`-Methode schreibt die übergebene Zeichenkette auf die Standardausgabe. Der Zeichenkette werden bei der Ausgabe das aktuelle Datum und die Uhrzeit vorangestellt. Eine weitere Funktion aus diesem Bereich ist `util.puts`. Diese verhält sich ähnlich zu `console.log` und gibt die übergebenen Werte auf der Standardausgabe aus. Im Gegensatz zur `util.log`-Methode akzeptiert `util.puts` mehrere Argumente und gibt diese jeweils in separaten Zeilen aus. Die dritte Methode, `util.print`, ist eine Abwandlung von `util.puts`. Der einzige Unterschied zwischen diesen beiden Methoden besteht darin, dass bei der Ausgabe mit `util.print` zwischen und nach den Werten kein Zeilenumbruch eingefügt wird. Analog zum Schreiben auf der Standardausgabe existieren mit `util.debug` und `util.error` zwei Methoden, mit denen Sie Informationen auf die Standardfehlerausgabe schreiben können. Die Methode `util.debug` akzeptiert lediglich eine Zeichenkette und gibt diese mit der Zeichenkette `DEBUG:` vorangestellt auf der Standardfehlerausgabe aus. `util.error` verhält sich wie `util.puts`, mit dem Unterschied, dass die Ausgabe auf der Standardfehlerausgabe erfolgt. Die Ausgaben der vorgestellten Methoden erfolgt synchron. Das bedeutet, dass die Ausführung der Applikation für die Zeit der Ausgabe angehalten und im Anschluss fortgesetzt wird. Im Gegensatz zu den `console`-Methoden, die ebenfalls zur Ausgabe, allerdings hauptsächlich in Ausnahmen zum Debugging, eingesetzt werden, sind die Ausgabemethoden des `Utility`-Moduls dazu gedacht, dass Sie sie produktiv in einer Applikation einsetzen, wenn es darum geht, Ausgaben auf der Konsole zu erzeugen.

Mit den bisherigen Methoden des `Utility`-Moduls können Sie Zeichenketten auf verschiedenen Ausgabekanälen ausgeben. In der Programmierung und hier vor allem bei der Fehlersuche benötigen Sie jedoch häufig eine Möglichkeit, ein Objekt im Detail zu betrachten. Mit der `inspect`-Methode erhalten Sie auch für diese Problemstellung eine passende Lösung. Diese Methode ermöglicht es Ihnen, zu bestimmen, ob Sie versteckte Eigenschaften anzeigen lassen möchten, bis in welche Tiefe Sie eine Objektstruktur betrachten wollen und sogar, ob Sie eine farbige Ausgabe erhalten möchten. Den Rückgabewert können Sie mit einer Ausgabefunktion des `Utility`-Moduls oder einem einfachen `console.log` ausgeben. Listing 4.24 zeigt Ihnen den Einsatz der `inspect`-Methode im Node.js REPL.

```
$ node
> var util = require('util');
undefined
> util.puts(util.inspect(Buffer));
{ [Function: Buffer]
  isEncoding: [Function],
  poolSize: 8192,
  isBuffer: [Function: isBuffer],
  byteLength: [Function],
  concat: [Function] }
undefined
>
```

Listing 4.24 inspect im REPL

Neben den bisher vorgestellten Methoden, die vor allem die Ausgabe von Daten in einer Applikation betreffen, weist das `Utility`-Modul noch weitere Funktionalitäten auf, die über die Ausgabe hinausgehen. In JavaScript ergibt sich immer wieder die Notwendigkeit, zu prüfen, von welchem Typ ein bestimmtes Objekt ist. Da JavaScript eine nicht typisierte Sprache ist, stellt diese Anforderung ein Problem dar. Node.js hat für diesen Zweck eine Reihe von Methoden, mit denen Sie prüfen können, ob ein Objekt von einem der nativen Datentypen von JavaScript, also `Array`, `RegExp` oder `Date`, ableitet. Die Methoden heißen dementsprechend auch `isArray`, `isRegExp` und `isDate`. Sie akzeptieren jeweils ein Objekt als Argument und geben `true` zurück, für den Fall, dass das Objekt vom jeweiligen Typ ist. Neben diesen Prüfungen existiert eine weitere Methode, `isError`, mit deren Hilfe Sie feststellen können, ob ein Objekt vom Typ `Error` ist. Listing 4.25 stellt Ihnen den Einsatz der `isArray`-Methode vor.

```
var arr = [1, 2, 3],
    util = require('util');
if (util.isArray(arr)) {
    util.log('The given object is an array.');
```

```

} else {
    util.log('The given object is not an array.');
```

Listing 4.25 Typenprüfung in Node.js

Eine der bedeutendsten Methoden aus dem Utility-Modul ist die `inherits`-Methode. JavaScript basiert nicht auf Klassen, stattdessen werden Prototypen verwendet, um die Struktur von Objekten zu definieren. Mithilfe der Prototypen können Sie auch Vererbung in JavaScript realisieren.

```

var util = require('util');

var Parent = function () {};
Parent.prototype.doSomething = function () {
    console.log('Parent: do something');
}

var Child = function () {};
util.inherits(Child, Parent);
Child.prototype.doSomethingElse = function () {
    console.log('Child: do something else');
}

var child = new Child();
child.doSomething();
child.doSomethingElse();
```

Listing 4.26 Vererbung in Node.js

Das Beispiel in Listing 4.26 veranschaulicht Ihnen, wie Vererbung in Node.js funktionieren kann. Zu diesem Zweck definieren Sie hier zunächst eine Konstruktor-Funktion `Parent` und erstellen in ihrem Prototyp eine Methode `doSomething`. Für den nächsten Schritt benötigen Sie eine weitere Konstruktor-Funktion `Child`. Danach kommt die `inherits`-Methode ins Spiel. Das erste Argument ist die Subklasse, das zweite die Elternklasse. Durch diesen Methodenaufruf erbt die Subklasse die Methode `doSomething` der Elternklasse. Diese wird im Prototyp der Subklasse verankert. Als Nächstes können Sie die Subklasse mit weiteren Methoden bestücken, im Falle des Beispiels in Listing 4.26 mit der `doSomethingElse`-Methode. Damit ist Ihre zweistufige Vererbungshierarchie vollendet, und Sie können ein Objekt der `Child`-Klasse erzeugen und verwenden.

Die `inherits`-Methode ist ein gutes Beispiel dafür, wie die Node.js-Plattform ihren Vorteil aus einer einheitlichen JavaScript-Umgebung zieht, ganz im Gegensatz zum clientseitigen JavaScript, bei dem mehrere JavaScript-Engines in verschiedenen Versionen und mit unterschiedlichem Funktionsumfang zum Einsatz kommen. So unterstützt Node.js den ECMAScript-Standard in der Version 5. Die gesamte Spezifikation dieses Standards finden Sie unter <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. Bei der Vererbung kommen verschiedene Funktionalitäten daraus zum Einsatz. So wird beispielsweise ein Klon des Prototyps der Elternklasse erstellt und die Konfiguration der `constructor`-Eigenschaft verändert, sodass bei der Iteration über die Eigenschaften eines Objekts der Konstruktor ausgeschlossen wird. Die gesamte Logik der `inherits`-Methode umfasst elf Zeilen Quellcode, was im Vergleich zu clientseitigem JavaScript, bei dem verschiedene Workarounds zur Browserunterstützung eingesetzt werden, sehr wenig ist.

4.2.3 Events

Im Verlauf dieses Kapitels haben Sie bereits einen kurzen Einblick in das `Events`-Modul erhalten. Da die Node.js-Plattform bei der Kommunikation zwischen Objekten häufig auf Events setzt, ist es sinnvoll, wenn Sie auch bei der Umsetzung Ihrer eigenen Funktionalität auf diese Architekturform zurückgreifen. Neben der einfachen Verwendung des `Events`-Moduls, wie Sie sie in Listing 4.9 sehen konnten, existieren noch weitere Methoden, mit denen Sie Events und deren Behandlung besser steuern können. Bisher haben Sie lediglich die Verwendung der Methoden `emit` zum Absetzen von Events und `on` zum Reagieren auf Events gesehen. Die `on`-Methode sorgt dafür, dass eine Callback-Funktion registriert und durch `emit` aufgerufen wird, wann immer das spezifizierte Event ausgelöst wird. Mit der `once`-Methode haben Sie die Möglichkeit, die Callback-Funktion nur beim ersten Mal auszuführen, wenn das Event ausgelöst wird. Bereits registrierte Eventlistener können Sie auch wieder entfernen. Dies können Sie entweder für einen speziellen Listener mit der `removeListener`-Methode oder für sämtliche Listener für ein bestimmtes Event mit `removeAllListeners` erreichen.

Listing 4.27 stellt Ihnen eine Kombination aus diesen Methoden vor. Außerdem nutzen wir hier die `listeners`-Methode, mit der Sie herausfinden können, welche Listener auf ein bestimmtes Event gebunden sind.

```
var emitter = new (require('events').EventEmitter)();
var callback = function () {
  console.log('Once ping event');
};
emitter.once('ping', callback);
emitter.on('ping', function() {console.log('On ping event');});
console.log(emitter.listeners('ping'));
```

```
emitter.removeListener('ping', callback);
console.log(emitter.listeners('ping'));
```

Listing 4.27 Management von Eventlistenern

Die Grundlage des Beispiels bildet eine Kopie des Objekts `EventEmitter`. Auf diese werden im ersten Schritt zwei Listener auf das `ping`-Event gebunden. Zur Demonstration erhalten Sie über den Aufruf der `listeners`-Methode mit der Zeichenkette `ping` ein Array der gebundenen Listener. Danach wird der Listener, der lediglich beim ersten Mal ausgeführt wird, wieder entfernt. Auch nach dieser Operation geben Sie die gebundenen Listener wieder aus, um den Effekt zu verdeutlichen.

Zusammenfassend lässt sich über das `Events`-Modul sagen, dass es die Funktionalität zur Verfügung stellt, die Sie benötigen, wann immer Sie in Ihrer Applikation mit Ereignissen umgehen müssen. Die Klasse `EventEmitter` sollte Ihnen daher als Basis aller Funktionalitäten dienen, die auf Events aufbauen, da Sie dadurch eine einheitliche Schnittstelle für das Eventhandling in Ihrer Applikation einführen.

4.2.4 OS

Auch das `OS`-Modul hat eine zentrale Bedeutung für Ihre Applikation. Hier stehen Ihnen vor allem Möglichkeiten zur Abfrage von Betriebssystemparametern zur Verfügung. Diese nutzen Sie nun, um in einem weiteren Beispiel eine kleine Übersicht über die Rahmenparameter Ihres Systems zu generieren. Diese Übersicht soll den Hostnamen, die Information über die Art des Betriebssystems, die Rechnerarchitektur und die Anzahl der CPUs anzeigen. Außerdem sollen die Uptime, der freie Arbeitsspeicher und die Rechenlast dargestellt werden. Da diese Informationen recht dynamisch sind, soll die Anzeige in regelmäßigen Abständen aktualisiert werden.

```
var os = require('os');

var Monitor = function () {
  this.hostname = os.hostname();
  this.sysInfo = 'on a ' + os.type() + ' ' + os.platform() + ' ' + os.arch()

    + ' System with ' + os.cpus().length + ' CPUs';
  this.totalmem = os.totalmem() / 1048576 + ' MB';
  this.uptime;
  this.freemem;
  this.load;
}

Monitor.prototype.setUptime = function() {
  var seconds = os.uptime();
```

```

    var hours = Math.floor(seconds / 3600),
        minutes = Math.floor((seconds - hours * 3600) / 60),
        seconds = (seconds - hours * 3600 - minutes * 60);
    this.uptime = hours + ':' + minutes + ':' + seconds;
}

Monitor.prototype.setFreemem = function() {
    this.freemem = this.formatNumber(os.freemem() / 1048576);
}

Monitor.prototype.setLoad = function () {
    this.load = this.formatNumber(os.loadavg()[0]);
}

Monitor.prototype.formatNumber = function(number) {
    return Math.round(number * 100) / 100;
}

Monitor.prototype.clearScreen = function () {
    for(var i = 0; i < process.stdout.rows; i++) {
        console.log('\r\n');
    }
}

Monitor.prototype.update = function () {
    this.setUptime();
    this.setFreemem();
    this.setLoad();
    return this;
}

Monitor.prototype.output = function() {
    this.clearScreen();
    console.log(this.hostname);
    console.log(this.sysInfo);
    console.log('Uptime: ' + this.uptime);
    var free = this.freemem;
    var total = this.totalmem;
    console.log('Freemem: ' + free + ' MB of ' + total + ' MB');
    console.log('Load: ' + this.load);
}

var sysMon = new Monitor();
setInterval(function() {sysMon.update().output()}, 500);

```

Listing 4.28 Systemmonitor in Node.js

Die Grundlage für das Beispiel in Listing 4.28 bildet die Klasse `Monitor`. Sie ist dafür zuständig, sämtliche Informationen zu kapseln, die Daten auf Anforderung zu aktualisieren und sie darzustellen. In der Konstruktor-Funktion `Monitor` definieren Sie die benötigten Eigenschaften. Einige dieser Eigenschaften sind statisch und können im Konstruktor gleich mit entsprechenden Werten belegt werden. Hierunter fallen beispielsweise der Hostname, die Systeminformationen und die Größe des Speichers.

Die Methoden der Klasse werden als Eigenschaften des Prototyps von `Monitor` definiert. Die Schnittstellen nach außen, die ein Benutzer verwenden sollte, sind die `update`-Methode zur Aktualisierung der dynamischen Daten und die `output`-Methode zur Ausgabe der Informationen auf der Konsole. Die dynamischen Informationen, die bei jeder Aktualisierung ausgelesen werden müssen, bestehen aus der Uptime des Systems, dem aktuell freiem Speicher und der Last des Systems. Die Informationen, die in den Eigenschaften der Instanz des Monitors festgehalten werden, werden über die Methoden `setUptime`, `setFreeMem` und `setLoad` aus dem System ausgelesen. Dabei ist zu beachten, dass Sie die Uptime im Format »Stunden:Minuten:Sekunden«, den freien Arbeitsspeicher in Megabyte und die Last auf zwei Nachkommastellen gerundet ausgeben sollten. Auch diese Aufgaben übernehmen in Ihrem Fall die Methoden, die auch schon die Informationen auslesen. Zur Formatierung der Zahlen erstellen Sie die Hilfsfunktion `formatNumber`, die die Rundung von Zahlen auf zwei Nachkommastellen übernimmt. Die Darstellung der Informationen geschieht in der `output`-Methode, die die Eigenschaften der `Monitor`-Instanz ausliest und sie auf der Standardausgabe ausgibt.

Damit Sie die Funktionsfähigkeit des Quellcodes testen können, müssen Sie lediglich die Klasse `Monitor` instanziiieren und auf diesem Objekt die `update`- und `output`-Methoden aufrufen. Allerdings war eine der Anforderungen, dass die angezeigten Informationen aktualisiert werden sollen. Aus diesem Grund müssen Sie einerseits in der `output`-Methode dafür sorgen, dass der Bildschirm geleert wird, was Sie mit der Hilfsfunktion `clearScreen` erreichen, und andererseits mit der Funktion `setInterval` dafür sorgen, dass die Methoden `update` und `output` in regelmäßigen Abständen aufgerufen werden.

Somit haben Sie einen sehr leichtgewichtigen Systemmonitor geschaffen, der sich durch die Features des `OS`-Moduls noch erweitern lässt. Außerdem besteht die Möglichkeit, weitere Informationen über das System aus dem Dateisystem und weiteren Systemwerkzeugen zu gewinnen und diese ebenfalls in den Systemmonitor zu integrieren. Auch eine grafische Aufbereitung der angezeigten Informationen ist denkbar.

4.2.5 Process

Wo Ihnen das `OS`-Modul auf Systemebene Informationen zur Verfügung stellt, operiert das `Process`-Objekt auf einer höheren Ebene. Dieses Objekt bildet die Schnitt-

stelle zum aktuell laufenden Node.js-Prozess. Im Gegensatz zu `OS` ist `Process` auch kein vollwertiges Node.js-Modul, dessen Quellcode im `lib`-Verzeichnis von Node.js als JavaScript-Datei zur Verfügung steht, sondern ein Objekt zur Abstraktion und Prozesssteuerung, das von Node.js direkt zur Verfügung gestellt wird.

Im letzten Beispiel, dem Systemmonitor aus Listing 4.28, haben Sie das `process`-Objekt bereits eingesetzt, als es darum ging, den Bildschirm zu leeren. Listing 4.29 zeigt diesen Teil des Quellcodes.

```
var clearScreen = function () {
    for(var i = 0; i < process.stdout.rows; i++) {
        console.log('\r\n');
    }
}
```

Listing 4.29 Clear-Screen mit dem `process`-Objekt

In diesem Beispiel stellt das `process`-Objekt die Eigenschaft `stdout` zur Verfügung, über die Sie beispielsweise die Anzahl der Spalten der Konsole auslesen können. Neben der Standardausgabe können Sie über die Eigenschaften `stderr` und `stdin` auch auf die Standardfehlerausgabe und Standardeingabe zugreifen. Über das `process`-Objekt haben Sie neben der Möglichkeit, direkt auf die Ausgabe und Eingabe zuzugreifen, außerdem eine Schnittstelle, mit deren Hilfe Sie den aktuellen Prozess beenden können. Der Standardweg, um eine Applikation zu beenden, ist neben der reinen Abarbeitung aller Kommandos, nach der der Node.js Prozess automatisch beendet wird, die Verwendung der `exit`-Methode des `process`-Objekts. Diese Methode akzeptiert ein Argument, das als Statuscode verwendet wird und im Normalfall an den Elternprozess, die Konsole des Systems zurückgegeben wird.

```
$ node
> process.exit(42);
$ echo $?
42
```

Listing 4.30 Rückgabewert von Node.js-Applikationen

Das Beispiel in Listing 4.30 zeigt Ihnen, wie Sie `process.exit` auch in der Node.js-Shell verwenden können. Mit `echo $?` können Sie auf der UNIX-Shell den Rückgabewert der letzten Applikation abfragen. Ein Rückgabewert von 0 bedeutet, dass die Applikation erfolgreich und ohne Fehler ausgeführt und beendet wurde. Ein Wert ungleich 0 bedeutet, dass ein Fehler bei der Ausführung aufgetreten ist. Den Wert, den Sie zurückgeben, können Sie dazu verwenden, um den Grund für den Abbruch anzugeben, er kann also als eine Art Fehlercode dienen. Einen Schritt drastischer ist die

Wirkung der Methode `abort`. Sie dient dazu, eine Applikation abubrechen. Sie wird also nicht regulär und kontrolliert beendet.

```
$ node
> process.abort();
Abort trap: 6
$ echo $?
134
```

Listing 4.31 Abbruch einer Applikation

Wie Sie in Listing 4.31 sehen können, verhält sich eine Node.js-Applikation grundlegend unterschiedlich, wenn sie durch einen Aufruf der `abort`-Methode abgebrochen wird. Zum einen erhalten Sie die systemseitige Ausgabe `Abort trap: 6`, was darauf hinweist, dass die Applikation durch ein `Abort`-Signal abgebrochen und nicht durch die Applikation gefangen wurde. Außerdem wird der Rückgabewert der Applikation automatisch auf den Wert 134 gesetzt. Dieser signalisiert ebenfalls, dass die Applikation abgebrochen wurde.

Die dritte und bei Weitem mächtigste Möglichkeit eine Applikation zu beenden ist der Einsatz der `kill`-Methode. Sie können diese Methode nicht nur dazu verwenden, um die eigene Applikation zu beenden, sondern jeden Prozess auf Ihrem System, wenn der Nutzer, unter dem die Node.js-Applikation ausgeführt wird, die Rechte dazu besitzt. Funktionalitäten wie diese sind ein Grund, warum eine Node.js-Applikation nicht von einem Administratornutzer ausgeführt werden sollte. Die `kill`-Methode akzeptiert zwei Argumente, das erste besteht aus der Prozess-ID, an die ein bestimmtes Signal geschickt werden soll. Das zweite Argument ist das Signal, das Sie senden möchten. Wird das zweite Argument nicht angegeben, wird die Aufforderung zum Beenden des Prozesses standardmäßig mit `SIGTERM` gesendet.

```
$ node
> process.kill(process.pid, 'SIGKILL');
Killed: 9
$ echo $?
137
```

Listing 4.32 Prozesse mit `kill` beenden

Das Beispiel in Listing 4.32 zeigt Ihnen zwei Fälle, in denen Sie auf das `process`-Objekt zurückgreifen müssen. Zum einen ist das die bereits erwähnte `kill`-Methode, um ein Signal an einen Prozess zu senden, und zum anderen die `pid`-Eigenschaft des `process`-Objekts, das die ID des aktuellen Node.js-Prozesses enthält, an den Sie in diesem Fall das Signal senden möchten. Angenommen, der Beispielprozess lässt sich nicht

mehr durch `SIGTERM` beenden, benötigen Sie in diesem Fall `SIGKILL` als Signal, das sich nicht durch den Prozess selbst abfangen lässt, sondern direkt an den Kernel gesendet wird, der sich darum kümmert, den Prozess zu beenden. Das bedeutet allerdings auch, dass der Prozess keine weitere Möglichkeit hat, seine Daten, die er eventuell im Arbeitsspeicher vorhält, auf die Festplatte zu persistieren. Ähnlich zum Beenden durch die `abort`-Methode wird auch bei `kill` mit `SIGKILL` vom System automatisch ein Rückgabewert von 137 gesetzt und die Zeichenkette `Killed: 9` ausgegeben. Wobei die Zahl 9 die Integerrepräsentation von `SIGKILL` und 15 die von `SIGTERM` darstellt.

Das `process`-Objekt ist recht umfangreich, wenn Sie seinen Funktionsumfang betrachten. Neben den bereits vorgestellten Eigenschaften, über die Sie auf Ein- und Ausgabe oder auf die Prozess-ID zugreifen können, stehen mit `argv` und `env` zwei sehr wertvolle Datenstrukturen zur Verfügung, die Sie bei der Entwicklung von Kommandozeilenwerkzeugen mit hoher Wahrscheinlichkeit benutzen werden. Die Eigenschaft `argv` bietet Ihnen ein JavaScript-Array, in dem die einzelnen Teile des Kommandozeilenaufrufs der aktuellen Applikation gespeichert sind. Angenommen, Sie rufen Ihre Applikation mit der Kommandozeile `node server.js --env=debug` auf, enthält `argv` drei Elemente. Das erste ist das `node`-Kommando selbst, das zweite ist der absolute Pfad und der Name der JavaScript-Datei, und das dritte Element ist schließlich die angegebene Option `--env=debug`. Auf diese Weise können Sie Ihre Applikation über Kommandozeilenoptionen konfigurierbar machen. `env` stellt, wie der Name bereits vermuten lässt, die Umgebungsvariablen des aktuellen Node.js Prozesses zur Verfügung. Unter den Informationen befinden sich beispielsweise der Name des Nutzers, die `PATH`-Variable oder das Heimatverzeichnis des Benutzers.

Mit dem `process`-Objekt haben Sie außerdem die Möglichkeit, sich die Version von Node.js selbst und einigen zentralen Bibliotheken anzeigen zu lassen. Sie erhalten die genaue Versionsinformation von Node.js über `process.version`. Durch `process.versions` erhalten Sie neben der Version von Node.js auch noch die entsprechenden Informationen über den `http_parser`, die V8-Engine, Ares, libuv, zlib und openssl.

Durch das `process`-Objekt können Sie jedoch nicht nur Informationen auslesen oder den Prozess beenden, sondern auch den aktuell laufenden Prozess beeinflussen. Beachten Sie, dass die folgenden Methoden nicht unter Windows verfügbar sind. Es besteht die Möglichkeit, mit `process.setuid` den Benutzer, unter dem der aktuelle Prozess läuft, zu ändern. Um den Nutzer zu verändern, benötigen Sie allerdings die Berechtigung den Nutzer zu wechseln, ansonsten erhalten Sie eine Fehlermeldung. Die aktuell gesetzte Benutzer-ID erhalten Sie durch die Methode `process.getuid`. Gleiches wie für die Benutzer-ID gilt auch für die Gruppen-ID mit den Methoden `process.setgid` und `process.getgid`.

4.2.6 Buffer

Das Buffer-Modul ist Ihnen im Verlauf dieses Buches bereits einige Male begegnet. Es wird sowohl verwendet, wenn Dateien ausgelesen werden, kann aber auch eingesetzt werden, um Daten in eine Datei zu schreiben. Wenn Daten mit dem `zlib`-Modul gepackt werden sollen, werden Buffer-Objekte verwendet, um die Daten in Octet-Streams zu kapseln und im Zusammenhang mit Datenstreams zu verwenden. Ein Buffer-Objekt hat einige wichtige Merkmale: Der Buffer hat eine feste Größe und kann nach der Erstellung nicht mehr verändert werden. Außerdem kann die Zeichencodierung der enthaltenen Daten festgelegt werden. Für die Erstellung eines Buffer-Objekts haben Sie mehrere Möglichkeiten. Ein Buffer-Objekt wird durch den Einsatz bestimmter Methoden wie beispielsweise `fs.readFile` erstellt. Sie können ein derartiges Objekt allerdings auch explizit durch den Konstruktor der Buffer-Klasse erzeugen. Dem Konstruktor kann eine Zahl übergeben werden. Dadurch wird ein leerer Buffer mit der angegebenen Anzahl von Oktetten generiert. Eine weitere Möglichkeit der Erstellung ist, dass dem Konstruktor ein Array aus Oktetten übergeben wird, die den Inhalt des Buffer-Objekts bilden. Die dritte und letzte Möglichkeit besteht schließlich darin, eine Zeichenkette und eine optionale Codierung anzugeben.

```
var myBuffer = new Buffer('Hello World'),
    yourBuffer = new Buffer(11);
myBuffer.write('World Hello World');
var length = myBuffer.length;
console.log(myBuffer.toString());
myBuffer.copy(yourBuffer);
console.log(Buffer.concat([myBuffer, yourBuffer]).toString());
```

Listing 4.33 Umgang mit Buffer-Objekten

Das Beispiel aus Listing 4.33 demonstriert Ihnen einen Teil der Methoden des Buffer-Moduls und damit einige Möglichkeiten, die Sie im Zuge der Interaktion mit Buffer-Objekten haben. Erzeugen Sie ein Buffer-Objekt, haben Sie die Möglichkeit, neue Daten in den Buffer zu schreiben. Wie bereits erwähnt, müssen Sie hier jedoch beachten, dass das Objekt eine feste Obergrenze hat. Im Beispiel liegt diese Obergrenze für das Objekt `myBuffer` bei elf Zeichen. Versuchen Sie nun, mit der `write`-Methode eine längere Zeichenkette in das Objekt zu schreiben, werden lediglich die ersten elf Zeichen übernommen. Alles, was über dieser maximalen Zeichenanzahl liegt, wird ignoriert. Würden Sie jetzt statt der zu langen Zeichenkette `World Hello World` lediglich `Earth` in den Buffer schreiben, werden durch diese Operation die ersten fünf Zeichen des Buffers überschrieben, sodass der Inhalt danach `Earth Hello` lauten würde. Die `length`-Eigenschaft gibt an, wie viele Zeichen das Buffer-Objekt aufnehmen kann. Im Beispiel liegt die Länge für beide Buffer-Objekte bei elf Zeichen. Geben Sie ein Buffer-

Objekt direkt über `console.log` oder eine andere Methode zur Ausgabe aus, erhalten Sie die interne Repräsentation des Objekts. Möchten Sie die Zeichenkette auslesen, die ein Buffer enthält, erreichen Sie dies über einen Aufruf der `toString`-Methode des Objekts. Der Rückgabewert besteht aus der entsprechenden Zeichenkette.

Mit der `copy`-Methode haben Sie außerdem die Möglichkeit, den Inhalt eines Buffer-Objekts in ein anderes zu kopieren. Auch hier sollten Sie beachten, dass die Länge des Buffer-Objekts, in das Sie kopieren möchten, die Kapazität vorgibt. Die `copy`-Methode verhält sich in diesem Fall wie auch bereits die `write`-Methode: Werte, die die maximale Kapazität überschreiten, werden abgeschnitten. Ist der Inhalt des Zielobjekts länger als der der Quelle, bleiben die übrigen Zeichen des Zielobjekts bestehen. Die `concat`-Methode ermöglicht es Ihnen schließlich, aus mehreren Buffer-Objekten ein großes Buffer-Objekt zu erstellen, indem die Buffer-Objekte aneinandergefügt werden. Die Eingabe für die `concat`-Methode besteht aus einem Array von Buffer-Objekten. Die Gesamtlänge des neuen Objekts ergibt sich aus der Summe der einzelnen Eingabeobjekte.

4.2.7 Path

Das letzte Modul, das Sie in diesem Abschnitt kennenlernen, ist das `Path`-Modul. Wie bereits angekündigt, beschäftigen Sie sich nun mit einigen Features dieses Moduls und sehen an einigen Beispielen, wie Sie es in eine Node.js-Applikation einbinden und dort verwenden können. Eine Pfadangabe im Betriebssystem ist normalerweise eine Aneinanderreihung mehrerer Verzeichnisnamen, die voneinander durch den Pfadtrenner des Betriebssystems getrennt sind. Neben normalen Verzeichnisnamen sind allerdings auch `.` und `..` gültige Verzeichnisse. Ein einzelner Punkt verweist auf das aktuelle Verzeichnis, zwei Punkte bezeichnen das Elternverzeichnis. Diese Elemente können zu Verzeichnisnamen wie `/tmp/./../tmp` führen. Um derartige Pfadangaben übersichtlicher zu gestalten, bietet Ihnen das `Path`-Modul die Methode `normalize`. Listing 4.34 zeigt Ihnen das Ergebnis der Normalisierung.

```
$ node
> require('path').normalize('/tmp/./../tmp');
'/tmp'
```

Listing 4.34 Pfadnormalisierung mit dem Path-Modul

Die `normalize`-Methode ist unabhängig vom aktuellen Arbeitsverzeichnis des Node.js-Prozesses und macht keine Unterscheidung zwischen absoluten und relativen Pfaden. Sie akzeptiert lediglich eine Zeichenkette und normalisiert diese.

Da Node.js größtenteils betriebssystemunabhängig ist, ergibt sich hier die Problemstellung, dass Applikationen, die sowohl auf Windows- als auch auf UNIX-Systemen

lauffähig sein sollen, mit dem jeweiligen Pfadtrenner des Systems arbeiten müssen. Zu diesem Zweck und um Ihnen als Entwickler unnötige Abfragen nach dem Betriebssystem zu ersparen, existiert im `Path`-Modul die Eigenschaft `sep`, die den gültigen Pfadtrenner für das aktuelle System beinhaltet.

Falls Sie einen gesamten Pfad aus mehreren Fragmenten aufbauen möchten, können Sie hierfür die `join`-Methode verwenden. Sie akzeptiert eine beliebige Anzahl von Verzeichnisnamen, verbindet diese zu einem Gesamtpfad und normalisiert das Ergebnis. In Listing 4.35 sehen Sie, wie Sie die `join`-Methode einsetzen können.

```
$ node
> var path = require('path');
undefined
> path.join(path.sep, 'home', 'root', '..', path.sep + 'sspringer');
'/home/sspringer'
```

Listing 4.35 Arbeiten mit Pfadfragmenten

Im Falle der `join`-Methode nimmt Ihnen Node.js einige Arbeit ab, so werden beispielsweise die einzelnen Pfadfragmente automatisch durch einen Pfadtrenner untereinander separiert. Durch die Normalisierung des Pfades wird auch dafür gesorgt, dass keine doppelten beziehungsweise unnötigen Pfadtrenner im Gesamtpfad vorhanden sind.

Eine weitere Navigationshilfe bietet die Methode `resolve`, mit deren Hilfe ein Ziel in einen absoluten Pfad umgewandelt werden kann. Zu diesem Zweck geben Sie einen Ausgangspfad und eine Reihe von Befehlen, mit denen Sie zum Ziel navigieren, an. Die Methode gibt im Anschluss den resultierenden Pfad zurück. Mit der `relative`-Methode erhalten Sie die Kommandos, mit denen Sie von einem absoluten Pfad zu einem anderen wechseln können. Falls Sie also die beiden Pfade `/home` und `/tmp` angeben, erhalten Sie als Ergebnis `../tmp`.

Hilfreich bei der Arbeit mit Pfaden und Dateien sind die Methoden `dirname`, `basename` und `extname`. Sie akzeptieren als Argumente jeweils eine Zeichenkette, die einen Pfad zu einer Datei beschreibt und geben den Pfad exklusiv des Dateinamens, nur den Dateinamen ohne den Pfad oder nur die Dateiendung zurück.

In den letzten Abschnitten haben Sie eine kurze Einführung in einige wichtige Module von Node.js erhalten, die Ihnen bei der Entwicklung von Applikationen für die Node.js-Plattform immer wieder begegnen werden und mit deren Hilfe Sie eine Reihe von Standardproblemen ohne großen Aufwand lösen können. Im nächsten Abschnitt beschäftigen Sie sich nun mit der Fragestellung, wie Sie Ihren eigenen Code gestalten und strukturieren, sodass er möglichst wartbar und flexibel bleibt.

4.3 Eigene Klassen erstellen und einbinden

Bisher waren die Beispiele relativ kurz und übersichtlich, und der Quellcode erstreckte sich jeweils über lediglich eine Datei. Sie haben allerdings auch schon gesehen, wie Sie zusätzliche Funktionalität in Form der Node.js-Module einbinden und so die Möglichkeiten Ihrer Applikation erheblich erweitern können. Das Einbinden von Modulen ermöglicht außerdem eine Wiederverwendung von Teilen des Quellcodes, sodass Sie keinen Code in Ihrer Applikation kopieren müssen. Was für die internen Node.js-Module gilt, ist natürlich auch für Ihren eigenen Quellcode erstrebenswert. Sobald Ihre Anwendung umfangreicher und komplexer wird und Sie erkennen können, dass sich Bereiche mit in sich geschlossener Funktionalität abzuzeichnen beginnen, mit denen über einheitliche Schnittstellen kommuniziert werden kann, ist es an der Zeit, diesen Quellcode auszulagern und als Modul einzubinden. Dies gilt noch viel mehr, wenn Sie beginnen, Quellcode innerhalb Ihrer Applikation zu kopieren. Das ist ein sicheres Anzeichen dafür, dass es sich bei dem kopierten Quellcode um Funktionalität handelt, die Sie in gleicher oder zumindest ähnlicher Form wiederverwenden können. Sie sollten dann dazu übergehen, den Quellcode in Funktionen oder Klassen auszulagern und diese zu verwenden.

Als Einstieg in dieses Thema soll Ihnen ein einfaches Beispiel dienen. Die Aufgabenstellung besteht darin, ein Kommandozeilenwerkzeug zu erstellen, dem zwei Zahlenangaben übergeben werden können. Ziel ist es nun, diese beiden Zahlen jeweils als Anzahl von Sekunden zu interpretieren und die Differenz zwischen beiden Werten zu berechnen. Soweit sollte Sie diese Aufgabe noch nicht vor größere Probleme stellen. Erschwerend kommt jedoch hinzu, dass beide Eingabewerte und die Differenz im Format »HH:MM:SS« auf der Standardausgabe ausgegeben werden sollen.

Beginnen Sie zuerst mit der formatierten Ausgabe einer Option des Kommandozeilenwerkzeugs. Listing 4.36 zeigt Ihnen, wie Sie vom Auslesen des Parameters bis zur fertigen Ausgabe gelangen.

```
var util = require('util'),
    numSecs = process.argv[2],
    hours = Math.floor(numSecs / 3600),
    minutes = Math.floor((numSecs - (hours * 3600)) / 60),
    seconds = numSecs - (hours * 3600) - (minutes * 60);

if(('' + hours).length === 1) {
    hours = '0' + hours;
}
if(('' + minutes).length === 1) {
    minutes = '0' + minutes;
}
```

```

if(('' + seconds).length === 1) {
    seconds = '0' + seconds;
}

util.puts("From: " + hours + ':' + minutes + ':' + seconds);

```

Listing 4.36 SecondsFormatter.js, der erste Ansatz

Das Script in Listing 4.36 besteht eigentlich aus drei Teilen: Im ersten lesen Sie die Daten ein und rechnen sie um, im zweiten formatieren Sie sie, und im dritten geben Sie sie schließlich aus. Im folgenden Abschnitt werden Sie sich die drei Teile kurz im Detail ansehen. Im Anschluss erweitern Sie den Code dahingehend, dass auch die übrigen Teile der Aufgabenstellung erfüllt werden können.

Der erste Codeblock der Applikation initialisiert sämtliche Variablen, die für die Applikation benötigt werden. Hierzu gehört in erster Linie das Einbinden des Util-Moduls, das Sie später zur Ausgabe benötigen. Außerdem wird die erste Option, die dem Script auf der Kommandozeile übergeben wurde, ausgelesen und der Variablen `numSecs` übergeben. Diese Variable bildet auch die Basis für die Berechnung der Stunden, Minuten und Sekunden, die später ausgegeben werden sollen. Damit die Ausgabe wie gewünscht erfolgen kann, sorgen Sie im nächsten Block dafür, dass allen Werten, die eine Länge von 1 haben, also aus nur einer Zahl bestehen, eine 0 vorangestellt wird. Die letzte Zeile sorgt schließlich mithilfe der `util.puts`-Methode dafür, dass die Zeichenkette im korrekten Format ausgegeben wird.

Wollen Sie jetzt die zweite Zahl in Ihre Applikation integrieren, können Sie den gesamten Codeblock kopieren, die einzulesende Option anpassen und bei der Ausgabe statt `From:` ein `Until:` ausgeben, und der zweite Teil der Anforderungen wäre erledigt. Für die dritte Teilaufgabe, die Berechnung und Ausgabe der Differenz, müssen Sie lediglich die zweite Kommandozeilenoption des Scripts von der ersten abziehen, die entsprechenden Codeblöcke abermals kopieren und erneut das Label anpassen, und die Aufgabe wäre damit erledigt. Wie würden Sie in diesem Fall allerdings verfahren, wenn Sie statt eines Doppelpunkts einen Bindestrich zur Trennung von Stunden, Minuten und Sekunden verwenden sollen? Sie müssten in einem solchen Fall an drei verschiedenen Stellen im Code genau die gleichen Operationen durchführen. Das bedeutet, dass die Wartbarkeit und die Erweiterbarkeit bereits in einer solch kleinen Applikation erheblich eingeschränkt werden, wenn Quellcode nicht ausgelagert und mehrfach wiederverwendet wird. Im nächsten Schritt versuchen Sie nun, den Quellcode einerseits auf eine möglichst große Wiederverwendbarkeit der einzelnen Teile hin zu optimieren und andererseits alle Anforderungen zu erfüllen.

Für die Trennung der verschiedenen Bereiche der Applikation können Funktionen verwendet werden. Durch die Verwendung von Argumenten ist es außerdem möglich, die Funktionen besser konfigurierbar zu gestalten. Eine weitere Optimierung

besteht darin, dass Sie die drei Teile in einer Klasse kapseln. Das Ziel ist es nun, eine Struktur zu schaffen, bei der dem Konstruktor eine Zahl übergeben wird und im Anschluss auf die entstandene Instanz die `toString`-Methode aufgerufen wird, die eine formatierte Zeichenkette zurückgibt.

```
var SecondsFormatter = function (numSeconds) {
  this.numSecs = numSeconds;
};
```

Listing 4.37 Konstruktor SecondsFormatter

Listing 4.37 zeigt den Konstruktor Ihrer neuen Klasse. Dieser erhält die Anzahl an Sekunden, die formatiert ausgegeben werden sollen, und speichert zu diesem Zweck den Wert in der Eigenschaft `numSecs`. Im nächsten Schritt trennen Sie die Eingangsdaten entsprechend in die einzelnen Teile auf und geben sie als Array zurück.

```
SecondsFormatter.prototype.calculateParts = function (numSecs) {
  var num = parseInt(numSecs);
  var result = [];
  result[0] = Math.floor(num / 3600),
  result[1] = Math.floor((num - (result[0] * 3600)) / 60),
  result[2] = num - (result[0] * 3600) - (result[1] * 60);
  return result;
}
```

Listing 4.38 SecondsFormatter, Aufteilung in Stunden, Minuten und Sekunden

Die Methode `calculateParts` aus Listing 4.38 kümmert sich darum, eine Anzahl von Sekunden in Stunden, Minuten und Sekunden aufzuteilen. Dazu erhält die Funktion diesen Wert als Argument. Zu Beginn wird dieser Wert in eine Ganzzahl umgerechnet, mit der im weiteren Verlauf gerechnet werden kann. Die Zerlegung in die einzelnen Bestandteile erfolgt analog zum ersten Beispiel, nur dass die einzelnen Werte nicht in separaten Variablen, sondern in einem Array gespeichert werden, da das Zusammenfügen der einzelnen Komponenten durch die Verwendung der `join`-Methode des Arrays erheblich vereinfacht wird. Als Nächstes müssen Sie dafür sorgen, dass die Formatierung der einzelnen Zahlen den Anforderungen entspricht. Das bedeutet, dass einstelligen Zahlen eine 0 vorangestellt werden muss und damit aus den übergebenen Zahlen Zeichenketten gemacht werden.

```
var util = require('util');
SecondsFormatter.prototype.padParts = function(parts) {
  if (!util.isArray(parts) && parts.length !== 3) {
    throw new Error('Not a valid parts Array!');
  }
  for (var i = 0; i < parts.length; i++) {
```

```

        if(('' + parts[i]).length === 1) {
            parts[i] = '0' + parts[i];
        }
    }
    return parts;
}

```

Listing 4.39 SecondsFormatter, Formatierung der Einzelteile

In Listing 4.39 sehen Sie die Methode `padParts`. Als Argument erhält sie ein Array, das aus drei Elementen besteht. Diese Voraussetzungen überprüfen Sie zu Beginn der Methode. Zu diesem Zweck ist es erforderlich, dass Sie das `Utility`-Modul einbinden, da Ihnen dieses mit der Methode `isArray` die Möglichkeit der Typenprüfung bietet. Sind die Voraussetzungen nicht erfüllt, werfen Sie einen Fehler. Ist der Test bestanden, iterieren Sie innerhalb der Methode über die einzelnen Teile und stellen, falls die Zahl einstellig sein sollte, eine 0 voran. Der Rückgabewert ist, wie auch schon bei der ersten Methode, ein Array aus drei Elementen.

Nachdem die beiden ersten Teile der Aufgabe bereits erledigt sind, widmen Sie sich nun dem dritten und letzten Teil: der Rückgabe des berechneten Wertes. Hier überschreiben Sie die `toString`-Methode, die jedes Objekt vom zugrunde liegenden `Function`-Objekt erbt. Diese Methode sorgt standardmäßig dafür, dass eine Zeichenkette zurückgegeben wird, die das aktuelle Objekt repräsentiert. In Ihrem Fall wäre das die Zeichenkette `HH:MM:SS`, bestehend aus den jeweils berechneten Werten.

```

SecondsFormatter.prototype.toString = function() {
    var parts = this.calculateParts(this.numSecs),
        parts = this.padParts(parts);
    return parts.join(':');
}

```

Listing 4.40 `toString`-Methode der `SecondsFormatter`-Klasse

Die `toString`-Methode aus Listing 4.40 benötigt keine Argumente, da sie den Wert aus der Eigenschaft `numSecs` des Objekts ausliest. Intern ruft die `toString`-Methode zuerst die Methode `calculateParts` auf und leitet den erhaltenen Rückgabewert an die Methode `padParts` weiter. Das Ergebnis wird schließlich mit der `join`-Methode mit dem Doppelpunkt als Trenner zusammengefügt und zurückgegeben. Fügen Sie nun die einzelnen Teile zusammen, erhalten Sie eine Klasse, mit deren Hilfe es möglich ist, aus einer Anzahl von Sekunden eine formatierte Zeichenkette zu generieren. Haben Sie erst einmal die Klasse `SecondsFormatter` implementiert, stehen nur noch wenige Zeilen Code zwischen Ihnen und der Lösung der Aufgabe.

```

if (process.argv.length !== 4) {
    throw new Error('Usage: node tracker.js <from> <to>');
}

var from = new SecondsFormatter(process.argv[2]),
    to = new SecondsFormatter(process.argv[3]),
    diff = new SecondsFormatter(process.argv[3] - process.argv[2]);

util.puts("From: " + from.toString());
util.puts("Until: " + to.toString());
util.puts("Difference: " + diff.toString());

```

Listing 4.41 Anwendung der SecondsFormatter-Klasse

In Listing 4.41 sehen Sie die letzten Schritte, die erforderlich sind, um die Applikation zu vollenden. Im ersten Schritt verwenden Sie das `process`-Objekt, um zu prüfen, ob der Anwendung die dafür nötigen Optionen zur Verfügung gestellt wurden. In diesem Fall muss der Verwender zwei Werte angeben, einen Startwert und einen Endwert. Mit diesen Werten erstellen Sie im Anschluss drei `SecondsFormatter`-Objekte. Eines mit dem Startwert, eines mit dem Endwert und eines mit der Differenz. Der letzte Schritt besteht darin, die drei Angaben mithilfe einer Kombination aus `util.puts` und der `toString`-Methode der Objekte auszugeben.

Ändern sich nun die Anforderungen, ist es für Sie ein Leichtes, die Anwendung entsprechend anzupassen. Soll das Ergebnis jetzt statt im Format »HH:MM:SS« in der Form »HH-MM-SS« ausgegeben werden, also mit Bindestrichen statt Doppelpunkten, können Sie dies mit der Änderung eines Zeichens im Quellcode erreichen. Es besteht allerdings auch die Möglichkeit, die `toString`-Methode zu parametrisieren und so den Trenner konfigurierbar zu machen. In Listing 4.42 sehen Sie die notwendigen Anpassungen der `toString`-Methode.

```

SecondsFormatter.prototype.toString = function(glue) {
    var parts = this.calculateParts(this.numSecs),
        parts = this.padParts(parts);
    glue = glue || ':';
    return parts.join(glue);
}

```

Listing 4.42 Anpassung der toString-Methode

Anhand des gezeigten Beispiels beschäftigen Sie sich im Folgenden näher mit der Strukturierung von Applikationen und den Möglichkeiten, die Ihnen die Node.js-Plattform in diesem Zusammenhang bietet.

4.3.1 Eigene Module in Node.js

Werden Applikationen umfangreicher, empfiehlt es sich, den Quellcode in logischen Einheiten zu strukturieren und diese in separaten Dateien abzulegen. Durch eine derartige Vorgehensweise wird der modulare Charakter einer Applikation deutlicher und die Kopplung der einzelnen Teile loser. Ziel ist es, einzelne Teile der Applikation so zu abstrahieren, dass der Code nicht nur in der einen Anwendung verwendet werden kann, sondern die Problemlösung so allgemeingültig ist, dass Sie sie auch in anderen Projekten wiederverwenden können. Dies trifft weniger auf applikationsspezifische Speziallösungen zu als auf Lösungen für allgemeingültigere Aufgabenstellungen. Die Entwickler der Node.js-Plattform haben für genau dieses Problem das `Modules`-Modul entwickelt. Im Kern ist das der Modulloader von Node.js. Ein Modul ist dabei eine relativ in sich geschlossene logische Einheit, die als Teil einer Applikation verwendet werden kann. Module werden in separaten Dateien abgelegt und stellen ihre Funktionalität über Schnittstellen zur Verfügung. In der Praxis bedeutet das, dass Module meistens Klassen oder Objekte sind, die in eine Applikation eingebunden werden. Ein gutes Beispiel liefern die Module, die Ihnen Node.js im Rahmen der Plattform zur Verfügung stellt. Sie funktionieren nach dem gleichen Prinzip und sind mit der `Buffer`-Klasse oder dem `util`-Objekt in beiden Varianten vertreten. Für das Beispiel bedeutet das, dass Sie die Klasse `SecondsFormatter` in eine separate Datei auslagern können und diese als Modul einbinden können. Alles, was Sie dafür tun müssen, ist das `require`-Statement, den Konstruktor und alle Methoden, die dem Prototyp hinzugefügt wurden, in eine separate Datei zu verschieben. Damit allein funktioniert das Modul allerdings noch nicht. Dafür sind noch ein paar Änderungen erforderlich. Die Basis für diese Änderungen werden Sie sich in den folgenden Abschnitten ansehen.

4.3.2 Eigene Node.js-Module

Die Node.js-Plattform bietet Ihnen als Entwickler die Möglichkeit, über das globale Objekt `exports` Funktionalität aus einem Modul zur Verfügung zu stellen. Dieses Objekt ist zwar in jedem Modul gleichermaßen verfügbar, jedoch stets nur für das aktuelle Modul. Das bedeutet, dass nicht mehrere Module das gleiche `exports`-Objekt verwenden und sich über dieses Funktionalität teilen können.

```
var ignore = /[\. ,]/g,
    separator = " ";
var wordCount = function (sentence) {
    var wordCount = {},
        words = sentence.replace(ignore, '')
            .toLowerCase().split(separator);
    for (var i in words) {
        wordCount[words[i]] = wordCount[words[i]] + 1 || 1;
    }
}
```



```

    return wordCount;
}
exports.wordCount = wordCount;

```

Listing 4.43 WordCount.js

In Listing 4.43 sehen Sie eine sehr einfache Variante eines Node.js-Moduls. Der Zweck dieses Moduls ist es, zu zählen, wie oft Worte in einer Zeichenkette verwendet werden. Dabei werden Punkt und Komma wie auch Groß- und Kleinschreibung ignoriert. Als Worttrenner wird das Leerzeichen angenommen. Diese Konfiguration speichern Sie am Anfang der Datei in Variablen. Diese sind nur für das Modul global und nicht von anderen Stellen in der Anwendung erreichbar. Die Funktion `wordCount` akzeptiert eine Zeichenkette als Argument, aus der im weiteren Verlauf Punkte und Kommas entfernt und sämtliche Großbuchstaben in Kleinbuchstaben umgewandelt werden. Danach wird die Zeichenkette an jedem Leerzeichen in einzelne Teile aufgesplittet, die in einem Array vorgehalten werden. Die einzelnen Worte dienen in einem Objekt als Schlüssel. Die Anzahl der Worte wird in einer Schleife ermittelt und zum jeweiligen Schlüssel gespeichert. Die Objektstruktur wird im Anschluss zurückgegeben. Insofern unterscheidet sich der Quellcode noch nicht von den Beispielen, die Sie bisher betrachtet haben. Die letzte Zeile mit der Anweisung `exports.wordCount = wordCount` sorgt allerdings dafür, dass die Funktion `wordCount` verfügbar gemacht wird, wenn Sie das Modul mit der `require`-Funktion in Ihre Applikation einbinden.

```

var wc = require('./WordCount.js'),
    util = require('util'),
    sentence = 'Wo viel Licht ist, ist auch viel Schatten.',
    wordCount = wc.wordCount(sentence);
util.puts(sentence);
for (var i in wordCount) {
    util.puts(wordCount[i] + ' x ' + i);
}

```

Listing 4.44 app.js – Einsatz des WordCount-Moduls

Der Quellcode aus Listing 4.44 zeigt Ihnen, wie Sie das eben erstellte Modul verwenden können. Mit der Anweisung `wc = require('./WordCount.js')` binden Sie das Modul in die aktuelle Anwendung ein. Dabei stehen Ihnen sämtliche Funktionen und Objekte zur Verfügung, die Sie im Modul dem `exports`-Objekt als Eigenschaft zugewiesen haben. Um die Schnittstellen Ihres Moduls verfügbar zu machen, können Sie entweder das `exports`-Objekt, wie im Beispiel gezeigt, verwenden oder die längere Variante `module.exports` benutzen. Die Funktionalität ist in beiden Fällen die gleiche.

Wie Sie bereits in Listing 4.44 sehen konnten, werden lediglich die Inhalte und Schnittstellen eines Moduls veröffentlicht, die über das `exports`-Objekt verfügbar gemacht

werden. Sämtliche andere Variablen, Funktionen oder Klassen, die innerhalb eines Moduls definiert werden, sind auch nur dort verfügbar. Sie müssen sich in diesem Fall also nicht darum kümmern, den globalen Scope Ihrer Applikation sauber zu halten.

4.3.3 Das Modules-Modul

Die beiden wichtigsten Komponenten des `Modules`-Moduls, also des Modulloaders von Node.js, sind auf der einen Seite das `exports`-Objekt, mit dessen Hilfe Sie Module veröffentlichen, und auf der anderen Seite die `require`-Methode, mit der Sie das Modul einbinden können. Neben diesen beiden Kernfeatures bietet Ihnen dieses Modul jedoch noch eine Reihe weiterer Funktionalitäten, die bei der Entwicklung und im Betrieb von Modulen sehr hilfreich sein können. Zu diesem Zweck stellt Ihnen die Node.js-Plattform das `module`-Objekt global zur Verfügung, Sie müssen es also nicht erst über die `require`-Methode verfügbar machen. Innerhalb eines Moduls können Sie auf Reihe von Eigenschaften des `Module`-Moduls zurückgreifen, mit denen Sie verschiedene Informationen über das aktuelle Modul abfragen können. Allen voran seien hier die Eigenschaften `module.id` und `module.filename` genannt. Diese beiden Eigenschaften geben im Normalfall den absoluten Dateinamen des Moduls an. Falls Sie auf dieser Information aufbauen möchten, sollten Sie die zweite Variante, also `module.filename`, verwenden. Die Eigenschaft `module.id` enthält den Wert `».«`, wenn Sie außerhalb eines Moduls, also beispielsweise in der Hauptdatei Ihrer Applikation, auf sie zugreifen. Auch im Node.js REPL können Sie auf diese beiden Eigenschaften des `module`-Objekts zugreifen. Die `id`-Eigenschaft weist in diesem Fall den Wert `repl` auf, und der `filename` besteht aus dem aktuellen Pfad und trägt den Dateinamen `repl`.

Über die Eigenschaft `loaded` des `module`-Objekts können Sie herausfinden, ob das aktuelle Modul bereits fertig geladen ist. In diesem Fall ist der Wert `true`. Falls das aktuelle Modul noch nicht geladen wurde oder sich noch im Ladeprozess befindet, ist der Wert dieser Eigenschaft `false`.

Wird eine Node.js-Applikation gestartet, wird die Eigenschaft `require.main` automatisch mit dem `module`-Objekt der Einstiegsdatei gefüllt. Das bedeutet, dass Sie den Dateinamen der Datei, der als Einstiegspunkt in Ihre Applikation dient, in jedem Modul über `require.main.filename` erreichen können.

Das `Module`-Modul hält neben diesen Informationen auch die Daten über die Beziehungen zwischen den Modulen vor, also welches Modul durch welches andere geladen wurde beziehungsweise welche Module durch ein bestimmtes Modul geladen werden. In der Eigenschaft `parent` des `module`-Objekts stehen Ihnen die Informationen über das Modul zur Verfügung, das das aktuelle Modul geladen hat. Diese Informationen liegen wiederum in Form eines `module`-Objekts vor. Dies bedeutet, dass Sie unter anderem Zugriff auf die Eigenschaften `id` oder `filename` des Elternmoduls haben. Die Eigenschaft `children` enthält ein Array von Modulen, die das aktuelle

Modul per `require` lädt. Das gilt allerdings nur für eigene Module. Die Module, die durch die Node.js-Plattform selbst zur Verfügung gestellt werden, werden nicht im `children`-Array aufgeführt. Die Objekte in dieser Datenstruktur weisen den gleichen Aufbau auf wie auch schon das `parent`-Objekt.

4.3.4 Der Modulloader

Das Einbinden von Modulen geschieht über die `require`-Methode. Zu diesem Zweck übergeben Sie, wie Sie bereits im Beispiel in Listing 4.44 sehen konnten, den Namen der Datei, die den Quellcode des zu ladenden Moduls enthält. Dabei können Sie den Dateinamen sowohl absolut als auch relativ zur aktuellen Datei angeben.

```
$ ls /srv/node/wordCount
WordCount.js app.js
```

Listing 4.45 Verzeichnislisting der WordCount-Applikation

Angenommen, Sie haben eine Verzeichnisstruktur, wie sie in Listing 4.45 zu sehen ist, also die zwei Dateien *WordCount.js*, die den Code des Moduls enthält, und *app.js*, der Einstiegspunkt für die Applikation. Die Anwendung können Sie mit dem Kommando `node app.js` starten. Das Modul können Sie entweder über `require('./WordCount.js')` wie in Listing 4.44 starten oder mit dem absoluten Pfad, also mithilfe von `require('/srv/node/wordCount/WordCount.js')`. Wollen Sie Ihre Applikation unabhängig von Ihrer aktuellen Verzeichnisstruktur betreiben, sollten Sie die relative Angabe von Modulen wählen. Dies ermöglicht es Ihnen, die Applikation auch auf anderen Systemen zu betreiben oder sie anderen Personen zur Verfügung zu stellen.

node_modules

Stellen Sie dem Modul, das Sie laden wollen, weder `/` noch `./` voran, lädt der Modulloader entweder ein internes Modul der Node.js-Plattform oder sucht im Verzeichnis *node_modules* nach einem Modul mit dem entsprechenden Namen. Der Modulloader beginnt mit der Suche im *node_modules*-Verzeichnis im aktuellen Verzeichnis. Wird das Modul dort nicht gefunden, wird eine Ebene höher versucht das Modul zu finden. In der Verzeichnisstruktur aus Listing 4.45 bedeutet das konkret, dass zuerst im Verzeichnis */srv/node/wordCount/node_modules*, danach in */srv/node/node_modules* und so weiter gesucht wird, bis der Modulloader im Wurzelverzeichnis angekommen ist. Wird das Modul in dieser Verzeichnisstruktur nicht gefunden, sucht der Modulloader in den Verzeichnissen, die in der Umgebungsvariablen `NODE_PATH` angegeben sind. Außerdem wird in den Verzeichnissen *<home>/node_modules*, *<home>/node_libraries* und *<install-prefix>/lib/node* gesucht. Beim Laden von Modulen durch die `require`-Methode sucht der Modulloader nicht nur nach der angegebenen Datei, sondern sucht, falls keine Dateieindung angegeben ist, nach dem

Dateinamen und hängt die Endungen *.js*, *.json* und *.node* an. Diese Art der Suche nach Modulen ermöglicht es Ihnen, verschiedene Versionen von Node.js-Modulen auf Ihrem System zu installieren und über die Platzierung der jeweiligen Module zu steuern, welche Version der Modulloader laden soll. So können Sie Versionskonflikte vermeiden, die dadurch entstehen, dass eine andere Applikation oder eine Bibliothek eine andere Version eines Moduls benötigt.

Je nachdem, wo Sie die Module speichern, können Sie die Geschwindigkeit des Ladeprozesses beeinflussen. Module, die im lokalen *node_modules*-Verzeichnis abgelegt werden, werden am schnellsten geladen, da ein langwieriger Suchprozess vermieden wird.

Die folgende Auflistung fasst Ihnen noch einmal kurz zusammen, in welchen Verzeichnissen im Beispiel nach Modulen gesucht wird.

- ▶ */srv/node/wordCount/node_modules*
- ▶ */srv/node/node_modules*
- ▶ */srv/node_modules*
- ▶ */node_modules*
- ▶ */home/sspringer/.node_modules*
- ▶ */home/sspringer/.node_modules*
- ▶ */usr/local/lib/node_modules*

Auf diesem System ist die Umgebungsvariable *NODE_PATH* leer, dadurch wird also kein zusätzlicher Pfad hinzugefügt. Mit dem Modulloader von Node.js gibt es noch eine weitere Art, um Module in Ihre Applikation einzubinden. Statt des Namens eines Moduls können Sie auch ein ganzes Verzeichnis angeben. Das hat den Hintergrund, dass häufig Bibliotheken in eigenen Verzeichnisstrukturen abgelegt werden. Für diese Module gibt es eine definierte Namenskonvention, wie die Dateien zu benennen sind, damit der Modulloader den Quellcode finden und korrekt einbinden kann. Übergeben Sie dem Modulloader ein Verzeichnis statt eines Dateinamens, versucht dieser im ersten Schritt, die Datei mit dem Namen *package.json* zu finden. Diese Datei wird Ihnen im Verlauf dieses Kapitels erneut begegnen, wenn es um den Package Manager der Node.js-Plattform und dessen Module geht. Hier sei vorab nur so viel gesagt: Bei dieser Datei handelt es sich um die Konfiguration und interne Beschreibung von Modulen. Sie können diese Datei verwenden, um auf den Einstiegspunkt eines Moduls innerhalb eines Verzeichnisses zu verweisen. Um diesen Fall zu demonstrieren, passen Sie das Beispiel mit dem Wortzähler etwas an. Zuerst müssen Sie ein Verzeichnis mit dem Namen *WordCount* erstellen. Dieses soll später Ihr Modul enthalten. In dieses Verzeichnis kopieren Sie dann die Datei *WordCount.js*, die den Quellcode des Moduls enthält, in ein Unterverzeichnis *lib*. Anschließend erstellen Sie im Verzeichnis *WordCount* eine Datei mit dem Namen *package.json*. Der Inhalt dieser Datei entspricht dem Quellcode aus Listing 4.46.

```
{
  "name": "WordCount",
  "main": "./lib/WordCount.js"
}
```

Listing 4.46 package.json für das WordCount-Modul

Die letzte Anpassung müssen Sie in der *app.js*, also im Einstiegspunkt der Applikation, vornehmen. Hier müssen Sie lediglich dafür sorgen, dass das *WordCount*-Modul über den Befehl `var stats = require('./WordCount')` eingebunden wird. Sollten Sie ein Verzeichnis angeben, wenn Sie ein Modul laden möchten, und dieses weist keine *package.json*-Datei auf, sucht der Modulloader alternativ nach den Dateien *index.js* und *index.node* als Einstiegspunkt in das Modul. Innerhalb Ihrer Applikation haben Sie die Möglichkeit, ein Modul nicht nur einmal zu Beginn einer Quellcodedatei oder auch nur in einer einzigen Datei zu laden, sondern Sie können es beliebig oft und an verschiedensten Stellen und somit auch mehrfach laden. Damit diese Module nicht mehrfach eingelesen werden müssen, nimmt der Modulloader eine Optimierung vor, indem das Modul zwischengespeichert wird. Das bedeutet, dass das Modul nur beim ersten Mal geladen und ausgeführt werden muss. Die übrigen Aufrufe von `require` mit dem Namen des Moduls werden dann aus dem Cache bedient. Dieses Feature hat zur Konsequenz, dass der Quellcode auch nur beim ersten `require` ausgeführt wird. Falls Sie bei jedem Aufruf einen Seiteneffekt erzielen möchten, müssen Sie dies über Funktionen erzwingen. In Listing 4.47 und Listing 4.48 sehen Sie, wie sich dieses Verhalten nachstellen lässt.

```
console.log('myModule called');
```

Listing 4.47 myModule.js

```
require('./myModule.js');
require('./myModule.js');
```

Listing 4.48 app.js

Wenn Sie den Quellcode mit dem Befehl `node app.js` ausführen, erhalten Sie die Ausgabe *myModule called*, und zwar nur einmal, da der zweite Aufruf von `require` direkt aus dem Cache bedient wird und die Datei *myModule.js* kein zweites Mal ausgeführt wird. Um beim zweiten Aufruf ebenfalls einen Effekt zu erzielen, müssen Sie den Code wie in Listing 4.49 und Listing 4.50 beschrieben umbauen.

```
exports.myFunc = function() {console.log('myModule called');};
```

Listing 4.49 myModule.js

```
require('./myModule.js').myFunc();
require('./myModule.js').myFunc();
```

Listing 4.50 app.js

Die Datei *myModule.js* exportiert die Funktion, die für die Ausgabe sorgt, in die Eigenschaft *myFunc*. Diese wird innerhalb der Datei *app.js* zweimal aufgerufen und sorgt so dafür, dass *myModule* *called* zweimal ausgegeben wird.

4.3.5 Die require-Funktionalität

Der Modulloader von Node.js ist relativ einfach aufgebaut. Dennoch kann es bei Applikationen mit vielen Modulen schwierig werden, den Überblick zu behalten. Herausfordernd ist außerdem der Umgang mit unterschiedlichen Versionen bestimmter Module auf einem System. Auch der Caching-Mechanismus des Modulloaders kann unter Umständen zu einem Problem werden. Neben den bisher vorgestellten Funktionalitäten des Node.js-Modulsystems existieren noch weitere Features, die Ihnen den Umgang mit Modulen erleichtern können.

Mit der *resolve*-Methode des *require*-Objekts sind Sie in der Lage herauszufinden, in welcher Datei ein bestimmtes Modul liegt. Zu diesem Zweck müssen Sie der Methode lediglich die Information mitgeben, welches Modul Sie laden möchten. Das Benennungsschema gleicht dem, das bei den normalen Ladeoperationen auch verwendet wird. Die *resolve*-Methode führt auch die gleichen Suchoperationen durch wie das eigentliche *require*, mit dem Unterschied, dass das Modul nicht geladen wird, sondern der absolute Pfad der Datei, in der sich das Modul befindet, als Zeichenkette zurückgegeben wird. Für interne Module gibt *resolve* lediglich den Namen des Moduls zurück. Diese Funktionalität ist außerdem nicht nur in Applikationen verfügbar, sondern auch im Node.js REPL.

Den Modulcache von Node.js haben Sie bereits kennengelernt. Das *require*-Objekt stellt Ihnen mit dem *cache*-Objekt eine Möglichkeit zur Verfügung, diesen Cache auszulesen und zu bearbeiten. Die Eigenschaft *cache* des *require*-Objekts ist eine Hashmap, also ein JavaScript-Objekt, in der mithilfe eines Key-Value-Stores Daten gespeichert werden. Die Schlüssel bestehen aus den absoluten Dateinamen der jeweiligen Module, der Wert ist jeweils das *module*-Objekt des Moduls. So bietet Ihnen das *cache*-Objekt die Möglichkeit, zu prüfen, ob ein Modul innerhalb Ihrer Applikation bereits geladen wurde. Einen wichtigen Punkt müssen Sie dabei allerdings beachten, und zwar werden im Cache keine internen Node.js-Module aufgeführt. Im Beispiel zum Modulcache in Listing 4.47 und Listing 4.48 haben Sie gesehen, dass ein Modul lediglich einmal evaluiert wird und beim zweiten Aufruf aus dem Cache bedient wird. Mit dem *cache*-Objekt haben Sie die Möglichkeit, dieses Verhalten zu beeinflussen.

```
require('./myModule.js');
delete(require.cache[require.resolve('./myModule.js')]);
require('./myModule.js');
```

Listing 4.51 app.js

Löschen Sie, wie in Listing 4.51 zu sehen, den Cacheeintrag für das Modul nach dem ersten `require`-Aufruf, wird das Modul komplett aus dem Cache entfernt, was dazu führt, dass es beim nächsten Aufruf erneut geladen und evaluiert wird und ein etwaiger Seiteneffekt, der durch das Modul erzielt werden soll, erneut auftritt. Nicht nur der Cache, sondern auch die Art und Weise, wie Node.js die Module lädt, die Sie über `require` einbinden, lässt sich beeinflussen. Das `require`-Modul verfügt zu diesem Zweck über die Eigenschaft `extensions`. Der Wert ist ein Objekt, bei dem die Schlüssel aus verschiedenen Dateiendungen und die zugehörigen Werte aus den Funktionen bestehen, die verwendet werden, um die Dateien mit den entsprechenden Endungen zu laden. Möchten Sie nun die Unterstützung für eine weitere Dateiendung hinzufügen, müssen Sie das `extensions`-Objekt lediglich um einen Schlüssel erweitern, der den Namen der Dateiendung trägt, und eine Funktion hinzufügen, die dafür zuständig ist, die Dateien zu laden. Die einfachste Möglichkeit hierfür besteht darin, einfach eine der existierenden Funktionen zu verwenden. Im nächsten Abschnitt werden Sie eine etwas umfangreichere Applikation umsetzen, in der Sie die verschiedenen Aspekte von Node.js in Aktion erleben können.

4.3.6 Die Time-Tracker-Applikation

Die Aufgabenstellung besteht darin, einen Time-Tracker zu implementieren. Ziel dieser Applikation ist es, einen Nutzer in die Lage zu versetzen, auf der Kommandozeile den Beginn und das Ende seiner Arbeitszeit zu erfassen und zudem anzugeben, welche Tätigkeit er in dieser Zeit erledigt hat. In Listing 4.52 sehen Sie, wie die Applikation verwendet werden kann.

```
$ node index.js start 'Implementierung der Timetracker Applikation'
$ node index.js ende
$ node index.js statistik 11.2012
```

Listing 4.52 Verwendung der Time-Tracker-Applikation

Über die Kommandozeile können Sie sowohl den Beginn als auch das Ende einer Aufgabe festlegen. Diese Informationen werden in einer Textdatei festgehalten. Auf Basis der Daten aus der Textdatei können Sie monatsweise Statistiken generieren lassen. Die Anwendung funktioniert, indem ein Nutzer Daten zur Verfügung stellt, die im Verlauf der Applikation verarbeitet werden. Die Interaktion mit der Applikation erfolgt über Optionen, die auf der Kommandozeile übergeben werden. Das bedeutet,

dass dies für Sie der erste Ansatzpunkt für die Implementierung ist. In Listing 4.53 sehen Sie den Quellcode, der dafür sorgt, dass die Optionen korrekt eingelesen und in einer Objektstruktur gespeichert werden.

```
var in_array = function(needle, haystack) {
    for(var i = 0; i < haystack.length; i++) {
        if (haystack[i] === needle) {
            return true;
        }
    }
    return false;
}

var availableModes = ['start', 'ende', 'statistik'];
exports.getData = function() {
    var result = {},
        mode = process.argv[2];

    if (in_array(mode, availableModes)) {
        result['mode'] = mode;
    } else {
        throw new Error('Verfügbare Modi: ' +
            availableModes.join(', '));
    }

    switch (mode) {
        case 'start':
            if (process.argv.length < 4) {
                throw new Error('Bitte Aufgabe angeben.');
            }
            result['task'] = process.argv[3];
            break;
        case 'statistik':
            if (process.argv.length < 4) {
                throw new Error('Bitte Zeitraum angeben.');
            }
            result['period'] = process.argv[3];
            break;
    }
    return result;
}
```

Listing 4.53 `getData.js` – Daten aus Kommandozeilenoptionen auslesen

Für die Aufgabenlösung benötigen Sie zwei Voraussetzungen. Zum einen implementieren Sie die Funktion `in_array`, die prüft, ob ein Wert in einem Array vorhanden ist,

zum anderen ein Array mit gültigen Werten für die verschiedenen Modi der Applikation. Der Kern dieses Moduls besteht aus einer Funktion, die Sie in der Eigenschaft `exports.getData` exportieren und so zur Verfügung stellen. Damit Sie auf die einzelnen Optionen zugreifen können, verwenden Sie das `process`-Objekt und hier speziell die `argv`-Eigenschaft. Die einzelnen Optionen werden entsprechend gewisser Regeln validiert. So kann ein Benutzer lediglich einen von drei Modi wählen. Beim Status `start` und `statistik` werden außerdem weitere Optionen eingelesen. Die Resultate speichern Sie in einem Objekt, das am Ende der Funktion zurückgegeben wird. Im nächsten Schritt binden Sie das Modul nun, wie Sie in Listing 4.54 sehen können, in die `index.js`-Datei, den zentralen Einstiegspunkt der Applikation, ein.

```
var getData = require('./lib/getData.js');
console.log(getData.getData());
```

Listing 4.54 Einbinden des `getData`-Moduls in die `index.js`-Datei

In diesem ersten Teil der Applikation setzen Sie auf einige bekannte Komponenten. Sie übergeben der `require`-Methode den relativen Pfad zu Ihrem Modul, das neben Hilfskonstrukten, die nach außen nicht sichtbar sind, eine Funktion beinhaltet, die mithilfe des `exports`-Objekts über den Rückgabewert der `require`-Methode verfügbar gemacht wird. Der Aufruf der `getData`-Funktion sorgt schließlich dafür, dass die Kommandozeilenoptionen eingelesen und in einer Objektstruktur verfügbar gemacht werden. Durch das `console.log`-Kommando geben Sie dieses Objekt auf der Konsole aus.

Mit dem bisherigen Quellcode der Applikation sind Sie in der Lage, die vom Benutzer übergebenen Optionen auszulesen. Diese wollen Sie jetzt nutzen, um aus diesen Informationen ein Objekt zu generieren, mit dem Sie innerhalb der Applikation arbeiten können. Da diese Objekte stets den gleichen Aufbau haben, empfiehlt es sich, zu diesem Zweck eine neue Klasse einzuführen. Listing 4.55 zeigt Ihnen den Aufbau der `TimeTracker`-Klasse, die Sie, wie auch schon die `getData`-Funktion, in ein eigenes Modul auslagern sollten.

```
var timeHelper = function (dateObj) {
    return dateObj.getHours() + ':' + dateObj.getMinutes();
}

var TimeTracker = function () {
    this._date;
    this._startTime;
    this._endTime;
    this._task;
}

TimeTracker.prototype = {
```

```

    get date () {return this._date;},
    set date (date) {this._date = date; return this;},
    get startTime () {return this._startTime;},
    set startTime (startTime) {this._startTime = startTime; return this;},
    get endTime () {return this._endTime;},
    set endTime (endTime) {this._endTime = endTime; return this;}
}

Object.defineProperty(TimeTracker.prototype, "task", {
  get: function() {return this._task;},
  set: function(task) {this._task = task; return this;}
});

TimeTracker.prototype.start = function (task) {
  var now = new Date();
  this.date = now.getDate() + '.' + (now.getMonth() + 1) + '.' +
    now.getFullYear();
  this.startTime = timeHelper(now);
  this.task = task;
};

TimeTracker.prototype.end = function () {
  var now = new Date();
  this.endTime = timeHelper(now);
}

TimeTracker.prototype.toString = function () {
  var result = this.date + ' ';
  result += this.startTime + ' ';
  result += this.endTime + ' ';
  result += this.task;
  return result;
}

module.exports = TimeTracker;

```

Listing 4.55 tracker.js – das TimeTracker-Modul

Das TimeTracker-Modul stellte eine weitere Möglichkeit vor, wie Sie ein Modul in Node.js gestalten können. Im Gegensatz zu `getData`, bei dem Sie eine Funktion exportierten, wird hier eine vollwertige Klasse erstellt und zur weiteren Verwendung exportiert. Die Grundlage für dieses Modul bildet die Konstruktor-Funktion `TimeTracker`, die am Ende des Moduls über das `exports`-Objekt nach außen verfügbar

gemacht wird. In der Konstruktor-Funktion werden die Eigenschaften definiert, die später durch die Methoden verwendet werden. Der Kern der Node.js-Plattform, die V8-Engine, ist in der Lage, Features aus dem ECMAScript-Standard in der Version 5 zu unterstützen. So sehen Sie, wie Sie beispielsweise Getter- und Setter-Methoden innerhalb Ihrer Klasse verwenden können.

Eine Möglichkeit, die Getter- und Setter-Methoden zu definieren, besteht darin, diese direkt im `prototype`-Objekt der Konstruktor-Funktion zu definieren. Mit der Anweisung `get <Variable> () {}` und `set <Variable> () {}` können Sie für die entsprechende Variable eine Getter- beziehungsweise Setter-Methode erstellen. Der Getter wird aufgerufen, sobald Sie auf die entsprechende Eigenschaft lesend zugreifen. Der Setter kommt zum Einsatz, wenn Sie versuchen, der Eigenschaft durch den `=`-Operator einen neuen Wert zuzuweisen. Innerhalb der geschweiften Klammern können Sie die jeweilige Logik implementieren, die zum Lesen oder Schreiben verwendet werden soll. Die jeweilige Funktion ist an den Kontext des Objekts der Klasse, in diesem Fall also an ein konkretes `TimeTrack`-Objekt, gebunden, sodass Sie über `this` Zugriff auf das Objekt selbst haben. Bei der Verwendung von Setter-Methoden sollten Sie beachten, dass die Variable, in der Sie den Wert speichern möchten, einen anderen Namen tragen muss als der Name, an den die Setter-Methode gebunden ist, da ansonsten eine Rekursion entsteht, die von Node.js nach etlichen Durchläufen mit einem `RangeError` beendet wird. Diese Rekursion entsteht dadurch, dass Sie innerhalb des Setters versuchen, eine Zuweisung auf eine Variable vorzunehmen, die wiederum den Aufruf des Setters verursacht.

Die zweite Möglichkeit, mit der Sie Getter- und Setter-Methoden definieren können, ist die Verwendung von `Object.defineProperty`. Auch diese Methode wurde im Rahmen des ECMAScript-Standards in der Version 5 neu eingeführt und dient dazu, neue Eigenschaften für ein Objekt zu definieren. Die Methode akzeptiert drei Argumente. Das erste ist das betroffene Objekt, im Falle des Beispiels also der Prototyp der Klasse `TimeTracker`. Das zweite Argument besteht aus einer Zeichenkette, die den Namen der Eigenschaft angibt. Das dritte Argument schließlich ist ein Objekt mit der Konfiguration für die Eigenschaft. Hier sollten Sie zwischen Datenbeschreibungen und Zugriffsbeschreibungen unterscheiden. Im Falle des Beispiels handelt es sich mit den Getter- und Setter-Methoden um Zugriffsbeschreibungen. Für diese können Sie die Schlüsselworte `configurable`, `enumerable`, `get` und `set` verwenden, wobei die beiden Letztgenannten in Listing 4.55 zum Einsatz kommen.

Neben dem Konstruktor und den Getter- und Setter-Methoden implementieren Sie für Ihre `TimeTracker`-Klasse noch drei weitere Methoden, die Ihnen eine Schnittstelle für Ihre Objekte bieten. Über die `start`-Methode lässt sich ein neuer Datensatz initialisieren und mit Werten für `_date`, `_startTime` und `_task` befüllen, wobei für die Datums- und Zeitangaben der aktuelle Zeitpunkt verwendet wird. Lediglich die Beschreibung der Aufgabe wird vom Benutzer angegeben. Die `end`-Methode dient

zum Abschluss eines Datensatzes und sorgt dafür, dass ein Wert für die `_endTime`-Eigenschaft eingetragen wird. Auch hier wird wieder der Zeitpunkt des Funktionsaufrufs verwendet. Die `toString`-Methode dient wie üblich dazu, eine Repräsentation des Objekts in Form einer Zeichenkette zu erhalten. Schließlich binden Sie Ihre Klasse in die Applikation ein und testen diese. Dazu fügen Sie den Codeblock aus Listing 4.56 in die `index.js`-Datei ein.

```
var Tracker = require('./lib/tracker.js');
var myTracker = new Tracker();
myTracker.start('Implementierung des TimeTrackers');
myTracker.end();
console.log(myTracker.toString());
```

Listing 4.56 Erweiterung der `index.js`-Datei

In Listing 4.56 binden Sie im ersten Schritt das `TimeTracker`-Modul ein und speichern die Klasse in der Variablen `Tracker`. Aus dieser Klasse erstellen Sie ein konkretes Objekt und speichern dieses in der Variablen `myTracker`. Danach rufen Sie nacheinander erst die Methode `start`, dann `end` und schließlich die `toString`-Methode auf, deren Ergebnis Sie mit `console.log` auf der Konsole ausgeben. Nachdem Sie die beiden Komponenten `getData` und `TimeTracker` implementiert haben, können Sie sich nun der Aufgabe widmen, beide miteinander zu verbinden. Die überarbeitete `index.js`, die Einstiegsdatei in die Applikation, sehen Sie in Listing 4.57.

```
var options = require('./lib/getData.js').getData();
var Tracker = require('./lib/tracker.js');

var tracker = new Tracker();
switch (options.mode) {
  case 'start':
    tracker.start(options.task);
    break;
  case 'ende':
    tracker.end();
    break;
}
```

Listing 4.57 `index.js` mit `getData` und `TimeTracker`

Anhand der Optionen, die über `getData` vom Kommandozeilenaufruf ausgelesen werden, verzweigen Sie nach der Erstellung des `TimeTracker`-Objekts entsprechend. Das Problem mit der Applikation bis jetzt ist, dass die Informationen zwar aufgenommen werden, aber nach dem Ende der Ausführung verloren gehen. Sie benötigen jetzt also eine Möglichkeit, die Daten zu persistieren. Mit dem `fs`-Modul haben

Sie bereits eine Möglichkeit kennengelernt, wie Sie sowohl lesend als auch schreibend auf das Dateisystem zugreifen können. Im nächsten Teil der Applikation widmen Sie sich also der Speicherung der Daten.

```
var fs = require('fs');
var util = require('util');

var Persister = function () {
  this.file = 'tracker.dat';
};

Persister.prototype.writeStart = function (tracker) {
  var data = tracker.date + ' ' + tracker.startTime + ' ' + tracker.task;

  fs.appendFile(this.file, data, function (err, data) {
    if (err) { throw err; }
    util.puts('Daten für ' + tracker.task + ' wurden gespeichert.');
```

```
  });
}

Persister.prototype.writeEnd = function (tracker) {
  fs.readFile(this.file, function (err, data) {
    var content = data.toString().split("\n");
    var lastLine = content.pop();
    var elements = lastLine.split(' ');

    var endLine = elements.shift() + ' ';
    endLine += elements.shift() + ' ';
    endLine += tracker.endTime + ' ';

    var task = elements.join(' ');

    endLine += task;

    var newContent = content.join("\n");
    newContent += "\n" + endLine + "\n";

    fs.writeFile(this.file, newContent, function (err, data) {
      if (err) {throw err};
      util.puts('Daten für ' + task + ' wurden gespeichert.')
```

```
    });
  });
}

module.exports = new Persister();
```

Listing 4.58 persister.js-Modul zur Speicherung von TimeTracker-Daten

Das `persister`-Modul in Listing 4.58 zeigt Ihnen eine weitere Variante von Modulen. Dieses Modul exportiert ein konkretes Objekt, mit dem Sie in Ihrer Applikation direkt arbeiten können. Dieses Objekt stellt zwei Methoden zur Persistierung zur Verfügung. Beide akzeptieren als Argument ein `TimeTracker`-Objekt, das persistiert wird.

Die erste Methode `writeStart` sorgt dafür, dass die Informationen eines neuen Datensatzes ans Ende der Protokolldatei angehängt werden. Nachdem die Daten erfolgreich geschrieben wurden, wird eine Erfolgsmeldung ausgegeben, in der die Aufgabe, die der Benutzer eingegeben hat, mit ausgegeben wird.

Die zweite Methode `writeEnd` ist etwas komplexer, da Sie zuerst auf die gespeicherten Daten zugreifen müssen, diese werden verändert und im Anschluss daran wieder gespeichert. Zu diesem Zweck wird die Datei ausgelesen, in die einzelnen Zeilen aufgeteilt und die letzte wird schließlich herausgelöst. Sie wird anhand von Leerzeichen in die einzelnen Bestandteile unterteilt und dann mit dem Endzeitpunkt wieder zusammengesetzt. Am Ende überschreiben Sie die gesamte Datei mit den bestehenden Inhalten und dem veränderten Datensatz und geben eine entsprechende Erfolgsmeldung aus.

Nachdem Sie die Daten nun in einer Datei festhalten, widmen Sie sich im nächsten Schritt der Aufgabe, die Daten dem Nutzer zur Verfügung zu stellen. Zu diesem Zweck erweitern Sie einige der bereits bestehenden Funktionalitäten. So muss beispielsweise im `Persister` dafür gesorgt werden, dass die entsprechenden Daten aus der Datei ausgelesen werden. Außerdem müssen Sie den `TimeTracker` dahingehend erweitern, dass er die Daten einer kompletten Zeile der Protokolldatei aufnehmen kann. Das `getData`-Modul ist bereits für die Statistik vorbereitet und liefert mit dem Rückgabewert den Berichtszeitraum, sodass Sie mit dem Auslesen der Daten und dem `Persister` beginnen können.

```
Persister.prototype.readFile = function (period) {
  fs.readFile(this.file, function (err, data) {
    var content = data.toString().split("\n");
    var tracker;
    for (var i = 0; i < content.length; i++) {
      if (content[i].substring(3, 10) === period) {
        tracker = new Tracker(content[i]);
        util.puts(tracker.toString());
      }
    }
  });
}
```

Listing 4.59 Erweiterung des `Persisters` um die Funktion `readFile`

Die Erweiterung der `Persister`-Klasse stellt Ihre modulare Applikation vor keine weiteren Probleme, wie Sie in Listing 4.59 sehen können. Durch die Methode `readFile` des `fs`-Moduls erhalten Sie die Inhalte der Datei und können diese, wie bereits zuvor in der `writeEnd`-Methode, in einzelne Zeilen unterteilen. Mit diesen Daten prüfen Sie, ob die Zeilen im Berichtszeitraum liegen und falls dies der Fall ist, generieren Sie daraus ein `TimeTracker`-Objekt und geben es durch eine Kombination von `util.puts` mit der `toString`-Methode des `TimeTracker`-Objekts auf der Konsole aus. Damit Ihre Applikation an dieser Stelle funktionieren kann, sind zwei weitere Anpassungen erforderlich. Die erste besteht darin, dass Sie auch in der Datei `persister.js` das `TimeTracker`-Modul einbinden müssen. Wie Sie aber bereits in den Abschnitten über den Modulloader erfahren haben, wirkt sich dieses mehrfache Laden von Modulen nicht negativ auf die Applikation aus, da das Modul aus dem Modulcache bedient wird und nicht erneut geladen werden muss. Das bedeutet, dass Sie zu Beginn der Datei `persister.js` die Zeile `var Tracker = require('./tracker.js');` einfügen müssen, damit Ihnen die Klasse zur Verfügung steht. Die zweite Änderung an der Applikation ist etwas umfangreicher. Aktuell haben Sie noch keine Möglichkeit, aus einer Zeichenkette ein neues `TimeTracker`-Objekt zu generieren. Sie müssen nun den Konstruktor dieser Klasse dahingehend erweitern, dass es möglich wird, dem Konstruktor ein Argument zu übergeben, mit dem sich ein Objekt mit Startwerten befüllen lässt. In Listing 4.60 passen Sie als nächsten Schritt den Konstruktor Ihrer `TimeTracker`-Klasse entsprechend an.

```
var TimeTracker = function (startValues) {
  if (startValues) {
    startValues = startValues.split(' ');
    this.date = startValues.shift();
    this.startTime = startValues.shift();
    this.endTime = startValues.shift();
    this.task = startValues.join(' ');
  }
}
```

Listing 4.60 Anpassung des `TimeTracker`-Konstruktors

Der Konstruktor wird lediglich dahingehend erweitert, dass aus einer möglichen Zeichenkette, die durch die Applikation übergeben wurde, nacheinander das Datum, die Uhrzeiten und die Aufgabendefinition ausgelesen und über die automatisch generierten Setter-Methoden in die Variablen gespeichert werden. Der letzte Schritt, damit Ihre Statistik funktionieren kann, besteht in einer Anpassung der `index.js`-Datei. Hier müssen Sie lediglich eine Zeile im `switch`-Statement unter dem `case: 'statistik'` einfügen, den `Persister` laden und jeweils Methodenaufrufe in die ver-

schiedenen Fälle des `switch`-Statements einsetzen. Listing 4.61 zeigt Ihnen die entsprechenden Anpassungen.

```
var options = require('./lib/getData.js').getData(),
    Tracker = require('./lib/tracker.js'),
    persister = require('./lib/persister.js');

var tracker = new Tracker();

switch (options.mode) {
  case 'start':
    tracker.start(options.task);
    persister.writeStart(tracker);
    break;
  case 'ende':
    tracker.end();
    persister.writeEnd(tracker);
    break;
  case 'statistik':
    persister.readFile(options.period);
    break;
}
```

Listing 4.61 Erweiterter Einstieg in die Applikation

Bisher haben Sie in Ihrer Applikation die Verwendung von eigenen Node.js-Modulen gesehen, die sowohl Funktionen als auch Klassen oder konkrete Objekte exportierten, dies alles jedoch in einer flachen Hierarchie, in der die Module nebeneinander existierten und maximal Funktionalität eines anderen Moduls verwendeten. Mit der `inherits`-Methode aus dem `Util`-Modul haben Sie außerdem die Möglichkeit, eine prototypenbasierte Vererbungshierarchie aufzubauen. Im Rahmen der `TimeTracker`-Applikation werden Sie Vererbung einsetzen, um die Funktionalität der `TimeTracker`-Klasse zu erweitern. Aktuell fehlt der Applikation noch die Möglichkeit, die Zeitspanne zu berechnen, die eine einzelne Aufgabe in Anspruch genommen hat. Dazu führen Sie eine neue Klasse `ExtendedTimeTracker` ein, die über eine derartige Funktionalität verfügt.

Im ersten Schritt der Erweiterung kümmern Sie sich um die Erstellung der neuen Klasse `ExtendedTimeTracker`. Diese baut auf den drei Modulen `Util`, `Persister` und `TimeTracker` auf. Dabei kommt das interne `Util`-Modul bei der Vererbung ins Spiel. Der `Persister` wird dazu verwendet, die Startzeit für die Berechnung auszulesen, und `TimeTracker` bildet schließlich die Basis für Ihre Vererbungshierarchie.


```

var util = require('util'),
    persister = require('./persister.js'),
    TimeTracker = require('./tracker.js');

var secondCalc = function (time) {
    var time = time.split(':');
    return parseInt(time[0]) * 60 + parseInt(time[1]);
}

var ExtendedTimeTracker = function(startValues) {
    if (startValues) {
        startValues = startValues.split(' ');
        this.date = startValues.shift();
        this.startTime = startValues.shift();
        this.endTime = startValues.shift();
        this.diff = startValues.shift();
        this.task = startValues.join(' ');
    }
}

util.inherits(ExtendedTimeTracker, TimeTracker);

Object.defineProperty(ExtendedTimeTracker.prototype, "diff", {
    get: function() {return this._diff;},
    set: function(diff) {this._diff = diff; return this;}
});

ExtendedTimeTracker.prototype.end = function () {
    TimeTracker.prototype.end.call(this);

    var startTime = persister.getLastLineData().split(' ')[1];
    var etSec = secondCalc(this.endTime);
    var stSec = secondCalc(startTime);
    var diff = etSec - stSec;
    var hours = diff === 0 ? 0 : Math.floor(diff/60);
    var minutes = diff - hours * 60;
    this.diff = hours + ':' + minutes;
}

TimeTracker.prototype.toString = function () {
    var result = this.date + ' ';

```

```

    result += this.startTime + ' ';
    result += this.endTime + ' ';
    result += this.diff + ' ';
    result += this.task;
    return result;
}

```

```
module.exports = ExtendedTimeTracker;
```

Listing 4.62 Die Klasse `ExtendedTimeTracker`

In Listing 4.62 sehen Sie den Quellcode für Ihre neue Klasse. Der Vorteil bei dieser Vorgehensweise ist, dass Sie sich jetzt lediglich um die Unterschiede zur Basisklasse kümmern müssen und die übrige Funktionalität wiederverwenden können. Die Unterschiede bestehen hauptsächlich in der Behandlung der zusätzlichen Eigenschaft, der Zeitdifferenz zwischen Start- und Endzeitpunkt. Zu diesem Zweck erhält die Klasse eine Eigenschaft `diff`, mit der Sie auf die gleiche Art und Weise verfahren wie mit den übrigen Eigenschaften auch. Das bedeutet, dass Sie über `Object.defineProperty` Getter- und Setter-Methoden dafür definieren und den Wert in der Hilfsvariablen `_diff` speichern. Damit Ihre Klasse mit dem neuen Wert korrekt umgehen kann, müssen Sie auch den Konstruktor entsprechend erweitern, da dieser mit einer veränderten Zeichenkette als Eingabe umgehen können muss. Nachdem Sie den Konstruktor definiert haben, kommt das `Util`-Modul mit seiner `inherits`-Methode zum Einsatz. Sie sorgt dafür, dass die Eigenschaften und Methoden der Klasse `TimeTracker` in Ihre neu erstellte Klasse übernommen werden. Schließlich müssen Sie noch die beiden Methoden `end` und `toString` anpassen. `toString` muss ähnlich wie schon der Konstruktor um die neue Eigenschaft erweitert werden. Bei der `end`-Methode ist der Fall nicht ganz so einfach. Hier müssen Sie zuerst darauf achten, dass die bestehende Funktionalität erhalten bleibt. Dies stellen Sie mit der Zeile `TimeTracker.prototype.end.call(this)` sicher, die dafür sorgt, dass die `end`-Methode der Elternklasse aufgerufen wird. Danach nutzen Sie das `persist`-Modul, um die letzte Zeile der Protokolldatei auszulesen und so den Startzeitpunkt des letzten Eintrags auszulesen. Dieser wird benötigt, um die Differenz zu berechnen. Im Zuge der `end`-Funktion wird der berechnete Wert über den zuvor generierten Setter zugewiesen. Mit dem letzten Schritt, dem Export der Klasse, damit diese als Modul eingebunden werden kann, ist die Klasse `ExtendedTimeTracker` fertiggestellt und kann in Ihre Applikation eingebunden werden.

Der erste Schritt der Einbindung besteht darin, dass Sie die `index.js`-Datei dahingehend anpassen, dass sie die Klasse `ExtendedTimeTracker` statt `TimeTracker` verwendet. Da beide Klassen kompatible Schnittstellen aufweisen, reicht es aus, wie Sie in Listing 4.63 sehen können, die Zeile anzupassen, über die das Modul eingebunden wird.

```

var options = require('./lib/getData.js').getData(),
    Tracker = require('./lib/etracker.js'),
    persister = require('./lib/persister.js').persister;

var tracker = new Tracker();

switch (options.mode) {
  case 'start':
    tracker.start(options.task);
    persister.writeStart(tracker);
    break;
  case 'ende':
    tracker.end();
    break;
  case 'statistik':
    persister.readFile(options.period, Tracker);
    break;
}

```

Listing 4.63 Einbindung des ExtendedTimeTracker-Moduls

Da sich außer dem Dateinamen und der verwendeten Klasse nichts weiter ändert, also die Variable, in der die Klasse zwischengespeichert wird, unverändert bleibt, war das an dieser Stelle auch die einzige Anpassung. Damit allerdings das `persister`-Modul weiterhin funktionieren kann, müssen Sie noch einige weitere Änderungen vornehmen. Die erste Schwierigkeit ergibt sich aus der Tatsache, dass der `persister` das `ExtendedTimeTracker`-Modul einbinden muss und das `ExtendedTimeTracker`-Modul wiederum den `persister` einbindet. Das führt zu einer zirkulären Abhängigkeit, die zu einer Endlosschleife führen würde, hätte nicht der Modulloader von Node.js einen Mechanismus, mit dem er in die Lage versetzt wird, mit solchen Fällen entsprechend umzugehen. Er lädt die Module und lässt sie in einem unfertigen Zustand. Danach werden die übrigen Komponenten eingebunden. Dieser Mechanismus ist in diesem Fall wenig hilfreich. Der `persister` benötigt die Klasse lediglich an einer Stelle, und zwar in der `readFile`-Methode. Deshalb bauen Sie am besten den `persister` dahingehend um, dass er die Klasse als Eingabe für die Methode akzeptiert und das Modul aus diesem Grund nicht mehr laden muss. Das heißt für Sie jetzt allerdings, dass Sie den `persister` an einigen Stellen modifizieren müssen. Welche das genau sind, können Sie im nachfolgenden Listing 4.64 sehen.

```

var fs = require('fs'),
    util = require('util');
...
Persister.prototype.readFile = function (period, Tracker) {

```

```

...
Persister.prototype.getLastLineData = function() {
    var content = fs.readFileSync(this.file).toString().split("\n");
    return lastLine = content.pop();
}
...

```

Listing 4.64 Anpassung des persister-Moduls

Wie bereits erwähnt, benötigen Sie hier die Einbindung des `TimeTracker`-Moduls nicht mehr. Sie können diese Zeile also getrost entfernen. Außerdem übergeben Sie mittlerweile der Methode `readFile` die Klasse `ExtendedTimeTracker`, also müssen Sie die Signatur dieser Methode um einen zweiten Parameter erweitern. Benennen Sie diesen Parameter wie die Variable, in die zuvor das `TimeTracker`-Modul geladen wurde, müssen Sie an der `readFile`-Methode nichts mehr anpassen. Die erweiterte `end`-Methode, wie Sie im `ExtendedTimeTracker` zum Einsatz kommt, basiert auf der neuen `getLastLineData`-Methode des `persister`-Moduls. Diese Methode gibt die letzte Zeile der Protokolldatei als Zeichenkette an den Aufrufenden zurück. Mit diesen Änderungen können Sie nun die Applikation verwenden. Über die Kommandozeile können Sie einen Arbeitszeitraum beginnen und ihn auch wieder beenden. Die Informationen werden in einer Datei festgehalten, die dann als Basis für Monatsstatistiken dient, die Sie sich ausgeben lassen können. Im Zuge der Applikation haben Sie einige der zuerst in der Theorie vorgestellten Themen auch in der praktischen Anwendung kennengelernt. Unter anderem haben Sie die Erstellung eigener Module behandelt und konnten sehen, dass ein Modul sowohl Klassen als auch Funktionen oder Objekte zur Verfügung stellen kann, mit denen Sie in Ihrer Applikation arbeiten können. Die Einsatzgebiete dieser drei Varianten unterscheiden sich erheblich voneinander.

In Listing 4.65 sehen Sie, wie Sie die `Time-Tracker`-Applikation verwenden können.

```

$ node index.js start TimeTracker
Daten für TimeTracker wurden gespeichert.
$ node index.js ende
Daten für TimeTracker wurden gespeichert.
$ node index.js statistik 02.2013
05.02.2013 12:25:00 14:38:00 02:13:00 TimeTracker

```

Listing 4.65 Verwendung der Time-Tracker-Applikation

Module, die einzelne Funktionen exportieren, verwenden Sie vor allem im Bereich von Werkzeugsammlungen. Ein gutes Beispiel für diese Art von Modul ist das `Utility`-Modul von `Node.js`, das eine Reihe recht lose zusammenhängender Funktionen zur Verfügung stellt.

Den Export von Klassen verwenden Sie, wenn Sie eine Möglichkeit benötigen, mehrere Objekte dieses Typs in Ihrer Applikation erzeugen zu können. Diese Art der Module entspricht am ehesten dem System, wie Sie es wahrscheinlich aus anderen Programmiersprachen kennen, wenn es um objektorientierte Programmierung geht. Hier gilt meist der Grundsatz: pro Klasse eine Datei.

Der Export von Objekten entspricht am ehesten dem Design Pattern des Singletons. Es sollte von dieser Art nur eine Instanz in Ihrer Applikation geben. Dies kann entweder dadurch verursacht sein, dass es keine zwei Instanzen innerhalb einer Applikation geben darf oder aber, weil es schlicht und ergreifend nicht erforderlich ist, mehrere Instanzen dieses Typs zu verwalten.

Sie haben in diesem Abschnitt erfahren, dass Ihnen die Node.js-Plattform im Zusammenspiel mit dem ECMAScript-Standard in der Version 5 eine Reihe mächtiger Funktionalitäten zur Verfügung stellt, mit denen Sie Objekte sowie deren Eigenschaften und Methoden beeinflussen können. So können Sie mithilfe von `Object.create` neue Objekte erstellen und ihnen einen Prototypen zuweisen. `Object.defineProperty` bietet Ihnen die Möglichkeit, gezielt Eigenschaften zu manipulieren, damit werden Sie in die Lage versetzt, echte Konstanten in JavaScript zu definieren oder, wie Sie im Beispiel in Listing 4.55 sehen konnten, Getter- und Setter-Methoden zu definieren. Auch die prototypenbasierte Vererbung ist problemlos zu implementieren, wie Sie in der Time-Tracker-Applikation erfahren konnten. Mithilfe der `inherits`-Methode des `Utility`-Moduls können Sie auf einfache Art eine Vererbungshierarchie in Ihrer Applikation aufbauen und dabei die Funktionalität Ihrer Klassen schrittweise, je nach Bedarf, ausbauen. Doch damit enden die Möglichkeiten von Node.js noch lange nicht. In den nächsten Abschnitten lernen Sie ein weiteres sehr wichtiges Feature der Plattform kennen: den Node Package Manager.

4.4 NPM

Der Node Package Manager, kurz NPM, bietet Ihnen als Entwickler von serverseitigen JavaScript-Applikationen auf Basis der Node.js-Plattform eine Möglichkeit, mit der Sie Module erstellen und diese anderen Entwicklern zur Verfügung stellen können. Genauso können Sie die Module, die andere Programmierer entwickelt haben, in Ihrer Applikation nutzen. Die Webseite des NPM lautet <https://npmjs.org/>. Hier finden Sie neben verschiedenen Ressourcen wie Modullisten auch eine ausführliche Dokumentation des NPM und aller verfügbaren Optionen. Der NPM wurde ursprünglich unabhängig von der Node.js-Plattform von Isaac Schlueter entwickelt. Seit der Version 0.6.3 von Node.js ist der NPM in die Plattform direkt integriert und muss nicht mehr zusätzlich installiert werden. Das bedeutet, dass Sie entweder den .msi-Installer unter Windows, das .pkg-Paket unter Mac OS X oder das Binärpaket unter Linux installieren und danach sofort den NPM nutzen können. Der NPM wird auf der Kommandozeile verwendet und über verschiedene Optionen gesteuert.

Der Node Package Manager ist ein vollwertiger Paketverwalter für die Node.js-Plattform. Mit ihm können Sie Pakete suchen, installieren, aktualisieren und auch wieder löschen. Die Node Packaged Modules dienen dazu, die Funktionalität der Node.js-Plattform zu erweitern. Das bedeutet, dass sie den Kern der Node.js-Plattform um die V8-Engine und die internen Module von Node.js ergänzen. Die Pakete des NPM werden zentral auf dem Server unter <http://registry.npmjs.org/> verwaltet. Greifen Sie von Ihrem Browser auf diese Adresse zu, erhalten Sie lediglich Datenstrukturen im JSON-Format. Von der Wurzel des Servers erhalten Sie allgemeine Informationen über das Repository wie beispielsweise die Anzahl der Module. Eine detaillierte Liste der Module finden Sie unter <http://registry.npmjs.org/-/all>. Auch diese Seite enthält eine JSON-Datenstruktur.

Joyent, das Unternehmen, das Node.js unterstützt, pflegt auf dem github-Account des Unternehmens eine Liste der wichtigsten Module. Diese Liste finden Sie unter <https://github.com/joyent/node/wiki/Modules>. Benötigen Sie beispielsweise einen XML-Parser für Ihre Applikation, werden Sie unter dem Punkt Parsers und XML fündig und erhalten dort eine Liste von Parseern, die Sie über den NPM installieren können. Um die Entscheidung besser treffen zu können, erhalten Sie in der Liste bereits eine kurze Beschreibung und einen Link auf die Projektseite des jeweiligen Moduls.

4.4.1 Pakete suchen

Im April 2013 befanden sich bereits über 28.000 Pakete im Repository. Aufgrund dieser großen Anzahl fällt es schwer, den Überblick zu behalten. Wenn Sie wissen, welche Funktionalität Sie in Ihrer Applikation benötigen, können Sie gezielt nach entsprechenden Paketen suchen. Eine Anlaufstelle ist die bereits erwähnte Liste der Module im github-Wiki von Joyent. Eine weitere Möglichkeit ist die Suche über das Kommandozeilenwerkzeug `npm` selbst. Über die Option `search` können Sie das Repository durchsuchen. Dabei stehen Ihnen die Felder `Name`, `Description`, `Author` und `Keywords` zur Suche zur Verfügung.

Angenommen, Sie möchten einen XML-Parser in Ihre Applikation einbinden und wissen nicht genau, wie das zu installierende Modul heißt, können Sie das Repository danach durchsuchen. In Listing 4.66 sehen Sie das Kommando und seine Ausgabe.

```
$ npm search xml parser
npm http GET https://registry.npmjs.org/-/all/since?sta...
npm http 200 https://registry.npmjs.org/-/all/since?sta...
NAME      DESCRIPTION      AUTHOR    DATE      KEYWORDS
big-xml   Lightweight XML  parser...  =jahewson 2012-04-03 22:23
```

Listing 4.66 Das Kommando »`npm search`«

Das Kommando `npm search` greift direkt auf die Informationen im Repository und auf die JSON-Datenstrukturen zu, um die gesuchten Informationen darzustellen. Die Ausgabe des Kommandos besteht aus dem Namen des Moduls, der kurzen Beschreibung, dem Autor des Moduls, dem Datum der letzten Aktualisierung und eventuellen, vom Ersteller des Moduls definierten Schlüsselworten.

4.4.2 Pakete installieren

Nachdem Sie das Paket gefunden haben und seinen Namen wissen, können Sie es nun im nächsten Schritt auf Ihrem System installieren. Dazu verwenden Sie den Befehl `npm install` und fügen den Paketnamen an. Angenommen, Sie möchten in Ihrer Applikation die Bibliothek `Underscore.js` installieren, können Sie das mit dem Befehl `npm install underscore` erreichen. Die entsprechende Ausgabe sehen Sie in Listing 4.67.

```
$ npm install underscore
npm http GET https://registry.npmjs.org/underscore
npm http 200 https://registry.npmjs.org/underscore
npm http GET https://registry.npmjs.org/underscore/-/underscore-1.4.2.tgz
npm http 200 https://registry.npmjs.org/underscore/-/underscore-1.4.2.tgz
underscore@1.4.2 node_modules/underscore
```

Listing 4.67 Installation von Underscore.js über NPM

Der Befehl `npm install` fragt im ersten Schritt die Registry nach Informationen über das gewünschte Paket ab. Wie Sie in Listing 4.67 sehen können, besteht die Abfrage aus einem ganz normalen GET-Request, wie Sie ihn auch von Ihrem Browser aus absetzen können. Öffnen Sie die angegebene URL in Ihrem Browser, sehen Sie, dass es sich bei dem Dokument um eine JSON-Datenstruktur handelt, die eine Reihe von Informationen über das Paket enthält. Unter anderem finden Sie im letzten Abschnitt die Informationen über die aktuelle Version des Moduls, inklusive der URL zum Paket selbst. Das Paket liegt auf dem Server als gezipptes Tar-Archiv vor. Mit einem weiteren GET-Request lädt der NPM dieses Archiv herunter und installiert es auf Ihrem Rechner. Die Installation von Modulen über den NPM erfolgt ohne weitere Kommandozeileingaben lokal in dem Verzeichnis, in dem Sie sich gerade in Ihrer Shell befinden. Das bedeutet für Sie, dass Sie keine Administratorberechtigungen benötigen, um Module lokal zu installieren. Das heruntergeladene Tar-Archiv wird in das Verzeichnis `node_modules` entpackt. Hier finden Sie pro Paket ein Unterverzeichnis mit dem Namen des jeweiligen Pakets. Im Falle des Beispiels wurde also das Verzeichnis `node_modules` angelegt, da es bislang noch nicht existierte. Darin wurde ein Unterverzeichnis `underscore` angelegt, in dem sich wiederum die Dateien des Moduls befinden. Der Node Package Manager kümmert sich jedoch nicht nur um das Herunterladen von Modulen und ihre Installation, sondern ist auch in der Lage, Abhängig-

keiten von Modulen aufzulösen. Abhängigkeiten sind in diesem Fall Module, die ein Paket benötigt, um funktionieren zu können. Ein Beispiel hierfür sehen Sie mit dem `mysql`-Modul in Listing 4.68. Das `mysql`-Modul stellt Ihnen einen Client für die MySQL-Datenbank zur Verfügung, mit dessen Hilfe Sie Abfragen an eine Datenbank absetzen und mit den Resultaten innerhalb Ihrer Applikation umgehen können.

```
$ npm install mysql
npm http GET https://registry.npmjs.org/mysql
npm http 200 https://registry.npmjs.org/mysql
npm http GET https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http 200 https://registry.npmjs.org/mysql/-/mysql-2.0.0-alpha4.tgz
npm http GET https://registry.npmjs.org/require-all/0.0.3
npm http 200 https://registry.npmjs.org/require-all/0.0.3
npm http GET https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
npm http 200 https://registry.npmjs.org/require-all/-/require-all-0.0.3.tgz
mysql@2.0.0-alpha4 node_modules/mysql
└─ require-all@0.0.3
```

Listing 4.68 Installation des `mysql`-Moduls

Während der Installation sehen Sie, dass der NPM nicht nur das `mysql`-Modul herunterlädt, sondern noch ein weiteres Modul mit dem Namen `require-all`. Dieses stellt dem `mysql`-Modul eine Möglichkeit zur Verfügung, alle Dateien aus einem Verzeichnis zu laden. Installieren Sie verschiedene Module auf Ihrem System, kann es vorkommen, dass Versionskonflikte zwischen den Abhängigkeiten der verschiedenen Module entstehen. Aus diesem Grund werden die Abhängigkeiten eines Moduls innerhalb des Moduls aufgelöst. Das bedeutet in diesem Fall für das `mysql`-Modul, dass innerhalb des `mysql`-Verzeichnisses, das den Quellcode für das Modul enthält, wiederum ein `node_modules`-Verzeichnis erstellt wird, in dem sich dann die Module befinden, auf denen das `mysql`-Modul aufbaut. So werden Versionskonflikte grundsätzlich vermieden. Das hat allerdings den Preis, dass sich auf Ihrem Dateisystem unter Umständen mehrfach die gleichen Module befinden. Zusammengefasst bedeutet das für Sie, dass Sie sich bei der Verwendung des NPM nicht um die Auflösung von Abhängigkeiten, den manuellen Download von Paketen oder das Entpacken und Installieren von Modulen kümmern müssen. All diese Aufgaben nimmt Ihnen der Paketmanager ab.

Neben der lokalen Installation im aktuellen Verzeichnis gibt es noch eine weitere Variante der Installation von Modulen. Sie können ein Node.js-Modul auch systemweit installieren, damit es nicht nur für Sie in Ihrem aktuellen Projekt, sondern für alle Nutzer und Projekte auf dem Rechner verfügbar ist. Sie können ein Modul global installieren, indem Sie die Kommandozeilenoption `-g`, für global, im Befehl zur Installation angeben. In Listing 4.69 sehen Sie ein Beispiel für die globale Installation.


```

$ sudo npm install -g grunt-cli
Password:
npm http GET https://registry.npmjs.org/grunt-cli
npm http 200 https://registry.npmjs.org/grunt-cli
npm http GET https://registry.npmjs.org/resolve
npm http GET https://registry.npmjs.org/nopt
npm http GET https://registry.npmjs.org/findup-sync
npm http 200 https://registry.npmjs.org/findup-sync
npm http 200 https://registry.npmjs.org/resolve
npm http GET https://registry.npmjs.org/resolve/-/resolve-0.3.1.tgz
npm http 200 https://registry.npmjs.org/nopt
npm http 200 https://registry.npmjs.org/resolve/-/resolve-0.3.1.tgz
npm http GET https://registry.npmjs.org/abbrev
npm http GET https://registry.npmjs.org/glob
npm http GET https://registry.npmjs.org/lodash
npm http 200 https://registry.npmjs.org/abbrev
npm http 200 https://registry.npmjs.org/glob
npm http 200 https://registry.npmjs.org/lodash
npm http GET https://registry.npmjs.org/minimatch
npm http GET https://registry.npmjs.org/graceful-fs
npm http GET https://registry.npmjs.org/inherits
npm http 200 https://registry.npmjs.org/inherits
npm http 200 https://registry.npmjs.org/graceful-fs
npm http 200 https://registry.npmjs.org/minimatch
npm http GET https://registry.npmjs.org/minimatch/-/minimatch-0.2.12.tgz
npm http 200 https://registry.npmjs.org/minimatch/-/minimatch-0.2.12.tgz
npm http GET https://registry.npmjs.org/sigmund
npm http GET https://registry.npmjs.org/lru-cache
npm http 200 https://registry.npmjs.org/sigmund
npm http 200 https://registry.npmjs.org/lru-cache
npm http GET https://registry.npmjs.org/lru-cache/-/lru-cache-2.3.0.tgz
npm http 200 https://registry.npmjs.org/lru-cache/-/lru-cache-2.3.0.tgz
/usr/local/bin/grunt -> /usr/local/lib/node_modules/grunt-cli/bin/grunt
grunt-cli@0.1.7 /usr/local/lib/node_modules/grunt-cli
├─ resolve@0.3.1
├─ nopt@1.0.10 (abbrev@1.0.4)
└─ findup-sync@0.1.2 (lodash@1.0.1, glob@3.1.21)

```

Listing 4.69 Globale Installation von grunt

Bei der Installation von globalen Modulen müssen Sie im Gegensatz zur lokalen Installation einige Dinge beachten. Als normaler Nutzer dürfen Sie Module nicht global installieren, da diese standardmäßig unter `/usr/local/lib/node_modules` gespei-

chert werden. Daher müssen Sie die Installation entweder direkt als Administrator durchführen oder auf UNIX-Systemen mit dem Kommando `sudo`. Die globale Installation von Modulen ist allerdings hauptsächlich für Kommandozeilenwerkzeuge gedacht, die mit Node.js umgesetzt wurden. Beispiele hierfür sind `grunt`, ein Build-tool für JavaScript-Applikationen, oder `lessc`, ein Werkzeug zum Übersetzen von Stylesheets im LESS-Format. Für normale Module wie beispielsweise `underscore` oder `mysql` empfiehlt es sich, diese Module lokal zu installieren. Der erste Grund dafür ist, dass lokale Module schneller geladen werden, da der Suchvorgang nach dem Modul erheblich verkürzt wird. Der zweite, auch nicht unwesentliche Grund ist, dass Ihre Applikation mit global installierten Modulen Abhängigkeiten nach außen hat. Das bedeutet, dass auf jedem System, auf dem die Applikation ausgeführt werden soll, diese globalen Module installiert sein müssen, da die Applikation ansonsten nicht lauffähig ist. Aus diesem Grund ist es empfehlenswert, alle Module, auf denen Ihre Applikation aufbaut auch lokal zu installieren, sodass sie möglichst unabhängig lauffähig ist.

Egal, ob Sie ein Modul lokal oder global installieren, erhalten Sie als Ausgabe des `node install`-Kommandos stets die Anfragen an den `npmjs.org`-Server, die jeweiligen Antworten vom Server und zum Abschluss eine Auflistung der installierten Pakete und deren Abhängigkeiten untereinander. Sie sind bei der Installation von Modulen über den NPM jedoch nicht abhängig von einer ständigen Verbindung zum Server. Sie haben neben der Installation der Module über eine bestehende Internetverbindung außerdem die Möglichkeit, Module direkt von Ihrem Rechner zu installieren. Die einzige Voraussetzung, die Sie hierfür erfüllen müssen, ist, dass Sie entweder das Tar-Archiv, das das Modul enthält, heruntergeladen haben oder sich ein Verzeichnis auf Ihrem Rechner befindet, das den Quellcode des Moduls und eine entsprechende `package.json`-Datei enthält. Um die lokale Installation zu testen, können Sie einfach das `underscore`-Modul herunterladen und es installieren. Listing 4.70 demonstriert, wie Sie hierbei vorgehen müssen.

```
$ wget http://registry.npmjs.org/underscore/-/underscore-1.4.2.tgz
...
2012-11-30 11:27:35 (114 KB/s) - 'underscore-1.4.2.tgz' saved [61836/61836]
$ npm install underscore-1.4.2.tgz
underscore@1.4.2 node_modules/underscore
```

Listing 4.70 Lokale Installation eines Moduls über NPM

Durch die Installation wird wie auch bei der Verwendung des Repositorys unter `npmjs.org`, ein `node_modules`-Verzeichnis erstellt, in das der Quellcode des Moduls kopiert wird. Nach erfolgreicher Installation können Sie das Modul in Ihrer Applikation verwenden.

4.4.3 Installierte Pakete anzeigen

Sämtliche Pakete, die Sie installieren, finden Sie im `node_modules`-Verzeichnis Ihrer Applikation wieder. Da die Abhängigkeiten allerdings pro Modul aufgelöst werden, wird es sehr schnell unübersichtlich, wenn Sie herausfinden möchten, welche Pakete innerhalb Ihrer Applikation installiert sind. Einen schnellen Überblick über die installierten Module und deren Abhängigkeiten bietet Ihnen das Kommando `npm list`.

```
$ npm list
/tmp/node
├─ mysql@2.0.0-alpha4
| └─ require-all@0.0.3
└─ underscore@1.4.2
```

Listing 4.71 Auflistung der installierten Module

Wie Sie in Listing 4.71 sehen können, handelt es sich bei der Auflistung der Module um eine Baumstruktur, aus der Sie direkt ablesen können, welche Module installiert sind und wie sich die Abhängigkeiten zwischen den Modulen definieren. Eine ähnliche Ansicht erhalten Sie auch nach der Installation eines Moduls. Diese enthält allerdings nur das Modul selbst und seine Abhängigkeiten, wohingegen `npm list` sämtliche Module in Ihrem lokalen `node_modules`-Verzeichnis auflistet. Mit der Option `-g` erhalten Sie vom NPM außerdem eine Liste aller global installierten Module Ihres Systems. Hier werden ebenfalls sämtliche Abhängigkeiten in einer Baumstruktur angezeigt.

4.4.4 Pakete verwenden

Wie Sie bereits sehen konnten, gibt es zwei verschiedene Arten von Modulen, die über den NPM installiert werden können. Zum einen sind dies Sammlungen von Funktionalitäten, die Sie direkt in Ihre Applikation einbinden können, und zum anderen aber auch Werkzeuge, die Sie direkt von der Kommandozeile aus verwenden können. Im Folgenden lernen Sie Vertreter beider Kategorien kennen. Dabei sehen Sie, wie Sie die Module entsprechend in der Praxis einsetzen können.

In Listing 4.69 haben Sie bereits das JavaScript-Buildtool `grunt-cli` global auf Ihrem System installiert. Zusätzlich dazu müssen Sie `grunt` als lokales Modul installieren. Danach können Sie es verwenden, um Ihre Applikation zu bauen oder verschiedene Aufgaben der Qualitätssicherung zu erledigen. Um `grunt` auszuführen, müssen Sie lediglich den Kommandozeilenbefehl `grunt` in einem Verzeichnis ausführen, in dem eine Konfigurationsdatei, das sogenannte *Gruntfile*, liegt. Die Konfigurationsdatei enthält die Informationen über die verschiedenen Aufgaben, die `grunt` für die Applikation erledigen soll. Der Name dieser Datei lautet im Normalfall *Gruntfile* und liegt

im JSON-Format im Wurzelverzeichnis Ihrer Applikation. Dabei ist das Kommando `grunt` in diesem Fall eine normale JavaScript-Applikation, bei der im Dateikopf ein Hinweis für das Betriebssystem, der sogenannte Shebang, steht, der der Shell anzeigt, mit welcher Anwendung das Script zu interpretieren ist. Im Falle von `grunt` lautet diese Zeile `#!/usr/bin/env node` und bedeutet, dass das Script mit Node.js ausgeführt werden soll. Mit diesen Anpassungen kann eine JavaScript-Applikation wie `grunt` auf der Kommandozeile als gewöhnliches Kommando verwendet werden und muss nicht als Parameter an Node.js übergeben werden.

Die zweite Möglichkeit, wie Sie Module verwenden können, die Sie über den NPM installieren, ist die direkte Einbindung der jeweiligen Funktionalität in Ihre Applikation. Im Verlauf dieses Abschnitts haben Sie bereits gesehen, wie Sie die Bibliothek `Underscore.js` lokal in Ihrer Applikation installieren können. Dieses Modul werden wir nun verwenden, um ein kleines Beispiel für den Einsatz von Modulen zu erstellen.

```
var _ = require('underscore'),
    arr1 = ['a', 'b', 'c'],
    arr2 = ['b', 'c', 'd'];
console.log(_.intersection(arr1, arr2));
```

Listing 4.72 Einfaches Beispiel für die Verwendung eines Moduls

In Listing 4.72 laden Sie über `require` das `underscore`-Modul, das Sie zuvor mit dem NPM lokal installiert haben. Danach definieren Sie zwei Arrays und lassen sich deren Schnittmenge ausgeben. Hierfür stellt Ihnen `Underscore.js` die Funktion `intersection` zur Verfügung. Die Rückgabe ist in diesem Fall ein Array mit den Elementen `b` und `c`.

Nach diesem einführenden Beispiel in die Verwendung von Modulen, die Sie mit dem NPM installieren können, steigen Sie gleich etwas praktischer ein und erweitern Ihre `Time-Tracker`-Applikation. Eines der größten Probleme der Applikation besteht im Schreiben und Lesen der Daten. Aktuell verwenden Sie für diesen Zweck eine einfache Protokolldatei. Insgesamt haben Sie an drei Stellen die gesamte Protokolldatei eingelesen und den Inhalt verarbeitet. Diese Art des Zugriffs eignet sich allerdings lediglich für kleine Dateien, sobald Sie es mit mehr als ein paar Zeilen in der Protokolldatei zu tun haben, werden die Zugriffe langsamer. Die einfachste Lösung für diese Problemstellung besteht im Einsatz einer Datenbank. Aus einer solchen können selektiv Datensätze gelesen werden, und es ist nicht nötig, den gesamten Datenbestand in ein Objekt im Arbeitsspeicher zu laden.

Mit dem modularen Ansatz, den Sie bei der Erstellung dieser Applikation verfolgten, ist es außerdem möglich, eine Datenbank statt der Protokolldatei zu verwenden, ohne direkt die komplette Applikation anpassen zu müssen. Als Datenbank verwenden Sie in diesem Fall die Open-Source-SQL-Datenbank `MySQL`. Diese Datenbank ist auf allen

Systemen, auf denen Node.js funktioniert, ebenfalls verfügbar. In Listing 4.73 sehen Sie die ersten Schritte, die in der Erstellung einer entsprechenden Datenbank und der zugehörigen Tabelle bestehen, die für den geplanten Umbau erforderlich sind.

```
$ mysql -u root -p
mysql> CREATE DATABASE `TimeTracker`;
Query OK, 1 row affected (0.00 sec)
mysql> use `TimeTracker`;
Database changed
mysql> CREATE TABLE `times` (
    `date` DATE,
    `startTime` TIME,
    `endTime` TIME,
    `diff` TIME,
    `task` VARCHAR(255)
);
Query OK, 0 rows affected (0.01 sec)
```

Listing 4.73 Erstellung der Datenbankstruktur

Nachdem die Datenbank angelegt ist und darin eine Tabelle erstellt ist, die in ihrer Struktur der Protokolldatei gleicht, also Felder für das Datum, den Start- und Endzeitpunkt, die Differenz zwischen beiden und ein Freitext-Feld für die Aufgabe aufweist, können Sie sich der Applikation widmen.

Der nächste Schritt besteht darin, dass Sie das `mysql`-Modul mithilfe des NPM installieren und so eine Möglichkeit erhalten, von Ihrer Node.js-Applikation auf die Datenbank zuzugreifen. Um das Modul zu installieren, müssen Sie lediglich im Wurzelverzeichnis der Time-Tracker-Applikation das Kommando `npm install mysql` absetzen. Die einzelnen Schritte der Installation haben Sie bereits in Listing 4.68 gesehen. Die umfangreichsten Änderungen müssen Sie, wie zu erwarten war, an der Stelle durchführen, die mit dem Speichern und Auslesen der Daten zu tun hat, am `persist`-Modul. Im Anschluss bauen Sie dieses Modul schrittweise um. In Listing 4.74 sehen Sie, wie Sie die benötigten Module einbinden und wie Sie die Konstruktor-Funktion entsprechend anpassen müssen.

```
var fs = require('fs'),
    util = require('util')
    mysql = require('mysql');

var Persister = function () {
    this.connection = mysql.createConnection({
        host      : 'localhost',
        user      : 'root',
```

```

        database : 'TimeTracker',
        password : ''
    });
    this.connection.connect();
};

```

Listing 4.74 Einbindung der Module und Anpassen des Konstruktors

Wie bereits angekündigt, binden Sie nun im `persister`-Modul das `mysql`-Modul ein, das Ihnen einen Client für die benötigte Datenbankverbindung bietet. Mit dem zurückgegebenen `mysql`-Objekt, dem Client, können Sie dann im Konstruktor die Verbindung aufbauen. Dazu müssen Sie mit der `createConnection`-Methode ein `connection`-Objekt erstellen und dazu ein Konfigurationsobjekt übergeben, das Server, Benutzer, Datenbank und Passwort enthält. Mit dem Aufruf der `connect`-Methode stellen Sie eine Verbindung zur Datenbank her. Über das `connection`-Objekt können Sie nun Abfragen an den Datenbankserver absetzen.

```

Persister.prototype.writeStart = function (tracker) {
    var data = {
        "date": tracker.date,
        "startTime": tracker.startTime,
        "task": tracker.task
    },
    query = 'INSERT INTO `times` SET ?';
    this.connection.query(query, data, function(err, result) {
        if (err) { throw err; }
        util.puts('Daten für ' + tracker.task + ' wurden gespeichert.');
```

```

        this.connection.end();
    }).bind(this));
}

```

Listing 4.75 Die writeStart-Methode

Die `writeStart`-Methode benutzen Sie im `TimeTracker` bisher dazu, eine neue Zeile in der Protokolldatei und damit einen neuen Eintrag zu erzeugen. Für den Umbau der Applikation müssen Sie diese Methode, wie Sie Listing 4.75 entnehmen können, umbauen. Das Äquivalent zu einer Zeile in der Protokolldatei ist in der Datenbank ein neuer Datensatz in der Tabelle. Dazu formulieren Sie eine Abfrage, die dafür sorgt, dass das Datum, der Startzeitpunkt und der Task in die Tabelle geschrieben werden. Sämtliche Datenbankoperationen werden asynchron durchgeführt. Aus diesem Grund müssen Sie der `query`-Methode neben der Zeichenkette, die die Abfrage enthält, und einem Objekt, das die Werte für die Abfrage enthält, eine Callback-Funktion übergeben. Dieser Callback wird ausgeführt, sobald die Abfrage durchgeführt

wurde und eine Rückmeldung vorliegt. Diese Rückmeldung kann entweder positiv sein, in diesem Fall enthält das zweite Argument, `result`, einen Wert, andernfalls liegt ein Fehler-Objekt als erstes Argument vor. Die Ausgabe an den Benutzer ist die gleiche wie auch schon bei der Verwendung der Protokolldatei. Ein wichtiger Punkt, den Sie bei der Verwendung von Datenbankverbindungen beachten müssen, ist, dass Sie eine bestehende Verbindung stets trennen müssen. Die Verbindung wird durch einen Aufruf der `end`-Methode auf das `connection`-Objekt beendet. Führen Sie diese Operation nicht durch, wird die Verbindung offen gehalten und die Applikation nach dem Schreiben des Datensatzes nicht beendet.

Das Gegenstück zur `writeStart`-Methode besteht aus der `writeEnd`-Methode, die dafür sorgt, dass der Endzeitpunkt und die Differenz zwischen Start und Ende in den Datensatz geschrieben werden. Der Datensatz wird durch einen leeren Wert im `endTime`-Feld identifiziert. In Listing 4.76 sehen Sie die vollständige Implementierung der `writeEnd`-Methode.

```
Persister.prototype.writeEnd = function (tracker) {
  var data = {
    "endTime": tracker.endTime,
    "diff": tracker.diff
  },
  query = 'UPDATE `times` SET ? WHERE `endTime` IS NULL';
  this.connection.query(query, data, function(err, result) {
    if (err) {throw err};
    util.puts('Daten für ' + tracker.task + ' wurden gespeichert.');
```

```
    this.connection.end();
  }).bind(this));
}
```

Listing 4.76 Die `writeEnd`-Methode

Die Abfrage erfolgt erneut asynchron mit einer entsprechenden Callback-Funktion, in der der Benutzer informiert wird, dass die Operation erfolgreich durchgeführt wurde. Außerdem wird die Verbindung zur Datenbank getrennt.

In Listing 4.77 beschäftigen Sie sich nun mit der Implementierung der `readFile`-Methode. Diese verwenden Sie später dazu, die Statistiken auszugeben.

```
Persister.prototype.readFile = function (period, Tracker) {
  var data = period.split('.');
  data = data[1] + '-' + data[0];
  var query = 'SELECT DATE_FORMAT(`date`, "%d.%m.%Y")
  AS `date`, `startTime`,
    `endTime`, `diff`, `task` FROM `times` WHERE `date` LIKE "' + data +
```

```

    '%";
    this.connection.query(query, function (err, data) {
        var row;
        for (var i = 0; i < data.length; i++) {
            row = [data[i].date, data[i].startTime, data[i].endTime,
                data[i].diff, data[i].task];
            tracker = new Tracker(row.join(' '));
            util.puts(tracker.toString());
        }
        this.connection.end();
    }).bind(this));
}

```

Listing 4.77 Die readFile-Methode

Das Kernstück der `readFile`-Methode besteht aus der Abfrage der Tabelle. Zu diesem Zweck müssen Sie das übergebene Datum so umwandeln, dass es als Suchstring in die Abfrage eingefügt werden kann. Außerdem nutzen Sie in der Abfrage die `DATE_FORMAT`-Funktion von MySQL, um das Datum korrekt formatiert zurückzugeben. Mit den zurückgelieferten Daten aus der asynchronen Abfrage der Datenbank werden Objekte vom Typ `ExtendedTimeTracker` generiert, die `toString`-Methode dieser Objekte wird aufgerufen und das Ergebnis dem Benutzer angezeigt.

Den abschließenden Codeblock des `persister`-Moduls sehen Sie in Listing 4.78.

```

Persister.prototype.getLastLineData = function(cb) {
    var query = 'SELECT `date`, `startTime`, `task` FROM `times` WHERE
        `diff` IS NULL';
    this.connection.query(query, function(err, result) {
        cb(result);
    });
}
module.exports = new Persister();

```

Listing 4.78 Die getLastLineData-Methode

Bisher war die `getLastLineData`-Methode synchron. Das bedeutet, dass die Ausführung der Applikation unterbrochen wurde, bis die gesamte Datei ausgelesen und die letzte Zeile zurückgegeben wurde. Bei der Verwendung des `mysql`-Moduls müssen Sie Ihre Applikation dahingehend umbauen, dass auch dieser Teil asynchron ausgeführt wird. Zu diesem Zweck erhält die `getLastLineData`-Methode eine Callback-Funktion als Argument, die nach Abschluss der Abfrage mit den Ergebnissen als Argumenten aufgerufen wird. Die letzte Zeile gleicht der des ursprünglichen `persister`-Moduls.

Hier wird die Klasse `Persister` instanziiert, das Objekt exportiert und somit nach außen verfügbar gemacht.

Die Änderungen am `persister`-Modul machen weitere Anpassungen in der Applikation erforderlich. So müssen Sie zunächst in der Datei `index.js`, also am Einstiegspunkt in die Applikation, den Code dahingehend anpassen, dass er den nunmehr asynchronen Charakter der `writeEnd`-Methode des `persister`-Moduls widerspiegelt.

```
case 'ende':
    tracker.end();
    break;
```

Listing 4.79 Anpassung der `index.js`-Datei

Wie Sie in Listing 4.79 sehen können, müssen Sie lediglich die Zeile `persister.writeEnd(tracker);` entfernen. Diese Funktionalität verlagern Sie in den `ExtendedTimeTracker`. Die Anpassungen, die Sie hier vornehmen müssen, zeigt Listing 4.80.

```
ExtendedTimeTracker.prototype.end = function () {
    TimeTracker.prototype.end.call(this);
    persister.getLastLineData(this.endCb.bind(this));
}

ExtendedTimeTracker.prototype.endCb = function(data) {
    var etSec = secondCalc(this.endTime);
    var stSec = secondCalc(data[0].startTime);
    var diff = etSec - stSec;
    var hours = diff === 0 ? 0 : Math.floor(diff/60);
    var minutes = diff - hours * 60;
    this.task = data[0].task;
    this.diff = hours + ':' + minutes;
    persister.writeEnd(this);
}
```

Listing 4.80 Anpassung des `ExtendedTimeTracker`-Moduls

Im Zuge des Umbaus des `persister`-Moduls haben Sie die Methode `getLastLineData` dahingehend umgebaut, dass sie nun eine Callback-Funktion akzeptiert und diese ausführt. Die Callback-Funktion ist in diesem Fall der ursprüngliche Inhalt der `end`-Methode. Außerdem steht an deren Ende der Aufruf der `writeEnd`-Methode des `persister`-Moduls, dem das `ExtendedTimeTracker`-Objekt übergeben wird.

Die letzte, eher geringfügige Anpassung müssen Sie im `TimeTracker`-Modul durchführen. Diese Änderung ist erforderlich, damit das Datum korrekt in die Datenbank

geschrieben wird. In Listing 4.81 sehen Sie, wie Sie die `start`-Methode des `TimeTracker`-Moduls anpassen müssen.

```
TimeTracker.prototype.start = function (task) {
    var now = new Date();

    this.date = now.getFullYear() + '-' + (now.getMonth() + 1) + '-'
        + now.getDate();
    this.startTime = timeHelper(now);
    this.task = task;
};
```

Listing 4.81 Anpassung des `TimeTracker`-Moduls

Mit diesen Anpassungen haben Sie erreicht, dass nun statt einer Protokolldatei eine MySQL-Datenbank zur Speicherung der Daten innerhalb Ihrer Applikation verwendet wird. Durch diese Änderung ist die Applikation in der Lage, mit einer größeren Datenmenge umzugehen. Außerdem muss nicht bei jeder Anfrage direkt die gesamte Protokolldatei ausgelesen werden. Dieser Umbau wurde erst möglich durch die Verwendung des `mysql`-Moduls, das Ihnen die entsprechenden Funktionalitäten zur Verfügung stellt. Die Verwendung der Applikation wurde durch die Änderung nicht beeinflusst. Das bedeutet, dass Sie die Applikation weiterhin verwenden können, wie Sie in Listing 4.65 sehen konnten.

Nachdem Sie den Einsatz des Node Package Managers und der Module, die durch ihn installiert werden können, in der Praxis gesehen haben, lernen Sie nun weitere Funktionalitäten des NPM kennen.

4.4.5 Pakete aktualisieren

Viele Module, die auf <http://npmjs.org> verfügbar sind, werden von ihren Autoren aktiv weiterentwickelt. Das bedeutet, dass es in mehr oder weniger regelmäßigen Abständen Aktualisierungen dieser Module gibt. In den folgenden Abschnitten werden Sie erfahren, wie der Aktualisierungsmechanismus des NPM funktioniert. Zu diesem Zweck greifen Sie auf ein weiteres Feature des `npm install`-Kommandos zurück. Dieser Befehl verfügt nicht nur über die Möglichkeit, Module in ihrer aktuellsten Version zu installieren, Sie können auch eine bestimmte Version eines Moduls angeben, die installiert werden soll. Für den Test der Paketaktualisierung nutzen Sie in diesem Fall das `underscore`-Modul, lassen es allerdings nicht in seiner aktuellsten Version, sondern in der älteren Version 1.2.0 installieren. Sie können ein Modul in einer bestimmten Version installieren, indem Sie im `install`-Kommando an den Modulnamen ein `@`-Zeichen und die gewünschte Versionsnummer anhängen. Der Installationsprozess läuft dann wie gewohnt ab. Der NPM fragt das Repository nach den Daten über das

Modul ab, erhält den Pfad zum Tar-Archiv, lädt dieses herunter und fügt es in das lokale `node_modules`-Verzeichnis ein. Ein Test mit dem `npm list`-Befehl zeigt, dass `Underscore.js` wirklich in der Version 1.2.0 installiert wurde. Im nächsten Schritt weisen Sie nun den NPM an, für alle Pakete im aktuellen Projekt nach Aktualisierungen zu suchen und diese zu installieren. Bei diesem Vorgang stellt der NPM dann den Versionsunterschied zwischen der von Ihnen installierten Version 1.2.0 von `Underscore.js` und der aktuellen Version fest. Das bedeutet, dass das Modul aktualisiert werden sollte. Das aktuelle Paket wird aus dem Repository heruntergeladen und installiert. Nach den Operationen, die Sie in Listing 4.82 detailliert mit den entsprechenden Aufgaben finden, liegt also im lokalen `node_modules`-Verzeichnis und somit für Ihre gesamte Applikation die aktuelle Version des `underscore`-Moduls vor.

```
$ npm install underscore@1.2.0
npm http GET https://registry.npmjs.org/underscore/1.2.0
npm http 200 https://registry.npmjs.org/underscore/1.2.0
npm http GET https://registry.npmjs.org/underscore/-/underscore-1.2.0.tgz
npm http 200 https://registry.npmjs.org/underscore/-/underscore-1.2.0.tgz
underscore@1.2.0 node_modules/underscore
$ npm ls
/tmp/node
└─ underscore@1.2.0
$ npm update
npm http GET https://registry.npmjs.org/underscore
npm http 304 https://registry.npmjs.org/underscore
npm http GET https://registry.npmjs.org/underscore/1.4.2
npm http 200 https://registry.npmjs.org/underscore/1.4.2
npm http GET https://registry.npmjs.org/underscore/-/underscore-1.4.2.tgz
npm http 200 https://registry.npmjs.org/underscore/-/underscore-1.4.2.tgz
underscore@1.4.2 node_modules/underscore
```

Listing 4.82 Module aktualisieren

Geben Sie auf der Kommandozeile im Verzeichnis Ihrer Applikation den Befehl `npm update` ohne weitere Optionen an, sucht der NPM für alle lokal in Ihrer Applikation installierten Module nach Aktualisierungen. Möchten Sie nur ein bestimmtes Modul aktualisieren, müssen Sie dazu nur den Namen des Moduls, das Sie aktualisieren möchten, an den Befehl hängen. Um also explizit nur das `underscore`-Modul zu aktualisieren, müssen Sie das Kommando `npm update underscore` absetzen. Normalerweise aktualisiert `npm update` nur die Module innerhalb der Verzeichnishierarchie, in der Sie sich aktuell befinden. Um global installierte Module zu aktualisieren, können Sie, wie bei den übrigen NPM-Kommandos auch, die Option `-g` verwenden, um globale Module zu aktualisieren.

Bisher haben Sie gesehen, wie Sie Module finden, installieren, die installierten Module anzeigen lassen und aktualisieren. Im nächsten Schritt sehen Sie, wie Sie installierte Module wieder entfernen können.

4.4.6 Pakete entfernen

Stellen Sie während der Entwicklung fest, dass Sie ein bestimmtes Modul nicht oder nicht mehr in Ihrer Applikation benötigen, sollten Sie es entfernen. Sie können Pakete ganz einfach manuell entfernen, indem Sie das entsprechende Unterverzeichnis des *node_modules*-Verzeichnisses löschen. Der NPM bietet allerdings auch eine Möglichkeit, Module zu entfernen. Das Kommando `npm uninstall` sorgt dafür, dass der NPM das angegebene Modul aus Ihrer Applikation entfernt. In Listing 4.83 entfernen Sie das *underscore*-Modul, das Sie zuvor lokal installiert haben, wieder aus Ihrer Applikation.

```
$ npm list
/tmp/node
└─ underscore@1.4.2
$ npm uninstall underscore
$ npm list
/tmp/node
└─ (empty)
```

Listing 4.83 Module deinstallieren

Mit dem Befehl `npm uninstall` wird das angegebene Modul deinstalliert. Existiert das Modul nicht, erhalten Sie eine entsprechende Fehlermeldung als Ergebnis der Operation. Für den Fall, dass die Deinstallation erfolgreich war, gibt das Kommando keine Meldung aus. Im Beispiel in Listing 4.83 sehen Sie allerdings, dass ein nachfolgendes `npm list` keine lokal installierten Module findet.

Auch global installierte Module können Sie mit `npm uninstall` entfernen. Das Kommando sorgt nicht nur dafür, dass der Quellcode des Moduls aus */usr/local/lib/node_modules* entfernt wird. Es werden auch alle symbolischen Links, die zur Ausführung eines Kommandozeilenwerkzeugs wie beispielsweise *grunt* erforderlich sind, gelöscht.

4.4.7 Die wichtigsten Kommandos im Überblick

In den letzten Abschnitten haben Sie gesehen, wie Sie mithilfe des Node Package Managers zusätzliche Module für Ihre Node.js-Plattform verwalten können. Tabelle 4.4 fasst für Sie die wichtigsten Befehle zusammen.

Kommando	Kurzschreibweise	Bedeutung
<code>npm search <Modul></code>	<code>se</code>	Durchsucht das Repository nach <Modul>.
<code>npm install <Modul></code>	<code>in</code>	Installiert das <Modul>-Modul.
<code>npm list</code>	<code>ls</code>	Zeigt alle installierten Module.
<code>npm update</code>	<code>up</code>	Aktualisiert alle lokalen Module.
<code>npm uninstall <Modul></code>	<code>rm</code>	Entfernt das <Modul>-Modul.

Tabelle 4.4 Die wichtigsten NPM-Kommandos

In Tabelle 4.4 finden Sie außerdem eine Spalte mit dem Namen »Kurzschreibweise«. Diese hier enthaltenen abkürzenden Schreibweisen für NPM-Kommandos stellen für Sie als Benutzer des Werkzeugs eine Vereinfachung dar, da sie die Befehle noch weiter vereinfachen. Statt `npm uninstall` können Sie beispielsweise auch `npm rm` schreiben. Die Wirkungsweise beider Befehle ist die gleiche.

Wie bereits beschrieben, können Sie bei den Kommandos `install`, `list`, `update` und `uninstall` die Option `-g` oder in der ausführlicheren Schreibweise `--global` anfügen, um Module systemweit zu installieren.

4.4.8 Der Aufbau eines Moduls

Bisher haben Sie die Module, die Sie mit NPM installieren können, lediglich aus Benutzersicht betrachtet. Nun gehen Sie einen Schritt weiter und sehen sich an, wie diese Module intern aufgebaut sind. Der Kern eines Moduls besteht aus der Konfigurationsdatei, der *package.json*-Datei. Wie die Dateieindung andeutet, liegt die Datei im JSON-Format vor. Die wichtigsten Angaben, die auch im `npmjs.org`-Repository durchsuchbar sind, sind die Eigenschaften `name`, `description`, `author` und `keywords`. Als Beispiel für eine konkrete *package.json*-Datei sehen Sie hier die entsprechende Datei von `grunt`. Listing 4.84 zeigt die dazugehörigen Informationen aus dieser Datei und damit das jeweilige Format der Daten.

```
{
  "name": "grunt-cli",
  "description": "The grunt command line interface.",
  "author": "Grunt Team",
  ...
}
```

Listing 4.84 Aufbau einer *package.json*-Datei, Teil 1

Die nächsten Direktiven, die Sie kennenlernen, dienen eher der internen Verwaltung des Moduls durch den NPM. Darunter sind die Daten `version`, also die aktuelle Version

des Pakets, `main`, der Einstiegspunkt in die Applikation, und `bin`, der Speicherort der ausführbaren Datei der Anwendung. Die Eigenschaft `preferGlobal` gibt an, ob das Modul global installiert werden soll. Unter dem Schlüssel `scripts` werden schließlich Informationen über spezielle Einstiegspunkte in die Applikation angegeben. Im Beispiel von `grunt` ist das `test`. So können Sie mit `npm test` die Tests von `grunt` ausführen lassen. Diese Direktiven und die entsprechenden Werte sehen Sie in Listing 4.85.

```
...
"version": "0.1.7",
"bin": "bin/grunt"
"preferGlobal": true,
"scripts": {
  "test": "./bin/grunt test"
},
...
```

Listing 4.85 Aufbau einer `package.json`-Datei, Teil 2

Bei der Installation des `mysql`-Moduls haben Sie bereits gesehen, dass der NPM in der Lage ist, Abhängigkeiten zu anderen Modulen aufzulösen. Diese Abhängigkeiten werden ebenfalls in der `package.json`-Datei definiert. Neben der Angabe `dependencies`, über die Sie Pakete definieren, die für die Funktionsfähigkeit Ihrer Anwendung erforderlich sind, können Sie mithilfe von `devDependencies` Pakete angeben, die zur Weiterentwicklung des Moduls benötigt werden. Diese Module lassen sich über `npm install --dev` installieren. Schließlich können Sie mit `engines` angeben, welche Version von Node.js zur Ausführung der Applikation benötigt wird.

```
...
"dependencies": {
  "nopt": "~1.0.10",
  "findup-sync": "~0.1.0",
  "resolve": "~0.3.1"
},

"devDependencies": {
  "grunt": "~0.4.0",
  "grunt-contrib-jshint": "~0.2.0"
},
"engines": {
  "node": ">= 0.8.0"
},
...
```

Listing 4.86 Aufbau einer `package.json`-Datei, Teil 3

In Listing 4.86 sehen Sie, dass Sie neben den Namen der Pakete, die benötigt werden, auch eine Versionsnummer angeben müssen. Gleiches gilt für `devDependencies` und `engines`. Sie können die Version auf verschiedene Arten angeben. Die einfachste Variante der Versionsangabe besteht darin, dass Sie eine bestimmte Version vorschreiben, indem Sie die Nummer wie beispielsweise `0.10.15` direkt eintragen. Hierbei hat sich allerdings als Best Practice herausgebildet, dass Sie zwar die Major- und die Minor-Version fest definieren, das Patch Level jedoch nicht vorschreiben. Gehen Sie nach dieser Empfehlung vor, erlauben Sie dem NPM damit, dass bei einer Aktualisierung Fehlerbehebungen berücksichtigt und installiert werden. Um dies zu erreichen, müssen Sie lediglich statt des Patch Levels ein `x` eintragen. Das führt dann zu einer Versionsangabe von `0.10.x`. Sie können den Platzhalter allerdings nicht nur für das Patch Level, sondern auch für die Minor-Version verwenden. Möchten Sie die Version nicht beschränken, können Sie statt einer Version ein `*`-Zeichen als Platzhalter für die aktuellste Version eintragen. Die Verwendung von Platzhaltern bei Major- und Minor-Versionen wird jedoch nicht empfohlen, da dies die Stabilität Ihrer Applikation beeinträchtigen kann, weil Sie nicht sicherstellen können, dass sich die Schnittstellen der Module nicht von Version zu Version ändern.

Neben der festen Angabe einer Version und der Verwendung von Platzhaltern haben Sie noch eine Reihe weiterer Möglichkeiten, um Versionsnummern anzugeben. In Tabelle 4.5 sehen Sie die verschiedenen Varianten, wie Sie Versionen angeben können.

Spannen von Versionen können mit einem Leerzeichen kombiniert werden, was eine logische UND-Verknüpfung bedeutet. Die zweite Variante besteht in der Kombination zweier Spannen mit `|`, was bedeutet, dass entweder die erste Spanne oder die zweite Spanne erfüllt sein muss.

Version	Bedeutung
<code>> 1.2.3</code>	jede Version, die größer als 1.2.3 ist
<code>>= 1.2.3</code>	jede Version, die gleich oder größer als 1.2.3 ist
<code>< 1.2.3</code>	jede Version, die kleiner als 1.2.3 ist
<code><= 1.2.3</code>	jede Version, die gleich oder kleiner als 1.2.3 ist
<code>~1.2.3</code>	alle Versionen <code>>= 1.2.3</code> und <code>< 1.3.0</code>
<code>~1.2</code>	alle Versionen <code>>= 1.2.0</code> und <code>< 1.3.0</code>
<code>~1</code>	alle Versionen <code>>= 1.0.0</code> und <code>< 2.0.0</code>

Tabelle 4.5 Angabe von Versionsnummern

Neben den bereits vorgestellten Konfigurationsmöglichkeiten stehen Ihnen bei der Erstellung einer *package.json* noch weitere Direktiven zur Verfügung. Mit den Informationen *homepage*, *repository* und *bugs* geben Sie die Webseite des Projekts an, das Repository, in dem die Applikation entwickelt wird und einen Bugtracker, in den die Benutzer Ihrer Applikation Fehlermeldungen eintragen können. Die Direktiven *licenses*, *readme* und *readmeFilename* geben an, unter welcher Lizenz das Projekt verfügbar ist, außerdem wird der Benutzer kurz in das Projekt eingeführt und darüber informiert, in welcher Datei diese Einführung zu finden ist. In Listing 4.87 sehen Sie, wie die Direktiven im *grunt*-Modul aussehen.

```
...
"homepage": "http://gruntjs.com",
"repository": {
  "type": "git",
  "url": "git://github.com/gruntjs/grunt-cli.git"
},
"bugs": {
  "url": "http://github.com/gruntjs/grunt-cli/issues"
},
"licenses": [
  {
    "type": "MIT",
    "url": "http://github.com/gruntjs/grunt-cli/blob/master/LICENSE-MIT"
  }
],
}
```

Listing 4.87 Aufbau einer *package.json*-Datei, Teil 4

Ansonsten weist das *grunt*-Modul einen Aufbau wie viele Node.js-Module auf. Das *bin*-Verzeichnis enthält die ausführbare Datei, über die Sie *grunt* auf der Kommandozeile ausführen können. Das *lib*-Verzeichnis enthält den Quellcode des Moduls. Die übrigen Verzeichnisse enthalten Hilfsdateien, Tests oder Dokumentationen.

4.4.9 Eigene Pakete erstellen

Im vorangegangenen Abschnitt haben Sie alle Themen kennengelernt, die nötig sind, damit Sie Ihr eigenes Paket für den NPM erstellen können. Sie können sich jetzt daran machen, zuerst die Time-Tracker-Applikation so umzubauen, dass sie als Kommandozeilenapplikation verwendet werden kann, und danach werden Sie ein NPM-Paket daraus erstellen. So erfahren Sie Schritt für Schritt, was im Einzelnen erforderlich ist, um ein Modul auf <http://npmjs.org> zu publizieren.

Der erste Schritt besteht darin, dass Sie ein *bin*-Verzeichnis innerhalb Ihrer Applikation erstellen. In diesem Verzeichnis benötigen Sie eine neue Datei mit dem Namen *tracker*. Diese dient den Nutzern der Applikation später als Kommandozeilenschnittstelle zur Applikation. Da Sie die Applikation ohnehin bereits auf der Kommandozeile betrieben haben, ist die Datei, die als Schnittstelle dient, sehr kurz, wie Sie in Listing 4.88 sehen können.

```
#!/usr/bin/env node
require('../index.js');
```

Listing 4.88 Kommandozeilenschnittstelle für den TimeTracker

Als Nächstes benötigen Sie eine *package.json*-Datei im Wurzelverzeichnis Ihrer Applikation. Listing 4.89 zeigt Ihnen deren Aufbau.

```
{
  "name": "TimeTracker",
  "description": "Time tracking application",
  "author": {
    "name": "Sebastian Springer",
    "email": "sebastian.springer@example.com"
  },
  "keywords": [
    "timeTracker"
  ],
  "version": "0.1.0",
  "main": "index",
  "bin": "bin/tracker",
  "preferGlobal": true,
  "dependencies": {
    "mysql": "~2.0"
  },
  "engines": {
    "node": ">= 0.6.0"
  },
  "homepage": "http://example.com",
  "repository": {
    "type": "git",
    "url": "git://example.com/tracker.git"
  },
  "bugs": {
    "url": "http://example.com/issues"
  },
  "licenses": [
```

```

    {
      "type": "MIT",
      "url": "http://example.com/LICENSE-MIT"
    }
  ]
}

```

Listing 4.89 Aufbau der package.json für den TimeTracker

Die Datei im *bin*-Verzeichnis und die *package.json* sind die einzigen Anpassungen, die Sie vornehmen müssen, um ein vollwertiges Paket für den NPM zu erstellen. Sie können nun Ihr Paket direkt installieren oder ein Tar-Archiv erstellen. Listing 4.90 demonstriert, wie Sie beim Packen und Installieren vorgehen müssen.

```

$ tar cvzf tracker.tgz tracker
a tracker
a tracker/bin
a tracker/index.js
...
$ sudo npm install -g tracker.tgz
npm WARN package.json TimeTracker@0.1.0 No README.md file found!
npm http GET https://registry.npmjs.org/mysql
npm http 304 https://registry.npmjs.org/mysql
npm http GET https://registry.npmjs.org/require-all/0.0.3
npm http 304 https://registry.npmjs.org/require-all/0.0.3
/usr/local/bin/TimeTracker -> /usr/local/lib/node_modules/TimeTracker/bin/
tracker
TimeTracker@0.1.0 /usr/local/lib/node_modules/TimeTracker
└─ mysql@2.0.0-alpha4 (require-all@0.0.3)
$ TimeTracker start 'Entwicklung einer Node.js Applikation'

```

Listing 4.90 Installation und Verwendung des TimeTrackers

Mit dem aktuellen Stand sind Sie nun in der Lage, Ihre Applikation zu verbreiten. Sie müssen dafür nichts anderes tun, als die Datei auf einen Rechner zu kopieren und die Datei mithilfe von `npm install -g tracker.tgz` zu installieren. Da die Applikation auf der Kommandozeile verfügbar sein soll, müssen Sie die Applikation global installieren, wofür Sie Administratorrechte benötigen. Diese Art der Verbreitung von Applikationen ist jedoch recht umständlich, und auch die Reichweite ist sehr gering. Falls Sie die Applikation weiterentwickeln oder einen Fehler entdecken und diesen beheben, müssen Sie diese Veränderung an alle Verwender Ihrer Applikation manuell verteilen. Diese Gründe sprechen dafür, dass Sie Ihre Applikation über `npmjs.org` verbreiten und keine Dateien dazu verwenden.

Wenn Sie Ihre Applikation auf <http://npmjs.org> hochladen, können Sie diese Infrastruktur nutzen, damit andere Benutzer Ihre Applikation über den Node Package Manager installieren können. Außerdem können Sie Aktualisierungen ebenfalls hochladen, und alle Nutzer können diese über `npm update` erhalten. Aber wie können Sie Ihre Applikation über npmjs.org verbreiten? Welche Voraussetzungen müssen Sie dazu erfüllen? Sie benötigen lediglich einen Nutzeraccount auf dem Server und ein Tar-Archiv, das Ihre Applikation enthält.

Den Nutzeraccount können Sie über das Kommando `npm useradd` generieren. Dieses Kommando leitet Sie durch einen interaktiven Prozess, an dessen Ende Sie einen aktiven Nutzeraccount auf npmjs.org besitzen. Listing 4.91 geleitet Sie durch diesen Prozess.

```
$ npm adduser
Username: <Name>
Password:
Email: <E-Mail>
npm http PUT https://registry.npmjs.org/-/user/org.couchdb.user:<Name>
npm http 201 https://registry.npmjs.org/-/user/org.couchdb.user:<Name>
```

Listing 4.91 Erstellung eines Nutzerkontos auf npmjs.org

Mit diesem Benutzerkonto können Sie nun Ihre Applikation publizieren. Diesen Prozess können Sie durch das Kommando `npm publish` anstoßen. Die Ausgabe dieses Kommandos sehen Sie in Listing 4.92. Dabei können Sie entweder das Verzeichnis, in dem sich Ihre Applikation befindet, oder das Tar-Archiv verwenden. Die einzige Voraussetzung ist, dass dieses Verzeichnis beziehungsweise das Archiv die Datei *package.json* enthält.

```
$ npm publish TimeTracker
npm WARN package.json TimeTracker@0.1.0 No README.md file found!
npm http PUT https://registry.npmjs.org/ TimeTracker
npm http 201 https://registry.npmjs.org/ TimeTracker
npm http GET https://registry.npmjs.org/ TimeTracker
npm http 200 https://registry.npmjs.org/ TimeTracker
npm http PUT https://registry.npmjs.org/TimeTracker/0.1.0/-tag/latest
npm http 201 https://registry.npmjs.org/TimeTracker/0.1.0/-tag/latest
npm http GET https://registry.npmjs.org/ TimeTracker
npm http 200 https://registry.npmjs.org/ TimeTracker
npm http PUT https://registry.npmjs.org/TimeTracker/-/TimeTracker-0.1.0.tgz/-
rev/2-0e2f2639cb09d0768871a4dfebdaa13f
npm http 201 https://registry.npmjs.org/TimeTracker/-/TimeTracker-0.1.0.tgz/-
rev/2-0e2f2639cb09d0768871a4dfebdaa13f
+ TimeTracker@0.1.0
```

Listing 4.92 Publizieren der Applikation

4.5 Zusammenfassung

In diesem Kapitel haben Sie vor allem den modularen Aufbau der Node.js-Plattform kennengelernt. Zuerst haben Sie sich mit den internen Modulen von Node.js beschäftigt und gesehen, wofür sie eingesetzt werden können. Mit diesem Wissen haben Sie mit dem TimeTracker eine eigene Applikation auf Basis der internen Node.js-Module erstellt. Da die erste Version dieser Applikation wenig flexibel und skalierbar war, haben Sie im nächsten Schritt den Node Package Manager dazu verwendet, das `mysql`-Modul zu installieren und die Applikation dahingehend umzubauen, dass eine MySQL-Datenbank zum Speichern der Daten verwendet wird. Im Zuge dieser Umbaumaßnahmen haben Sie sich außerdem mit den Möglichkeiten des NPM beschäftigt, und schließlich haben Sie mit dem TimeTracker ein eigenes Paket erstellt und dieses auf npmjs.org publiziert.

Im nächsten Kapitel erfahren Sie, wie Sie mit Hilfe der Node.js-Plattform auf die Dateien und das Dateisystem des Rechners zugreifen können.

Kapitel 5

Arbeiten mit Dateien

Wer glaubt, etwas zu sein, hat aufgehört, etwas zu werden.
– Sokrates

Im vorangegangenen Kapitel konnten Sie bereits sehen, dass es sich beim Umgang mit Dateien um eine zentrale Problemstellung in vielen Applikationen handelt. Sie verwenden Dateien, um Ihren Quellcode zu speichern, und binden ihn über das Modulsystem der Node.js-Plattform wieder ein. Ein weiteres Einsatzgebiet haben Sie ebenfalls bereits in Kapitel 4, »Anpassung und Erweiterung«, kennengelernt. Dateien werden häufig in einer Applikation dazu verwendet, um Informationen zu persistieren und sie so dauerhaft verfügbar zu machen. Entwickeln Sie also eine Applikation, haben Sie immer die Möglichkeit, die Daten, die innerhalb Ihrer Applikation entstehen, in einer Datei zu speichern. Aber auch das Auslesen von Dateien kommt häufig zum Einsatz, und sei es nur, um Konfigurationsinformationen für Ihre Applikation vorzuhalten.

Warum aber sollten Sie Dateien verwenden, wenn Sie Ihre Informationen um ein Vielfaches schneller und bequemer im Arbeitsspeicher Ihres Rechners ablegen können? Dateien haben gegenüber dem Arbeitsspeicher den entscheidenden Vorteil, dass ihr Inhalt auch nach einem Neustart noch verfügbar ist. Ein Nachteil von Dateien ist in der Tat, dass sie umständlicher in der Verwendung sind als beispielsweise der Arbeitsspeicher. Auf Dateien können Sie nicht ohne Weiteres wahlfrei zugreifen. Entweder greifen Sie sequenziell auf die Datei zu und lesen sie Stück für Stück, oder Sie wissen genau, an welcher Stelle sich die gewünschte Information befindet und können diese dann direkt extrahieren. Sämtliche dieser Aufgaben, also sowohl das Lesen als auch das Schreiben von Dateien innerhalb einer Applikation, werden Sie in den folgenden Abschnitten aus der theoretischen wie auch aus der praktischen Sichtweise kennenlernen.

Zuerst möchte ich Ihnen noch einige Hinweise zum Umgang mit Dateien mit auf den Weg geben. Wenn Sie Dateien schreiben oder lesen möchten, sollten Sie beachten, dass Sie mit den Berechtigungen des Benutzers arbeiten, unter dem die Applikation aktuell ausgeführt wird. Ein weiterer wichtiger Punkt, den Sie beachten müssen, besteht darin, den korrekten Pfad zu verwenden. Sie haben stets die Möglichkeit, die Pfade absolut oder relativ anzugeben. Dabei sollten Sie allerdings bedenken, dass

eine relative Angabe von Pfadinformationen mehr zur Portierbarkeit Ihrer Applikation auf andere Systeme beiträgt. Die einzelnen Bestandteile des Pfades, also die Verzeichnisse und der Dateiname, werden durch den sogenannten Pfadtrenner separiert. Das eigentliche Problem hierbei ist, dass sich dieser Pfadtrenner auf den verschiedenen Systemen unterscheidet. So verwenden Windows-Systeme den Backslash, wohingegen unter UNIX-basierten Systemen wie Linux oder Mac OS X ein normaler Schrägstrich zum Einsatz kommt. Abhilfe für dieses Problem schafft die Eigenschaft `sep` des `path`-Moduls. Diese enthält den für das aktuell verwendete System korrekten Pfadtrenner. Damit Sie sich allerdings nicht um die Erstellung korrekter Pfade innerhalb Ihrer Applikation kümmern müssen, stellt Ihnen die Node.js-Plattform über das `path`-Modul mit der `join`-Methode eine elegante Möglichkeit zur Verfügung, um mit Pfaden umzugehen. In Listing 5.1 sehen Sie anhand eines kurzen Beispiels die Verwendung dieser Methode.

```
var path = require("path");
console.log(path.join(path.sep, "usr", "local", "bin", "node"));
```

Listing 5.1 Die Verwendung der `path.join`-Methode

Im Beispiel wird die `join`-Methode dazu verwendet, den Pfad `/usr/local/bin/node` mit dem korrekten Pfadtrenner aufzubauen. Bei absoluten Pfaden ergibt sich natürlich das Problem, dass diese auf UNIX-basierten Systemen mit einem Schrägstrich beginnen und auf Windows-Systemen mit der Partition, die verwendet werden soll, als Präfix, also beispielsweise `C:`. Um diese Unterscheidung müssen Sie sich aktuell noch selbst kümmern. Die einfachste Art, Probleme dieser Art zu vermeiden, ist die bereits erwähnte Verwendung von relativen Pfaden, die Ihre Applikation ohnehin um einiges stabiler macht.

Neben Berechtigungen und Pfaden möchte ich Ihnen noch eine weitere Anmerkung zu den Dateisystemoperationen unter Node.js mit auf den Weg geben. Für alle Operationen, außer der `watch`-Funktionalität, existiert sowohl eine synchrone als auch eine asynchrone Version der jeweiligen Methode. Was bedeutet das für Sie als Entwickler nun konkret? Eine synchrone Dateisystemoperation heißt für Ihre Applikation, dass Sie einen Aufruf absetzen, um beispielsweise eine Datei auszulesen. Dieses Kommando wird an die Node.js-Plattform weitergegeben und anschließend an das Betriebssystem weitergereicht, das die Leseoperation schließlich ausführt und das Ergebnis an Node.js und damit an Ihre Applikation zurückgibt. Während dieses Prozesses wird Ihre Applikation nicht weiter ausgeführt, der gesamte Thread ist blockiert. Liegt das Ergebnis vor, läuft die Applikation ganz normal weiter. Kommen in Ihrer Applikation häufig Dateisystemoperationen vor oder müssen Sie Operationen an großen Dateien durchführen, können sich die synchronen Methoden des `fs`-Moduls durch die Unterbrechungen der Programmausführung negativ auf Ihre Applikation

auswirken. Um dies zu umgehen, existieren die asynchronen Methoden. Sie arbeiten auf die gleiche Weise wie ihre synchronen Gegenstücke, nur dass in diesem Fall das Ergebnis der Operation nicht als Rückgabewert an Ihre Applikation zurückgegeben, sondern durch eine Callback-Funktion verarbeitet wird. Diese Callbacks haben in Node.js stets eine ähnliche Signatur. Das erste Argument ist meist ein Fehler-Objekt, das nur einen Wert aufweist, wenn bei der Operation Probleme aufgetreten sind. Die weiteren übergebenen Werte sind das Ergebnis der Operation, mit denen Sie innerhalb Ihres Callbacks umgehen können. Das Beispiel in Listing 5.2 verdeutlicht Ihnen diesen Sachverhalt.

```
var fs = require('fs');

fs.readFile('/tmp/test.txt', function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data.toString());
  }
});
```

Listing 5.2 Callback-Funktionen in Node.js

5.1 Dateien lesen

Im vorangegangenen Abschnitt haben Sie sich bereits kurz damit beschäftigt, die Inhalte einer Datei auszulesen. In diesem Abschnitt stelle ich Ihnen verschiedene Möglichkeiten vor, wie Sie dies umsetzen können. Zuerst lernen Sie die klassische Art kennen, eine Datei auszulesen. Bei dieser Methode erstellen Sie im ersten Schritt einen Datei-Handle, nutzen diesen, um die Datei auszulesen, und schließen den Datei-Handle anschließend wieder. Sollten Sie bereits Erfahrung in der Programmierung mit anderen Programmiersprachen wie beispielsweise C oder PHP haben, kommt Ihnen diese Art des Dateizugriffs zu Recht bekannt vor. Im Normalfall passieren die einzelnen Schritte synchron. Das bedeutet, dass Sie die Kommandos ausführen und die Ausführung des Programms so lange unterbrochen wird, bis ein Rückgabewert vorliegt, mit dem Sie weiterarbeiten können. Unter Node.js steht Ihnen, wie bereits beschrieben, neben der synchronen Variante auch die Möglichkeit zur Verfügung, diese Operationen asynchron durchzuführen.

Den Einsatz der verschiedenen Operationen, mit deren Hilfe Sie Dateien auslesen können, sehen Sie im Folgenden an einem konkreten Beispiel. Die Applikation soll dazu dienen, Dateien auszulesen, deren Inhalt im *ini*-Format vorliegt. Dieses Dateiformat wird häufig für Konfigurationsdateien verwendet, da es im Gegensatz zu XML

sehr einfach und leichtgewichtig ist. Im Beispiel soll der Inhalt einer *ini*-Datei in eine Objektstruktur umgewandelt werden, die den Zugriff auf die einzelnen Werte erlaubt. Unterstützt werden sollen Sektionen und Schlüssel-Wert-Paarungen. Um das Beispiel einfach zu halten, werden Kommentare und die Vererbung von Sektionen nicht unterstützt. In Listing 5.3 sehen Sie ein Beispiel einer *ini*-Datei, die durch die Applikation verarbeitet werden soll.

```
[production]
  user=root
  password = mySecret
  host = "localhost"
  database="mysql"

[development]
user=root

password = mySecret

host = "localhost"

database="mysql"
```

Listing 5.3 Eingabedatei im ini-Format

Der *ini*-Parser, den Sie im weiteren Verlauf implementieren werden, soll aus der Struktur in Listing 5.3 eine Objekthierarchie erstellen, sodass Sie beispielsweise über `production.password` auf den entsprechenden Wert zugreifen können. Als kleine Herausforderung sollen die doppelten Anführungszeichen nicht mit übernommen werden. Außerdem sollen leere Zeilen und Leerzeichen vor und nach den Schlüsseln und Werten ignoriert werden. In Listing 5.4 sehen Sie den gesamten Quellcode für das Beispiel, den ich Ihnen im Anschluss Schritt für Schritt näherbringen werde.

```
var fs = require('fs'),
    filename = 'config.ini';

fs.open(filename, 'r', function (err, handle) {
  fs.stat(filename, function (err, stats) {

    var size = stats.size,
        buffer = new Buffer(size);

    fs.read(handle, buffer, 0, size, 0, function (err, bytes, content) {
      var config = content.toString().split("\n");
```

```

var result = {};
var section= /^s*\[s*([^\]]*)\s*\]\s*$/;
var currentSection = 'default';
var tempSection;
var keyVal;
var key;
var value;

for (var i = 0; i < config.length; i += 1) {

    tempSection = config[i].match(section);

    if (tempSection !== null) {
        currentSection = tempSection[1];
        result[currentSection] = {};
    } else if (config[i].trim() !== '') {
        keyVal = config[i].split('=');
        key = filterString(keyVal[0]);
        value = filterString(keyVal[1])
        result[currentSection][key] = value;
    }
}

fs.close(handle);
console.log(result);

});
});
});

var filterString = function(input) {
    var output = input.trim();
    output = output.replace(/"/g, '');
    return output;
};

```

Listing 5.4 ini-Parser

Von besonderer Bedeutung sind in diesem Beispiel die Aufrufe der Methoden `open`, `read` und `close`, die durch das `fs`-Modul zur Verfügung gestellt werden.

Der Einstieg in die Applikation erfolgt mit einem Aufruf der `open`-Methode. Als ersten Wert akzeptiert `open` den Namen der Datei, die geöffnet werden soll. Der zweite Wert gibt an, wie die Datei geöffnet werden soll, das sogenannte Flag. Hier stehen Ihnen

mehrere Möglichkeiten zur Auswahl. Tabelle 5.1 zeigt die verschiedenen Flags. Das dritte Argument ist schließlich die Callback-Funktion. Sie erhält als Argumente ein Objekt, das etwaige Fehler enthält, die beim Öffnen aufgetreten sind, und den Datei-Handle, über den Sie auf die Datei zugreifen können.

Flag	Bedeutung
r	Datei wird nur für lesenden Zugriff geöffnet.
r+	Datei wird für lesenden und schreibenden Zugriff geöffnet.
rs	synchroner Lesemodus, der den lokalen Dateisystemcache umgeht
rs+	synchroner Schreib- und Lesemodus
w	Datei wird nur für schreibenden Zugriff geöffnet. Datei wird erstellt beziehungsweise geleert.
wx	Datei wird für exklusiv schreibenden Zugriff geöffnet.
w+	Datei wird für schreibenden und lesenden Zugriff geöffnet. Datei wird erstellt beziehungsweise geleert.
wx+	Öffnet die Datei wie mit wx, allerdings für exklusiven Zugriff.
a	Öffnet die Datei, um Daten anzuhängen, und erstellt die Datei, falls nötig.
ax	Wie a, nur dass die Datei exklusiv geöffnet wird.
a+	Öffnet die Datei, um Daten anzuhängen, und für lesenden Zugriff.
ax+	wie a+, jedoch mit exklusivem Zugriff auf die Datei.

Tabelle 5.1 Zugriffsflags der open-Methode

Da die Anforderungen für das Beispiel lediglich das Auslesen von *ini*-Dateien vorsehen, reicht es aus, wenn Sie die Datei mit dem *r*-Flag nur für lesenden Zugriff öffnen. Über das *handle*-Argument des Callbacks können Sie dann über verschiedene Methoden des *fs*-Moduls auf den Inhalt der Datei zugreifen. Versuchen Sie allerdings, mit der *write*-Methode innerhalb des Callbacks der *open*-Methode in die Datei zu schreiben, erhalten Sie einen Fehler, da der Datei-Handle nicht schreibbar ist.

Im nächsten Schritt kommt mit der *stat*-Methode ein weiteres Feature des *fs*-Moduls zum Einsatz. Sie benötigen diese Methode, um die Informationen über die Konfigurationsdatei auszulesen. Im Fall des Beispiels lesen Sie die Dateigröße aus, die Sie später benötigen, um den Puffer, der die Daten aufnehmen soll, richtig zu dimensionieren. Auch die *stat*-Methode ist wieder asynchron, das bedeutet für Sie einen weiteren Callback, in dem Sie über das zweite Argument Zugriff auf die Statistik

der Datei haben. Neben der Callback-Funktion, dem zweiten Argument dieser Methode, ist der Dateiname das Einzige, was Sie dieser Methode übergeben müssen. Im Callback der `stat`-Methode erstellen Sie schließlich ein Buffer-Objekt, das den Inhalt der *ini*-Datei aufnehmen soll. Da Buffer-Objekte nicht dynamisch wachsen können, müssen Sie bereits bei der Erstellung die Größe festlegen. Den korrekten Wert erhalten Sie aus der `size`-Eigenschaft des zweiten Arguments des `stat`-Callbacks.

Mit dem Buffer-Objekt und dem Datei-Handle haben Sie nun sämtliche Voraussetzungen, um die *ini*-Datei auszulesen. Die Argumente der `read`-Methode, die diese Aufgabe für Sie übernimmt, sind der Handle, der Buffer, dann die Stelle im Buffer, an der mit dem Schreiben begonnen werden soll, in Ihrem Fall die Zahl 0, die bedeutet, dass am Anfang des Buffers begonnen werden soll. Danach folgt die Anzahl an Bytes, die gelesen werden sollen. Hier sollten Sie erneut auf die `size`-Eigenschaft aus dem `stat`-Callback zurückgreifen. Bevor es an den Callback geht, der die eingelesenen Daten verarbeitet, müssen Sie noch die Position der Datei angeben, ab der gelesen werden soll. Hier sollten Sie abermals den Wert 0 verwenden, um am Anfang der Datei mit dem Lesen zu beginnen. Das letzte Argument ist schließlich, wie schon angekündigt, die Callback-Funktion. Diese erhält drei Argumente. Das erste ist, wie bei Node.js üblich, ein Objekt, das eventuelle Fehlermeldungen enthält. Das zweite ist die Anzahl an Bytes, die gelesen wurden. Das dritte und für Sie relevante Argument ist schließlich das Buffer-Objekt, das den ausgelesenen Inhalt enthält.

Innerhalb der Callback-Funktion werden zuerst einige Variablen deklariert. Dabei wird das Buffer-Objekt in eine Zeichenkette umgewandelt und anschließend in ein Array zerlegt, das die einzelnen Zeilen der Datei enthält. Über diese Zeilen wird anschließend iteriert und anhand eines regulären Ausdrucks entschieden, ob es sich bei der Zeile um die Definition einer Sektion oder um ein Schlüssel-Wert-Paar handelt. Leere Zeilen werden bei dieser Operation ignoriert. Die Sektionen bilden dabei die erste Ebene im Ausgabeobjekt. Die Schlüssel-Wert-Paare werden dann entsprechend der zugehörigen Sektion zugeordnet.

Einen geöffneten Datei-Handle sollten Sie stets schließen, wenn er nicht mehr benötigt wird. Mit dem Schließen des Datei-Handles sorgen Sie dafür, dass die Datei nicht mehr irrtümlich verwendet werden kann. Dies geschieht auch im *ini*-Parser mit der `close`-Methode, nachdem über alle Zeilen iteriert wurde. Diese Methode erhält als einziges Argument den Datei-Handle.

Mit der Hilfsfunktion `filterString` werden unnötige Leerzeichen und Anführungszeichen entfernt, sodass damit sämtliche Anforderungen erfüllt sind. Die Ausgabe Ihrer Applikation, die Sie über den Aufruf von `console.log` erzeugen, sehen Sie in Listing 5.5.

```

$ node readFile.js
{ production:
  { user: 'root',
    password: 'mySecret',
    host: 'localhost',
    database: 'mysql' },
  development:
  { user: 'root',
    password: 'mySecret',
    host: 'localhost',
    database: 'mysql' }
}

```

Listing 5.5 Ausgabe des ini-Parsers

Die ganze Applikation lässt sich erheblich verkürzen, wenn Sie statt der Kombination aus `open`, `read` und `close` nur die Methode `readFile` verwenden. Diese Methode kapselt alle drei Methoden. Die Konsequenz für Sie besteht darin, dass Sie den Inhalt der Callback-Funktion, die Sie für die `read`-Methode verwenden, unverändert auch in `readFile` benutzen können.

```

var fs = require('fs'),
    filename = 'config.ini';

fs.readFile(filename, function (err, content) {
    var config = content.toString().split("\n"),
    ...
    console.log(result);
});

```

Listing 5.6 ini-Parser mit readFile

Wie Sie in Listing 5.6 sehen können, fallen sämtliche Aufrufe der bereits erwähnten Methoden weg. Das zweite Argument der `readFile`-Methode ist, wie bereits bei der `read`-Methode, ein Buffer-Objekt. Dadurch können die ursprüngliche Callback-Funktion und die Hilfsfunktion wiederverwendet werden. Bisher wurde das erste Argument der Callback-Funktionen bei der Entwicklung ignoriert. Im Normalfall hat dieses Argument den Wert `null`. Tritt allerdings ein Fehler auf, finden Sie nähere Informationen dazu in diesem Objekt. Sie können die Fehlerbehandlung beim Umgang mit Dateien ganz einfach testen. In Listing 5.7 sehen Sie, wie Sie derartige Fehler hervorrufen und mit ihnen umgehen können.

```
var fs = require('fs');
fs.open('/root/test', 'r', function (err, handle) {
  if (err) throw err;
});
```

Listing 5.7 Fehler beim Lesen einer Datei

In diesem Beispiel sehen Sie, wie Fehler im Dateisystemmodul von Node.js entstehen können. Mit diesem Codestück versuchen Sie beispielsweise, lesend auf eine Datei im Heimatverzeichnis des Systemadministrators zuzugreifen. Solange Sie Ihre Applikationen unter Node.js nicht mit Administratorberechtigungen ausführen, was Sie keinesfalls tun sollten, haben Sie keinen Zugriff auf dieses Verzeichnis. Führen Sie den Beispielcode aus, erhalten Sie folgende Fehlermeldung: `Error: EACCES, open '/root/test'`. Ähnlich ist das Verhalten des `fs`-Moduls, wenn Sie versuchen, aus einer nicht existierenden Datei zu lesen. Die Fehlermeldung, die Sie in diesem Fall erhalten, lautet: `Error: ENOENT, open '/tmp/nonexistent'`. Je nachdem, wo Sie die Fehlerbehandlung durchführen möchten, können Sie dies an verschiedenen Stellen tun. Einerseits können Sie die Fehler direkt dort, wo sie auftreten, nämlich in Ihrer Callback-Funktion selbst, behandeln, oder aber Sie werfen, wie Sie es in Listing 5.7 sehen, das Fehler-Objekt mithilfe von `throw` weiter und verlassen sich darauf, dass sich eine andere Stelle im Quellcode Ihrer Applikation darum kümmern wird, mit diesem Fehler umzugehen. Die dritte und schlechteste Möglichkeit, die Ihnen beim Auftreten von Fehlern zur Verfügung steht, ist, dass Sie den Fehler einfach ignorieren.

Wie Sie im Verlauf dieses Buches bereits sehen konnten, ist es mit der Node.js-Plattform nicht nur möglich, aus Dateien zu lesen, sondern Sie können auch Informationen in Dateien schreiben.

5.2 Dateien schreiben

Wenn Sie Applikationen für die Node.js-Plattform erstellen, erhalten Sie meist Eingaben, die entweder direkt von Benutzern oder auch von anderen Applikationen stammen. Auf Basis dieser Eingaben führen Sie dann Berechnungen durch, aggregieren die Informationen oder wollen die Daten einfach nur festhalten. Sie können unter Node.js sämtliche Informationen in Variablen speichern. Diese liegen im Arbeitsspeicher, und Sie können sehr schnell sowohl lesend als auch schreibend darauf zugreifen. Möchten Sie allerdings die Informationen über einen längeren Zeitraum persistieren, müssen Sie auf andere Speichermethoden zurückgreifen, und hier kommen nun Dateien ins Spiel.

Das Schreiben in Dateien gestaltet sich ähnlich zum Auslesen. Auch hier werden die Methoden `open` und `close` verwendet, um den Zugriff vorzubereiten beziehungs-

weise den Datei-Handle nach Gebrauch wieder zu schließen. Das `fs`-Modul stellt Ihnen für den schreibenden Zugriff auf Dateien mehrere Methoden zur Verfügung. Im ersten Schritt sehen Sie den Einsatz der `write`-Methode.

Auch das Schreiben in eine Datei lernen Sie anhand eines kleinen Beispiels kennen. Zu diesem Zweck erstellen Sie nun einen sogenannten Logger. Das ist eine kleine Applikation beziehungsweise ein Modul für größere Applikationen, mit dessen Hilfe Sie Informationen in eine Datei schreiben können. Logger kommen in Applikationen immer dann zum Einsatz, wenn es darum geht, Ereignisse im Betrieb oder während der Entwicklung zur späteren Analyse festzuhalten. So können Sie beispielsweise beim Auftreten von Fehlern die entsprechende Meldung in die Protokolldatei schreiben lassen.

Die wichtigste Anforderung an den Logger ist, dass er möglichst leichtgewichtig einzubinden ist, damit er den Quellcode der Applikation nicht unnötig unleserlich macht. Der Logger soll außerdem die Möglichkeit bieten, Einträge mit verschiedenen Prioritäten, wie beispielsweise Fehler, Warnung oder Information, zu schreiben. Eine Applikation wird normalerweise in verschiedenen Stufen eingesetzt. So soll es für die Entwicklung möglich sein, mehr Meldungen zu schreiben, als es im Produktivbetrieb in der Regel der Fall ist. Um diese Anforderung zu erfüllen, ist es notwendig, dass Sie den Logger so gestalten, dass Sie konfigurieren können, welche Arten von Meldungen protokolliert werden sollen.

In Listing 5.8 können Sie den Quellcode des Loggers nachvollziehen. Im Folgenden erhalten Sie zu den einzelnen Schritten der Implementierung weitere Erklärungen.

```
var fs = require('fs'),
    EventEmitter = require('events').EventEmitter,
    util = require('util');

var EventLogger = function (file, levels) {
  this.file = file;
  this.levels = levels;

  this.on('error', this.log.bind(this, 'ERR'));
  this.on('warning', this.log.bind(this, 'WARN'));
  this.on('info', this.log.bind(this, 'INFO'));
};

util.inherits(EventLogger, EventEmitter);

Object.defineProperty(EventLogger.prototype, 'file', {
  set: function (value) {this._file = value;},
  get: function () {return this._file;}
```

```

});

Object.defineProperty(EventLogger.prototype, 'levels', {
  set: function (value) {this._levels = value;},
  get: function () {return this._levels;}
});

EventLogger.prototype.log = function (level, data) {

  if (this.levels.indexOf(level) > -1) {
    fs.open(this.file, 'a', function (err, handle) {

      var now = new Date(),
          dateString = util.format('%s-%s-%s %s:%s',
            now.getFullYear(),
            (now.getMonth() + 1),
            now.getDate(),
            now.getHours(),
            now.getMinutes(),
            now.getSeconds()),
          buffer = new Buffer(util.format('%s (%s) %s\n',
            dateString, level, data));
      fs.write(handle, buffer, 0, buffer.length, null, function (err,
        written, buffer) {
        fs.close(handle);
      });
    });
  }
};

var logger = new EventLogger('error.log', ['ERR', 'WARN']);

logger.emit('error', 'Something happened');
logger.emit('warning', 'Something else happened');
logger.emit('info', 'Not relevant');

```

Listing 5.8 Dateien schreiben mit dem EventLogger

Für die Umsetzung des Loggers benötigen Sie neben dem `fs`-Modul außerdem noch das `events`- und das `util`-Modul. Nach dem Einbinden dieser Komponenten ist der nächste Schritt, den Sie unternehmen müssen, um den Logger zu implementieren, die Definition der Konstruktor-Funktion.

Als Argumente erhält der Konstruktor zwei Werte. Der erste Wert gibt den Pfad zur Datei an, in die die Protokolleinträge geschrieben werden sollen. Das zweite Argument ist ein Array, das die Kategorien von Meldungen enthält, die protokolliert werden sollen. In Ihrem Fall sind dies `ERR` und `WARN`. Innerhalb des Konstruktors wird der Dateiname für die spätere Verwendung festgehalten. Bitte beachten Sie hierbei, dass es sich bei dem Dateinamen um einen gültigen Pfad handeln muss. Außerdem werden die Kategorien in eine Eigenschaft des aktuellen Objekts gespeichert.

Damit Sie den Logger möglichst leichtgewichtig einsetzen können, sollten Sie ihn auf einer eventgetriebenen Basis entwickeln. Der erste Schritt besteht darin, dass Sie den `EventLogger` von der Klasse `EventEmitter` des `events`-Moduls erben lassen. Damit implementiert der Logger bereits die wichtigsten Methoden für einen eventgetriebenen Ansatz. Konkret sind dies die Methoden `on`, um auf Ereignisse zu reagieren, und `emit`, um Ereignisse zu publizieren. Die Vererbung der Struktur des `EventEmitters` erreichen Sie durch die Verwendung der `inherits`-Methode des `util`-Moduls. Der Aufruf dieser Methode erfolgt direkt nach dem Konstruktor und noch vor der Definition der übrigen Methoden des `EventLoggers`.

Da Ihnen durch die Ableitung von der Klasse `EventEmitter` nun die `on`-Methode zur Verfügung steht, können Sie diese im Konstruktor dazu verwenden, an bestimmte Ereignisse konkrete Methodenaufrufe zu binden. Damit die aufgerufenen Methoden im korrekten Kontext ablaufen, sollten Sie die `bind`-Methode einsetzen. Diese sorgt dafür, dass `this` innerhalb der gebundenen Methode auf das korrekte Objekt referenziert. In diesem konkreten Fall wird bei jedem Ereignis die `log`-Methode des `EventLoggers` aufgerufen. Mit ihrer Hilfe wird schließlich der entsprechende Protokolleintrag geschrieben. Zu diesem Zweck akzeptiert die Methode zwei Argumente. Das erste Argument stellt den Typ des Protokolleintrags dar, das zweite die Nachricht, die geschrieben werden soll. Um das erste Argument bereits beim Binden des Ereignisses an den entsprechenden Callback zu definieren, wird eine Technik verwendet, die *Currying* genannt wird. Diese bindet einen Wert als Argument an eine Funktion und gibt diese als Rückgabewert zurück, sodass das erste Argument fixiert ist und nur noch die folgenden variabel sind. In Ihrem Fall binden Sie die Art des Ereignisses, also beispielsweise `ERR`, fest an den Aufruf. Die Nachricht, die in die Protokolldatei geschrieben werden soll, variiert dann je nachdem, welche Nachricht Sie zur Verfügung stellen.

Der Zugriff auf die Eigenschaften `file` und `levels` sollte normalerweise nicht direkt erfolgen. Stattdessen sollten Sie Getter- und Setter-Methoden für diesen Zugriff implementieren. Hierzu können Sie auf zwei Alternativen zurückgreifen. Sie können die vier notwendigen Methoden einerseits selbst implementieren, oder Sie greifen auf die `defineProperty`-Methode der `Object`-Klasse zurück und fangen den Zugriff auf die Eigenschaft ab. Die zweite Variante sehen Sie im Beispiel in Listing 5.8. Hierbei speichern Sie die Werte in einer internen Hilfsvariablen, der ein Unterstrich vorangestellt wird, um per Konvention zu signalisieren, dass auf diese Eigenschaft nicht direkt zugegriffen werden sollte.

Das Kernstück des Loggers bildet natürlich die Methode `log`. Sie sorgt dafür, dass die korrekten Nachrichten in die Datei geschrieben werden. Um die Anforderungen an den Logger zu erfüllen, sollten Sie im ersten Schritt, wie im Beispiel zu sehen ist, prüfen, ob die gewünschte Meldung überhaupt festgehalten werden soll. Danach kommt eine Kombination aus `open`, `write` und `close` zum Einsatz. Wie Sie bereits im vorherigen Abschnitt erfahren haben, öffnen Sie die Datei mithilfe der `open`-Methode. In diesem Beispiel müssen Sie allerdings ein anderes Zugriffsflag wählen als im letzten Beispiel. Statt des `r` für lesenden Zugriff kommt hier das `w` für den schreibenden Zugriff zum Einsatz. Innerhalb der Callback-Funktion bauen Sie dann die Zeichenkette zusammen, die schließlich im Log festgehalten werden soll. Das Format sollte folgendermaßen aussehen: das Datum und die Uhrzeit, gefolgt von der Art der Meldung und schließlich die eigentliche Nachricht. Da die `write`-Methode die Zeichenkette nur in die Datei schreibt und diese Operation nicht zeilenweise erfolgt, sollten Sie die Escapesequenz für einen Zeilenumbruch, `\n`, anhängen. Die so formatierte Zeichenkette müssen Sie anschließend in ein Buffer-Objekt einbetten, das Sie als zweites Argument an die `write`-Methode übergeben. Das erste Argument ist der Datei-Handle, den Sie innerhalb der Callback-Funktion der `open`-Methode als Argument zur Verfügung haben. Die nächsten beiden Argumente der `write`-Methode dienen dazu, den korrekten Abschnitt aus dem Buffer-Objekt auszuwählen, der geschrieben werden soll. In Ihrem Fall ist der Offset 0 und die Länge der zu schreibenden Zeichenkette die gesamte Länge des Buffer-Objekts. Das vorletzte Argument ist schließlich die Position innerhalb der Datei, an die geschrieben werden soll. Wählen Sie, wie im Beispiel von Listing 5.8, den Wert `null`, bedeutet dies, dass Sie an die aktuelle Position, also ans Ende der Datei, schreiben möchten. Das letzte Argument ist schließlich eine Callback-Funktion. Diese sollten Sie dazu nutzen, den Datei-Handle mithilfe der `close`-Methode wieder zu schließen.

Mit diesem Quellcode ist nun der Logger voll funktionsfähig und kann verwendet werden. Wie dies funktioniert, sehen Sie in den letzten Zeilen des Beispiels in Listing 5.8. Im ersten Schritt müssen Sie den Logger instanziiieren und dem Konstruktor dabei den Namen der Datei und die Arten der Meldungen, die Sie festhalten wollen, im Beispiel sind dies `ERR` und `WARN`, übergeben. Danach können Sie den Logger über die `emit`-Methode, die ursprünglich aus der Klasse `EventEmitter` stammt, ansprechen. Die Events, die Sie in Ihrer konkreten Implementierung verwenden können, sind `error`, `warning` und `info`. Im Beispiel sehen Sie den Einsatz dieser drei Events, wobei durch die Konfiguration, die Sie bei der Instanziierung vorgenommen haben, lediglich die Events `error` und `warning` geloggt werden.

Der Vorteil dieser Implementierung eines Loggers ist, dass die Operationen asynchron und eventgesteuert passieren. Das bedeutet für Sie konkret, Sie setzen den Aufruf der `emit`-Methode ab, und Ihre Applikation wird danach ganz normal weiter ausgeführt, ohne zu blockieren. Sie müssen sich also um nichts weiter kümmern.

Eine mögliche Änderung wäre nun im nächsten Schritt ein kleiner Umbau, damit Sie den Logger als Modul verwenden können. Hierzu sollten Sie die Instanziierung und die Aufrufe der `emit`-Methode entfernen und stattdessen die Klasse `EventLogger` per `module.exports` zur späteren Verwendung als Modul exportieren. Mit diesen Anpassungen können Sie den Logger dann ganz einfach per `require` in Ihre Applikation einbinden, ihn instanzieren und verwenden, um beispielsweise beim Auftreten von Fehlern eine entsprechende Meldung in die Logdatei Ihrer Applikation zu schreiben.

Ähnlich wie schon beim Auslesen von Dateien existieren auch für das Schreiben in Dateien weitere Methoden, die Sie einsetzen können. Interessant sind in diesem Zusammenhang vor allem die Methoden `writeFile` und `appendFile`. In der Funktionsweise gleichen sich beide Methoden, außer, dass `writeFile` die Datei in ihrer Gesamtheit überschreibt, wohingegen `appendFile` die Daten lediglich an die bereits bestehende Datei anhängt. Intern stellen beide Methoden eine Kapselung der drei Methodenaufrufe `open`, `write` und `close` dar und dienen aus diesem Grund eher der Vereinfachung Ihres Applikationscodes. Im `EventLogger` sollten Sie die Methode `appendFile` verwenden, da Sie kontinuierlich Daten an die Logdatei anhängen möchten und keine Informationen überschreiben und verlieren wollen. Die Anpassungen, die Sie am Quellcode des `EventLoggers` durchführen müssen, beschränken sich auf die Methode `log`. Listing 5.9 zeigt Ihnen die erforderlichen Anpassungen.

```
EventLogger.prototype.log = function (level, data) {
  if (this.levels.indexOf(level) > -1) {
    var now = new Date(),
        dateString = util.format('%s-%s-%s %s:%s',
                                now.getFullYear(),
                                (now.getMonth() + 1),
                                now.getDate(),
                                now.getHours(),
                                now.getMinutes(),
                                now.getSeconds()),
        message = util.format('%s (%s) %s\n', dateString, level, data);
    fs.appendFile(this.file, message);
  }
};
```

Listing 5.9 EventLogger mit `appendFile`

Wie Sie sehen können, halten sich die notwendigen Änderungen im Rahmen. Sie müssen lediglich die Generierung der Logdaten aus der Callback-Funktion von `open` auslagern und die Aufrufe von `open`, `write` und `close` durch einen Aufruf von `appendFile` mit dem Dateinamen und der Nachricht als Argumente ersetzen.

Wie auch schon beim Lesen von Informationen aus Dateien gilt auch für das Schreiben von Dateien, dass Fehler auftreten können. Es kann beispielsweise der Fall sein, dass der Benutzer, unter dem Ihre Applikation ausgeführt wird, keine Zugriffsberechtigungen für eine Datei besitzt, die geschrieben werden soll. Für diesen Fall steht Ihnen auch in den Callback-Funktionen von `write`, `writeFile` und `appendFile` als erstes Argument ein Objekt zur Verfügung, das die entsprechenden Fehlermeldungen enthält. Im Verlauf der Callback-Funktion können Sie dann mit dem Auftreten von Fehlern entsprechend umgehen. Greifen Sie beispielsweise mit `appendFile` auf eine Datei zu, für die Sie keine Schreibberechtigung besitzen, erhalten Sie folgende Meldung: `EACCES, open '/root/error.log'`. Das Fehler-Objekt enthält in diesem Fall ein Array mit Fehlern, eine Fehlernummer, einen Fehlercode und den Pfad, auf den zugegriffen werden sollte. Mit diesen Informationen können Sie entweder einen aussagekräftigen Eintrag in der Logdatei Ihrer Applikation generieren oder dem Benutzer Ihrer Anwendung eine passende Fehlermeldung präsentieren.

Nun, da Sie sowohl lesend als auch schreibend mit Dateien umgehen können, ist es an der Zeit, dass Sie sich in den folgenden Abschnitten mit einer weiteren Kategorie von Features des `fs`-Moduls beschäftigen: den Verzeichnisoperationen.

5.3 Verzeichnisoperationen

Die Node.js-Plattform bietet Ihnen nicht nur die Möglichkeit, Dateien auszulesen oder Daten in sie zu schreiben. Sie können auch ganze Verzeichnisse manipulieren. Dabei reicht die Spanne der verfügbaren Operationen vom Erstellen über das Umbenennen bis hin zum Löschen von Verzeichnissen. Aber auch die Einsatzgebiete für diese Art der Dateisystemoperationen sind vielfältig. Erstellen Sie beispielsweise eine Installationsroutine für Ihre Applikation in Node.js, können Sie die notwendige Verzeichnisstruktur generieren.

Verzeichnisse kommen in Ihrer Applikation immer dann vor, wenn Sie größere Mengen an Dateien strukturieren müssen. Ein konkretes Beispiel hierfür sind Cache-dateien, die Sie zum schnelleren Zugriff in eine Verzeichnisstruktur platzieren. Die Suchoperationen der Betriebssysteme sind meist schneller in einer tieferen Verzeichnisstruktur mit jeweils relativ wenigen Dateien pro Verzeichnis im Vergleich zu sehr vielen Dateien in einem einzigen Verzeichnis.

Im Verlauf dieses Abschnitts lernen Sie verschiedene Möglichkeiten zum Navigieren in Verzeichnisstrukturen und deren Manipulation kennen. Als Beispiel werden Sie einen sehr leichtgewichtigen Dateisystembrowser implementieren, mit dessen Hilfe es möglich ist, interaktiv über die Kommandozeile den Inhalt von Verzeichnissen aufzulisten, sich innerhalb der Verzeichnisstruktur zu bewegen und Dateien und Verzeichnisse zu erstellen, umzubenennen und zu löschen.

Den gesamten Quellcode dieses Beispiels finden Sie in Listing 5.10. Im Anschluss an dieses Beispiel erkläre ich Ihnen die einzelnen Schritte und die jeweiligen Besonderheiten bei den einzelnen verwendeten Methoden.

```
var fs = require('fs'),
    util = require('util');

var Explorer = function () {
  process.stdin.resume();
  process.stdin.setEncoding('utf8');
  process.stdin.on('data', this.dispatch.bind(this));
}

Explorer.prototype.stdin = {};
Explorer.prototype.commandRegex = /(\w*) (.*)/;

Explorer.prototype.dispatch = function (chunk) {
  var result = chunk.match(this.commandRegex),
      command, path;

  if (result === null) {
    command = "exit";
    path = '';
  } else {
    command = result[1];
    path = result[2];
  }

  switch (command) {
    case 'list':
      this.display(path);
      break;
    case 'change':
      process.chdir(path);
      this.listCurrent();
      break;
    case 'file':
      fs.writeFile(path, this.listCurrent.bind(this));
      break;
    case 'remove':
      fs.stat(path, function(err, stat) {
        if (stat.isFile()) {
          fs.unlink(path, this.listCurrent.bind(this));
        }
      });
    }
  }
}
```

```

        } else if (stat.isDirectory()) {
            fs.rmdir(path, this.listCurrent.bind(this));
        }
    }.bind(this));
    break;
case 'directory':
    fs.mkdir(path, this.listCurrent.bind(this));
    break;
case 'rename':
    var paths = path.split(' ');
    fs.rename(paths[0], paths[1], this.listCurrent.bind(this));
    break;
default:
    process.exit();
    break;
}
}

Explorer.prototype.listCurrent = function () {
    this.dispatch('list ' + process.cwd());
}

Explorer.prototype.display = function (path) {
    util.print('Aktuelles Verzeichnis: ' + process.cwd() + '\n\n');

    fs.readdir(path, function (err, files) {
        var result = '';

        for (var i = 0; i < files.length; i++) {
            result += files[i] + '\t';
            if ((i+1) % 4 === 0) {
                result += '\n';
            }
        }

        var diff = process.stdout.rows - files.length;
        if (diff > 0) {
            for (var i = 0; i < diff; i++) {
                result += '\n';
            }
        }
    });
}

```

```

        util.print(result + '\n\nVerfügbare Kommandos: ' +
            'list, change, file, remove, directory, rename, exit\n' +
            'Ihre Eingabe: ');
    });
}

var explorer = new Explorer();
explorer.listCurrent();

```

Listing 5.10 Dateisystembrowser in Node.js

Die Basis der Applikation bildet die Konstruktor-Funktion. Sie sorgt dafür, dass die Standardeingabe dafür verwendet werden kann, die Befehle, die für die Steuerung der Applikation erforderlich sind, über die Kommandozeile anzunehmen. Die Standardeingabe ist im Normalfall pausiert und muss mithilfe von `resume` zur Verwendung gestartet werden. Danach können Sie beliebige Funktionen auf das `data`-Event binden. Dieses Event wird immer geworfen, wenn ein Benutzer eine Eingabe macht und diese mit  absendet. An dieses Event binden Sie die `dispatch`-Methode der `Explorer`-Klasse.

Zur Navigation im Dateisystem greifen Sie auf die Funktionalität des `process`-Moduls zurück. Mit der `chdir`-Methode können Sie in der Verzeichnishierarchie des Dateisystems navigieren. Die `cwd`-Methode gibt Ihnen das Verzeichnis, in dem Sie sich aktuell befinden, zurück. Das aktuelle Arbeitsverzeichnis dient als Ausgangspunkt für relative Pfadangaben. Auf diesem Prinzip baut die Applikation auf.

Damit Sie sich im Dateisystem bewegen können, ist es erforderlich, dass Sie im ersten Schritt die Eingabe des Nutzers anhand eines regulären Ausdrucks in die Bestandteile `command`, den Befehl, der ausgeführt werden soll, und `path`, der Pfad, auf den der Befehl angewendet werden soll, zu zerlegen. Aufgrund des angegebenen Kommandos erfolgt eine Verzweigung in verschiedene Äste der Programmlogik. Der mitunter wichtigste Teil der Programmlogik dient der Anzeige des aktuellen Verzeichnispfades, dessen Inhalt und der verfügbaren Optionen und ist in eine separate Methode, `display`, ausgelagert. Hier greifen Sie auch auf Bestandteile des `process`-Moduls in Form der `cwd`-Methode zu, um das aktuell gewählte Arbeitsverzeichnis auszugeben. Des Weiteren nutzen Sie die `readdir`-Methode des `fs`-Moduls, um die Inhalte des aktuellen Verzeichnisses auszugeben. Diese Methode akzeptiert zwei Argumente. Das erste ist der Name des Verzeichnisses, dessen Inhalte ausgelesen werden sollen, und das zweite Argument ist eine Callback-Funktion, die ihrerseits ein Fehler-Objekt und die Inhalte des Verzeichnisses als Argumente erhält. Die Dateien und Unterverzeichnisse des gewünschten Verzeichnisses liegen in der Callback-Funktion als ein Array von Zeichenketten vor. Bitte beachten Sie bei der Verwendung dieser Methode, dass diese nur auf eine Ebene der Verzeichnishierarchie wirkt und nicht rekursiv arbeitet.

Die Navigation innerhalb der Verzeichnisstruktur, also konkret das Wechseln des aktuellen Arbeitsverzeichnisses, übernimmt die `chdir`-Methode des `process`-Moduls für Sie. Dieses Kommando arbeitet synchron und erfordert aus diesem Grund keine Callback-Funktion. Nach erfolgtem Wechsel des Arbeitsverzeichnisses zeigen Sie durch den Aufruf von `listCurrent`, was intern wiederum die Ausführung der `display`-Methode nach sich zieht, den Inhalt des nun aktuellen Arbeitsverzeichnisses an.

Über die Kommandos `file` und `directory` können Sie innerhalb des Dateisystembrowsers neue Dateien und Verzeichnisse anlegen. Das `fs`-Modul stellt auch für diesen Fall die entsprechenden Methoden zur Verfügung. Eine Datei können Sie im einfachsten Fall über einen Aufruf der `writeFile`-Methode mit dem Dateinamen und einer Callback-Funktion als Argumente erstellen. Bei der Erstellung eines Verzeichnisses können Sie auf eine ähnliche Weise vorgehen. Hierbei kommt die Methode `mkdir` zum Einsatz. Sie akzeptiert ebenfalls zwei Argumente, den Namen des Verzeichnisses, das erstellt werden soll, und eine Callback-Funktion. Die Methode `mkdir` kann lediglich eine Verzeichnisebene erstellen und ist nicht in der Lage, mit einem Aufruf eine Struktur aus Verzeichnis und Unterverzeichnis zu erstellen. Die Callback-Funktion erhält als einziges Argument ein Objekt mit eventuellen Fehlern, die bei der Erstellung des Verzeichnisses aufgetreten sind. Im Beispiel in Listing 5.10 können Sie in beiden Fällen, also sowohl für Dateien als auch für Verzeichnisse, mit der `listCurrent`-Methode die gleiche Callback-Funktion verwenden. Sie müssen lediglich mit einem Aufruf der `bind`-Methode dafür sorgen, dass die Methode im korrekten Kontext ausgeführt wird.

Die Möglichkeit, Dateien und Verzeichnisse umzubenennen, ist ein weiteres Feature des Dateisystembrowsers auf Basis der Node.js-Plattform. Sie können zur Umsetzung dieser Funktionalität auf die `rename`-Methode des `fs`-Moduls zurückgreifen. Mit ihrer Hilfe können Sie sowohl Dateien als auch Verzeichnisse umbenennen. Die Argumente, die Sie zu diesem Zweck verwenden müssen, sind der Name der Datei beziehungsweise des Verzeichnisses, das Sie umbenennen möchten, das zweite Argument besteht aus dem neuen Namen, und das dritte Argument ist schließlich die Callback-Funktion, die ausgeführt werden soll, sobald die Umbenennung vollzogen ist. Im Falle des Dateisystembrowsers verwenden Sie die Methode `listCurrent` als Callback-Funktion. Bisher reichte die Kombination aus Kommando und Pfad aus, um die Kommandos anzusprechen. Im Falle der Umbenennung müssen Sie allerdings zwei Werte übergeben. Diesen Sonderfall können Sie direkt vor dem Aufruf von `rename` behandeln, indem Sie den Pfad, der übergeben wurde, anhand von Leerzeichen in mehrere Teile auftrennen. So wird es möglich, mit dem Kommando `rename oldPath newPath` die Datei oder das Verzeichnis mit dem Namen `oldPath` in `newPath` umzubenennen.

Der letzte Schritt in der Umsetzung der Applikation besteht schließlich darin, dass Sie dem Benutzer eine Möglichkeit bieten, die Anwendung auch wieder zu beenden. Dies erreichen Sie beispielsweise, indem Sie im `switch`-Statement einen `default`-Zweig

angeben, der dafür sorgt, dass der aktuelle Prozess mit einem Aufruf von `process.exit` beendet wird.

Die gesamte Applikation können Sie nun starten, indem Sie die Klasse `Explorer` instanziiieren und auf dem entstandenen Objekt die Methode `listCurrent` aufrufen. Sie sorgt dafür, dass die `dispatch`-Methode mit dem Kommando `list` und dem aktuellen Arbeitsverzeichnis als Argumente aufgerufen wird.

In den letzten Abschnitten dieses Kapitels erfahren Sie mehr über weitere Features des `fs`-Moduls, mit denen Sie beispielsweise Metainformationen von Dateien auslesen, Dateien auf Veränderungen beobachten oder den Eigentümer und Berechtigungen von Dateien manipulieren können.

5.4 Weiterführende Operationen

Bis jetzt haben Sie sich vor allem mit den Grundlagen der Dateisystemoperationen beschäftigt. Konkret haben Sie gesehen, wie Sie mit Dateien arbeiten oder verschiedene Operationen auf Verzeichnisse anwenden können. Hier erfahren Sie nun, welche weiteren Operationen Ihnen das `fs`-Modul zur Verfügung stellt.

Die erste Operation, die ich Ihnen hier vorstellen möchte, haben Sie im letzten Beispiel bereits ansatzweise kennenlernen können. Es handelt sich um die `stat`-Methoden des `fs`-Moduls, die Sie verwenden können, um Informationen über Dateien und Verzeichnisse zu gewinnen. Im Dateisystembrowser haben Sie die `stat`-Methode dazu verwendet, um zu unterscheiden, ob es sich bei einem gegebenen Pfad um eine Datei oder ein Verzeichnis handelt und entsprechend dem Ergebnis eine unterschiedliche Routine zur Behandlung aufgerufen. Es gibt drei verschiedene Varianten der `stat`-Methode. Die `stat`-Methode selbst wird dazu verwendet, verschiedene Informationen einer Datei oder eines Verzeichnisses auszulesen. Wie alle Methoden, die Sie bisher gesehen haben, ist auch `stat` asynchron. Das bedeutet, dass sie neben dem Pfad, der inspiziert werden soll, als zweites Argument eine Callback-Funktion akzeptiert. Diese erhält, wenn die Operation abgeschlossen ist, ein Fehler-Objekt, das eventuelle Exceptions enthält, die während der Abarbeitung des Befehls aufgetreten sind. Das zweite Argument ist ein Objekt vom Typ `Stats`. Dieses enthält die gewünschten Informationen und einige hilfreiche Methoden. Wird die `stat`-Methode auf einen symbolischen Link aufgerufen, erhalten Sie die Informationen über die Datei, auf die der Link verweist. Tabelle 5.2 zeigt Ihnen die Informationen, die Sie aus dem `Stats`-Objekt erhalten.

Neben den vorgestellten Eigenschaften verfügt das `Stats`-Objekt über eine Reihe von Methoden, mit denen der Typ des Objekts bestimmt werden kann, auf das die `stat`-Methode angewandt wurde. So können Sie beispielsweise mit `stats.isFile` oder `stats.isDirectory` feststellen, ob es sich bei dem angegebenen Pfad um eine Datei

oder ein Verzeichnis handelt. Des Weiteren können Sie mithilfe von `stats.isBlockDevice` und `stats.isCharacterDevice` prüfen, ob es sich bei dem Pfad um eine blockorientierte oder eine zeichenorientierte Gerätedatei handelt. Außerdem können Sie die Methoden `stats.isSymbolicLink`, `stats.isFIFO` und `stats.isSocket` verwenden.

Eigenschaft	Beschreibung
<code>dev</code>	Nummer des Geräts, auf dem die Datei liegt
<code>mode</code>	Zugriffsberechtigungen für die Datei
<code>nlink</code>	Anzahl der Links auf die Datei
<code>uid</code>	ID des Eigentümers der Datei
<code>gid</code>	ID der Gruppe, der die Datei gehört
<code>rdev</code>	Typ des Dateisystems, auf dem die Datei liegt
<code>blksize</code>	Größe der Blöcke im Dateisystem
<code>ino</code>	Nummer des Inodes der Datei
<code>size</code>	Größe der Datei in Byte
<code>blocks</code>	Anzahl der durch die Datei belegten Blöcke
<code>atime</code>	Zeitpunkt des letzten Zugriffs
<code>mtime</code>	Zeitpunkt der letzten Änderung
<code>ctime</code>	Zeitpunkt der Erstellung

Tabelle 5.2 Informationen, die die `stat`-Methode liefert

Wie auch bei allen übrigen Dateisystemoperationen existiert auch für die `stat`-Methode neben der asynchronen Version mit `statSync` eine synchrone Variante. Der einzige Unterschied zwischen beiden ist, dass die synchrone Version die Ausführung der Anwendung blockiert, bis ein Ergebnis vorliegt, und dieses dann von der Methode als Rückgabewert an den Aufrufer zurückgeliefert wird. Daher ist keine Callback-Funktion erforderlich.

Zusätzlich zur `stat`-Methode existieren mit `lstat` und `fstat` noch zwei weitere Varianten dieser Methode. `lstat` unterscheidet sich dahingehend, dass, wenn diese Methode auf einen symbolischen Link angewandt wird, sie die Informationen über den Link selbst und nicht über die Zieldatei zurückliefert. Die Methode `fstat` akzeptiert im Gegensatz zu den anderen beiden Methoden einen Filedescriptor, wie er durch einen Aufruf von `fs.open` erzeugt wird. Sowohl für `lstat` als auch für `fstat`

existiert ein synchrones Pendant. Listing 5.11 zeigt Ihnen den Einsatz der `stat`-Methode.

```
var fs = require('fs');
var util = require('util');

fs.stat('/usr/local/bin/node', function (err, stat) {
  util.puts('Size of the file "/usr/local/bin/node" is ' + stat.size +
    ' Bytes');
});
```

Listing 5.11 Verwendung der `stat`-Methode

5.4.1 watch

Das nächste Feature, das ich Ihnen hier vorstellen möchte, ist die Methode `watch`. Obwohl das gesamte `fs`-Modul einen Stabilitätsindex von 3, also `Stable`, aufweist, ist der Wert bei dieser Methode 2, also `Unstable`. Sie sollten demzufolge bei der Verwendung dieser Methode in Ihrem Live-System vorsichtig sein, vor allem, wenn es um Versionsupdates der Node.js-Plattform geht, da sich hier eventuell die Signatur oder das Verhalten der Methode ändern könnte. Die `watch`-Methode existiert nur als asynchrone Methode und ist damit die einzige ohne synchrones Gegenstück im gesamten Modul. Das erste Argument, das diese Methode akzeptiert, ist der Name der Datei, die beobachtet werden soll. Das zweite optionale Argument dient der Konfiguration der Methode und ist ein Objekt mit der Eigenschaft `persistent`, die standardmäßig den Wert `true` enthält, was bedeutet, dass der Prozess so lange ausgeführt wird, wie der Watcher auf die Datei aktiv ist. Das letzte Argument dieser Methode ist eine Callback-Funktion, die aufgerufen wird, wenn der Inhalt oder der Name der Datei verändert wird. Der Watcher kann auf zwei verschiedene Arten beendet werden. Die erste Variante besteht darin, die Methode `unwatchFile` mit dem Dateinamen als Argument zu verwenden. Die zweite und von den Node.js-Entwicklern empfohlene Variante ist, den Rückgabewert der `watch`-Methode, ein Objekt vom Typ `FSWatcher`, zu verwenden. Auf diesem Objekt können Sie die `close`-Methode aufrufen, um den Watcher zu beenden.

Die Callback-Funktion, die der `watch`-Methode als letztes Argument übergeben wird, erhält zwei Argumente. Das erste Argument ist die Art des Ereignisses, das aufgetreten ist. Dies kann entweder `change` oder `rename` sein. Das zweite Argument ist der Name der Datei, die unter Beobachtung steht. Hierbei sollten Sie beachten, dass das zweite Argument nicht auf allen Betriebssystemen zur Verfügung steht. Dieser Umstand rührt vor allem daher, dass sich die `watch`-Funktionalität nicht über alle Systeme hinweg konsistent verhält und auch auf den verschiedenen Systemen auf unterschiedlichen Betriebssystemfunktionen aufsetzt. Bei Linux-Systemen ist dies beispielsweise `inotify`, andere UNIX-Systeme, wie auch Mac OS X, verwenden `kqueue`,

und unter Windows schließlich wird `ReadDirectoryChangesW` verwendet. In Listing 5.12 sehen Sie in einem Beispiel, wie Sie die `watch`-Methode einsetzen können. Beachten Sie dabei bitte, dass der Prozess erst durch den Aufruf von `watcher.close` beendet wird. Lassen Sie diese Zeile weg, müssen Sie die Ausführung mit der Tastenkombination `[Strg] + [C]` beenden.

```
var fs = require('fs'),
    file = 'error.log';

var watcher = fs.watch('error.log', function (event) {
  console.log(file + ' has been ' + event + 'd');
  watcher.close();
});
```

Listing 5.12 Einsatz der `watch`-Methode

Neben der Methode `watch` existiert eine weitere Methode mit dem Namen `watchFile`, die dieselbe Funktionalität zur Verfügung stellt, allerdings zugunsten von `watch` nicht weiter verwendet werden sollte. Das Gleiche gilt, wie bereits erwähnt, für die Methode `unwatchFile`.

5.4.2 Zugriffsberechtigungen

Der letzte Teil dieses Kapitels soll Ihnen die Möglichkeiten der Manipulation von Zugriffsberechtigungen von Dateien unter Node.js näherbringen. Mit den Dateisystemberechtigungen kommen Sie spätestens in Berührung, wenn Sie versuchen, eine Datei oder ein Verzeichnis zu erstellen oder zu modifizieren, für das Sie keine Berechtigung besitzen. Bevor Sie jedoch dazu übergehen, die Berechtigungen einer Datei oder eines Verzeichnisses zu verändern, sollen Sie hier zuerst eine Möglichkeit kennenlernen, mit der Sie die aktuellen Berechtigungen auszulesen imstande sind. Die Informationen, die im Zusammenhang mit Berechtigungen relevant sind, sind die Benutzer-ID, die ID der Gruppe, der die Datei zugeordnet ist, und schließlich die Berechtigungen. Diese Informationen erhalten Sie über die `stat`-Methode, deren Verwendung Sie in Listing 5.13 sehen können.

```
var fs = require('fs');

fs.stat('myFile.txt', function (err, stat) {
  console.log('User: ' + stat.uid);
  console.log('Gruppe: ' + stat.gid);
  console.log('Permissions: ' + stat.mode.toString(8));
});
```

Listing 5.13 Relevante Informationen über Berechtigungen abfragen

Die einzelnen Eigenschaften des `stat`-Objekts haben Sie bereits kennengelernt. Die einzige Besonderheit besteht im Auslesen der Berechtigungen in Form der `mode`-Eigenschaft. Hier müssen Sie über die `toString`-Methode mit dem Argument 8 zugreifen, um die oktale Repräsentation dieses Wertes zu erhalten.

Die Methode, mit der Sie die Berechtigungen, die Sie mit der `mode`-Eigenschaft auslesen, setzen können, heißt `chmod` und existiert sowohl als synchrone als auch als asynchrone Version. Außerdem gibt es, wie bei der `stat`-Methode, eine `lchmod`-Version, die auf Links und nicht auf deren Ziel wirkt, und eine `fchmod`-Methode, die statt eines Dateinamens einen Filedescriptor akzeptiert. Alle drei Methoden akzeptieren neben der Datei, deren Berechtigungen geändert werden sollen, die Berechtigungen, die gesetzt werden sollen. Im Idealfall geben Sie die Berechtigungen in Oktalnotation an, also mit einer vorangestellten Null. So können Sie die für UNIX übliche Schreibweise verwenden. Bei den asynchronen Methoden übergeben Sie als drittes Argument eine Callback-Funktion, die ausgeführt wird, sobald die Berechtigungen angepasst sind.

Die letzte Methode, die ich Ihnen aus dem `fs`-Modul vorstellen möchte, ist die `chown`-Methode. Sie dient dazu, den Eigentümer und die Gruppe einer Datei anzupassen. Auch für diese Methode existieren mit `lchown` und `fchown` verschiedene Varianten, für die es jeweils auch eine synchrone und eine asynchrone Version gibt. Die Argumente für diese Methoden sind der Name der Datei oder des Verzeichnisses, danach die ID des Benutzers und die ID der Gruppe. Bei den asynchronen Methoden kommt noch eine Callback-Funktion hinzu, die ausgeführt wird, wenn Benutzer und Gruppe gesetzt wurden. Das Beispiel in Listing 5.14 zeigt die Methoden im Einsatz.

```
var fs = require('fs');

fs.stat('error.log', function (err, stat) {
  showPerms(err, stat);
  fs.chmod('error.log', 0777, function (err) {
    fs.chown('error.log', 0, 0, function (err) {
      fs.stat('error.log', showPerms);
    });
  });
});

showPerms = function (err, stat) {
  console.log('User: ' + stat.uid);
  console.log('Gruppe: ' + stat.gid);
  console.log('Permissions: ' + stat.mode.toString(8));
};
```

Listing 5.14 Anpassen von Berechtigungen

In Listing 5.14 sehen Sie, wie Sie die `stat`-Methode und eine Hilfsfunktion `showPerms` verbinden und im Anschluss `chmod` und `chown` zum Setzen neuer Berechtigungen und neuer Eigentümer ausführen können. Zum Schluss wird das Ergebnis der Änderungen unter Zuhilfenahme der Hilfsfunktion angezeigt.

5.5 Zusammenfassung

In diesem Kapitel haben Sie verschiedene Methoden des `fs`-Moduls kennengelernt, die es Ihnen ermöglichen, mit Dateien und Verzeichnissen umzugehen. Mit `Node.js` können Sie Dateien auslesen und Daten in Dateien schreiben oder Verzeichnisse erstellen, umbenennen und wieder löschen. Im nächsten Kapitel stelle ich Ihnen verschiedene Methoden und Konzepte vor, die Sie zur Kommunikation einsetzen können.

Kapitel 6

Kommunikation

Seit man begonnen hat, die einfachsten Behauptungen zu beweisen, erwiesen sich viele von ihnen als falsch.

– Bertrand Russell

Software im Webumfeld lebt von der Interaktion mit ihren Nutzern. Da die Node.js-Plattform aus genau diesem Bereich der Softwareentwicklung heraus entstanden ist, gilt diese Aussage ebenfalls für serverseitiges JavaScript. Interaktion ist jedoch nur möglich, wenn die Software entsprechende Schnittstellen für die Benutzer oder auch andere Systeme bietet. In diesem Kapitel stelle ich Ihnen einige Konzepte der Node.js-Plattform vor, die es Ihnen erlauben, Software zu erstellen, die mit ihrer Umwelt kommunizieren kann.

Kommunikation wird in Applikationen immer nötig, wenn Eingaben für Berechnungen gebraucht werden. Als Berechnungen sind in diesem Fall nicht nur mathematische Berechnungen gemeint, sondern jegliche Art von Logik innerhalb Ihrer Applikation. Unter Node.js haben Sie verschiedenste Möglichkeiten, um die Ausführung von Applikationslogik von außen zu beeinflussen. Die erste Variante, die Sie bereits kennengelernt haben, sind das Ausführen von JavaScript-Logik direkt über die Kommandozeile und die Steuerung dieser Logik über Kommandozeilenoptionen. Eine weitere Möglichkeit, mit einer bereits laufenden Applikation zu interagieren, ist die Steuerung über die Tastatur per Standardeingabe über das `process.stdin`-Objekt. Diese Mittel beschränken sich auf eine lokal laufende Installation Ihrer Node.js-Applikation. In den folgenden Abschnitten sehen Sie, welche Möglichkeiten Ihnen Node.js bietet, wenn es um die Kommunikation über das Netzwerk geht.

Zur Kommunikation mit Benutzern über das Netzwerk, aber auch zwischen Applikationen kommen Datenströme zum Einsatz. Ein Datenstrom ist in diesem Fall nichts anderes als ein mehr oder weniger kontinuierlicher Fluss von Paketen, die über ein standardisiertes Kommunikationsmedium, wie beispielsweise eine TCP- oder UDP-Verbindung oder einen UNIX Domain Socket, versendet werden. Ein weiteres Merkmal der Netzwerkkommunikation ist, dass der Austausch von Informationen in beide Richtungen funktioniert. Sie können von einer derartigen Verbindung also sowohl Daten lesen als auch Daten über diese Verbindung versenden.

Die Entwickler von Node.js setzen hier den eventgetriebenen Ansatz konsequent fort. Versenden Sie Daten über eine bestehende Verbindung, verursacht dies auf der Gegenseite ein Ereignis, auf das entsprechend reagiert werden kann. Dieser Sachverhalt bedeutet wiederum, dass die Kommunikation asynchron stattfindet und Ihre Applikation aus diesem Grund nicht blockiert wird.

Im ersten Teil dieses Kapitels möchte ich Ihnen das `http`-Modul der Node.js-Plattform näher vorstellen und Ihnen eine Möglichkeit zeigen, wie Sie auf sehr einfache Art Webservices mit Node.js umsetzen können.

6.1 Der Webserver

Das klassische Beispiel der Kommunikation mit einem Nutzer einer Node.js-Applikation erfolgt über das `http`-Modul und damit über den HTTP-Server von Node.js. In Kapitel 3, »Ein erstes Beispiel«, haben Sie bereits einen Einblick in die Möglichkeiten des `http`-Moduls von Node.js erhalten. Die folgenden Abschnitte beschäftigen sich nun mit den Features und Konzepten dieses Moduls.

Die beiden Gegenseiten bei einer Kommunikation mit HTTP werden als Client und Server bezeichnet. Als Client können Sie in den Beispielen entweder einen Webbrowser oder ein Kommandozeilenprogramm wie `cURL` verwenden, das es Ihnen ermöglicht, HTTP-Nachrichten zu senden und zu empfangen.

Damit Sie die Serverseite mit Node.js implementieren können, benötigen Sie eine Instanz des Node.js-HTTP-Servers.

6.1.1 Das Server-Objekt

Da der Umgang mit dem Server eine sehr häufig auftretende Aufgabe unter Node.js ist, gibt Ihnen die Plattform einige Abkürzungen beim Aufbau des Servers zur Hand. Im Beispiel in Listing 6.1 sehen Sie, wie Sie in nur wenigen Schritten einen funktionsfähigen Server aufbauen.

```
var http = require('http');

http.createServer(
  function(request, response) {
    response.writeHead(200, {'content-type': 'text/plain'});
    response.end('Hello World');
  }
).listen(8080, 'localhost');
```

Listing 6.1 Ein Webserver in Node.js

Was in diesem Beispiel genau passiert, zeigt Ihnen Listing 6.2.

```
var Server = require('http').Server;

var server = new Server();

server.addListener('request', function (request, response) {
  response.statusCode = 200;
  response.setHeader('content-type', 'text/plain');
  response.write('Hello World');
  response.end();
});

server.listen(8080, 'localhost');
```

Listing 6.2 Der Webserver im Detail

Der Quellcode aus den Beispielen in Listing 6.1 und Listing 6.2 verfügt über die gleiche Funktionalität, mit dem Unterschied, dass bei Listing 6.2 darauf geachtet wurde, dass die einzelnen Schritte, die im Hintergrund passieren, für Sie deutlicher werden.

Die Methode `createServer` ist nichts anderes als ein Wrapper zur Erstellung einer neuen Instanz der Klasse `Server`. Das gleiche Resultat können Sie erreichen, indem Sie die Klasse, die Ihnen das `http`-Modul unter der Eigenschaft `Server` zur Verfügung stellt, mit dem `new`-Operator selbst instanziiieren. Wie Sie es auch erstellen, das `Server`-Objekt dient als Grundlage für Ihren HTTP-Server. Mit diesem Objekt können Sie nun auf Anfragen reagieren.

Die Instanz des Servers allein hilft Ihnen allerdings auch nur begrenzt weiter, wenn Sie mit Ihren Benutzern kommunizieren möchten. Noch ist der Server nicht gebunden. Das bedeutet, dass er noch keine Anfragen akzeptiert. Zu diesem Zweck müssen Sie unter Verwendung der `listen`-Methode definieren, worauf der Server gebunden werden soll. Die einfachste Art, wie Sie sie auch hier im Beispiel gesehen haben, besteht darin, den Server auf die Kombination aus TCP-Port und Hostnamen zu binden. Bei der Angabe des Hostnamens sind Sie allerdings nicht auf DNS-Namen, wie beispielsweise `www.mydomain.com`, beschränkt. Sie können an dieser Stelle auch IP-Adressen verwenden. Wie auch immer Sie die Information zur Verfügung stellen möchten, achten Sie stets darauf, dass Sie sowohl den Hostnamen als auch die IP-Adresse als eine Zeichenkette übergeben. Lassen Sie den Hostnamen komplett weg, wird der Server auf sämtliche auf dem System verfügbaren IPv4-Adressen gebunden.

Der Aufruf der `listen`-Methode akzeptiert ein drittes Argument, über das Sie die maximale Anzahl an Verbindungen angeben können, die als wartend in der Warteschlange gehalten werden sollen. Dieser Wert wird als Ganzzahl übergeben, ist allerdings optional. Das bedeutet, dass, wenn Sie diesen Wert nicht definieren, der

Standard von 511 angenommen wird. Durch interne Berechnungen des Kernels führt dies zu 512 Verbindungen in der Warteschlange.

Das letzte Argument der `listen`-Methode ist schließlich ein Callback, der ausgeführt wird, sobald der Server erfolgreich an die Adress-Port-Kombination gebunden ist.

Beim Binden des Servers sind Sie nicht auf IP-Adressen und Portnummern beschränkt. Sie können den HTTP-Server außerdem an einen UNIX Domain Socket binden. Zu diesem Zweck übergeben Sie entweder den Pfad zur Socket-Datei oder einen Datei-Handle, wie Sie ihn als Rückgabewert von der `open`-Methode des `fs`-Moduls erhalten. Sie können den Server außerdem an einen anderen Server binden.

Haben Sie den Server durch den Aufruf der `listen`-Methode gebunden, wartet dieser auf Anfragen. Sie können diesen Vorgang beenden, indem Sie die `close`-Methode auf dem Server-Objekt aufrufen. Diese sorgt dafür, dass der Server keine neuen Anfragen annimmt.

Versuchen Sie einen Server auf einen Port zu binden, der bereits verwendet wird, führt dies dazu, dass ein Fehler auftritt. Der ausgegebene Code der Fehlermeldung lautet `EADDRINUSE`. Fangen Sie diesen Fehler nicht korrekt ab, führt dieser Fehler dazu, dass die Ausführung Ihrer Applikation abgebrochen wird.

Mit der Eigenschaft `maxHeadersCount` verfügt der HTTP-Server über eine Eigenschaft, über die sich die Anzahl der Schlüssel-Wert-Paare im Header einer Nachricht limitieren lässt. Setzen Sie diesen Wert explizit, werden die Header-Informationen, die diese zulässige Zahl überschreiten, abgeschnitten und stehen Ihnen auf Serverseite nicht zur Verfügung. Der Standard für diesen Wert liegt bei 1000. Ein Wert von 0 bedeutet, dass es keine Beschränkung gibt und der Client beliebig viele Header-Informationen übermitteln kann.

6.1.2 Server-Events

Die eventgetriebene Architektur von Node.js zieht sich auch durch das `http`-Modul. Innerhalb der Plattform können Sie auf eine Reihe von Ereignissen zurückgreifen, mit deren Hilfe Sie asynchron auf verschiedene Vorkommnisse reagieren können.

In Listing 6.1 sehen Sie indirekt, wie Sie eines der Events, das Ihnen das Server-Objekt bietet, nutzen können. Die Methode `createServer` akzeptiert eine Callback-Funktion als Argument. Sie enthält eine Funktion, die aufgerufen wird, sobald eine Anfrage eines Clients eingeht. In Listing 6.2 sehen Sie, was im Hintergrund eigentlich passiert. Die Callback-Funktion wird über die `addListener`-Methode an das `request`-Ereignis gebunden. Dieses Ereignis tritt immer dann auf, wenn ein Client eine URL beim Server nachfragt. Die Callback-Funktion erhält als Argumente zwei Objekte. Das erste ist der Request, also die vom Client kommende Anfrage, und das zweite ist die Response, die Antwort, die vom Server zum Client geschickt wird. Im Laufe dieses Kapitels erfahren Sie noch mehr über diese beiden Objekte und den Umgang mit ihnen.

Tabelle 6.1 zeigt Ihnen die Events, die im Umfang des Server-Objekts enthalten sind. Auf diese Ereignisse werde ich im Folgenden näher eingehen, und Sie werden erfahren, wie Sie mit den einzelnen Ereignissen Ihre Applikation steuern können.

Ereignis	Bedeutung
request	eingehende Client-Anfrage
connection	Verbindung mit einem Client wurde aufgebaut.
close	Server wird beendet.
checkContinue	Tritt auf, wenn erwartet wird, dass der Server mit Statuscode 100 antworten soll.
connect	Ein Client fragt mit der HTTP-Methode CONNECT beim Server an.
upgrade	Ein Client beantragt ein HTTP-Upgrade.
clientError	Ein Client hat eine Fehlermeldung gesendet.

Tabelle 6.1 Server-Events

Immer wenn sich ein Client mit dem Server verbindet und dadurch eine TCP-Verbindung aufbaut, wird das `connection`-Event geworfen. Registrieren Sie einen Listener auf dieses Ereignis, erhält dieser als einziges Argument die Verbindung, die aufgebaut wurde. Dieses Objekt ist vom Typ `net.Socket`, und Sie können jederzeit auch außerhalb dieser Callback-Funktion über die `connection`-Eigenschaft des `Request`-Objekts darauf zugreifen.

Das nächste Ereignis `close` wird immer dann gefeuert, wenn Sie die `close`-Methode des `Server`-Objekts aufrufen. Registrieren Sie einen Callback auf dieses Ereignis, wird dieser immer ausgeführt, wenn es auftritt. Sie haben allerdings auch die Möglichkeit, eine Callback-Funktion direkt an die `close`-Methode zu übergeben. Dieser Callback wird durch die Methode `once` nur einmalig aufgerufen. Diesen Sachverhalt sollten Sie beachten, wenn Sie durch die Verwendung von Callback-Funktionen auf das Beenden des Servers reagieren möchten.

Das Ereignis `checkContinue` wird ausgelöst, wenn ein Client beim Server anfragt und im Header die Information `Expect: 100-continue` enthalten ist. Im Normalfall müssen Sie auf dieses Ereignis nicht weiter reagieren, da Node.js von sich aus mit einer Nachricht mit dem Statuscode 100 `Continue` antwortet. Dies geschieht allerdings nur, wenn kein Listener auf dieses Ereignis registriert ist. Die Callback-Funktion für dieses Ereignis erhält als Argumente die Anfrage und die Antwort, wie Sie es vom `request`-Event bereits kennen. Wenn Sie korrekt mit dem Ereignis umgehen möchten, sollten Sie dem Client mitteilen, ob er die eigentliche Nachricht übermitteln soll. Sie erreichen dies, indem Sie die Methode `writeContinue` des `Response`-Objekts innerhalb der

Callback-Funktion aufrufen. Soll der Client den Körper der Nachricht nicht senden, können Sie mit einer Antwort mit einem Statuscode aus dem Bereich 400 bis 499 reagieren. Sobald Sie auf `checkContinue` reagieren, wird für diese Anfrage kein weiteres request-Ereignis geworfen.

Sie haben die Möglichkeit, nicht nur mit den HTTP-Methoden GET und POST Anfragen an den Server zu schicken, sondern auch mit den übrigen HTTP-Methoden, beispielsweise mit CONNECT. Die Methode CONNECT wird normalerweise verwendet, wenn eine Verbindung über einen Proxyserver getunnelt wird und dabei das HTTP-Protokoll als Transportmedium dienen soll. Damit Sie diese Methode in Ihrer Applikation unterstützen können, stellt Ihnen die Node.js-Plattform ein eigenes Event zur Verfügung, das gefeuert wird, sobald ein Client mit der HTTP-Methode CONNECT anfragt. Die Callback-Funktion, die Sie mit diesem Event verbinden können, akzeptiert drei Argumente. Das erste ist das `Request`-Objekt, das die Anfrage repräsentiert, die zur Entstehung des Ereignisses geführt hat. Das zweite Argument besteht aus einer Referenz auf die Verbindung zwischen Client und Server. Das letzte schließlich ist ein Buffer-Objekt, das das erste Paket des Datenstroms enthält. Innerhalb der Callback-Funktion müssen Sie sich darum kümmern, dass mit den Daten, die über die Verbindung gesendet werden, korrekt umgegangen wird. Das bedeutet konkret, dass Sie eine Callback-Funktion definieren und an das `data`-Event des `Socket`-Objekts der Verbindung binden sollten. Reagieren Sie auf dieses Ereignis nicht, also registrieren Sie keine Callback-Funktion dafür, werden Verbindungen, die mit der Methode CONNECT erzeugt werden, von Node.js geschlossen.

Als Benutzer können Sie eine HTTP-Nachricht an einen Server senden und den `upgrade`-Header dabei verwenden. Upgrade bedeutet in diesem Zusammenhang, dass eine normale HTTP-Verbindung in eine andere Art von Verbindung umgewandelt werden soll. Konkrete Beispiele für die Verwendung von `upgrade`-Headern sind die Verwendung von TLS und Websockets. Im Falle von TLS bauen Sie eine normale Verbindung zum Server auf und teilen diesem im Anschluss mit, dass Sie TLS, also Transport Layer Security, für diese Verbindung und damit für die Verschlüsselung der Daten, verwenden möchten. Ähnliches gilt für Websockets. Hier erwarten Sie vom Server, dass dieser vom HTTP auf das Websockets-Protokoll für die Verbindung wechselt. Die Node.js-Plattform bietet Ihnen zur Unterstützung des `upgrade`-Headers ein separates Event mit dem Namen `upgrade`, auf das Sie eine entsprechende Callback-Funktion binden können, die dann für die korrekte Behandlung der Verbindung sorgt.

TLS

Früher bekannt unter der Bezeichnung Secure Socket Layer oder kurz SSL, ist TLS die Weiterentwicklung dieses Verschlüsselungsprotokolls in der Version 3.0. TLS kommt vor allem bei der Verschlüsselung von HTTP-Verbindungen zum Einsatz.

Websocket

Die Websocket-API wurde im Zuge von HTML5 entwickelt, um die Kommunikation über HTTP um einen Rückkanal vom Server zum Client zu erweitern. Diese Verbindung bietet Ihnen als Entwickler die Möglichkeit bidirektional zwischen Client und Server zu kommunizieren. Websocket ist dabei ein eigenständiges Protokoll, das auf der gleichen Ebene wie HTTP operiert, also auf Basis von TCP. Die Schnittstelle wird vom World Wide Web Consortium, kurz W3C, spezifiziert und ist unter <http://www.w3.org/TR/websockets/> zu finden.

Das letzte Ereignis, auf das Sie reagieren können, ist das `clientError`-Event. Dieses Ereignis stellt in gewisser Weise eine Art Exception-Mechanismus für HTTP-Verbindungen dar. Sie haben über diesen Mechanismus die Möglichkeit, auf Fehler oder Probleme im Zusammenhang mit Verbindungen zu Ihren Clients zu reagieren. Die Signatur der Callback-Funktion ist denkbar einfach. Als einziges Argument akzeptiert diese ein Objekt, das die Fehlermeldung repräsentiert.

Mit den hier vorgestellten Events können Sie nicht nur auf normale Anfragen von Clients reagieren, sondern auch das Verhalten Ihres HTTP-Servers um ein Vielfaches feiner steuern. Außerdem ist es möglich, zusätzliche Features des HTTP-Standards zu unterstützen. Die Kommunikation bei HTTP ist grundsätzlich requestbasiert. Das bedeutet, dass die Kommunikation durch Anfragen von Clients zustande kommt. Aus diesem Grund erfahren Sie in den folgenden Abschnitten mehr über das Request-Objekt, das innerhalb der Node.js-Plattform eine derartige Anfrage eines Clients repräsentiert.

6.1.3 Das Request-Objekt

Das Request-Objekt stellt die Anfrage des Clients an den Server dar. Es enthält sämtliche Informationen, die der Server benötigt, um die Anfrage korrekt zu beantworten.

Eine HTTP-Anfrage besteht im Normalfall aus zwei Teilen: dem Header und dem Nachrichtenkörper. Der Header enthält Metainformationen wie beispielsweise in welchem Format die Antwort des Servers erfolgen soll. Der Nachrichtenkörper, kurz Body, enthält die Nutzdaten der Anfrage. Der Body muss, je nach Typ der Anfrage, nicht zwingend Daten enthalten. Ein konkretes Beispiel hierfür ist eine Anfrage an einen Webserver, in der Sie formulieren, dass Sie eine bestimmte Seite sehen möchten. Sie verschicken derartige Anfragen beim täglichen Surfen im Internet. In diesem Fall wird kein Body benötigt. Der Header enthält sämtliche relevanten Informationen.

Anders ist der Fall beim Absenden von Formularen. Hier müssen Sie Daten an den Server übermitteln. Den Aufbau der entsprechenden HTTP-Anfrage übernimmt Ihr

Browser. Er formuliert den korrekten Header und übernimmt die Formulardaten als Schlüssel-Wert-Paare in den Körper der Nachricht. Auf der Serverseite können Sie nun auf die Formularwerte innerhalb der Anfrage zugreifen. Ein weiteres Beispiel, in dem Sie Daten per Anfragen an den Server übermitteln, sind Dateiuploads. Je nachdem, welche Art von Daten Sie hochladen, können in diesem Fall auch sehr große Mengen anfallen, mit denen Sie auf Serverseite korrekt umgehen müssen.

Eine Mischung aus beiden Typen, also Anfragen mit leerem Nachrichtenkörper und Anfragen, bei denen Daten übermittelt werden, sind GET-Requests, die eigentlich dazu gedacht sind, Ressourcen zum Auslesen vom Server anzufragen, bei denen Daten in der URL verschlüsselt untergebracht sind. Sie erreichen dies ganz einfach, indem Sie in einem HTML-Formular im `form`-Tag das Attribut `method` mit dem Wert `get` versehen. Aber auch mit derartigen Fällen können Sie innerhalb Ihrer Node.js-Applikation ohne größere Probleme umgehen.

Wie bereits erwähnt, ist das `Request`-Objekt das Mittel, mit dessen Hilfe der Client dem Server mitteilt, was dieser tun soll. Die Spanne reicht hier von der Auslieferung von Ressourcen wie Webseiten, Bildern oder Videos bis hin zur Speicherung von Informationen und ganzen Dateien.

Sie können einen Request nicht als ein Datenpaket betrachten. Das Bild eines Datenstroms trifft die Realität eher. Die Header-Informationen liegen direkt als Eigenschaften des `Request`-Objekts vor, der Nachrichtenkörper allerdings muss gesondert verarbeitet werden.

Im Folgenden möchte ich Ihnen zunächst die Eigenschaften und Methoden des `Request`-Objekts vorstellen, bevor ich zur Behandlung des Bodys übergehe.

HTTP bietet verschiedene Methoden. Möchten Sie diese korrekt behandeln, müssen Sie die Methode eines Requests auslesen können. Node.js bietet Ihnen zu diesem Zweck die Eigenschaft `method`. Diese Eigenschaft sollten Sie lediglich `read-only` verwenden. Eine Änderung dieser Eigenschaft wirkt sich lediglich auf das Verhalten Ihrer Applikation, nicht aber auf den Request an sich aus. Der Wert dieser Eigenschaft ist eine Zeichenkette, die beispielsweise den Wert `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `TRACE` oder auch `OPTIONS` annehmen kann. Möchten Sie Node.js wie einen normalen Webserver, wie beispielsweise Apache, verwenden, kommen Sie wahrscheinlich mit den Methoden `GET` und `POST` aus. Wollen Sie jedoch die gesamte Bandbreite von HTTP ausnutzen, kann diese Eigenschaft des `Request`-Objekts den Namen jeder definierten HTTP-Methode tragen.

Neben der verwendeten Methode enthält das `Request`-Objekt auch noch die URL, die der Benutzer mit dieser Anfrage angefordert hat. Die URL identifiziert die Ressource, die der Server benutzen soll. Zusammen mit der Methode entsteht daraus der Entschluss, welche Ressource auf welche Art verwendet werden soll. Eine URL für eine HTTP-Ressource besteht normalerweise aus dem Schema `http://` beziehungsweise

`https://`, dem Hostnamen und dem Port. Im Falle des Beispiels aus Listing 6.1 ist dies `localhost:8080`. Außerdem kommt ein URL-Pfad zum Einsatz, das sind Zeichenketten, die durch Schrägstriche voneinander getrennt sind und eine Art Hierarchie bilden, über die sich die Ressourcen auf dem Server ansprechen lassen. Den letzten Teil der URL bildet schließlich der Query String, der zur Übermittlung von Daten in einem GET-Request dient. Die Eigenschaft `url` des `Request`-Objekts enthält lediglich den URL-Pfad und den Query String. Wie auch schon die `method`-Eigenschaft, sollte auch die `url` nicht nachträglich auf Serverseite angepasst werden, da so nur Informationen verfälscht werden.

Die `headers`-Eigenschaft stellt einen weiteren wichtigen Bestandteil des `Request`-Objekts dar. Über die HTTP-Header kann ein Benutzer beziehungsweise das Client-programm des Benutzers dem Server Metainformationen übermitteln. So kann beispielsweise über den `Accept`-Header definiert werden, welches Format die Antwort des Servers aufweisen sollte. Machen Sie hier mehrere Angaben, werden die zuerst genannten bevorzugt. Neben dem Format können Sie beispielsweise mithilfe von `Accept-Charset` angeben, welche Zeichensätze Sie in der Antwort akzeptieren. Mit den Informationen aus dem HTTP-Header lässt sich jedoch nicht nur das Verhalten des Servers direkt beeinflussen. Auch das Caching von Inhalten sowohl auf Client- als auch auf Serverseite lässt sich mit Direktiven wie `cache-control` steuern. Insgesamt spezifiziert das W3C über 40 verschiedene Header-Direktiven für HTTP. Details zu diesem Thema finden Sie unter <http://www.ietf.org/rfc/rfc2616.txt>. Auf die Daten des HTTP-Headers greifen Sie über die `headers`-Eigenschaft zu. Die einzelnen Schlüssel und ihre entsprechenden Werte liegen in einer Objektstruktur vor.

Das Gegenstück zu den Header-Informationen sind die sogenannten Trailer. Auf sie können Sie ähnlich wie schon auf die Header über die Eigenschaft `trailers` zugreifen.

Neben diesen Informationen können Sie auch die Version des HTTP, die für die Anfrage verwendet wurde, auslesen. Zu diesem Zweck stellt Ihnen das `Request`-Objekt drei Eigenschaften zur Verfügung: `httpVersion`, `httpVersionMajor` und `httpVersionMinor`. Über die Eigenschaft `httpVersion` erhalten Sie eine Zeichenkette, die die komplette HTTP-Version, also beispielsweise 1.1, enthält. Über `httpVersionMajor` beziehungsweise `httpVersionMinor` können Sie auf die einzelnen Bestandteile vor und nach dem Punkt zugreifen.

Die letzte Eigenschaft, auf die Sie über das `Request`-Objekt zugreifen können, ist die `connection`-Eigenschaft. Sie enthält ein `Socket`-Objekt, das die Verbindung zwischen Client und Server darstellt. Es handelt sich dabei um das gleiche Objekt, das Sie als Argument für die Callback-Funktion erhalten, wenn Sie einen Eventlistener für das `connection`-Event des Servers erstellen.

Die drei wichtigsten Eigenschaften des `Request`-Objekts, mit denen Sie wahrscheinlich des Öfteren in Berührung kommen werden, sind `method`, `url` und `headers`. Geben

Sie diese Informationen für das Beispiel in Listing 6.1 für einen beliebigen Browser als Client aus, erhalten Sie eine Ausgabe, die in etwa der in Listing 6.3 entsprechen sollte.

```
method: 'GET'
url: '/'
headers: { host: 'localhost:8080',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.97 Safari/537.11',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-encoding': 'gzip,deflate,sdch',
  'accept-language': 'en-US,en;q=0.8',
  'accept-charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.3' }
```

Listing 6.3 method, url und headers für eine Anfrage

Nachdem Sie nun gesehen haben, wie Sie auf die Metainformationen eines Requests zugreifen können, erfahren Sie jetzt, wie Sie die eigentlichen Nutzinformationen des Requests verwenden können.

Die eigentliche Problemstellung beim Zugriff auf den Body einer HTTP-Anfrage ist, dass dieser in mehreren Teilen übertragen werden kann. Aus diesem Grund bietet Node.js hier mehrere Events, mit deren Hilfe mit derartigen Anfragen umgegangen werden kann. Insgesamt handelt es sich dabei um die drei Events `data`, `end` und `close`. In Listing 6.4 sehen Sie die Verwendung der Events am konkreten Beispiel.

```
var http = require('http');

http.createServer(
  function(request, response) {
    var body = '';
    request.on('data', function (data) {
      body += data.toString();
    });
    request.on('end', function () {
      console.log(body);
    });
    response.writeHead(200, {'content-type': 'text/plain'});
    response.end(body);
  }).listen(8080, 'localhost');
```

Listing 6.4 Auslesen des Request-Bodys

Die Daten eines Requests können in mehreren Chunks übertragen werden. Das sind einzelne Datenpakete, die auf Serverseite wieder zu einem Gesamtpaket zusammengefügt werden müssen, damit sie erneut die komplette Anfrage ergeben. Für jeden Teil der Nachricht, der übermittelt wird, wird ein `data`-Event des `Request`-Objekts ausgelöst. In Listing 6.4 sehen Sie, wie sich das `data`-Event dazu nutzen lässt, die einzelnen Chunks in der Variablen `body` wieder zusammenzusetzen. Die Callback-Funktion erhält dabei ein Buffer-Objekt als Argument. Hier ist es wichtig, dass Sie die `toString`-Methode verwenden, falls Sie den Nachrichtenkörper als Zeichenkette weiterverwenden möchten.

Sobald die Übermittlung sämtlicher Teile der Nachricht abgeschlossen ist, wird das `end`-Event gefeuert. Sobald dieses Event auftritt, können Sie sicher sein, dass für diesen Request keine weiteren Daten gesendet werden und Sie so gefahrlos die Anfrage bearbeiten können. Auch hier müssen Sie, wie beim `data`-Event, erneut eine Callback-Funktion definieren, innerhalb derer Sie die eigentliche Bearbeitung des Requests vornehmen. In Listing 6.4 wird lediglich der zusammengesetzte Body der Nachricht ausgegeben. Sie können allerdings hier beliebige Operationen durchführen.

Das dritte Event `close` schließlich signalisiert, dass die Verbindung beendet wurde. Das bedeutet für Sie, dass Sie nicht sicher sein können, dass auch wirklich sämtliche Teile der Nachricht erfolgreich übermittelt wurden. Wie beim `end`-Event treten auch hier keinesfalls weitere `data`-Events auf, die zum aktuellen Request gehören.

Sowohl `end` als auch `close` können jeweils nur einmal pro Request gefeuert werden. Es besteht auch die Möglichkeit, dass für einen Request beide Events auftreten. Ist dies der Fall, kann ein `end`-Event allenfalls vor einem `close` auftreten.

Mit den vorgestellten Methoden, Eigenschaften und Ereignissen können Sie nun eingehende Anfragen von Clients auswerten. In den nächsten Abschnitten erfahren Sie, wie Sie nach einer Anfrage eine passende Antwort an den Client schicken können.

6.1.4 Das Response-Objekt

Das `Response`-Objekt ist das Gegenstück zum `Request`. Es dient zur Beantwortung einer Anfrage, die durch einen Client formuliert und an den Server gesendet wurde.

Die Anfrage eines Clients setzt sich aus der Methode, der URL, den Header-Informationen und dem Nachrichtenkörper zusammen. Die Antwort des Servers auf der Gegenseite besteht aus einem Statuscode, ebenfalls bestimmten Header-Informationen und einem Nachrichtenkörper. `Request` und `Response` sind also grundsätzlich unterschiedlich aufgebaut. Da HTTP ein requestbasiertes Protokoll ist, sollte es für jeden Request vom Client eine entsprechende Response vom Server geben. Der Server wiederum kann ungefragt keine Responses zum Client senden. Er ist also darauf angewiesen, dass der Client die Kommunikation initiiert.

Sie haben im Verlauf dieses Kapitels bereits die verschiedenen Events kennengelernt, die Ihnen der HTTP-Server von Node.js zur Verfügung stellt. Davon ist eines, das request-Event, für Sie an dieser Stelle von besonderer Bedeutung. Sobald eine Anfrage eines Clients beim Server eingeht, wird ein solches Event gefeuert. Die Callback-Funktion erhält zwei Objekte als Argumente. Das erste ist das Request-Objekt, das die Anfrage repräsentiert. Das zweite Objekt ist das Response-Objekt, mit dessen Hilfe Sie die Antwort an den Client formulieren können. Dieses Objekt lernen Sie im folgenden Abschnitt näher kennen.

Eine Nachricht, die über HTTP versendet wird, besteht aus drei Teilen, dem Header, dem Body und dem Trailer. Der Header wird von den Clients als Erstes geprüft, damit er weiß, wie mit der Nachricht umzugehen ist. Was der Client mit der URL und der Methode bei der Anfrage formuliert, beantwortet der Server mit den sogenannten Statuscodes. Der Statuscode ist eine standardisierte Möglichkeit, dem Client den Status der Bearbeitung seiner Anfrage mitzuteilen.

Der Statuscode ist eine vierstellige Zahl, die aus einer von fünf Kategorien stammt. Tabelle 6.2 zeigt Ihnen die verschiedenen Kategorien.

Kategorie	Beschreibung
100 – 199	Information zur Anfrage
200 – 299	Rückmeldung über den Erfolg der Anfrage
300 – 399	Der Client muss weitere Aktionen unternehmen, Umleitung.
400 – 499	Fehler auf der Seite des Clients
500 – 599	Fehler auf der Seite des Servers

Tabelle 6.2 Kategorien der Statuscodes

Nicht nur die Kategorien der Statuscodes sind standardisiert, auch die einzelnen Codes unterliegen strengen Regeln. Node.js bietet mit dem `http.STATUS_CODES`-Objekt eine Möglichkeit, mit dessen Hilfe Sie die einzelnen Statuscodes in Zeichenketten auflösen können. In Listing 6.5 können Sie sehen, wie Sie auf der Node.js-Eingabeaufforderung die Übersetzung eines Statuscodes in die entsprechende Zeichenkette vornehmen können. Konkret lassen Sie sich hier den Statuscode 404 in die Zeichenkette `not found` übersetzen.

```
$ node
> var http = require('http');
undefined
```

```
> http.STATUS_CODES[404];
'Not Found'
>
```

Listing 6.5 Auflösung des HTTP-Statuscodes

Wie Sie in Listing 6.5 sehen können, besteht das `STATUS_CODES`-Objekt aus Schlüssel-Wert-Paaren, wobei der Schlüssel der Statuscode und der Wert die zugehörige Zeichenkette ist.

Die einfachste Variante, um einen korrekten Answerheader zu formulieren, besteht in der Verwendung der Methode `writeHead`, die im Normalfall zwei Argumente benötigt. Das erste besteht aus einer Zahl, die den Statuscode repräsentiert, und das zweite aus einem Objekt mit Header-Informationen. Setzen Sie diese Methode ein, haben Sie nach dem Aufruf keine Möglichkeit, weitere Header-Informationen anzugeben. Sie müssen diese Methode außerdem aufrufen, bevor Sie mit `write` oder `end` den Body der Nachricht versenden. Diese Methode kümmert sich außerdem um den korrekten Umgang mit dem Statuscode, den Sie als erstes Argument übergeben.

Eine weitere Möglichkeit, mit den Header-Informationen der Response zu arbeiten, besteht darin, die Informationen von Hand zu setzen. Den Statuscode können Sie über die Eigenschaft `statusCode` des `Response`-Objekts setzen. Hierbei müssen Sie lediglich die vierstellige Zahl, die den Statuscode darstellt, zuweisen. Die einzelnen Felder des HTTP-Headers können Sie mit der Methode `setHeader` setzen. Das erste Argument dieser Methode ist der Name des Headers, also zum Beispiel `content-type`. Das zweite Argument ist der Wert, der diesem Header zugewiesen werden soll, also etwa `text/html`. Rufen Sie diese Methode mehrmals für den gleichen Header auf, wird der Wert überschrieben. Möchten Sie mehrere Werte setzen, müssen Sie als zweites Argument ein Array aus Zeichenketten statt einer Zeichenkette verwenden.

Verwenden Sie die `setHeader`-Methode zum Aufbau des HTTP-Headers, können Sie die Werte beliebig oft bis zum Zeitpunkt, zu dem die Header zum Client gesendet werden, modifizieren. Zu diesem Zweck stehen Ihnen zwei weitere Methoden zur Verfügung. Mit `getHeader` können Sie den Wert, der für einen Header vorgesehen ist, auslesen, und mit `removeHeader` können Sie einen bereits gesetzten Header wieder entfernen. Beide Methoden erhalten den betroffenen Header als Zeichenkette als Argument im Aufruf.

Der HTTP macht es erforderlich, dass ein Date-Header mit der Antwort an den Client gesendet wird. Node.js fügt diesen Header automatisch ein. Möchten Sie dieses Verhalten beeinflussen, müssen Sie die Eigenschaft `sendDate` des `Response`-Objekts auf den Wert `false` setzen.

Nachdem Sie den Antwortheader wie gewünscht formuliert haben, können Sie im nächsten Schritt dazu übergehen, die eigentliche Antwort an den Client zu erstellen. Diese liegt in den meisten Fällen im Body der Nachricht.

Mit der Formulierung eines Requests beginnt ein Client die Kommunikation mit einem Server. Ziel dieser Kommunikation ist es, eine bestimmte Reaktion des Web-servers zu erzeugen. Zu diesem Zweck teilt der Client dem Server im Request mit, welche Antwort er vom Server erwartet. Sie als Entwickler müssen dann dafür sorgen, dass Ihr Server in der Lage ist, die eingehenden Requests Ihrer Clients korrekt zu analysieren, eine entsprechende Antwort zu generieren und diese zurück an den Client zu senden.

Insgesamt gibt es im Rahmen des `Response`-Objekts zwei Methoden, mit deren Hilfe Sie den Body Ihrer Nachricht formulieren können. Wie auch schon beim `Request`-Objekt müssen Sie hier auch wieder beachten, dass eine HTTP-Nachricht aus mehreren Teilen bestehen kann. Mit der `write`-Methode schicken Sie einen Chunk, also einen bestimmten Teil einer HTTP-Nachricht, zum Client. Sie können diese Methode beliebig oft für eine Response aufrufen. Um das Senden der Nachricht abzuschließen, verwenden Sie die `end`-Methode.

Sie können außerdem direkt mit der `end`-Methode Daten an den Client schicken. Zu diesem Zweck übergeben Sie die Daten als Argument an diese Methode. Sie verhält sich dann, als würden Sie zuerst die `write`-Methode mit den Daten und anschließend die `end`-Methode ohne weitere Argumente aufrufen. Zur Formatierung der zu sendenden Daten können Sie auf zwei Varianten zurückgreifen. Zum einen können Sie die Daten als Zeichenkette an die Methode übergeben, und zum anderen können Sie die Daten als Buffer-Objekt angeben. Sollten Sie sich für die Zeichenkette entscheiden, können Sie als zweites Argument das Encoding der Zeichen angeben. Als Standard nimmt Node.js hierfür UTF-8 an.

Haben Sie vor dem Aufruf von `write` noch nicht `writeHead`, die Methode zur Vorbereitung zum Senden der Header-Informationen, aufgerufen, erledigt dies `write` für Sie und sendet die impliziten Header-Informationen zum Client.

Der Rückgabewert der `write`-Methode sagt aus, ob die Nachricht erfolgreich an den Kernel übergeben oder ob die Nachricht in eine Warteschlange gestellt wurde. Der erste Fall führt dazu, dass der Wert `true` zurückgegeben wird, der zweite führt zu einem `false`. Damit Sie wissen, wann der Puffer wieder frei ist, wird ein `drain`-Event gefeuert, sobald dies der Fall ist. Sie können einen Eventlistener auf dieses Event binden und erst dann mit dem Senden weiterer Nachrichten fortfahren, sobald dieses Ereignis auftritt.

Bevor Sie mit einem Aufruf der `end`-Methode die Nachricht abschließen, können Sie Trailer-Informationen an die Nachricht anhängen. Mit diesen können Sie zusätzlich

zum Header weitere Metainformationen zur Nachricht hinzufügen. Mit der Methode `addTrailers` können Sie ein Objekt mit den entsprechenden Informationen angeben, das dann zum Client gesendet wird.

Zu beachten ist außerdem, dass Sie eine Nachricht lediglich einmal mit einem Aufruf von `end` abschließen können. Jeder weitere Aufruf dieser Methode führt zwar zu keinem Fehler, die Daten, die Sie an diesen Methodenaufruf übergeben, werden allerdings ignoriert und kommen nicht beim Client an.

Mit den vorgestellten Objekten und Konzepten sind Sie nun in der Lage, einen voll funktionsfähigen Webserver unter Node.js zu implementieren und ihn zu betreiben. Sie können die eingehenden Anfragen von Clients auslesen und analysieren und auf Basis dieser Informationen entsprechende Antworten generieren und diese zum Client zurücksenden. Als Nächstes sehen Sie, wie Node.js in die Rolle eines HTTP-Clients schlüpft, selbst Anfragen an Server stellen und diese weiterverarbeiten kann. In Listing 6.2 haben Sie bereits gesehen, wie Sie mit dem `Response`-Objekt Antworten an den Client erstellen können.

6.2 HTTP-Client mit Node.js

Mit dem HTTP-Server können Sie auf die Anfragen von Clients reagieren. Wenn Sie allerdings Daten von anderen Webservern, beispielsweise über einen Webservice, abrufen möchten, ist der HTTP-Server wenig hilfreich. An dieser Stelle kommt der HTTP-Client ins Spiel. Mit dem Client können Sie Anfragen an Server formulieren und haben sämtliche Aspekte des Requests vom Header bis zum Body unter Ihrer Kontrolle.

Bislang konnten Sie zur Erstellung eines HTTP-Clients analog zum Server die Methode `createClient` verwenden. Diese Methode ist allerdings mittlerweile von den Entwicklern als *deprecated* markiert und wird mittelfristig aus dem `http`-Modul entfernt. Statt dieser Methode sollten Sie die `request`-Methode verwenden. Sie tritt an die Stelle von `createClient` und sorgt dafür, dass Sie Anfragen formulieren können.

In Listing 6.6 sehen Sie ein sehr einfaches Beispiel, wie Sie mit der `request`-Methode eine Anfrage an Google senden können.

```
var http = require('http');

http.request('http://www.google.de', function (response) {
  var body = '';
  response.on('data', function (data) {
```

```

        console.log(data.toString());
    });
}).end();

```

Listing 6.6 Erstellen einer einfachen Anfrage

Das Resultat dieser Anfrage ist eine Repräsentation der Startseite von Google in HTML. Der Request wird, wie in Node.js üblich, asynchron ausgeführt. Zu diesem Zweck übergeben Sie der Methode als erstes Argument die Optionen des Requests und als zweites Argument eine Callback-Funktion, die ausgeführt werden soll, sobald die Antwort des Servers vorliegt. Die Optionen können entweder in Form einer Zeichenkette, wie Sie es im Beispiel sehen, oder aber als Objektstruktur übergeben werden. Entscheiden Sie sich für die Zeichenkette, wird diese intern von Node.js mit der `parse`-Methode des `url`-Moduls in ein Objekt umgewandelt.

Wichtig an diesem Beispiel ist außerdem der Aufruf der `end`-Methode. Dieser sorgt dafür, dass die Anfrage an den Server gesendet wird. Führen Sie die `end`-Methode nicht aus, wird das Script nicht beendet, da es auf das Senden des Requests wartet, und Sie erhalten keine Ausgabe, da keine Rückmeldung vom Server vorliegen wird.

Innerhalb der Callback-Funktion wird auf Events zurückgegriffen, die gefeuert werden, sobald Daten vom Server ankommen. Eines dieser Ereignisse, die im Verlauf dieses Kapitels noch näher betrachtet wird, ist das `data`-Event. Ähnlich wie beim HTTP-Server können Sie über dieses Event auf die Daten in der Antwort des Servers zugreifen und diese, wie im Beispiel gezeigt, ausgeben.

Neben der `request`-Methode haben Sie eine weitere Möglichkeit, Anfragen zu erstellen. Da es sich bei einem Großteil der Requests um Anfragen mit der HTTP-Methode GET handelt, bietet die Node.js-Plattform die Methode `get` an. Der einzige Unterschied zur `request`-Methode besteht zum einen darin, dass hier bereits der korrekte Wert für die HTTP-Methode, also `get`, gesetzt wird, und zum anderen, dass Sie `end` nicht mehr manuell aufrufen müssen, um die Anfrage abzuschicken. In Listing 6.7 sehen Sie den Einsatz der `get`-Methode. Die Aufgabenstellung ist die gleiche wie schon in Listing 6.6.

```

var http = require('http');

http.get('http://www.google.de', function (response) {
    response.on('data', function (data) {
        console.log(data.toString());
    });
});

```

Listing 6.7 Request mit der `get`-Methode

6.2.1 Der http.Agent

Damit Sie Anfragen an andere Server senden können, stellt Ihnen Node.js mehrere Verbindungen zur Verfügung, über die die Requests gesendet werden können. Die Verwaltung dieser Verbindungen übernimmt die Plattform selbst, sodass Sie sich darum nicht zu kümmern brauchen. Intern kommt zu diesem Zweck die Klasse `http.Agent` zum Einsatz, die einen Pool von Sockets verwaltet und bei einem Request einen entsprechenden Socket zur Verfügung stellt. Die aktuelle Implementierung sorgt dafür, dass sämtliche ausgehende Verbindungen an einer zentralen Stelle verwaltet werden. Eine Konsequenz aus diesem Pool ist, dass Sie das Schließen inaktiver Verbindungen nicht selbst übernehmen müssen. Existiert für eine Verbindung zu einem Server kein Request mehr, der versendet werden soll, wird die Verbindung geschlossen. Sie können auch manuell dafür sorgen, dass ein Socket aus dem Pool entfernt wird, indem Sie ihn entweder mit dem `close`-Event oder mit dem Event `agentRemove` schließen.

Auf den Agent können Sie über die Eigenschaft `globalAgent` zugreifen. Über diese Instanz können Sie beispielsweise konfigurieren, wie viele Verbindungen einem Client maximal zu einem Host zur Verfügung stehen. Diese Information wird in der Eigenschaft `maxSockets` vorgehalten und weist den Standardwert 5 auf.

Die Instanz des Agents hat zwei weitere Eigenschaften, `sockets` und `requests`. Beide sollten Sie nicht bearbeiten, um die Funktionsfähigkeit Ihrer Applikation nicht negativ zu beeinflussen. Die Eigenschaft `sockets` enthält ein Array aller aktiven Socket-Verbindungen zu anderen Servern. Unter `requests` finden Sie eine Objektstruktur, die diejenigen Requests enthält, die noch keiner Socket-Verbindung zugeordnet sind.

Sie können die Verwendung des Socket-Poolings auch umgehen, indem Sie in den Optionen Ihres Requests die Option `agent` auf den Wert `false` setzen. Mit dieser Option haben Sie bereits eine erste Möglichkeit kennengelernt, wie Sie Requests durch zusätzliche Optionen steuern können. Im folgenden Abschnitt erfahren Sie mehr über die verfügbaren Optionen und deren Einsatzgebiet.

6.2.2 Die Anfrage-Optionen

In Listing 6.6 und Listing 6.7 haben Sie gesehen, dass eine einfache Zeichenkette und eine Callback-Funktion ausreichen, um eine Anfrage an einen Server zu senden. Damit Sie aber mehr Kontrolle über die Kommunikation mit einem Server haben, können Sie auch ein Objekt mit sehr viel mehr Optionen als nur einer URL angeben und so den Aufbau des Requests beeinflussen. Tabelle 6.3 listet die verfügbaren Optionen für Sie auf.

Option	Bedeutung
host	Zieladresse der Anfrage in Form einer IP-Adresse oder eines Hostnamens als Zeichenkette. Wird kein Wert angegeben, wird localhost angenommen.
hostname	Diese Option sollte zugunsten von host nicht verwendet werden.
port	Portnummer, an die die Anfrage gesendet werden soll. Werden keine Angaben gemacht, wird Port 80 verwendet.
localAddress	Lokale Schnittstelle, an die gebunden werden soll, zum Beispiel: 127.0.0.2.
socketPath	Soll ein bestimmter Socket für die Verbindung verwendet werden, kann hier der Pfad angegeben werden. Der Agent wird in diesem Fall umgangen.
method	Angabe der HTTP-Methode, die verwendet werden soll. Standardwert ist GET. Möglich sind auch andere HTTP-Methoden wie beispielsweise POST, PUT oder DELETE.
path	Pfad, an den die Anfrage gesendet werden soll. Dieser Wert sollte die gesamte URL nach dem Hostnamen und dem Port enthalten. Standardwert ist /.
headers	Objektstruktur mit Header-Informationen für die Anfrage
auth	Die Authentifizierungsinformationen, also eine Kombination aus Benutzername und Passwort. Es wird ein Authorization-Header erstellt.
agent	Mit dieser Option kann das Verhalten des Agents beeinflusst werden. Wird kein Wert gesetzt, wird der allgemeine Agent verwendet. Außerdem kann ein Agent-Objekt übergeben werden, das in diesem Fall verwendet wird, oder mit dem Wert false angegeben werden, dass kein Agent verwendet werden soll.

Tabelle 6.3 Verfügbare Request-Optionen

Ein Beispiel in Listing 6.8 zeigt Ihnen die Verwendung einiger häufig benötigter Optionen.

```
var http = require('http');

var request = http.request({
  host: 'www.google.de',
  port: 80,
```

```

    method: 'GET',
    path: '/?q=node.js',
    headers: {
      'accept': 'text/html',
      'userAgent': 'node.js'
    },
    agent: false
  }, function (response) {
    response.on('data', function (data) {
      console.log(data.toString());
    });
  });
request.end();

```

Listing 6.8 Request mit Optionen

In diesem Beispiel wird erneut eine HTTP-GET-Anfrage an Google gesendet. Dieses Mal wird jedoch die Zeichenkette `node.js` in das Eingabefeld eingefügt. Außerdem werden die zwei Felder `accept` und `userAgent` im Header der Anfrage gesetzt. Mit der Option `agent: false` teilen wir dem Request-Objekt mit, dass in diesem Fall kein Agent für das Connection-Pooling eingesetzt werden soll. Die Anfrage wird schließlich mit einem Aufruf der `end`-Methode abgeschickt.

Der wichtigste Teil der Optionen verbirgt sich hinter der Eigenschaft `headers`. Mit dieser setzen Sie beliebige Header-Informationen in Ihrem Request und beeinflussen so die Antwort des Servers. So können Sie beispielsweise durch den `accept`-Header angeben, welches Format die Antwort haben soll. Der Server auf der Gegenseite muss natürlich die entsprechenden Header auswerten und korrekt antworten.

6.2.3 Die Klasse `ClientRequest`

Das Ergebnis eines Aufrufs der `request`-Methode ist eine Instanz der Klasse `ClientRequest`. Diese Klasse bietet Ihnen neben der bereits bekannten `end`-Methode eine Reihe weiterer Methoden, Eigenschaften und Events, mit deren Hilfe Sie mit dem Request und der darauffolgenden Response des Servers arbeiten können.

Das erste Event, das ich in diesem Zusammenhang behandeln möchte, ist das `response`-Event. Es wird einmalig ausgeführt, sobald eine Antwort des Servers vorliegt. Die Callback-Funktion, die Sie beim Aufruf der `request`-Methode übergeben, wird an dieses Event gebunden. Da es sich bei der Klasse `ClientRequest` um eine von der Klasse `EventEmitter` abgeleitete Klasse handelt, haben Sie die Möglichkeit, mehrere Eventlistener an ein Event zu binden. Sämtliche Callback-Funktionen, die an dieses Event gebunden sind, erhalten eine Instanz der Klasse `ClientResponse`. Über die

Behandlung der Antwort des Servers erfahren Sie im Verlauf der nächsten Abschnitte noch mehr.

Die Klasse `ClientRequest` unterstützt vier weitere Events. Das `socket`-Event wird ausgelöst, sobald einem Request ein Socket zugeteilt wird. Die Callback-Funktion erhält das entsprechende Socket-Objekt als Argument.

`connect` lautet der Name des Events, das auftritt, wenn eine Anfrage mit der HTTP-Methode `CONNECT` ausgeführt wurde. Die Argumente der Callback-Funktion sind im Einzelnen: ein Objekt, das die Antwort des Servers beinhaltet, das Socket-Objekt, das für die Verbindung verantwortlich ist, und das dritte Argument, mit dem Sie schließlich Zugriff auf den ersten Teil der übermittelten Daten haben.

Das nächste Event `upgrade` wird ausgelöst, wenn ein Server auf die Anfrage nach einem Protokoll-Upgrade antwortet. Dies kann beispielsweise der Fall bei einer Verbindung mit TLS oder Websockets sein. Als Argumente erhält die Callback-Funktion die gleichen Objekte wie schon bei `connect`, also die Antwort des Servers, eine Referenz auf die Socket-Verbindung und den Beginn der Antwort des Servers.

Das vierte und letzte Event, das ich Ihnen hier vorstellen möchte, ist das `continue`-Event. Bei der Beschreibung des HTTP-Servers haben Sie bereits gesehen, dass es ein spezielles Event `checkContinue` auf der Seite des Servers gibt, wenn der Client mit dem Header `Expect: 100-continue` anfragt. Ist der Server in Node.js implementiert, haben Sie die Möglichkeit, mit der Methode `response.writeContinue` den Body des Requests vom Client anzufragen. Der Aufruf dieser Methode wiederum führt beim Client zum Event `continue`. Hier sollten Sie sich dann darum kümmern, dass der Nachrichtenkörper korrekt zum Server gesendet wird. Die Callback-Funktion erhält keinerlei Argumente.

Ebenfalls eine wesentliche Rolle beim Aufbau einer Anfrage an den Server spielen die beiden Methoden `end` und `write`. Wie Sie bereits gesehen haben, wird `end` zum Absenden der Anfrage an den Server benötigt. Die `write`-Methode wird verwendet, um Daten an den Server zu senden. Analog zur Serverseite können Sie auch der `end`-Methode ein Buffer-Objekt oder eine Zeichenkette übergeben. Bei Requests mit der GET-Methode werden im Normalfall keine Daten übermittelt. Verwenden Sie statt GET jedoch die Methode POST oder PUT, können Sie sehr wohl Daten an den Server senden, wie Sie anhand eines einfachen Beispiels in Listing 6.9 sehen.

```
var http = require('http');
```

```
var request = http.request({
  host: 'localhost',
  port: 8080,
```

```

    method: 'POST',
    path: '/users',
    headers: {
        'Transfer-Encoding': 'chunked'
    }
}, function (response) {});
request.write(['{"firstname": "James", "surname": "Bond"},');
request.write(['{"firstname": "Sherlock", "surname": "Holmes"}']);
request.end();

```

Listing 6.9 Daten zum Server senden

Den Aufbau der Header-Informationen müssen Sie im Vergleich zum Beispiel in Listing 6.8 nur geringfügig anpassen. Die wichtigsten Änderungen sind die Anpassung in der Eigenschaft `method`, deren Wert auf `POST` geändert wird, und die hinzugefügte Direktive `'Transfer-Encoding': 'chunked'` in der Eigenschaft `headers`. Ansonsten können Sie die Angaben für Host, Port und Pfad so anpassen, dass die Nachricht an einen Server geschickt werden kann, bei dem Sie die eingehende Nachricht auswerten können. Das Ergebnis der beiden `write`-Aufrufe auf Serverseite, wenn Sie die Anfrage mit Node.js annehmen und die Daten auf der Standardausgabe anzeigen lassen, sehen Sie in Listing 6.10.

```

[{"firstname": "James", "surname": "Bond"}, {"firstname": "Sherlock", "surname": "Holmes"}]

```

Listing 6.10 Eingehende Daten beim Server

Die so entgegengenommenen Daten können Sie mithilfe des `JSON`-Objekts und der Methode `parse` in eine Objektstruktur umwandeln, mit der Sie weiterarbeiten können.

Neben den beiden vorgestellten Methoden `write` und `end` bietet Ihnen das `ClientRequest`-Objekt noch einige weitere Methoden. So können Sie beispielsweise mit der `abort`-Methode einen Request abbrechen.

Mithilfe der Methode `setTimeout` können Sie eine Zeitspanne definieren. Die Timeoutspanne gilt für die Zeit, in der der Socket nicht benutzt wird. Nach Ablauf der Zeitspanne wird ein `timeout`-Event ausgelöst. Die Socket-Verbindung wird allerdings nicht automatisch getrennt, dafür müssen Sie mit einem Aufruf von `end` oder `destroy` selbst sorgen.

Mit der Methode `setNoDelay` können Sie verhindern, dass die Daten, die Sie mithilfe von `write` zum Server senden möchten, gepuffert werden. Die Node.js-Plattform sorgt standardmäßig bereits dafür, dass diese Pufferung umgangen wird.

Um die Verbindung zwischen Client und Server offen zu halten und zu prüfen, ob die Gegenseite noch verfügbar ist, kann die Methode `setSocketKeepAlive` verwendet werden. Diese Methode akzeptiert zwei Argumente. Das erste ist ein boolescher Wert, der angibt, ob die Keep-Alive-Funktionalität aktiviert oder deaktiviert sein soll. Der Standardwert hierfür ist `false`, also ist Keep-Alive standardmäßig deaktiviert. Das zweite Argument ist optional und gibt an, wie viele Millisekunden gewartet werden soll, bevor zum ersten Mal geprüft wird. Wird keine Angabe gemacht, wird sofort geprüft.

Die Methoden `setTimeout`, `setNoDelay` und `setSocketKeepAlive` basieren auf den Sockets, die dem HTTP-Client zugrunde liegen. Ein Socket stellt in diesem Fall eine Abstraktion eines Datenstroms dar. Unter Node.js ist dies eine eventbasierte Schnittstelle zum Senden und Empfangen von Daten. Aus diesem Grund haben diese Methoden erst konkrete Auswirkungen, sobald einem Request ein Socket zugewiesen und dieser verbunden ist.

Innerhalb der Callback-Funktion für das `response`-Event erhalten Sie Zugriff auf ein Objekt der Klasse `ClientResponse`. Dieses Objekt repräsentiert die Antwort des Servers auf Ihre Anfrage. In den folgenden Abschnitten lernen Sie die Details dieses Objekts kennen und wie Sie mit ihm arbeiten können.

6.2.4 Die Antwort des Servers

Die Kommunikation über HTTP beginnt stets durch eine Anfrage des Clients. Der Server antwortet seinerseits auf diese Anfragen. Je nach Art der Anfrage unterscheiden sich die Antworten teils erheblich voneinander. Der Standardfall im Internet ist, dass ein Client durch die HTTP-Methode GET eine Ressource, meist eine Webseite, anfragt. Der Server liefert diese dann aus. Erstellen Sie einen HTTP-Client mit Node.js, möchten Sie in den meisten Fällen die Antwort des Servers automatisiert weiterverarbeiten. Die von einem Webserver ausgelieferten HTML-Seiten sind für diesen Zweck eher ungeeignet. Sinnvoller ist in diesem Fall, wenn der Server mit einer strukturierten XML-Datei oder besser noch mit Daten im JSON-Format antwortet. Diese Daten eignen sich wesentlich besser für eine automatische Weiterverarbeitung.

Neben den reinen Nutzdaten übermittelt der Server in seiner Antwort noch weitere Metainformationen. Eine der wichtigsten Eigenschaften eines `Response`-Objekts im Zusammenhang mit diesen Metainformationen ist der Statuscode, auf den Sie über die Eigenschaft `statusCode` zugreifen können. Für diesen werden die gleichen Werte verwendet wie auch schon für den Server. Eine Übersicht über die verschiedenen Kategorien erhalten Sie in Tabelle 6.2. Die Bedeutung der einzelnen Statuscodes können Sie über die Eigenschaft `STATUS_CODES` des `http`-Moduls auflösen. Außerdem können Sie über `httpVersion`, `httpVersionMajor` und `httpVersionMinor` die verwendete HTTP-Version auslesen. Auf die Header beziehungsweise Trailer der Nachricht kön-

nen Sie über die Eigenschaft `headers` und `trailers` zugreifen. Das Resultat ist jeweils ein Objekt mit den Informationen als Schlüssel-Werte-Paare.

Für den Umgang mit der eigentlichen Nachricht stehen Ihnen drei Ereignisse zur Verfügung, auf die Sie jeweils Callback-Funktionen binden können. Damit Sie beim Empfang der Nachricht keine Probleme mit der Codierung der Zeichen bekommen, steht Ihnen die Methode `setEncoding` zur Verfügung, mit deren Hilfe Sie den verwendeten Zeichensatz einstellen können. Die drei Ereignisse, die beim Empfang der Nachricht eine Rolle spielen, entsprechen denen, die Sie bereits vom `ServerRequest`-Objekt her kennen. Das `data`-Event wird ausgelöst, sobald der Server auf der Gegenseite Daten sendet. Die Antwort des Servers kann aus mehreren Paketen bestehen. Aus diesem Grund kann es vorkommen, dass innerhalb einer Response das `data`-Event mehrmals auftritt. Sobald das `end`-Event ausgelöst wird, können Sie sicher sein, dass die Antwort des Servers vollständig ist und keine weiteren Datenpakete gesendet werden. Wie bereits auf Serverseite steht auch im Client das dritte Event `close` dafür, dass die Verbindung geschlossen wurde, bevor die Gegenseite mit der `end`-Benachrichtigung den Sendevorgang abschließen konnte.

Das Beispiel in Listing 6.11 zeigt Ihnen, wie Sie mithilfe des `ClientResponse`-Objekts mit der Antwort eines Webserver umgehen können.

```
var http = require('http');

var request = http.request({
  host: 'localhost',
  port: 80,
  method: 'GET',
  path: '/'
});

request.on('response', function (response) {
  if (response.statusCode === 200 && response.headers['content-type'] ===
    'text/html') {
    var body = '';
    response.on('data', function (data) {
      body += data;
    });
    response.on('end', function () {
      console.log(body);
    });
  } else {
    console.log('an error occurred');
  }
});
```

```
});
```

```
request.end();
```

Listing 6.11 Umgang mit der Antwort eines Servers

In diesem Beispiel wird die Antwort des Servers ausgegeben, und zwar nur, wenn der Statuscode der Antwort den Wert 200 besitzt und der `content-type` im Header der Nachricht vom Typ `text/html` ist. Denkbar wäre hier beispielsweise, dass Sie ein Dokument im JSON- oder XML-Format anfordern und dieses anhand des Wertes im `content-type` in den korrekten Parser leiten und anschließend mit dem entstandenen Objekt weiterarbeiten.

Das `ClientResponse`-Objekt bietet Ihnen schließlich mit den beiden Methoden `pause` und `resume` eine Möglichkeit, mit der Sie das Auftreten der `data`-Events steuern können. Werden sehr viele `data`-Events ausgelöst, kann es vorkommen, dass Sie mit der Verarbeitung nicht nachkommen und den Strom der eingehenden Daten in Form der `data`-Events unterbrechen müssen. Dies erreichen Sie durch den Aufruf der `pause`-Methode. Haben Sie den Rückstand aufgearbeitet, können Sie den Prozess mit `resume` wieder fortsetzen.

In diesem Abschnitt haben Sie nun einen Blick hinter die Kulissen des `http`-Moduls erhalten. Sie haben gesehen, dass Node.js sowohl in die Rolle eines HTTP-Servers als auch in die eines HTTP-Clients schlüpfen kann.

6.3 Umgang mit URLs

Sowohl in der Rolle des Servers als auch als Client müssen Sie mit URLs arbeiten. Im Falle des Servers erhalten Sie eine Zeichenkette und müssen diese in die verschiedenen Teile aufbrechen, um sie korrekt weiterverarbeiten zu können. Im `ServerRequest`-Objekt können Sie über die Eigenschaft `url` auf die Adresse zugreifen, die der Benutzer in seinem Browser angegeben hat. Sie sollten dabei allerdings beachten, dass es sich bei der Zeichenkette, die Ihnen hier zur Verfügung steht, nicht um eine vollständige URL handelt. Es fehlen hier das Protokoll, der Hostname, der Port und eine eventuell angegebene Kombination aus Benutzername und Passwort. Auch auf einen eventuell angegebenen Hashwert können Sie nicht zugreifen. Konkret bedeutet das, dass Sie lediglich mit dem Pfad und dem Query String arbeiten können. Nehmen Sie als Ausgangssituation für Listing 6.12, dass der Benutzer die URL `http://localhost:8080/p/a/t/h?query=string` in seinem Browser angegeben hat.

```
var http = require('http'),
    url = require('url'),
    querystring = require('querystring'),
```

```

server = http.createServer();

server.on('request', function (req, res) {
  var urlObj = url.parse(req.url),
      path = urlObj.pathname.substr(1).split('/'),
      query = querystring.parse(urlObj.query);

  console.log(path);
  console.log(query);

  res.end();
});
server.listen(8080, 'localhost');

```

Listing 6.12 URLs im HTTP-Server

In Listing 6.12 haben Sie dann innerhalb der Callback-Funktion des `request`-Events über die Variablen `path` Zugriff auf ein Array mit den einzelnen Bestandteilen des URL-Pfades, und über `query` können Sie über ein Objekt auf die Teile des Query Strings zugreifen.

Auch auf Clientseite können Sie das `url`-Modul nutzen, um eine URL für eine Anfrage an einen Server zusammenzubauen. Der Vorteil dieser Vorgehensweise gegenüber der Behandlung von URLs als reinen Zeichenketten liegt einerseits darin, dass sich die URL durch das Objekt besser beschreiben lässt, andererseits nimmt Ihnen das Modul den korrekten Zusammenbau der URL ab. Listing 6.13 zeigt die Vorgehensweise an einem konkreten Beispiel.

```

var http = require('http'),
    url = require('url');

var reqUrl = {
  protocol: 'http',
  hostname: 'localhost',
  port: '8080',
  pathname: 'p/a/t/h',
  query: { query: 'string' }
};

http.request(url.format(reqUrl)).end();

```

Listing 6.13 URLs im HTTP-Client

Die `request`-Methode akzeptiert eine Reihe von Optionen. Verwenden Sie allerdings die `format`-Methode des `url`-Moduls im Zusammenspiel mit einem Objekt, das die URL beschreibt, können Sie mehr und besser detaillierte Informationen angeben. Die `format`-Methode sorgt dafür, dass eine korrekt formatierte Zeichenkette an die `request`-Methode übergeben wird. Ist das erste Argument dieser Methode eine Zeichenkette, wird diese automatisch mit `url.parse` in ein Objekt gewandelt, mit dem weitergearbeitet wird.

Sie können Listing 6.12 und Listing 6.13 auch kombinieren. Angenommen, Sie speichern den Quellcode aus Listing 6.12 in einer Datei mit dem Namen `server.js`, können Sie den Server mit dem Kommando `node server.js` starten. Anschließend führen Sie den Client aus Listing 6.13 aus, den Sie beispielsweise in einer Datei `client.js` speichern. Dies geschieht mit dem Kommando `node client.js`. Die Ausgabe, die Sie auf der Serverseite erhalten, sehen Sie in Listing 6.14.

```
$ node server.js
[ 'p', 'a', 't', 'h' ]
{ query: 'string' }
```

Listing 6.14 Kombination von HTTP-Client und -Server

Sowohl der HTTP-Client als auch der -Server basieren auf dem Konzept der Streams. Die Node.js-Plattform hat zu diesem Zweck eigene Schnittstellen geschaffen, die Sie im Folgenden genauer kennenlernen werden.

6.4 Streams in Node.js

Datenströme kommen in der Node.js-Plattform in mehreren Modulen zum Einsatz. Ein konkretes Beispiel von Datenströmen in Node.js, den sogenannten Streams, haben Sie im Zuge des `http`-Moduls kennengelernt. Ein Stream ist ein Kanal, über den, sobald er geöffnet ist, Daten gesendet und empfangen werden können. Innerhalb von Node.js wird unterschieden zwischen Streams, in die geschrieben wird, und solchen, aus denen gelesen werden kann.

Eine weitere Eigenschaft von Streams ist, dass über sie nicht kontinuierlich Daten gesendet werden müssen. Ein Stream wird also zwischen seinen beiden Endpunkten erstellt und kann dann zur Kommunikation verwendet werden. Je nach Art des Streams kann hierbei entweder zwischen Client und Server unterschieden werden, oder beide Endpunkte des Streams sind gleichberechtigt. Durch die Tatsache, dass nicht kontinuierlich Daten gesendet werden, wird es erforderlich, den Empfänger zu benachrichtigen, sobald Daten verfügbar sind. Aus diesem Grund werden alle Streams von `EventEmitter` abgeleitet. Nach der Initialisierung des Streams können

Sie über die `on`-Methode Callback-Funktionen auf die verschiedenen Events binden, die Ihnen der Stream zur Verfügung stellt.

Da innerhalb von Node.js an mehreren Stellen Streams zum Einsatz kommen, wurde mit dem `stream`-Modul eine Schnittstelle implementiert, die die Basis für alle übrigen Umsetzungen bildet und somit für Einheitlichkeit sorgt.

6.4.1 Readable Stream

Ein prominenter Vertreter der lesbaren Streams ist das `ServerRequest`-Objekt im `http`-Modul. Aber auch weitere Module setzen auf diese Klasse von Objekten. Arbeiten Sie mit einem Stream-Objekt und wollen herausfinden, ob es sich dabei um einen Readable Stream handelt, können Sie die Eigenschaft `readable` dieses Objekts abfragen. Weist es den Wert `true` auf, handelt es sich bei dem Objekt um einen Readable Stream.

Sobald Sie Daten über den Stream empfangen, wird ein `data`-Event ausgelöst. Ist zu diesem Zeitpunkt noch keine Callback-Funktion an dieses Event gebunden, gehen die Daten verloren. In der Callback-Funktion steht Ihnen als einziges Argument ein Objekt zur Verfügung, das die Informationen der Nachricht enthält. Haben Sie über einen Aufruf der `setEncoding`-Methode die Zeichencodierung der Nachricht definiert, ist das Argument vom Typ `String`, also eine gewöhnliche Zeichenkette. Haben Sie den Zeichensatz nicht festgelegt, erhalten Sie ein Objekt vom Typ `Buffer`.

Zur Fehlerbehandlung können Sie das `error`-Event verwenden. In der Callback-Funktion dieses Events steht Ihnen ein Objekt zur Verfügung, das den aufgetretenen Fehler repräsentiert. Innerhalb dieser Funktion können Sie dann den Fehler entsprechend behandeln und für die weitere einwandfreie Funktionsfähigkeit Ihrer Applikation sorgen.

Da eine Nachricht, die über einen Stream versendet wird, aus mehreren Teilen, also aus mehreren `data`-Events, bestehen kann, wird ein Mechanismus benötigt, um zu signalisieren, dass das Ende der Nachricht erreicht ist. Dieses Ende wird durch das Auftreten des `end`-Events gekennzeichnet. Erhalten Sie dieses Event, können Sie sicher sein, dass keine weiteren `data`-Events auf diesem Stream auftreten können.

Wie beim `ServerRequest` des `http`-Moduls, existiert auch bei Readable Streams im Allgemeinen noch ein weiteres Event, das `close`-Event. Dieses wird ausgelöst, falls die Verbindung getrennt wird, bevor das `end`-Event empfangen wurde.

Sie haben die Möglichkeit, einen Stream aktiv zu schließen, indem Sie die `destroy`-Methode aufrufen. Daten, die noch nicht gesendet wurden, werden gelöscht. Nachdem diese Methode ausgeführt wurde, kann der Stream, je nach Implementierung, ein `close`-Event auslösen.

Wie Sie bereits im `http`-Modul gesehen haben, können Sie über die Methoden `pause` und `resume` das Auftreten von `data`-Events unterbrechen beziehungsweise diese Unterbrechung wieder aufheben.

`Readable` Streams implementieren die `pipe`-Methode. Mit ihr lassen sich zwei Streams zu einer Pipe verbinden. Dabei werden die Daten aus einem `Readable` Stream direkt in einen `Writable` Stream weitergeleitet. Ein sehr einfaches Beispiel sehen Sie in Listing 6.15.

```
var http = require('http');

http.createServer(function (req, res) {
  req.pipe(process.stdout);
  res.end();
}).listen(8080);
```

Listing 6.15 Webserver und Standardausgabe verknüpfen

In diesem Beispiel senden Sie die Daten, die über den `Readable` Stream `ServerRequest` eingehen, direkt an die Standardausgabe des aktuellen `Node.js`-Prozesses. Um diesen Code zu testen, müssen Sie lediglich das Script ausführen und mit einem Browser oder mit `cURL` eine `POST`-Anfrage senden. Die Daten werden dann direkt ausgegeben. In Listing 6.16 sehen Sie die Ausgabe, die Sie erhalten, wenn Sie das Script starten und danach das Kommando `curl -X POST http://localhost:8080 -d '{"Hello": "World"}'` auf einer anderen Kommandozeile absetzen.

```
$ node server.js
{"Hello": "World"}
```

Listing 6.16 Ausgabe eines mit der Standardausgabe verknüpften Webservers

Neben den `Readable` Streams existieren auch Streams, über die Sie Daten versenden können.

6.4.2 Writable Stream

Wie schon bei den `Readable` Streams kommen Sie auch mit ihren Gegenstücken, den `Writable` Streams, bei der Entwicklung von Applikationen auf Basis der `Node.js`-Plattform häufiger in Berührung. Vertreter dieser Klasse von Datenströmen sind beispielsweise die Standardausgabe eines Prozesses, auf die Sie über `process.stdout` zugreifen können. Aber auch die `ServerResponse` des `http`-Moduls ist ein `Writable` Stream.

Sie können `Writable` Streams ähnlich identifizieren wie bereits `Readable` Streams. Sie besitzen eine Eigenschaft mit dem Namen `writable`. Diese weist den Wert `true` auf,

wenn es sich um einen Writable Stream handelt. Sobald der Stream nicht mehr schreibbar ist, also wenn Sie `end` oder `destroy` aufgerufen haben oder ein Fehler aufgetreten ist, wird diese Eigenschaft automatisch auf den Wert `false` gesetzt.

Für Writable Streams stehen Ihnen im Normalfall vier Events zur Verfügung, mit deren Hilfe Sie auf bestimmte Vorkommnisse reagieren können, wenn Sie daran entsprechende Callback-Funktionen binden.

Eines der wichtigsten Events ist das `error`-Event. Wie üblich bei Node.js wird dieses Event ausgelöst, sobald ein Fehler im Zusammenhang mit der Streamverbindung auftritt. Die Callback-Funktion erhält als einziges Argument ein Objekt, das den Fehler repräsentiert.

Bei Datenströmen kann es immer wieder vorkommen, dass die Verbindung getrennt wird. Auch in diesem Fall möchten Sie reagieren können. Das `close`-Event steht Ihnen hier als Schnittstelle zur Verfügung. Die Callback-Funktion erhält in diesem Fall keine Argumente.

Beim Schreiben in einen Stream kann es vorkommen, dass schneller in den Stream geschrieben wird, als die Daten versendet werden können. Um Datenverluste zu vermeiden, werden die Nachrichten gepuffert. Ein Aufruf der `write`-Methode zum Schreiben von Daten liefert den Wert `false` zurück, falls der Kernel-Puffer voll ist. Sobald sämtliche Nachrichten aus diesem Puffer verschickt sind und er geleert ist, wird ein `drain`-Event ausgelöst. Dieses signalisiert, dass Sie problemlos mit dem Senden von Daten fortfahren können.

Das letzte Event, das Sie im Zusammenhang mit den Writable Streams kennenlernen, ist das `pipe`-Event. Es wird ausgelöst, wenn ein Writable Stream über die `pipe`-Methode eines Readable Streams in diesen umgeleitet wird. Das Argument der Callback-Funktion ist ein Objekt, das den Readable Stream repräsentiert.

Neben diesen Events verfügt ein Writable Stream außerdem noch über einige Methoden, die Ihnen als Entwickler die direkte Interaktion mit dem Stream-Objekt erlauben. Die wichtigste Operation, die ein Writable Stream zur Verfügung stellt, ist das Schreiben in den Datenstrom. Zu diesem Zweck können Sie die `write`-Methode verwenden. Sie können diese Methode auf zwei verschiedene Arten benutzen. Die erste Variante besteht darin, dass Sie dieser Methode als Argument eine Zeichenkette übergeben, die Sie als Teil einer Nachricht versenden möchten. Optional können Sie außerdem die Zeichencodierung dieser Zeichenkette als zweites Argument übergeben. Die zweite Art des Aufrufs ist die Verwendung eines Buffer-Objekts, das einen Teil der zu versendenden Nachricht enthält.

Eine Nachricht, die Sie über einen Datenstrom versenden, kann aus mehreren Teilen bestehen. Das bedeutet, Sie können auf einem Stream-Objekt mehrmals die `write`-Methode aufrufen. Jeder dieser Aufrufe führt dazu, dass ein Teil einer Nachricht ver-

sendet wird. Möchten Sie signalisieren, dass die Nachricht komplett übermittelt wurde, erreichen Sie dies durch den Aufruf der `end`-Methode. Diese können Sie auch verwenden, um den letzten Teil Ihrer Nachricht zu versenden. Dazu können Sie die gleichen Werte verwenden wie auch schon bei `write`, also entweder eine Zeichenkette oder ein Buffer-Objekt als Argument. Nach dem Aufruf von `end` ist der Stream nicht länger `writable`, und Sie können die `write`-Methode dieses Objekts nicht mehr aufrufen.

Neben dem ordnungsgemäßen Beenden des Versandvorgangs haben Sie zusätzlich die Möglichkeit, die Verbindung mit sofortiger Wirkung zu zerstören. Mithilfe der `destroy`-Methode wird der dem Stream zugrunde liegende `FileDescriptor` geschlossen. Das bedeutet für Sie, dass der Stream weder les- noch schreibbar ist. Nachrichten, die zum Zeitpunkt des Aufrufs von `destroy` noch nicht versandt wurden, werden gelöscht und dem Empfänger nicht zugestellt. Ein Aufruf der `destroy`-Methode zieht bei den meisten `Writable` Streams ein `close`-Event nach sich, auf das Sie dann eine entsprechende Callback-Funktion binden können, um auf das Beenden der Verbindung zu reagieren.

Ist Ihnen die Wirkung der `destroy`-Methode zu schwerwiegend und möchten Sie vermeiden, dass Teile von Nachrichten, die sich noch im Puffer befinden, gelöscht werden, können Sie die Methode `destroySoon` verwenden. Die Wirkung dieser Methode gleicht der der `destroy`-Methode – außer dem Faktor, dass die durch die `write`-Methode bereits versandten Nachrichtenteile dem Empfänger noch zugestellt werden.

Die `Readable` und `Writable` Streams bilden die Grundlage für mehrere Module innerhalb der Node.js-Plattform. Das `http`-Modul und die Standardein- und -ausgaben von Prozessen sind nur einige Beispiele, in denen Datenströme zum Einsatz kommen. In den folgenden Abschnitten lernen Sie weitere Repräsentanten dieser Kategorie von Datenströmen kennen.

6.5 Sockets

Neben dem `http`-Modul ist das `net`-Modul eines der bedeutendsten Vertreter der Stream-Implementierungen. Mithilfe dieses Moduls können Sie Verbindungen über Datenströme aufbauen. Die Streams sind dabei nicht nur auf reine TCP-Netzwerkverbindungen beschränkt, wie der Name vermuten ließe, sie können auch Verbindungen über UNIX Domain Sockets aufbauen.

6.5.1 TCP

Wie schon beim `http`-Modul können Sie auch beim `net`-Modul sowohl einen Server als auch einen Client instanziiieren. Die Kommunikation findet hier allerdings nicht auf der Ebene der Kommunikationsprotokolle statt, sondern eine Ebene tiefer, und

zwar auf der Transportschicht. Das bedeutet, dass bei der Kommunikation weniger Overhead entsteht, da beispielsweise die HTTP-Header entfallen.

TCP

TCP ist ein Übertragungsprotokoll, das im Netzwerkbereich sehr häufig zum Einsatz kommt. Eines der wichtigsten Merkmale von TCP ist, dass eine zuverlässige Verbindung zwischen Client und Server aufgebaut wird. Zur Adressierung der Gegenstelle wird zum einen die IP-Adresse eines Hosts verwendet, und zum anderen kommt die TCP-Portnummer zum Einsatz. Jedes Protokoll, das auf TCP aufbaut, erhält eine Portnummer, über die die Schnittstelle erreichbar ist. Die verfügbare Spanne von Ports reicht von 0 bis 65535, wobei der Bereich von 0 bis 1023 bereits für bestimmte Protokolle reserviert ist. Alle übrigen Ports können Sie frei verwenden. TCP ist ein paketorientiertes Protokoll, bei dem jedes Paket aus einem Header mit bestimmten Informationen und einem Body mit der Nutzlast des Pakets besteht. Zu Beginn der Kommunikation wird eine Verbindung zwischen Client und Server aufgebaut, über die dann die Pakete in beide Richtungen gesendet werden. Ist die Kommunikation beendet, sollte die Verbindung wieder geschlossen werden, um die daran gebundenen Ressourcen, besonders den TCP-Port, wieder freizugeben. TCP operiert auf der Transportebene des OSI-Modells (<http://de.wikipedia.org/wiki/OSI-Modell>) und basiert damit auf einem Protokoll der Vermittlungsschicht, wie IP. Es bildet die Basis für die Protokolle der darüberliegenden Schichten wie beispielsweise HTTP.

Analog zum `http`-Modul generieren Sie im `net`-Modul die Serverkomponente durch einen Aufruf der `createServer`-Methode. Listing 6.17 zeigt Ihnen ein sehr einfaches Beispiel für einen TCP-Server.

```
var net = require('net');

net.createServer(function (conn) {
  conn.write('Hello Client');
  conn.end();
}).listen(1234);
```

Listing 6.17 TCP-Server

Der Rückgabewert der Methode `createServer` liefert ein Server-Objekt zurück, auf das Sie die `listen`-Methode aufrufen können, um eine Verbindung aufzubauen, die auf eingehende Anfragen wartet. Die Funktion, die die `createServer`-Methode als Argument akzeptiert, wird an das `connection`-Event gebunden. Dieses Event wird bei jeder eingehenden Anfrage ausgelöst. Sie erhalten ein Objekt vom Typ `Socket` als Argument in dieser Callback-Funktion. Das `Socket`-Objekt repräsentiert die Verbindung zwischen den beiden Kommunikationsendpunkten. Sie können es, wie Sie im Bei-

spiel in Listing 6.17 sehen, dazu nutzen, um über die bestehende Verbindung Daten an die Gegenstelle zu senden.

Das Gegenstück zum TCP-Server ist der Client. Server und Client in einer TCP-Verbindung spielen allerdings nicht die Rollen, wie sie es bei HTTP tun, sondern sind eher als gleichberechtigte Kommunikationspartner anzusehen. In Listing 6.18 sehen Sie die Implementierung eines rudimentären TCP-Clients.

```
var net = require('net');

var client = net.connect({port: 1234}, function () {
  console.log('client connected');
});

client.setEncoding('utf8');

client.on('data', function (data) {
  console.log('Server said: ' + data);
});
```

Listing 6.18 TCP-Client

Einen TCP-Client können Sie mit einem Aufruf der `connect`-Methode erstellen. Als erstes Argument akzeptiert diese Methode ein Konfigurationsobjekt, mit dessen Hilfe Sie die Einstellungen der Verbindung angeben können. Für die Verbindung mit dem TCP-Server aus Listing 6.17 ist lediglich die Angabe der Portnummer notwendig, da sowohl Client als auch Server auf dem gleichen Rechner ausgeführt werden und als Standardwert für den Hostnamen `localhost` angenommen wird. Möchten Sie einen anderen Host angeben, können Sie dies über die `host`-Direktive tun. Die Funktion, die die `connect`-Methode als zweites Argument erhält, wird direkt auf das `connect`-Event des Sockets gebunden und ausgeführt, sobald die Verbindung hergestellt wird. Konkret bedeutet das für das Beispiel, dass die Zeichenkette `client connected` auf der Kommandozeile ausgegeben wird, sobald die Verbindung zum Server hergestellt ist.

Ist die Verbindung zwischen dem Client und dem Server hergestellt, wird seitens des Servers einmalig für diese Verbindung das `connection`-Event ausgelöst und die darauf gebundenen Callback-Funktionen ausgeführt. Im Beispiel aus Listing 6.17 wird in dieser Funktion die Zeichenkette `Hello Client` in den Datenstrom geschrieben und die Verbindung beendet. Diese Schreiboperation wiederum verursacht beim Client das `data`-Event, auf das Sie in Listing 6.18 eine Callback-Funktion registrieren, die ihrerseits die Antwort des Servers ausgibt.

Mit dieser einfachen Variante der Client-Server-Kommunikation haben Sie einen ersten Einblick in die Möglichkeiten erhalten, die Ihnen das `net`-Modul hinsichtlich

des Datenaustauschs zwischen zwei Kommunikationspartnern bietet. Im Gegensatz zum Datenaustausch über HTTP sind hier die beiden Endpunkte allerdings gleichberechtigt, und beide können daher sowohl Daten senden als auch empfangen. Um dies zu ermöglichen, erhalten Sie auf Serverseite in der Callback-Funktion der `createServer`-Methode und auf Clientseite als Rückgabewert der `connect`-Methode ein `Stream`-Objekt, über das Sie beide Operationen durchführen können.

Die zweite Variante der Kommunikation mit dem `net`-Modul ist der Informations-transfer über UNIX Domain Sockets, also nicht über das Netzwerk, sondern über das Dateisystem.

6.5.2 UNIX Domain Sockets

Für die Kommunikation über TCP wird eine Netzwerkverbindung benötigt. Auch wenn Client und Server auf dem gleichen Rechner laufen, muss eine Netzwerkschnittstelle vorhanden sein. Diese Notwendigkeit ist bei der Kommunikation über UNIX Domain Sockets nicht gegeben. Statt eines Hostnamens und einer Portnummer geben Sie einen Pfad an, über den die Node.js-Plattform den Socket finden beziehungsweise erstellen kann.

Der Nachteil dieser Kommunikationsform ist natürlich, dass Sie hierbei auf das lokale System beschränkt sind, wohingegen Sie bei TCP-Verbindungen auch zwischen getrennten Systemen kommunizieren können.

Auch diese Variante der Kommunikation über das `net`-Modul sehen Sie anhand eines kurzen Beispiels in Listing 6.19.

```
var net = require('net');

net.createServer(function (conn) {
  conn.write('Hello Client');
  conn.end();
}).listen('/tmp/node.sock');
```

Listing 6.19 UNIX Domain Socket-Server

Der Quellcode in Listing 6.19 gleicht dem Code aus Listing 6.17, bis auf die Tatsache, dass der Server nicht auf einen Host und einen Port, sondern auf einen UNIX Domain Socket gebunden wird. Durch den Aufruf der `listen`-Methode wird der Socket erstellt und kann zur Kommunikation verwendet werden.

Die Gegenseite der Kommunikation, der Client, sieht recht ähnlich aus. Auch hier gleicht der Quellcode wieder dem aus Listing 6.18, bis auf das Konfigurationsobjekt. Aus diesem Grund ist dieses Listing auch nur auf die Verbindung beschränkt, die Callback-Funktion des `data`-Events können Sie aus Listing 6.18 übernehmen.


```

Var net = require('net');
var client = net.connect({path: '/tmp/node.sock'}, function () {
    console.log('client connected');
});

```

Listing 6.20 UNIX Domain Socket-Client

Das Ergebnis des Beispielcodes gleicht dem aus dem TCP Socket-Beispiel. Falls Sie sich nun die Frage stellen, ob wegen des Namens »UNIX Domain Sockets« diese Art der Kommunikation auf UNIX-Systeme beschränkt ist, kann dies mit einem klaren Nein beantwortet werden. Auch unter Windows-Systemen können Sie die Kommunikation über das Dateisystem nutzen. In diesem Fall werden dann nicht UNIX Domain Sockets verwendet, sondern Windows Pipes. Der Unterschied wird vor allem bei der Benennung der Ressource deutlich. Unter UNIX-Systemen mit den UNIX Domain Sockets müssen Sie lediglich einen Pfad zu einer Datei angeben. Zur Verwendung der Windows Pipes müssen Sie eine besondere Namenskonvention beachten. Der Pfad zu einer Pipe besteht aus dem Servernamen, dem Schlüsselwort `pipe` und einem frei wählbaren Namen für die Pipe. Als Trennzeichen wird, wie unter Windows üblich, der Backslash verwendet. Möchten Sie lokal kommunizieren, geben Sie statt eines Servernamens lediglich einen Punkt an. Listing 6.21 zeigt Ihnen anhand der Erstellung eines Socket-Servers, wie Sie eine Pipe unter Windows benennen.

```

var net = require('net');

net.createServer(function (conn) {
    conn.write('Hello Client');
    conn.end();
}).listen('\\\\.\\pipe\\node-pipe');

```

Listing 6.21 Pipe-Server unter Windows

Die Namenskonvention der Pipes unter Windows ist zwar etwas gewöhnungsbedürftig, sorgt allerdings dafür, dass Sie auf allen Betriebssystemen, für die Node.js zur Verfügung steht, auch über das Dateisystem zwischen Applikationen kommunizieren können, wobei hier die Kommunikation nicht bei Node.js endet. Sie können durch Node.js auch mit Applikationen kommunizieren, die in anderen Sprachen wie beispielsweise C, Java oder PHP geschrieben sind. Hier müssen Sie sich lediglich auf eine Art Kommunikationsprotokoll einigen, sodass beide Applikationen die Daten korrekt interpretieren können, die über den Datenstrom versendet werden.

6.6 Zusammenfassung

Im Verlauf dieses Kapitels haben Sie mehr über die Hintergründe der Kommunikationsmöglichkeiten von Node.js erfahren. Ein Großteil der Kommunikation, die an den Schnittstellen einer Node.js-Applikation stattfindet, verläuft über Datenströme. Die Entwickler von Node.js haben zu diesem Zweck mit der `Stream`-Klasse eine standardisierte Schnittstelle geschaffen, die von zahlreichen Modulen der Plattform implementiert wird. Die prominentesten Vertreter sind das `http`-Modul, das Ihnen sowohl HTTP-Server als auch -Clients bereitstellt, die Standardein- und -ausgabe von Prozessen und nicht zuletzt das `net`-Modul, mit dessen Hilfe Sie über TCP und UNIX Domain Sockets mit anderen Servern und Applikationen kommunizieren können. Im nächsten Kapitel beschäftigen Sie sich mit den Möglichkeiten der asynchronen Programmierung und damit, wie Sie rechenintensive Programmteile aus Ihrer Hauptapplikation auslagern können.

Kapitel 7

Asynchrone Programmierung

*Das Denken gehört zu den größten Vergnügungen
der menschlichen Rasse.
– Berthold Brecht*

Bisher waren sämtliche Beispiele, die Sie umgesetzt haben, vor allem von einer Tatsache gekennzeichnet: Der Quellcode wurde in einem Thread ausgeführt. Das hatte zum einen den Vorteil, dass der Code relativ einfach zu lesen war und Sie sich nicht um Nebenläufigkeit kümmern mussten. In diesem Kapitel lernen Sie, wie Sie bestimmte Problemstellungen asynchron lösen und so Applikationen umsetzen können, die die zur Verfügung stehenden Ressourcen optimal ausnutzen.

7.1 Grundlagen asynchroner Programmierung

Viele Konzepte von Node.js setzen auf Asynchronität. Das bedeutet ganz einfach gesagt, dass Berechnungen ausgelagert werden. Der Grund, warum derartige Konzepte erforderlich werden, liegt in der Architektur der Node.js-Plattform. Node.js selbst verfolgt für den Quellcode, den Sie für Ihre Applikation umsetzen, einen Single-Threaded-Ansatz. Das bedeutet, dass dieser Code lediglich in einem Thread läuft. Dauert eine Berechnung länger, kann im Normalfall keine weitere Operation parallel dazu erfolgen.

Erstrebenswert ist es nun, solche potenziell blockierenden Operationen aus dem Haupt-Thread auszulagern. Die Node.js-Plattform stellt Ihnen hierfür mehrere Möglichkeiten in Form von Modulen zur Verfügung, mit deren Hilfe Sie Operationen auslagern können.

Was sich in der Theorie nach einem abstrakten Konzept anhört, kommt in vielen Modulen von Node.js zum Einsatz. Konkret können Sie asynchrone Operationen beispielsweise im `fs`-Modul beobachten. Listing 7.1 zeigt Ihnen, wie das asynchrone Auslesen von Dateien funktioniert.

```
var fs = require('fs');

fs.readFile('/tmp/input.txt', 'utf-8', function (err, data) {
  console.log(data);
});
```

Listing 7.1 Asynchrones Auslesen von Dateien

Im Beispiel wird der Inhalt der Datei ausgelesen, und die Informationen werden anschließend ausgegeben. Für ein einziges Mal Auslesen einer kleinen Textdatei stellt diese Art der Operation kein wirkliches Problem dar. Sollen allerdings sehr viele kleine Dateien oder wenige große Dateien ausgelesen werden, nehmen diese Operationen in Summe unter Umständen viel Zeit in Anspruch. Verarbeiten Sie diese Informationen abseits von Benutzern oder anderen Systemen, also in einer zeitunkritischen Umgebung, können Sie die Dateien auch direkt in Ihrer Applikation im Haupt-Thread durchführen. Kommen jedoch Benutzer ins Spiel, die parallel mit Ihrer Applikation arbeiten, kommt es durch die parallelen Anfragen zu Verzögerungen, da die Anfragen nur sequenziell abgearbeitet werden können.

Die asynchrone Abarbeitung von Aufgaben setzt auf die eventgetriebene Infrastruktur von Node.js. Im Beispiel in Listing 7.1, in dem Sie eine Datei auslesen, geben Sie mit dem Aufruf der Methode `readFile` den Auftrag, die Datei auszulesen, die Sie als erstes Argument angeben. Das zweite Argument bezeichnet den zu verwendenden Zeichensatz. Das dritte Argument schließlich stellt die Callback-Funktion dar, die ausgeführt werden soll, sobald der Inhalt der Datei vorliegt.

Intern wird die Anfrage an das Betriebssystem weitergegeben. Für den Fall, dass die Antwort zurückgeliefert wird, wird die Callback-Funktion registriert. Das Betriebssystem kümmert sich ab diesem Zeitpunkt um die Abarbeitung des Auftrags. Das bedeutet für die Logik Ihrer Applikation, dass das nächste Kommando nach dem Aufruf der `readFile`-Methode ausgeführt wird und so die Abarbeitung der Applikation nicht durch die Leseoperation blockiert ist. Ist die Leseoperation schließlich beendet, wird das Ergebnis an Node.js zurückgegeben. Die Node.js-Plattform kümmert sich dann darum, dass die zuvor registrierte Callback-Funktion aufgerufen wird. Damit die Callback-Funktion ausgeführt werden kann, muss die Abarbeitung der Applikation kurz unterbrochen werden.

Grundsätzlich sollten Sie sämtliche rechenintensive Operationen aus dem Haupt-Thread auslagern. Da dies allerdings bei Applikationscode, den Sie als Entwickler verfasst haben, nicht so ohne Weiteres möglich ist, beschränkt sich die Auslagerung auf die Basisfunktionalität von Node.js. Neben den bereits erwähnten Dateisystemoperationen sind asynchrone Methoden auch in zahlreichen anderen Modulen, wie beispielsweise im `http`-Modul, anzutreffen. Wenn Sie hier einen Server erstellen, wartet dieser auf Anfragen von Clients. Sobald eine Anfrage eintrifft, wird sie bearbeitet, und der Server fällt zurück in Wartestellung.

Ein Merkmal der asynchronen Bearbeitung ist die Registrierung von Callback-Funktionen. Der Unterschied hier wird vor allem deutlich, wenn Sie die Methoden im `fs`-Modul betrachten. Hier existiert für fast jede asynchrone Methode ein synchrones Gegenstück. So gibt es die `readFile`-Methode als `readFileSync` auch in ihrer synchronen Form. Beide Methoden unterscheiden sich lediglich in der Anzahl der verfügbaren Argumente und in ihrem Rückgabewert. Die synchrone Version der Methode erhält ein Argument weniger. Sie benötigen in diesem Fall keine Callback-Funktion, da die Methode die Ergebnisse direkt zurückgibt. Die asynchrone `readFile`-Methode gibt `undefined` als Wert zurück. Die Bearbeitung der zurückgelieferten Inhalte erfolgt, sobald die Ergebnisse vorliegen, innerhalb der Callback-Funktion.

Tritt ein Fehler bei der Abarbeitung der Methode auf, also existiert beispielsweise die Datei, die `readFile` auslesen soll, nicht, erhalten Sie als erstes Argument der Callback-Funktion ein Objekt, das die entsprechende Fehlermeldung enthält. Wenn Sie dieses Objekt ignorieren, wird Ihre Applikation trotzdem weiter ausgeführt. Anders ist der Fall bei der synchronen Version der Methode. Hier steht Ihnen lediglich ein Rückgabewert zur Verfügung, die Fehlermeldung kann also nicht in ein separates Objekt zusätzlich zu den zurückgelieferten Daten gekapselt werden. Kann die Datei, die mit `readFileSync` ausgelesen werden soll, nicht gefunden werden, wirft die Methode eine `Exception`, die Sie mit einem `try-catch`-Block behandeln müssen, da ansonsten die Ausführung Ihrer gesamten Applikation unterbrochen wird.

Damit Sie auch innerhalb Ihrer Applikation nicht auf asynchrone Programmierung verzichten müssen, stellt Ihnen die Node.js-Plattform mit dem `child_process`-Modul mehrere Möglichkeiten zur Verfügung, mit denen Sie Teile Ihrer Applikation aus dem Haupt-Thread auslagern können. Dabei entstehen allerdings auch einige Probleme, wie beispielsweise die parallele Abarbeitung zweier Logikstränge, die unter Umständen miteinander synchronisiert werden müssen.

7.1.1 Das `child_process`-Modul

Das `child_process`-Modul fasst eine Reihe von Methoden zur asynchronen Programmierung von Applikationen für Sie zusammen. Hier erfahren Sie zunächst mehr über den grundsätzlichen Aufbau des Moduls und der darin enthaltenen Klasse `ChildProcess`.

Jeder Kindprozess hat, wie der Elternprozess, also Ihre eigentliche Applikation, drei Streams: eine Standardeingabe, eine Standardausgabe und die Standardfehlerausgabe. Der Kindprozess kann alle drei Streams gemeinsam mit dem Elternprozess nutzen. Sie können allerdings auch Stream-Objekte erzeugen und dem Kindprozess zuweisen. Hierfür stehen Ihnen die drei Eigenschaften `stdin`, `stdout` und `stderr` zur Verfügung. In Kapitel 6 erfahren Sie mehr über Datenströme und wie Sie diese zur Kommunikation einsetzen können.

Das Objekt, das den Kindprozess repräsentiert, erhalten Sie als Rückgabewert der Funktionen `spawn`, `exec` und `fork`. Diese können Sie nutzen, um einen Kindprozess zu starten. In den folgenden Abschnitten erfahren Sie mehr über die einzelnen Methoden und ihr Einsatzgebiet. Hier sei nur erwähnt, dass sie als Rückgabe die Repräsentation des Kindprozesses liefern, die Sie innerhalb Ihrer Applikation dazu nutzen können, um mit dem Kindprozess zu interagieren.

Eine Operation, die Ihnen für die Kindprozesse zur Verfügung steht, ist die Beendigung dieser Prozesse vom Elternprozess, aber auch vom Kindprozess aus. Hierfür gibt es die beiden Methoden `kill` und `disconnect`.

Mithilfe der `kill`-Methode können Sie ein Signal an den Prozess senden. Der Name dieser Methode ist etwas irreführend, da Sie den Prozess nicht zwingend beenden müssen. Stattdessen können Sie diese Methode auch zur Interprozesskommunikation nutzen. Interprozesskommunikation bedeutet in diesem Fall, dass Sie zwischen zwei parallel laufenden Prozessen oder Threads Informationen austauschen. Gültige Werte für diese Methode sind Namen gültiger Signale, wie sie auch für das `kill`-Kommando auf UNIX-Systemen verwendet werden. So können Sie beispielsweise `SIGKILL` oder `SIGTERM` zum Beenden des Prozesses verwenden. Geben Sie beim Aufruf der Methode keinen Wert mit, wird standardmäßig `SIGTERM` verwendet. Dieses Signal dient im Normalfall dazu, einen Prozess zu beenden, wobei dieser die Gelegenheit erhält, vor Beendigung verschiedene Aufräumarbeiten durchzuführen. In Node.js wird diese Art der Interprozesskommunikation über eine Eventarchitektur gelöst. Rufen Sie auf dem Kindprozess die `kill`-Methode auf, wird im Kindprozess ein `exit`-Ereignis erzeugt, auf das Sie eine Callback-Funktion registrieren können. Das erste Argument dieser Callback-Funktion ist der Rückgabewert des Kindprozesses, falls dieser normal beendet wurde. Wurde der Prozess durch ein Signal beendet, weist dieses Argument den Wert `null` auf. Das zweite Argument ist das Signal, das an den Kindprozess gesendet wurde, also beispielsweise die Zeichenkette `SIGTERM`.

Die zweite Methode `disconnect` dient dazu, die Kommunikationsverbindung zwischen Eltern- und Kindprozess zu beenden. Konkret bedeutet das für Sie, dass Sie keine Nachrichten mehr zwischen dem Eltern- und Kindprozess austauschen können. Der Aufruf dieser Methode verursacht das `disconnect`-Ereignis im Kindprozess, auf das Sie sich wiederum mit einer Callback-Funktion binden können, um entsprechend reagieren zu können. Neben dem Auslösen des Ereignisses wird außerdem die Eigenschaft `connected` vom Wert `true` auf `false` gesetzt.

Neben den bereits vorgestellten Events existiert auch noch das `close`-Event. Dieses stellt insofern eine Besonderheit dar, als dass es ausgelöst wird, sobald das Stream-Objekt, das die Standardeingabe des Kindprozesses repräsentiert, durch die Methode `end` beendet wird. Auch auf dieses Event können Sie wieder eine Callback-Funktion binden.

Die übrigen Methoden zur Kommunikation zwischen Eltern- und Kindprozess lernen Sie im Zusammenhang mit der `fork`-Methode am konkreten Beispiel kennen.

Prozesse und Threads

Ein Prozess ist die Repräsentation eines Programms in einem Betriebssystem. Ein Prozess belegt dabei einen Bereich des Arbeitsspeichers, der ihm im Normalfall exklusiv zur Verfügung steht. Außerdem erhält ein Prozess Rechenzeit des Prozessors, um Berechnungen durchzuführen. Weitere Informationen zu Prozessen finden Sie unter [http://de.wikipedia.org/wiki/Prozess_\(Informatik\)](http://de.wikipedia.org/wiki/Prozess_(Informatik)).

Ein Thread hingegen ist ein Teil eines Prozesses und bezeichnet einen Ausführungsstrang innerhalb eines Programms. Das bedeutet konkret, dass ein Prozess aus mehreren Threads bestehen kann. Ein Thread hat Zugriff auf die Ressourcen des Prozesses, dem er zugeordnet ist. Jeder Thread eines Prozesses besitzt einen eigenen Stack, das ist ein Speicher, in den der Thread Elemente speichern und wieder auslesen kann. Dieser Stack ist exklusiv für einen Thread. Die übrigen Threads des Prozesses können nicht darauf zugreifen. Mehr zum Thema Threads finden Sie unter [http://de.wikipedia.org/wiki/Thread_\(Informatik\)](http://de.wikipedia.org/wiki/Thread_(Informatik)).

Führen Sie Node.js aus, wird ein Prozess gestartet. Innerhalb dieses Prozesses läuft ein einziger Thread, da die Node.js-Plattform, wie Sie bereits in Kapitel 1, »Grundlagen«, erfahren haben, einen Single-Threaded-Ansatz verfolgt.

7.2 Externe Kommandos asynchron ausführen

Mit den Methoden `spawn`, `exec` und `execFile`, die Ihnen Node.js bietet, können Sie Kommandos unabhängig von der Plattform ausführen. Das bedeutet, dass Sie über diese Methoden sämtliche Befehle, die Ihnen auf der Kommandozeile Ihres Betriebssystems zur Verfügung stehen, auch hier verwenden können.

Diese Art der Befehlsausführung hat den Vorteil, dass Sie direkt auf die Funktionalität des Betriebssystems beziehungsweise auf Dienstprogramme zugreifen können und dass diese Funktionalität nicht erst mühsam in JavaScript umgesetzt werden muss. Damit Sie mit diesen Kommandos arbeiten können, können Sie das Verhalten jeweils über Optionen steuern und erhalten Zugriff auf die Ausgaben der jeweils ausgeführten Befehle.

7.2.1 Die `exec`-Methode

Die einfachere der beiden Varianten zur Ausführung externer Kommandos stellen die beiden `exec`-Befehle dar. In Listing 7.2 sehen Sie ein sehr einfaches Beispiel der Ausführung der `exec`-Methode.


```
var exec = require('child_process').exec,
    cmd = 'find /usr/local -iname "node"';

exec(cmd, function (err, stdout, stderr) {
    if (err) throw err;

    console.log(stdout);
});
```

Listing 7.2 Ausführung eines externen Kommandos mit `exec` auf einem UNIX-System

Im ersten Schritt benötigen Sie die `exec`-Methode, die Sie mithilfe der `require`-Methode erhalten. Als Nächstes formulieren Sie das Kommando, das Sie absetzen möchten, im Falle des Beispiels ist dies die Suche nach Dateien und Verzeichnissen mit dem Namen `node` unterhalb des Verzeichnisses `/usr/local`. Diese Zeichenkette dient als erstes Argument für den Aufruf der `exec`-Methode. Das zweite Argument ist eine Callback-Funktion, die ausgeführt werden soll, sobald das Ergebnis des Kommandos vorliegt. Die Argumente, die Ihnen innerhalb der Callback-Funktion zur Verfügung stehen, sind ein Fehler-Objekt, das einen Wert besitzt, falls bei der Ausführung des Kommandos ein Fehler aufgetreten ist, als zweites Argument ein Buffer-Objekt, das die Standardausgabe des Kommandos enthält, und als drittes Argument schließlich ein weiteres Buffer-Objekt, das die Werte der Standardfehlerausgabe beinhaltet.

Das angegebene Kommando wird durch den Aufruf von `exec` in einer Subshell, also auf einer separaten Kommandozeile, ausgeführt, seine Ausgabe gepuffert und das Ergebnis schließlich im gerade beschriebenen Buffer-Objekt der obigen Callback-Funktion zur Verfügung gestellt. Die Ausführung erfolgt asynchron, das bedeutet, dass Ihre Applikation durch die Abarbeitung des externen Kommandos nicht blockiert wird.

Durch die Angabe verschiedener Optionen als zweites Argument beim Aufruf der `exec`-Methode können Sie deren Ausführung beeinflussen. Die einzelnen Optionen, deren Bedeutung und ihre Standardwerte finden Sie in Tabelle 7.1.

Option	Standardwert	Bedeutung
<code>cwd</code>	<code>null</code>	aktuelles Arbeitsverzeichnis für den Kindprozess.
<code>customFds</code>	<code>undefined</code>	Dateideskriptoren für Ein- und Ausgaben. Diese Option sollte nicht mehr verwendet werden.
<code>encoding</code>	<code>utf8</code>	Zeichencodierung für die Ausgabe

Tabelle 7.1 Optionen zur Konfiguration der `exec`-Methode

Option	Standardwert	Bedeutung
env	null	Umgebungsvariablen für den Kindprozess als Schlüssel-Wert-Paare
killSignal	SIGTERM	Signal, mit dessen Hilfe ein Prozess, der zu lange läuft, beendet wird.
maxBuffer	204800	Maximale Größe für Standardausgabe und Standardfehlerausgabe. Wird diese Größe überschritten, wird der Kindprozess beendet.
stdio	undefined	Konfiguration für die Ein- und Ausgaben des Kindprozesses
timeout	0	Anzahl an Millisekunden, nach deren Ablauf der Kindprozess zwangsweise mit dem unter kill-Signal angegebenen Signal beendet wird.

Tabelle 7.1 Optionen zur Konfiguration der exec-Methode (Forts.)

Neben der exec-Methode steht Ihnen auch die etwas leichtgewichtigere execFile-Methode zur Verfügung. Mit dieser Methode können Sie die Inhalte einer Datei direkt ausführen. Listing 7.3 zeigt Ihnen den Inhalt einer derartigen ausführbaren Datei.

```
#!/usr/local/bin/node
console.log('Hello World');
```

Listing 7.3 Ausführbare Datei als Eingabe für execFile

Speichern Sie diesen Quellcode als input.sh im gleichen Verzeichnis wie Ihre Node.js-Applikation, dann können Sie den Quellcode in Listing 7.4 ausführen und erhalten die Ausgabe Hello World auf der Kommandozeile.

```
var execFile = require('child_process').execFile;

execFile('./input.sh', function (err, stdout, stderr) {
    console.log(stdout);
});
```

Listing 7.4 Ausführung einer Datei mit execFile

In der Eingabedatei aus Listing 7.3 sehen Sie, dass Sie mit dieser Methode auch Java-Script-Code ausführen können. Alles, was Sie hierfür tun müssen, ist, in der ersten Zeile der Eingabedatei anzugeben, mit welcher Applikation die Datei ausgeführt wer-

den soll. In diesem speziellen Fall ist dies `/usr/local/bin/node`, was Sie auf einem UNIX-System mit dem Kommando `which node` herausfinden können. Sie können allerdings auch ganz normale Shell-Kommandos wie beispielsweise `find` ausführen. Hier müssen Sie die Datei auch nicht zwingend mit `#!/bin/bash` beginnen, sondern lediglich die Kommandos angeben, die der Reihe nach ausgeführt werden sollen.

Die eigentliche Applikation, deren Quellcode Sie in Listing 7.4 finden, sorgt lediglich dafür, dass die Datei ausgeführt wird und die daraus entstehenden Resultate auf der Standardausgabe ausgegeben werden.

Die zweite Möglichkeit, die Ihnen zur Ausführung externer Kommandos zur Verfügung steht, ist die `spawn`-Methode. Über diese erfahren Sie im folgenden Abschnitt mehr.

7.2.2 Die `spawn`-Methode

Im Gegensatz zu den beiden `exec`-Methoden arbeitet `spawn` nicht mit einem Puffer der Ausgabe des Kommandos. Stattdessen stehen sowohl die Ein- als auch die Ausgaben des Prozesses als Datenströme zur Verfügung. Das bedeutet, die Daten, auf die Sie hier zugreifen können, sind kein einmaliges Abbild, sondern ein kontinuierlicher Strom von Daten. Das hat zum einen den Vorteil, dass Sie bei der Verwendung von `spawn` nicht auf eine maximale Größe der Ausgabe beschränkt sind, und zum anderen erhalten Sie Zugriff auf die Daten zum Zeitpunkt ihrer Entstehung und nicht erst, wenn das Kommando beendet ist.

In Listing 7.5 sehen Sie eine einfache Variante des Einsatzes der `spawn`-Methode.

```
var spawn = require('child_process').spawn,
    find = spawn('find', ['. ', '-iname', 'node'], {cwd: '/usr/local'});

find.stdout.on('data', function (data) {
    console.log(data.toString());
});
```

Listing 7.5 Die `spawn`-Methode im Einsatz

Wie schon im Beispiel für die `exec`-Methode wird auch in Listing 7.5 wieder nach Dateien und Verzeichnissen mit dem Namen `node` gesucht. Die Unterschiede zwischen `spawn` und `exec` werden bereits in der Angabe des Kommandos selbst sichtbar. Wo Sie bei `exec` das Kommando und sämtliche zugehörigen Optionen als eine Zeichenkette zur Verfügung gestellt haben, übergeben Sie mit dem zweiten Argument von `spawn` ein Array von Optionen für das Kommando. Wie schon bei `exec` können Sie auch bei `spawn` die Methode durch weitere Optionen, die Sie im dritten Argument angeben, zusätzlich beeinflussen. Im Beispiel in Listing 7.5 ist dies das aktuelle

Arbeitsverzeichnis des Prozesses, das Sie über die Option `cwd` spezifizieren. Alle weiteren möglichen Optionen können Sie Tabelle 7.2 entnehmen. Bitte beachten Sie hier, dass sich die Optionen erheblich von denen unterscheiden, die im Rahmen der `exec`-Methode verfügbar sind.

Option	Bedeutung
<code>customFds</code>	Mit dieser mittlerweile veralteten Option konnten die Dateideskriptoren für die Ein- und Ausgaben definiert werden.
<code>cwd</code>	aktuelles Arbeitsverzeichnis für den Prozess
<code>detached</code>	Wird diese Option auf <code>true</code> gesetzt, wird der Kindprozess unabhängig vom Elternprozess. Das bedeutet, dass der Kindprozess nicht zwangsläufig beendet wird, wenn der Elternprozess beendet wird.
<code>env</code>	Mit dieser Option können Sie die Werte von Umgebungsvariablen als Schlüssel-Wert-Paare angeben.
<code>gid</code>	Gruppen-ID, unter der der Kindprozess ausgeführt werden soll. Beachten Sie, dass Sie die entsprechenden Berechtigungen für diese Operation besitzen müssen.
<code>stdio</code>	Konfigurationsoptionen für die Ein- und Ausgaben
<code>uid</code>	ID des Benutzers, unter dem der Kindprozess ausgeführt werden soll. Auch hier benötigen Sie, wie schon bei <code>gid</code> , die entsprechenden Berechtigungen.

Tabelle 7.2 Optionen für die `spawn`-Methode

Da Sie sowohl auf die Standardeingabe als auch auf die Standardausgabe Zugriff haben, können Sie auch zwei Befehle miteinander kombinieren. Dies bewerkstelligen Sie auf der UNIX-Kommandozeile mithilfe der Pipe. Das Beispiel in Listing 7.6 soll den gleichen Effekt haben wie die Kommandozeile `find /usr/local -iname 'node' | grep 'bin'`. Das bedeutet, Sie suchen nach Dateien und Verzeichnissen mit dem Namen `node` im Verzeichnis `/usr/local` und filtern die gefundenen Treffer dann noch auf das Vorhandensein der Zeichenkette `bin`.

```
var spawn = require('child_process').spawn,
    find = spawn('find', ['.', '-iname', 'node'], {'cwd': '/usr/local'}),
    grep = spawn('grep', ['bin']);

find.stdout.on('data', function (data) {
    grep.stdin.write(data);
});
```

```

find.on('exit', function (code) {
  grep.stdin.end();
});

grep.stdout.on('data', function (data) {
  console.log('' + data);
});

```

Listing 7.6 Zusammensetzen von zwei Kommandos mit spawn

Für jedes Kommando in der Kette müssen Sie die `spawn`-Methode aufrufen. Sie erhalten ein Objekt vom Typ `ChildProcess` als Rückgabewert, mit dem Sie im Folgenden weiterarbeiten werden. Im ersten Codeblock nach der Initialisierung binden Sie eine Callback-Funktion an das `data`-Event der Standardausgabe des `find`-Prozesses. Diese wird aufgerufen, sobald dieser Prozess Daten auf die Standardausgabe schreibt. Diese Daten dienen dann als Eingabe für den zweiten Prozess `grep`. Dies erreichen Sie, indem Sie das Argument, das Sie in der Callback-Funktion erhalten, mithilfe der `write`-Methode wiederum als Eingabe in die Standardeingabe des `grep`-Prozesses schreiben. Sobald die Ausgabe des `find`-Prozesses beendet ist, wird das `exit`-Event ausgelöst. In der Callback-Funktion dieses Events sollten Sie die Standardeingabe des `grep`-Prozesses beenden, indem Sie die `end`-Methode auf die Standardeingabe dieses Prozesses aufrufen. Schließlich können Sie, wie Sie es bereits beim `find`-Prozess gesehen haben, eine Callback-Funktion auf das `data`-Event der Standardausgabe des `grep`-Prozesses binden, innerhalb derer Sie schließlich die gewonnenen Informationen dem Benutzer auf der Standardausgabe präsentieren können.

Wie Sie gesehen haben, können Sie sowohl mit `spawn` als auch mit `exec` Kommandos ausführen, wie Sie sie auch auf der Kommandozeile Ihres Systems absetzen würden. Dabei können Sie Kommandos des Betriebssystems direkt nutzen oder eigens erstellte Shellscripte ausführen. Ein solches Shellscript kann, wie Sie wissen, auch JavaScript-Code enthalten. Bei dieser Variante der Ausführung haben Sie jedoch kaum Möglichkeiten, um zwischen den Prozessen zu kommunizieren. Sie können lediglich recht umständlich die Standardeingabe und die Standardausgabe als Kommunikationsmedium verwenden. Eine bessere Art, um JavaScript-Code in einem Kindprozess auszuführen, bietet Ihnen die Methode `fork`, die Sie im folgenden Abschnitt kennenlernen werden.

7.3 Fork

In Ihrer Node.js-Applikation möchten Sie wahrscheinlich nicht nur externe Programme ausführen, sondern vor allem auch Teile der Applikationslogik in eigene

Threads auslagern. Gerade Codefragmente, deren Abarbeitung sehr zeitintensiv ist, eignen sich gut für derartige Maßnahmen. Mit dieser Vorgehensweise verfolgen Sie den gleichen Ansatz, den die Entwickler für die gesamte Node.js-Plattform vorgesehen haben. Konkret bedeutet das, dass Sie rechenintensive Funktionen auslagern, um die Abarbeitung Ihrer Applikation nicht zu blockieren. Sobald das Ergebnis der Funktion vorliegt, arbeitet Ihre Applikation mit diesem Ergebnis innerhalb einer Callback-Funktion weiter.

Damit Sie Funktionalität aus dem Haupt-Thread Ihrer Applikation auslagern können, bietet Ihnen die Node.js-Plattform die Methode `fork` aus dem `child_process`-Modul. Der Aufruf der `fork`-Methode ähnelt dem der `spawn`- und `exec`-Methoden, wobei das erste Argument der Name des Moduls ist, das ausgeführt werden soll. Das Modul bezeichnet die Datei, die den JavaScript-Quellcode enthält, der die Grundlage des Kindprozesses bildet. Das zweite Argument enthält die Werte, die dem Kindprozess übergeben werden sollen, und das dritte Argument ist schließlich ein Konfigurationsobjekt. Im Konfigurationsobjekt stehen Ihnen wie auch bei den übrigen Methoden des `child_process`-Moduls verschiedene Möglichkeiten zur Verfügung, mit deren Hilfe Sie das Verhalten des Kindprozesses beeinflussen können. Tabelle 7.3 listet Ihnen die verfügbaren Optionen auf, auf die Sie beim Aufruf zurückgreifen können.

Option	Bedeutung
<code>cwd</code>	aktuelles Arbeitsverzeichnis des Kindprozesses
<code>encoding</code>	Zeichencodierung, die innerhalb des Kindprozesses verwendet werden soll. Standardwert ist hier <code>utf8</code> .
<code>env</code>	Umgebungsvariablen für den Kindprozess in Form von Schlüssel-Wert-Paaren
<code>silent</code>	Wird dieser Wert auf <code>true</code> gesetzt, werden die Standardausgaben des Elternprozesses nicht verwendet.

Tabelle 7.3 Optionen für die `fork`-Methode

Der Quellcode in Listing 7.7 bildet die Basis für die weiteren Beispiele im Zusammenhang mit der `fork`-Methode.

```
var fork = require('child_process').fork;

if (process.argv[2] && process.argv[2] === 'child') {
  console.log('Child Process');
  process.exit();
} else {
```

```

    console.log('Parent Process');
    fork(process.argv[1], ['child']);
  }

```

Listing 7.7 Beispielanwendung mit der fork-Methode

Die Ausgabe der Beispielapplikation ist zuerst die Zeichenkette `Parent Process` und danach `Child Process`. Mit dieser Art von Applikation können Sie den Quellcode des Elternprozesses gemeinsam mit dem Code des Kindprozesses in einer einzigen Datei verwalten. Dabei nutzen Sie sowohl die Möglichkeit, dem Kindprozess über die `fork`-Methode zusätzliche Argumente mitzugeben, als auch über `process.argv` auf diese Argumente zuzugreifen. In diesem Beispiel übergeben Sie ein Argument mit dem Namen `child` an den Kindprozess. So können Sie nun über die Eigenschaft `process.argv[2]` prüfen, ob diese existiert und den Wert `child` enthält, und so feststellen, ob Sie sich bei der aktuellen Ausführung des Quellcodes im Eltern- oder im Kindprozess befinden. Im Produktivbetrieb sollten Sie jedoch die Codeblöcke für den Eltern- und für den Kindprozess in separate Dateien auslagern. Im Beispielcode sehen Sie außerdem eine weitere Besonderheit: Im Aufruf der `fork`-Methode geben Sie normalerweise als erstes Argument den Namen der Datei an, die im Kindprozess ausgeführt werden soll. Im Beispiel verwenden Sie mit der Eigenschaft `process.argv[1]` den Namen der aktuellen Datei und sorgen so dafür, dass die gleiche Datei auch im Kindprozess geladen wird.

Sie sollten außerdem beachten, dass Sie im Kindprozess die Methode `exit` des `process`-Objekts aufrufen müssen, um den Kindprozess manuell zu beenden. Wird der Kindprozess nicht beendet, wird auch der Elternprozess nicht beendet, und die Applikation läuft weiter. In späteren Versionen der Node.js-Plattform soll dieses Verhalten allerdings so angepasst werden, dass Sie den Prozess nicht mehr manuell beenden müssen.

Der Aufruf von `fork` führt dazu, dass ein neuer Node.js-Prozess gestartet wird. Die als Argument im Aufruf angegebene Datei wird geladen und als JavaScript-Code interpretiert. Da jede Verwendung von `fork` dazu führt, dass ein neuer Prozess erstellt wird, sollten Sie nicht unbegrenzt viele Kindprozesse erstellen, da diese auch eine gewisse Menge an Ressourcen beanspruchen. Konkret sind dies vor allem Arbeitsspeicher und etwas Prozessorleistung, je nachdem, welche Operationen Sie ausführen lassen.

Bisher macht das Beispiel aus Listing 7.7 noch recht wenig Sinn, da der Kindprozess keinerlei Logik aufweist und auch Eltern- und Kindprozess noch nicht miteinander kommunizieren. In Listing 7.8 sehen Sie, wie Sie den Code dieses Beispiels dahingehend erweitern müssen, um die `fork`-Methode in Aktion erleben zu können.

```

var fork = require('child_process').fork;

if (process.argv[2] && process.argv[2] === 'child') {
  var i, n = 1, results = 1;
  outerLoop: while (results <= 1000) {
    n += 1;
    for (i = 2; i <= Math.sqrt(n); i += 1) {
      if (n % i === 0) {
        continue outerLoop;
      }
    }
    process.send({"prime": n});
    results += 1;
  }
  process.exit();
} else {
  var child1 = fork(process.argv[1], ['child']);
  var child2 = fork(process.argv[1], ['child']);
  var child3 = fork(process.argv[1], ['child']);

  child1.on('message', function (data) {
    console.log('child1: ' + data.prime);
  });
  child2.on('message', function (data) {
    console.log('child2: ' + data.prime);
  });
  child3.on('message', function (data) {
    console.log('child3: ' + data.prime);
  });
}

```

Listing 7.8 Erweitertes fork-Beispiel

Der Quellcode in Listing 7.8 sorgt dafür, dass in drei voneinander unabhängigen Kindprozessen jeweils 1.000 Primzahlen berechnet und an den Elternprozess zurückgeliefert werden, wo dieser für die Ausgabe sorgt.

Um dieses Ergebnis zu erreichen, starten Sie im ersten Schritt drei Kindprozesse über jeweils einen Aufruf der `fork`-Methode, wie Sie es bereits aus Listing 7.7 kennen. Die Logik für den Kindprozess finden Sie im ersten Teil des `if`-Statements. Hierbei wird durch eine Schleife geprüft, ob es sich bei einer bestimmten Zahl um eine Primzahl handelt. Dieser Vorgang wird so lange durchlaufen, bis die erforderliche Anzahl an Primzahlen gefunden ist. Die Rückmeldung an den Elternprozess erfolgt über die

send-Methode des `process`-Objekts. Die Nachricht an den Elternprozess, die Sie dieser Methode als erstes Argument übergeben, liegt in Form eines Objekts vor.

Auf der Seite des Elternprozesses gehen Sie mit den Nachrichten aus den Kindprozessen so um, dass Sie pro instanziierten Kindprozess eine Callback-Funktion auf das `message`-Event registrieren. Damit Sie bei der Ausgabe besser unterscheiden können, von welchem Kindprozess die jeweiligen Primzahlen stammen, sollten Sie bei der Ausgabe ein Identifikationsmerkmal wie beispielsweise `child1` für den ersten Kindprozess voranstellen.

Die Kommunikation zwischen Eltern- und Kindprozess ist allerdings nicht begrenzt auf die einseitige Kommunikation vom Kindprozess in Richtung des Elternprozesses. Auch eine Datenübermittlung in die andere Richtung ist möglich. In Listing 7.9 sehen Sie, wie Sie Nachrichten vom Elternprozess in Richtung des Kindprozesses senden können.

```
var fork = require('child_process').fork;

if (process.argv[2] && process.argv[2] === 'child') {
  process.on('message', function (data) {
    console.log('in child process: ' + data);
  });
} else {
  var child = fork(process.argv[1], ['child']);

  for (var i = 0; i < 10; i += 1) {
    child.send(i);
  }
}
```

Listing 7.9 Kommunikation vom Elternprozess in Richtung des Kindprozesses

Auf dem Objekt, das Sie von der `fork`-Methode als Rückgabewert erhalten, können Sie die `send`-Methode aufrufen. Diese Methode wirkt dabei genauso wie auch im Kindprozess. Sie übergeben die Nachricht, die Sie an den Kindprozess senden möchten, als Argument an die `send`-Methode. Das Format der Nachricht können Sie beliebig wählen. So ist es möglich, sowohl eine Zeichenkette als auch ein komplex strukturiertes Objekt zu übergeben.

Auf der Seite des Kindprozesses löst der Aufruf der `send`-Methode ein `message`-Event aus. Über die `on`-Methode des `process`-Objekts können Sie eine Callback-Funktion auf dieses Event registrieren, innerhalb derer Sie entsprechend auf das Ereignis und auf die gesendete Nachricht reagieren können. Die Daten der Nachricht stehen Ihnen als Argument in der Callback-Funktion zur Verfügung.

7.4 Das cluster-Modul

Grundsätzlich laufen Applikationen unter Node.js in einem Thread. Mit den Methoden des `child_process`-Moduls sind Sie allerdings in der Lage, Workerprozesse auszuführen und Teile Ihrer Applikation aus dem Haupt-Thread auszulagern. Einen ähnlichen Ansatz verfolgt das `cluster`-Modul der Node.js-Plattform. Es baut auf der Funktionalität des `child_process`-Moduls auf, um Workerprozesse zu erstellen. Mit diesem Modul wird eine Art Loadbalancing für Node.js-Applikationen ermöglicht. Sie können damit auf Systemen mit mehreren Prozessorkernen mehrere Server vom Typ HTTP-Server oder TCP-Server starten. Diese laufen in separaten Threads und erlauben so eine bessere Ausnutzung der verfügbaren Ressourcen.

Den Einstiegspunkt in diesen sogenannten Thread-Pool bildet der Masterprozess. Dieser Prozess sorgt für die Erstellung und Steuerung der Workerprozesse. Auch die Handler für die Anfragen werden über den Masterprozess verwaltet. Wird in einem Workerprozess die `listen`-Methode aufgerufen, werden die Argumente an den Masterprozess weitergeleitet. Dieser prüft, ob bereits ein Handle mit dieser Konfiguration existiert. Ist dies der Fall, wird das Objekt, das diesen Handle repräsentiert, an den Workerprozess zurückgegeben. Existiert noch kein Handle, erstellt ihn der Masterprozess und gibt das neu erstellte Objekt an den Workerprozess zurück.

Sobald Anfragen eingehen, werden diese an den entsprechenden Thread weitergeleitet. Gibt es mehrere Threads die auf die angefragte Kombination von Adresse und Port gebunden sind, sorgt das Betriebssystem für die Verteilung der Anfragen auf die entsprechenden Threads. In diesem Fall übernimmt das Betriebssystem das Loadbalancing. Das bedeutet, dass Sie sich nicht um diese Aufgabe kümmern müssen und auch die Node.js-Plattform nicht dafür verantwortlich ist.

Die Workerprozesse sind nicht nur unabhängig voneinander, sondern wissen auch nichts voneinander. Sie haben keinerlei Möglichkeit, untereinander zu kommunizieren. Die Prozesse teilen auch keinen gemeinsamen Speicherbereich. Eine Kommunikation kann nur über den Masterprozess erfolgen. Dieser hält Referenzen auf die einzelnen Workerprozesse.

7.4.1 Der Masterprozess

Der Masterprozess ist der Kern des `cluster`-Moduls. Bevor Sie die Funktionalitäten dieses Moduls nutzen können, müssen Sie zuerst den Masterprozess instanziiieren. Sie können den Masterprozess mithilfe der Methode `setupMaster` erstellen und konfigurieren. Sie können dieser Methode beim Aufruf ein Objekt übergeben, das die Konfiguration enthält. Mit insgesamt drei Eigenschaften des Konfigurationsobjekts können Sie das Verhalten des Masterprozesses beeinflussen. Die verschiedenen Eigenschaften und ihre Bedeutung sehen Sie in Tabelle 7.4.

Eigenschaft	Standardwert	Bedeutung
exec	<code>__filename</code>	Name der Datei, die als Workerprozess ausgeführt werden soll.
args	<code>Process.argv.slice(2)</code>	Argumente, die dem Workerprozess übergeben werden sollen.
silent	<code>false</code>	Wird diese Eigenschaft auf <code>true</code> gesetzt, wird die Standardausgabe des Masterprozesses nicht verwendet.

Tabelle 7.4 Konfiguration für den Masterprozess

Die Einstellungen, die Sie über `setupMaster` durchführen, werden in der Eigenschaft `settings` des `cluster`-Objekts gespeichert. Sie sollten allerdings Änderungen an den Einstellungen stets über die `setupMaster`-Methode durchführen und die `settings`-Eigenschaft nicht direkt bearbeiten. Bitte beachten Sie, dass Sie die `setupMaster`-Methode lediglich einmal aufrufen können. Beim zweiten Aufruf haben Änderungen an den Einstellungen keine Wirkung. Ein mehrmaliger Aufruf dieser Methode hat also keine Konsequenzen auf Ihre Applikation. Sie können diese Methode lediglich im Kontext des Masterprozesses und nicht von einem Workerprozess aus aufrufen.

Möchten Sie keine Anpassungen an den Standardeinstellungen vornehmen, müssen Sie die `setupMaster`-Methode auch nicht aufrufen. Sobald Sie einen Workerprozess mit der `fork`-Methode erstellen, wird automatisch die `setupMaster`-Methode aufgerufen. Das führt dazu, dass Sie `setupMaster` nicht manuell aufrufen müssen, wenn Sie keine speziellen Einstellungen vornehmen möchten. Auf der anderen Seite bedeutet das aber auch, dass Sie nach einem Aufruf von `fork` keine Änderungen mehr an den Einstellungen Ihres Masterprozesses vornehmen können.

Falls Sie den Quellcode des Masterprozesses und der Workerprozesse in einer Datei verwalten möchten, stellt Ihnen das `cluster`-Modul mit den Eigenschaften `isMaster` und `isWorker` eine Möglichkeit zur Verfügung, mit der Sie prüfen können, in welchem Kontext Sie sich aktuell in Ihrer Applikation bewegen. Intern greift diese Funktionalität auf die Eigenschaft `NODE_UNIQUE_ID` von `process.env` zu. Diese Eigenschaft verwendet das `cluster`-Modul, um einem Workerprozess eine eindeutige Identifikationsnummer zuzuweisen. Befinden Sie sich im Masterprozess, weist die Eigenschaft den Wert `undefined` auf. Sobald ein Kindprozess erstellt wird, wird diesem eine eindeutige Identifikationsnummer zugewiesen.

Das `cluster`-Modul ist wie viele andere Komponenten der Node.js-Plattform auch eventgetrieben aufgebaut. Das bedeutet, dass verschiedene Vorgänge bei der Verwendung dieses Moduls bestimmte Ereignisse auslösen. Das erste Ereignis, mit dem

Sie in Berührung kommen, ist das `setup`-Event. Es wird ausgelöst, sobald Sie den Masterprozess mit der `setupMaster`-Methode initialisieren. Dies geschieht entweder explizit durch den Aufruf der Methode oder implizit, indem Sie `fork` aufrufen. In beiden Fällen wird jedoch das `setup`-Event ausgelöst. Die Callback-Funktion, die Sie an dieses Event binden können, erhält kein Argument.

Der nächste Schritt bei der Verwendung des `cluster`-Moduls besteht darin, dass Sie Workerprozesse durch die `fork`-Methode erstellen. Der Aufruf dieser Methode führt zu einem `fork`-Event. Innerhalb der Callback-Funktion können Sie über das Argument, das diese Funktion erhält, auf den Workerprozess zugreifen.

Sobald der Workerprozess gestartet ist, wird auf dem `cluster`-Objekt das `online`-Event ausgelöst. Auch in diesem Fall erhalten Sie innerhalb der Callback-Funktion über das Argument Zugriff auf den Workerprozess.

Das `cluster`-Modul ist spezialisiert auf das Loadbalancing zwischen mehreren Serverprozessen. Sowohl beim Net-Server als auch beim HTTP-Server binden Sie Ihren Server über einen Aufruf der `listen`-Methode an einen Port beziehungsweise an einen Socket. Sobald Sie diese Methode in Ihrem Workerprozess aufrufen, wird das `listening`-Event auf dem `cluster`-Objekt ausgelöst, und Sie können im Masterprozess entsprechend darauf reagieren. Diese Methode erhält neben dem Objekt, das den Workerprozess repräsentiert, als zweites Argument die Adresse, auf die der Server des Workerprozesses gebunden ist.

Die beiden letzten Ereignisse werden beim Beenden des Workerprozesses ausgelöst. Das `disconnect`-Event signalisiert, dass die Kommunikationsverbindung zwischen Master- und Workerprozess beendet wurde und so keine Möglichkeit der Interaktion mehr besteht. Dieses Event wird im Normalfall als Auswirkung auf einen Aufruf der Methoden `disconnect` oder `destroy` auf den Workerprozess ausgelöst. Das `disconnect`-Event wird auch ausgelöst, wenn der Workerprozess von außerhalb, beispielsweise durch das `kill`-Kommando des Betriebssystems, beendet wird. Als einziges Argument steht Ihnen innerhalb der Callback-Funktion, die Sie an dieses Ereignis binden können, die Objektrepräsentation des Workerprozesses zur Verfügung.

Das zweite Ereignis, das beim Beenden eines Workerprozesses ausgelöst wird, ist das `exit`-Event. Es zeigt dem Masterprozess an, dass einer der Workerprozesse beendet wurde. Dieses Event wird im Normalfall nach dem `disconnect`-Event ausgelöst. In der Callback-Funktion dieses Events stehen Ihnen neben dem Objekt für den Workerprozess, der beendet wurde, noch zwei weitere Argumente zur Verfügung. Das zweite Argument ist eine Zahl, die den Exitcode des Workerprozesses angibt. Das dritte und letzte Argument ist der Name des Signals, das zur Beendigung des Workerprozesses geführt hat, also beispielsweise `SIGHUP`.

Tabelle 7.5 fasst für Sie die Ereignisse, die Ihnen im Zusammenhang mit dem `cluster`-Modul zur Verfügung stehen, nochmals kurz zusammen.

Ereignis	Bedeutung
setup	Dieses Ereignis wird ausgelöst, sobald die <code>setupMaster</code> -Methode ausgeführt wird, um den Masterprozess zu konfigurieren.
fork	Das <code>fork</code> -Event tritt auf, wenn Sie mit der <code>fork</code> -Methode einen neuen Workerprozess starten.
online	Der Workerprozess meldet mit dem <code>online</code> -Event seine Bereitschaft zurück.
listening	Sobald innerhalb des Workerprozesses die <code>listen</code> -Methode aufgerufen wird, wird das <code>listening</code> -Event ausgeführt.
disconnect	Sobald die Kommunikationsverbindungen zwischen Master- und Workerprozess beendet sind, erhalten Sie das <code>disconnect</code> -Event.
exit	Dieses Ereignis signalisiert, dass ein Worker beendet wurde.

Tabelle 7.5 Zusammengefasste Cluster-Ereignisse

Innerhalb des Masterprozesses können Sie auf drei Methoden zurückgreifen. Die Verwendung der ersten Methode `setupMaster` haben Sie bereits kennengelernt. Sie dient zur Konfiguration des generellen Verhaltens des Masterprozesses. Die `fork`-Methode kennen Sie ebenfalls schon von anderer Stelle, nämlich aus dem `child_process`-Modul. Im `cluster`-Modul dient diese Methode dazu, die Workerprozesse zu starten. Das Beispiel in Listing 7.10 verdeutlicht Ihnen die Verwendung der verschiedenen Methoden.

```
var cluster = require('cluster');

if (cluster.isMaster) {
  cluster.setupMaster({silent: true});

  setTimeout(cluster.disconnect, 2000);

  cluster.on('exit', function () {
    console.log('Worker wurden nach 2000ms beendet');
  });

  cluster.fork();
  cluster.fork();
}

if (cluster.isWorker) {
```

```

var http = require('http');
http.createServer(function (req, res) {
  console.log('Worker ' + cluster.worker.id);
  res.end('Worker ' + cluster.worker.id);
}).listen('8080');
}

```

Listing 7.10 Das cluster-Modul im Einsatz

Alle drei Methoden, also sowohl `setupMaster`, `fork` als auch `disconnect`, können nur im Masterprozess ausgeführt werden. Rufen Sie eine dieser drei Methoden aus einem Workerprozess heraus auf, resultiert dies in einem Fehler und dem Abbruch des Workerprozesses. Damit Sie nicht auf dieses Problem stoßen, können Sie die beiden Eigenschaften `isMaster` und `isWorker`, wie Sie in Listing 7.10 sehen, verwenden.

Der Masterprozess verfügt mit `workers` über eine weitere sehr hilfreiche Eigenschaft. `workers` ist ein Objekt, das Verweise auf sämtliche Workerprozesse hält. Die Schlüssel des Objekts sind die IDs der jeweiligen Workerprozesse, und die Werte sind die Objektrepräsentationen der Workerprozesse. Mit dieser Struktur können Sie herausfinden, wie viele Workerprozesse aktuell gestartet sind und auch steuernd auf die einzelnen Worker zugreifen.

7.4.2 Die Workerprozesse

Mithilfe der `fork`-Methode können Sie aus dem Masterprozess heraus eine beliebige Anzahl an Workerprozessen erzeugen. Diese Worker werden innerhalb Ihrer Applikation durch Objekte repräsentiert, die Ihnen verschiedene Eigenschaften, Methoden und Ereignisse zur Interaktion mit diesen Prozessen bieten.

Bevor Sie sich nun mit den Worker-Objekten beschäftigen, sollten Sie wissen, wie Sie an diese Objekte gelangen. Zu diesem Zweck gibt es verschiedene Möglichkeiten. Befinden Sie sich im Masterprozess, können Sie entweder über die Eigenschaft `cluster.workers` auf die verschiedenen Worker zugreifen. Eine weitere Möglichkeit aus dem Masterprozess heraus ist, dass Sie Callback-Funktionen an verschiedene Ereignisse wie beispielsweise das `online`-Event binden. Auch hier erhalten Sie in Form von Argumenten Zugriff auf die Worker-Objekte. Aus dem Workerprozess heraus können Sie über `cluster.worker` auf das Objekt des aktuellen Workers zugreifen. Sie haben hier keine Möglichkeit, die übrigen Worker zu beeinflussen.

Jeder Workerprozess hat eine eindeutige Kennung. Diese können Sie mithilfe der `id`-Eigenschaft des Workerprozesses auslesen. Weitere Informationen über den entsprechenden Workerprozess bietet die Eigenschaft `process`. Diese Eigenschaft verdeutlicht wie auch schon die `fork`-Methode die Basis des `cluster`-Moduls. Die Grundlage für die Erstellung von Workerprozessen bildet die `fork`-Methode des `child_process`-

Moduls. So ist auch das Objekt, das in der Eigenschaft `process` des Workerprozesses gespeichert ist, vom Typ `ChildProcess`. Über dieses Objekt können Sie beispielsweise über `cluster.worker.process.pid` auf die Prozess-ID des Workers zugreifen. Die letzte Eigenschaft, die Ihnen im Rahmen des Workerprozesses zur Verfügung steht, trägt den Namen `suicide`. Diese Eigenschaft weist im Normalfall den Wert `undefined` auf. Er wird allerdings auf den Wert `true` gesetzt, sobald sich der Worker in Beendigung befindet, also entweder `disconnect` oder `destroy` aufgerufen wurde.

Die Methoden des Worker-Objekts bilden die Schnittstelle, über die Sie mit dem Workerprozess interagieren können. Das Beispiel in Listing 7.11 verdeutlicht, wie Sie mithilfe der `send`-Methode Nachrichten zwischen Master- und Workerprozessen versenden können. Wie Sie in diesem Beispiel auch sehen können, funktioniert die Kommunikation in beide Richtungen.

```
var cluster = require('cluster'),
    http = require('http');

if (cluster.isMaster) {
  cluster.fork();
  cluster.fork();

  cluster.on('listening', function (worker) {
    console.log('Worker ' + worker.id + ' ready and listening');

    setTimeout(function () {
      worker.send('Hello Worker');
    }, 2000);
  });

  for (var i in cluster.workers) {
    cluster.workers[i].on('message', function (i, msg) {
      console.log('Worker ' + i + ' said: ' + msg);
    }).bind(this, i);
  }
} else {
  http.createServer(function (req, res) {
    res.end('Hello Client');
  }).listen('8080');

  cluster.worker.on('message', function (msg) {
    console.log('Master said to Worker ' + cluster.worker.id + ': ' + msg);
  });
}
```

```
});

cluster.worker.send('Hello Master');
}
```

Listing 7.11 Kommunikation zwischen Master- und Workerprozessen

Nach dem Start des Workerprozesses sendet dieser über seine eigene `send`-Methode eine Nachricht mit dem Inhalt `Hello Master` an den Masterprozess. Dieser wiederum bindet in einer `for`-Schleife Callback-Funktionen an die `message`-Events der verschiedenen Workerprozesse, um auf deren Nachrichten reagieren zu können. Sobald die Nachricht beim Masterprozess eingeht, wird diese mit einigen weiteren Informationen auf der Konsole ausgegeben. Der Masterprozess seinerseits bindet eine Callback-Funktion an das `listening`-Event des Clusters. Sobald in einem Workerprozess die `listen`-Methode aufgerufen wird, wird der Masterprozess benachrichtigt und gibt eine entsprechende Information auf der Konsole aus. Innerhalb dieser Callback-Funktion wird ein Timeout auf zwei Sekunden gesetzt. Nach dessen Ablauf wird eine Nachricht an den Workerprozess geschickt. Der Workerprozess hat ebenfalls eine Callback-Funktion auf das `message`-Event registriert und kann so mit der Nachricht des Masterprozesses umgehen und diese ausgeben.

Neben der Möglichkeit, Nachrichten zu versenden, stehen Ihnen im Zusammenhang mit Workerprozessen zwei weitere Methoden zur Beendigung des Workerprozesses zur Verfügung. Die erste Methode `disconnect` kennen Sie bereits aus dem Masterprozess. Im Master führt diese Methode nichts weiter aus, als dass sie neben einigen Aufräumarbeiten die `disconnect`-Methode der einzelnen Workerprozesse aufruft. Sobald Sie die `disconnect`-Methode aufgerufen haben, akzeptiert der Worker keine weiteren eingehenden Verbindungen. Bestehen noch Verbindungen zu Clients, werden diese allerdings erst bedient, bevor der Workerprozess beendet wird.

Mit der `destroy`-Methode beenden Sie einen Workerprozess. Die Form hängt davon ab, von wo aus Sie die Methode aufrufen. Vom Masterprozess aus wird die `process.kill`-Methode ausgeführt. Befinden Sie sich im Kontext des Workerprozesses, wird `process.exit` ausgeführt. In beiden Fällen wird allerdings die Eigenschaft `suicide` auf `true` gesetzt, damit Sie feststellen können, ob ein Prozess durch `destroy` oder einen Systemfehler beendet wurde.

Auch im Workerprozess stehen Ihnen verschiedene Ereignisse zur Verfügung, mit deren Hilfe Sie auf das Eintreten verschiedener Situationen reagieren können. Im Lebenszyklus eines Workers treten die verschiedenen Ereignisse chronologisch auf. Die Existenz des Workerprozesses beginnt mit dem `online`-Event. Dieses wird ausgelöst, sobald der Worker in den Status `online` wechselt. Innerhalb der Callback-Funktion haben Sie über das Argument dieser Funktion Zugriff auf das Objekt, das den Worker repräsentiert.

Das nächste Ereignis, das im Zusammenhang mit einem Workerprozess auftritt, ist das `listening`-Event. Analog zum gleichnamigen Cluster-Ereignis wird dieses ausgelöst, sobald die `listen`-Methode innerhalb des Workerprozesses aufgerufen wird und der Prozessstatus von `online` auf `listening` wechselt. Als Argument steht Ihnen in der Callback-Funktion ein Objekt zur Verfügung, das die Adresse repräsentiert, an die der Worker gebunden ist.

Während der Laufzeit des Workerprozesses können Sie zwischen dem Master- und den Workerprozessen Nachrichten versenden. Das entsprechende `message`-Event haben Sie bereits kennengelernt. Als einziges Argument erhält die Callback-Funktion die Nachricht, die versandt wurde.

Am Ende der Existenz eines Workerprozesses stehen wiederum zwei Ereignisse. Das `disconnect`-Event wird ausgelöst, sobald der Workerprozess seine Kommunikationsverbindungen nach außen getrennt hat, und leitet damit das Ende des Workers ein. Es geht einher mit dem Wechsel des Prozessstatus zu `disconnected`. Das `exit`-Event schließlich bedeutet, dass der Workerprozess beendet ist. Über die Argumente `Exitcode` und `Signalname` können Sie herausfinden, was die Umstände der Beendigung des Prozesses waren. Der Exitcode ist eine Zahl, die Sie über die `process.exit`-Methode beeinflussen können. Im Normalfall bedeutet hier eine 0, dass der Prozess erfolgreich und fehlerfrei beendet wurde. Das zweite Argument, der `Signalname`, ist eine Zeichenkette, die den Namen des Signals trägt, mit dem der Prozess beendet wurde. Beide Argumente weisen nicht in jedem Fall einen Wert auf.

Mit den hier vorgestellten Möglichkeiten können Sie Serverapplikationen unter Node.js entwickeln, die nicht nur in einem Thread laufen. Sie können stattdessen beliebig viele Workerprozesse forken, die sich wiederum um die Abarbeitung von Clientanfragen kümmern. Die Workerprozesse können auf die gleichen Ressourcen wie beispielsweise TCP-Port oder UNIX-Socket gebunden werden. Das Betriebssystem kümmert sich in diesem Fall um das Loadbalancing zwischen den Prozessen, sodass eine möglichst optimale Auslastung erreicht wird. Der Vorteil dieser Umsetzung ist, dass weder Sie noch die Node.js-Plattform für die gleichmäßige Verteilung der Anfragen auf die verschiedenen Workerprozesse verantwortlich sind.

In den folgenden Abschnitten erfahren Sie mehr über das Konzept der Promises. Diese dienen als ein Hilfsmittel für Applikationen, in denen sehr viele Aufgaben asynchron erledigt werden und dadurch der Lesefluss des Quellcodes gestört ist.

7.5 Die Grundlagen von Promises

JavaScript ist eine Scriptsprache, die synchrone Programmierung schon aus dem Sprachkern heraus unterstützt. Bei der Entwicklung von Applikationen auf Basis der Node.js-Plattform werden Sie häufig mit asynchronen Funktionsaufrufen in Berüh-

rung kommen. Ein klares Zeichen dafür, dass eine Funktion asynchron ausgeführt wird, ist, dass Sie als Argument eine Callback-Funktion definieren. Handelt es sich dabei lediglich um einen Aufruf einer Funktion, besteht noch kein Problem. Schwieriger wird die Situation, wenn Sie damit beginnen, asynchrone Funktionsaufrufe zu schachteln, also eine asynchrone Funktion innerhalb der Callback-Funktion einer anderen aufzurufen. Diese Verschachtelung können Sie beliebig tief betreiben. An sich ist die häufige Verwendung asynchroner Funktionen auch kein Problem, da JavaScript dies auch sehr gut unterstützt. Der Nachteil bei dieser Art der Programmierung ist, dass die Lesbarkeit des Quellcodes erheblich leidet. Entweder Sie geben als Callback-Funktion jeweils nur ein Funktionsobjekt an, dann müssen Sie, wenn Sie den Quellcode nachvollziehen möchten, immer zwischen den verschiedenen Funktionsdefinitionen hin- und herspringen. Definieren Sie die Callback-Funktionen direkt inline, hat dies den Nachteil, dass Ihr Quellcode spätestens nach der zweiten oder dritten in sich geschachtelten Callback-Funktion sehr umfangreich und damit auch unübersichtlich wird.

Ein Problem der asynchronen Programmierung, die de facto zur Verschachtelung von Callback-Funktionen führt, ist, dass das Ergebnis einer asynchronen Operation nicht zum Zeitpunkt des Aufrufs als Rückgabewert vorliegt, sondern erst, wenn die Abarbeitung des Kommandos abgeschlossen ist. Sobald dies der Fall ist, wird die übergebene Callback-Funktion mit dem Resultat der Operation aufgerufen und die Logik der Callback-Funktion ausgeführt. Die JavaScript-Engine ist zu diesem Zeitpunkt mit der Abarbeitung des Applikationscodes bereits an einer ganz anderen Stelle angelangt. Aus diesem Grund ist es einem Entwickler nun nicht mehr möglich, den Quellcode linear von oben nach unten durchzulesen.

Und genau um diese Probleme zu lösen, wurde das Konzept der Promises eingeführt. Eine Promise ist ganz einfach ausgedrückt ein Objekt, das einen Rückgabewert oder eine Exception einer asynchronen Operation darstellt.

Bis zur Version 0.2 der Node.js-Plattform war das Konzept der Promises ein fester Bestandteil der Plattform selbst. Nach dieser Version wurde diese Funktionalität wieder entfernt mit dem Verweis, dass sich die Entwickler von Applikationen selbst um die Behandlung asynchroner Operationen mit Promises kümmern müssten. Daher gibt es mittlerweile mehrere Bibliotheken, die als NPM-Pakete vorliegen und die Sie installieren können, falls Sie Promises in Ihrer Applikation verwenden möchten.

Klarer wird das Konzept, das hinter den Promises steckt, anhand eines einfachen Beispiels. Angenommen, Sie möchten den Inhalt einer Datei auslesen und ihn auf der Konsole ausgeben, dann besteht die einfachste Implementierung in der Verwendung des `fs`-Moduls, wie Sie es in Listing 7.12 sehen.

```
var fs = require('fs'),
    filename = '/tmp/input.txt';
```

```

fs.open(filename, 'r', function (err, fd) {
  fs.stat(filename, function (err, stats) {

    var size = stats.size,
        buffer = new Buffer(size);

    fs.read(fd, buffer, 0, size, 0, function (err, bytes, content) {
      console.log(content.toString());

      fs.close(fd);
    });
  });
});

```

Listing 7.12 Asynchrones Auslesen einer Datei

In diesem Beispiel haben Sie gesehen, dass der Quellcode einer derart einfachen Lösung mit asynchronen Methoden bereits relativ unleserlich wird. Die gleiche Umsetzung mit Promises sehen Sie in Listing 7.13.

```

var fs = require('fs'),
    filename = '/tmp/input.txt';

fs.open(filename, 'r').then(function (err, fd) {
  return fs.stat(filename);
}).then(function (err, stats) {
  var size = stats.size,
      buffer = new Buffer(size);

  return fs.read(fd, buffer, 0, size, 0);
}).then(function (err, bytes, content) {
  console.log(content.toString());

  fs.close(fd);
});

```

Listing 7.13 Asynchrones Auslesen einer Datei mit Promises

Das Beispiel in Listing 7.13 geht davon aus, dass die asynchronen Dateisystemoperationen nun keine Callback-Funktionen mehr akzeptieren, sondern Promises zurückliefern. Verfolgen Sie diesen Ansatz konsequent weiter, erhalten Sie am Ende Quellcode wie in Listing 7.14.

```
var fs = require('fs'),
    filename = '/tmp/input.txt';

fs.open(filename, 'r').then(getStats).then(readFile).then(printResult);
```

Listing 7.14 Asynchrones Auslesen einer Datei mit Promises und Funktionsobjekten

Sollten Sie den Quellcode aus Listing 7.13 und Listing 7.14 noch nicht verstanden haben, nur keine Sorge. Im Folgenden werden Sie sich eingehender mit Promises, den Konzepten, die dahinterstehen, und den Schnittstellen beschäftigen.

Eine Promise ist ein Objekt, das den Rückgabewert einer asynchronen Operation repräsentiert. Eine Promise verfügt über eine Methode mit dem Namen `then`, diese akzeptiert zwei Funktionen als Argumente. Die erste Funktion wird im Falle einer Erfüllung der Promise ausgeführt, wenn also ein Rückgabewert aus der asynchronen Operation vorliegt. Die zweite Funktion wird nur ausgeführt, wenn während der asynchronen Operation ein Fehler aufgetreten ist, die Promise also abgewiesen wurde. Diese zweite Funktion dient also dem asynchronen Exception Handling.

Gibt einer dieser beiden Funktionen wiederum eine Promise zurück, können Sie innerhalb Ihrer Applikation mit dieser weiterarbeiten. Durch dieses Vorgehen lassen sich Promises verketteten.

Lesen Sie den Quellcode aus Listing 7.14, sagt er aus: »Öffne die Datei `filename` für den lesenden Zugriff, dann hole die Statistiken über diese Datei, dann lies die Datei aus, dann gib das Resultat aus.« Genau diese Art der Lesbarkeit des Quellcodes soll durch Promises erreicht werden.

7.6 Promises in CommonJS

Mit dem ECMAScript-Standard wird versucht, den Sprachkern von JavaScript zu vereinheitlichen. Diese Standardisierung beinhaltet vor allem Konstrukte wie Schleifen, das `if`-Statement, Operatoren und einige Standardobjekte. Seitens des Browsers kommt hier das Document Object Model hinzu, auf dessen Elemente über die sogenannten Locators zugegriffen werden kann und das Ereignisse bietet, die Sie mithilfe von JavaScript abfangen und behandeln können.

Abseits des Browsers existiert neben dem zentralen ECMAScript-Standard das CommonJS-Projekt. Ziel dieses Projekts ist es, JavaScript weiter zu standardisieren. Unter anderem enthält CommonJS Vorgaben für Module, Pakete oder auch Promises. Mithilfe dieser Standardisierung werden die Schnittstellen für die entsprechenden Konzepte vereinheitlicht.

Wie die meisten Komponenten von CommonJS liegen auch Promises als Vorschläge vor. Diese Vorschläge wurden maßgeblich von Kris Zyp und Kris Kowal beeinflusst, von denen die meisten Beiträge zu dieser Standardisierung stammen.

7.7 Libraries

Wie bereits erwähnt, existieren für Node.js verschiedene Implementierungen des Promise-Konzepts. Zwei dieser Implementierungen möchte ich Ihnen im weiteren Verlauf dieses Kapitels vorstellen. Zum einen ist dies die Q-Bibliothek von Kris Kowal und zum anderen PromisedIO von Kris Zyp. Beide Bibliotheken setzen den CommonJS-Standard um und sind als NPM-Paket verfügbar. Weitere Informationen zum Thema CommonJS finden Sie unter <http://www.commonjs.org/>.

7.8 Q

Die Grundlage von Promises bildet ein Promise-Objekt. Hier stellt sich nun die Frage, wie Sie an ein derartiges Objekt gelangen. Da Promises nicht nativ von der Node.js-Plattform unterstützt werden, müssen Sie auf Bibliotheken wie Q zurückgreifen. Diese Bibliotheken bieten Ihnen verschiedene Möglichkeiten, um Promise-Objekte zu erzeugen. Im einfachsten Fall nutzen Sie die Methode `fcall`. Diese gibt Ihnen ein Promise-Objekt für die Funktion zurück, die Sie dieser Methode als Argument übergeben. In Listing 7.15 sehen Sie ein einleitendes Beispiel für die Verwendung der Q-Bibliothek.

```
var q = require('q');

var promise = q.fcall(function () {
  if (new Date().getTime() % 2 === 0) {
    throw new Error('Even number of milliseconds');
  }
  return 'Odd number of milliseconds'
});

promise.then(
  function (value) {
    console.log('Fulfillment: ' + value)},
  function (err) {
    console.log('Rejection: ' + err)
  });
```

Listing 7.15 Einsatz der `fcall`-Funktion zur Erzeugung von Promises

Das Beispiel in Listing 7.15 stellt eine Funktion zur Verfügung, die einen Fehler wirft, falls Sie sie zu einem Zeitpunkt aufrufen, an dem der aktuelle Zeitstempel durch zwei teilbar ist. Der Wert des Fehler-Objekts ist eine entsprechende Fehlermeldung. Rufen Sie die Funktion auf und ist der aktuelle Zeitstempel nicht durch zwei teilbar, erhalten Sie eine Zeichenkette mit der Meldung, dass es sich um eine ungerade Zahl von Millisekunden handelt.

Im Kern bedeutet das, dass der Aufruf von `q.fcall` als Rückgabewert ein Promise-Objekt liefert. Diese Promise gilt als erfüllt, wenn ein regulärer Wert zurückgegeben wird. Eine Abweisung der Promise liegt vor, wenn innerhalb der Funktion ein Fehler geworfen wird. Auf dem Promise-Objekt können Sie im Anschluss die `then`-Methode aufrufen und jeweils eine Funktion an den Erfolgs- beziehungsweise an den Fehlerfall binden.

7.8.1 Deferred

Arbeiten Sie mit asynchronen und Callback-basierten Funktionen, stößt die `fcall`-Methode an ihre Grenzen, und Sie müssen stattdessen auf die sogenannten Deferreds zurückgreifen. Listing 7.16 zeigt Ihnen den Einsatz der `defer`-Methode der Q-Bibliothek.

```
var q = require('q'),
    fs = require('fs');

var fileReader = function (filename) {
  var deferred = q.defer();

  fs.readFile(filename, 'utf8', function (err, data) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(data);
    }
  });

  return deferred.promise;
};

fileReader('/tmp/input.txt').then(function (data) {
  console.log(data);
}, function (err) {
  console.log(err);
});
```

Listing 7.16 Arbeiten mit Deferreds

Verwenden Sie Methoden der Node.js-Plattform, von denen eine Vielzahl asynchron aufgebaut ist, können Sie dies, wie im Beispiel in Listing 7.16 zu sehen ist, sehr einfach lösen. Sie müssen lediglich eine Wrapper-Funktion um die eigentliche asynchrone Methode erstellen. Innerhalb dieser Wrapper-Funktion erstellen Sie mithilfe der `defer`-Methode der Q-Bibliothek ein `Deferred`-Objekt. Die `promise`-Eigenschaft dieses Objekts enthält die Promise, die für den Rückgabewert oder einen eventuellen Fehler der Wrapper-Funktion steht. Innerhalb der Wrapper-Funktion können Sie nun die eigentliche Logik implementieren. Im Falle des Beispiels rufen Sie an dieser Stelle die `readFile`-Methode des `fs`-Moduls auf. Innerhalb der Callback-Funktion dieser Methode können Sie dann mit der `resolve`-Methode des `Deferred`-Objekts, auf das Sie auch in der Callback-Funktion durch das Scoping von JavaScript Zugriff haben, die Promise erfüllen und ihr einen entsprechenden Wert zuweisen. Für den Fall, dass ein Fehler auftritt, können Sie mit einem Aufruf der `reject`-Methode des `Deferred`-Objekts dafür sorgen, dass die Promise abgewiesen wird. Je nachdem, ob Sie die Promise erfüllt oder abgewiesen haben, wird die erste beziehungsweise zweite Funktion aufgerufen, die Sie an die `then`-Methode der Promise übergeben haben.

7.8.2 Node.js und Promises

Setzen Sie Promises häufiger innerhalb Ihrer Applikation ein, werden Sie öfter mit den vorgestellten Wrapper-Funktionen arbeiten müssen. Aus diesem Grund stellt die Q-Bibliothek speziell für die Callback-basierten asynchronen Methoden der Node.js-Plattform Methoden zur Verfügung, die Ihnen die Arbeit mit Promises erleichtern und Ihnen Schreibarbeit ersparen. Listing 7.17 zeigt Ihnen, wie Sie sich mit einer Kombination aus den Methoden `defer` und `makeNodeResolver` die Implementierung derartiger Wrapper-Funktionen ersparen können.

```
var q = require('q'),
    fs = require('fs'),
    deferred = q.defer();

fs.readFile('/tmp/input.txt', deferred.makeNodeResolver());

deferred.promise.then(function(data) {
    console.log(data);
}, function (err) {
    console.log(err);
});
```

Listing 7.17 Deferred Promises ohne Wrapper-Funktionen

Die Q-Bibliothek bietet mit `nfcalls`, `nfinvoke` und `nfbind` drei weitere Methoden, mit deren Hilfe Sie die eigentlich auf Callbacks basierenden Methoden der Node.js-Platt-

form dahingehend transformieren können, dass Sie durchgängig Promises verwenden können. In Listing 7.18 sehen Sie die Verwendung von `nfcalls`. Die Wirkungsweise des Beispiels gleicht der aus dem Beispiel von Listing 7.17.

```
var FS = require('fs'),
    Q = require('q');

var deferred = Q.nfcalls(FS.readFile, '/tmp/input.txt', 'utf8');

deferred.then(function (content) {
    console.log(content.toString());
}, function (err) {
    console.log(err);
});
```

Listing 7.18 Einsatz der `nfcalls`-Methode

In diesem Beispiel erhalten Sie als Rückgabewert der `nfcalls`-Methode direkt ein `Deferred`-Objekt, das Sie im Weiteren direkt verwenden können. Wie gewohnt, können Sie dann über die `promise`-Eigenschaft auf die `Promise` zugreifen und darauf die `then`-Methode aufrufen. Die `nfinvoke`-Methode ähnelt der `nfcalls`-Methode mit dem Unterschied, dass Sie hier statt eines Funktionsobjekts als erstes Argument das Objekt übergeben, auf dem die Methode aufgerufen werden soll. Als zweites Argument wird der Name der aufzurufenden Methode in Form einer Zeichenkette übergeben.

Die dritte Methode `nfbind` dient schließlich dazu, ein Funktionsobjekt zu erzeugen, das Sie innerhalb Ihrer Applikation mehrmals verwenden können. Listing 7.19 zeigt Ihnen einen entsprechenden Aufruf.

```
var q = require('q'),
    fs = require('fs');

var readFile = q.nfbind(fs.readFile);

var deferred = readFile('/tmp/input.txt', 'utf8');

deferred.then(
    function (value) {
        console.log(value);
    }, function (err) {
        console.log(err);
    });
```

Listing 7.19 Funktionsobjekte erzeugen, die `Deferred`-Objekte zurückliefern

Die `then`-Methode eines `Promise`-Objekts haben Sie in den vorangegangenen Beispielen bereits mehrfach benutzt. Sie können diese Methode verwenden, um Funktionen zu registrieren, die im Falle einer Erfüllung oder einer Abweisung einer `Promise` ausgeführt werden. Die erste Funktion, die Sie dieser Methode übergeben, wird ausgeführt, wenn die `Promise` erfüllt wird. Dies kann beispielsweise durch einen Aufruf der `resolve`-Methode auf das zugehörige `Deferred`-Objekt geschehen. Das zweite Argument der `then`-Methode ist eine Funktion, die im Falle einer Abweisung ausgeführt wird. Sie weisen eine `Promise` beispielsweise durch einen Aufruf der `reject`-Methode auf dem `Deferred`-Objekt ab. Beide Funktionen erhalten den Wert, mit dem Sie die `Promise` erfüllt beziehungsweise abgewiesen haben, als Eingabewerte.

Sollten Sie nur an der Erfüllung einer `Promise` interessiert sein, können Sie lediglich eine Funktion angeben. Diese wird dann im Falle einer Erfüllung der `Promise` ausgeführt. Im Fehlerfall wird dann für diese `Promise` keine Funktion ausgeführt. Anders ist der Fall, wenn Sie `Promises` verketteten, also wenn eine `then`-Methode wiederum ein `Promise`-Objekt zurückgibt und auf diesem wiederum die `then`-Methode aufgerufen wird. Listing 7.20 verdeutlicht diesen Fall.

```
step1()
  .then(step2)
  .then(step3)
  .then(step4, error1);
```

Listing 7.20 Verkettung von `Promises` mit einem `Errorhandler`

In diesem Beispiel geben die Funktionen `step1` bis `step3` jeweils ein `Promise`-Objekt zurück. Im Erfolgsfall wird die jeweils nächste Funktion ausgeführt. So folgt auf `step1` der Funktionsaufruf von `step2` und so weiter. Wird die `Promise` in `step1` allerdings nicht erfüllt, sondern abgewiesen, wird die Funktion `error1` ausgeführt, weil keine Funktion für den Fehlerfall definiert ist. Die abgewiesene `Promise` wird also so lange durchgereicht, bis entweder das Ende der Kette erreicht ist, in diesem Fall wird eine `Exception` geworfen, oder eine Funktion zur Behandlung einer abgewiesenen `Promise` existiert.

Sollten Sie statt an einer Erfüllung einer `Promise` lediglich an deren Abweisung interessiert sein, können Sie statt der `then`-Methode auch die `fail`-Methode verwenden.

```
var q = require('q');

q.fcall(function () {
  if (process.env.USER === 'root') {
    throw new Error('Executed as super user');
  }
  console.log('Hello ' + process.env.USER);
```

```
}).fail(function () {
    console.log('Please change your user');
});
```

Listing 7.21 Einsatz der fail-Methode

In Listing 7.21 sehen Sie ein Beispiel, in dem lediglich auf einen auftretenden Fehler innerhalb einer Funktion reagiert wird. Durch die Verwendung der `fail`-Methode wird ein Promise-Objekt erzeugt, das bei Auftreten des Fehlers automatisch abgewiesen wird. Da hier statt der `then`-Methode die `fail`-Methode verwendet wird, wird die Zeichenkette `Please change your user` nur ausgegeben, falls Sie als Benutzer mit dem Namen `root` angemeldet sind.

Das Konzept der Promises kann auch zur Fehlerbehandlung verwendet werden. Wenn Sie, wie im Beispiel in Listing 7.21, mit dem `throw`-Statement einen Fehler werfen, wird die Funktion ausgeführt, die Sie für den Fall einer Abweisung der Promise registriert haben. In diesem Fall verhält sich das Konstrukt sehr ähnlich zu einem `try-catch`-Block, bei dem der Fehler durch das `catch` entgegengenommen wird. Analog zu `try-catch`, bei dem es mit dem `finally`-Statement eine Möglichkeit gibt, Aufräumarbeiten durchzuführen, existiert für Promises mit der Q-Bibliothek die Methode `fin` in einem Promise-Objekt. Die Funktion, die Sie über diese Methode registrieren, wird in jedem Fall ausgeführt, unabhängig davon, ob die Promise erfüllt oder abgewiesen wird.

Alternativ zur `then`-Methode eines Promise-Objekts können Sie auch die `when`-Methode der Q-Bibliothek direkt ausführen. Sie akzeptiert insgesamt drei Argumente. Das erste ist ein Promise-Objekt, das zweite eine Funktion, die ausgeführt werden soll, falls die Promise erfüllt ist, und das dritte Argument schließlich ist die Funktion, die ausgeführt werden soll, falls die Promise abgewiesen wird.

Beachten Sie bitte, dass Sie nicht nur jeweils eine Funktion auf Erfüllung oder Abweisung einer Promise registrieren können, sondern beliebig viele. Das bedeutet, dass Sie die `then`- und `fail`-Methode mehrmals auf eine Promise anwenden können. Die angegebenen Funktionen werden registriert und alle entsprechend bei Erfüllung beziehungsweise Abweisung ausgeführt.

Neben der Q-Bibliothek existieren noch einige weitere Implementierungen von Promises für Node.js. Eine von ihnen ist `PromisedIO` von Kris Zyp, die Sie im Folgenden kennenlernen werden.

7.9 PromisedIO

`PromisedIO` verfolgt einen ähnlichen Ansatz wie die Q-Bibliothek. Sie bietet eine Implementierung von Promises für die Node.js-Plattform, wobei sich die grund-

legenden Schnittstellen beider Bibliotheken sehr ähneln. Listing 7.22 gibt Ihnen einen Überblick über die Verwendung der PromisedIO-Bibliothek.

```
var fs = require('fs'),
    promisedIO = require('promised-io/promise'),
    Deferred = promisedIO.Deferred;

var readFile = function (filename) {
    var deferred = new Deferred();

    fs.readFile(filename, 'utf8', function (err, data) {
        if (err) {
            deferred.reject(err);
        } else {
            deferred.resolve(data);
        }
    });

    return deferred.promise;
};

var promise = readFile('/tmp/input.txt');

promise.then(
    function (data) {
        console.log(data);
    },
    function (err) {
        console.log('error: ' + err);
    }
);

promisedIO.when(promise, function (data) {
    console.log('WHEN success: ' + data);
}, function (err) {
    console.log('WHEN error: ' + err);
});
```

Listing 7.22 PromisedIO im Einsatz

Wie Sie sehen, existieren einige der Methoden, die Sie bereits aus der Q-Bibliothek kennen, auch hier und verhalten sich sehr ähnlich. Ein Grund hierfür ist, dass beide auf den Spezifikationen für Promises von CommonJS aufbauen. Eine Besonderheit von PromisedIO ist, dass ein Modul mit dem Namen `fs` existiert. Hier werden die

Dateisystemoperationen von Node.js gekapselt und unverzüglich für die Verwendung mit Promises vorbereitet. So können Sie dann direkt `promisedIO.fs.readFile` aufrufen und müssen keine Callback-Funktion registrieren, sondern erhalten ein Promise-Objekt zurück.

7.10 Zusammenfassung

Im Verlauf dieses Kapitels haben Sie erfahren, wie Sie mit der Node.js-Plattform Aufrufe externer Kommandos und sogar ganze Teile Ihrer Applikation aus dem Haupt-Thread in die sogenannten Kind- oder Workerprozesse auslagern können. Aufbauend auf diesem Konzept bietet Ihnen das `cluster`-Modul die Möglichkeit, mehrere Serverprozesse für Net- oder HTTP-Server zu starten und dem Betriebssystem das Loadbalancing zwischen diesen Prozessen zu überlassen.

Schließlich haben Sie mit den Promises ein Konzept kennengelernt, mit dem Sie asynchrone Aufrufe innerhalb Ihres Quellcodes besser strukturieren und so den Quellcode lesbarer gestalten können. Promises sind dabei auch ein Mittel der Fehlerbehandlung im asynchronen Kontext. Eine Promise steht für das positive oder negative Ergebnis einer asynchronen Operation und kann erfüllt oder abgewiesen werden, wobei die Erfüllung für den Erfolgsfall steht und dafür sorgt, dass die Funktion, die der `then`-Methode als erstes Argument übergeben wird, ausgeführt wird. Eine Abweisung einer Promise bedeutet einen Fehlschlag der asynchronen Operation und ist einer Exception gleichzusetzen. Aufgrund einer abgewiesenen Promise wird die Funktion, die Sie als zweites Argument der `then`-Methode übergeben haben, aufgerufen. Die hier vorgestellten Features stellen lediglich einen Teil der Möglichkeiten dar, die die jeweiligen Bibliotheken zur Verfügung stellen. Sollten Sie sich mehr für den jeweiligen Funktionsumfang interessieren, möchte ich Sie an dieser Stelle auf die Online-Dokumentation der entsprechenden Bibliothek verweisen.

Im nächsten Kapitel sehen Sie, wie Sie mit Node.js verschiedene Datenbanken ansprechen und die den jeweiligen Datenbanken zugrunde liegenden Konzepte zu Ihrem Vorteil ausnutzen können.

Kapitel 8

Anbindung von Datenbanken

Phantasie ist wichtiger als Wissen, denn Wissen ist begrenzt.
– Albert Einstein

Wie Sie bereits in den vergangenen Kapiteln gesehen haben, müssen Sie bei der Implementierung von Applikationen auch immer wieder Daten speichern. Node.js stellt Ihnen zur Persistierung von Daten lediglich das `fs`-Modul zur Verfügung. Der Nachteil dieser Methode ist, dass der Zugriff auf die Daten problematisch ist. Im schlechtesten Fall müssen Sie die gesamte Datei nach den gewünschten Informationen durchsuchen, was je nach Dateigröße sehr zeitintensiv sein kann. Dies ist ein Grund, warum Sie zur Persistierung von Informationen in Ihrer Applikation Datenbanken verwenden sollten.

Der Zugriff auf Datenbanken erfolgt in der Regel nicht direkt. In den meisten Fällen benötigen Sie eine Art von Treiber, der Ihnen als Schnittstelle zur Datenbank dient. Diese Treiber sind kein fester Bestandteil der Node.js-Plattform, sondern liegen als NPM-Module vor, die Sie je nach Bedarf installieren können.

Durch diese Struktur ist die Unterstützung von Datenbanken nicht nur auf einige wenige Systeme beschränkt, stattdessen können Sie beispielsweise die relationalen Datenbanken MySQL, MSSQL oder SQLite verwenden. Auch viele nicht-relationale Datenbanken wie Redis, CouchDB oder MongoDB werden mittlerweile unterstützt.

In diesem Kapitel erfahren Sie mehr über die Anbindung verschiedener Datenbanken an Ihre Node.js-Applikation. Konkret werden Sie MySQL, SQLite, Redis und MongoDB einsetzen. Für jede dieser Datenbanken werden Sie die grundlegenden Datenbankoperationen, die im Akronym CRUD zusammengefasst sind, umsetzen. CRUD bedeutet Create, Read, Update und Delete. Sie erstellen also neue Datensätze, lesen diese aus, aktualisieren Werte in der Datenbank und entfernen bestehende Datensätze wieder aus der Datenbank.

Außerdem erfahren Sie mehr über die Einsatzgebiete und Vor- und Nachteile der einzelnen Datenbanken. Das Kapitel startet mit dem Einsatz von relationalen Datenbanken.

8.1 Node.js und relationale Datenbanken

Relationale Datenbanken haben sich über Jahre zu einem Standard für Datenbanken in der Informationstechnologie etabliert. Dieser Typ von Datenbanken basiert auf einer Struktur von Tabellen, in denen die Daten vorgehalten werden. Eine Tabelle definiert dabei den Aufbau einzelner Datensätze.

Sie können sich eine Tabelle in einer relationalen Datenbank wie eine ganz gewöhnliche Tabelle vorstellen. Die Zeilen stellen die einzelnen Datensätze dar und die Spalten die jeweiligen Eigenschaften. Das bedeutet, dass sämtliche Datensätze einer Tabelle die gleiche Struktur aufweisen.

In relationalen Datenbanken haben Sie die Möglichkeit, nicht nur eine Tabelle, sondern nahezu beliebig viele zu definieren. Diese Tabellen können Sie dann über sogenannte *foreign keys*, auf Deutsch *Fremdschlüssel*, in Verbindung zueinander setzen. So können Sie beispielsweise eine Tabelle definieren, in der Sie die Daten Ihrer Benutzer wie den Namen oder das Geburtsdatum speichern. In einer zweiten Tabelle können Sie dann die Adressinformationen speichern. Über einen Fremdschlüssel können Sie zwischen den Datensätzen aus beiden Tabellen eine Beziehung herstellen.

Als Sprache zur Formulierung von Abfragen der Datenbank hat sich die Structured Query Language, kurz SQL, durchgesetzt. SQL wurde durch die ISO und IEC standardisiert, wobei jede konkrete Implementierung eigene Erweiterungen zu diesem Standard hinzufügt. Weitere Informationen finden Sie unter <http://de.wikipedia.org/wiki/SQL>.

Die Abschnitte zu den einzelnen Datenbanken folgen dabei stets dem gleichen Aufbau. Sie erfahren einige Detailinformationen über die verschiedenen Datenbanken und die jeweiligen Vor- und Nachteile. Darauf folgen die Installation des jeweiligen Treibers für die Datenbank und ein konkretes Beispiel zur Verwendung der Datenbank innerhalb einer Node.js-Applikation. Als Beispiel dient eine Datenbank für Adressdaten.

Der erste Teil dieses Kapitels beschäftigt sich mit der Anbindung einer MySQL-Datenbank.

8.1.1 MySQL

In den vorangegangenen Kapiteln haben Sie bereits mehrmals mit MySQL als Datenbank in Node.js-Applikationen zu tun gehabt. Diese Abschnitte wiederholen die Konzepte, die Sie bereits angewandt haben.

MySQL ist eine der am weitesten verbreiteten Datenbanken im Web. Mit nahezu allen bedeutenden Programmiersprachen können Sie auf diese Datenbank zugreifen, so auch von Node.js aus.

MySQL hat sich bereits seit vielen Jahren auch in sehr großen Applikationen als Datenbank bewährt. Das System verfügt nicht nur über eine einfache Serverkomponente, sondern erlaubt es auch, dass Sie eine Datenbank auf mehreren Servern im Master-Slave-Verbund betreiben. Das hat den Vorteil, dass Sie die Anfragen auf mehrere Systeme verteilen und so auch Ausfallsicherheit gewährleisten können.

Für sehr große Datenmengen lässt sich eine Datenbank auch partitionieren und auf mehrere Systeme verteilen, was weitere Möglichkeiten hinsichtlich Performancesteigerungen bietet.

MySQL bietet viele nützliche Features. Diese umfassen beispielsweise Trigger, also Funktionen, die bei bestimmten Operationen ausgeführt werden, beziehungsweise Transaktionen. Dies ist eine Gruppe von Operationen, die nur in ihrer Gesamtheit oder überhaupt nicht ausgeführt werden dürfen.

Ein weiterer Vorteil von MySQL ist, dass diese Datenbank auf sehr vielen verschiedenen Betriebssystemen wie beispielsweise Linux, Windows oder Mac OS X verfügbar ist. Diese Verfügbarkeit und die weite Verbreitung haben zu einer sehr aktiven Community geführt, auf die Sie bei Fragen oder Problemen zurückgreifen können.

Ein Nachteil von MySQL ist, dass es hinsichtlich des Featuresets noch nicht ganz mit den großen SQL-Datenbanken wie Oracle oder DB2 mithalten kann. Dieser Nachteil wird aber immer mehr durch die Weiterentwicklung der Datenbank ausgeglichen.

Generell bestehen zwei verschiedene Ansätze zur Verbindung mit einer MySQL-Datenbank. Zum einen existiert ein Treiber, der das MySQL-Protokoll selbst implementiert und komplett in JavaScript geschrieben ist. Für diesen Treiber wird keine weitere Software benötigt, und er kann direkt verwendet werden, um mit einer Datenbank zu arbeiten.

Eine weitere Variante von MySQL-Treibern basiert auf den MySQL-Clientbibliotheken. Diese haben den Vorteil, dass sie etwas performanter als die komplett in JavaScript implementierten Treiber arbeiten. Sie weisen allerdings den Nachteil auf, dass die MySQL-Clientbibliotheken auf dem System, auf dem die Node.js-Applikation ausgeführt wird, installiert sein müssen.

Bisher haben Sie lediglich den Einsatz des MySQL-Treibers mitverfolgen können, der komplett in JavaScript geschrieben wurde. In den nächsten Abschnitten sehen Sie, wie Sie einen Treiber verwenden können, der auf den MySQL-Clientbibliotheken basiert.

Installation

Der auf den MySQL-Clientbibliotheken basierende Treiber ist als NPM-Modul verfügbar und kann über die Kommandozeile installiert werden. Wie bereits erwähnt, müssen Sie jedoch darauf achten, dass auf Ihrem System die entsprechenden

Bibliotheken installiert sind. In Listing 8.1 sehen Sie die Kommandos, die erforderlich sind, um das Modul `db-mysql` auf einem Linux-System zu installieren.

```
apt-get install libmysqlclient-dev  
npm install db-mysql
```

Listing 8.1 Installation von `db-mysql`

Nachdem Sie diese Kommandos abgesetzt haben, können Sie von Ihrer Applikation aus auf eine MySQL-Datenbank zugreifen.

Datenbankstruktur

Damit Sie die nachfolgenden Beispiele nachvollziehen können, benötigen Sie eine lauffähige Instanz einer MySQL-Datenbank auf einem Server. Im Falle dieses Beispiels wird angenommen, dass sich Applikation und Datenbank auf dem gleichen Rechner befinden.

```
CREATE DATABASE `node`;  
USE `node`;  
CREATE TABLE `Addresses` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `street` varchar(255) DEFAULT NULL,  
  `place` varchar(255) DEFAULT NULL,  
  `country` varchar(255) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Listing 8.2 Struktur der MySQL-Datenbank

Haben Sie die Statements aus Listing 8.2 ausgeführt, können Sie im nächsten Schritt mit der Umsetzung der Beispielapplikation beginnen.

Verbindung aufbauen

In jeder Applikation, die mit einer Datenbank arbeitet, muss vor der Verwendung dieser Datenbank zuerst eine Verbindung aufgebaut werden, über die einerseits die Kommandos an das Datenbanksystem gesendet werden und über die auf der anderen Seite die Informationen der Datenbank an die Applikation zurückfließen können. Listing 8.3 zeigt Ihnen, wie Sie eine solche Verbindung aufbauen.

```
var mysql = require('db-mysql');  
  
new mysql.Database({  
  hostname: 'localhost',
```

```

    user: 'root',
    password: '',
    database: 'node'
  }).on('error', function (err) {
    console.log('An error occurred: ' + err);
  }).connect(function (err) {
    console.log('connection established');
  });

```

Listing 8.3 Aufbau einer Datenbankverbindung

Im Quellcode von Listing 8.3 sehen Sie, dass Sie der `Database`-Methode eine Objektstruktur übergeben können, die die Konfiguration der Verbindung enthält. Auf dem zurückgegebenen Objekt können Sie verschiedene Methoden aufrufen. Über die `on`-Methode können Sie beispielsweise direkt Callback-Funktionen an Ereignisse binden, die im Verlauf des Verbindungsaufbaus auftreten können. Konkret sind dies die Ereignisse `error`, das im Fehlerfall auftritt, und `ready`, das ausgelöst wird, sobald die Datenbankverbindung hergestellt wurde.

Die wichtigste Methode im Zusammenhang mit der Datenbankverbindung ist `connect`. Diese Methode akzeptiert eine Callback-Funktion als Wert, die aufgerufen wird, sobald eine Verbindung hergestellt wurde. Sämtliche Datenbankoperationen finden also innerhalb dieser Callback-Funktion statt.

Neue Datensätze anlegen

Noch verfügt Ihre Datenbank lediglich über eine leere Tabelle. Bevor Sie also Datensätze auslesen oder diese verändern können, müssen Sie zuerst neue Datensätze anlegen. In Listing 8.4 sehen Sie, wie Sie dies bewerkstelligen können. Den Code dieses und der folgenden Listings sollten Sie innerhalb der Callback-Funktion der `connect`-Methode platzieren.

```

this.query().insert('Addresses',
  ['street', 'place', 'country'],
  ['Broadway 1', 'New York', 'United States of America']
).execute(function (err, result) {
  if (err) throw err;
  console.log(result);
});

```

Listing 8.4 Neue Datensätze in MySQL anlegen

Sie legen einen neuen Datensatz durch eine Kombination der Methoden `query`, `insert` und `execute` an. Dabei stellt Ihnen die `Query`-Methode das `Query`-Objekt zur

Verfügung, auf dem Sie die `insert`-Methode aufrufen. Dieser Methode müssen Sie als ersten Wert den Namen der Tabelle übergeben, in die Sie die Daten einfügen möchten. Das zweite Argument besteht aus einem Array von Spaltennamen, denen im dritten Argument durch ein Array Werte zugewiesen werden.

Die `execute`-Methode, die Sie auf dem Rückgabewert der `insert`-Methode aufrufen, führt schließlich die Abfrage aus. Diese Methode akzeptiert wiederum eine Callback-Funktion, die ausgeführt wird, sobald das Ergebnis der Abfrage vorliegt. Als erstes Argument erhält diese Callback-Funktion ein Fehlerobjekt, das einen Wert beinhaltet, falls bei der Abfrage ein Fehler aufgetreten ist. Das zweite Argument beinhaltet das Resultat der Abfrage. Im Falle eines Inserts sind dies die ID des neuen Datensatzes, die Anzahl der Datensätze, die betroffen waren, und die Anzahl an Warnungen, die bei der Operation aufgetreten sind.

Datensätze auslesen

Nachdem Sie Datensätze in die Datenbank eingefügt haben, werden Sie diese in den meisten Fällen auch wieder auslesen wollen. Der MySQL-Treiber bietet Ihnen hierfür die `select`-Methode des `Query`-Objekts. In Listing 8.5 sehen Sie den konkreten Einsatz dieser Methode.

```
this.query().select('*').from('Addresses').
execute(function(err, rows, cols) {
    if (err) throw err;
    console.log(rows);
    console.log(cols);
});
```

Listing 8.5 Datensätze aus einer MySQL-Tabelle auslesen

Wie Sie in Listing 8.5 sehen können, geschieht die Abfrage der Datenbank wiederum über eine Kombination von Methoden. Am Anfang steht die `query`-Methode, die das `Query`-Objekt zur Verfügung stellt. Danach verketten Sie die verschiedenen Methoden, die schließlich die Abfrage bilden, die dann mithilfe von `execute` ausgeführt wird. Die `execute`-Methode akzeptiert eine Callback-Funktion als Wert. Diese Callback-Funktion erhält drei Argumente. Das erste ist, wie in Node.js üblich, ein Fehlerobjekt. Das zweite Argument ist das eigentliche Ergebnis der Abfrage. Im Falle des Beispiels ist dies ein Array von Objekten, die die gefundenen Datensätze darstellen. Das dritte Argument enthält schließlich ein Array mit Objekten, die die gefundenen Spalten repräsentieren.

Zum Aufbau einer Abfrage können Sie auf verschiedene Methoden zurückgreifen. Die Basis bilden die Methoden `select` und `from`. Zur Einschränkung der gefundenen Datensätze können Sie mit der `where`-Methode Bedingungen definieren, die die Datensätze

erfüllen müssen. Mit `join` können Sie mehrere Tabellen innerhalb einer Abfrage verbinden. Um den Rahmen dieses Kapitels nicht zu sprengen, werde ich hier nicht auf weiterführende Operationen wie die Verbindung mehrerer Tabellen eingehen.

Datensätze aktualisieren

Der Vorteil einer Datenbank ist, dass Sie sehr schnell auf Daten zugreifen und diese bei Bedarf auch ändern können. Diese Aktualisierung erreichen Sie über die `update`-Methode. Listing 8.6 zeigt Ihnen, wie Sie bei Datenaktualisierungen vorgehen.

```
this.query().update('Addresses').
set({'street': 'Tower Hill', 'place': 'London', 'country': 'United Kingdom'}).
where('id = ?', [1]).
execute(function(err, result) {
    if (err) throw err;
    console.log(result);
});
```

Listing 8.6 Datensätze in einer MySQL-Datenbank ändern

In Listing 8.6 formulieren Sie die Abfrage zum Ändern von Datensätzen ähnlich, wie Sie es auch direkt in SQL machen würden, also mit einer Kombination aus `query`, `update`, `set`, `where` und `execute`. Mit `update` geben Sie an, welche Tabelle betroffen ist. `set` erhält die Werte, die geändert werden sollen, in einer Objektstruktur als Argument. Mit `where` schränken Sie die Datensätze ein, die aktualisiert werden sollen. Das Fragezeichen in der Zeichenkette dient als Platzhalter, der bei der Ausführung durch den Wert im Array, das Sie als zweites Argument übergeben, ersetzt wird. Dabei wird dieser Wert korrekt escaped, um Angriffe mit SQL-Injections zu vermeiden. `execute` führt schließlich die Abfrage aus. Auch diesmal erhält diese Methode wieder eine Callback-Funktion mit einem Fehlerobjekt und einem Resultat-Objekt. Dieses Objekt enthält eine ID-Eigenschaft, die allerdings den Wert 0 aufweist, die Anzahl der betroffenen Zeilen und die Anzahl der Warnungen.

Datensätze entfernen

Die letzte Operation, die Sie hier kennenlernen, ist das Löschen von Datensätzen aus der Datenbank. Das Löschen von Datensätzen lässt sich nicht ohne Weiteres rückgängig machen. Arbeiten Sie mit referenzieller Integrität über Fremdschlüssel, kann das Löschen eines Datensatzes eine Kaskade von Löschungen anderer Datensätze nach sich ziehen, die auf diesem Datensatz aufbauen. Eine weiterführende Erklärung zu Fremdschlüsseln unter MySQL finden Sie unter <http://dev.mysql.com/doc/refman/5.5/en/create-table-foreign-keys.html>. Beim Löschen von Datensätzen sollten Sie also stets Vorsicht walten lassen. Listing 8.7 zeigt Ihnen, wie Sie Datensätze aus Ihrer Datenbank entfernen können.

```

this.query().delete().from('Addresses').
where('id = ?', [ 1 ]).
execute(function(err, result) {
    if (err) throw err;
    console.log(result);
});

```

Listing 8.7 Löschen von Datensätzen aus einer MySQL-Datenbank

Auch in Listing 8.7 sehen Sie die gewohnte Kombination von Methoden zum Formulieren der Abfrage. Die Callback-Funktion der `execute`-Methode erhält zwei Argumente. Das erste Argument enthält eventuelle Fehler, die bei der Löschoperation aufgetreten sind. Das zweite Argument enthält das Ergebnis der Abfrage. Im Fall des Löschens eines Datensatzes ist dies ein Objekt, das insgesamt drei Eigenschaften enthält. Die Eigenschaft `id` enthält immer den Wert 0, `affected` gibt an, wie viele Zeilen gelöscht wurden und `warnings` enthält die Anzahl der Warnungen, die während der Abfrage aufgetreten sind.

Eine Alternative zum endgültigen Löschen von Datensätzen ist das Markieren von Datensätzen. In der Datenbank wird dies durch ein weiteres Feld innerhalb der Tabelle repräsentiert. Dieses Feld enthält den Wert 0 für aktive Datensätze und den Wert 1 für gelöschte Datensätze. Der Nachteil dieser Variante ist, dass Sie sich selbst um die referenzielle Integrität Ihrer Datenbank kümmern müssen, also abhängige Datensätze selbst als gelöscht markieren müssen.

Eine leichtgewichtige Alternative zu MySQL als Datenbank für Ihre Node.js-Applikation ist SQLite.

8.1.2 SQLite

SQLite ist im Vergleich zu MySQL ein Leichtgewicht. Einer der wichtigsten Unterschiede ist, dass bei SQLite kein Serverprozess benötigt wird. Die Clientsoftware greift bei Abfragen direkt auf die in einer Datei gespeicherten Datenbank zu. SQLite erfordert außerdem keine Konfiguration. Sie können die Datenbank nach der Initialisierung sofort verwenden.

SQLite steht Ihnen auf den verschiedensten Systemen zur Verfügung. So gibt es vor-kompilierte Binärpakete für Linux, Mac OS X und auch für Windows.

Da SQLite die Datenbankdaten in einer Datei speichert, können diese auf sehr einfache Weise kopiert und gesichert werden.

Im Gegensatz zu MySQL verfügt SQLite über keine eigene Benutzerverwaltung. Sie haben also keine Möglichkeit, Berechtigungen zu vergeben, was ein Mehrbenutzersystem unmöglich macht.

Installation

Für SQLite existieren verschiedene Treiber, mit deren Hilfe Sie auf die Datenbank zugreifen können. In den folgenden Abschnitten lernen Sie den `sqlite3`-Treiber kennen. Dieser liegt als NPM-Paket vor und kann mit dem Kommando, das Sie in Listing 8.8 sehen, installiert werden.

```
$ npm install sqlite3
```

Listing 8.8 Installation des `sqlite3`-Treibers

Sobald Sie das Paket installiert haben, können Sie eine SQLite-Datenbank an Ihre Applikation anbinden und diese zur Persistierung von Daten verwenden.

Datenbankstruktur

Bevor Sie jedoch mit der Datenbank interagieren können, müssen Sie diese anlegen. Listing 8.9 führt Sie durch diesen Prozess, an dessen Ende eine funktionierende Datenbank mit einer Adresstabelle steht, die Sie in den folgenden Abschnitten als Ausgangssituation verwenden werden.

```
$ sqlite3 node.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE `Addresses` (
  `street` varchar(255) DEFAULT NULL,
  `place` varchar(255) DEFAULT NULL,
  `country` varchar(255) DEFAULT NULL
);
sqlite>
```

Listing 8.9 Struktur der SQLite-Datenbank

Beim Aufbau der Tabelle fällt Ihnen bestimmt das Fehlen der ID-Spalte im Vergleich zu MySQL auf. Ein solches Feld ist bei SQLite unnötig, da jeder Datensatz automatisch eine Zahl zugeordnet bekommt, die in der Tabelle eindeutig ist und über die der Datensatz identifiziert werden kann. Zugreifen können Sie auf diesen Wert über den Spaltennamen `rowid`.

Verbindung aufbauen

Da SQLite wie bereits erwähnt über keine interne Benutzerverwaltung verfügt, ist der Aufbau einer Verbindung relativ problemlos. Listing 8.10 zeigt Ihnen, wie Sie die Verbindung zu einer SQLite-Datenbank aufbauen können, die in der Datei `node.db` liegt.

```
var sqlite = require('sqlite3');
var db = new sqlite.Database('node.db');
```

Listing 8.10 Aufbau einer Verbindung zu einer SQLite-Datenbank

Um eine Verbindung zu Ihrer Datenbank aufzubauen, reichen zwei Statements aus. Im ersten Schritt binden Sie das NPM-Paket ein, das Ihnen die Verbindung zur Datenbank ermöglicht. Im zweiten Schritt initialisieren Sie schließlich die eigentliche Datenbankverbindung, indem Sie ein Datenbank-Objekt erzeugen und dabei den Namen der Datei angeben, in der sich die Datenbank befindet. Im weiteren Verlauf Ihrer Applikation können Sie mit diesem Datenbank-Objekt Ihre Abfragen an die Datenbank senden.

Zunächst nutzen Sie nun die bestehende Verbindung zur Datenbank, um einen neuen Datensatz anzulegen.

Neue Datensätze anlegen

Noch verfügt die Datenbank in der Datei `node.db` lediglich über eine leere Tabelle `Addresses`. Im ersten Schritt fügen Sie nun einen neuen Datensatz ein. Listing 8.11 zeigt Ihnen den dazu erforderlichen Quellcode. Dabei handelt es sich um eine Erweiterung von Listing 8.10, und es wird davon ausgegangen, dass `db` das Datenbank-Objekt enthält.

```
db.run('INSERT INTO Addresses VALUES (?, ?, ?)',
  ['Broadway 1', 'New York', 'United States of America'],
  function (err) {
    if (err) throw err;
    console.log('ID: ' + this.lastID);
  });
```

Listing 8.11 Neue Datensätze in eine SQLite-Tabelle einfügen

Mit der `run`-Methode des Datenbank-Objekts können Sie Abfragen zum Anlegen neuer Datensätze an die Datenbank absetzen. Diese Methode akzeptiert drei Argumente, von denen das zweite optional ist. Das erste Argument besteht aus der SQL-Abfrage, die an die Datenbank gesendet werden soll. Das zweite Argument enthält Parameter, die in die Abfrage eingebunden werden sollen. Sie sehen in Listing 8.11, dass die SQL-Abfrage drei Fragezeichen enthält. Diese werden durch das Array im zweiten Argument mit konkreten Werten ersetzt. Dabei kümmert sich der Datenbanktreiber um das korrekte Escaping der Werte. Das dritte Argument ist schließlich eine Callback-Funktion, die aufgerufen wird, sobald die Antwort des Servers vorliegt. Das einzige Argument, auf das Sie in dieser Callback-Funktion zugreifen können, ist ein Fehlerobjekt, das existiert, falls bei der Abfrage Probleme aufgetreten sein sollten.

Über `this` können Sie innerhalb der Callback-Funktion auf weitere Informationen zugreifen. Im Falle einer `INSERT`-Abfrage steht Ihnen hier die Eigenschaft `lastID` zur Verfügung. Sie enthält die `rowid` des Datensatzes, den Sie mit diesem Kommando eingefügt haben.

Das Resultat der Operation können Sie entweder direkt über einen Kommandozeilenclient für SQLite prüfen, oder Sie erweitern Ihren Code um eine weitere Abfrage, die die entsprechenden Werte wieder aus Ihrer Datenbank ausliest. Der nächste Abschnitt beschäftigt sich mit genau dieser Problemstellung.

Datensätze auslesen

Haben Sie erst einmal Daten in Ihre Datenbank geschrieben, möchten Sie diese zu einem bestimmten Zeitpunkt auch wieder auslesen. Der von Ihnen bisher verwendete SQLite-Treiber verfügt über mehrere Methoden, mit deren Hilfe Sie Daten aus Ihrer Datenbank auslesen können.

Die einfachste Variante, Daten aus Ihrer Datenbank auszulesen, ist die Verwendung der `get`-Methode. Mit ihr kann ein einzelner Datensatz abgefragt werden. Wie Sie hier genau vorgehen sollten, sehen Sie in Listing 8.12.

```
db.get('SELECT * FROM Addresses WHERE rowid = ?',
    [1],
    function (err, row) {
        if (err) throw err;
        console.log(row);
    });
```

Listing 8.12 Einen Datensatz aus einer SQLite-Datenbank auslesen

Die Signatur ist sehr ähnlich zur `run`-Methode. Der erste Wert, den Sie an den Aufruf dieser Methode übergeben, ist ebenfalls eine Zeichenkette, die die SQL-Abfrage enthält, die Sie ausführen möchten. Auch hier können Sie wieder Fragezeichen als Platzhalter verwenden, die durch die Werte im zweiten Argument ersetzt werden. Das dritte Argument besteht aus der Callback-Funktion, die ausgeführt wird, sobald das Ergebnis der Abfrage vorliegt. Das erste Argument dieser Funktion ist wie bei Node.js üblich ein Fehlerobjekt. Das zweite Argument besteht aus einem Objekt, das den ausgelesenen Datensatz repräsentiert. Die Spaltennamen sind dabei die Eigenschaften. Die jeweiligen Werte der Felder des Datensatzes werden dann diesen Eigenschaften zugewiesen.

Zusätzlich zur `get`-Methode bietet Ihnen der SQLite-Treiber zwei weitere Methoden. Mithilfe der `all`-Methode können Sie nicht nur einen einzelnen Datensatz, sondern sämtliche Datensätze, die eine `SELECT`-Abfrage zurückliefert, verarbeiten. Die Verwendung der `all`-Methode gleicht der der `get`-Methode, bis auf die Tatsache, dass Sie

in der Callback-Funktion nicht ein Objekt, sondern ein Array von Objekten als zweites Argument erhalten.

Die dritte Methode `each` ermöglicht es Ihnen schließlich, eine Callback-Funktion zu definieren, die für jeden Datensatz, den eine Abfrage zurückliefert, ausgeführt wird. Diese Methode verhält sich wie die `get`-Methode, außer dass sie ein viertes Argument akzeptiert. Dieses Argument ist eine weitere Callback-Funktion, die aufgerufen wird, sobald sämtliche Callbacks für die einzelnen Datensätze abgearbeitet wurden.

Der folgende Abschnitt zeigt Ihnen, wie Sie vorgehen können, falls Sie Datensätze, die sich bereits in Ihrer Datenbank befinden, ändern möchten.

Datensätze aktualisieren

In SQLite können Sie wie auch schon in MySQL die Informationen, die als Datensätze in Ihrer Datenbank liegen, über bestimmte Abfragen ändern. Der SQL-Standard sieht für diesen Fall die `UPDATE`-Abfrage vor. Um diese Art von Abfragen abzusetzen, müssen Sie auf die `run`-Methode zurückgreifen, die Sie bereits zum Anlegen von Datensätzen verwendet haben. Listing 8.13 stellt Ihnen vor, wie Sie bei der Aktualisierung von Datensätzen vorgehen sollten.

```
db.run('UPDATE Addresses SET street = ?, place = ?, country = ? WHERE rowid =
?',
    ['Tower Hill', 'London', 'United Kingdom', 1],
    function (err) {
        if (err) throw err;
        console.log(this.changes);
    });
```

Listing 8.13 Datensätze in einer SQLite-Datenbank aktualisieren

Der Quellcode dieses Listings funktioniert ähnlich zu dem, den Sie implementiert haben, als Sie Datensätze angelegt haben. Die einzigen Unterschiede sind die geänderte SQL-Abfrage im ersten Argument der `run`-Methode und die Tatsache, dass in der Callback-Funktion im dritten Argument statt auf `this.lastID` auf `this.changes` zugegriffen wird. Diese Eigenschaft enthält die Anzahl der betroffenen Zeilen.

Die letzte Operation, die Sie hier zu SQLite kennenlernen, ist das Entfernen von Datensätzen.

Datensätze entfernen

Um Datensätze aus Ihrer Datenbank wieder zu entfernen, können Sie wie beim Einfügen und Aktualisieren auch auf die `run`-Methode zurückgreifen. In Listing 8.14 sehen Sie, wie Sie Ihr Ziel mit ein paar einfachen Anpassungen des Quellcodes aus Listing 8.13 erreichen können.

```

db.run('DELETE FROM Addresses WHERE rowid = ?',
      [1],
      function (err) {
        if (err) throw err;
        console.log(this.changes);
      });

```

Listing 8.14 Datensätze aus einer SQLite-Datenbank löschen

In der `run`-Methode müssen Sie im Vergleich zur Aktualisierung von Datensätzen lediglich die SQL-Abfrage anpassen und die Parameter, die Sie verwenden möchten, entsprechend ändern. In der Callback-Funktion erfahren Sie wiederum über `this.changes`, wie viele Datensätze von der Löschung betroffen waren.

Wie schon in den Abschnitten über MySQL ist auch hier ein Wort der Warnung angebracht. Das Löschen von Datensätzen kann nicht umgekehrt werden. Das bedeutet, dass einmal gelöschte Datensätze unwiederbringlich verloren sind, es sei denn, Sie haben eine funktionierende Datensicherung, aus der Sie die Datensätze wiederherstellen können.

Die hier vorgestellten Features von SQLite und des SQLite-Treibers sind lediglich ein kleiner Ausschnitt des vollen Funktionsumfangs. Möchten Sie mehr darüber erfahren, was Sie mit dieser Datenbank und dem zugehörigen Treiber umsetzen können, verweise ich Sie an dieser Stelle auf die jeweilige Online-Dokumentation unter <http://www.sqlite.org/docs.html> für SQLite und <https://github.com/developmentseed/node-sqlite3/wiki> für den SQLite-Treiber.

Mit MySQL und SQLite haben Sie zwei Vertreter relationaler Datenbanken kennengelernt. In den folgenden Abschnitten erfahren Sie mehr über eine andere Kategorie von Datenbanken, die nicht-relationalen Datenbanken.

8.2 Node.js und nicht-relationale Datenbanken

Lange Zeit waren relationale Datenbanken die einzige weitverbreitete Art, Daten strukturiert zu speichern und sie für schnellen Zugriff wieder zur Verfügung zu stellen. Mittlerweile hat sich allerdings auch noch eine weitere Art von Datenbanken auf dem Markt etabliert. Sie setzen bewusst auf einen anderen Ansatz. Im Gegensatz zu relationalen Datenbanken mit ihren fest definierten Tabellenstrukturen wird bei nicht-relationalen Datenbanken keine derartige Struktur vorgeschrieben.

Häufig werden in nicht-relationalen Datenbanken Objekt- oder Dokumentstrukturen gespeichert, auf die Sie über bestimmte Schlüssel- beziehungsweise Indexwerte zugreifen können. Diese werden unter dem Sammelbegriff NoSQL zusammenge-

fasst. Weiterführende Informationen finden Sie unter <http://de.wikipedia.org/wiki/NoSQL>. In den folgenden Abschnitten lernen Sie mit Redis und MongoDB zwei Vertreter der Kategorie der nicht-relationalen Datenbanken kennen.

8.2.1 Redis

Im Kern ist Redis ein einfacher Key-Value-Store. Sie haben hier die Möglichkeit, Werte zu speichern und sie mit einem Schlüssel zu versehen. Über diesen Schlüssel können Sie wieder auf diesen Wert zugreifen.

Redis speichert die Werte im Arbeitsspeicher, was zu einer sehr hohen Leistung bei Lese- und Schreiboperationen führt. Der Nachteil dieser Technologie ist, dass der Inhalt des Arbeitsspeichers nach einem Systemabsturz unwiederbringlich verloren ist. Die Entwickler von Redis haben für dieses Problem eine sehr elegante Lösung gefunden. Das System kann zu definierbaren Zeitpunkten eine Sicherheitskopie der Datenbank auf die Festplatte schreiben, von der aus eine Wiederherstellung der Inhalte problemlos möglich ist.

Redis ist primär für POSIX-Systeme wie Linux, BSD oder Mac OS X verfügbar und wird auch auf diesen Systemen entwickelt. Für Windows existiert zwar eine Version des Redis-Servers, diese ist allerdings noch experimentell.

Installation

Der Redis-Client für Node.js liegt wie die übrigen Treiber für Datenbanken auch als NPM-Paket vor und kann über die Kommandozeile installiert werden. Das dafür notwendige Kommando können Sie Listing 8.15 entnehmen.

```
$ npm install redis
```

Listing 8.15 Installation des Redis-Clients

Sobald Sie den Redis-Server und das NPM-Paket mit dem Redis-Client auf Ihrem System installiert haben, können Sie Redis in Ihrer Node.js-Applikation verwenden.

Da Redis auf keine festen Strukturen setzt, müssen Sie keine weiteren Operationen zur Initialisierung einer Datenbank unternehmen, sondern können sich direkt um die Verbindung mit dem Server kümmern.

Verbindung aufbauen

Der Verbindungsaufbau zu einem Redis-Server ist ähnlich einfach wie der zu einer SQLite-Datenbank. In Listing 8.16 sehen Sie, wie Sie die Verbindung in nur wenigen Zeilen Quellcode herstellen können.

```

var redis = require('redis'),
    client = redis.createClient();

client.on('error', function (err) {
  console.log('An error occurred: ' + err);
});

client.quit();

```

Listing 8.16 Verbindung zum Redis-Server aufbauen

Wie Sie Listing 8.16 entnehmen können, benötigen Sie für die eigentliche Verbindung lediglich zwei Kommandos. Das erste, `require`, dient dazu, das NPM-Paket in Ihre Applikation einzubinden. Mit einem Aufruf von `createClient` erzeugen Sie ein Objekt, über das Sie mit dem Redis-Server kommunizieren können.

Die Kommunikation erfolgt asynchron und eventbasiert. Das bedeutet konkret, dass Sie Callback-Funktionen auf verschiedene Ereignisse binden und damit jeweils auf das Ereignis reagieren können. In Listing 8.16 sehen Sie dies anhand des `error`-Events, das ausgelöst wird, falls ein Fehler bei der Verbindung mit dem Redis-Server auftritt. Weitere Ereignisse, auf die Sie reagieren können, sind `ready`, `connect`, `end`, `drain` und `idle`.

Sobald Sie die Verbindung zum Server nicht mehr benötigen, sollten Sie diese mit einem Aufruf der Methode `exit` oder `quit` beenden. Andernfalls wird die Verbindung offen gehalten und die Applikation nicht beendet. Der Unterschied zwischen beiden Methoden liegt darin, dass `exit` die Verbindung sofort beendet, egal ob noch Antworten vom Server ausstehen oder nicht. `quit` dagegen wartet, bis sämtliche Antworten vom Server vorliegen und beendet dann die Verbindung.

Datensätze anlegen

Mit der Methode `set` können Sie neue Datensätze in Redis anlegen. In Listing 8.17 sehen Sie, wie diese Methode konkret verwendet wird. Dieses Listing ist eine direkte Erweiterung von Listing 8.16. Achten Sie darauf, dass Sie den Aufruf von `quit` am Ende von Listing 8.16 auf jeden Fall entfernen, da ansonsten die Verbindung in der Callback-Funktion beendet wird.

```

client.set('node.js', 'Hello World', function (err, res) {
  if (err) throw err;
  console.log(res);
  client.quit();
});

```

Listing 8.17 Einen Datensatz in Redis anlegen

Die `set`-Methode erhält insgesamt drei Argumente. Das erste Argument ist eine Zeichenkette, die den Schlüssel angibt, über den Sie später wieder auf die Daten, die Sie im zweiten Argument angeben, zugreifen können. Das dritte Argument ist schließlich eine Callback-Funktion, die ausgeführt wird, sobald der Server eine Rückmeldung sendet. Diese Callback-Funktion erhält zwei Werte, der erste ist ein Fehlerobjekt, das im Standardfall den Wert `null` aufweist und nur im Fehlerfall weitere Informationen enthält. Der zweite Wert ist die Antwort des Servers. Gab es keine Probleme beim Einfügen der Daten, erhalten Sie hier die Zeichenkette `OK`.

In Listing 8.17 sehen Sie außerdem, dass innerhalb der Callback-Funktion die Methode `quit` aufgerufen wird, um die Verbindung nach erfolgreichem Einfügen der Daten zu beenden.

Haben Sie Daten eingefügt, möchten Sie diese irgendwann auch wieder auslesen. Sie sehen im folgenden Abschnitt, wie Sie mit wenig Aufwand auf Ihre Daten zugreifen können.

Datensätze auslesen

Ähnlich einfach wie das Erstellen ist auch das Auslesen von Datensätzen. Mit der `get`-Methode können Sie über einen Schlüssel auf einen gespeicherten Wert zugreifen, wie Sie auch in Listing 8.18 sehen können.

```
client.get('node.js', function (err, res) {
  if (err) throw err;
  console.log(res);
  client.quit();
});
```

Listing 8.18 Einen Wert aus Redis auslesen

Im Gegensatz zur `set`-Methode akzeptiert `get` lediglich zwei Eingabewerte. Der erste ist der Schlüssel des Werts, den Sie auslesen möchten, und der zweite eine Callback-Funktion, die ausgeführt werden soll, sobald das Ergebnis vom Server vorliegt.

Die Callback-Funktion erhält zwei Argumente. Das erste besteht aus einem Fehlerobjekt, und das zweite enthält die Antwort des Servers, bestehend aus dem Wert für den zuvor angegebenen Schlüssel. Innerhalb der Callback-Funktion können Sie wiederum die Verbindung beenden.

Datensätze aktualisieren

Möchten Sie Datensätze, die in Ihrer Datenbank liegen, modifizieren, können Sie hierbei auf zwei verschiedene Arten vorgehen. Wollen Sie den Datensatz grundlegend verändern, erreichen Sie dies, indem Sie den neuen Wert einfach per `set` speichern. Dies überschreibt den bereits bestehenden Datensatz in der Datenbank.

Möchten Sie allerdings lediglich Informationen an den Datensatz anhängen, können Sie die `append`-Methode verwenden. Listing 8.19 zeigt Ihnen in einem stark verkürzten Beispiel, wie Sie in einem solchen Fall vorgehen müssen.

```
client.set('node.js', 'Hello', function (err, res) {
  client.append('node.js', ' World', function (err, res) {
    client.get('node.js', function (err, res) {
      console.log(res);
      client.quit();
    });
  });
});
```

Listing 8.19 Informationen an einen bestehenden Datensatz anhängen

Die `append`-Methode wird wie `set` aufgerufen, nur dass sie den bestehenden Datensatz nicht ersetzt, sondern lediglich die Informationen anhängt.

Im Verlauf Ihrer Applikation kann es immer wieder vorkommen, dass Sie Daten aus Ihrer Datenbank wieder entfernen möchten. Der folgende Abschnitt beschäftigt sich damit, wie Sie Datensätze aus Ihrer Datenbank entfernen.

Datensätze entfernen

Der Befehlssatz des Redis-Servers enthält das Kommando `del`, mit dem Sie Einträge aus Ihrer Datenbank entfernen können. Listing 8.20 beschreibt den Einsatz der `del`-Methode anhand eines einfachen Beispiels.

```
client.del('node.js', function (err, res) {
  console.log(res);
});
```

Listing 8.20 Daten aus einer Redis-Datenbank löschen

Um einen Datensatz aus Ihrer Datenbank zu entfernen, übergeben Sie der `del`-Methode den Schlüssel des Datensatzes, den Sie löschen möchten. Als zweites Argument akzeptiert diese Methode eine Callback-Funktion, die ausgeführt wird, sobald die Löschung erfolgt ist. Sie erhalten hier Zugriff sowohl auf ein Fehlerobjekt als auch auf die Anzahl der Datensätze, die bei dieser Operation gelöscht wurden.

Diese Betrachtung von Redis und des Redis-Clients für Node.js ist lediglich ein kleiner Ausschnitt des Featuresets, das Ihnen in diesem Zusammenhang zur Verfügung steht. Über den Redis-Client können Sie auf den gesamten Befehlssatz, den Ihnen der Redis-Server bietet, zugreifen. So können Sie beispielsweise Hashes und Listen erstellen und mit diesen arbeiten. Sollten Sie sich näher mit Redis beschäftigen wollen, ist <http://redis.io> dafür ein guter Ausgangspunkt.

Mit MongoDB lernen Sie in den nächsten Abschnitten einen weiteren Vertreter nicht-relationaler Datenbanken kennen.

8.2.2 MongoDB

MongoDB ist ein Vertreter der dokumentenorientierten Datenbanken. Das bedeutet, dass die Grundlage der Speicherung von Informationen in dieser Datenbank auf Dokumenten basiert, die im BSON-Format vorliegen. Weitere Informationen zu diesem Format finden Sie unter <http://bsonspec.org/>.

MongoDB eignet sich aufgrund seiner guten Performance auch für größere Applikationen und bringt darüber hinaus einige weitere Features mit, die gerade bei sehr großen Datenmengen entscheidende Vorteile bringen.

MongoDB ist für die verschiedensten Betriebssysteme wie Linux, Windows, Solaris und Mac OS X verfügbar. Sie können MongoDB also auf allen Systemen installieren, auf denen auch Node.js lauffähig ist.

Installation

Sobald Sie die Serversoftware von MongoDB auf Ihrem System installiert haben, benötigen Sie lediglich noch einen Treiber, über den Sie auf Ihre Datenbank zugreifen können.

Für Node.js existiert ein in JavaScript implementierter Treiber als NPM-Paket, das Sie über das Kommando installieren können, das Sie in Listing 8.21 sehen.

```
$ npm install mongodb
```

Listing 8.21 Installation des MongoDB-Treibers

Im ersten Schritt erfahren Sie nun wie bei den vorangegangenen Datenbanken auch, wie Sie eine Verbindung zu Ihrer Datenbank herstellen können, über die Sie später dann Datensätze erzeugen, auslesen, modifizieren und wieder löschen können.

Verbindung aufbauen

Die Verbindung zur Datenbank basiert auf dem MongoDB-Client. Diesen müssen Sie korrekt initialisieren. Listing 8.22 beschreibt, wie Sie hierbei vorgehen müssen.

```
var Db = require('mongodb').Db,
    Server = require('mongodb').Server;

var client = new Db('test', new Server("127.0.0.1", 27017, {}), {w: 1});

client.open(function (err, client) {
```

```

    client.createCollection('Addresses', function (err, collection) {
    });
  });

```

Listing 8.22 Verbindung zur Datenbank aufbauen

Zum Aufbau der Verbindung benötigen Sie sowohl die `Db`-Klasse als auch die `Server`-Klasse des MongoDB-Clients. Das Client-Objekt erhalten Sie, indem Sie eine neue Instanz der `Db`-Klasse erzeugen. Der Konstruktor erhält den Namen der Datenbank, das Serverobjekt und die Optionen für die Verbindung. Rufen Sie schließlich die `open`-Methode des Clients auf, wird die Verbindung hergestellt. Eine weitere Besonderheit von MongoDB ist, dass die Dokumente, die Sie anlegen, nicht direkt in der Datenbank abgelegt werden, sondern in sogenannten Collections. Das sind Sammlungen von Dokumenten, die sich im weitesten Sinne mit Tabellen vergleichen lassen.

Eine Collection können Sie über das Client-Objekt mit einem Aufruf der Methode `createCollection` erzeugen. Als ersten Wert akzeptiert diese Methode den Namen der Collection. Der zweite Wert besteht aus einer Callback-Funktion, die für diese Collection ausgeführt werden soll. Alle im Folgenden vorgestellten Operationen werden innerhalb dieser Callback-Funktion ausgeführt.

Im nächsten Schritt sehen Sie, wie Sie neue Datensätze anlegen können.

Datensätze anlegen

Die Datensätze in MongoDB liegen in einem JSON-ähnlichen Format vor. Das bedeutet, dass Sie mit Node.js auf sehr einfache Art neue Datensätze erzeugen können. Der Client stellt Ihnen zu diesem Zweck die Methode `insert` zur Verfügung. Die Verwendung dieser Methode können Sie Listing 8.23 entnehmen.

```

var address = {
  street: 'Broadway 1',
  place: 'New York',
  country: 'United States of America'
};
collection.insert(address, {safe: true}, function (err, res) {
  if (err) throw err;
  console.log(res);
});

```

Listing 8.23 Datensätze in eine MongoDB einfügen

Die `insert`-Methode erhält als erstes Argument eine Objektstruktur, die den neuen Datensatz repräsentiert. Im Falle des Beispiels in Listing 8.23 ist dies eine Adresse. Das zweite Argument besteht aus einem Objekt mit Konfigurationsoptionen. Möchten

Sie im dritten Argument eine Callback-Funktion angeben, die nach erfolgter Einfügung ausgeführt werden soll, müssen Sie hier das Schlüssel-Werte-Paar `safe: true` angeben. Nur dann wird die Callback-Funktion erst nach dem Einfügen ausgeführt.

Die Callback-Funktion erhält als erstes Argument ein Fehlerobjekt und als zweites das Ergebnis der Einfügung. Das zweite Objekt besteht aus den Werten, die eingefügt wurden, plus der eindeutigen ID, die MongoDB diesem Datensatz zugewiesen hat.

Datensätze auslesen

Haben Sie erst einmal Datensätze in der Datenbank angelegt, können Sie diese auch wieder mit der `find`-Methode auslesen.

```
collection.find().toArray(function (err, docs) {
  if (err) throw err;
  console.log(docs);
});
```

Listing 8.24 Datensätze aus einer MongoDB auslesen

Auf die mit `createCollection` erstellte Sammlung von Dokumenten können Sie die `find`-Methode aufrufen, um dort nach Dokumenten zu suchen, die bestimmten Kriterien genügen. Übergeben Sie der `find`-Methode keinerlei Werte, werden sämtliche Dokumente ausgelesen. Sie können hier allerdings auch Abfragen definieren und so nur bestimmte Dokumente auswählen. So ist es beispielsweise möglich, nur Dokumente auszulesen, deren `street`-Schlüssel einen bestimmten Wert aufweist.

Auf das von `find` zurückgegebene Objekt können Sie die `toArray`-Methode aufrufen. Diese sorgt dafür, dass Ihnen in einer Callback-Funktion neben einem Fehlerobjekt ein Array mit allen gefundenen Dokumenten zur Verfügung steht.

Haben Sie auf diese Weise Dokumente gefunden, kann es sein, dass Sie eines dieser Dokumente anpassen möchten. Der nächste Abschnitt zeigt Ihnen, wie Sie hierbei vorgehen müssen.

Datensätze aktualisieren

Wie bei vielen anderen Datenbanken haben Sie auch bei MongoDB die Möglichkeit, die Werte bestehender Datensätze anzupassen. Mit der `update`-Methode des MongoDB-Clients können Sie dies auf eine sehr einfache Weise erreichen, wie Ihnen Listing 8.25 zeigt.

```
var newAddress =
  {street: 'Tower Hill', place: 'London', country: 'United Kingdom'};
collection.update({street: 'Broadway 1'}, {$set: newAddress}, {safe: true}, fu
```

```

action (err) {
    if (err) throw err;
});

```

Listing 8.25 Datensätze in einer MongoDB aktualisieren

Die `update`-Methode rufen Sie wie auch schon die `find`-Methode auf eine Collection auf. Im ersten Argument formulieren Sie mit einem Objekt eine Bedingung, die ein Datensatz erfüllen muss, damit er aktualisiert wird. Das zweite Argument gibt an, welche Werte ersetzt werden müssen. Mit dem Konfigurationsobjekt im dritten Argument können Sie mithilfe von `safe: true` die Callback-Funktion im vierten Argument aktivieren, die ausgeführt wird, sobald die Daten aktualisiert sind.

Datensätze entfernen

Sind Datensätze in Ihrer Datenbank veraltet oder müssen sie aus einem anderen Grund entfernt werden, können Sie dies durch einen Aufruf der `remove`-Methode erreichen. Listing 8.26 zeigt Ihnen die Verwendung dieser Methode.

```

collection.remove({street: 'Tower Hill'}, {safe: true}, function (err, res) {
    if (err) throw err;
});

```

Listing 8.26 Entfernen von Datensätzen aus der Datenbank

Die `remove`-Methode erhält als ersten Wert die Bedingung, mit der die Datensätze identifiziert werden, die gelöscht werden sollen.

MongoDB bietet eine Vielzahl weiterer Operationen, mit denen Sie die Datenspeicherung innerhalb Ihrer Applikation unterstützen können. Damit wird MongoDB zu einer hochperformanten Datenbank, wenn es darum geht, viele Daten in Form von Dokumenten zu speichern.

8.3 Zusammenfassung

Wie Sie im Verlauf dieses Kapitels sehen konnten, unterstützt Node.js durch die als NPM-Pakete verfügbaren Datenbanktreiber eine Vielzahl von Datenbanken. Dabei ist die Unterstützung nicht nur auf relationale Datenbanken beschränkt. Sie können stattdessen auch verschiedenste nicht-relationale Datenbanken wie Redis oder MongoDB verwenden.

Mithilfe der Treiber können Sie neben den grundlegenden Datenbankfunktionen wie dem Anlegen von Datensätzen und dem Auslesen, Modifizieren und Löschen von Datensätzen auch auf erweiterte Features der Datenbanken zurückgreifen.

Je nachdem, welche Aufgabe Sie mit Ihrer Applikation lösen müssen, haben Sie die freie Wahl, auf die dafür passende Datenbank zurückzugreifen.

Im nächsten Kapitel erfahren Sie mehr über verschiedene Qualitätssicherungsverfahren in Node.js-Applikationen wie beispielsweise die statische Codeanalyse oder die Absicherung Ihrer Software durch Unittests.

Kapitel 9

Qualitätssicherung

Es lohnt sich, die Entdeckungen anderer zu studieren, dass für uns selbst eine neue Quelle für Erfindungen entspringt.

– Gottfried Wilhelm Leibniz

Ein sehr wichtiges Feld in der Softwareentwicklung ist die Qualitätssicherung. Der Begriff der Qualitätssicherung ist eher schwammig und wird von jedem Entwickler anders ausgelegt. Jeder hat andere Ansprüche an sich und seine Software. Bei der Entwicklung von Applikationen gilt es daher, ein gemeinsames Verständnis für Softwarequalität zu erlangen. Softwarequalität bedeutet in den meisten Fällen, dass die erstellte Software anhand einer Reihe von Metriken gemessen wird. Zu diesen messbaren Kennzahlen kommen weitere, nicht oder schlechter greifbare Kriterien wie beispielsweise das Vorhandensein und die Qualität von Unittests.

Sie lernen in diesem Kapitel, wie Sie Unittests für Ihre Applikation erstellen können. Außerdem erfahren Sie mehr über die Überprüfung Ihres Quellcodes mit statischer Codeanalyse.

Schließlich lesen Sie zum Abschluss dieses Kapitels, wie Sie den Node.js Debugger als ein sehr wertvolles Werkzeug zum Auffinden von Softwarefehlern in Ihrer Applikation verwenden können.

9.1 Assertion Testing

Die Node.js-Plattform stellt Ihnen eine sehr einfache Infrastruktur zur Erstellung von Unittests für Ihre Applikation zur Verfügung. Unittests dienen dabei der Absicherung der Funktionsfähigkeit der Funktionen innerhalb Ihrer Applikation. Das bedeutet, dass Unittests auf Funktionsebene angewandt werden und die kleinsten Bestandteile Ihrer Applikation, also gewissermaßen die Basis, absichern. Erstellen Sie Unittests, sollten Sie ein paar Grundregeln beachten.

Anforderungen an Unittests

- ▶ Sie sollten Ihre Unittests so formulieren, dass sie isoliert voneinander sind. Das bedeutet, dass keine Abhängigkeiten zwischen den Tests bestehen und kein Test auf einem anderen aufbaut.
- ▶ Beachten Sie, dass pro Test lediglich ein Testfall abgedeckt werden soll. Ein Test ist nur für einen bestimmten Sachverhalt verantwortlich. Schlägt ein Test fehl, wissen Sie genau, aus welchem Grund.
- ▶ Unittests bedürfen keiner manuellen Eingriffe, damit sie laufen. Die einzige Interaktion, die Sie mit Ihren Tests haben sollten, sind der Start der Tests und das Überprüfen der Ergebnisse.
- ▶ Unittests dokumentieren Ihre Software. Schreiben Sie Ihre Tests stets so, dass ein anderer Entwickler, der diese Tests liest, aus ihnen ableiten kann, wie Sie erwarten, dass die Schnittstellen innerhalb Ihrer Applikation verwendet werden.
- ▶ Die Qualitätsmaßstäbe, die für Ihre Applikation gelten, sollten im selben Maß auch für Ihre Unittests gelten. Unittests leben genauso lange wie der eigentliche Quellcode Ihrer Applikation. Und wie auch der Applikationscode sollte sich auch der Code Ihrer Tests über den Zeitverlauf hinweg weiterentwickeln.
- ▶ Testen Sie nur den Code, der auch getestet werden muss. Getter- und Setter-Methoden, die keinerlei weitere Logik als das Auslesen oder Setzen von Objekteigenschaften beinhalten, müssen nicht zwangsläufig getestet werden.
- ▶ Während der Entwicklung führen Sie Ihre Unittests sehr häufig aus. Aus dieser Tatsache ergibt sich die Anforderung, dass die Tests schnell durchlaufen müssen, um schnelles Feedback über den Zustand Ihrer Applikation an Sie zurückzuliefern.

Wenn Sie Unittests für Ihre Applikation erstellen, sollten Sie darauf achten, dass der Code der Unittests und der Code Ihrer Applikation getrennt voneinander gespeichert werden. Im Normalfall legen Sie Ihre Tests in einer Verzeichnisstruktur in einem Verzeichnis mit dem Namen `test` ab. Damit ist jedem Entwickler, der Ihre Applikation betrachtet, klar, was er in diesem Verzeichnis finden wird. Diese Benennung hat sich mittlerweile als Standard etabliert. Auch der Quellcode der Node.js-Plattform selbst weist ein solches Verzeichnis auf. Der Quellcode von Node ist unter <http://nodejs.org/dist/> oder auf Github unter <https://github.com/joyent/node> verfügbar.

Die Tests der Node.js-Plattform selbst sind ebenfalls mit dem `assert`-Modul erstellt. Ein guter Ansatz, um die korrekte Formulierung von Unittests unter Node.js mit dem `assert`-Modul zu lernen, ist, wenn Sie sich den Quellcode der Node.js-Plattform herunterladen und sich die Tests im Verzeichnis `test/simple` ansehen. Hier existieren für nahezu alle Module und Features von Node.js Tests, die einerseits die Qualität der Plattform sicherstellen und andererseits die Verwendung der jeweiligen Funktionalität dokumentieren.

Möchten Sie selbst einen Unittest erstellen, verläuft dies immer nach dem gleichen Schema. Im ersten Schritt sorgen Sie für die korrekte Umgebung für den Test. Dieser Vorgang wird als Arrange bezeichnet und beinhaltet meist die Instanziierung von Objekten und das Setzen bestimmter Werte, die Voraussetzung für den Test sind. Der zweite Schritt besteht aus der eigentlichen Aktion, deren Auswirkungen Sie testen möchten. Diese Phase eines Unittests heißt Act. Zum Ende eines Tests müssen Sie nur noch überprüfen, ob die Aktion das gewünschte Ergebnis geliefert hat. Dies wird als Assert bezeichnet.

Assertions heißen auch die Methoden, die Ihnen Ihr Testframework, in diesem Fall das `assert`-Modul, bietet, um zu prüfen, ob ein Wert Ihren Erwartungen entspricht. Im Verlauf dieses Kapitels erfahren Sie noch mehr über die im `assert`-Modul enthaltenen Assertion-Methoden. Zunächst sehen Sie in Listing 9.1 ein sehr einfaches Beispiel eines Unittests, der mit den Mitteln des `assert`-Moduls erstellt wurde.

```
var assert = require('assert');

var myBuffer = new Buffer('Hello World');

var result = myBuffer.toString();

assert.equal(result, 'Hello World');
```

Listing 9.1 Beispiel eines Unittests mit dem `assert`-Modul

Der Aufruf der `require`-Methode ist erforderlich, damit Sie das `assert`-Modul benutzen können, und steht aus diesem Grund in keinem direkten Zusammenhang mit dem Unittest selbst. Die Instanziierung des `Buffer`-Objekts stellt den ersten Schritt (Arrange) des Tests dar. Hier sorgen Sie für die Grundlage Ihres Tests. Der nächste Schritt (Act) besteht aus einem einfachen Funktionsaufruf. Sie führen hier die `toString`-Methode des `Buffer`-Objekts aus und weisen das Ergebnis einer Variablen zu. Am Ende des Tests prüfen Sie schließlich mithilfe der `equal`-Methode des `assert`-Moduls, ob der Rückgabewert der `toString`-Methode auch tatsächlich dem Wert `Hello World` entspricht.

Der Test ist damit komplett, und Sie können ihn ausführen. Hierfür ist nichts weiter erforderlich, als dass Sie den Test, der in diesem Fall in der Datei `buffer.js` gespeichert ist, auf der Kommandozeile mit Node.js ausführen, wie Sie in Listing 9.2 sehen können.

```
$ node buffer.js
$
```

Listing 9.2 Ausführen eines `assert`-Tests

Sie sollten von der Ausgabe des Tests nicht verblüfft sein. Solange die Tests erfolgreich ablaufen, erhalten Sie keinerlei Ausgabe über den Verlauf. Anders ist der Fall, wenn Sie den Test etwas modifizieren, sodass er fehlschlägt. Also wenn Sie beispielsweise in der letzten Zeile des Tests die Zeichenkette `Hello Sun` statt `Hello World` schreiben, erhalten Sie die Ausgabe, die Sie in Listing 9.3 finden.

```
$ node buffer.js

assert.js:102
  throw new assert.AssertionError({
    ^
AssertionError: "Hello World" == "Hello Sun"
    at Object.<anonymous> (/srv/node/buffer.js:7:8)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Module.runMain (module.js:492:10)
    at process.startup.processNextTick.process._tickCallback (node.js:244:9)
$
```

Listing 9.3 Fehlschlagender Test

Das `assert`-Modul sorgt im Falle eines fehlschlagenden Tests dafür, dass ein Fehler vom Typ `AssertionError` geworfen wird. Die Zeichenkette, die das Fehler-Objekt enthält, liefert Rückschlüsse auf den Grund des Fehlschlags. Sie sollten beachten, dass die Ausführung des Tests beim ersten Auftreten eines Fehlers abbricht, es sei denn, Sie fangen den Fehler in einem `try-catch`-Block ab. So kann es vorkommen, dass durch eine Änderung am Quellcode nicht nur ein Test fehlerhaft ist, sondern mehrere, und Sie merken dies erst, sobald Sie den ersten Test repariert haben.

Wie bereits angekündigt, verfügt das `assert`-Modul über eine Reihe von Assertion-Methoden. Deren Namen und Bedeutung finden Sie in Tabelle 9.1.

Assertion	Bedeutung
<code>assert.fail(actual, expected, message, operator)</code>	Wirft einen Fehler und zeigt die Werte für <code>actual</code> und <code>expected</code> , getrennt durch den Wert von <code>operator</code> . Außerdem wird <code>message</code> ausgegeben.
<code>assert.ok(value, [message])</code>	Prüft, ob der übergebene Wert wahr ist. Die Prüfung ist nicht typensicher, also auch der Wert <code>1</code> ist beispielsweise wahr.

Tabelle 9.1 Assertion-Methoden des `assert`-Moduls

Assertion	Bedeutung
<code>assert.equal(actual, expected, [message])</code>	Diese Methode prüft, ob die beiden Werte gleich sind. Bei diesem Vergleich wird der <code>==</code> -Operator verwendet.
<code>assert.deepEqual(actual, expected, [message])</code>	Prüft, ob komplexe Strukturen wie Objekte gleich sind.
<code>assert.strictEqual(actual, expected, [message])</code>	Mithilfe dieser Methode können Sie typensichere Vergleiche durchführen. In diesem Fall wird der <code>===</code> -Operator verwendet.
<code>assert.throws(block, [error], [message])</code>	Diese Methode erwartet, dass im übergebenen Codeblock ein Fehler geworfen wird. Ist dies nicht der Fall, schlägt der Test fehl.
<code>assert.ifError(value)</code>	Der Aufruf dieser Assertion-Methode schlägt fehl, falls der übergebene Wert <code>true</code> ist.

Tabelle 9.1 Assertion-Methoden des `assert`-Moduls (Forts.)

Allen Assertion-Methoden können Sie eine, meist optionale, Nachricht übergeben. Diese Zeichenkette wird statt der Standardnachricht bei der Ausgabe des Fehlers bei einem Fehlschlag des Tests angezeigt.

Damit Sie in Ihren Tests nicht dafür sorgen müssen, dass Sie bestimmte Ausdrücke erst umständlich negieren, existieren für die Methoden `equal`, `deepEqual` und `strictEqual` bereits negierte Versionen. Zu diesem Zweck wird den Methoden einfach ein `not` im Namen vorangestellt. Das Gegenstück zu `strictEqual` ist also beispielsweise `notStrictEqual`. Diese Methode testet die Ungleichheit mit dem `!==`-Operator. Die Negierung der `throws`-Methode entspricht nicht diesem Schema, ihr Name lautet `doesNotThrow`. Die Wirkungsweise gleicht allerdings der der anderen Methoden. Sie sorgt dafür, dass der Test fehlschlägt, falls im übergebenen Codeblock ein Fehler geworfen wird.

Mit dem `assert`-Modul können Sie nicht nur einfache Vergleiche durchführen und einzelne Funktionen testen. Wie die Tests der Node.js-Plattform selbst beweisen, eignet sich das `assert`-Modul auch für das Testen komplexer Applikationen. Allerdings erschwert die Abwesenheit von Möglichkeiten zur Gruppierung der Tests und weiteren Features, die man von einem Unittest-Framework erwartet, die Formulierung von Unittests. Aus diesem Grund lernen Sie in den nun folgenden Abschnitten weitere Frameworks kennen, mit denen Sie die Unittests für Ihre Applikation formulieren können.

9.2 jasmine-node

Für clientseitiges JavaScript existieren zahlreiche Unittest-Frameworks. So wurde beispielsweise im Zuge von jQuery das Testframework QUnit entwickelt. Die Testframeworks lassen sich grob in zwei Kategorien einteilen. Zum einen existieren die Frameworks, die eng an eine JavaScript-Bibliothek oder ein Framework gebunden sind, und zum anderen existieren unabhängige Testframeworks, mit denen Sie Applikationen testen können, unabhängig vom verwendeten Framework. Jasmine ist ein Vertreter dieser zweiten Kategorie, und mit jasmine-node wurde dieses erfolgreiche Testframework auf Node.js portiert.

Jasmine ist kein typisches Unittest-Framework, es verfolgt stattdessen den Ansatz des Behavior-Driven Developments, kurz BDD. Der Kerngedanke von BDD ist, dass nicht nur Entwickler Tests formulieren können, sondern diese auch von Personen erstellt werden können, die nicht direkt in die Entwicklung eingebunden sind. Mit BDD testen Sie, im Gegensatz zu Unittests, nicht nur atomare Einheiten im Quellcode, sondern mehr das Verhalten bestimmter Komponenten Ihrer Applikation. Jasmine kann jedoch auch für klassisches Unittesting verwendet werden. Listing 9.4 zeigt Ihnen einen Unittest mit Jasmine, der dem Test aus Listing 9.1 gleicht.

```
describe("Buffer", function () {

    it("should return String representation if toString is called", function () {

        var myBuffer = new Buffer('Hello World');

        var result = myBuffer.toString();

        expect(result).toEqual('Hello World');
    });

});
```

Listing 9.4 Test in jasmine-node

Der Aufbau eines Jasmine-Tests unterscheidet sich grundlegend von einem Test des assert-Moduls. Jasmine bietet Ihnen die Möglichkeit, Ihre Tests logisch zu gruppieren. Zu diesem Zweck können Sie die `describe`-Methode verwenden. Das erste Argument, das diese Methode akzeptiert, ist eine Zeichenkette, die beschreibt, was in den folgenden Tests geprüft wird. Das zweite Argument ist eine Funktion, die dann die eigentlichen Tests enthält.

Die Testfunktionen werden durch einen Aufruf der `it`-Methode von Jasmine gekapselt. Das erste Argument ist auch hier eine kurze Beschreibung, was der Test prüft. Im

zweiten Argument formulieren Sie innerhalb einer Funktion den Test selbst. Auch hier gehen Sie wieder nach dem Schema Arrange, Act, Assert vor. Die beiden ersten Schritte sind die gleichen wie auch schon im `assert`-Modul. Sie instanziiieren ein Buffer-Objekt, rufen die `toString`-Methode auf und speichern das Ergebnis in einer Variablen. Für den Assert-Schritt bietet auch Jasmine eine Reihe von Assertion-Methoden, die Sie allerdings nicht direkt aufrufen können, sondern nur im Zusammenspiel mit der `expect`-Methode. Diese akzeptiert einen Wert als Argument und gibt ein Objekt zurück, auf dem Sie dann die verschiedenen Assertion-Methoden ausführen können.

Damit Sie nun den Test auch ablaufen lassen können, müssen Sie zuerst `jasmine-node` installieren. Es ist als NPM-Paket verfügbar und sollte global installiert werden, sodass Sie es systemweit als Kommandozeilenwerkzeug verwenden können. Listing 9.5 zeigt Ihnen die zur Installation erforderliche Kommandozeile.

```
npm install jasmine-node -g
```

Listing 9.5 Installation von jasmine-node

Auch bei der Organisation der Tests geht Jasmine seinen eigenen Weg. Hier wird empfohlen, dass Sie den Quellcode Ihrer Applikation in einem Verzeichnis mit dem Namen `src` verwalten und die Tests parallel dazu im Verzeichnis `spec` ablegen. Damit `jasmine-node` Ihre Tests finden und ausführen kann, müssen Sie die Namenskonvention beachten. Die Testdateien müssen auf `.spec.js` enden, ansonsten werden sie nicht ausgeführt. Sie sollten den Code aus Listing 9.4 also in einer Datei mit dem Namen `buffer.spec.js` speichern und diese im Verzeichnis `spec` ablegen. Wie das Ergebnis nach der Ausführung des Tests aussieht, zeigt Ihnen Listing 9.6.

```
$ jasmine-node spec
.

Finished in 0.009 seconds
1 test, 1 assertion, 0 failures

$
```

Listing 9.6 Ausführung des Tests mit jasmine-node

Grundsätzlich müssen Sie nicht mehr tun, als dem Aufruf von `jasmine-node` mitzugeben, an welcher Stelle die Tests liegen. Dann sucht das Programm in diesem Verzeichnis und allen darunter angelegten Verzeichnissen nach Dateien mit der Endung `spec.js` und führt sie als Tests aus.

Gegenüber dem `assert`-Modul von Node.js hat `jasmine-node` den klaren Vorteil, dass die Ausgabe für Sie als Nutzer um einiges informativer ist. Auch im Erfolgsfall erhalten Sie eine Rückmeldung über den Verlauf der Tests. Wie Sie in Listing 9.6 sehen

können, liefert Ihnen das Werkzeug folgende Informationen: die Zeit, die die Tests im aktuellen Durchlauf in Anspruch genommen haben, wie viele Tests durchgeführt wurden, wie viele Assertions in diesen Tests aufgerufen wurden und wie viele Fehlschläge aufgetreten sind.

Fügen Sie nun einen zweiten, fehlschlagenden Test hinzu, erhalten Sie eine Ausgabe wie in Listing 9.7.

```
$ jasmine-node spec
```

```
F.
```

```
Failures:
```

```
1) Buffer should return String representation if toString is called
   Message:
     Expected 'Hello Sun' to equal 'Hello World'.
   Stacktrace:
     Error: Expected 'Hello Sun' to equal 'Hello World'.
       at new jasmine.ExpectationResult (/usr/local/lib/node_modules/jasmine-
node/lib/jasmine-node/jasmine-2.0.0.rc1.js:102:32)
     ...
       at jasmine.Suite.execute (/usr/local/lib/node_modules/jasmine-node/lib/
jasmine-node/jasmine-2.0.0.rc1.js:2215:14)

Finished in 0.013 seconds
2 tests, 2 assertions, 1 failure
```

```
$
```

Listing 9.7 Fehlschlagender Test mit jasmine-node

Gerade bei fehlschlagenden Tests zeigt sich ein weiterer Vorteil von jasmine-node gegenüber dem `assert`-Modul. Schlägt ein Test fehl, werden die übrigen Tests trotzdem weiter ausgeführt. Zu Beginn der Ausgabe repräsentiert ein `». «` einen erfolgreichen Test. Ein `F` steht für einen fehlschlagenden Test.

Neben den Informationen, die Sie auch nach einem erfolgreichen Testdurchlauf erhalten, stehen Ihnen bei einem Fehlschlag noch weitere Informationen zur Verfügung. So wird zu jedem fehlschlagenden Test beispielsweise die Zeichenkette angezeigt, die den Test identifiziert und beschreibt, was dieser Test prüfen sollte. Außerdem wird die Erwartung, die Sie mit der `expect`-Methode und der Assertion formuliert haben, in Klartext ausgegeben. Schließlich erhalten Sie einen umfangreichen Stacktrace, der Ihnen weitere Aufschlüsse über die Herkunft des Problems liefert.

Damit Sie die Tests besser organisieren können, erlaubt Ihnen Jasmine, mehrere Aufrufe von `describe` ineinanderzuschachteln und so die Struktur Ihrer Applikation nachzubilden.

Im Beispiel in Listing 9.4 konnten Sie bereits den Einsatz von Assertion-Methoden kennenlernen. Im Gegensatz zum `assert`-Modul können Sie Assertions nicht direkt benutzen, sondern müssen zuerst die `expect`-Methode aufrufen. Jasmine stellt Ihnen einige Assertions mehr zur Verfügung, als dies beim `assert`-Modul der Fall ist. In Tabelle 9.2 sehen Sie einen Überblick über die wichtigsten Assertions.

Assertion	Bedeutung
<code>toBe(value)</code>	Diese Methode vergleicht typsicher mit dem <code>===</code> -Operator.
<code>toEqual(value)</code>	Prüft, ob der Eingabewert von <code>expect</code> dem angegebenen Wert entspricht. Es wird nicht typsicher mit dem <code>===</code> -Operator verglichen.
<code>toMatch(regex)</code>	Vergleicht die Eingabe mit einem regulären Ausdruck.
<code>toBeDefined()</code>	Prüft, ob der Eingabewert nicht dem <code>undefined</code> -Wert entspricht.
<code>toBeNull()</code>	Prüft, ob der eingegebene Wert dem Wert <code>null</code> entspricht.
<code>toBeTruthy()</code> , <code>toBeFalsy()</code>	Diese Funktion prüft, ob ein Wert nach einem Cast nach Boolean dem Wert <code>true</code> beziehungsweise <code>false</code> entspricht.
<code>toContain(value)</code>	Prüft, ob ein Eingabearray das in <code>value</code> angegebene Element enthält.
<code>toThrow()</code>	Erwartet, dass die Funktion, die an <code>expect</code> übergeben wird, einen Fehler wirft.

Tabelle 9.2 Assertions in jasmine-node

Neben diesen Assertions, die unter Jasmine im Übrigen als Matcher bezeichnet werden, existieren noch einige weitere Methoden, mit deren Hilfe Sie einen Wert mit einer bestimmten Erwartung vergleichen können. Weitere Informationen zu Jasmine finden Sie unter <http://pivotal.github.io/jasmine/>.

Wie auch schon im `assert`-Modul existieren auch für Jasmine Negierungen für die einzelnen Matcher. In diesem Fall müssen Sie allerdings keine andere Methode aufrufen, sondern lediglich ein `.not` vor den Methodenaufruf setzen. Listing 9.8 verdeutlicht Ihnen diese Vorgehensweise.

```
describe("Negation", function () {
  it("should work in positive case", function () {
    expect('Node.js').toEqual('Node.js');
```

```

    });

    it("should work in negative case", function () {
        expect('Node.js').not.toEqual('jQuery');
    });
});

```

Listing 9.8 Negierung in Jasmine-Tests

Mit den Matchern, die in Jasmine enthalten sind, können Sie die meisten Fälle, auf die Sie in Ihren Unittests stoßen werden, abdecken. Sollten Sie allerdings einmal einen Spezialfall vorfinden, der nur sehr umständlich mit den Standard-Matchern gelöst werden kann, können Sie auch eigene Matcher definieren. Dies lohnt sich vor allem dann, wenn der Fall, für den Sie den Matcher benötigen, mehrmals in Ihren Tests auftritt. Dies gilt vor allem, wenn Sie bestimmte Operationen auf von Ihnen definierten Klassen von Objekten testen möchten.

Neben den Matcher-Methoden implementiert Jasmine mit den Spys ein weiteres bedeutendes Feature, wenn es um das Testen von Applikationen geht. Ein Spy stellt ganz allgemein einen Wrapper um Funktionen dar. Mit diesem Wrapper können Sie im einfachsten Fall den Aufruf einer Methode überwachen.

```

describe("Spy", function () {

    var buffer;

    beforeEach(function () {
        buffer = new Buffer('Hello World');
    });

    it("should track calls to 'toString'", function () {
        spyOn(buffer, 'toString');

        buffer.toString();

        expect(buffer.toString).toHaveBeenCalled();
    });

    it("should return a different result", function () {
        spyOn(buffer, 'toString').andReturn('Hello Node.js');

        var result = buffer.toString();

        expect(buffer.toString).toHaveBeenCalled();
    });
});

```

```

        expect(result).toEqual('Hello Node.js');
    });
});

```

Listing 9.9 Spys in Jasmine

Im ersten Test in Listing 9.9 sehen Sie, wie Sie mit der `spyOn`-Methode einen Wrapper um die `toString`-Methode des `buffer`-Objekts erstellen können. Dieser Wrapper sorgt dann dafür, dass Sie über die `toHaveBeenCalled`-Methode prüfen können, ob die `toString`-Methode durch den Test aufgerufen wurde. Diese Funktionalität ist bei direkten Aufrufen wie im Beispiel nicht wirklich interessant. Möchten Sie allerdings komplexere Workflows testen, bei denen eine Methode weitere Methoden aufruft, wird diese Art der Prüfung wieder relevant.

Der zweite Test aus Listing 9.9 stellt Ihnen ein weiteres Feature der Spys von Jasmine vor. Sie können mithilfe der Spys den Rückgabewert einer Methode beeinflussen. Zu diesem Zweck rufen Sie auf dem Rückgabewert der `spyOn`-Methode die Methode `andReturn` auf. Dieser übergeben Sie schließlich den Wert, den diese Methode ausgeben soll. So können Sie, wie Sie im Beispiel sehen, die `toString`-Methode dahingehend modifizieren, dass diese eine andere Zeichenkette zurückgibt.

Das Beispiel stellt Ihnen noch ein weiteres Feature von Jasmine vor. Wenn Sie Ihre Tests nach dem Schema Arrange – Act – Assert aufbauen, werden Sie in den meisten Fällen die erste Phase zumindest für eine Reihe von Tests einfach kopieren, um die korrekte Umgebung für Ihre Tests zu initialisieren. Ähnliches gilt für eventuelle Aufräumarbeiten nach den Tests. Aus diesem Grund bietet Jasmine die beiden Methoden `beforeEach` und `afterEach`. Sie definieren in beiden jeweils eine Funktion, die vor beziehungsweise nach jedem Test ausgeführt wird.

Die hier vorgestellten Features stellen lediglich einen Teil des Funktionsumfangs von Jasmine dar. Möchten Sie tiefer in diese Materie einsteigen, möchte ich Sie an die Online-Dokumentation von Jasmine verweisen. Sie sollten allerdings beachten, dass Jasmine ein Framework für Frontend-Testing ist und `jasmine-node` lediglich eine Portierung von Jasmine für Node.js darstellt.

Im nächsten Abschnitt lernen Sie mit `nodeunit` ein weiteres Testframework für Node.js kennen.

9.3 nodeunit

Im Gegensatz zu `jasmine-node`, das nur eine Portierung des clientseitigen Jasmine-Testframeworks darstellt, ist `nodeunit` exklusiv für die Node.js-Plattform entwickelt worden. Bereits die Formulierung der Tests deutet darauf hin. In Listing 9.10 sehen

Sie einen ersten Test. Wie auch schon bei jasmine-node gleicht der Test dem ursprünglichen Test, den Sie bereits für das `assert`-Modul erstellt haben.

```
exports.testBufferToString = function (test) {
  var myBuffer = new Buffer('Hello World');

  var result = myBuffer.toString();

  test.equal(result, 'Hello World');

  test.done();
};
```

Listing 9.10 Test in nodeunit

Die Tests für nodeunit sind wie ganz gewöhnliche Node.js-Module aufgebaut. Sie exportieren einfach über das `exports`-Objekt Ihre Testfunktionen. Der Name, unter dem Sie die Funktion exportieren, wird später in der Ausgabe angezeigt und dient als Referenz auf den Test, sodass Sie auf einen Blick feststellen können, welcher Test erfolgreich war und welcher fehlgeschlagen ist. Aus diesem Grund sollten Sie hierfür kurze, aber beschreibende Namen wählen. Die Funktion erhält ein Argument, das Ihnen innerhalb der Funktion als Basis zur Formulierung Ihrer Tests dient und über das Sie Zugriff auf die Methoden von nodeunit erhalten. Normalerweise sollten Sie dieses Argument mit dem Namen `test` versehen.

Der Aufbau des Tests gleicht dem der beiden anderen Frameworks. Auch hier bereiten Sie die Umgebung des Tests im ersten Schritt mit der Instanziierung der benötigten Objekte vor und führen im zweiten Schritt die zu testende Aktion aus. Der dritte Schritt besteht aus einer Assertion, mit der Sie prüfen, ob die durchgeführte Aktion zum gewünschten Ergebnis geführt hat. Sie sollten jeden Test mit einem Aufruf der `done`-Methode abschließen.

Damit Sie den Test ausführen können, benötigen Sie nodeunit. Dieses Kommandozeilenwerkzeug auf Basis von Node.js ist als NPM-Paket verfügbar und sollte wie auch schon jasmine-node global installiert werden, damit Sie es systemweit einsetzen können. Listing 9.11 zeigt die zur Installation erforderliche Kommandozeile.

```
npm install nodeunit -g
```

Listing 9.11 Installation von nodeunit

Nach der Installation können Sie nodeunit ähnlich verwenden wie schon jasmine-node. Sie geben das nodeunit-Kommando und die Datei, die Ihre Tests enthält, an, und nodeunit kümmert sich selbstständig um die Ausführung der Tests. Alternativ können Sie auch ein ganzes Verzeichnis angeben, und nodeunit führt sämtliche darin ent-

haltenen Tests aus. Lassen Sie nun den Code aus Listing 9.10 ablaufen, erhalten Sie eine Ausgabe, die ähnlich wie in Listing 9.12 aussehen sollte.

```
$ nodeunit test/

buffer
✓ testBufferToString
```

OK: 1 assertions (2ms)

Listing 9.12 Ausführung von Tests mit nodeunit

nodeunit liefert Ihnen nach der Ausführung wertvolle Informationen über den Verlauf der Tests. Die Zeichenkette `buffer` stellt im Beispiel den Namen der Datei dar, die die Tests beinhaltet. Liegen im angegebenen Verzeichnis mehrere Dateien, können Sie anhand des angezeigten Namens unterschieden werden. Unterhalb des jeweiligen Dateinamens werden die einzelnen Aufrufe der Testfunktionen mit dem jeweiligen Status und dem von Ihnen gewählten Namen aufgelistet. Ein erfolgreicher Test ist durch ein ✓-Symbol gekennzeichnet. Ein fehlschlagender Test wird durch ein ✗-Symbol gekennzeichnet. Außerdem ist die Schrift der entsprechenden Zeile rot gefärbt. Am Ende der Ausgabe erfahren Sie zudem, wie viele Assertions ausgeführt wurden und wie viel Zeit die Tests in Anspruch genommen haben.

Listing 9.13 zeigt Ihnen die Ausgabe, die Sie erhalten, falls einer oder mehrere Tests fehlschlagen. Wie auch schon in jasmine-node führt der Fehlschlag eines Tests nicht zum Abbruch des gesamten Testlaufs.

```
$ nodeunit test

buffer
✓ testBufferToString
✗ testBufferToStringFail

AssertionError: 'Hello World' == 'Hello Sun'
    at Object.assertWrapper [as equal] (/usr/local/lib/node_modules/nodeunit/
lib/types.js:83:39)
    at Object.exports.testBufferToStringFail (/srv/node/test/buffer.js:16:10)
    ...
    at _concat (/usr/local/lib/node_modules/nodeunit/deps/async.js:510:17)
```

FAILURES: 1/2 assertions failed (5ms)

Listing 9.13 Fehlschlagender nodeunit-Test

In der Ausgabe, die der Testlauf in Listing 9.13 zurückliefert, sehen Sie, dass Sie neben den bereits erwähnten Informationen auch einen detaillierten Stacktrace für jeden fehlschlagenden Test erhalten, der Ihnen Rückschlüsse auf die Ursache liefert.

Das Herzstück der Tests unter `nodeunit` sind wie schon bei `jasmine-node` und den Tests mit dem `assert`-Modul die Assertions. `nodeunit` greift hierbei auf die bestehenden Assertions des `assert`-Moduls zurück. Das bedeutet für Sie konkret, dass Ihnen sämtliche Assertions aus Tabelle 9.1 zur Formulierung von Tests zur Verfügung stehen.

Wie Sie schon im Beispiel in Listing 9.10 sehen konnten, müssen Sie für den Zugriff auf die Assertion-Methoden nicht erst das `assert`-Modul einbinden, sondern können diese direkt über das Argument der Test-Methode verwenden.

In `nodeunit` gibt es mehrere Möglichkeiten, wie Sie Ihre Tests organisieren. Die einfachste Variante ist, dass Sie zusammengehörige Tests in einer Datei gruppieren. So erzeugen Sie in der Ausgabe eine visuelle Gruppierung und können außerdem diese Gruppe von Tests getrennt von den übrigen Tests Ihres Systems ablaufen lassen. Eine weitere Art der Gruppierung von Tests besteht in der Gruppierung einzelner Testfunktionen innerhalb einer Datei.

```
exports.withoutGroup = function (test) {
  var buffer = new Buffer('Hello World');

  var result = buffer.toString();

  test.equals(result, 'Hello World');
  test.done();
};

exports.buffer = {

  setUp: function (callback) {
    this.buffer = new Buffer('Hello World');
    callback();
  },

  toString: function (test) {
    var result = this.buffer.toString();

    test.equals(result, 'Hello World');
    test.done();
  }
};
```

```

exports.path = {
  normalize: function (test) {
    var path = '/tmp/../../etc/';

    var result = require('path').normalize(path);

    test.equals(result, '/etc/');
    test.done();
  }
};

```

Listing 9.14 Gruppierung von nodeunit-Tests

Sie können Tests gruppieren, indem Sie statt einer Funktion eine Objektstruktur exportieren. Sie können pro Datei mehrere dieser Gruppen definieren. Die Ausgabe, die Sie erhalten, wenn Sie die Tests aus Listing 9.14 ausführen, finden Sie in Listing 9.15.

```
$ nodeunit group.js
```

```

group.js
✓ withoutGroup
✓ buffer - toString
✓ path - normalize

```

```
OK: 3 assertions (2ms)
```

Listing 9.15 Ausgabe bei der Ausführung gruppierter Tests

Der erste Test, der ausgeführt wird, ist nicht Teil einer Gruppe und wird aus diesem Grund auch normal dargestellt. Den Tests einer Gruppe werden jeweils der Gruppenname und ein Bindestrich vorangestellt. Die Tests einer Datei werden trotz Gruppierung bei der Ausgabe der Datei zugeordnet. Sie gewinnen mit der Gruppierung von Tests also eine weitere Ebene der Strukturierung hinzu.

Im Quellcode von Listing 9.14 sehen Sie ein weiteres Feature von nodeunit: die Methoden `setUp` und `tearDown`. Sie dienen wie die gleichnamigen Methoden in `jasmine-node` dazu, die Umgebung vor der Ausführung eines Tests vorzubereiten beziehungsweise danach wieder aufzuräumen. Beachten Sie bei der Implementierung der `setUp`- und `tearDown`-Methode, dass Sie als Argument eine Funktion erhalten. Sie müssen diese Funktion in jedem Fall innerhalb der `setUp`- beziehungsweise `tearDown`-Methode aufrufen. Führen Sie diese Funktion nicht aus, erhalten Sie die Meldung `FAILURES: Undone tests (or their setups/teardowns):`. Alle nachfolgenden Tests werden nicht ausgeführt. Gleiches gilt für den Fall, dass Sie die `done`-Methode am Ende eines Tests nicht aufrufen.

Mit dem nodeunit-Framework erhalten Sie ein Werkzeug, mit dem Sie die Methoden Ihrer Applikation testen und absichern können. In den nächsten Abschnitten sehen Sie anhand eines praktischen Beispiels, wie Sie nodeunit in Ihrer Applikation verwenden können.

9.4 Praktisches Beispiel von Unittests mit nodeunit

Unittests werden auf Funktionsebene erstellt, sie sichern also damit die Bausteine Ihrer Applikation ab. In der Praxis bedeutet das, dass Sie eine Funktion mit bestimmten Parametern aufrufen und dann den Rückgabewert dieser Funktion mit dem Wert vergleichen, von dem Sie erwarten, dass er zurückgeliefert wird. Für die Entwicklung der Funktion bedeutet dies wiederum, dass sie so wenige Abhängigkeiten wie möglich nach außen haben sollte. Sind Abhängigkeiten erforderlich, sollten diese über die Eingabewerte an die Funktion übergeben werden.

Sie sollten also bei der Implementierung Ihres Quellcodes darauf achten, dass der Code testbar ist. Das heißt: Eingaben sollten möglichst wenig über globale Variablen oder Objekteigenschaften erfolgen, sondern explizit als Eingabewerte an die Funktion übergeben werden. Das erleichtert Ihnen die Vorbereitung der Umgebung in der Arrange-Phase des Tests. Diese Technik der Umsetzung wird als Dependency Injection bezeichnet und bedeutet lediglich, dass externe Abhängigkeiten einer Funktion explizit übergeben werden.

Ähnliches wie für die Eingaben gilt auch für die Ausgaben einer Funktion. Die Rückgabe einer Funktion dient Ihnen als Indikator, ob die Funktion ihre Arbeit ordnungsgemäß verrichtet hat. Bestes Beispiel hierfür sind mathematische Funktionen. Sie akzeptieren einen oder mehrere Werte und transformieren diese anhand eines gegebenen Algorithmus in eine Ausgabe. Um die Testbarkeit Ihres Codes zu verbessern, sollten Sie also stets aussagekräftige Rückgabewerte verwenden. Seiteneffekte einer Funktion lassen sich teilweise nicht vermeiden, vor allem nicht, wenn sie als Methode innerhalb eines Objektkontexts verwendet werden und das Objekt selbst modifizieren. Ein Beispiel für diesen Fall sind Setter-Methoden, die eine Eigenschaft eines Objekts setzen. Hier wird das Testen etwas aufwendiger, da Sie nicht mehr nur die Ein- und Ausgabe betrachten dürfen, sondern das Objekt, das verändert wurde, prüfen müssen.

Möchten Sie Ihre Node.js-Applikation mit Unittests absichern, hat das zur Konsequenz, dass Sie sich schon vor der eigentlichen Implementierung des Quellcodes Ihrer Applikation Gedanken über die Testbarkeit Ihrer Funktionen machen sollten.

Eine sehr praktikable Art, Tests für Ihre Applikation zu erstellen, ist, wenn Sie anstatt zuerst den Quellcode zu schreiben und im Anschluss daran die Funktionalität zu tes-

ten, die Tests im Idealfall vor dem eigentlichen Quellcode umsetzen. Diese Vorgehensweise heißt testgetriebene Entwicklung und bietet eine Reihe von Vorteilen. Die wichtigsten sind, dass Sie sich mit dem zu lösenden Problem auseinandersetzen müssen, um es als Unittest formulieren zu können, und dass Sie, wenn Sie so vorgehen, eine sehr hohe Testabdeckung Ihrer Anwendung erreichen.

Ein kleines Beispiel soll Ihnen diese Vorgehensweise verdeutlichen. Angenommen, Sie entwickeln eine Applikation, in der mit Produkten umgegangen werden muss, die in verschiedenen Kategorien organisiert sind. Die Produktdaten erhält Ihre Applikation im JSON-Format von einem Webservice. Ihre Aufgabe besteht nun darin, aus der Datenstruktur der Produkte die vorhandenen Kategorien zu extrahieren. Die Liste der Kategorien soll dabei aufsteigend nach ihrer Position sortiert werden.

Gehen Sie testgetrieben vor, erstellen Sie aus diesen Angaben zuerst einen Test. In Listing 9.16 sehen Sie, wie ein derartiger Test aussehen könnte.

```
var product = require('../src/product.js');

exports.testGetCategoriesFromJSON = function (test) {
  var input = [{
    "id": "800001",
    "name": "Papier A4",
    "category": {"id": 1, "name": "Papier", "position": "3"}
  }, {
    "id": "90273",
    "name": "Kugelschreiber",
    "category": {"id": 3, "name": "Schreibmaterial", "position": "1"}
  }];

  var output = product.getCategories(input);

  test.deepEqual(output, ['Schreibmaterial', 'Papier']);
  test.done();
};
```

Listing 9.16 Erster Schritt der testgetriebenen Entwicklung

Führen Sie diesen Test nun aus, schlägt dieser erwartungsgemäß fehl, da noch kein Quellcode existiert, der getestet werden könnte.

```
$ nodeunit test/product.js
```

```
product.js
✖ testGetCategoriesFromJSON
```

```
TypeError: Object #<Object> has no method 'getCategories'
...
```

Listing 9.17 Fehlschlagender Test

Ihr Ziel sollte es nun nicht sein, die Logik zu implementieren, die dafür sorgt, dass der Test erfolgreich durchläuft. Stattdessen sollten Sie lediglich dafür sorgen, dass der Test erfolgreich ist, und zwar, indem Sie die `getCategories`-Methode das geforderte Array zurückgeben lassen. Mit diesem Vorgehen verfolgen Sie gleich mehrere Schritte. Sie sollten sich stets darum kümmern, dass Ihre Tests erfolgreich sind, und Sie sollten dazu in möglichst kleinen Schritten vorgehen. In Listing 9.18 sehen Sie die Implementierung, die Ihren Test erfolgreich ablaufen lässt.

```
exports.getCategories = function (data) {
  return ['Schreibmaterial', 'Papier'];
};
```

Listing 9.18 Einfache Implementierung von `getCategories`

Sollten Sie immer die gleichen Werte an Ihre `getCategories`-Methode übergeben, wären Sie mit dieser Implementierung am Ende Ihrer Problemlösung. Lassen Sie Ihren Test durchlaufen, erhalten Sie eine Erfolgsmeldung. Da Sie in der Realität nicht davon ausgehen können, dass der Webservice immer die zwei gleichen Produkte liefert, müssen Sie einen weiteren Test schreiben.

```
exports.testGetFurtherCategoriesFromJSON = function (test) {
  var input = [{
    "id": "600320",
    "name": "Radiergummi",
    "category": {"id": 1, "name": "Zubehör", "position": "4"}
  }, {
    "id": "90273",
    "name": "Kugelschreiber",
    "category": {"id": 3, "name": "Schreibmaterial", "position": "1"}
  }];

  var output = product.getCategories(input);

  test.deepEqual(output, ['Schreibmaterial', 'Zubehör']);
  test.done();
};
```

Listing 9.19 Testgetriebene Entwicklung, ein zweiter Test

Der Test in Listing 9.19 gleicht dem ersten Test bis auf die Unterschiede in den Eingabedaten und den Ausgabedaten. Führen Sie die Tests erneut aus, wird Ihnen eine Meldung präsentiert, die besagt, dass zwar der erste Test fehlerfrei durchläuft, der zweite allerdings fehlschlägt. Hier erhalten Sie eine Fehlermeldung, die Sie darüber informiert, dass der erwartete Wert und der übergebene Wert einander nicht entsprechen. Der nächste Schritt besteht nun aus der Implementierung der Logik, die dafür sorgt, dass beide Tests erfolgreich durchlaufen. Einen Vorschlag für eine derartige Implementierung finden Sie in Listing 9.20.

```
exports.getCategories = function (data) {
  var categories = [];
  for (var i = 0; i < data.length; i++) {
    categories[data[i].category.position] = data[i].category.name;
  }
  return categories.filter(function () {return true;});
};
```

Listing 9.20 Implementierung der getCategories-Methode

Führen Sie nun die Tests aus, erhalten Sie eine Ausgabe, die der in Listing 9.21 ähneln sollte.

```
$ nodeunit test/product.js
```

```
product.js
```

```
✓ testGetCategoriesFromJSON
✓ testGetFurtherCategoriesFromJSON
```

```
OK: 2 assertions (3ms)
```

Listing 9.21 Ausgabe erfolgreicher Tests

Wichtig ist in diesem Zusammenhang, dass Sie Ihre bestehenden Tests sehr häufig ausführen, um eine schnelle Rückmeldung über die Auswirkungen Ihrer Änderungen zu erhalten. Kombiniert mit sehr kleinen Implementierungsschritten, bei denen Sie stets die einfachste Lösung anstreben sollten, sorgt testgetriebene Entwicklung für qualitativ sehr hochwertigen Code. Diese Codequalität schlägt sich auch in den meisten Fällen in einer sehr geringen Anzahl von Fehlern in der Software nieder.

Die Qualitätssicherung endet jedoch nicht an Funktionsgrenzen oder mit dem Einsatz von Unittests. Die folgenden Abschnitte widmen sich weiteren Bereichen der Qualitätssicherung. Konkret erfahren Sie zunächst, wie Sie Ihren Quellcode mit verschiedenen Werkzeugen automatisiert analysieren lassen und ihn mit den dadurch gewonnenen Informationen verbessern können.

9.5 Statische Codeanalyse

Ein Kriterium für die Qualität einer Softwarelösung ist das Vorhandensein von Unit-tests. Dies ist jedoch nicht das einzige Mittel, mit dem Sie bewerten können, ob der Quellcode einer Software einem gewissen Standard entspricht. Das Feld der statischen Codeanalyse beschäftigt sich mit der Struktur des Quellcodes. Der Gedanke hinter diesem Konzept ist, dass Sie Ihre Qualitätsansprüche an eine Software in konkrete Maßzahlen übersetzen und den Programmcode mit dieser Erwartung vergleichen. Führen Sie diesen Vergleich regelmäßig durch, wissen Sie, auf welchem Stand Ihre Software ist. Je nachdem, ob Ihr Programmcode den Maßstäben genügt, die Sie ansetzen, oder nicht, haben Sie die Möglichkeit, verschiedene Maßnahmen zu ergreifen, um die Qualität auf einen zufriedenstellenden Stand zu bringen und diesen auch über einen längeren Zeitraum zu halten.

Werkzeuge zur statischen Codeanalyse existieren für fast jede Sprache, angefangen bei klassischen Sprachen wie C oder Java bis hin zu Websprachen wie PHP oder JavaScript. In diesem Zusammenhang gibt es nicht das eine Werkzeug, mit dem Sie Ihre gesamte Applikation analysieren können und das Ihnen eine Zusammenfassung des Qualitätsstands liefert, stattdessen existiert eine Reihe von spezialisierten Werkzeugen, die jeweils nur bestimmte Aspekte betrachten. Im weiteren Verlauf dieses Kapitels lernen Sie die beiden Werkzeuge JSLint und PMD CPD kennen.

9.5.1 JSLint

JavaScript hat, wie jede andere Programmiersprache auch, seine Vor- und Nachteile. Douglas Crockford, der Entwickler des JSON-Formats, hat sich zum Ziel gesetzt, JavaScript zu verbessern. Zu diesem Zweck hat er das Programm JSLint entwickelt. Es durchsucht Ihren Quellcode nach bekannten Antipatterns und weist Sie darauf hin. Die Muster, die JSLint sucht, stellen bekannte Schwächen von JavaScript dar. Außerdem werden Sie auf Codestellen aufmerksam gemacht, die häufig zu Fehlern in Applikationen führen.

Da JSLint JavaScript im Allgemeinen prüft, kann es nicht nur für clientseitiges JavaScript, sondern auch für serverseitige Implementierungen wie Node.js eingesetzt werden. Wie viele Werkzeuge für Node.js steht auch JSLint als NPM-Paket im npmjs.org-Repository zur Verfügung. Mit der Kommandozeile aus Listing 9.22 können Sie JSLint auf Ihrem System installieren.

```
sudo npm install jslint -g
```

Listing 9.22 Installation von JSLint

Nach der systemweiten Installation von JSLint können Sie dieses Werkzeug auf der Kommandozeile verwenden, um Ihren Quellcode überprüfen zu lassen. Listing 9.23

zeigt Ihnen eine Funktion, die mithilfe von JSLint auf potenzielle Probleme hin analysiert werden soll.

```
function example(input) {
  var i;
  if (input.length < 2) {
    throw new Error('Too few characters');
  }

  for (i = 0; i < 10; i++) {
    output += i + '. Output: ' + input;
  }

  var output;

  return output;
  output += 'result';
}
```

Listing 9.23 Funktion, die mit JSLint analysiert werden soll

Gehen Sie vom Quellcode in Listing 9.23 aus, speichern ihn in einer Datei mit dem Namen `jslint.js` und führen JSLint auf dieser Datei aus, erhalten Sie die Ausgabe, die Sie in Listing 9.24 sehen können.

```
$ jslint jslint.js
```

```
jslint.js
#1 Missing 'use strict' statement.
  var i; // Line 2, Pos 1
#2 Expected 'var' at column 5, not column 1.
  var i; // Line 2, Pos 1
#3 Unexpected '++'.
    for (i = 0; i < 10; i++) { // Line 7, Pos 26
#4 'output' was used before it was defined.
    output += i + '. Output: ' + input; // Line 8, Pos 9
#5 Combine this with the previous 'var' statement.
    var output; // Line 11, Pos 9
#6 'output' was used before it was defined.
    var output; // Line 11, Pos 9
#7 'output' was used before it was defined.
    return output; // Line 13, Pos 12
#8 Unreachable 'output' after 'return'.
    output += 'result'; // Line 14, Pos 5
```

Listing 9.24 Ausgabe von JSLint

Das erste Problem, das JSLint im Quellcode findet, ist das Fehlen eines `use strict`-Statements, das den Strict Mode von ECMAScript 5 aktiviert. In diesem Ausführungsmodus werden einige Nachlässigkeiten von JavaScript, die bisher ohne weitere Fehlermeldungen akzeptiert wurden, korrigiert und werfen nun Fehler. So können beispielsweise keine globalen Variablen mehr versehentlich erzeugt werden. Befindet man sich innerhalb einer Funktion und schreibt den Namen einer Variablen falsch, was eine neue globale Variable erzeugen würde, wird ein `ReferenceError` geworfen.

Verwenden Sie Variablen innerhalb Ihres Quellcodes, bevor Sie diese definieren, macht Sie JSLint auch auf diesen Umstand aufmerksam. Normalerweise sollten Sie alle Variablendefinitionen zu Beginn einer Funktion durchführen, da JavaScript sämtliche Definitionen innerhalb einer Funktion sammelt und sie vor der Ausführung der eigentlichen Funktionslogik durchführt.

JSLint achtet allerdings nicht nur auf die syntaktische Korrektheit Ihres Quellcodes und weist Sie auf dessen potenzielle Schwachstellen hin. Dieses Werkzeug schreibt auch eine bestimmte Art der Quellcodeformatierung vor. So sehen Sie in Listing 9.23 beispielsweise, dass die Definition der Variablen `i` falsch eingerückt ist. JSLint macht Sie darauf aufmerksam und gibt auch die Position an, an der dieses Statement erwartet wird.

Die letzte Meldung in der Ausgabe von Listing 9.24 bezieht sich auf nicht erreichbaren Code. Die letzte Zeile in Listing 9.23 wird niemals ausgeführt, da die Funktion zuvor durch ein `return`-Statement verlassen wird.

JSLint verfügt über eine Reihe von Regelsätzen, die standardmäßig alle aktiviert sind. Möchten Sie beispielsweise keine Warnung mehr erhalten, wenn Sie kein `use strict` verwenden, können Sie dies über die Kommandozeilenoption `--sloppy` erreichen. Listing 9.25 zeigt die entsprechende Kommandozeile.

```
jslint --sloppy jslint.js
```

Listing 9.25 Kommandozeilenoption von JSLint

Geben Sie auf der Kommandozeile lediglich den Befehl `jslint` an, erhalten Sie weitere Informationen über die möglichen Optionen.

In vielen Entwicklungsumgebungen wie Eclipse oder WebStorm ist JSLint entweder bereits fester Bestandteil oder kann als Plugin eingebunden werden. So können Sie schon während der Entwicklung Ihrer Applikation darauf achten, ob Ihr Quellcode bestimmten Qualitätskriterien genügt.

9.5.2 PMD CPD

Neben JSLint existieren noch viele weitere Programme, die sich mit den verschiedensten Aspekten der Qualitätssicherung im Bereich JavaScript beschäftigen. Eines

dieser Werkzeuge ist PMD. Es stammt aus der Java-Welt und ist auch komplett in Java implementiert. Es stellt eine Sammlung von Werkzeugen für die statische Codeanalyse von Applikationen zur Verfügung. Unter anderem ist eine sogenannte Copy-and-Paste-Detection enthalten. Zweck dieses Moduls ist das Auffinden von Duplikaten im Quellcode.

Duplikate im Quellcode stellen ein erhebliches Problem für die Wartbarkeit und Erweiterbarkeit einer Applikation dar. Wird Quellcode innerhalb einer Applikation kopiert und an anderer Stelle eingefügt, muss der gleiche Quellcode mehrfach gewartet werden. Dies wird vor allem relevant, wenn Fehler in diesem Quellcode auftreten und dann an sämtlichen Stellen behoben werden müssen, an denen der Code eingefügt wurde. Ähnliches gilt auch für Erweiterungen. Wird ein Codeblock verbessert oder weiterentwickelt, muss dies ebenfalls an allen Stellen geschehen, an denen dieser Block verwendet wird.

Dieses Problem existiert nicht nur in Java-Applikationen, sondern in allen Programmiersprachen. So wurde die bestehende Infrastruktur verwendet, die PMD bietet, und der Copy-and-Paste-Detector, kurz CPD, dahingehend erweitert, dass er auch mit JavaScript-Code umgehen kann.

Die Grundlage für die Anwendung von PMD CPD bildet der Beispielcode, den Sie in Listing 9.26 sehen. Speichern Sie diesen in einer Datei mit dem Namen `cpd.js` ab, können Sie im Anschluss die CPD darauf anwenden.

```
function duplicate1(input) {
    var output = '';
    for (var i = 0; i < 10; i++) {
        output += input;
    }
    return output;
}

function duplicate2(input) {
    var output = '';
    for (var i = 0; i < 10; i++) {
        output += input;
    }
    return output;
}
```

Listing 9.26 Quellcode mit Duplikaten

Der Quellcode in Listing 9.26 stellt kein Problem dar, da beide Funktionen unterschiedlich benannt sind, allerdings die gleiche Logik implementieren.

Im nächsten Schritt benötigen Sie PMD, die Plattform, die Ihnen unter anderem die CPD zur Verfügung stellt. Sie erhalten die Applikation auf der Webseite <http://pmd.sourceforge.net>. Sobald Sie das Archiv entpackt haben, können Sie es verwenden. Sie müssen keine Installationsroutine oder Ähnliches ausführen. Im `bin`-Verzeichnis von PMD liegt die Datei `run.sh` für UNIX-basierte Systeme beziehungsweise verschiedene `.bat`-Dateien für Windows. Führen Sie nun PMD mit dem in Listing 9.27 angegebenen Kommando in dem Verzeichnis aus, in dem Ihr JavaScript-Quellcode liegt, wird dieser auf Duplikate geprüft.

```
$ /opt/pmd/bin/run.sh cpd --files . --minimum-tokens 4 --encoding utf8 --language ecma-script
```

Listing 9.27 Ausführung der CPD

In Listing 9.27 wird davon ausgegangen, dass Sie PMD unter `/opt` installiert haben. Sollten Sie das Archiv an anderer Stelle entpackt haben, müssen Sie den Pfad entsprechend modifizieren. Die einzelnen Optionen sind in Tabelle 9.3 erklärt.

Option	Bedeutung
<code>cpd</code>	Applikation der PMD-Plattform, die ausgeführt werden soll
<code>--files</code>	Dateien, die überprüft werden sollen
<code>--minimum-tokens</code>	Anzahl der Sprach-Tokens, die mindestens übereinstimmen müssen
<code>--encoding</code>	Zeichencodierung des Quellcodes
<code>--language</code>	Programmiersprache, in der der Quellcode vorliegt

Tabelle 9.3 Kommandozeilenoptionen für PMD

Die Ausgabe, die Sie von diesem Kommando erhalten, sehen Sie in Listing 9.28.

```
Found a 7 line (22 tokens) duplication in the following files:
Starting at line 0 of /tmp/node/./cpd.js
Starting at line 8 of /tmp/node/./cpd.js

function duplicate1(input) {
    var output = '';
    for (var i = 0; i < 10; i++) {
        output += input;
    }
    return output;
```

Listing 9.28 Ausgabe der CPD

Sie können mit CPD allerdings nicht nur Duplikate innerhalb einer Datei finden. Es ist auch möglich, duplizierte Codeblöcke über Dateigrenzen hinweg aufzuspüren. Damit Sie dies erreichen, müssen die Dateien lediglich in dem Pfad liegen, den Sie über die `--files`-Option angegeben haben.

Die hier vorgestellten Werkzeuge dienen der automatisierten Qualitätssicherung von Applikationen. In den nächsten Abschnitten lernen Sie mit dem Node.js Debugger ein Werkzeug kennen, mit dem Sie Ihre Applikation aktiv zur Laufzeit analysieren und dabei Fehler schnell und komfortabel aufspüren können.

9.6 Node.js Debugger

Setzen Sie eine umfangreichere Applikation mit Node.js um, kommen Sie früher oder später im Entwicklungsprozess an die Stelle, an der Sie einen Fehler in Ihrem eigenen Quellcode finden und beheben müssen.

Die meisten Fehler entstehen durch fehlerhaft belegte Variablen. Aus diesem Grund sind Sie in den meisten Fällen bei der Fehlersuche an der Belegung bestimmter Variablen beziehungsweise an der Struktur bestimmter Objekte interessiert. Im einfachsten Fall fügen Sie an bestimmten Stellen im Quellcode `console.log`-Statements ein, mit deren Hilfe Sie die Werte bestimmter Variablen ausgeben können. In Listing 9.29 sehen Sie, wie ein derartiges Stück Quellcode aussehen kann.

```
var myObj = {a: 1, b: 2};
console.log(myObj);
myObj.a = 'Hello';
console.log(myObj);
```

Listing 9.29 Debugging mit `console.log`

Diese Art des Debuggings einer Applikation weist gleich mehrere Probleme auf. Damit Sie Ausgaben erhalten, müssen Sie den Quellcode aktiv bearbeiten und die Statements an den betreffenden Stellen einbauen. Das bedeutet konkret, dass Sie den Quellcode für die Fehleranalyse zusätzlich anpassen müssen und ihn aus diesem Grund weiter verändern und eventuell zusätzliche Probleme schaffen.

Eine weitere Schwierigkeit dieser Methode des Debuggings liegt darin, dass Sie durch die Verwendung der `console.log`-Statements kein Abbild der Umgebung zur Laufzeit der Applikation erhalten, sondern lediglich einen sehr speziellen Ausschnitt, der durch gewisse Einflüsse sogar noch weiter verfälscht werden kann.

Abhilfe für diese Problemstellungen schafft der Debugger von Node.js. Der Vorteil des Debuggers ist, dass er fester Bestandteil der Node.js-Plattform ist und Sie aus diesem Grund keine weitere Software installieren müssen.

Die Ausgangssituation für das Debugging bildet ein einfacher Webserver. Den Quellcode finden Sie in Listing 9.30.

```
var http = require('http');

http.createServer(function (req, res) {
  res.end('Hello Client');
}).listen(8080);
```

Listing 9.30 Beispielapplikation fürs Debugging

Speichern Sie diesen Quellcode in einer Datei mit dem Namen `webserver.js`, können Sie den Node.js-Debugger ganz einfach auf der Kommandozeile starten. Listing 9.31 zeigt Ihnen den dafür erforderlichen Befehl.

```
$ node debug webserver.js
< debugger listening on port 5858
connecting... ok
break in webserver.js:1
  1 var http = require('http');
  2
  3 http.createServer(function (req, res) {
debug>
```

Listing 9.31 Debugging mit Node.js

Sie starten eine Node.js-Applikation im Debug-Modus, indem Sie die Option `debug` bei der Ausführung des Scripts einfügen. Sobald Sie das Kommando absetzen, startet die Applikation im interaktiven Debug-Modus.

Die erste Information, die Sie erhalten, ist, dass der Debugger eingehende Verbindungen auf Port 5858 entgegennimmt. Des Weiteren erfahren Sie, in welcher Datei und welcher Zeile der Debugger die Ausführung unterbrochen hat. Schließlich werden Ihnen die ersten drei Zeilen des Applikationscodes angezeigt, und dabei wird das Statement, an dem unterbrochen wurde, in Grün hervorgehoben.

9.6.1 Navigation im Debugger

Der `debug>`-Prompt signalisiert, dass der Debugger nun auf Ihre Eingabe wartet. Welche Kommandos Ihnen hier zur Verfügung stehen, sehen Sie in Tabelle 9.4.

Im Beispiel in Listing 9.31 können Sie mit dem Kommando `n` beziehungsweise `next` mit dem Debugger zum nächsten Statement springen. Gekennzeichnet wird dies, indem in Zeile drei die Zeichenkette `http` in Grün dargestellt wird.

Kommando	Bedeutung	Beschreibung
c	Continue	Setzt die Ausführung der Applikation bis zum nächsten Breakpoint fort.
n	Step Next	Überspringt eine Subroutine.
s	Step in	Springt in eine Subroutine.
o	Step Out	Springt aus einer Subroutine in die nächsthöhere Ebene zurück.
pause		Pausiert die Ausführung.

Tabelle 9.4 Kommandos für den Debugger

9.6.2 Informationen im Debugger

Möchten Sie nun beispielsweise wissen, welchen Wert die Variable `http` beinhaltet, erreichen Sie dies, indem Sie innerhalb des Debuggers den Befehl `repl` eingeben. Dieses Kommando bewirkt, dass eine interaktive Shell gestartet wird, von der aus Sie Zugriff auf die Umgebung des Debuggers haben. Sie können dann beispielsweise die Zeichenkette `http` eingeben und erhalten die Struktur dieser Variablen. Durch die Tastenkombination `[Strg] + [C]` können Sie von der interaktiven Shell wieder zurück in den Debugger wechseln. Daraufhin können Sie wie gewohnt die Kommandos aus Tabelle 9.4 zur Navigation verwenden.

Haben Sie in der interaktiven Shell eine Ausgabe erzeugt und sind im Anschluss wieder zurück in den Debug-Modus gesprungen, sehen Sie nicht mehr, an welcher Stelle in Ihrem Quellcode Sie sich befinden. Eine Abhilfe für dieses Problem schafft das `list`-Kommando. Die Funktion sorgt dafür, dass Ihnen der Debugger die aktuell ausgeführte Zeile des Quellcodes anzeigt und zusätzlich eine bestimmte Anzahl von Zeilen vor und nach dieser Zeile. Wie viele Zeilen Sie sehen möchten, können Sie als Argument angeben. Geben Sie keine Zahl an, wird der Wert 5 angenommen. Listing 9.32 zeigt Ihnen ein Beispiel für die Verwendung von `list`.

```
debug> list(1)
  2
  3 http.createServer(function (req, res) {
  4     res.end('Hello Client');
```

Listing 9.32 Die `list`-Funktion des Debuggers

Eine weitere Funktion, die Ihnen der Debugger bietet, ist die Ausgabe eines Backtraces. Dieser ist vor allem hilfreich, wenn Sie mit dem `s`-Kommando in verschiedene Subroutinen gesprungen sind und nun herausfinden möchten, wie Sie an die aktu-

elle Stelle gekommen sind. Mit den Funktionen `backtrace` beziehungsweise der kürzeren Variante `bt` können Sie sich den Backtrace der aktuellen Ausführung anzeigen lassen. Listing 9.33 zeigt Ihnen die Ausgabe des Backtraces für den Fall, dass Sie im Beispiel des Webserver in die Methode `createServer` und deren Subroutinen gesprungen sind.

```
debug> bt
#0 events.js:26:15
#1 net.js:780:23
#2 http.js:1630:14
#3 exports.createServer http.js:1650:10
#4 webserver.js:3:6
#5 Module._compile module.js:449:26
#6 Module._extensions..js module.js:467:10
#7 Module.load module.js:356:32
#8 Module._load module.js:312:12
#9 Module.runMain module.js:492:10
```

Listing 9.33 Backtraces mit dem Debugger

Durchlaufen Sie mit dem Debugger den Quellcode Ihrer Applikation, sind Sie in den meisten Fällen an den Werten bestimmter Variablen interessiert. Diese können Sie über das `repl`-Kommando und die interaktive Shell problemlos gewinnen. Diese Variante ist in vielen Fällen jedoch unpraktikabel, da Sie jedes Mal wieder in die Shell wechseln müssen und dort erst die Werte auslesen können. Eine andere Möglichkeit, mit der Sie die Werte von Variablen herausfinden können, liegt in der Verwendung der `watch`-Funktion des Debuggers. In diesem Fall übergeben Sie den Namen der Variablen, an der Sie interessiert sind, als Zeichenkette an diese Funktion. Steppen Sie nun durch Ihre Applikation, wird Ihnen in jedem Schritt der Wert dieser Variablen angezeigt.

```
debug> watch("http")
debug> n
break in webserver.js:3
Watchers:
  0: http = {"parsers":{"list":[]
, "max":1000, "name": "parsers"}, "globalAgent":...
  1 var http = require('http');
  2
  3 http.createServer(function (req, res) {
  4   res.end('Hello Client');
  5 }).listen(8080);
debug>
```

Listing 9.34 Watchers mit Node.js

Listing 9.34 zeigt Ihnen die Verwendung der `watch`-Funktion. Die Ausgabe dieser Funktion ist im Listing stark verkürzt, da das `http`-Objekt sehr umfangreich ist. Hier sehen Sie auch den größten Nachteil dieser Methode. Setzen Sie einen Watcher auf ein Objekt, ist die Ausgabe nicht strukturiert. Watcher bieten daher hauptsächlich für Variablen mit skalaren Werten oder kleine Objekte einen Vorteil. Für umfangreichere Objekte empfiehlt sich die Verwendung des `repl`-Kommandos.

Im Zuge der `watch`-Funktion existieren zwei weitere Funktionen. Mit der `unwatch`-Funktion können Sie bereits gesetzte Watcher wieder entfernen. Zu diesem Zweck müssen Sie der Funktion lediglich den Namen der Variablen als Zeichenkette übergeben, und der Watcher wird entfernt. Die zweite Funktion `watchers` nimmt keinerlei Argumente entgegen und listet die bestehenden Watcher und die zugehörigen Werte der entsprechenden Variablen auf.

9.6.3 Breakpoints

Mit dem aktuellen Stand ist es Ihnen lediglich möglich, den Quellcode Ihrer Applikation von Beginn an schrittweise bis zu der Stelle durchzugehen, an der Sie ein Problem vermuten. Bei umfangreichen Applikationen kann dies zu einem Problem werden, da es hier unter Umständen sehr lange dauern kann, bis Sie zur betreffenden Stelle gelangen. Für diese Fälle bietet der Node.js-Debugger das Feature der Breakpoints. Ein Breakpoint stellt dabei eine Markierung im Quellcode dar, an der der Debugger automatisch anhält.

```
debug> setBreakpoint(6);
  1 var a = 1,
  2   b = 2,
  3   c;
  4
  5 c = a + b;
* 6 b = a + c;
debug> c
break in webserver.js:6
  4
  5 c = a + b;
* 6 b = a + c;
  7 a = b + c;
  8
```

Listing 9.35 Breakpoints in Node.js

Mit der Funktion `setBreakpoint` des Debuggers können Sie einen Breakpoint definieren. Im Falle von Listing 9.35 setzen Sie mit `setBreakpoint` zu Beginn des Durchlaufs

einen Breakpoint auf Zeile sechs. Mit dem Befehl `c` setzen Sie die Ausführung fort, die dann am Breakpoint angehalten wird. Sie können Breakpoints mit `setBreakpoint` auf verschiedene Arten setzen. Wie Sie bereits sehen konnten, können Sie eine bestimmte Zeile angeben, auf der Sie den Breakpoint setzen möchten. Geben Sie keinen Wert an, wird der Breakpoint auf der aktuellen Zeile gesetzt. Des Weiteren können Sie eine Funktion als Zeichenkette angeben, an deren erstem Statement die Ausführung angehalten wird. Bei der letzten Variante geben Sie schließlich einen Dateinamen und eine Zeilennummer an. Erreicht die Ausführung der Applikation diese Datei und die Zeile, wird die Ausführung unterbrochen.

Sie können alternativ auch Breakpoints direkt in Ihrem Quellcode setzen. Dies erreichen Sie, indem Sie das Statement `debugger` einfügen. In Listing 9.36 sehen Sie die Verwendung des `debugger`-Statements anhand des Webserverbeispiels.

```
var http = require('http');

http.createServer(function (req, res) {
  debugger;
  res.end('Hello Client');
}).listen(8080);
```

Listing 9.36 Verwendung des Debugger-Statements

Starten Sie nun Ihr Script, wie es in Listing 9.37 zu sehen ist, und setzen die Ausführung mit dem `c`-Kommando fort, ist die Applikation bereit, Anfragen von Clients zu akzeptieren.

```
$ node debug webserver.js
< debugger listening on port 5858
connecting... ok
break in webserver.js:1
  1 var http = require('http');
  2
  3
debug> c
```

Listing 9.37 Ausführung einer Applikation im Debug-Modus

Greifen Sie jetzt über einen Webbrowser auf den Webserver über die URL `http://localhost:8080` zu, wird die Ausführung innerhalb der Callback-Funktion unterbrochen. Hier stehen Ihnen die bekannten Features des Debuggers zur Verfügung.

9.7 Debugging in der Entwicklungsumgebung

In den seltensten Fällen entwickeln Sie Ihre Node.js-Applikation auf der Kommandozeile. Häufig setzen Sie eine Entwicklungsumgebung ein, um Ihren Quellcode zu programmieren.

Sowohl für Eclipse als auch für WebStorm existieren Plugins, über die Sie Node.js-Applikationen debuggen können. Damit Sie den Debugger nutzen können, müssen Sie eine Konfiguration für Ihre Applikation erstellen. Wie Sie dies im Einzelnen erreichen können, entnehmen Sie der Dokumentation Ihrer Entwicklungsumgebung.

Nutzen Sie den Debugger in Ihrer Entwicklungsumgebung, können Sie auf die gleichen Features des Debuggers zugreifen wie auch schon auf der Kommandozeile. Sie können also Subroutinen überspringen oder tiefer in die Strukturen gehen. Sie haben aber auch die Möglichkeit, Breakpoints zu setzen oder die Werte bestimmter Variablen auszulesen.

Diese Features, kombiniert mit den Möglichkeiten einer grafischen Entwicklungsumgebung, steigern den Komfort während der Entwicklung und Wartung von Node.js-Applikationen.

9.8 Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie Sie dafür sorgen, dass Ihre Applikationen ein hohes Maß an Qualität aufweisen und wie Sie die Qualität auch über längere Zeit auf einem hohen Level halten.

Dabei haben Sie gesehen, dass es sowohl eine statische als auch eine dynamische Codeanalyse gibt. Die statische Codeanalyse beschäftigt sich mit der Formatierung Ihres Quellcodes und hilft Ihnen dabei, bekannte Antipatterns bei der Entwicklung zu finden und diese zu vermeiden. Die dynamische Codeanalyse beschäftigt sich mit dem Laufzeitverhalten von Applikationen.

Die letzten Abschnitte dieses Kapitels widmeten sich dem Debugging von Applikationen und den Möglichkeiten, die Ihnen der Node.js Debugger bietet.

Im nächsten Kapitel erfahren Sie, wie Sie Ihre Applikationen installieren und skalieren können.

Kapitel 10

Skalierbarkeit und Deployment

Wer noch nie einen Fehler gemacht hat, hat noch nie etwas neues ausprobiert.

– Albert Einstein

Bisher haben Sie gelernt, welche Möglichkeiten Ihnen die Node.js-Plattform zur Implementierung von Webapplikationen zur Verfügung stellt. Hierunter fallen unter anderem ein mehrschichtiges Modulsystem und eine Sammlung hochperformanter Bibliotheken, die Ihnen die Ausführung von JavaScript und den Zugriff auf das Betriebssystem ermöglichen. Haben Sie einen Teil Ihrer Applikation implementiert, stellt sich Ihnen die Frage, wie die Benutzer das bis dahin vorliegende Ergebnis testen können, um ihr Feedback in den Entwicklungsprozess zurückfließen zu lassen. Spätestens an diesem Punkt müssen Sie sich mit dem Thema Deployment beschäftigen. Also mit dem Thema, wie der Quellcode Ihrer Applikation auf ein System gelangt und so konfiguriert wird, dass die Applikation benutzbar wird.

Dieses Kapitel soll Ihnen einen Überblick über die verschiedenen Möglichkeiten des Deployments und der Skalierbarkeit von Node.js geben. Dabei werden Ihnen die Themenkomplexe kurz vorgestellt. Sollten Sie sich mit einem der Themengebiete tiefergehend beschäftigen wollen, finden Sie im Verlauf des Kapitels Verweise auf weiterführende Informationen.

In diesem Kapitel sehen Sie, wie Sie Ihre Applikation auf den Server bringen. Hier können Sie je nach Art der Applikation, Grad der Optimierung und verfügbarem Budget auf die verschiedensten Varianten des Deployments zurückgreifen. In den folgenden Abschnitten lernen Sie vom einfachen Kopieren des Quellcodes bis hin zum automatisierten Deployment verschiedene Strategien kennen.

Beindet sich die Applikation schließlich auf einem System, auf dem sich die Applikation von Nutzern verwenden lässt, ergeben sich weitere Problemstellungen. Wird die Applikation von den Nutzern akzeptiert, erweitert sich die Nutzerbasis schnell, und die Last, mit der die Applikation und die darunterliegenden Systeme umgehen müssen, steigt rasch. Aber nicht nur eine wachsende Nutzerbasis, sondern auch Lastspitzen, die beispielsweise durch Werbung oder Ähnliches verursacht werden, stellen Sie als Betreiber einer Webapplikation vor Herausforderungen.

Dieses Kapitel widmet sich im zweiten Teil den Strategien, mit denen Sie dafür sorgen, dass Ihr System mit den Anfragen Ihrer Nutzer über den gesamten Applikations-

lebenszyklus umgehen kann. Diese Strategien reichen von lokalen Optimierungen über die Verteilung der Last über mehrere Systeme bis hin zu cloudbasierten Lösungen.

10.1 Deployment

Deployment beschreibt den Prozess der Verteilung einer Software auf ein oder mehrere Zielsysteme. Dieser Prozess deckt dabei allerdings nicht nur die erste Installation der Software auf einer Plattform ab, sondern kümmert sich auch um die Aktualisierung der Software für den Fall, dass eine neue Version vorliegt. In der traditionellen Softwareentwicklung wurde initial ein Release durchgeführt, und anschließend wurden in größeren Abständen von mehreren Wochen bis mehreren Monaten Aktualisierungen vorgenommen. In der Webentwicklung sind die Releasezyklen wesentlich kürzer. Werden entsprechende Methoden eingesetzt, kann es in sehr dynamischen Projekten vorkommen, dass mehrfach pro Tag released wird. Diese Tatsache wiederum macht einen hohen Grad von Automatisierung beim Deployment der Software auf das Zielsystem erforderlich, da ansonsten der entstehende Aufwand derart kurze Releasezyklen unmöglich macht.

Weil es nicht die eine richtige Lösung für das Deployment einer Applikation gibt, lernen Sie im Zuge dieses Kapitels mehrere Varianten kennen. Diese reichen von sehr einfachen Lösungen, die lediglich aus dem Kopieren von Quellcode auf die Zielplattform bestehen, bis hin zu umfangreichen Szenarien mit Automatismen.

10.1.1 Einfaches Deployment

Die einfachste Variante des Deployments besteht darin, den Quellcode von dem System, auf dem er entwickelt wird, auf das Zielsystem zu kopieren. Im Normalfall sollten beide Systeme, also das Produktivsystem und das Entwicklersystem, getrennte Systeme sein. Damit Sie also den Quellcode auf das Zielsystem kopieren können, benötigen Sie eine Verbindung zwischen beiden Systemen. Da Node.js auf den verschiedensten Systemen lauffähig ist, sind auch die Verbindungsmöglichkeiten vielfältig. Handelt es sich bei dem Zielsystem beispielsweise um eine UNIX-Plattform wie ein Linux-System, steht Ihnen das Secure Shell-Protokoll zur Verfügung. Weiterführende Informationen zu diesem Protokoll finden Sie unter <http://data-tracker.ietf.org/wg/secsh/>. Mit dem Kommando `scp` können Sie über eine sichere Verbindung den Quellcode zwischen den beiden Systemen austauschen. In Listing 10.1 sehen Sie ein Beispiel für eine entsprechende Kommandozeile.

```
$ scp -r /srv/node/myapp nodejs@prod.myexample.com:/srv/node
```

Listing 10.1 Deployment einer Applikation durch Kopieren des Quellcodes

Die Kommandozeile aus Listing 10.1 bewirkt, dass das Verzeichnis */srv/node/myapp*, das den aktualisierten Quellcode Ihrer Applikation enthält, auf das System *prod.my.example.com* in das Verzeichnis */srv/node* kopiert wird. Sie sollten auf jeden Fall darauf achten, dass die Option *-r* enthalten ist, da ansonsten nicht rekursiv kopiert wird und Sie eine Fehlermeldung erhalten. Ein Nachteil dieser Methode ist, dass keine Synchronisierung der Informationen durchgeführt wird. Das bedeutet, dass neue Dateien korrekt angelegt werden. Dateien, die auf dem Zielsystem einen anderen Stand aufweisen, werden einfach mit dem neuen Inhalt überschrieben. Dateien, die in der neuen Version des Quellcodes nicht mehr existieren, bleiben auf dem Zielsystem jedoch erhalten und werden nicht automatisch gelöscht. Dieses Problem können Sie umgehen, wenn Sie das Zielverzeichnis vor dem Kopiervorgang entweder komplett löschen oder es zumindest leeren.

Kopieren Sie den Quellcode Ihrer Applikation vom System, auf dem Sie die Applikation entwickeln, auf das Produktivsystem, sollten Sie sich bewusst sein, dass dieser Vorgang sämtliche Dateien umfasst, also nicht nur den Quellcode, sondern auch beispielsweise Dateien und Verzeichnisse, die durch die Verwendung eines Versionskontrollsystems wie *git* (<http://git-scm.com/>) oder *Subversion* (<http://subversion.tigris.org/>) angelegt wurden. Diese Informationen sind nicht Bestandteil Ihrer Applikation, dürfen also auch nicht im Produktivsystem auftauchen, auf das Ihre Nutzer Zugriff haben. Diese Forderung hat einen sehr einfachen Hintergrund. Verwenden Sie *git* zur Versionskontrolle Ihrer Applikation, enthält das *.git*-Verzeichnis einen kompletten Klon Ihres Repositories. Das kann zu einem Sicherheitsrisiko werden, falls die Nutzer Ihrer Applikation Zugriff auf dieses Verzeichnis erhalten und seine Inhalte auslesen können. In einem solchen Repository sind meist auch sicherheitskritische Informationen wie beispielsweise Zugangsinformationen zum Datenbankserver hinterlegt. Beim Deployment sollten Sie also das *.git* vom Kopiervorgang ausschließen. *Subversion* bietet zu diesem Zweck das Kommando *export*, das genau diesen Zweck erfüllt. Es erzeugt eine Kopie des aktuellen Standes Ihres Repositories ohne die Metainformationen des Versionskontrollsystems.

Ist Ihr Zielsystem, auf dem Sie Ihre Applikation betreiben möchten, ein Windows-System, können Sie Ihren Quellcode über Verzeichnisfreigaben auf das System kopieren. Nachdem Sie das freigegebene Verzeichnis in Ihr lokales System eingehängt haben, können Sie die Dateien entweder über die Kommandozeile oder über den Windows Explorer kopieren. Eine weitere systemunabhängige Art, mit der Sie den Quellcode Ihrer Applikation deployen können, besteht in der Verwendung eines FTP-Servers. FTP ist ein Protokoll, das speziell auf den Transfer von Dateien ausgelegt ist. FTP-Clients sind für jedes System verfügbar, sodass Sie hier keine Probleme bekommen sollten. Damit Sie FTP zum Deployment benutzen können, muss auf dem Zielsystem ein FTP-Server installiert und konfiguriert sein, damit Sie das Zielverzeichnis auch erreichen können.

Haben Sie den Quellcode Ihrer Applikation erfolgreich auf das Zielsystem kopiert, sind Sie nur noch einen Schritt davon entfernt, dass Sie und Ihre Benutzer die Applikation verwenden können. Sie müssen Ihre Applikation an dieser Stelle nur noch starten. Auch wenn Sie die Software nur aktualisiert haben, ist ein Neustart der Applikation erforderlich, damit Node.js den Quellcode der Applikation neu einliest und die Änderungen endgültig wirksam werden. Verzichten Sie auf einen Neustart, kann es zu Problemen und Inkonsistenzen kommen, da sämtliche Komponenten der Applikation, die direkt ausgeführt werden, bereits eingelesen und größtenteils von der V8-Engine in Maschinencode umgewandelt sind. Lesen Sie Dateien wie beispielsweise Templates erst zur Laufzeit ein, weisen diese auch ohne einen Neustart einen aktuellen Stand auf.

10.1.2 Dateisynchronisierung mit *rsync*

Eines der Probleme, das beim Deployment mit der einfachsten Variante, dem Kopieren, entsteht, ist, dass nicht weiter benötigte Dateien erhalten bleiben. Aber auch für diesen Fall existieren Werkzeuge, die Ihnen das Leben erleichtern. Eines dieser Werkzeuge auf UNIX-basierten Systemen ist *rsync*, das Sie unter <http://rsync.samba.org/> finden können. Dieses besteht aus zwei Teilen: einem Programm und einem Protokoll. Der Zweck dieses Tools ist die Synchronisierung von Dateien zwischen zwei Systemen. Dabei erfolgt die Synchronisierung stets von einer Seite aus. Für Sie bedeutet das konkret, dass Sie wie zuvor auch schon den Quellcode Ihrer Applikation auf einem System entwickeln und von diesem aus die Synchronisierung auf das Serversystem, auf dem die Applikation betrieben werden soll, anstoßen. *rsync* ist als freie Software unter der GPL-Lizenz verfügbar. In Listing 10.2 sehen Sie, wie einfach Sie mit dem *rsync*-Kommandozeilenbefehl auf Ihrem Entwicklersystem den Quellcode Ihrer Applikation synchronisieren können. Als Grundlage dient dabei das Beispiel in Listing 10.1.

```
$ rsync -ar --delete /srv/node/myapp nodejs@prod.myexample.com:/srv/node/myapp
```

Listing 10.2 Synchronisierung von Quellcode mit *rsync*

Die Kommandozeile aus Listing 10.2 bedeutet, dass Sie eine Synchronisierung anstoßen möchten, bei der mit der Option *-a* möglichst viele Eigenschaften der Dateien beibehalten werden sollen. *-r* bedeutet, dass die Synchronisierung rekursiv, also auch über Unterverzeichnisse hinweg, stattfinden soll. Die Option *--delete* sorgt dafür, dass auf der Gegenseite überflüssige Dateien entfernt werden. Die letzten beiden Optionen geben schließlich die Quelle und das Ziel der Synchronisierung an. Sind Sie nicht sicher, ob der *rsync*-Befehl genau das tut, was Sie beabsichtigen, können Sie mit der Option *--dry-run* die Ausführung nur simulieren und sehen, welche Auswirkungen das Kommando auf Ihr Zielsystem hat. Befinden sich auf dem Quellsystem neue Dateien oder solche, bei denen sich der Inhalt geändert hat, werden

diese Dateien auf dem Zielsystem angelegt beziehungsweise die Änderungen übernommen. Existiert eine Datei auf dem Quellsystem nicht mehr, auf dem Zielsystem jedoch schon noch, wird diese Datei gelöscht.

Der Vorteil von `rsync` besteht darin, dass Sie bei diesem Werkzeug auf SSH zum Transport zurückgreifen können. Der Transport Ihrer Daten erfolgt also verschlüsselt. Für den Neustart Ihrer Applikation gilt allerdings das Gleiche wie auch schon beim einfachen Kopieren der Dateien. Sie müssen den Prozess Ihrer Applikation neu starten, damit die aktualisierten Quellcodedateien neu eingelesen werden. Das nächste Problem, vor dem Sie beim Betrieb einer Node.js-Applikation stehen, ist, dass Ihre Applikation nicht mehr läuft, sobald Sie Ihr System neu starten oder es aus irgendwelchen Gründen selbst einen Neustart durchgeführt hat.

10.1.3 Die Applikation als Dienst

Beim besten Betriebssystem kann es von Zeit zu Zeit vorkommen, dass es neu gestartet werden muss. Sei es, weil Updates eingespielt wurden oder weil durch einen Fehler in der Soft- oder Hardware oder durch einen Bedienungsfehler das System abstürzt. Geschieht dies, wird Ihre Applikation, die Sie über die Kommandozeile gestartet haben, nicht automatisch auch mit neu gestartet. Eine weitere Problemstellung, die mit dieser Art des Startens Ihrer Applikation einhergeht, ist, dass Sie sich immer mit dem Server verbinden und auf die Kommandozeile zugreifen müssen, um Ihre Applikation zu starten. Abhilfe schafft hier das Konzept von Diensten beziehungsweise Daemons. Im Kern bedeutet das, dass Sie Ihre Applikation zu einem Dienst des Serversystems machen, der beim Systemstart automatisch hochgefahren wird. In diesem Fall müssen Sie sich nicht mehr am System anmelden und die Applikation manuell starten. Sie müssen lediglich dafür sorgen, dass das System hochfährt. Alles Weitere erledigt Ihr Betriebssystem für Sie. Im Folgenden erfahren Sie, wie Sie auf sehr einfache Art und Weise dafür sorgen können, dass Ihre Applikation unter verschiedenen Systemen als Dienst ausgeführt wird.

Dienste unter UNIX

Die verschiedenen UNIX-Distributionen unterscheiden sich in der Art, wie die Dienste des Systems gestartet werden. Hier sämtliche Varianten aufzulisten würde den Rahmen dieses Buches sprengen. Sie erfahren in diesem Abschnitt, wie Sie grundsätzlich dafür sorgen können, dass Ihr Node.js-Server auf einem Ubuntu-System automatisch beim Systemstart gestartet wird. Der Start von Linux-Systemen basiert auf dem Konzept der Runlevel. Ein Runlevel definiert, welche Dienste auf dem System ausgeführt werden sollen. Umgesetzt wird dies durch eine Reihe von Skripten, die sich im Verzeichnis `/etc/init.d` befinden und die dafür sorgen, dass Dienste gestartet und angehalten werden. Je nach Runlevel werden verschiedene dieser

Scripte aufgerufen. Ein guter Ausgangspunkt für Ihr eigenes Script ist die Datei */etc/init.d/skeleton*. Diese Datei sollten Sie kopieren und als Vorlage verwenden. Listing 10.3 zeigt eine vereinfachte Variante eines solchen Scripts.

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          Webserver in Node.js
# Required-Start:
# Required-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: HTTP-Server implementierung in Node.js
# Description:       Stellt einen Webserver auf Port 80 zur Verfügung
### END INIT INFO
# Author: Name <root@example.com>

# Aktionen
case "$1" in
    start)
        /srv/node/server.js start
        ;;
    stop)
        /srv/node/server.js stop
        ;;
    restart)
        /srv/node/server.js restart
        ;;
esac

exit 0
```

Listing 10.3 Startscript für einen Node.js-Dienst unter Ubuntu

Verwenden Sie den Quellcode aus Listing 10.3, müssen Sie ihn dahingehend anpassen, dass das Script in *server.js* dafür sorgt, dass der Serverprozess gestartet wird, wenn dem Script die Option *start* übergeben wird. Außerdem sollten Sie die Optionen *stop* zum Beenden des Dienstes und *restart* für das Beenden und anschließende Starten unterstützen.

Dienste unter Windows

Über einige Umwege ist es möglich, einen Node.js-Server unter Windows als Dienst zu starten. Damit Sie Ihr Ziel erreichen können, benötigen Sie den Quellcode Ihrer Applikation und zwei Hilfsdateien. Auf der Kommandozeile müssen Sie das Kommando aus Listing 10.4 absetzen.

```
instsrv.exe nodeServer C:\srvany.exe
```

Listing 10.4 Erstellen eines Dienstes

Mit dem Kommando aus Listing 10.4 erstellen Sie einen Eintrag in der Windows Registry. Mit dessen Hilfe können Sie im weiteren Verlauf Ihre Applikation als Dienst starten. Mit dem Werkzeug `regedit` können Sie anschließend über den Pfad `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\nodeServer` einen neuen Schlüssel mit dem Namen `Parameters` anlegen. Diesen benennen Sie als `Application` und weisen ihm den Typ `REG_SZ` und den Wert `nodeC:\server.js` zu. Der letzte Schritt besteht schließlich darin, dass Sie über die Dienstverwaltung Ihres Systems den Dienst starten.

Die folgenden Abschnitte sollen Ihnen noch einige bewährte Vorgehensweisen im Umgang mit Modulen in Applikationen mit auf den Weg geben.

10.1.4 `node_modules` beim Deployment

In den meisten Fällen setzen Sie in Ihrer Applikation auf Module aus externen Quellen wie beispielsweise dem NPM-Repository. Beim Deployment Ihrer Applikation haben Sie zwei Varianten, wie Sie mit diesen Modulen umgehen können. Sie können die Module entweder auf Ihrem Entwicklersystem installieren und sie von hier aus auf das Produktivsystem deployen, oder Sie können die Module von der Synchronisierung ausschließen und sie auf dem Produktivsystem separat installieren.

Für NPM-Module, die keine externen Abhängigkeiten haben wie zum Beispiel *express.js* oder *underscore.js*, ist die erste Variante, bei der Sie das `node_modules`-Verzeichnis einfach mit synchronisieren, kein weiteres Problem, da diese systemunabhängig sind und einfach kopiert werden können. Schwieriger wird die Situation bei Modulen, die auf bestimmten Bibliotheken aufbauen. Beispiele hierfür sind vor allem Datenbanktreiber wie der MySQL-Treiber, der auf `libmysql` aufbaut, oder der Treiber für SQLite3, der eine funktionsfähige Version von `sqlite3` benötigt. Solange das Quell- und Zielsystem gleich beziehungsweise sehr ähnlich aufgebaut ist, sind kaum Probleme zu erwarten. Weisen die Systeme allerdings unterschiedliche Softwarestände oder gar unterschiedliche Betriebssysteme auf, werden diese Module nicht funktionieren. In diesem Fall müssen Sie auf die zweite Variante des Umgangs mit dem `node_modules`-Verzeichnis zurückgreifen und das komplette Verzeichnis beziehungsweise zumindest die Module, die Ihnen Probleme bereiten, von der Synchronisierung ausschließen. Im Anschluss an das Deployment müssen Sie dann die betreffenden Module auf dem Zielsystem manuell installieren beziehungsweise aktualisieren. Listing 10.5 zeigt Ihnen, wie Sie mit der `--exclude`-Option von `rsync` die Synchronisierung des `mysql`-Treibers vermeiden.

```
$ rsync -ar -delete -exclude='node_modules/mysql' /srv/node/myapp ↻
nodejs@prod.myexample.com: /srv/node/myapp
```

Listing 10.5 Ausschluss von Modulen von der Synchronisierung

Neben diesen Arten des Deployments einer Applikation können Sie natürlich auch den Node Package Manager verwenden, um Ihre Applikation auf einem System zu installieren.

10.1.5 Applikationen mit dem Node Package Manager installieren

Bereits in Kapitel 4, »Anpassung und Erweiterung«, konnten Sie sehen, wie einfach es ist, Ihr eigenes NPM-Paket auf <http://npmjs.org> zu publizieren und so einer breiten Öffentlichkeit zugänglich zu machen. Diese Plattform können Sie auch zum Deployment Ihrer Applikation benutzen. Der Vorteil dieser Variante des Deployments liegt darin, dass Sie auf die Infrastruktur von <http://npmjs.org> zugreifen können. Sie müssen sich also nicht darum kümmern, ein Repository zu hosten und die dazu erforderlichen Server zur Verfügung zu stellen. Ein weiterer Vorteil der Verwendung von NPM als Deployment-Werkzeug besteht darin, dass Sie auf die Funktionalität des Paketmanagers zurückgreifen können. Das bedeutet konkret, dass Sie Ihre Applikation mit nur einem Kommando installieren können. Dabei müssen Sie sich nicht um die Auflösung von Abhängigkeiten kümmern. Stattdessen nimmt Ihnen der NPM diese Aufgabe ab und lädt alle Pakete, die in der *package.json*-Datei Ihrer Applikation als Abhängigkeiten gelistet sind, herunter und platziert sie im *node_modules*-Verzeichnis. Der entscheidende Nachteil dieser Methode ist, dass jedes Paket, das Sie über <http://npmjs.org> veröffentlichen, frei im Internet verfügbar ist. Das ist für Open-Source-Projekte in der Regel kein Problem. Für Applikationen, die Sie für sich selbst oder für ein Unternehmen erstellen, kommt eine Veröffentlichung meist nicht infrage. Häufig enthalten derartige Applikationen Zugangsdaten zu Datenbanken oder Algorithmen und Workflows, die nicht in die Hände Dritter gelangen sollten. Wenn Sie also NPM und das NPM-Repository unter <http://npmjs.org> nutzen möchten, sollten Sie sich dies genau überlegen und die Vor- und Nachteile abwägen.

Ein weiterer Aspekt, den Sie bei der Verwendung von NPM beim Deployment beachten sollten, ist, dass Sie auf die Verfügbarkeit des Repositories angewiesen sind. Das bedeutet, dass Sie Ihre Applikation nicht installieren können, wenn Sie auf <http://npmjs.org> nicht zugreifen können, weil beispielsweise der Server offline ist. Aber auch für diese Problemstellung existiert eine Lösung. Das Repository basiert auf Standardkomponenten, und so können Sie in nur wenigen Schritten das Repository auf Ihrem eigenen Server replizieren und sind anschließend nicht mehr abhängig davon, ob der zentrale Server verfügbar ist. Die Grundlage für Ihr eigenes Repository bildet CouchDB. CouchDB ist eine Datenbank, die einen ähnlichen Ansatz wie Mon-

goDB verfolgt. Die Inhalte dieser Datenbanken werden in Dokumenten organisiert. Listing 10.6 zeigt Ihnen, wie Sie die Replikation einrichten und dafür sorgen, dass sie kontinuierlich aktualisiert wird.

```
curl -X POST http://127.0.0.1:5984/_replicate -d '{"source":"http://isaacs.␣
iriscouch.com/registry/", "target":"registry", "continuous":true, ␣
"create_target":true}' -H "Content-Type: application/json"
```

Listing 10.6 Kommando zur Replikation des NPM-Repositorys

Ansonsten müssen Sie lediglich das Github-Repository von *http://npmjs.org* klonen, damit Sie das Repository betreiben können. Listing 10.7 zeigt Ihnen die dafür erforderlichen Kommandos.

```
git clone git://github.com/isaacs/npmjs.org.git
cd npmjs.org
sudo npm install -g couchapp
npm install couchapp
npm install semver
couchapp push registry/app.js http://localhost:5984/registry
couchapp push www/app.js http://localhost:5984/registry
```

Listing 10.7 Installation eines eigenen NPM-Repositorys

Danach müssen Sie nur noch die Domain in der Konfigurationsdatei */usr/local/etc/couchdb/local.ini* anpassen, und die Installation Ihres Repositorys ist abgeschlossen. Verwenden Sie das *npm*-Kommando nun auf Ihrem System, wird nach wie vor das globale Repository auf *http://npmjs.org* verwendet. Auch hier müssen Sie noch eine Anpassung vornehmen, bevor Sie hier Ihr eigenes Repository verwenden können. Wie diese aussieht, sehen Sie in Listing 10.8.

```
npm config set http://localhost:5984/registry/_design/app/_rewrite
```

Listing 10.8 Setzen der Registry-URL für NPM

Der Node Package Manager bietet Ihnen allerdings noch eine weitere Möglichkeit, um Applikationen auf einem System zu installieren. Mehr zu diesem Thema erfahren Sie im folgenden Abschnitt.

10.1.6 Pakete lokal installieren

Eines der größten Probleme von NPM ist, dass Sie bei der Installation auf die Registry zugreifen, die für alle zugänglich im Internet liegt. Möchten Sie Ihre Applikation jedoch nicht veröffentlichen, ist dieses Repository keine Option. Stattdessen können

Sie NPM auch nutzen, um Pakete lokal zu installieren. Mit dieser Art der Installation nutzen Sie die Vorteile beider Welten. Einerseits halten Sie Ihre eigene Applikation unter Verschluss und machen sie nur einem berechtigten Personenkreis verfügbar, und andererseits können Sie für die Abhängigkeiten auf das Repository unter <http://npmjs.org> zurückgreifen. Alles, was Sie hierfür tun müssen, ist, eine *package.json*-Datei für Ihre Applikation zu erstellen, die unter anderem die Abhängigkeiten auflistet. Packen Sie diese Datei zusammen mit dem Quellcode Ihrer Applikation in ein komprimiertes Tar-Archiv und stellen dieses über einen Webserver zum Download zur Verfügung, können Sie Ihre Applikation ganz einfach über die Kommandozeile installieren. Listing 10.9 geht davon aus, dass Sie einen lokalen Webserver betreiben und dort die Datei *myApp.tgz* zur Verfügung stellen.

```
$ npm install http://localhost/myApp.tgz
```

Listing 10.9 Installation eines NPM-Pakets von einem Webserver

Im Falle von Listing 10.9 wird die Datei *myApp.tgz* heruntergeladen und entpackt. Im Anschluss daran wird die *package.json*-Datei ausgewertet, und die entsprechenden Abhängigkeiten werden heruntergeladen und ebenfalls installiert. Alternativ können Sie die Datei auch lokal platzieren und von dort unter Angabe des Dateinamens installieren.

10.1.7 Toolunterstützung mit Grunt

Mit Grunt als Buildtool für Ihre Applikation haben Sie bereits an mehreren Stellen Bekanntschaft gemacht. An dieser Stelle sollen Sie noch einmal erfahren, wie Grunt Ihnen vom Beginn der Entwicklung bis zur Fertigstellung Ihrer Applikation immer wieder Hilfestellung bieten und so auch zu einem erfolgreichen Deployment beitragen kann. Damit Sie Grunt verwenden können, müssen Sie sicherstellen, dass das Werkzeug auf Ihrem System installiert ist. Listing 10.10 zeigt Ihnen, wie Sie herausfinden können, ob das Werkzeug bereits auf Ihrem System vorhanden ist.

```
npm ls -g grunt-cli
/usr/local/lib
└─ grunt-cli@0.1.6
```

Listing 10.10 Prüfung, ob grunt-cli auf dem System installiert ist

Sollte Grunt noch nicht auf Ihrem System installiert sein, können Sie es mit dem Kommando `npm install -g grunt-cli` tun. Des Weiteren benötigen Sie innerhalb Ihrer Applikation zwei Dateien: *package.json* und *Gruntfile.js*. Die Datei *package.json* wird von NPM zur Installation Ihrer Applikation verwendet. Alles, was Sie hier tun müssen, ist, Grunt und sämtliche Plugins von Grunt, die Sie für Ihr Projekt verwen-

den möchten, unter `devDependencies` aufzulisten. Die zweite Datei *Gruntfile.js* enthält schließlich die Konfiguration für Grunt. Sie sorgt dafür, dass die Plugins geladen werden, und ermöglicht es Ihnen, eigene Tasks zu definieren.

Bevor Sie eine Applikation deployen, müssen Sie eine Reihe von Aufgaben ausführen, die den Quellcode auf die Auslieferung vorbereiten. Für viele dieser Aufgaben existieren Plugins für Grunt. Im Folgenden sehen Sie, wie Sie einige dieser Plugins einbinden und wozu sie verwendet werden. Beginnen Sie mit der Erstellung der Datei *Gruntfile.js* und dem grundlegenden Inhalt der Datei, wie Sie ihn in Listing 10.11 sehen können.

```
module.exports = function (grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
  });
};
```

Listing 10.11 Grundgerüst einer Gruntfile.js-Datei

In Listing 10.11 sehen Sie, dass die *Gruntfile.js* aufgebaut ist wie ein gewöhnliches Node.js-Modul. Über `module.exports` stellen Sie Funktionalität nach außen zur Verfügung, mit dem Unterschied, dass Sie in diesem Fall keine Applikationsfunktionalität exportieren, sondern stattdessen die Konfiguration für Grunt. Im ersten Schritt rufen Sie die Methode `grunt.initConfig` auf. Diese akzeptiert ein Objekt mit der Konfiguration für Grunt. Im Falle von Listing 10.11 beinhaltet dieses Objekt die Eigenschaft `pkg`, deren Wert der Inhalt der *package.json*-Datei Ihrer Applikation ist.

Im Folgenden werden Sie für Ihr Projekt eine Reihe von Grunt-Plugins einsetzen und konfigurieren. Als Basis hierfür benötigen Sie jedoch die Plugins in Ihrem Projekt. Wie bereits erwähnt, erreichen Sie dies, indem Sie Grunt selbst und die Plugins unter den `devDependencies` in Ihrer *package.json*-Datei auflisten. Listing 10.12 zeigt Ihnen einen Ausschnitt aus der für die Beispiele verwendeten Datei.

```
"devDependencies": {
  "grunt": "~0.4.1",
  "grunt-contrib-nodeunit": "~0.1.2",
  "grunt-contrib-uglify": "~0.2.0",
  "grunt-contrib-cssmin": "~0.5.0",
  "grunt-jshint": "~0.2.5"
}
```

Listing 10.12 devDependencies in der package.json-Datei

Nachdem Sie diese Pakete in Ihrem Projekt installiert haben, können Sie dazu übergehen, die Plugins in Ihre *Gruntfile.js* einzubinden und damit zu laden. Das Einbin-

den von Plugins geschieht über die Methode `grunt.loadNpmTasks`. In Listing 10.13 sehen Sie, wie die vier Plugins, die Sie zuvor in die *package.json* eingetragen haben, in Ihrer *Gruntfile.js* geladen werden.

```
grunt.loadNpmTasks('grunt-contrib-nodeunit');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-cssmin');
grunt.loadNpmTasks('grunt-jshint');
```

Listing 10.13 Laden von Plugins in der Gruntfile.js

Sind die Plugins geladen, müssen Sie die einzelnen Tasks noch konfigurieren, damit Sie sie ausführen können. Die Konfiguration erfolgt im Aufruf der `grunt.initConfig`-Methode, genauer im Objekt, das dieser Methode beim Aufruf übergeben wird.

Das erste Modul, das Sie konfigurieren, ist dafür verantwortlich, die Tests, die Sie für Ihr Projekt erstellt haben, ablaufen zu lassen. Voraussetzung für diesen Task ist, dass Sie Ihre Tests für Nodeunit verfasst haben. Die Konfiguration des Nodeunit-Tasks sehen Sie in Listing 10.14.

```
nodeunit: {
  all: ['test/*_test.js']
}
```

Listing 10.14 Konfiguration des Nodeunit-Tasks

Die Konfiguration in Listing 10.14 geht davon aus, dass alle Tests, die ausgeführt werden sollen, im Verzeichnis *test* liegen und die Dateinamen jeweils auf *_test.js* enden. Sie können nun bereits Ihre Tests über Grunt ausführen, indem Sie das Kommando `grunt nodeunit:all` auf der Kommandozeile absetzen. Der nächste Task, den Sie hier kennenlernen, beschäftigt sich mehr mit den Frontend-Komponenten einer Webapplikation, und zwar mit der Optimierung von CSS-Dateien. Listing 10.15 stellt Ihnen die Konfiguration des *cssmin*-Plugins vor.

```
cssmin: {
  compress: {
    files: {
      'public/css/min.css': ['src/css/dialog.css',
        'src/css/app.css']
    }
  }
}
```

Listing 10.15 Konfiguration zur Optimierung von CSS

Zur Konfiguration dieses Plugins benötigen Sie, ähnlich wie schon bei Nodeunit, kaum Informationen. Sie müssen lediglich innerhalb der `files`-Direktive die Zielfeile und in einem Array die zu komprimierenden Dateien angeben. Der nächste Task widmet sich ebenfalls dem Frontend Ihrer Webapplikation, und zwar der Optimierung des JavaScript-Codes, der an das Frontend ausgeliefert wird. Die Bearbeitung wird dabei an UglifyJS weitergeleitet.

```
uglify: {
  businessRoutines: {
    files: {
      'public/js/logic.min.js': ['src/js/validators.js',
        'src/js/tools.js']
    }
  }
}
```

Listing 10.16 Komprimierung von JavaScript-Quellcode

Geben Sie wie in Listing 10.16 lediglich die Zielfeile und die Quelldateien an, werden diese mit den Standardoptionen komprimiert und zusammengefügt. Mit dem nächsten Task, den Sie hier kennenlernen, können Sie einen weiteren Bereich der Qualitätssicherung für Ihre Applikation automatisieren. Mit dem JSLint-Plugin können Sie Ihren Code über Grunt analysieren lassen. Eine Beispielkonfiguration sehen Sie in Listing 10.17.

```
jslint: {
  files: ['src/js/*.js']
}
```

Listing 10.17 Basiskonfiguration von JSLint unter Grunt

Neben der Konfigurationsoption aus Listing 10.17 existieren noch zahlreiche weitere Optionen, mit deren Hilfe Sie in der Lage sind, das Verhalten von JSLint bei der Überprüfung Ihres Quellcodes zu beeinflussen. So können Sie analog zur `files`-Direktive mit `exclude` auch Dateien ausschließen. Mit der Eigenschaft `directives` können Sie bestimmte Einstellungen an JSLint selbst vornehmen wie zum Beispiel die Definition einer Reihe globaler Variablen, die JSLint bei der Überprüfung ignorieren soll. Zu guter Letzt können Sie über `options` die Ausgabe und das Laufzeitverhalten des Plugins steuern. Die Eigenschaft `failOnError` sorgt beispielsweise dafür, dass der JSLint unterbrochen wird, sobald ein Fehler auftritt.

Haben Sie alle beschriebenen Anpassungen an Ihrer *Gruntfile.js* vorgenommen, können Sie die von Ihnen konfigurierten Tasks verwenden. Einen Task führen Sie aus, indem Sie auf der Kommandozeile Grunt in dem Verzeichnis ausführen, in dem Ihre *Gruntfile.js* liegt. Listing 10.18 zeigt Ihnen einen solchen Aufruf.


```
$ grunt nodeunit
```

Listing 10.18 Ausführen von Grunt-Tasks

Als Option akzeptiert Grunt den Namen eines Tasks, der ausgeführt werden soll. Gültige Namen sind diejenigen Zeichenketten, die Sie über `grunt.initConfig` konfiguriert haben, also in diesem konkreten Beispiel der Task `nodeunit`, der für die Ausführung der Unittests sorgt. Möchten Sie nun aber alle Tasks ausführen, hilft Ihnen Grunt mit nur einem Task als Option nicht weiter, es sei denn, Sie stoßen alle Tasks nacheinander an. In Grunt können Sie allerdings auch einen sogenannten Standardtask definieren. Listing 10.19 zeigt Ihnen, wie Sie hier vorzugehen haben.

```
grunt.registerTask('default', ['nodeunit', 'cssmin', 'uglify', 'jshint']);
```

Listing 10.19 Standardtask in Grunt

Fügen Sie im ersten Schritt die Zeile aus Listing 10.19 in Ihre *Gruntfile.js* ein. Dadurch erstellen Sie den Standardtask. Dieser sorgt dafür, dass Sie Grunt auch ganz ohne Kommandozeilenoptionen im Verzeichnis der *Gruntfile.js* aufrufen können. In diesem Fall werden sämtliche Tasks ausgeführt, die Sie der Methode `registerTask` als zweites Argument angeben.

Die Tasks, die Sie hier sehen konnten, stellen lediglich einen Bruchteil der existierenden Tasks dar, die Sie für Ihr Projekt einsetzen können. Weitere Informationen finden Sie unter <http://www.gruntjs.com>. Ein sehr hilfreiches Plugin für Grunt möchte ich Ihnen dennoch vorstellen. Das `grunt-contrib-watch`-Plugin bietet Ihnen einen Weg, über den Sie Tasks ausführen können, falls sich Dateien innerhalb Ihres Projekts verändern. Listing 10.20 zeigt die notwendigen Anpassungen Ihrer *Gruntfile.js*.

```
...
grunt.initConfig({
...
  watch: {
    files: ['src/js/*.js'],
    tasks: ['jshint']
  }
});
...
grunt.loadNpmTasks('grunt-contrib-watch');
...
});
```

Listing 10.20 Einbinden des Watch-Plugins

Bevor Sie das Watch-Plugin jedoch verwenden können, müssen Sie es mit dem Kommando `npm install grunt-contrib-watch` installieren. Danach können Sie den Dateisystem-Watcher mit dem Befehl `grunt watch` starten. Ändern Sie eine der Dateien, die Sie unter `files` angegeben haben, werden automatisch sämtliche Tasks, die unter `tasks` gelistet sind, ausgeführt. Nach der Installation und Konfiguration können Sie diesen speziellen Task ausführen. Beachten Sie dabei, dass es sich bei diesem Task um einen kontinuierlich laufenden Task handelt. Konkret heißt das für Sie, dass Sie diesen Task nicht in die Standardtasks mit aufnehmen sollten.

Durch seinen modularen Charakter ist Grunt ein flexibles Hilfsmittel, das Sie sowohl im Entwicklungsprozess als auch zum Deployment Ihrer Applikation einsetzen können. Der Vorteil von Grunt liegt dabei darin, dass das Werkzeug in JavaScript für JavaScript entwickelt wurde. Arbeiten Sie also an einer JavaScript-Applikation, ist Grunt ein Werkzeug, das sich sehr gut in den Entwicklungsprozess integrieren lässt.

In den nächsten Abschnitten sehen Sie nun, wie Sie damit umgehen können, wenn Sie nach dem Deployment feststellen, dass Ihre Applikation mit dem Datenaufkommen oder der Anzahl der Anfragen nicht umgehen kann.

10.2 Skalierung

Ein wichtiges Kriterium bei der Entwicklung und Stabilisierung von Node.js ist der Performanceaspekt. Ein erklärtes Ziel der Entwickler der Plattform ist es, den Entwicklern ein sehr leistungsfähiges Werkzeug zu bieten, mit dem sich auf einfache Art stabile und performante Applikationen umsetzen lassen. Die Anzahl von Anfragen und auch die Datenmenge, die von einer Instanz einer Applikation bedient werden kann, sind nicht unbegrenzt, sondern abhängig von zahlreichen äußeren und inneren Faktoren. Die Dinge, die Sie als Entwickler direkt beeinflussen können, liegen vor allem in der Art und Weise, wie Sie Ihre Applikation gestalten. Hierunter fällt beispielsweise die Gestaltung des Quellcodes im Allgemeinen, also ob Sie hierbei grundsätzlich auf Performance achten. Des Weiteren beeinflusst die Verwendung von Frameworks unter Umständen die Performance Ihrer Applikation negativ, vor allem Datenbank-Abstraktionslayer sind dafür bekannt, dass sie bei falscher Verwendung erhebliche Beeinträchtigungen verursachen. Verwenden Sie innerhalb Ihrer Applikation Caches, um häufig verwendete Daten oder vorgerenderte Templates bereits vorzuhalten, können Sie mit diesen Mitteln an Performance in Ihrer Applikation gewinnen.

Einflussfaktoren, die nicht direkt in Ihrer Applikation begründet sind, sind beispielsweise die verfügbaren Ressourcen, also freier Arbeitsspeicher oder Zugriff auf den Prozessor. Je weniger Ressourcen Ihnen zur Verfügung stehen, desto problematischer ist die Situation. Was Sie darüber hinaus beachten sollten, ist der Durchsatz bei Schreiboperationen. Entwickeln Sie etwa eine Applikation, in der sehr viele Daten

persistiert werden müssen, sollten Sie darauf achten, dass Sie ein System zur Verfügung stellen, das mit der erwarteten Menge an Daten auch umgehen kann. Hier reicht die Spanne von arbeitsspeicherbasierten Lösungen wie Redis über das Speichern auf der Festplatte bis hin zu Speicherlösungen im Netzwerk, wobei Schreiboperationen in den Arbeitsspeicher wesentlich schneller sind als solche auf entfernte Medien über das Netzwerk.

Auch wenn Sie noch so viele Ressourcen auf einem System zur Verfügung stellen können: Mit einer stetig wachsenden Anzahl an Benutzern gelangen Sie irgendwann an die Grenzen des stärksten Serversystems, beziehungsweise die Aufrüstung eines Servers wird ab einem Punkt unverhältnismäßig teuer. In den folgenden Abschnitten erfahren Sie, wie Sie dafür sorgen können, dass Ihre Applikation trotz knapp werdender Ressourcen und einer stetig anwachsenden Anzahl von Benutzern weiterhin benutzbar bleibt. Die Lösung für diese Problemstellung liegt in einer soliden Skalierungsstrategie für Ihre Plattform. Wie auch schon beim Deployment können Sie, je nachdem, wie groß Ihre Applikation ist und über wie viel Budget Sie verfügen, auf die verschiedensten Lösungsstrategien zurückgreifen. Zuerst lernen Sie nun Strategien kennen, die sich auf ein System konzentrieren und die Ausnutzung mehrerer Prozessorkerne vorsehen. Im Anschluss daran erfahren Sie, wie Sie Ihre Applikation mit Loadbalancing beziehungsweise mit Cloud-Lösungen über mehrere Systeme verteilen können.

10.2.1 Kindprozesse

Wie Sie bereits wissen, verfolgt Node.js den Ansatz, dass der eigentliche Code Ihrer Applikation in einem Thread ausgeführt wird. Bei einem Benutzer, der auf das System zugreift, führt dies noch zu keinem Problem. Relevant wird die Art und Weise, wie Node.js mit den Prozessorressourcen umgeht, erst, wenn mehrere Anfragen an die Applikation parallel eingehen oder Sie Aufgaben innerhalb Ihrer Applikation haben, die sehr rechenintensiv sind und andere Aufgaben blockieren. Hier kommt der eventgetriebene Ansatz von JavaScript und Node.js ins Spiel. Mit dem `child_process`-Modul haben Sie ein Werkzeug, mit dem Sie dafür sorgen können, dass die Ressourcen Ihres Servers besser ausgenutzt werden. Die Möglichkeiten des `child_process`-Moduls haben Sie bereits in Kapitel 7, »Asynchrone Programmierung«, kennengelernt. In den folgenden Abschnitten erhalten Sie eine kurze Zusammenfassung und einige Hinweise, wie Sie mit JavaScript erfolgreich Teilaufgaben in Ihrer Applikation parallelisieren können.

Das `child_process`-Modul bietet Ihnen mehrere Methoden, mit deren Hilfe Sie abseits des Mainthreads Aufgaben erledigen können. Die wichtigste Methode ist hier die `fork`-Methode. Mit ihrer Hilfe können Sie einen eigenständigen Workerprozess erstellen, der mit seinem Elternprozess über eine bidirektionale Kommunikations-

Verbindungen Nachrichten austauschen kann. Diese Kommunikation funktioniert auf Basis von Events. Das bedeutet, dass Sie eine Callback-Funktion auf ein Event binden können, das vom Eltern- beziehungsweise vom Kindprozess ausgelöst wird. Die `fork`-Methode erhält als erstes Argument den Pfad zu einem Modul, das im Kindprozess ausgeführt werden soll. Das bedeutet, dass sich der Quellcode des Kindprozesses in einer separaten Datei befinden sollte. Bei der Entwicklung von parallel laufenden Threads liefert Ihnen das `child_process`-Modul wenig Unterstützung. Sie müssen sich um die Probleme, die bei paralleler Programmierung auftreten, selbst kümmern.

Ressourcenzugriff

Die erste Schwierigkeit besteht im Zugriff auf Ressourcen. Greifen mehrere Prozesse lediglich lesend auf eine Ressource zu, besteht kein Problem. Schwieriger wird die Situation, wenn schreibender Zugriff ins Spiel kommt. Versuchen zwei Threads gleichzeitig in eine Datei zu schreiben, überschreibt der zweite Thread im schlimmsten Fall die Änderungen des ersten Threads. Diese gingen dabei also verloren. Um dieses Problem zu lösen, sollte ein sogenannter Locking-Mechanismus verwendet werden. Das bedeutet, dass immer nur ein Thread schreibend auf eine Datei zugreifen darf. Ein übliches Mittel für ein derartiges Locking ist die Erstellung von Lock-Dateien. Existiert eine solche Datei, ist dies ein Signal an andere Prozesse, dass diese Ressource exklusiv für schreibende Zugriffe von diesem Prozess aus geöffnet ist. Node.js bringt für diese Problemstellung eine Erleichterung. In Listing 10.21 sehen Sie Quellcode, mit dem Sie eine Datei für exklusiv schreibenden Zugriff öffnen.

```
var fs = require('fs');
fs.open('input.txt', 'wx+', function (e, fd) {
  if (e) throw e;

  console.log(fd);
});
```

Listing 10.21 Datei für exklusiv schreibenden Zugriff öffnen

Binden Sie diese Datei nun als Kindprozess ein und forken diesen Kindprozess zweimal, funktioniert der erste Fork problemlos. Der zweite Fork schlägt mit einer Fehlermeldung fehl. Eine ähnliche Situation wie beim Schreiben von Dateien besteht auch beim Zugriff auf Datenbanken. Auch hier müssen Sie dafür sorgen, dass ein Thread nicht die Änderungen eines anderen überschreibt. Die meisten Datenbanken sperren Tabellen beziehungsweise einzelne Datensätze für parallel schreibenden Zugriff. Das verhindert jedoch nicht, dass die Daten zu einem späteren Zeitpunkt überschrieben werden. In diesem Fall müssen Sie innerhalb Ihrer Software für die Synchronisierung der Informationen sorgen beziehungsweise eine entsprechende Konfliktauflösung bereithalten.

Mehrere parallele Prozesse

In modernen Webapplikationen reicht es in den meisten Fällen nicht aus, nur einen bestimmten Teil der Applikationslogik mit einem separaten Workerprozess abzudecken. Nähere Informationen zum Thema asynchroner Programmierung mit Node.js finden Sie in Kapitel 7. Stattdessen müssen Sie in vielen Fällen mehrere Teile Ihrer Applikation auslagern, die sich oft parallelisieren lassen. Aus dieser Konstellation entsteht die Schwierigkeit, dass Sie dafür sorgen müssen, dass der Quellcode Ihrer Applikation, der nach diesen parallelen Aufgaben ausgeführt werden soll, auch wirklich erst ausgeführt wird, sobald sämtliche Aufgaben erledigt sind, die dafür notwendig sind. In Listing 10.22 sehen Sie, wie eine Lösung für dieses Problem aussehen kann.

```
var childProcess = require('child_process');

var child1 = childProcess.fork('logic1.js'),
    child2 = childProcess.fork('logic2.js');

var count = 0;

function handleResults() {
    // wird nach child1 und child2 ausgeführt
}

child1.on('message', function (data) {
    count +=1;

    // logic to handle child1 result

    if (count >= 2) {
        // handle final results
        handleResults();
    }
});

child2.on('message', function (data) {
    count +=1;

    // logic to handle child2 result

    if (count >= 2) {
        // handle final results
        handleResults();
    }
});
```

Listing 10.22 Parallele Prozesse zusammenführen

In Listing 10.22 werden zwei Kindprozesse erstellt, in denen Logik ausgeführt wird. Beide Kindprozesse senden eine Nachricht an den Elternprozess, sobald sie ihre Verarbeitung abgeschlossen haben. Im Elternprozess wird entsprechend eine Callback-Funktion auf das `message`-Event gebunden. In dieser wird ein Zähler verwendet, um zu prüfen, ob beide Kindprozesse bereits abgeschlossen sind. Ist dies der Fall, wird die Funktion `handleResults` ausgeführt. Benötigen Sie das Ergebnis eines oder beider Kindprozesse, sollten Sie dies in einer Variablen speichern, auf die Sie in beiden Callback-Funktionen Zugriff haben. Auf diese Weise können Sie die Ausführung der Programmlogik innerhalb Ihrer Applikation sehr flexibel steuern und eine maximale Parallelisierung Ihrer Workflows erreichen.

Sonstiges

Nutzen Sie das `child_process`-Modul zur Parallelisierung in Ihrer Applikation, sollten Sie darauf achten, dass Sie das Forken von Kindprozessen unter Kontrolle halten. Werden zu viele Kindprozesse erstellt, hat dies negative Auswirkungen auf Ihr System. Jeder Kindprozess benötigt Arbeitsspeicher und Rechenzeit des Prozessors. Übersteigen die Ressourcenforderungen die vorhandenen Ressourcen, wird Ihr System langsamer, und Sie können die Vorteile der Kindprozesse nicht ausnutzen.

Setzen Sie innerhalb Ihrer Applikation auf Kindprozesse, haben Sie die Möglichkeit, einen einmal gestarteten Kindprozess für mehr als nur eine Berechnung zu verwenden. Diese Vorgehensweise hat den Vorteil, dass der Kindprozess nicht erst gestartet werden muss. Sie eignet sich jedoch nur, falls Sie häufig ähnliche Berechnungen durchführen lassen. Die Node.js-Plattform bietet für derartige Problemstellungen weitere Module, die Sie in Ihre Applikation einbauen können. Zu diesen Modulen zählt unter anderem das `child_process`- oder auch das `cluster`-Modul. In Kapitel 7, »Asynchrone Programmierung«, haben Sie bereits ein detailliertes Beispiel zu diesem Thema gesehen und damit die Möglichkeit kennengelernt, das Loadbalancing Ihres Servers an Ihr Betriebssystem auszulagern.

10.2.2 Loadbalancer

Es gibt allerdings auch Fälle, in denen Sie bei der Skalierung Ihrer Applikation an die Grenzen Ihres Systems stoßen. In diesem Fall bleibt Ihnen nur die Möglichkeit, zusätzliche Systeme einzubinden, um die Last Ihres Systems zu verteilen. Sie müssen Ihre Applikation allerdings so aufbauen, dass sie mit dieser Art der Infrastruktur umgehen kann. Das bedeutet für Sie konkret, dass nicht jede Applikation durch das Einfügen eines weiteren Servers skaliert werden kann. Grundsätzlich sollten die Instanzen Ihrer Applikation, die auf den verschiedenen Servern laufen, möglichst unabhängig voneinander sein und eine gemeinsame Datenbank zur Synchronisierung der Stände verwenden. Haben Sie diese Bedingungen erfüllt, benötigen Sie eine

weitere Software, die dafür sorgt, dass die Anfragen an Ihre Applikation entsprechend auf die verfügbaren Server verteilt werden. Es gibt zahlreiche Systeme, die diese Aufgabe übernehmen können. In den folgenden Abschnitten lernen Sie zwei verschiedene Loadbalancer kennen: HAproxy und Nginx.

HAproxy

Sie können HAproxy als frei verfügbaren Loadbalancer sowohl für Applikationen verwenden, die auf HTTP-Verbindungen basieren, als auch für TCP-Verbindungen. Die Seite dieses Projekts finden Sie unter <http://haproxy.1wt.eu/>. Diese Software ist darauf ausgelegt, als eine hochperformante Schnittstelle zwischen Ihren Benutzern und Ihrer Applikation zu fungieren. In dieser Rolle kann HAproxy mehrere Zehntausend eingehende Verbindungen verwalten. Die Software ist vor allem für UNIX-Systeme wie Linux und Solaris verfügbar. Unter Windows kann HAproxy über Umwege mithilfe von Cygwin ebenfalls installiert werden. Die Konfiguration des Loadbalancers erfolgt wie bei UNIX-Programmen üblich über eine Konfigurationsdatei. Einen Auszug einer solchen Konfigurationsdatei sehen Sie in Listing 10.23.

```
listen http 192.168.0.1:8080
    mode tcp
    option tcplog
    balance roundrobin
    maxconn 10000
    server web01 192.168.0.2:8080 maxconn 5000
    server web02 192.168.0.3:8080 maxconn 5000
```

Listing 10.23 Konfiguration von HAproxy

Mit der `listen`-Direktive definieren Sie den Proxyserver. Die beiden `server`-Angaben geben die Systeme an, an die die Anfragen weitergeleitet werden. Mithilfe von `maxconn` legen Sie fest, wie viele Anfragen der Loadbalancer akzeptiert. Die `maxconn`-Angaben bei den jeweiligen Servern geben an, wie viele Verbindungen jeweils an die Server weitergeleitet werden. Wie HAproxy entscheidet, wann welches System welche Verbindung erhält, definieren Sie über den Algorithmus, der verwendet wird. In Listing 10.23 wird Round Robin als Algorithmus eingesetzt. Hierbei werden die Anfragen gleichmäßig auf die Server verteilt. Dieser Algorithmus erlaubt es Ihnen außerdem, die verschiedenen Server zu gewichten. Je nach Gewichtung erhalten Server mehr oder weniger Anfragen zugeteilt. Fällt ein Server aus, werden die Anfragen auf die übrigen Server verteilt. Das bedeutet, dass der Ausfall einer Maschine das Gesamtsystem nicht negativ beeinflusst. Ist die Maschine zu einem späteren Zeitpunkt wieder verfügbar, wird sie wieder integriert und erhält erneut Anfragen. Mit dieser Architektur steht Ihnen ein hochperformantes System mit größtmöglicher Ausfallsicherheit zur Verfügung.

Ein weiteres System zum Loadbalancing von Anfragen stellt Nginx dar. Der nächste Abschnitt soll Ihnen zeigen, wie Sie bei der Verwendung dieses Systems vorgehen.

Nginx

Nginx ist vor allem als leichtgewichtiger und performanter Webserver bekannt. Diese Software bietet allerdings wesentlich mehr als die reine Webserverfunktionalität. Mit Nginx können Sie eine ähnliche Infrastruktur implementieren wie schon mit HAproxy. Nginx ist allerdings nicht so spezialisiert auf Proxy- und Loadbalancing-Funktionalität wie HAproxy. In Listing 10.24 sehen Sie die relevanten Auszüge aus der Konfigurationsdatei von Nginx, die dafür sorgen, dass Nginx zu einem Loadbalancer wird.

```
http {
    ...
    upstream node_upstream {
        server 192.168.0.2:8080 weight=5;
        server 192.168.0.3:8080;
    }
    ...
    server {
        ...
        location / {
            ...
            proxy_pass http://node_upstream
        }
        ...
    }
}
```

Listing 10.24 Konfiguration eines Nginx-Loadbalancers

Damit Ihre Nginx-Instanz zu einem Loadbalancer werden kann, benötigen Sie die Unterstützung eines Moduls von Nginx, und zwar das *HttpProxyModule*. Zu diesem Zweck definieren Sie eine *upstream*-Ressource in Ihrer Konfigurationsdatei und geben hier die Server an, an die Sie die Last verteilen möchten. Im nächsten Schritt erweitern Sie Ihre Standard Location um die Option *proxy_pass*, die als Wert den Namen der *upstream*-Ressource erhält. Wie auch schon bei HAproxy können Sie die verschiedenen Server gewichten. Mithilfe der Angabe von *weight*, die Sie beim ersten Server sehen können, weisen Sie einem Server eine bestimmte Gewichtung zu. Die Angabe des Wertes 5 bedeutet, dass der erste Server fünf Anfragen erhält und der zweite lediglich eine. Als Algorithmen für das Loadbalancing bietet Nginx Round Robin, Least Connections und IP-Hash, wobei Round Robin die Standardeinstellung

ist. Mit Nginx können Sie wie auch schon mit HAProxy eine sehr performante Infrastruktur für Ihre Applikation aufbauen.

Setzen Sie bei der Skalierung Ihrer Applikation auf einen Loadbalancer vor Ihren Servern, bedeutet das für Sie, dass Sie Ihre Infrastruktur selbst verwalten müssen. Das hat zur Konsequenz, dass Sie dafür sorgen müssen, dass die Hard- und Software für die Infrastruktur zur Verfügung stehen muss. Bei Ausfällen haben Sie dafür Sorge zu tragen, dass Sie die Server wieder reparieren. Eine Alternative zum eigenen Betrieb der Infrastruktur ist, dass Sie diese Verantwortung an einen externen Dienstleister übergeben. Eine konkrete Möglichkeit besteht darin, dass Sie Ihre Applikation in die Cloud bringen. Im folgenden Abschnitt sehen Sie, welche Möglichkeiten Sie hierbei haben.

10.2.3 Node in der Cloud

Der Begriff Cloud ist mittlerweile vor allem bei Webapplikationen weit verbreitet. Eine eindeutige Definition für dieses Schlagwort existiert jedoch nicht. So kann Cloud sowohl als Cloud Storage interpretiert werden als auch als Cloud Computing und sogar bis hin zu Plattform as a Service-Dienstleistungen, bei denen Sie auf die verschiedenen Ausprägungen von Clouds zurückgreifen können. Es existiert eine Vielzahl von Cloud-Dienstanbietern auf dem Markt. Im Rahmen dieses Abschnitts erhalten Sie einen kurzen Überblick über zwei dieser Plattformen. Zum einen ist das Heroku, eine eigenständige Cloud Application-Plattform, und zum anderen Windows Azure, die Cloud-Plattform von Microsoft.

Heroku

Damit Sie Ihre Applikation in die Heroku Cloud deployen können, benötigen Sie eine Hilfsapplikation mit dem Namen *Heroku Toolbelt*. Diese muss auf Ihrem System installiert sein, damit Ihnen das `heroku`-Kommandozeilenwerkzeug zur Verfügung steht. Mit diesem Werkzeug können Sie dann auch Ihren Heroku-Account erstellen, mit dem Sie später Ihre Applikation in die Cloud laden. Weitere Informationen und Ressourcen zu Heroku finden Sie unter <https://www.heroku.com/>. Eine wichtige Voraussetzung, damit Heroku Ihre Applikation korrekt deployen kann, ist das Vorhandensein einer *package.json*-Datei, wie Sie sie auch für ein reguläres NPM-Paket erstellen müssen. In dieser Datei müssen Sie unter anderem die Abhängigkeiten Ihrer Applikation auflisten. Außerdem benötigen Sie eine Datei mit dem Namen *Procfile*, mit deren Hilfe Sie definieren, welcher Befehl verwendet wird, um Ihre Applikation zu starten. Listing 10.25 zeigt Ihnen ein Beispiel einer *Procfile*.

```
web: node server.js
```

Listing 10.25 Beispiel einer Procfile-Datei

Die *Procfile* aus Listing 10.25 sorgt dafür, dass auf dem Server die Applikation mit dem Kommando `node server.js` gestartet wird. Sind Ihre Abhängigkeiten lokal installiert, können Sie mit dem Befehl `foreman start` Ihre Applikation lokal starten. Das Kommando `foreman` ist ein Bestandteil der Heroku Toolbelt-Applikation. Im nächsten Schritt sollten Sie Ihre Applikation in ein Git Repository hinzufügen. Mit den beiden Kommandos, die Sie in Listing 10.26 sehen, deployen Sie Ihre Applikation in die Heroku Cloud.

```
$ heroku create
...
$ heroku push heroku master
...
$ heroku ps:scale web=1
$ heroku ps
...
$ heroku open
```

Listing 10.26 Deployment in die Heroku Cloud

Mit dem Kommando `heroku ps:scale web=1` stellen Sie sicher, dass lediglich ein Prozess gestartet wird. Das `heroku ps`-Kommando zeigt Ihnen daraufhin den aktuellen Stand Ihrer Applikation. Mit `heroku open` öffnen Sie schließlich Ihre Applikation im Browser. Nach dieser sehr einfachen Einführung in Heroku erfahren Sie im nächsten Abschnitt, wie Sie Ihre Node.js-Applikation in der Windows Azure Cloud deployen können.

Windows Azure

Ähnlich wie auch schon bei Heroku benötigen Sie auch bei Windows Azure einen Account, bevor Sie den Dienst in Anspruch nehmen können. Einen Einstieg in die Welt von Windows Azure finden Sie unter <http://www.windowsazure.com/>. Sie können Ihren Account über das *Windows Azure Management*-Portal erstellen. Auf dieser Plattform können Sie über einen Assistenten eine neue Website generieren, Sie müssen dafür lediglich die URL und eine entsprechende Region auswählen. Danach können Sie Git Publishing aktivieren, um den Quellcode Ihrer Applikation per Git zu deployen. Sie müssen in diesem Fall noch einen Benutzernamen und ein Passwort konfigurieren. Nach dieser Konfiguration erhalten Sie die wichtigste Information über Ihr Repository: die URL. Mit diesen Informationen können Sie nun an die Entwicklung Ihrer Applikation gehen. In Listing 10.27 sehen Sie, welche Schritte Sie noch unternehmen müssen, um Ihren Code in die Cloud zu bringen.

```
$ git init
$ git add .
$ git commit -m "added source code"
$ git remote add azure https://user@test.scm.azurewebsites.net/
$ git push azure master
```

Listing 10.27 Quellcode in die Cloud bringen

Wie Sie anhand der beiden Beispiele gesehen haben, ist es problemlos möglich, Ihre Node.js-Applikation in die Cloud zu bringen. Die Plattformen unterstützen Sie bei diesen Aufgaben entweder direkt durch Werkzeuge oder mit einer Online-Plattform.

10.3 Zusammenfassung

Dieses Kapitel widmete sich der Problemstellung, wie Ihre Applikation auf das Zielsystem gelangt. Angefangen von der einfachsten Form des Deployments über das Kopieren des Quellcodes bis hin zum Deployment über den Node Package Manager. Der zweite Teil dieses Kapitels beschäftigte sich damit, wie Sie mit erhöhtem Daten- und Nutzeraufkommen in Ihrer Applikation umgehen. Die Möglichkeiten reichen dabei von der lokalen Skalierung per verfügbarer Prozessorkerne über die Verwaltung der eigenen Infrastruktur hinter einem Loadbalancer bis hin zum Deployment Ihrer Applikation in der Cloud. Im nächsten Kapitel dieses Buches erfahren Sie, wie Sie den HTTP-Server von Node.js in der Praxis einsetzen können. Dies wird anhand des konkreten Beispiels eines REST-Servers gezeigt.

Kapitel 11

Sicherheitsaspekte

Man kann keinen Eierkuchen backen, ohne ein paar Eier zu zerschlagen.

– Napoleon Bonaparte

Neben der Performance und einer guten, intuitiven Benutzbarkeit ist die Sicherheit ein weiterer wesentlicher Aspekt von Webapplikationen. Sicherheit ist dabei ein Überbegriff für eine Vielzahl einzelner Disziplinen, die allerdings alle die Erhaltung und Verbesserung der Vertrauenswürdigkeit der Applikation gegenüber dem Nutzer zum Ziel haben.

Angriffe auf eine Applikation und deren Nutzer können vielfältig sein. Die Spanne reicht vom Ausspionieren persönlicher Daten wie Adressen, Bank- oder Kreditkarteninformationen über die Ausführung von Schadsoftware bis hin zur Abschaltung bestimmter Dienste durch gezielte Überlastung einer Anwendung.

Wie für alle Programmiersprachen, die Ihnen Zugriff auf Systemressourcen bieten, gilt auch für serverseitiges JavaScript mit Node.js, dass Sie sich Ihrer Verantwortung bewusst sein sollten. Mit Schwachstellen in Ihrer Applikation öffnen Sie Angreifern ein Tor zu Ihrem System, durch das sie Ihr System beschädigen oder unter Kontrolle bringen können. Diese Tatsache stellt auch einen der bedeutendsten Unterschiede zum clientseitigen JavaScript dar. Im Browser bewegen Sie sich mit Ihrer JavaScript-Applikation innerhalb einer Sandbox, von der aus Sie im Normalfall keinerlei Schnittstellen nach außen auf das Gesamtsystem des Benutzers haben. Im schlimmsten Fall meldet der Browser, dass ein Script zu lange läuft, und bietet Ihnen an, die Ausführung dieses Scripts zu unterbrechen.

In diesem Kapitel lernen Sie die verschiedenen Möglichkeiten von Angriffen auf eine Webapplikation kennen und welche Maßnahmen Sie als Entwickler und Betreiber einer solchen Applikation unternehmen sollten, um Ihre Nutzer zu schützen.

11.1 Filter Input und Escape Output

Es gibt zwei grundsätzliche Vorgehensweisen, wenn es um die Absicherung von Webapplikationen geht. Diese heißen *Filter Input* und *Escape Output*.

Als Erstes lernen Sie das Konzept Filter Input näher kennen. Für Eingaben bedeutet das generell, dass diese als nicht vertrauenswürdig eingestuft werden, vor allem wenn sie von einem Benutzer stammen. Das heißt, Sie bringen Ihren Benutzern grundsätzlich kein Vertrauen entgegen. Sämtliche Eingaben durchlaufen einen Filterprozess und werden erst im Anschluss daran weiterverarbeitet.

Für die Filterung von Eingaben gibt es nun zwei verschiedene Vorgehensweisen. Sie können ungültige Daten entweder so umformulieren, dass sie zu gültigen Daten werden. Das beinhaltet allerdings das Risiko, dass Sie dabei Informationen verfälschen oder sogar ganz verlieren. Die zweite Variante besteht darin, dass Sie ungültige Eingaben mit einer entsprechenden Fehlermeldung zurückweisen und eine erneute, korrekte Eingabe anfordern.

Das zweite Sicherheitsprinzip Escape Output betrifft die Daten, die aus Ihrer Applikation herauslaufen. Ziel ist es, keinen Schadcode weiterzugeben. Das wirft allerdings zwei Fragen auf: »Welche Systeme sind potenzielles Ziel für Daten?« und »Was wird von diesen Systemen als Schadcode betrachtet?«. Die erste Frage lässt sich leicht beantworten: Die Spanne der Zielsysteme reicht vom Webbrowser eines Benutzers bis hin zu einem Datenbankserver, was gleich zur Beantwortung der zweiten Frage führt. Schadcode ist für jedes System individuell. Wo für einen Browser eingeschleuster JavaScript-Code einen potenziellen Schaden für den Benutzer bedeutet, ist ein gültiger Datenbankbefehl für einen Datenbankserver eine weitaus größere Gefahr. Aus diesem Grund müssen Sie von Fall zu Fall entscheiden, wie Sie mit den Ausgabedaten Ihrer Applikation umgehen. In den folgenden Abschnitten erfahren Sie mehr über mögliche Sicherheitsrisiken und den Umgang damit.

11.2 Absicherung des Servers

Die folgenden Abschnitte beschäftigen sich mit der Absicherung des Servers. Konkret bedeutet der Begriff Absicherung in diesem Fall die Sicherstellung der kontinuierlichen Lauffähigkeit der Anwendung und den Schutz des Systems vor Angreifern.

11.2.1 Benutzerberechtigungen

Ein Dienst oder eine Anwendung werden stets unter einem bestimmten Nutzer ausgeführt. Starten Sie Ihre Applikation auf der Kommandozeile, so verfügt die Applikation über dieselben Zugriffsberechtigungen auf das Dateisystem wie der aktuell angemeldete Benutzer. Das bedeutet, dass die Applikation die gleichen Dateien erstellen, modifizieren und löschen kann wie Sie selbst.

Grundsätzlich stellt diese Tatsache noch kein Problem dar. Die Zugriffsberechtigungen werden allerdings zum Problem, wenn Sie über Administratorberechtigungen

verfügen. In diesem Fall hat Node.js Zugriff auf das gesamte System. Die Bedeutung des Nutzers, unter dem Sie Ihre Applikation ausführen lassen, wird schnell klar, wenn Sie sich die Berechtigungen bestimmter Systemverzeichnisse ansehen.

Unter UNIX-Systemen beinhaltet beispielsweise das `/bin`-Verzeichnis die meisten Kommandos, die Sie als normaler Nutzer ausführen können. Löschen Sie dieses Verzeichnis, ist das System nur noch begrenzt nutzbar. Noch verheerender ist der Fall, wenn Sie das `/etc`-Verzeichnis, das sämtliche Konfigurationsdateien des Systems enthält, löschen. Damit die Dateien in diesen Verzeichnissen nicht ohne Probleme von jedem Nutzer gelöscht werden können, haben Sie auf die meisten dieser Dateien lediglich die Berechtigung, sie zu lesen und auszuführen, nicht aber zu schreiben.

Solange Sie keine Schreibberechtigung auf Verzeichnisebene besitzen, können Sie innerhalb dieses Verzeichnisses auch keine Dateien löschen. Haben Sie die Berechtigung, in einzelne Dateien zu schreiben, können Sie diese allerdings überschreiben, was einem Löschen gleichkommt. Listing 11.1 demonstriert Ihnen diese Tatsache. Sie sollten die Kommandos aus diesem Beispiel allerdings lediglich auf einem Testsystem und keinesfalls auf einem kritischen System ausführen.

```
$ rm -rf /bin
rm: /bin/: Permission denied
rm: /bin/: Permission denied
...
```

Listing 11.1 Wirkung von Zugriffsrechten

Ganz konkret bedeutet das für Ihre Applikation, dass Sie diese niemals mit Administratorberechtigungen ausführen sollten. Als Standard hat sich etabliert, dass Sie für eine bestimmte Applikation einen separaten Benutzer anlegen und die Applikation unter diesem Benutzer mit dessen Berechtigungen ausführen. Benötigen Sie Zugriff auf bestimmte Ressourcen, können Sie dies durch die Mitgliedschaft des Benutzers in bestimmten Gruppen abdecken, ohne dass Sie ihm umfangreichen Zugriff auf das System gewähren müssen.

Ein weiteres Problem kann durch die Architektur von Node.js entstehen. Der Single-Threaded-Ansatz der Plattform bietet Angreifern einen Ansatzpunkt, um Ihre Applikation zu beeinflussen.

11.2.2 Single-Threaded-Ansatz

Das Prinzip der Node.js-Plattform lautet, dass sämtliche I/O-Operationen asynchron ausgeführt werden. Lediglich Ihr Applikationscode läuft in einem Thread. Verarbeiten Sie Anfragen von Clients direkt in Ihrem Applikations-Thread, bedeutet das, dass

rechenintensive Anfragen das Potenzial haben, die Ausführung Ihrer Applikation zu blockieren. Listing 11.2 soll Ihnen zur Verdeutlichung dieser Problemstellung dienen.

```
require('http').createServer(function (req, res) {
  var number = require('url').parse(req.url, true).query.number,
      result = 1,
      start = new Date().getTime();

  if (number !== undefined) {
    for (var i = 1; i <= number; i++) {
      result += i * i;
    }
  }

  var end = new Date().getTime();

  res.end('Time : ' + (end - start) + ' Result: ' + result);
}).listen(8080);
```

Listing 11.2 Blockierende Anfrage

Im Beispiel in Listing 11.2 sehen Sie eine einfache Implementierung eines Webservers. Der Code macht nichts weiter, als Anfragen von Clients zu akzeptieren und, falls ein Parameter *number* übergeben wurde, eine Berechnung durchzuführen. Starten Sie nun von einem Browser aus eine Anfrage über die URL *http://localhost:8080/?number=100*, sollten Sie eine Ausgabe wie in Listing 11.3 erhalten.

```
Time: 0 Result: 338351
```

Listing 11.3 Ausgabe bei einer Anfrage mit 100 Durchläufen

Die Antwort erhalten Sie in diesem Fall sehr schnell. Diese Art von Anfrage stellt für den Webserver kein größeres Problem dar. Passen Sie allerdings im nächsten Schritt die URL auf den Wert *http://localhost:8080/?number=100000000* an, müssen Sie bereits bedeutend länger warten. Nach einigen Sekunden erhalten Sie eine Ausgabe in Ihrem Browser, die der in Listing 11.4 ähneln sollte.

```
Time: 11467 Result: 3.333333383334632e+23
```

Listing 11.4 Ausgabe bei einer Anfrage mit 100.000.000 Durchläufen

Da die Applikation in diesem Fall keinerlei Beschränkung aufweist, können Sie den Parameter beliebig erhöhen. Handelt es sich nun nicht nur um einzelne Anfragen, sondern fragen gleichzeitig mehrere Clients beim Server an, blockieren derartige Anfragen die gesamte Applikation.

Für diese Problemstellung existieren zwei Teillösungen. Der erste Ansatz besteht darin, sämtliche Argumente, die Grundlage für umfangreiche Berechnungen auf Serverseite sind, durch einen bestimmten Wertebereich zu beschränken. Der zweite Teil der Lösung ist, derartige Berechnungen in einen separaten Prozess auszulagern, sodass der Haupt-Thread nicht durch Berechnungen blockiert wird. Eine entsprechende Implementierung der skizzierten Lösung sehen Sie in Listing 11.5.

```
var child_process = require('child_process');

if (process.argv[2] === 'child') {
  var number = process.argv[3],
      result = 1,
      start = new Date().getTime();

  if (number !== undefined) {
    for (var i = 1; i <= number; i++) {
      result += i * i;
    }
  }

  var end = new Date().getTime();

  process.send({time: end - start, result: result});
} else {
  require('http').createServer(function (req, res) {

    var number = require('url').parse(req.url, true).query.number;

    if (number < 0 || number > 1000) {
      res.end('Please provide a number between 0 and 1000');
    }

    var child = child_process.fork(__filename, ['child', number]);

    child.on('message', function (data) {
      res.end('Time: ' + data.time + ' Result: ' + data.result);
    });
  }).listen(8080);
}
```

Listing 11.5 Umgang mit problematischen Anfragen

Bei allen Vorteilen der Auslagerung in Kindprozesse sollten Sie dennoch beachten, dass auch diese Technik ihre Nachteile mit sich bringt. Die Kindprozesse benötigen ihrerseits ebenfalls Systemressourcen, sodass die maximale Anzahl von Kindprozessen durch die verfügbaren Ressourcen begrenzt ist. Auch an diese Gefahr sollten Sie bei der Umsetzung Ihrer Applikation immer denken. Im Zweifelsfall ist es immer auch eine Option, Anfragen, die über die Kapazität Ihres Systems hinausgehen, abzuweisen, bevor Ihr gesamtes System nicht mehr funktioniert.

Wie Sie in der Umsetzung in Listing 11.5 sehen können, erfordert eine Absicherung gegen bestimmte Angriffsarten immer etwas mehr Arbeit als die Implementierung einer ungesicherten Applikation. Sie sollten allerdings stets bedenken, dass sich diese Einsparung zu einem späteren Zeitpunkt im Lebenszyklus Ihrer Software rächen kann – spätestens, wenn Ihre Applikation durch einen derartigen Angriff außer Betrieb gesetzt wird.

11.2.3 Denial of Service

Eine weitere Methode, über die externe Angreifer Ihre Applikation bis hin zur Unerreichbarkeit für Ihre Nutzer beeinflussen können, sind verschiedene Varianten von Denial of Service-Attacken. Bei dieser Art von Angriffen werden die Dienste Ihrer Anwendung überbeansprucht und damit so viele Ressourcen in Anspruch genommen, dass der Dienst nicht mehr zuverlässig zur Verfügung gestellt werden kann.

Gegen derartige Angriffe können Sie allerdings recht wenige Gegenmaßnahmen innerhalb einer Applikation unternehmen. Gehen die Anfragen von einer Quelle, also von einer IP-Adresse, aus, haben Sie die Möglichkeit, die Anfragen dieser Quelle direkt abzuweisen.

```
var http = require('http');

var sources = {},
    timeThreshold = 10000,
    countLimit = 10;

http.createServer(function (req, res) {
    var client = req.connection.remoteAddress,
        now = new Date().getTime();

    if (sources[client]) {
        if (now - sources[client].time < timeThreshold && sources[client]
            .count
                > countLimit) {
            res.statusCode = 400;
        }
    }
})
```

```

        res.end('Bad Request');
        return false;
    } else if (now - sources[client].time > timeThreshold) {
        sources[client] = {time: now, count: 1};
    } else {
        sources[client].count += 1;
    }
} else {
    sources[client] = {time: now, count: 1};
}

res.end('Hello Client');
}).listen(8080);

```

Listing 11.6 Abweisen bestimmter Anfragen

Der Sourcecode in Listing 11.6 sorgt dafür, dass von jeder IP-Adresse nur zehn Anfragen pro zehn Sekunden kommen können. Jede Anfrage, die diesen Rahmen überschreitet, wird automatisch mit dem Status 400 abgewiesen. Nach zehn Sekunden werden die Werte wieder zurückgesetzt. Da die Konfigurationsoptionen in separaten Variablen vorgehalten werden, ist es einfach möglich, diese Direktiven zu ändern.

Eine bessere Lösung besteht darin, dass Sie vor Ihren Node.js-Server eine Firewall schalten. Diese bietet in den meisten Fällen bereits Mechanismen, um mit derartigen Angriffen umzugehen, und weist automatisch Anfragen ab, die für DOS-Attacken infrage kommen.

Innerhalb einer Node.js-Applikation ist jedoch nicht nur der Server selbst anfällig für Angriffe. Auch Datenbanken werden oft Ziel von Attacken. Die nächsten Abschnitte zeigen Ihnen, welchen Schaden Angreifer in Ihrer Datenbank anzurichten in der Lage sind und welche Maßnahmen Sie dagegen ergreifen können.

11.2.4 SQL-Injections

Sobald Nutzer Ihre Applikation verwenden, generieren sie Informationen. Diese Informationen werden im Normalfall in einer Datenbank festgehalten. Die Datenbasis einer Applikation stellt ihr Herzstück und damit das eigentliche Kapital dar. Aus diesem Grund sollten Sie Ihre Datenbank immer besonders vor Angriffen von außen schützen.

Mit den sogenannten SQL-Injections zielen externe Angreifer auf das Datenbank-Backend Ihrer Applikation ab. Mit dieser Art von Attacke wird die Möglichkeit von SQL-Datenbanken ausgenutzt, Befehle abzusetzen. Listing 11.7 zeigt Ihnen eine Applikation, die anfällig für SQL-Injections ist.

```

var http = require('http'),
    mysql = require('mysql');

var connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'root',
  password  : '',
  database  : 'test',
  multipleStatements: true
});

http.createServer(function (req, res) {
  var id = require('url').parse(req.url, true).query.id;

  if (id != undefined) {
    var sql = 'SELECT * FROM users WHERE id = ' + id;

    connection.query(sql, function (err, result) {
      var data = result[0];
      res.end('ID: ' + data.id + ' name: ' + data.name);
    });
  }
}).listen(8080);

```

Listing 11.7 Unsichere SQL-Applikation

Starten Sie die Applikation aus Listing 11.7, und rufen Sie von Ihrem Browser aus die URL `http://localhost:8080/?id=1;truncate%20users`; auf, hat dies zur Folge, dass Sie die gesamte `users`-Tabelle leeren. Die Ausführung derartiger Kommandos ist typisch für SQL-Injections. Eine weitere Variante von SQL-Injections ist das unberechtigte Auslesen von Daten.

Das `mysql`-Paket für Node.js weist einige Sicherheitsmechanismen auf, mit deren Hilfe SQL-Injections verhindert werden können. Ein erster Schritt besteht darin, dass Sie in der Konfiguration der Verbindung den Wert `multipleStatements` auf seinem Standardwert `false` belassen. Dadurch wird verhindert, dass innerhalb einer Abfrage mehrere Befehle abgesetzt werden können.

Eine weitere Gegenmaßnahme besteht darin, dass Sie die Eingabewerte für Abfragen escapen. Sie müssen sich allerdings nicht selbst um das korrekte Escaping kümmern. Die `escape`-Methode des `connection`-Objekts übernimmt diese Aufgabe für Sie. Schreiben Sie das Statement für die Abfrage wie in Listing 11.8 um, kann auch hier kein Schadcode mehr eingeschleust werden.

```
var sql = 'SELECT * FROM users WHERE id = ' + connection.escape(id);
```

Listing 11.8 Sichere Abfragen durch Escaping

Mit diesen einfachen Maßnahmen erhöhen Sie den Schutz Ihrer Applikation um ein Vielfaches, indem Sie es potenziellen Angreifern schwerer machen, Kommandos auf Datenbankebene abzusetzen.

11.2.5 Eval

Der ECMAScript-Standard definiert die Funktion `eval`. Mit ihr sind Sie in der Lage, eine Zeichenkette als JavaScript interpretieren und ausführen zu lassen. So hilfreich diese Funktion auch erscheinen mag, so viel Risiko und Angriffspotenzial bietet sie. Sie sollten vor allem darauf achten, dass Sie keinem Benutzer die unkontrollierte Ausführung von JavaScript-Code erlauben.

Im ungünstigsten Fall kann ein Benutzer Ihrer Applikation eine Zeichenkette einschleusen, die dann durch `eval` ausgewertet und ausgeführt wird. Listing 11.9 demonstriert Ihnen einen derartigen Fall, bei dem es einem Angreifer möglich ist, durch eine Serveranfrage Schadcode auszuführen.

```
var http = require('http');

http.createServer(function (req, res) {
  "use strict";
  if (req.method === 'POST') {
    var command = '';
    req.on('data', function (data) {
      command += data.toString();
    });
    req.on('end', function () {
      eval(command);
      res.end();
    });
  }
}).listen(8080);
```

Listing 11.9 Sicherheitslücken durch eval

Die Sicherheitslücke sticht in diesem Beispiel geradezu heraus. Einem Angreifer wird durch dieses Stück Quellcode die Möglichkeit gegeben, beliebigen Quellcode, den er mit einem POST-Request eingeschleust hat, auszuführen. Einen derartigen POST-Request sehen Sie in Listing 11.10.

```
curl -X POST --data "command=console.log('This could harm your system')
" localhost:8080
```

Listing 11.10 Einschleusen von Schadcode

Führen Sie den Quellcode aus Listing 11.9 aus und setzen dann auf der Kommandozeile den Request aus Listing 11.10 ab, wird das `console.log`-Kommando ausgeführt. Damit lässt sich noch kein direkter Schaden anrichten. Wenn Sie sich allerdings vor Augen führen, dass Sie über sämtliche Berechtigungen verfügen, die der Benutzer hat, unter dem die Applikation ausgeführt wird, wird das volle Ausmaß des Risikos absehbar. In Kombination mit weiteren Antipatterns wie beispielsweise der Ausführung von Applikationen mit Administratorberechtigungen kann diese Sicherheitslücke verheerende Folgen haben.

Eine weitere Möglichkeit, die Sicherheit Ihrer Applikation etwas zu erhöhen, ist die Verwendung des Strict Modes. Diesen aktivieren Sie, indem Sie innerhalb einer Funktion die Zeichenkette `use strict` angeben, wie Sie in Listing 11.9 sehen können. Dieser Strict Mode sorgt dafür, dass Sie innerhalb des Quellcodes, den Sie mit `eval` ausführen, keine neuen Variablen initialisieren können.

Eine mögliche Maßnahme gegen dieses Risiko ist, dass Sie komplett auf die Verwendung von `eval` verzichten und die Features, auf die ein Benutzer Ihrer Applikation zugreifen kann, direkt als Methoden zur Verfügung stehen, die entsprechend der Benutzeranfragen ausgeführt werden. So können Sie das Risikopotenzial innerhalb Ihrer Applikation erheblich senken.

11.2.6 Method Invocation

Eine weitere Gefahrenstelle, wenngleich nicht so schwerwiegend wie die Codeausführung mit `eval`, stellt die von Benutzern ausgelöste Methodenausführung dar. Im Normalfall verursacht eine Anfrage eines Clients beim Server die Ausführung eines ganz bestimmten Teils des Quellcodes der Applikation. In diesem Fall haben Sie als Betreiber der Applikation die Kontrolle darüber, welche Routinen die Benutzer Ihrer Applikation ausführen können und welche Teile der Applikation nicht oder nicht direkt erreichbar sind. In JavaScript haben Sie allerdings auch die Möglichkeit, die Namen der Methoden, die Sie aufrufen, dynamisch anzugeben. Listing 11.11 verdeutlicht diesen Sachverhalt.

```
myObj[methodName]();
```

Listing 11.11 Variable Methodennamen

Die Variable `methodName` aus Listing 11.11 kann den Wert jeder in `myObj` verfügbaren Methode annehmen. Insgesamt sorgt der Quellcode in Listing 11.11 dann dafür, dass

diese Methode aufgerufen wird. Gehen Sie nun von diesem Quellcode aus und verbinden ihn mit einem HTTP-Server, über den dann die Nutzer Ihrer Applikation dynamisch Methoden auf bestimmte Objekte aufrufen können, hat dies Potenzial für eine Sicherheitslücke innerhalb Ihrer Applikation.

```
require('http').createServer(function (req, res) {
  var reqData = require('url').parse(req.url, true).query,
      user = new User(reqData.id);

  res.end(user[reqData.method]());

}).listen(8080);
```

Listing 11.12 Dynamischer Aufruf von Methoden

In Listing 11.12 wird bei einer Anfrage an den Webserver anhand der übergebenen Daten eine Instanz der Klasse `User` generiert. Im weiteren Verlauf wird das Ergebnis eines dynamischen Methodenaufrufs an den Client zurückgesendet. Hilfreich kann derartiger Quellcode sein, wenn Informationen wie beispielsweise E-Mail-Adressen oder der Name des Benutzers ausgelesen werden sollen. Schwieriger wird die Situation, wenn es um vertrauliche Daten wie Passwörter geht. Der Quellcode aus Listing 11.12 erlaubt jedoch nicht nur den Aufruf von Methoden zum Auslesen von Nutzereigenschaften. Durch diesen Quellcode können sämtliche Methoden des `User`-Objekts aufgerufen werden. Weist das `User`-Objekt beispielsweise eine `delete`-Methode auf, mit der sich das Objekt löschen lässt, können Sie diese durch den in Listing 11.13 dargestellten Aufruf auslösen.

```
curl -X GET 'http://localhost:8080/?id=1&method=delete'
```

Listing 11.13 Aufruf der delete-Methode

Sie können dieses Problem auf mehrere Arten lösen. Die einfachste Art ist, wenn Sie komplett auf dynamischen Methodenaufruf verzichten und stattdessen sämtliche Methoden direkt über URLs verfügbar machen. So können Sie sofort kontrollieren, welche Funktionalität Sie veröffentlichen möchten. Wollen Sie nicht auf dynamische Methodenaufrufe verzichten, können Sie dies auch auf einfache Art absichern, indem Sie eine Whitelist führen. Eine Whitelist bedeutet in diesem Fall, dass Sie eine Liste der Methoden führen, die Sie erlauben möchten. Der Aufruf der übrigen Methoden ist nicht erlaubt. In Listing 11.14 sehen Sie, wie eine derartige Umsetzung aussehen könnte.

```
require('http').createServer(function (req, res) {
  var reqData = require('url').parse(req.url, true).query,
      user = new User(reqData.id),
```

```

    whitelist = ['getEmail', 'getName'];

    if (reqData.method && whitelist.indexOf(reqData.method) != -1) {
        res.end(user[reqData.method]());
    }
}).listen(8080);

```

Listing 11.14 Whitelisting für Methodenaufrufe

Die Angriffsvektoren, die Sie bisher kennengelernt haben, zielen vor allem auf den Server ab. Im folgenden Abschnitt sehen Sie, welche Gefahren den Benutzern Ihrer Applikation drohen und wie Sie sie vor Schaden schützen können.

11.3 Schutz des Clients

Nicht nur der Server ist bei einer Webapplikation schützenswert. Auch der Client muss vor Angriffen bewahrt werden. Das Risiko für Ihre Nutzer geht wieder, wie auch schon beim Server, von Ihren Nutzern selbst aus. Das bedeutet allerdings nicht, dass sich die Verwender Ihrer Applikation selbst gefährden würden, sondern vielmehr, dass andere Benutzer unter Umständen die Möglichkeit haben, Ihre Applikation oder die Daten in Ihrer Applikation so zu modifizieren, dass eine Gefahr daraus entsteht.

Im folgenden Abschnitt lernen Sie einige, vor allem in Webapplikationen häufig angewandte Angriffsvektoren kennen und erfahren außerdem, wie Sie mit derartigen Risiken umgehen können.

11.3.1 Cross-Site-Scripting

Attacken gegen Clients einer Webapplikation nutzen stets die Tatsache aus, dass diese Benutzer die Webapplikation als vertrauenswürdig einstufen. Das bedeutet, dass sie nicht erwarten, dass von der Applikation eine Gefahr ausgeht. Aus diesem Grund werden die zur Verfügung gestellten Inhalte auch nicht oder nur sehr wenig auf sicherheitskritische Aspekte geprüft. Und genau diesen Sachverhalt nutzen Angreifer bei einer Cross-Site-Scripting-Attacke aus. Beim Cross-Site-Scripting handelt es sich um eine Art HTML-Injection. Dabei wird Schadcode in die Seite eingeschleust, den die Nutzer der Applikation dann ausführen.

Bei Cross-Site-Scripting-Angriffen können Sie generell zwei Arten unterscheiden. Im ersten Fall werden die Daten, die den Schadcode enthalten, nicht innerhalb der Applikation gespeichert. Die Angriffsdaten sind also flüchtig und gehen verloren. Listing 11.15 zeigt Ihnen hierfür ein konkretes Beispiel.

```
var http = require('http');

http.createServer(function (req, res) {
  var input = require('url').parse(req.url, true).query.input;
  res.end('<html><body>' + input + '</body></html>');
}).listen(8080);
```

Listing 11.15 Anfälligkeit für Cross-Site-Scripting im Quellcode

Rufen Sie diese Seite über die URL `http://localhost:8080/?input=<script>alert('hello')</script>` auf, wird Ihnen ein Alert-Fenster angezeigt. Gehen Sie nun davon aus, dass Ihre Applikation anfällig für derartige Angriffe ist, ist es ein Leichtes, derart präparierte Links zu versenden, um beliebigen JavaScript-Quellcode bei Ihren Benutzern auszuführen. Nutzen Sie hierbei noch einen Link-Shortener, können die Opfer einer derartigen Attacke den Angriff noch nicht einmal über die URL erkennen.

Schlimmer noch als die nicht-persistenten Cross-Site-Scripting-Attacken sind solche, die innerhalb Ihrer Applikation persistiert werden. Das bedeutet, dass die HTML-Injection in irgendeiner Form gespeichert wird und von allen Nutzern Ihrer Applikation wieder abgerufen werden kann. Fügen Sie beispielsweise Schadcode in Ihren Benutzernamen ein und lassen diesen für andere Personen anzeigen, wird automatisch der darin enthaltene JavaScript-Code ausgeführt.

Für Cross-Site-Scripting-Attacken existiert eine relativ einfache Gegenmaßnahme, die sowohl die persistente als auch die nicht-persistente Variante abdeckt und somit verhindern kann. In der Regel sollten Sie Sonderzeichen, die im Zuge von HTML verwendet werden, maskieren. Das betrifft vor allem das Größer-als- und das Kleiner-als-Zeichen. Sie müssen sich jedoch nicht selbst um die Lösung dieses Problems kümmern, sondern können auf existierende Softwarelösungen zugreifen. In Listing 11.16 wird das `entities`-NPM-Paket verwendet, um das Beispiel aus Listing 11.15 gegen Cross-Site-Scripting-Attacken abzusichern.

```
var http = require('http');
var url = require('url');
var entities = require('entities');

http.createServer(function (req, res) {
  var input = url.parse(req.url, true).query.input;
  res.end('<html><body>' +
    entities.encode(input) +
    '</body></html>');
}).listen(8080);
```

Listing 11.16 Sicherheit vor Cross-Site-Scripting-Attacken

Pakete wie `entities` sorgen dafür, dass die potenziell schädlichen `<script>`-Tags so codiert werden, dass sie als normale Zeichenketten im Browser ausgegeben werden und das zwischen ihnen liegende JavaScript nicht ausgeführt wird. Auf diese Art können Sie Ihre Benutzer vor Angriffen schützen. Sie sollten Ausgaben an Benutzer, die aus anderen Systemen wie beispielsweise von Nutzern direkt, Datenbanken oder Webservices stammen, stets absichern.

11.4 Zusammenfassung

Die zwei Grundsätze der Webapplikationssicherheit heißen *Filter Input* und *Escape Output*. Dabei sollten Sie immer davon ausgehen, dass Daten, die in Ihre Applikation gelangen, stets eine potenzielle Gefährdung darstellen, mit der Sie entsprechend umgehen müssen. Außerdem sollten Sie die Daten, die Ihre Applikation in Richtung eines anderen Systems verlassen, immer so codieren, dass für das andere System kein Risiko durch potenziellen Schadcode entsteht. Bieten Sie den Benutzern Ihrer Applikation die Möglichkeit, Logik auszuführen, sollten Sie die verfügbaren Methoden bewusst beschränken, um die Offenlegung interner Schnittstellen zu verhindern. Das nun folgende Kapitel widmet sich dem Herzen einer Webapplikation, dem HTTP-Server.

Kapitel 12

HTTP-Server

*Gebt mir einen Hebel, der lang genug, und einen Angelpunkt,
der stark genug ist, dann kann ich die Welt mit einer Hand bewegen.*
– Archimedes

Der Webserver in Node.js ist der klassische Einstieg in die Plattform. Demonstriert dieses Beispiel doch sehr anschaulich die Möglichkeiten von Node.js, vor allem wenn es um sehr leichtgewichtige Entwicklung geht. In Listing 12.1 sehen Sie den Code einer sehr einfachen Implementierung eines Webserver auf Basis des `http`-Moduls. Dieser dient im weiteren Verlauf des Kapitels als Grundlage für den REST-Service, den Sie im Anschluss implementieren werden.

```
var http = require('http');

http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('I am a webserver in node.js.');
```

Listing 12.1 Webserver in Node.js

Wie Sie im Beispiel in Listing 12.1 sehen können, reichen fünf Zeilen Programmcode aus, um einen funktionsfähigen Webserver umzusetzen. Dieser antwortet auf jede Anfrage auf die Kombination der Adresse `http://localhost` und dem Port 8080 mit der Zeichenkette `I am a webserver in node.js`. Sie können den Programmcode ganz einfach testen, indem Sie die Applikation auf der Kommandozeile starten, danach Ihren Browser öffnen und in die Adresszeile `http://localhost:8080` eingeben. Als Resultat wird die Zeichenkette in Ihrem Browser ausgegeben.

Dieses Beispiel wurde bereits im Detail in Kapitel 3, »Ein erstes Beispiel«, behandelt, sodass ich hier nicht näher auf die einzelnen Methodenaufrufe eingehen werde. Stattdessen möchte ich Ihnen anhand des Beispiels eines REST-Webservice den Einsatz und die verschiedenen Features des `http`-Moduls näherbringen.

Die zu lösende Aufgabe besteht darin, eine einfache Verwaltung für Postadressen zu implementieren. Ein Benutzer hat dabei die Möglichkeit, über verschiedene URLs

Adressen auszulesen, neue Adressen anzulegen, bestehende zu modifizieren oder zu löschen.

Die Technologie, in der die Aufgabe umgesetzt werden soll, nämlich REST, ist lediglich ein Paradigma, mit dessen Hilfe die Features, die das HTTP-Protokoll bietet, möglichst umfassend ausgenutzt werden sollen. REST entstand im Rahmen der Dissertation von Roy Fielding. Nähere Informationen zu diesem Thema finden Sie unter http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Der einfachste Weg, um die Adressen festzuhalten, ist der Einsatz einer Datenbank. Wie bereits in Kapitel 4, »Anpassung und Erweiterung«, kommt auch in diesem Beispiel die relationale Datenbank MySQL zum Einsatz. Listing 12.2 enthält die Querys, die Sie absetzen müssen, um die Datenbank für das Beispiel vorzubereiten.

```
mysql> CREATE DATABASE `addressbook`;
Query OK, 1 row affected (0.00 sec)

mysql> USE addressbook;
Database changed

mysql> CREATE TABLE `addresses` (
  `id` INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  `firstname` VARCHAR(255) NOT NULL,
  `surname` VARCHAR(255) NOT NULL,
  `street` VARCHAR(255),
  `postcode` VARCHAR(255),
  `place` VARCHAR(255),
  `country` VARCHAR(255));
Query OK, 0 rows affected (0.08 sec)

mysql> INSERT INTO `addresses` (`firstname`, `surname`, `street`, `postcode`,
  `place`, `country`)
  VALUES ('Sherlock', 'Holmes', 'Baker Street 221b', 'NW1 6XE', 'London', 'Groß
  britannien');
Query OK, 1 row affected (0.01 sec)
```

Listing 12.2 Datenbankstruktur für das Adressbuch

Haben Sie die Datenbankstruktur, wie in Listing 12.2 beschrieben, angelegt, müssen Sie im nächsten Schritt das `mysql`-Paket mit dem Kommando `npm install mysql` in Ihrem Applikationsverzeichnis installieren. Der letzte Schritt der Vorbereitung besteht für Sie in der Erstellung des Rahmens der Applikation, wie Sie ihn in Listing 12.3 sehen können. Speichern Sie diesen Quellcode beispielsweise in einer Datei `index.js` ab, können Sie die Applikation bereits mit dem Kommando `node index.js` ausführen. Sie können die Applikation auch mit Ihrem Browser nutzen, indem Sie sich mit der Adresse <http://localhost:8080> verbinden. Als Ausgabe werden Sie die

Zeichenkette Hier entsteht das neue Adressbuch erhalten. Damit Sie Ihre Applikation testen können, benötigen Sie eine Applikation, mit der es möglich ist, die verschiedenen HTTP-Kommandos wie GET, POST, PUT und DELETE abzusetzen. Sie können in diesem Fall auf die Open-Source-Software cURL zurückgreifen.

12.1 GET – lesender Zugriff

Der erste Teil der Applikation, die Sie nun implementieren werden, besteht darin, lesend auf die Adressdaten zuzugreifen. REST sieht allgemein vor, dass alle Bestandteile eines Service einzelne Ressourcen sind, auf die über eindeutige URLs zugegriffen werden kann. Dabei liefert eine URL stets das gleiche Ergebnis zurück. Das bedeutet für Ihre Applikation, dass Sie einen Einstieg in das Adressbuch benötigen. So könnten Sie beispielsweise über die URL *http://localhost:8080/addresses/* auf den gesamten Inhalt des Adressbuchs zugreifen. Die Zeichenkette *addresses* stellt in diesem Fall eine Ressource dar. Darüber hinaus sollten Sie Ihren Nutzern die Möglichkeit bieten, auf einzelne Adressen zuzugreifen. Sie erreichen dies über einzelne Ressourcen pro Adresse, die Sie über die ID der Adresse identifizieren. So ist *http://localhost:8080/addresses/1* die Ressource, über die Sie auf die Adresse mit der ID 1 zugreifen können.

Im ersten Schritt sollten Sie den Quellcode Ihrer Applikation etwas umstrukturieren und in eine Klasse kapseln. In Listing 12.3 sehen Sie den Aufbau der Klassen und die Initialisierung des Webservers und der Datenbank.

```
var http = require('http'),
    mysql = require('mysql'),
    util = require('util'),
    querystring = require('querystring');

var AddressBook = function () {
  this.connection = mysql.createConnection({
    host      : 'localhost',
    user      : 'root',
    database  : 'addressbook',
    password  : ''
  });
  this.connection.connect();

  http.createServer(this.dispatcher.bind(this)).listen(8080, 'localhost');
};
```

Listing 12.3 Konstruktor der AddressBook-Klasse

Die Konstruktor-Funktion der `AddressBook`-Klasse sorgt dafür, dass eine Verbindung zur Datenbank aufgebaut wird, die im späteren Verlauf für verschiedene Operationen verwendet werden kann. Außerdem wird ein HTTP-Server generiert. Hier wird die Callback-Funktion allerdings nicht wie bislang ausnahmslos direkt definiert, sondern es wird lediglich eine Referenz auf die `dispatcher`-Methode eingefügt und diese in den korrekten Kontext gesetzt. Der Webserver wird wie bereits in den vorangegangenen Beispielen auf `localhost:8080` gebunden.

```
AddressBook.prototype.dispatcher = function (request, response) {
    response.setHeader('Content-Type', 'application/json; charset=utf-8');

    var url = request.url.split('/');

    if (url[1] !== 'addresses') {
        this.error(response, 404, util.format('%s not found', url[1]));
        return false;
    }

    switch (request.method) {
        case 'GET':
            var id = url.length > 2 ? url[2] : null;
            this.getAddress(response, id);
            break;
        default:
            this.error(response, 405, 'Method not allowed');
    }
};

AddressBook.prototype.error = function (response, code, msg) {
    response.statusCode = code;
    response.end(util.format('{%s}', msg));
};
```

Listing 12.4 Dispatcher

Wie Sie in Listing 12.4 sehen können, übernimmt die `dispatcher`-Methode die wichtigsten Aufgaben innerhalb der Applikationslogik. Im ersten Schritt wird die HTTP-Header-Information `Content-Type` gesetzt. Sie dient dazu, den Client darüber zu informieren, welche Art von Inhalten in der HTTP-Nachricht enthalten ist. In Ihrem Fall sind dies Daten im JSON-Format. Eine wichtige Erweiterung ist die Angabe des Character Sets, das dafür sorgt, dass auch Sonderzeichen wie Umlaute korrekt dargestellt werden. Danach wird geprüft, ob der Benutzer versucht, auf die Ressource `addresses` zuzugreifen, andernfalls tritt ein Fehler auf. Die Fehlerbehandlung erfolgt in der Hilfsmethode `error`. Sie sorgt dafür, dass der korrekte Statuscode, in diesem

Fall 404, über die Eigenschaft `statusCode` des Response-Objekts gesetzt und die Antwort durch einen Aufruf von `response.end` zum Benutzer geschickt wird. Sie haben die Möglichkeit, die Header-Informationen und den Statuscode entweder über die Methode `writeHead` zu schreiben oder dies getrennt wie im Beispiel in Listing 12.4 über `statusCode` und `setHeader` zu bewerkstelligen. Die zweite Variante erlaubt es Ihnen, die Header-Informationen an verschiedenen Stellen innerhalb der Applikation aufzubauen.

Der nächste Schritt besteht aus einer Verzweigung der Applikationslogik anhand der vom Benutzer verwendeten HTTP-Methode. Der lesende Zugriff erfolgt über GET, ein eigener Zweig im `switch`-Statement sorgt dafür, dass die `getAddress`-Methode aufgerufen wird. Wird eine nicht unterstützte Methode verwendet, erhält der Benutzer eine entsprechende Fehlermeldung mit dem Statuscode 405 – Method not allowed.

Nun, da Sie den Rahmen der Applikation implementiert haben, können Sie dazu übergehen, die einzelnen Zugriffsmethoden zu implementieren. Starten Sie mit dem Auslesen von Adressen und der Implementierung der `getAddress`-Methode. Den entsprechenden Quellcode sehen Sie in Listing 12.5.

```
AddressBook.prototype.getAddress = function (response, id) {
  var query = 'SELECT * from `addresses`';
  if (id) {
    query += util.format(' WHERE `id` = "%s"', id);
  }
  this.connection.query(query, function (err, data) {
    if (err) {
      this.error(response, 500, err.code);
      return false;
    }
    var result = [];
    for (var i = 0; i < data.length; i += 1) {
      var temp = [];
      for (var j in data[i]) {
        if (typeof data[i][j] === 'function') { continue; }
        temp.push(util.format('%s:"%s"', j, data[i][j]));
      }
      result.push(util.format('{%s}', temp.join(',')));
    }

    response.statusCode = 200;
    response.end('[' + result.join(',') + ']');
  }.bind(this));
};
```

Listing 12.5 Adressen auslesen

Sie können per HTTP-GET Adressen auslesen. Je nachdem, welche URL Sie benutzen, kann dies entweder eine Adresse sein, oder es handelt sich um sämtliche existierenden Adressen. Um diese Funktionalität umzusetzen, müssen Sie innerhalb der Methode prüfen, ob der Benutzer eine oder alle Adressen auslesen möchte. Sie müssen dann lediglich noch die Abfrage entsprechend anpassen. Innerhalb der Callback-Funktion der Abfrage sollten Sie zuerst mit eventuellen Fehlern umgehen. Diese führen zu einer Antwort an den Client mit einem Statuscode 500. War die Abfrage erfolgreich, sollten Sie eine valide JSON-Zeichenkette erstellen und diese mit dem Statuscode 200 an den Benutzer zurückgeben.

Damit der Service funktionieren kann, müssen Sie die Klasse nur noch instanziiieren. Außerdem sollten Sie dem Benutzer durch eine Ausgabe auf der Kommandozeile signalisieren, dass der Service erfolgreich gestartet wurde. Die zwei dafür erforderlichen Zeilen sehen Sie in Listing 12.6.

```
var myAddressBook = new AddressBook();
console.log('Service is running...');
```

Listing 12.6 Initialisierung des Addressbuchs

Mithilfe von cURL können Sie nun Informationen mithilfe des Service abrufen. Listing 12.7 zeigt einen solchen Aufruf.

```
$ curl -X GET http://localhost:8080/addresses/1
[{"id":"1","firstname":"Sherlock","surname":"Holmes","street":"Baker Street ↻
221b","postcode":"NW1 6XE","place":"London","country":"Großbritannien"}]
$
```

Listing 12.7 GET-Anfrage per cURL

Nachdem Sie das Auslesen der Ressourcen implementiert haben, sollten Sie sich im nächsten Schritt darum kümmern, dass die Benutzer Ihres Service neue Ressourcen anlegen können.

12.2 POST – Anlegen neuer Ressourcen

Webapplikationen haben aufgrund ihrer Dynamik einen Vorteil gegenüber klassischen Desktop-Applikationen. Die Datenbasis ist stets aktuell, und jeder Nutzer hat einen annähernd gleichen Informationsstand. Durch Push-Technologien wie Websockets wird es den Webservern außerdem ermöglicht, Aktualisierungen zum Client zu schicken. Im Falle Ihres Webservice müssen sich allerdings die Clients selbst um die Aktualisierung ihres Datenstands kümmern. Damit sich dieser Datenstand

ändern kann, benötigen die Benutzer eine Möglichkeit, neue Datensätze anzulegen. Im Falle des Beispiels sind dies neue Adressen.

Arbeiten Sie mit REST, können Sie mit GET bestehende Daten auslesen und mithilfe von POST neue Daten einfügen. Anders als bei Anfragen mit GET werden bei POST neben den Header-Informationen auch noch Nutzdaten übergeben, die die Informationen für den neuen Datensatz enthalten.

Für die korrekte Umsetzung einer Antwort auf eine POST-Anfrage sollten drei Bedingungen erfüllt sein: Sie müssen den Statuscode 201 zurückgeben, im Header sollte ein Feld `Location` mit einem Link zur neu angelegten Ressource existieren, und die Nachricht sollte die Daten der Ressource enthalten inklusive der neu vergebenen ID.

Als Ausgangssituation für dieses Beispiel dient die cURL-Anfrage aus Listing 12.8.

```
$ curl -X POST --data "firstname=James&surname=Bond&street=
Albert Embankment 85&postcode=SE1 7TP&place=London&country=
Großbritannien" http://localhost:8080/addresses
```

Listing 12.8 POST-Anfrage per cURL

Durch die Option `-X POST` wird eine POST-Anfrage generiert. Das bedeutet für Ihre Applikation, dass Sie nun nicht nur auf GET als HTTP-Methode reagieren müssen, sondern das `switch`-Statement in der `dispatcher`-Methode um einen weiteren Fall erweitern müssen. Den Code für diese Erweiterung finden Sie in Listing 12.9.

```
switch (request.method) {
  ...
  case 'POST':
    this.newAddress(response, request);
    break;
  default:
    ...
}
```

Listing 12.9 Reagieren auf POST-Anfragen

Mit diesen Änderungen haben Sie bereits erreicht, dass die Applikation nicht mit einer Fehlermeldung auf POST-Anfragen reagiert. Mit diesem Stand der Applikation würde der Prozess dennoch mit einer Fehlermeldung abbrechen, da die Methode `newAddress` noch nicht existiert. Innerhalb dieser Methode müssen Sie nun dafür sorgen, dass die übrigen Anforderungen erfüllt werden. Listing 12.10 zeigt Ihnen die Implementierung dieser Methode.

```
AddressBook.prototype.newAddress = function (response, request) {
  request.on('data', function (data) {
```



```

var values = querystring.parse(data.toString()),
    query = 'INSERT INTO `addresses` SET ?';

this.connection.query(query, values, function (err, result) {
  if (err) {
    this.error(response, 500, err.code);
    return false;
  }

  var id = result.insertId;

  response.statusCode = 201;
  response.setHeader('Location', 'http://localhost:8080/addresses/'
    + id);

  values.id = id;
  var result = this.objToString([values]);
  response.end(result);
}.bind(this));
}.bind(this));
};

```

Listing 12.10 Die Methode `newAddress` zum Anlegen neuer Adressen

Anfragen von Clients, die Daten beinhalten, verursachen beim Server ein Ereignis. Wie Sie im Beispiel sehen können, ist das Request-Objekt vom Typ `EventEmitter`. Das bedeutet, dass Sie mit der `on`-Methode auf derartige Ereignisse reagieren können, indem Sie einen entsprechenden Callback definieren. Um das Beispiel nicht unnötig zu verkomplizieren, sollten Sie davon ausgehen, dass die Daten in einem Stück übermittelt werden. In diesem Fall reicht es aus, dass Sie auf das `data`-Event reagieren. Neben diesem Fall kann es sein, dass der Client die Daten in sogenannten Chunks übermittelt. Das bedeutet für Sie konkret, dass eine Anfrage aus mehreren Teilen zusammengesetzt ist. Hierbei können Sie die einzelnen Datenpakete in der Callback-Funktion zum `data`-Event empfangen und entsprechend verketteten. Der Abschluss einer derartigen mehrteiligen Anfrage markiert das `end`-Event. Nachdem dieses Event gesendet wurde, ist die Anfrage beendet, und es werden keine weiteren Pakete gesendet, die zu dieser Anfrage gehören.

Mit den Daten, die vom Client übermittelt werden, können Sie die SQL-Abfrage erstellen, mit deren Hilfe Sie die Daten in die Datenbank schreiben können. In deren Callback-Funktion sollten Sie zum Abschluss der Antwort zuerst auf etwaige Fehler reagieren. Wurde der Datensatz erfolgreich geschrieben, können Sie den Statuscode 201 verwenden, der dem Client signalisiert, dass der Datensatz erfolgreich erstellt wurde. Des Weiteren nutzen Sie die ID des neu angelegten Datensatzes, um inner-

halb des Location-Headers auf die URL der neu angelegten Ressource zu verweisen. Die eigentliche Nachricht an den Client besteht schließlich aus den Daten des neu angelegten Datensatzes. Zu diesem Zweck sollten Sie den Codeblock aus der Methode getAddress auslagern, der dafür verantwortlich ist, die Zeichenkette für die Antwort zu erstellen. Listing 12.11 zeigt die entsprechende Methode.

```
AddressBook.prototype.objToString = function (data) {
    var result = [];
    for (var i = 0; i < data.length; i += 1) {
        var temp = [];
        for (var j in data[i]) {
            if (typeof data[i][j] === 'function') { continue; }
            temp.push(util.format('%s:%s', j, data[i][j]));
        }
        result.push(util.format('{%s}', temp.join(',')));
    }
    return '[' + result.join(',') + ']';
};
```

Listing 12.11 Die objToString-Methode zur Formatierung der Antwort

Bisher können Sie mit Ihrer Applikation neue Datensätze erstellen und die existierenden auslesen. Als nächsten Schritt sollten Sie sich darum kümmern, bestehende Daten zu aktualisieren. Im nächsten Abschnitt sehen Sie, wie Sie Ihren Webservice um die PUT-Methode erweitern können, mit deren Hilfe Sie Aktualisierungen an bestehenden Datensätzen vornehmen können.

12.3 PUT – Aktualisierung bestehender Daten

Angenommen, eine der Personen in Ihrem Adressbuch zieht um, und Sie müssen die Adresse ändern. Mit den bisherigen Mitteln bleibt Ihnen nichts anders übrig, als über die Datenbankadministration die Daten direkt in der Datenbank anzupassen. Mit diesem Vorgehen umgehen Sie jedoch den Webservice für den Zugriff auf die Daten. Aus diesem Grund sollten Sie im nächsten Schritt eine Schnittstelle implementieren, die das Ändern eines Datensatzes erlaubt. REST sieht für diesen Fall die HTTP-Methode PUT vor. Listing 12.12 zeigt Ihnen ein Beispiel für einen cURL-Aufruf zum Verändern der Daten.

```
$ curl -X PUT --data "firstname=James&surname=Bond&street=8-10 Broadway&
postcode=SW1H OBG&place=London&country=Großbritannien" http://localhost:8080/
addresses/2
```

Listing 12.12 cURL-Aufruf zum Aktualisieren bestehender Daten

Damit Sie auf derartige Anfragen in Ihrer Applikation reagieren können, müssen Sie in einem ersten Schritt den Quellcode dahingehend erweitern, dass die PUT-Methode als eine gültige Methode erkannt wird. Zu diesem Zweck müssen Sie den Quellcode der dispatcher-Methode wie in Listing 12.13 erweitern.

```
switch (request.method) {
  ...
  case 'PUT':
    var id = url.length > 2 ? url[2] : null;
    this.updateAddress(response, request, id);
    break;
  default:
    ...
}
```

Listing 12.13 Reagieren auf PUT-Anfragen im dispatcher

Der nächste Schritt besteht nun darin, die Anfrage in der updateAddress-Methode korrekt zu verarbeiten. In Listing 12.14 sehen Sie, wie sich diese Methode implementieren lässt.

```
AddressBook.prototype.updateAddress = function (response, request, id) {
  request.on('data', function (data) {
    var values = querystring.parse(data.toString()),
        query = 'UPDATE `addresses` SET ? WHERE `id` = ' + id;

    this.connection.query(query, values, function (err, result) {
      if (err) {
        this.error(response, 500, err.code);
        return false;
      }
      response.statusCode = 200;
      values.id = id;
      var result = this.objToString([values]);
      response.end(result);
    }.bind(this));
  }.bind(this));
};
```

Listing 12.14 Datensätze in der Datenbank aktualisieren

Die Aufgabe der updateAddress-Methode besteht darin, anhand einer ID und den Daten, die mit der Anfrage des Clients übermittelt wurden, einen Datensatz in der Datenbank entsprechend zu aktualisieren. Wie schon beim Anlegen einer Adresse ist es auch hier wichtig, dass Sie sich bewusst sind, dass die Anfrage des Clients zum Zeitpunkt des Aufrufs der Methode noch nicht komplett übermittelt ist. Das bedeutet,

dass Sie dafür sorgen müssen, dass die Logik zum Aktualisieren des Datensatzes erst ausgeführt wird, sobald sichergestellt ist, dass die Daten vorliegen. Die vorbereitete Abfrage wird beim Aufruf der `query`-Methode mit Werten bestückt. Die Callback-Funktion, die Sie dieser Methode übergeben, sorgt dann dafür, dass die korrekte Antwort an den Client gesendet wird. An den Client wird nach erfolgter Aktualisierung eine Antwort mit dem Statuscode 200 und der Objektrepräsentation als Zeichenkette zurückgegeben. Diese Zeichenkette wird mithilfe der `objToString`-Methode, die Sie bereits aus Listing 12.11 kennen, erzeugt. Mit dem bisher implementierten Quellcode sind Sie in der Lage, Adressen anzulegen, ihre Daten wieder auszulesen und sie bei Bedarf zu ändern. Für eine komplette Verwaltung der Daten fehlt noch die Möglichkeit, Daten aus dem System zu löschen. Der nächste Abschnitt widmet sich diesem Feature Ihrer Applikation.

12.4 DELETE – Löschen vorhandener Daten

Das HTTP-Protokoll stellt auch für das Löschen von Ressourcen eine eigene Methode zur Verfügung. Mithilfe der DELETE-Methode können Sie über einen einfachen Aufruf einer URL eine Ressource entfernen. Wie bei den übrigen Methoden müssen Sie auch für DELETE das `switch`-Statement in der `dispatcher`-Methode, wie Sie es in Listing 12.15 sehen können, erweitern.

```
switch (request.method) {
  ...
  case 'DELETE':
    var id = url.length > 2 ? url[2] : null;
    this.deleteAddress(response, request, id);
    break;
  default:
    ...
}
```

Listing 12.15 Erweiterung der `dispatcher`-Methode um DELETE

Mit der Eingabe des Kommandos `curl -XDELETE http://localhost:8080/addresses/2` können Sie nach Abschluss der Anpassungen den Datensatz mit der ID 2 löschen. Die `deleteAddress`-Methode erhält wie die übrigen Methoden auch ebenfalls eine Instanz der `ServerResponse`- und der `ServerRequest`-Klasse. Außerdem benötigt diese Methode die ID des zu löschenden Datensatzes. Die Implementierung der `deleteAddress`-Methode sehen Sie in Listing 12.16.

```
AddressBook.prototype.deleteAddress = function (response, request, id) {
  var query = 'DELETE FROM `addresses` WHERE `id` = ' + id;
```

```

    this.connection.query(query, [], function (err, result) {
        if (err) {
            this.error(response, 500, err.code);
            return false;
        }
        response.statusCode = 204;
        response.end();
    }).bind(this));
};

```

Listing 12.16 Implementierung der deleteAddress-Methode

Die Umsetzung der Methode `deleteAddress` folgt einem ähnlichen Schema wie bereits die übrigen Methoden auch. Die SQL-Abfrage wird vorbereitet und mithilfe der `query`-Methode an den Server gesendet. In diesem Fall ist die Umsetzung jedoch etwas einfacher. Sie müssen nicht mit Daten umgehen, die in die Datenbank eingefügt werden. Stattdessen reicht es aus, nur die ID des zu löschenden Datensatzes zu kennen. Auch die Rückmeldung an den Client besteht im Erfolgsfall lediglich aus einem HTTP-Statuscode, 204, der dem Client den Erfolg der Operation anzeigt, jedoch keinen Nachrichtenkörper erforderlich macht. Sollten Sie Ihren Benutzern auch bei einer Löschung Daten zurückübermitteln wollen, müssen Sie den Statuscode 200 verwenden. Dieser steht für einen Erfolg, ermöglicht Ihnen allerdings, weitere Daten über die Löschung zu senden. Das können beispielsweise die Daten des gelöschten Datensatzes sein, die Größe der gelöschten Daten oder die Anzahl der abhängigen Datensätze, die ebenfalls gelöscht wurden. In der Umsetzung sind Sie in diesem Fall nicht begrenzt. Sie sollten in jedem Fall den korrekten Statuscode verwenden, um die Umsetzung von Schnittstellen zu erleichtern. Bisher haben Sie die Grundzüge eines Servers auf HTTP-Methoden erstellt. Im folgenden Abschnitt sehen Sie nun, wie Sie diesen Server weiter ausbauen und mehr und bessere Dienste für Ihre Benutzer anbieten können.

12.5 Accept-Header

Mit dem aktuellen Stand Ihres Servers sind Ihre Benutzer in der Lage, über die HTTP-Methoden POST, PUT, GET und DELETE Adressen innerhalb Ihrer Applikation zu verwalten und sie über diese Schnittstelle auch auszulesen. Zur Kommunikation werden primär die beschriebenen HTTP-Methoden und Daten verwendet, die über die URL beziehungsweise als Nutzlast in der Anfrage mit gesendet werden. Die Rückmeldung des Servers erfolgt im JSON-Format in Kombination mit einem korrekten Statuscode. Entkoppeln Sie die verschiedenen Komponenten Ihrer Applikation wie in diesem Fall über eine REST-Schnittstelle, kann die Anforderung bestehen, neben JSON als Aus-

tauschformat für Informationen weitere Darstellungsformen umzusetzen. Beispiele hierfür sind XML oder HTML. HTTP sieht für diesen Zweck einen eigenen Header, den Accept-Header, vor, mit dessen Hilfe Sie das Ausgabeformat des Servers steuern können. Im ersten Schritt sollten Sie dafür sorgen, dass Ihr Client das gewünschte Format übermittelt. In Listing 12.17 sehen Sie, wie Sie dies auf der Kommandozeile mit cURL erreichen und so Ihre Applikation auf einfache Art testen können.

```
$ curl -X GET -H "Accept: application/json" http://localhost:8080/addresses/1
```

Listing 12.17 cURL-Aufruf mit Header-Information

Die Erweiterung der Anfrage hat noch keinerlei Auswirkungen. Um dies zu ändern, müssen Sie im ersten Schritt die Zeile, in der der Content-Type gesetzt wird, in die `objToString`-Methode beziehungsweise deren untergeordnete Methoden verschieben. In Listing 12.18 sehen Sie, wie diese Methode nach der Anpassung aussieht.

```
AddressBook.prototype.objToString = function (request, response, data) {
    switch (request.headers['accept']) {
        case 'application/xml':
            return this.objToXmlString(response, data);
            break;
        case 'application/json':
        default:
            return this.objToJsonString(response, data);
            break;
    }
};
```

Listing 12.18 Anpassung der `objToString`-Methode

Da sich die Signatur dieser Methode geändert hat, müssen Sie diese Anpassung in der gesamten Applikation nachziehen. Die `objToJsonString`-Methode entspricht der ursprünglichen `objToString`-Methode. Diese Methode weist nun auch die Zeile auf, mit deren Hilfe der Content-Type gesetzt wird, in diesem Fall auf den Wert `application/json`. Die größte Änderung besteht in der `objToXmlString`-Methode. Diese sorgt dafür, dass der Rückgabewert an den Client nun nicht mehr aus einem JSON-String besteht, sondern aus einer XML-Struktur. In Listing 12.19 finden Sie die Implementierung dieser Methode.

```
AddressBook.prototype.objToXmlString = function (response, data) {
    response.setHeader('Content-Type', 'application/xml; charset=utf-8');
    var result = '<addresses>';
    for (var i = 0; i < data.length; i += 1) {
        result += '<address>';
```

```

    for (var j in data[i]) {
        if (typeof data[i][j] === 'function') { continue; }
        result += util.format('<%s>%s</%s>', j, data[i][j], j);
    }
    result += '</address>'
}
return result + '</addresses>';
};

```

Listing 12.19 Die objToXmlString-Methode

Der letzte Schritt besteht nun darin, dass Sie die getAddress-Methode so anpassen, dass sie die neue objToString-Methode verwendet. In Listing 12.20 sehen Sie die erforderlichen Änderungen.

```

AddressBook.prototype.getAddress = function (request, response, id) {
    var query = 'SELECT * from `addresses`';
    if (id) {
        query += util.format(' WHERE `id` = "%s"', id);
    }
    this.connection.query(query, function (err, data) {
        if (err) {
            this.error(response, 500, err.code);
            return false;
        }

        var result = this.objToString(request, response, data);
        response.end(result);
    }.bind(this));
};

```

Listing 12.20 Anpassung der getAddress-Methode

Im Anschluss daran müssen Sie dafür sorgen, dass die getAddress-Methode aus dem dispatcher mit request, response und id als Argumenten aufgerufen wird. Passen Sie nun die Abfrage aus Listing 12.17 so an, wie Sie es in Listing 12.21 sehen, erhalten Sie eine Antwort im XML-Format.

```
$ curl -X GET -H "Accept: application/xml" http://localhost:8080/addresses/1
```

Listing 12.21 Daten im XML-Format auslesen

Mit dieser Erweiterung verfügt Ihre Applikation über zusätzliche Funktionalität, die Sie Ihren Benutzern anbieten können. Sie haben außerdem gesehen, wie einfach es ist, eine bestehende Applikation zu erweitern.

12.6 Zusammenfassung

In diesem Kapitel haben Sie gesehen, wie Sie mit der Hilfe des `http`-Moduls von Node.js einen Webservice erstellen und diesen nach und nach mit weiterer Funktionalität bestücken können. Die Umsetzung eines Webservices in Node.js endet allerdings nicht beim Umgang mit GET- und POST-Anfragen von Benutzern. Sie können auf beliebige Anfragen reagieren. Node.js unterstützt sämtliche HTTP-Methoden. Außerdem können Sie die Header-Informationen einer Anfrage auslesen und entsprechend eine passende Antwort generieren. Bei der Erstellung einer Antwort an einen Benutzer können Sie auf eine breite Palette von Hilfsmitteln zurückgreifen. So können Sie den Statuscode der Antwort beeinflussen oder selbst Header-Informationen setzen. Im nächsten Kapitel lernen Sie einen weiteren Vertreter der Stream-API kennen, die Sockets.

Kapitel 13

Socket-Server

Form follows function.

– Lous Sullivan

In diesem Kapitel erfahren Sie mehr über Sockets unter Node.js. Sockets werden vor allem zur Kommunikation zwischen Applikationen verwendet. Dabei ist es nicht zwingend erforderlich, dass die Applikationen auf dem gleichen System ausgeführt werden. In diesem Kapitel erfahren Sie, wie Sie sowohl Server als auch Clients für die verschiedenen Sockets erstellen können.

Sämtlichen Sockets liegt das Konzept der Datenströme zugrunde. In Node.js wird dieses durch das `stream`-Modul umgesetzt. Bei diesem Modul handelt es sich um eine Abstraktionsschicht, die eine einheitliche Schnittstelle auf die verschiedenen Datenströme darstellt. Sie können einen Datenstrom dazu nutzen, um Daten an die Gegenseite zu senden oder Daten von der Gegenseite zu empfangen. Die gesamte Kommunikation beziehungsweise die Behandlung der Kommunikation innerhalb Ihrer Applikation erfolgt asynchron. Einen sehr populären Vertreter von Datenströmen haben Sie bereits mit dem `http`-Modul kennengelernt. In diesem Kapitel bewegen Sie sich nun im OSI-Schichten-Modell von den Schichten fünf bis sieben, in denen das HTTP-Protokoll angesiedelt ist, eine Schicht tiefer auf die Transportebene mit ihren Protokollen TCP und UDP. Näheres zu diesem Thema finden Sie unter <http://de.wikipedia.org/wiki/Internetprotokollfamilie>. Da HTTP auf einer höheren Kommunikationsschicht liegt, sollten Sie bei der Implementierung von Servern und Clients einige Regeln beachten. So steht Ihnen bei HTTP ein Satz von Methoden zur Kommunikation zur Verfügung. Die einzelnen Ressourcen auf Ihrem Server sprechen Sie über URLs an. Die Kommunikation können Sie über die Informationen steuern, die im HTTP-Header mitgesendet werden. Der Server generiert aufgrund der Informationen, die mit der Anfrage des Clients gesendet werden, eine Antwort. Diese Antwort besteht aus einem Header mit verschiedenen Feldern und einem Statuscode und dem Nachrichtenkörper, mit dem die eigentliche Nutzlast der Antwort gesendet wird. Auf der TCP-Ebene gestaltet sich die Kommunikation erheblich einfacher. Zur Umsetzung von Clients und Servern stellt Node.js das `net`-Modul (<http://nodejs.org/api/net.html>) zur Verfügung. Dieses Modul basiert auf den Schnittstellen des `stream`-Moduls (<http://nodejs.org/api/stream.html>). Der entscheidende Vorteil dieses Moduls liegt darin, dass Sie nicht darauf angewiesen sind, dass Client und Ser-

ver auf unterschiedlichen Rechnern ausgeführt werden müssen. Sie können die Applikationen auch auf dem gleichen Rechner ausführen. Wählen Sie diese Art der Verbindung zwischen zwei Applikationen, können Sie neben TCP als Kommunikationsprotokoll auch auf Sockets zur Kommunikation zurückgreifen. Beim Einsatz von Sockets kommunizieren Sie auf Dateisystembasis zwischen zwei Applikationen. In diesem Kapitel lernen Sie die verschiedenen Arten der Sockets kennen, die Sie mit Node.js verwenden können. Sie beginnen mit der Implementierung zweier Applikationen, die über einen UNIX-Socket lokal kommunizieren. Nach dieser Umsetzung gehen Sie dazu über, die Applikationen mit TCP beziehungsweise UDP zu verbinden.

13.1 UNIX-Sockets

Die einfachste Variante, über die Sie zwei unabhängige Applikationen miteinander sprechen lassen können, besteht aus der Verwendung von dateibasierten Sockets. In diesem Fall benötigen Sie keine Netzwerkschnittstelle, über die Sie Informationen versenden und empfangen können. Ein weiterer Vorteil der dateisystembasierten Kommunikation besteht darin, dass der Overhead, der durch die verschiedenen Netzwerkschichten entsteht, entfällt. In Listing 13.1 sehen Sie ein sehr einfaches Beispiel eines Servers.

```
var net = require('net');

var server = net.createServer(function (conn) {
  conn.on('readable', function (data) {
    console.log(conn.read().toString());
  });

  conn.on('end', function () {
    console.log('connection ended');
  })
});

server.listen('/tmp/nodejs.sock', function () {
  console.log('Server listening on /tmp/nodejs.sock');
});
```

Listing 13.1 UNIX-Socket-Server

Mithilfe der `createServer`-Methode schaffen Sie die Voraussetzungen, mit einem Client kommunizieren zu können, indem Sie eine konkrete Instanz eines Servers erstellen. Dieser Server ist zu diesem Zeitpunkt noch unabhängig von der verwendeten Kommunikationsvariante. Das bedeutet, dass Sie hier sowohl TCP als auch UNIX-

Sockets verwenden können. Mit dem Aufruf der `listen`-Methode auf dem Server-Objekt wird aus dem allgemeinen `net`-Server erst ein UNIX-Socket-Server. Beide Methoden akzeptieren als zweites Argument eine Callback-Funktion, die auf ein bestimmtes Event gebunden wird. Bei der `createServer`-Methode wird die Callback-Funktion an das `connection`-Event gebunden. Dieses Event wird ausgelöst, sobald sich ein Client mit dem Server verbindet. Die Callback-Funktion, die Sie der `listen`-Methode übergeben, wird an das `listening`-Event gebunden, das ausgelöst wird, sobald der Server bereit für eingehende Verbindungen ist.

Im nächsten Schritt implementieren Sie nun einen Client, der in der Lage ist, sich mit dem zuvor generierten Server zu verbinden und mit ihm zu kommunizieren. Listing 13.2 zeigt Ihnen, wie Sie hierbei vorgehen müssen.

```
var net = require('net');

var client = net.connect('/tmp/nodejs.sock', function () {
  console.log('connected to the server');
  client.write('Hello Server');
});
```

Listing 13.2 UNIX-Socket-Client

Ähnlich wie schon bei der Erstellung des Servers müssen Sie auch beim Client eine konkrete Instanz erzeugen. In diesem Fall geschieht dies jedoch mit nur einem Aufruf. Sie verwenden die `connect`-Methode, um eine Instanz des UNIX-Socket-Clients zu erzeugen. Als erstes Argument erhält diese Methode das Ziel des Clients. In diesem Fall ist dies der Name der Socket-Datei als Zeichenkette. Das zweite Argument ist die Callback-Funktion, die ausgeführt wird, sobald die Verbindung hergestellt ist. Die Callback-Funktion wird intern an das `connect`-Event gebunden.

Starten Sie nun den Serverprozess, erhalten Sie auf der Kommandozeile die Ausgabe `Server listening on /tmp/nodejs.sock`. Dies signalisiert Ihnen, dass der Server bereit ist, auf eingehende Verbindungen zu reagieren. Führen Sie nun den Client aus, erhalten Sie auf dem Server die Ausgabe `Hello Server`. Der Client sendet diese Zeichenkette mithilfe der `write`-Methode über den Socket an den Server. Dieser wiederum reagiert in der Callback-Funktion, die auf das `connection`-Event gebunden ist, auf `readable`-Events. Dieses Ereignis wird ausgelöst, sobald auf dem Stream lesbare Daten vorliegen. Durch einen Aufruf der `read`-Methode können Sie auf die Zeichenkette in Form eines Buffer-Objekts zugreifen und dieses auf der Konsole ausgeben.

13.1.1 Zugriff auf den Socket

Ein Vorteil bei der Verwendung von UNIX-Sockets zur Kommunikation zweier lokaler Applikationen besteht darin, dass Sie den Zugriff auf diesen Socket vollständig

unter Ihrer Kontrolle haben. Das bedeutet zum einen, dass kein Zugriff von außerhalb des Systems auf den Socket erfolgen kann, und zum anderen, dass Sie auf Ihrem System ebenfalls die vollständige Kontrolle über den Socket haben und den Zugriff auf diesen regeln können. UNIX-Sockets werden als Datei im Dateisystem dargestellt. Sie sehen dies, wenn Sie wie in Listing 13.3 ein Directory Listing auf das Verzeichnis durchführen, in dem sich der Socket befindet.

```
$ ll /tmp/
srwxr-xr-x 1 nodejs nodejs 0 Apr 1 08:20 /tmp/nodejs.sock
```

Listing 13.3 Directory Listing auf den Socket

Wie Sie Listing 13.3 entnehmen können, kann in diesem Fall jeder Benutzer des Systems auf `/tmp/nodejs.sock` lesend und ausführend zugreifen. Passen Sie nun die Dateisystemberechtigungen dahingehend an, dass nur noch der Benutzer `nodejs` auf den Socket zugreifen kann, bedeutet das, dass ausschließlich diejenigen Applikationen auf diesen Socket zugreifen können, die mit diesem Benutzer ausgeführt werden. Listing 13.4 zeigt Ihnen, wie Sie auf sehr einfache Art und Weise den Zugriff auf diesen Socket aus Ihrer Applikation heraus anpassen können.

```
var net = require('net'),
    fs = require('fs');

...

server.listen('/tmp/nodejs.sock', function () {
  fs.chmod('/tmp/nodejs.sock', 0700, function () {
    console.log('Server listening on /tmp/nodejs.sock');
  });
});
```

Listing 13.4 Beschränkung der Berechtigungen auf den Socket

Als Grundlage für Listing 13.4 dient der Socket-Server aus Listing 13.1. In diesem Beispiel werden die Berechtigungen mithilfe der `chmod`-Methode des `fs`-Moduls angepasst. Die korrekte Stelle, an der Sie die Anpassungen vornehmen sollten, ist die Callback-Funktion des `listening`-Events. An dieser Stelle können Sie sicher sein, dass der Socket existiert und Sie Zugriff darauf haben. Die zweite Voraussetzung neben dem Vorhandensein des Sockets ist, dass Sie das `fs`-Modul einbinden, das Ihnen die `chmod`-Methode zur Verfügung stellt. Ist beides erfüllt, können Sie die Berechtigungen auf den Wert `0700` setzen, was gleichbedeutend mit lesendem, schreibendem und ausführendem Zugriff für den Eigentümer des Sockets ist. Führen Sie nun den Server mit dem Benutzer `nodejs` aus, hat lediglich dieser Zugriff auf den Socket. Um die Wir-

kung der Änderung zu verdeutlichen, sollten Sie nun den Client unter einem anderen Benutzer ausführen. Das Ergebnis, das Sie dann erhalten, sehen Sie in Listing 13.5.

```
$ node client.js
```

```
events.js:72
    throw er; // Unhandled 'error' event
      ^
Error: connect EACCES
    at errnoException (net.js:878:11)
    at Object.afterConnect [as oncomplete] (net.js:869:19)
```

Listing 13.5 Fehlermeldung bei unberechtigtem Zugriff auf den Socket

Beachten Sie bei der Entwicklung, dass die Socket-Datei nicht automatisch gelöscht wird, sobald der Server gestoppt wird. Beenden Sie den Server und starten ihn erneut, erhalten Sie die Fehlermeldung `Error: listen EADDRINUSE`. Mit der `listen`-Methode des Servers erstellen Sie die Socket-Datei. Sobald Sie ein zweites Mal versuchen, die Socket-Datei zu erstellen, wird der Fehler geworfen. Damit der Server gestartet werden kann, müssen Sie die erstellte Socket-Datei löschen. Danach können Sie die Applikation starten.

Als Nächstes erweitern Sie in den folgenden Abschnitten sowohl den Client als auch den Server dahingehend, dass beide miteinander kommunizieren und Sie so die Vorteile eines bestehenden Kommunikationskanals zwischen beiden Endpunkten der Kommunikationsstrecke ausnutzen können.

13.1.2 Bidirektionale Kommunikation

Die Ausdrücke Client und Server sind etwas irreführend, da beide Stellen in gewisser Weise in der Kommunikation gleichberechtigt sind. Der Server sorgt dafür, dass der Socket erstellt wird, und der Client verbindet sich mit dem Socket. Danach können beide sowohl lesen als auch schreiben. Zur Demonstration dieses Sachverhalts implementieren Sie nun eine Applikation, die auf Clientseite Zufallszahlen erzeugt und diese anschließend an einen Server sendet. Der Server wiederum summiert die Zahlen auf und sendet bei jedem zehnten Paket eine Zwischensumme an den Client zurück. In Listing 13.6 sehen Sie die Implementierung der Serverseite.

```
var net = require('net');

var server = net.createServer(function (conn) {
    var sum = 0,
        count = 0;
```

```

conn.on('readable', function (data) {

    sum += parseInt(conn.read().toString());
    count += 1;

    if (count >= 10) {
        count = 0;
        conn.write(sum.toString());
    }
});

server.listen('/tmp/nodejs.sock', function () {
    console.log('Server listening on /tmp/nodejs.sock');
});

```

Listing 13.6 Bidirektionale Kommunikation (serverseitig)

Als Basis für die serverseitige Implementierung dient der Quellcode aus Listing 13.1. Innerhalb der Callback-Funktion, die Sie auf das `readable`-Event binden, lesen Sie die Daten ein, die vom Client gesendet werden, summieren diese auf und halten einen internen Zähler vor, der Ihnen anzeigt, wann das zehnte Paket eingegangen ist und Sie die aktuelle Zwischensumme wieder an den Client zurückschicken müssen. Das Objekt, das Sie als Argument in dieser Callback-Funktion erhalten, ist eine Instanz der Klasse `net.Socket` und repräsentiert die Socket-Verbindung zum Client. Auf diesem Objekt können Sie im Verlauf die `write`-Methode aufrufen, um die Daten an den Client zu versenden. Beachten Sie beim Versenden, dass Sie entweder nur eine Zeichenkette oder ein Buffer-Objekt schreiben dürfen, da Sie beim Versuch, eine Ganzzahl zu versenden, eine Fehlermeldung erhalten und Ihre Applikation abgebrochen wird. Das Gegenstück zur Implementierung des Servers bildet der Socket-Client, den Sie in Listing 13.7 sehen können.

```

var net = require('net');

var client = net.connect('/tmp/nodejs.sock', function () {
    var count = 0,
        number = 0,
        interval = setInterval(function () {
            if (count < 100) {
                number = Math.ceil(Math.random() * 100);
                console.log(number);
                client.write(number.toString());
                count += 1;
            }
        }, 1000);
});

```

```

        } else {
            clearInterval(interval);
            client.end();
        }
    }, 500);
});

client.on('readable', function () {
    console.log('subtotal: ' + client.read().toString());
});

```

Listing 13.7 Bidirektionale Kommunikation (clientseitig)

Der Client verbindet sich mithilfe der `connect`-Methode mit dem zuvor vom Server erstellten Socket. Innerhalb der Callback-Funktion senden Sie alle 500 Millisekunden eine Zahl zwischen 0 und 100 über den Socket an den Server. Nach 100 Zahlen wird das Intervall mit einem Aufruf der `clearInterval`-Methode gelöscht, und es werden keine Zahlen mehr an den Server gesendet. Mit einem Aufruf der `end`-Methode wird die Verbindung getrennt. Die Ausgabe besteht aus zehn aufeinanderfolgenden Zufallszahlen, gefolgt von einer Zeile mit der Zwischensumme der bisher ausgegebenen Zahlen. Diese Folge wird insgesamt zehn Mal wiederholt. Das Clientscript wird beendet, sobald alle 100 Zahlen aufsummiert sind. Der Server jedoch läuft weiterhin. Wie bei einem HTTP-Server können sich auch bei einem Server auf Basis von UNIX-Sockets mehrere Clients gleichzeitig mit dem Server verbinden. Konkret bedeutet das, dass Sie mehrere Clients auf dem gleichen Socket starten können. Die Clients kommunizieren allerdings nur direkt mit dem Server und nicht untereinander.

Verwenden Sie Sockets zur dateisystembasierten Kommunikation in Ihrer Applikation, müssen Sie auch unter Windows-Systemen nicht darauf verzichten. Statt der UNIX-Sockets kommen hier Windows Pipes zum Einsatz. Sie müssen hier lediglich eine abweichende Namenskonvention beachten. Ansonsten können Sie auf die gleichen Schnittstellen zurückgreifen wie bei den UNIX-Sockets. Näheres zur Verwendung von Windows Pipes und ein konkretes Beispiel können Sie in Kapitel 6, »Kommunikation«, nachlesen. Ein entscheidender Nachteil, den die dateisystembasierte Kommunikation mit sich bringt, unabhängig davon, auf welchem System sie eingesetzt wird, ist, dass sich die Applikationen, die miteinander kommunizieren sollen, auf einem System befinden müssen. Eine Alternative, die diese Beschränkung umgeht, sind netzwerkbasierte Verbindungen wie beispielsweise über TCP.

13.2 TCP-Sockets

Dieser Abschnitt beruht auf der bereits getroffenen Annahme, dass Sie dafür sorgen möchten, die Kommunikation zwischen zwei verschiedenen Applikationen zu

ermöglichen. Wie bei allen Socket-Lösungen haben Sie dabei eine maximale Flexibilität, wenn es darum geht, welche Art von Applikationen miteinander kommunizieren. Es muss sich bei den beiden Endpunkten der Verbindung, also dem Client und dem Server, nicht zwingend um Implementierungen in Node.js handeln. So könnte Ihr Server zum Beispiel in Java umgesetzt sein und der Client in Node.js. Aber auch beliebige andere Kombinationen sind denkbar. Nutzen Sie zur Verbindung keine dateisystembasierten Sockets, sondern netzwerkbasierter, erhalten Sie weitere Flexibilität. Die beiden Applikationen müssen nun nicht mehr zwingend auf dem gleichen System ausgeführt werden. Sie können allerdings immer noch zwischen verschiedenen Applikationen innerhalb des gleichen Systems kommunizieren. Voraussetzung für diese Art der Kommunikation ist, dass beide Rechner, auf denen die Software ausgeführt wird, über eine bestehende Netzwerkverbindung verfügen und die jeweilige Gegenseite erreichbar ist.

Übermitteln Sie Ihre Daten mit TCP (http://de.wikipedia.org/wiki/Transmission_Control_Protocol), werden diese fragmentiert. Angenommen, Sie möchten eine Datei von einem System auf ein anderes übertragen und diese Datei hat eine Größe von etwa 2 MB, dann wird diese in mehrere kleine Teile zerteilt, die auf dem Zielsystem automatisch wieder in die korrekte Reihenfolge gebracht werden. Sie bekommen weder vom Zerlegen noch vom Zusammensetzen in Ihrer Applikation etwas mit. Die Größe dieser einzelnen Fragmente beträgt maximal 65.535 Byte, ist in der Regel aber durch tiefer liegende Protokolle auf 1.500 Byte beschränkt. TCP übernimmt jedoch nicht nur den Versand der Pakete, sondern kümmert sich auch um die Sicherstellung der Verbindung. Das bedeutet, dass das Protokoll dafür sorgt, dass keine Pakete bei der Übermittlung verloren gehen. Verschiedene Mechanismen des TCP-Protokolls bewirken, dass keine Daten verfälscht beim Empfänger ankommen. Die Pakete, innerhalb derer die Daten über eine TCP-Verbindung versendet werden, bestehen aus einem Header und dem Nachrichtenkörper. Der Header ist unterteilt in verschiedene Felder, wovon eines aus einer Prüfsumme über die Daten des Pakets und einige Header-Felder besteht. Diese Prüfsumme wird vom Sender gebildet und vom Empfänger dazu verwendet, um zu prüfen, ob das Datenpaket ohne Fehler angekommen ist. Ein weiteres Feld beinhaltet eine Sequenznummer, die die Reihenfolge der Pakete angibt. Es wird verwendet, damit die Einzelteile wieder zu einem Gesamtpaket zusammengesetzt werden können. Bemerkt der Empfänger, dass in der Prüfsumme ein Fehler enthalten ist oder dass in der Reihe der Pakete eines fehlt, wird dieser Fehler behandelt und entsprechend reagiert. Diese Mechanismen sorgen dafür, dass die Kommunikation über TCP als zuverlässig bezeichnet werden kann.

Beim Einsatz von TCP-Sockets sollten Sie beachten, dass stets eine Verbindung zwischen den beteiligten Systemen aufgebaut werden muss. Dieser Verbindungsaufbau erfolgt über den sogenannten Handshake. Dabei sendet der Client ein SYN-Paket. Der Name stammt von dem englischen Begriff »synchronize« und bedeutet, dass sich Client und Server zur Kommunikation synchronisieren. Der Server antwortet mit

einem SYN/ACK-Paket. Diese Bezeichnung steht für »synchronize acknowledgement«, bedeutet also die Bestätigung der Synchronisierung. Der Client schickt abschließend ebenfalls ein ACK-Paket, also seinerseits ebenfalls eine Bestätigung. Nach diesem Drei-Wege-Handshake gilt die Verbindung als aufgebaut und kann von beiden Seiten zur Kommunikation verwendet werden. Mit diesem Verbindungsaufbau kommen Sie als Entwickler nicht direkt in Berührung. Sie sollten allerdings daran denken, wenn es um die Performance Ihrer Applikation geht. Die Kommunikation über das Netzwerk bringt immer einen bestimmten Overhead mit sich. Ein Teil davon besteht aus dem beschriebenen Handshake, der eine gewisse Zeit in Anspruch nimmt. Weitere potenzielle Performanceeinbußen müssen Sie hinnehmen, wenn die Pakete über die einzelnen Netzwerkschichten wie IP und Ethernet versendet werden. Um von der Datenquelle zum Ziel der Kommunikation zu gelangen, muss ein Paket einen bestimmten Weg nehmen. Auf diesem Weg können wiederum Netzknoten wie beispielsweise Router liegen, die das Paket umleiten oder überprüfen, was ebenfalls Zeit kostet. Bei TCP handelt es sich um ein Netzwerkprotokoll. Das bedeutet, dass Sie im Gegensatz zu dateisystembasierter Kommunikation über UNIX-Sockets nicht mehr die uneingeschränkte Kontrolle darüber haben, wer sich mit Ihrer Applikation verbindet. Natürlich existieren auch Sicherheitsmechanismen wie Firewalls, mit denen Sie den Zugriff auf einen TCP-Server beschränken können. So einfach, wie sich die Zugriffssteuerung bei den UNIX-Sockets gestaltet, ist die Aufgabe bei TCP nicht. Um den Kreis der Benutzer eines Dienstes zu beschränken, können Sie beispielsweise auf Firewalls zurückgreifen, die nur Anfragen von bestimmten Quellen zulassen.

13.2.1 Datenübertragung

Als Grundlage für den folgenden Abschnitt sehen Sie hier eine sehr einfache Implementierung eines TCP-Servers. Listing 13.8 zeigt Ihnen den dafür erforderlichen Quellcode.

```
var net = require('net');

net.createServer(
  function (socket) {
    socket.on('readable', function () {
      var data = socket.read();
      if (Buffer.isBuffer(data)) {
        console.log(data.toString());
      }
    });
  }
).listen(8080, '127.0.0.1');
```

Listing 13.8 TCP-Server

Die Funktionalität dieses Servers besteht lediglich darin, dass er eine Verbindung auf dem lokalen Rechner auf Port 8080 öffnet und auf eingehende Verbindungen wartet. Geht eine Verbindung ein, sorgt ein Aufruf von `console.log` dafür, dass die übermittelten Informationen auf der Konsole ausgegeben werden. Die Umsetzung eines TCP-Clients, der mit diesem Server in Verbindung tritt, sehen Sie in Listing 13.9.

```
var net = require('net');

var client = net.connect({port: 8080, host: '127.0.0.1'}, function () {
  client.end('Hello Server!');
});
```

Listing 13.9 TCP-Client

In diesem Beispiel macht der Client nichts anderes, als sich auf den zuvor vom Server geöffneten TCP-Port zu verbinden und eine Nachricht zu senden. Danach wird die Applikation beendet. Der Serverprozess läuft weiter und akzeptiert weiterhin eingehende Verbindungen. Die Konsolenausgabe des Servers besteht aus der Zeichenkette `Hallo Server!`. Im nächsten Schritt erweitern Sie nun dieses einfache Beispiel dahingehend, dass der Client eine Datei an den Server sendet.

13.2.2 Dateiübertragung

Mit ein paar Anpassungen am Quellcode in Listing 13.8 und Listing 13.9 können Sie einen einfachen Dateitransfer umsetzen. Listing 13.10 enthält den angepassten Quellcode des Servers.

```
var net = require('net'),
    fs = require('fs');

net.createServer(
  function (socket) {
    var file = '';

    socket.on('readable', function () {
      file += socket.read();
    });

    socket.on('end', function () {
      var input = new Buffer(file, 'base64');
      fs.writeFile('dest.jpg', input);
    });
  }
);
```

```

    });
  }
).listen(8080, '127.0.0.1');

```

Listing 13.10 TCP-Server für den Dateitransfer

Der erste Unterschied zu Listing 13.8 ist, dass Sie in diesem Fall neben dem `net`-Modul zusätzlich das `fs`-Modul benötigen, um die übermittelte Datei im Dateisystem zu speichern. Der Empfang der Daten in der Callback-Funktion wird in mehrere Schritte aufgeteilt. Das bedeutet, dass im Zuge des Empfangs mehrere Pakete angenommen und entsprechend zusammengesetzt werden. Gehen Sie von einer Bilddatei mit einer Größe von etwa 400 KB aus, werden mit einem Aufruf der `read`-Methode lediglich etwa 50 KB empfangen. Aus diesem Grund müssen Sie die Daten in einer Variablen, in diesem Fall `file`, sammeln und können die Bearbeitung erst nach Beendigung des Vorgangs in der Callback-Funktion des `end`-Events fortsetzen. Hier werden die empfangenen Daten mithilfe eines Buffer-Objekts umcodiert und in einer Datei mit dem Namen `dest.jpg` gespeichert. Diese Codierung findet statt, damit das Bild, das in Binärform vorliegt, problemlos vom Client zum Server übertragen und dort dann entsprechend weiterverwendet werden kann. Listing 13.11 enthält den Quellcode für den Client, der ein Bild an den Server versendet.

```

var net = require('net'),
    fs = require('fs');

fs.readFile('test.jpg', function (err, data) {
  var client = net.connect({port: 8080, host: '127.0.0.1'}, function () {
    client.end(data.toString('base64'));
  });
});

```

Listing 13.11 TCP-Client für den Datentransfer

Auch im Client benötigen Sie das `fs`-Modul, um den Inhalt der zu versendenden Datei vom Dateisystem zu lesen. Innerhalb der Callback-Funktion der `readFile`-Methode stellen Sie die Verbindung zum Server her und senden mit der `end`-Methode des geöffneten Sockets den Inhalt der Datei. Wie bereits beim Server erwähnt, müssen Sie dafür sorgen, dass die Daten in einem Format versendet werden, das Ihnen erlaubt, die Datei verlust- und fehlerfrei zu übertragen, sodass sie nach der Übermittlung weiterhin verwendet werden kann. Zu diesem Zweck können Sie die Base64-Codierung verwenden. Sie kommt häufig im Web zum Einsatz, wenn es darum geht, Dateien zu übermitteln, wie zum Beispiel beim Versand von E-Mails mit Anhängen. Nachdem Sie die Anpassungen am Client und Server vorgenommen haben, können Sie zuerst den Server starten, damit der TCP-Port geöffnet wird, und danach den Cli-

ent. Dieser liest die Datei *test.jpg* vom Dateisystem aus und übermittelt sie Base64-codiert an den Server. Dieser empfängt die einzelnen Bestandteile der Datei, setzt sie zusammen, decodiert das gesamte Paket und speichert das Ergebnis wiederum auf dem Dateisystem. Mit dem `net`-Modul haben Sie neben der reinen Datenübertragung noch weitere Möglichkeiten, steuernd einzugreifen. Die nächsten Abschnitte stellen Ihnen weitere Konzepte der Sockets vor.

13.2.3 Flusssteuerung

Über eine Socket-Verbindung können Sie große Mengen an Daten übertragen. Dies gilt sowohl für TCP-Sockets als auch für die dateisystembasierten Lösungen wie UNIX-Sockets und Windows Pipes. In den Versionen vor 0.10 gab Ihnen Node.js Methoden an die Hand, um den Datenfluss entsprechend zu steuern. Typische Situationen, in denen Sie diese Features benötigen, könnten sein, dass die Daten schneller ankommen, als Sie sie verarbeiten können, oder dass Sie Daten schneller in einen Stream schreiben, als dieser die Daten ausliefern kann.

Die `write`-Methode von Streams, unabhängig davon, ob es sich dabei um netzwerk- oder dateisystembasierte Streams handelt, liefert immer einen Wert zurück. Dieser gibt an, ob die Daten bereits versendet oder im Arbeitsspeicher zwischengespeichert wurden. Für lesbare Datenströme existieren die beiden Methoden `pause` und `resume`. Die `pause`-Methode sorgt dafür, dass keine weiteren `data`-Events ausgelöst werden. Mit der `resume`-Methode wird das Senden des Datenstroms wieder aktiviert, und es treten erneut weitere `data`-Events auf. Der Quellcode in Listing 13.12 zeigt Ihnen, wie sich diese Mechanismen innerhalb einer Applikation miteinander kombinieren lassen.

```
var net = require('net');

var client = net.connect({host: '127.0.0.1', port: 8080}, function () {
  net.createServer(function (socket) {

    client.on('data', function (data) {

      // calculate

      var flushed = socket.write(data);

      if (!flushed) {
        client.pause();
      }
    });
  });
});
```

```

        socket.on('drain', function () {
            client.resume();
        });
    }).listen(8181, '127.0.0.1');
});

```

Listing 13.12 Flusssteuerung in Sockets

In Listing 13.12 werden zwei Streams miteinander kombiniert. Zuerst wird ein Readable Stream als Client auf TCP-Port 8080 erzeugt. In der Callback-Funktion auf das connect-Event wird anschließend ein TCP-Server auf Port 8181 als Writable Stream erzeugt. Sobald Daten auf Port 8080 eingehen, können diese verarbeitet und schließlich an die Clients weitergesendet werden, die sich mit dem Server auf Port 8181 verbunden haben. Nach jeder schreibenden Aktion wird geprüft, ob die Daten erfolgreich in den Stream geschrieben wurden. Ist dies nicht der Fall und die Daten wurden stattdessen im Arbeitsspeicher zwischengespeichert, wird der lesende Stream auf Port 8080 pausiert. Sobald das drain-Event ausgelöst wird, das signalisiert, dass wieder in den Stream geschrieben werden kann, wird der lesbare Datenstrom fortgesetzt. Auf diese Weise verhindern Sie, dass der Schreibpuffer überläuft, und kontrollieren damit auf sehr einfache Art den Datenfluss. Mit Node.js 0.10 wurde die Stream API, die sämtlichen Socket-Servern zugrunde liegt, überarbeitet. Zwar funktionieren die beschriebenen Methoden weiterhin, Sie sollten aber das readable-Event statt des data-Events verwenden und zum Lesen der Daten dann die read-Methode benutzen. Da nun auch die beiden Methoden `pause` und `resume` wegfallen, benötigen Sie eine neue Möglichkeit der Flusssteuerung. Sie rufen einfach die `read`-Methode so oft auf, wie Sie mit den erhaltenen Daten umgehen können. Diese Methode akzeptiert eine Ganzzahl als Wert, die angibt, wie viele Bytes mindestens im Lesebuffer vorhanden sein müssen, damit sie ausgegeben werden. Sind keine Daten vorhanden oder weniger als angegeben, liefert `read` den Wert `null` zurück. Für die hier vorgestellten Streams existiert innerhalb von Node.js eine Reihe von abstrakten Schnittstellen, die Sie im Folgenden noch kennenlernen werden.

13.2.4 Duplex

Sie haben bereits gesehen, dass Sie in einen TCP-Stream sowohl schreiben als auch daraus lesen können. In Node.js wird ein derartiger Stream als *Duplex-Stream* bezeichnet. Ein Duplex-Stream bietet neben der Methode `read` auch die Möglichkeit des Schreibens mit der `write`-Methode. Möchten Sie Ihren eigenen Duplex-Stream implementieren, müssen Sie die Methoden `_read` und `_write` überschreiben, die entsprechend das Lesen und Schreiben übernehmen. Möchten Sie zwei Streams wie in Listing 13.12 miteinander verbinden, können Sie auf das Konzept der Pipes zurückgreifen.

13.2.5 Pipe

Eine Pipe bezeichnet bei den Node.js-Streams die Verbindung eines lesbaren Datenstroms mit einem schreibbaren Datenstrom. Die Methode `pipe` gehört zu den Readable Streams. Sie akzeptiert eine Instanz eines schreibbaren Datenstroms und ein Objekt mit Konfigurationsdirektiven. Als Konfiguration können Sie hier `close: false` mitgeben. Damit erreichen Sie, dass der schreibbare Datenstrom nicht automatisch beendet wird, sobald der lesbare Datenstrom zu Ende ist. Dies ermöglicht es Ihnen, Daten auch noch nach Beendigung des lesbaren Datenstroms in den Datenstrom zu schreiben. Im einfachsten Fall sieht eine derartige Pipe wie in Listing 13.13 aus.

```
var net = require('net');

var writable = net.connect({host: '127.0.0.1', port: 8181}, function () {
  var readable = net.connect({host: '127.0.0.1', port: 8080}, function () {
    readable.pipe(writable);
  });
});
```

Listing 13.13 Einfache Pipe

Implementieren Sie in Ihrer Applikation eine Pipe, wie Sie sie in Listing 13.13 sehen, öffnen Sie zuerst den schreibbaren Datenstrom, das Ziel der Pipe. Ist die Verbindung aufgebaut, können Sie innerhalb der Callback-Funktion den lesbaren Datenstrom öffnen. In der Callback-Funktion des `connect`-Events verbinden Sie nun mithilfe der `pipe`-Methode beide Datenströme miteinander. Sobald Daten auf dem lesbaren Stream verfügbar sind, werden diese direkt in den schreibbaren Datenstrom geschrieben. Sie sollten beachten, dass Sie nicht ohne Weiteres auf die Daten zugreifen können. Haben Sie die Datenströme per `pipe` verbunden, können Sie die Daten nicht mehr per `read` abgreifen. Je nachdem, wo Sie den Aufruf der `read`-Methode platzieren, werden die Daten entweder abgegriffen, bevor sie in den anderen Datenstrom geschrieben werden, beziehungsweise die Daten werden weitergeleitet und aus dem Puffer entfernt, bevor Sie versuchen, die Daten per `read` auszulesen. Möchten Sie erreichen, dass Sie die Daten verarbeiten und gleichzeitig an einen weiteren Stream weiterleiten können, müssen Sie sich selbst darum kümmern. Das bedeutet, dass Sie in diesem Fall nicht auf die `pipe`-Methode zurückgreifen können, sondern die Daten mit `read` aus dem lesbaren Datenstrom auslesen und mit einem Aufruf von `write` in den schreibbaren Datenstrom schreiben müssen.

Neben den netzwerkbasierten TCP-Sockets existiert noch eine weitere Art, Daten über das Netzwerk zwischen zwei Applikationen auszutauschen. Mit UDP können Sie ebenfalls Daten senden und empfangen.

13.3 UDP-Sockets

Das UDP-Protokoll arbeitet auf der gleichen Ebene wie TCP. Im Gegensatz zu diesem weist UDP allerdings grundlegend andere Merkmale auf. Bei UDP wird keine Verbindung aufgebaut. Das bedeutet, dass ein Handshake für einen eventuellen Verbindungsaufbau entfällt, was UDP wesentlich leichtgewichtiger als TCP macht. UDP eignet sich durch die fehlende Verbindung zwischen den beiden Endpunkten der Kommunikation außerdem nicht dafür, größere Datenmengen zu übertragen, da das Protokoll nicht über die Sicherungsmaßnahmen von TCP verfügt und so nicht sichergestellt werden kann, dass keine Informationen verloren gehen. Wie bei TCP ist auch bei UDP die Größe der Pakete begrenzt, sodass beispielsweise eine zu übertragende Datei unter Umständen in mehrere kleine Teile unterteilt werden muss. Bei TCP sorgt die Sequenznummer dafür, dass die Pakete in der korrekten Reihenfolge beim Empfänger zusammengesetzt werden können. Ein solches Feature existiert bei UDP nicht. Das bedeutet, dass Sie beim Versand von Daten über UDP nicht davon ausgehen können, dass Ihre Pakete auch wirklich ankommen. Wenn die Informationen beim Empfänger ankommen, kann ebenfalls nicht gewährleistet werden, ob die Pakete in der korrekten Reihenfolge ankommen. Zusammenfassend wird UDP als unzuverlässiges und verbindungsloses Protokoll bezeichnet. Was im ersten Augenblick nicht sehr vertrauenerweckend klingt, erfüllt in der Informationstechnologie einen bedeutenden Zweck in der Datenübertragung. UDP kommt immer dann zum Einsatz, wenn Daten schnell übertragen werden müssen und es nicht weiter relevant ist, wenn einzelne Pakete nicht beim Empfänger ankommen. Außerdem spielt bei der Verwendung von UDP die Reihenfolge der Pakete keine Rolle. Eine klassische Anwendung dieses Protokolls ist das Network Time Protocol, mit dessen Hilfe Rechner im Internet ihre Uhren synchronisieren. Auch das Domain Name System, kurz DNS, das zur Auflösung von Namen im Netzwerk verwendet wird, benutzt eine UDP-Verbindung zwischen Client und Server, um Anfragen zu versenden. Diese beiden Dienste sind nur ein Bruchteil von Beispielen für den Einsatz von UDP in der Netzwerkkommunikation. Weil UDP ein so bedeutendes Protokoll ist, unterstützt auch Node.js die Umsetzung von Servern und Clients für UDP. Zu diesem Zweck steht Ihnen das `dgram`-Modul zur Verfügung. Die Bezeichnung *Datagramm* bezeichnet eine Einheit bei der Übermittlung von Daten über das Netzwerk. Die Schnittstellen des `dgram`-Moduls von Node.js unterscheiden sich etwas von denen der Module, die Sie bisher zur Kommunikation zwischen Systemen kennengelernt haben. Im Folgenden erfahren Sie, wie Sie eine sehr einfache Client-Server-Implementierung zur Kommunikation über UDP umsetzen können, die Sie dann schrittweise erweitern.

13.3.1 Grundlagen eines UDP-Servers

Das UDP-Protokoll funktioniert bei der Adressierung ähnlich wie TCP. Sie erstellen einen Server und binden diesen an eine bestimmte Adresse und einen Port. Über

diese Kombination können sich dann Clients mit dem Server verbinden und mit ihm kommunizieren. Listing 13.14 zeigt die Implementierung eines UDP-Servers.

```
var dgram = require('dgram');

var socket = dgram.createSocket('udp4');

socket.on('message', function (data) {
  console.log(data.toString());
});

socket.bind(8080, '127.0.0.1', function () {
  console.log('server listening on 127.0.0.1:8080');
});
```

Listing 13.14 UDP-Server

Nachdem Sie das Modul eingebunden haben, können Sie mithilfe von `createSocket` einen UDP-Socket erstellen. Dieser Methode geben Sie den Typ des Sockets mit. Zur Auswahl stehen hier entweder `udp4` oder `udp6`. Dieser Typ gibt an, ob UDP mit IPv4 oder IPv6 verwendet werden soll. Nach dem Aufruf dieser Methode haben Sie einen ungebundenen Socket, mit dem Sie weiterarbeiten können. Wie die übrigen Sockets, sind auch UDP-Sockets eventbasiert. Das bedeutet, dass Sie auch hier über die `on`-Methode des Socket-Objekts Callback-Funktionen an bestimmte Ereignisse binden können. Im Beispiel von Listing 13.14 sehen Sie bereits zwei dieser Ereignisse. Zum einen binden Sie eine Callback-Funktion an das `message`-Event, das ausgelöst wird, sobald Daten über den Socket gesendet werden. Hier können Sie auf ein Buffer-Objekt zugreifen, das die entsprechenden Daten enthält. Außerdem akzeptiert die `bind`-Methode eine Callback-Funktion, die an das `listening`-Event gebunden wird. Dieses Event wird ausgelöst, sobald der Socket erfolgreich an eine Adresse und einen Port gebunden ist und damit eingehende Verbindungen zulässt. Die Funktionalität dieses Servers besteht darin, dass eingehende Daten auf der Konsole ausgegeben werden. Außerdem werden Sie durch eine Meldung benachrichtigt, sobald die Verbindung hergestellt ist und auf eingehende Datagramme wartet. Damit Sie die Funktionsfähigkeit Ihres Servers testen können, sollten Sie im nächsten Schritt einen UDP-Client implementieren. Listing 13.15 zeigt Ihnen, wie Sie hierbei vorgehen müssen.

```
var dgram = require('dgram');

var message = new Buffer("Hello Server");

var client = dgram.createSocket("udp4");
```

```

client.send(message, 0, message.length, 8080, "127.0.0.1", function(err, bytes
) {
    client.close();
});

```

Listing 13.15 UDP-Client

Die clientseitige Implementierung eines UDP-Sockets gestaltet sich um ein Vielfaches einfacher. Sie generieren wie auch schon auf der Serverseite einen ungebundenen Socket. Statt ihn jedoch an eine Adresse und einen Port zu binden, rufen Sie auf dem Objekt die `send`-Methode auf. Dieser Methode übergeben Sie ein Buffer-Objekt, das die zu sendenden Daten enthält. Danach müssen Sie den Offset und die Länge der Daten innerhalb des Buffers angeben, die gesendet werden sollen. Schließlich müssen Sie noch angeben, an welchen Port und welche Adresse Sie die Daten senden möchten. Die Callback-Funktion wird ausgeführt, sobald die Nachricht versandt wurde. Innerhalb dieser haben Sie Zugriff auf ein eventuelles Fehler-Objekt und die Anzahl an Bytes, die gesendet wurden. Am Ende dieser Funktion sollten Sie den Socket auch wieder durch die `close`-Methode schließen, da ein geöffneter Socket dafür sorgt, dass das Programm nicht beendet wird.

13.3.2 Beispiel zum UDP-Server

In der Praxis eignet sich UDP vor allem zur Benachrichtigung von Clients über unkritische Ereignisse oder beim Empfang von Massendaten. Im zweiten Fall müssen Sie jedoch sicherstellen, dass so viele Daten vorliegen, dass das Ergebnis durch den Verlust einzelner Pakete nicht beeinträchtigt wird. Im Folgenden erweitern Sie das Beispiel von Listing 13.14 und Listing 13.15, indem Sie nicht nur eine Zeichenkette zwischen Client und Server austauschen, sondern einen kontinuierlichen Datenstrom versenden und auf diesen subscriben. Listing 13.16 zeigt Ihnen den modifizierten Quellcode des Servers.

```

var dgram = require('dgram'),
    socket = dgram.createSocket('udp4'),
    count = 0;

socket.on('message', function (data) {
    console.log((count++) + ': ' + data.toString());

    if (count >= 1000) {
        socket.close();
    }
});

```

```

socket.on('close', function () {
    console.log('received ' + count + ' datagrams');
});

socket.bind(8080, '127.0.0.1', function () {
    console.log('server listening on 127.0.0.1:8080');
});

```

Listing 13.16 UDP-Server zum Empfang von mehreren Datagrammen

Der Server ist durch die Modifikationen darauf ausgelegt, über die Verbindung insgesamt 1.000 Datagramme zu empfangen. Danach wird die Verbindung getrennt und die Anzahl der empfangenen Datagramme ausgegeben. Zu diesem Zweck binden Sie eine Callback-Funktion auf das `close`-Event. Dieses wird ausgelöst, sobald die `close`-Methode auf dem Socket aufgerufen wird. Da keine Verbindung zwischen Client und Server besteht, haben Sie hier keine Möglichkeit, direkt auf den Client zu reagieren, also muss der Server selbst entscheiden, wann die Verbindung unterbrochen werden soll. Listing 13.17 stellt Ihnen den Quellcode des Clients vor.

```

var dgram = require('dgram'),
    client = dgram.createSocket("udp4"),
    count = 1;

var sendData = function () {
    var message = new Buffer(Math.ceil(Math.random() * 100) + '');

    client.send(message, 0, message.length, 8080, "127.0.0.1", function(err, bytes) {

        count += 1;

        if (count > 1000 || err) {
            client.close();
            clearInterval(interval);
        }
    });
};

var interval = setInterval(sendData, 1);

```

Listing 13.17 UDP-Client zum Versenden von mehreren Datagrammen

Als Beispiel für eine Datenquelle für den UDP-Client werden in Listing 13.17 Zufallszahlen erzeugt. Diese werden in Abständen von einer Millisekunde erzeugt und an

den Server gesendet. Sobald ein Fehler auftritt oder 1.000 Pakete gesendet wurden, wird die Verbindung beendet und das Intervall geschlossen. Starten Sie zuerst den Server und anschließend den Client, sehen Sie, wie die 1.000 Datagramme übertragen werden. Trotz der hohen Frequenz kommen alle Daten fehlerfrei und ohne Probleme an. Auch wenn Sie die Anzahl der Datagramme erhöhen, verändert sich das Verhalten nicht. Beim Einsatz von UDP müssen Sie noch einen weiteren Aspekt beachten: UDP-Datagramme unterliegen wie auch TCP-Pakete einer Größenbeschränkung. Diese Größenbeschränkung kann je nach Verbindung variieren. Übersteigt die Größe des Datagramms die zulässige Größe beim Empfänger, wird das Datagramm verworfen. Der Empfänger erhält hier allerdings keine Fehlermeldung.

UDP eignet sich daher aufgrund seiner Eigenschaften lediglich zur Übermittlung unkritischer Daten. Kommt es bei Ihrer Applikation darauf an, dass sämtliche Daten zuverlässig beim Empfänger ankommen, sollten Sie auf das TCP-Protokoll setzen.

13.4 Zusammenfassung

Im Verlauf dieses Kapitels haben Sie gelernt, wie Sie Daten auf verschiedene Arten übermitteln können. Als Alternative zur Client-Server-Kommunikation über HTTP können Sie auch verschiedene Arten von Sockets einsetzen. Diese operieren auf einer tieferen Ebene als HTTP. Sie sind also an weniger Regeln und Konventionen gebunden, müssen allerdings einige Methoden, die Ihnen das HTTP-Protokoll bietet, selbst implementieren. Neben der netzwerkbasierten Kommunikation können Sie auf UNIX-Sockets oder Windows Pipes zur dateisystembasierten Übermittlung von Daten zurückgreifen. In diesem Fall bleibt Ihnen der Overhead, der durch den Verbindungsaufbau und die verschiedenen Netzwerkschichten entsteht, erspart. Hierbei sind Sie allerdings darauf beschränkt, dass beide Kommunikationsendpunkte auf demselben Rechner ausgeführt werden. Stellen Sie eine Verbindung über das Netzwerk her, können Sie TCP verwenden, falls Sie eine zuverlässige Verbindung benötigen, bei der sichergestellt ist, dass der Empfänger alle Daten korrekt erhält. Alternativ dazu verwenden Sie das UDP-Protokoll, falls es unkritisch ist, wenn einzelne Pakete verloren gehen. In den nächsten Kapiteln bewegen Sie sich im OSI-Schichten-Modell wieder eine Ebene höher und lernen dabei, wie Sie mit Node.js komplette Webapplikationen umsetzen können, ohne dabei einen zusätzlichen Webserver zu benötigen.

Kapitel 14

Multi-Page Webapplikationen

Denn es ist zuletzt doch nur der Geist, der jede Technik lebendig macht.

– Johann Wolfgang von Goethe

Node.js bietet Ihnen alle Hilfsmittel, die Sie benötigen, um eine voll funktionsfähige Webapplikation umzusetzen. Der interessante Aspekt an der Webentwicklung mit Node.js ist, dass Sie zum Betrieb einer Applikation keinen zusätzlichen Webserver benötigen, da Node.js diese Aufgabe selbst übernehmen kann. Im Bereich der Webapplikationen gibt es die verschiedensten Formen, von kleinen Umsetzungen, mit denen einem Benutzer lediglich wenige Informationen präsentiert werden, über mittelgroße Anwendungen mit einer überschaubaren Anzahl an Benutzern, bis hin zu umfangreichen Applikationen, mit deren Hilfe betriebliche Prozesse nachgebildet werden können. Wie die Anwendungsgebiete, so variieren die technischen Implementierungen von Webapplikationen erheblich. Dabei reicht die Spanne von reinen HTML-Seiten über Applikationen mit dynamisch generierten Inhalten bis hin zu echtzeitfähigen Webapplikationen. Dieses Kapitel zeigt Ihnen, wie Sie eine einfache Applikation mit mehreren Seiten umsetzen können. Zu diesem Zweck lernen Sie zuerst mit Express ein Framework für Node.js kennen, das Sie bei der Umsetzung von Webapplikationen unterstützt und Ihnen einige Arbeit abnimmt. Im weiteren Verlauf des Kapitels sehen Sie außerdem, wie Sie mit Jade serverseitig HTML-Templates verwalten können. Das Kapitel wird mit praktischen Erweiterungen Ihrer Applikation abgeschlossen. So sehen Sie, wie Sie die Authentifizierung Ihrer Benutzer lösen oder statische Inhalte wie Bilder oder JavaScript-Dateien ausliefern können.

14.1 Das Web Application-Framework Express

Bisher haben Sie bei der Entwicklung von Webapplikationen direkt das `http`-Modul von Node.js verwendet. Dieses Modul weist den Nachteil auf, dass Sie sehr viele Aufgaben selbst erledigen müssen. Zwar hilft Ihnen Node.js mit einigen Abkürzungen, Ihren Code kurz und übersichtlich zu halten, allerdings löst es damit nicht den Großteil der Standardaufgaben, mit denen Sie alltäglich konfrontiert werden. Sencha Labs hat aus diesem Grund das Connect-Framework entwickelt. Diese Software ist eine Erweiterung des `http`-Moduls und stellt Ihnen eine Möglichkeit zur Verfügung, soge-

nannte Middleware zu registrieren. Diese Middleware wird bei Anfragen ausgeführt. Aufbauend auf diesem Framework, greift Express das Bedürfnis nach Unterstützung bei Standardaufgaben auf und geht einen Schritt weiter als Node.js und Connect, indem es einen spezialisierten Satz an Objekten und Methoden bietet, die zielgerichtet in Webapplikationen eingesetzt werden können. Das Framework unterstützt Sie bei der Umsetzung Ihrer Webapplikation und bietet Ihnen Features wie die Behandlung von Einstellungen, Routing oder das Rendering von Views. Im Verlauf dieses Kapitels lernen Sie einige Komponenten von Express kennen und sehen, wie sie in der Praxis eingesetzt werden können.

14.1.1 Installation

Express ist wie viele andere Frameworks für Node.js als NPM-Paket erhältlich. Im Normalfall installieren Sie Express wie gewohnt lokal in Ihrer Applikation. Das dazu erforderliche Kommando lautet `npm install express`. Alternativ können Sie auch eine *package.json*-Datei für Ihre Applikation anlegen und dort Express als Abhängigkeit definieren. Weitere Informationen zum Format der *package.json*-Datei finden Sie unter <http://package.json.nodejitsu.com/>. Listing 14.1 zeigt Ihnen einen Ausschnitt aus der entsprechenden Datei.

```
{
  ...
  "dependencies": {
    "express": "3.x"
  }
  ...
}
```

Listing 14.1 *package.json* für eine Express-Applikation

Mithilfe der *package.json*-Datei können Sie dann mit dem Kommando `npm install` die Abhängigkeiten automatisch auflösen lassen. Sind diese Voraussetzungen erfüllt, können Sie im nächsten Schritt dazu übergehen, die eigentliche Applikation aufzusetzen.

14.1.2 Setup und Initialisierung der Applikation

Zum Betrieb Ihrer Applikation benötigen Sie wie bisher bei allen Node.js-Applikationen eine Einstiegsdatei, die auf der Kommandozeile gestartet wird und damit den Server instanziiert, eine Verbindung öffnet und auf eingehende Anfragen von Benutzern wartet. Listing 14.2 stellt Ihnen einen typischen Einstieg in eine Webapplikation mit Express vor, die Sie im Verlauf dieses Kapitels schrittweise erweitern werden.

```

var express = require('express');

var app = express();

app.get('/hello', function (req, res) {
  var body = 'Hello Client';

  res.set({'Content-Type': 'text/plain',
    'Content-Length': body.length});

  res.send(body);
});

app.listen(8080);

console.log('Server started on port 8080');

```

Listing 14.2 Grundlegender Aufbau einer Applikation mit Express – index.js

Im Kern besteht Express aus einem Wrapper um das `http`-Modul von Node.js. Im Codebeispiel von Listing 14.2 sehen Sie den Aufbau einer sehr einfachen Webapplikation. Hier erkennen Sie auch gewisse Parallelen zum Aufbau eines HTTP-Servers. Nach der Einbindung des Moduls erstellen Sie den Server, indem Sie die Funktion, die Sie aus diesem Modul erhalten, aufrufen. Das daraus entstandene Server-Objekt verwenden Sie im Folgenden, um Ihre Applikation aufzubauen. Die Callback-Funktion der `get`-Methode wird aufgerufen, wenn Anfragen eingehen. Sie erfahren in den folgenden Abschnitten noch mehr über die Möglichkeiten dieser und noch weiterer Methoden, mit deren Hilfe Sie mit Anfragen entsprechend umgehen. Schließlich rufen Sie die `listen`-Methode des Servers auf und geben an, auf welchen TCP-Port Sie Ihre Applikation binden möchten. Express ist jedoch mehr als ein bloßer Wrapper um das `http`-Modul, wie Sie in den folgenden Abschnitten sehen werden.

14.1.3 Routing

Bei der Verwendung des `http`-Moduls formulieren Sie im Normalfall eine Callback-Funktion, die Sie an das `request`-Event binden. Innerhalb dieser Funktion erstellen Sie anhand des Request-Objekts, das die eingehende Anfrage repräsentiert, eine entsprechende Antwort und schicken diese zum Client. Für die Zuordnung der aufgerufenen URLs zu den passenden Callback-Funktionen bietet Express einen Satz von Methoden, die Sie hierbei unterstützen. In Listing 14.2 haben Sie bereits mit der `get`-Methode die erste dieser Methoden kennengelernt. Neben `get` stehen Ihnen unter anderem `post`, `put`, `delete`, `head` und `options` zur Verfügung. Die jeweilige Methode, also beispiels-

weise get, erhält als erstes Argument den Pfad der URL und als weitere Argumente Callback-Funktionen, die ausgeführt werden, wenn ein Client diese URL anfragt. Im einfachsten Fall gestalten Sie den Aufruf der get-Methode wie in Listing 14.3.

```
app.get('/', function (req, res) {
  var body = '<html><head><title>Hello</title></head><body>Hello
    Client</body></bod></html>';

  res.set({'Content-Type': 'text/html',
    'Content-Length': body.length});

  res.send(body);
});
```

Listing 14.3 Standard-get-Aufruf – index.js

Im Falle von Listing 14.3 können Sie einfach über *http://localhost:8080* auf Ihre Applikation zugreifen und erhalten die in der Callback-Funktion definierte Antwort zurück. Ein Vorteil des Routings von Express ist, dass Sie Flexibilität im Aufbau der Routen haben. Es greift stets die Route, die der Anfrage entspricht. Das bedeutet, wenn Sie eine Route */* und eine Route */user* definieren, greift die erste nur, wenn der Client die URL *http://localhost:8080* anfragt, und die zweite nur, wenn die URL *http://localhost/user* lautet. Mit diesen Varianten stellen Sie Ihren Benutzern statische Routen zur Verfügung. In Webapplikationen besteht allerdings häufig auch die Anforderung, dass Sie dynamisch aufgebaute URLs zur Verfügung stellen. Auch hier hilft Ihnen Express. Sie können Routen nicht nur mit einfachen Zeichenketten definieren, sondern auch reguläre Ausdrücke einsetzen. Angenommen, Sie möchten eine Applikation entwickeln, die es einem Benutzer erlaubt, die Profile anderer Nutzer anzusehen. In diesem Fall greift der Benutzer über eine URL auf ein Profil zu. Zur Unterscheidung muss ein Teil der URL variabel sein, und zwar der, der die ID des anzuzeigenden Profils spezifiziert. Müssten Sie nun für jedes Profil in Ihrer Applikation eine eigene Route definieren, wird Ihre Applikation sehr schnell unübersichtlich. Die Möglichkeit einer Fallback-Route wie beispielsweise */profile* existiert nicht. Greifen Sie auf *http://localhost/profile/13* zu, erhalten Sie die Meldung *Cannot GET /profile/13*. Damit Sie auch diese Anforderung erfüllen können, müssen Sie Ihre Routen etwas anders definieren, und zwar mit einem regulären Ausdruck. Reguläre Ausdrücke sind ein eigener Objekttyp, die nativ von JavaScript unterstützt werden. Listing 14.4 zeigt Ihnen, wie Sie die Route für die Profilanzeige definieren müssen.

```
app.get(/^\/profile\/(\d+)$/, function (req, res) {
  var body = 'Requested profile of User ' + req.params[0];

  res.set({'Content-Type': 'text/html',
```

```

    'Content-Length': body.length});

    res.send(body);
  });

```

Listing 14.4 Routen mit regulären Ausdrücken – index.js

In diesem Beispiel sehen Sie neben der Definition von variablen Routen außerdem, wie Sie auf Komponenten dieser Route zugreifen können. Beachten Sie, dass Sie die Route als ein Objekt vom Typ regulärer Ausdruck angeben müssen und nicht als Zeichenkette. Die angegebene Route ist den URLs zugeordnet, die mit der Zeichenkette `/profile/` beginnen und durch mindestens eine Zahl beendet werden. Das bedeutet, dass diese Route sowohl auf `http://localhost/profile/1` als auch auf `http://localhost/profile/123` zutrifft. Innerhalb der Callback-Funktion haben Sie Zugriff auf das Request-Objekt. Dieses verfügt über eine Eigenschaft `params`, in der die verschiedenen Gruppierungen der Route gespeichert sind. Eine Gruppierung bezieht sich dabei auf die Zeichengruppen innerhalb eines regulären Ausdrucks, die Sie durch eine Klammerung von bestimmten Komponenten erreichen. Die erste Gruppe innerhalb dieser Route ist `(\d+)`. Das bedeutet, dass Sie auf die Zahl in der URL `http://localhost/profile/123` über `req.params[0]` zugreifen und sie so innerhalb Ihrer Callback-Funktion verwenden können. Sie können innerhalb Ihrer Routendefinition beliebig viele Gruppen definieren und auf diese dann über das `params`-Array entsprechend ihrem Auftreten zugreifen. Für den Fall, dass Sie für jede der HTTP-Methoden die gleiche Callback-Funktion registrieren möchten, existiert in Express eine hilfreiche Abkürzung zu diesem Zweck. Sie können ganz einfach die `all`-Methode nutzen. Diese Methode erhält als erstes Argument den Namen der Route, die mit der angeforderten URL übereinstimmen soll. Die zweite ist die Callback-Funktion, die bei einer solchen Anfrage aufgerufen wird. Grundsätzlich verhält sich diese `all`-Methode also genau wie die übrigen Methoden, die Express für die einzelnen HTTP-Methoden zur Verfügung stellt. Listing 14.5 demonstriert Ihnen diese Tatsache.

```

app.all('/news', function (req, res) {
  var body = 'List of recent news articles\n';

  res.set({'Content-Type': 'text/plain',
    'Content-Length': body.length});

  res.send(body);
});

```

Listing 14.5 Alle HTTP-Methoden abfangen – index.js

Am einfachsten können Sie die Funktionalität der `all`-Methode mithilfe von `cURL` testen. In Listing 14.6 sehen Sie das Ergebnis sowohl für eine `GET`- als auch für eine `POST`-Anfrage.

```
$ curl -X GET http://localhost:8080/news
List of recent news articles
$ curl -X POST http://localhost:8080/news
List of recent news articles
```

Listing 14.6 Ergebnisse von Anfragen mit verschiedenen Methoden

Das Problem der Routen, die Sie bisher definiert haben, ist, dass sie noch wenig flexibel waren. Sie haben die Route entweder direkt als Zeichenkette angegeben oder als regulären Ausdruck. Möchten Sie nun aber eine URL und alle untergeordneten URLs, also beispielsweise `http://localhost:8080/articles`, aber auch `http://localhost:8080/articles/415`, mit einer Route abdecken und die weitere Behandlung in nur einer Callback-Funktion durchführen, können Sie auch Wildcards verwenden. Hierfür müssen Sie nur einen Stern an das Ende einer Route setzen. Die Route müssten Sie dementsprechend als `/articles*` definieren. Haben Sie Ihre Applikation sehr modular aufgebaut, kann es vorkommen, dass Sie für eine Route gleich mehrere Callback-Funktionen aufrufen möchten. Statt nun Wrapper-Funktionen zu definieren, die nacheinander alle Funktionen aufrufen, können Sie den Express Routing-Methoden auch mehrere Callback-Funktionen übergeben. Hierbei sollten Sie allerdings beachten, dass Sie dafür sorgen müssen, dass die nachfolgenden Callback-Funktionen auch aufgerufen werden. Express besitzt hier keinen Automatismus, der dies für Sie erledigt. Im nächsten Beispiel implementieren Sie eine Seite, die lediglich angezeigt wird, wenn der Client Ihrer Applikation in der URL eine korrekte Kombination aus Namen und Passwort übermittelt. Ansonsten geben Sie eine Fehlermeldung mit dem entsprechenden Statuscode zurück. In Listing 14.7 sehen Sie den hierfür erforderlichen Quellcode.

```
var login = function (req, res, next) {
  if (req.params[0] === 'admin' && req.params[1] === 'test') {
    next();
  }
  res.send(401, 'You are not allowed to do this!');
}

var start = function (req, res) {

  var hello = 'Welcome ' + req.params[0];

  var body = '<html><head><title>Start Page</title></head><body>' + hello +
```

```

    '</body></bod></html>';

    res.set({'Content-Type': 'text/html',
            'Content-Length': body.length});

    res.send(body);
}

app.get(/^\/start\/(\w+)\/(\w+)$/, login, start);

```

Listing 14.7 Aneinanderreihung von Callback-Funktionen – index.js

In Listing 14.7 schalten Sie beim Aufruf der URL die zwei Callback-Funktionen `login` und `start` hintereinander. Damit dies funktionieren kann, müssen Sie in der ersten Callback-Funktion dafür sorgen, dass die zweite ebenfalls aufgerufen wird. In jeder Callback-Funktion erhalten Sie neben dem Request- und dem Response-Objekt noch ein drittes Argument, die `next`-Funktion. Rufen Sie diese innerhalb der Callback-Funktion auf, wird die nächste Callback-Funktion Ihrer Route aufgerufen. Im Fall des Beispiels wird die `start`-Callback-Funktion nur aufgerufen, wenn der Benutzername, der als erste Gruppe in der Route herausgegriffen wird, der Zeichenkette `admin` entspricht und das Passwort `test` lautet. Andernfalls senden Sie die Antwort mit einer Fehlermeldung und dem Statuscode 401 zurück zum Client. Der `start`-Callback sorgt dafür, dass Ihrem Client die Startseite Ihrer Applikation in HTML ausgeliefert wird. Haben Sie innerhalb Ihrer Applikation viele Routen definiert, können Sie die `routes`-Eigenschaft Ihrer Applikation verwenden, um die Routen auflisten zu lassen. Das resultierende Ergebnis ist nach den definierten HTTP-Methoden geordnet, wie Sie in Listing 14.8 sehen können.

```

{ get:
  [ { path: '/news',
      method: 'get',
      callbacks: [Object],
      keys: [],
      regexp: /^\/news\/?$/i },
    { path: /^\/start\/(\w+)\/(\w+)$/,
      method: 'get',
      callbacks: [Object],
      keys: [],
      regexp: /^\/start\/(\w+)\/(\w+)$/ } ],
  post:
  [ { path: '/news',
      method: 'post',
      callbacks: [Object],

```

```

    keys: [],
    regexp: /^\/news\/?$/i } ],
...

```

Listing 14.8 Liste der definierten Routen

Haben Sie eine Route über die `all`-Methode definiert, wird diese unter allen HTTP-Methoden aufgelistet. Alle übrigen werden unter den jeweiligen HTTP-Methoden aufgeführt. Sie können die `routes`-Eigenschaft auch dazu verwenden, um definierte Routen wieder zu entfernen. Neben dem Kernstück, dem Routing, bietet Ihnen Express noch eine Vielzahl weiterer Funktionalitäten. Wichtig und erwähnenswert ist das Feature der Middleware. Im folgenden Abschnitt erfahren Sie mehr über den Einsatz von Middleware bei Express.

14.1.4 Middleware

Wie zu Beginn dieses Kapitels erwähnt, basiert Express auf Connect. Durch diese Basis erhalten Sie in Express eine Reihe sehr hilfreicher Middleware-Features. Diese Middleware können Sie mithilfe der `use`-Methode einbinden. Ein konkretes Beispiel für eine Middleware ist der Logger. Die Funktionalität dieser Middleware ist vergleichbar mit dem Access-Log eines traditionellen Webservers wie Apache. In Listing 14.9 sehen Sie, wie Sie den Logger einbinden können.

```

var express = require('express');
var app = express();

var logFile = require('fs').createWriteStream('./
application.log', {flags: 'a'});
app.use(express.logger({stream: logFile}));

app.get('/', function (req, res) {
    res.send('hallo client');
});

app.listen(8080);

```

Listing 14.9 Einbindung eines Loggers – index.js

Die verschiedenen Middleware-Komponenten sind direkt im `express`-Paket verfügbar. So können Sie auf den Logger über `express.logger` zugreifen. Diesen binden Sie über die `use`-Methode ein. Der Logger benötigt mit der Konfigurationsoption `stream` einen schreibbaren Datenstrom, der mit einer Ausgabedatei verbunden ist. Diesen erstellen Sie mit dem `fs`-Modul und übergeben ihn als Wert der `stream`-Option. Star-

ten Sie Ihren Server und greifen darauf mit Ihrem Browser zu, wird Ihre Anfrage in die Datei geschrieben. Ein weiteres Beispiel für Middleware, die ursprünglich aus Connect stammt und auf die Sie im Rahmen einer Express-Applikation zurückgreifen können, ist `responseTime`. Dieses Feature fügt automatisch einen zusätzlichen Header `X-Response-Time` in die Antwort an den Client ein. Dieser Header enthält die Zeit in Millisekunden, die die Antwort gedauert hat. Mit diesem Header werden sehr einfache Performancemessungen Ihrer Applikation möglich.

Bei der Verwendung von Express in Ihrer Applikation sind Sie jedoch nicht nur auf vorgefertigte Middleware beschränkt. Sie können auch eigene Funktionen schreiben und diese wie die vordefinierte Middleware in Ihre Applikation einbauen. Die Signatur dieser Middleware-Funktionen ist immer gleich aufgebaut. Das erste Argument ist das Request-Objekt, mit dem die Anfrage eines Clients repräsentiert wird. Das zweite Argument besteht aus dem Response-Objekt, über das Sie die Antwort an den Client beeinflussen können und beispielsweise Informationen oder zusätzliche Header-Felder anfügen können. Das dritte Argument schließlich ist die `next`-Funktion, die Sie aufrufen müssen, damit die nächste Callback-Funktion für die aktuelle Anfrage ausgeführt wird. Mit einer selbst geschriebenen Middleware könnten Sie beispielsweise die Prüfung auf Benutzername und Passwort aus Listing 14.7 direkt in eine Middleware auslagern. Zu diesem Zweck müssen Sie die eigentliche Funktion nur leicht anpassen, da diese auf den Gruppierungen der Routen aufbaut. Außerdem wird diese Funktion nun nicht mehr innerhalb der Route, sondern in einem separaten Aufruf der `use`-Methode eingebunden. Dies ist möglich, da sowohl Middleware-Funktionen als auch Routing-Funktionen die gleiche Signatur aufweisen. In Listing 14.10 sehen Sie das angepasste Beispiel.

```
var express = require('express');

var app = express();

var login = function (req, res, next) {
  var params = req.url.split('/');

  if (params[2] === 'admin' && params[3] === 'test') {
    next();
  }
  res.send(401, 'You are not allowed to do this!');
};

var start = function (req, res) {
  ...
};
```

```
app.use(login);
app.get(/^\/start\/(\w+)\/(\w+)$/ , start);

app.listen(8080);
```

Listing 14.10 Einsatz eigener Middleware – index.js

Die Umsetzung in Listing 14.10 besitzt die Beschränkung, dass hier davon ausgegangen wird, dass stets der zweite und dritte Teil der URL aus dem Benutzernamen und dem Passwort bestehen. Für eine reale Applikation ist dies nicht praktikabel. Aber auch für diese Anforderung existiert eine separate Middleware. Mit `basicAuth` können Sie auf eine Middleware zurückgreifen, mit deren Hilfe Sie die HTTP Basic-Authentifizierung in Ihrer Applikation umsetzen können. Mehr zum Einsatz dieser Middleware erfahren Sie im Laufe dieses Kapitels.

Bei der Verwendung von Middleware und mehreren Routen sollten Sie allgemein beachten, dass die Reihenfolge des Aufrufs der `use`- beziehungsweise der jeweiligen HTTP-Methoden auch die Reihenfolge der Ausführung vorgibt. Das bedeutet, dass Sie sich darauf verlassen können, dass diese Reihenfolge eingehalten wird. Dies ist besonders wichtig, wenn verschiedene Callback-Funktionen aufeinander aufbauen. Die nächste Callback-Funktion in der Reihe wird erst durch einen Aufruf der `next`-Funktion ausgelöst. Das heißt, dass Sie die Kette bewusst unterbrechen können, indem Sie in einer Callback-Funktion nicht `next` aufrufen. Einen konkreten Einsatzzweck können Sie in Listing 14.10 sehen, in dem in der `login`-Funktion nur `next` aufgerufen wird, falls der Benutzer einen gültigen Namen und ein dazu passendes Passwort angegeben hat. Andernfalls wird eine Fehlermeldung mit dem Statuscode 401 zum Client zurückgeschickt. Verwenden Sie die `use`-Methode wie in Listing 14.10, wird die dort angegebene Callback-Funktion bei jeder eingehenden Anfrage aufgerufen.

Je nachdem, welche Middleware Sie verwenden, kann es durchaus vorkommen, dass Sie eine Middleware nur für bestimmte URLs einsetzen möchten. Dies erreichen Sie, indem Sie die `use`-Methode wie die Routing-Funktionen verwenden und der Callback-Funktion den Pfad voranstellen, für den die Middleware zum Einsatz kommen soll. Bisher haben Sie gesehen, dass Ihnen Express einige Aufgaben bei der Entwicklung von Webapplikationen abnimmt. In den folgenden Abschnitten erfahren Sie nun, wie Sie Express mit einer Template-Engine kombinieren und durch den Einsatz verschiedener Middleware eine komplette Webapplikation erstellen können.

14.2 Templates mit Jade

Normalerweise besteht eine Webapplikation aus mindestens einer HTML-Seite mit mehr oder weniger Inhalt. Da die Darstellung der Informationen Ihrer Applikation in

HTML stattfindet, fließt auch einiger Entwicklungsaufwand in die Erstellung dieser Repräsentation, vor allem, wenn es um die Darstellung dynamischer Inhalte geht. Ein wichtiges Qualitätsmerkmal von Webapplikationen ist die strikte Trennung der verschiedenen Bereiche der Applikation. Im Backend findet eine solche Trennung meist über ein Model View Controller-Pattern statt, bei dem die Darstellung von der eigentlichen Geschäftslogik getrennt wird. Diese Trennung lässt sich auch im Frontend Ihrer Applikation fortsetzen, indem Sie hier Ihren HTML-Code zum Aufbau und zur Auszeichnung der Struktur Ihrer Seite verwenden. CSS kommt bei der Darstellung und Positionierung der einzelnen Elemente der Seite zum Einsatz, und in JavaScript bilden Sie schließlich die Logik im Frontend ab. Im Idealfall geht diese Trennung so weit, dass jede Sprache in separaten Dateien und Dateistrukturen abgelegt wird. Bei der Erstellung des Frontends müssen Sie allerdings auch nicht auf Unterstützung von Werkzeugen verzichten. Ein wichtiges Hilfsmittel bieten die sogenannten Template-Engines. Sie stellen Ihnen eine Plattform zur Erzeugung von HTML-Seiten zur Verfügung, mit deren Hilfe Sie auf einfache Art die einzelnen Komponenten organisieren und damit wartbar halten können. Außerdem werden Ihnen Standardaufgaben abgenommen wie beispielsweise das Ersetzen bestimmter Platzhalter durch konkrete Werte oder das Zusammenfügen von einzelnen Teilen der Seite. Eine weitverbreitete Template-Engine im Bereich von Node.js ist Jade. In den folgenden Abschnitten sehen Sie, wie Sie diese Engine in Ihre Node.js-Applikation einbinden und zusammen mit Express nutzen können.

14.2.1 Installation

Wie in Node.js üblich, ist auch die Template-Engine Jade als NPM-Paket verfügbar. In Listing 14.11 sehen Sie das Kommando, das zur Installation erforderlich ist.

```
$ npm install jade
npm http GET https://registry.npmjs.org/jade
...
jade@0.30.0 node_modules/jade
├─ character-parser@1.0.1
├─ mkdirp@0.3.5
├─ commander@1.1.1 (keypress@0.1.0)
├─ monocle@0.1.47 (readdirp@0.2.4)
└─ transformers@2.0.1 (promise@2.0.0, css@1.0.8, uglify-js@2.2.5)
```

Listing 14.11 Installation von Jade

Nach der Installation können Sie Jade in Ihrer Applikation verwenden und die Engine per `require` einbinden. In den folgenden Abschnitten erfahren Sie mehr über die Vorteile und konkreten Einsatzmöglichkeiten von Jade.

14.2.2 Ein einfaches Beispiel

Mit Jade lassen sich HTML-Templates auf eine sehr einfache Weise erstellen. Der Vorteil von Jade ist, dass durch die Struktur und die verwendete Auszeichnung der Aufbau der entstehenden Seite auf den ersten Blick sichtbar wird. Außerdem erspart Ihnen die Engine durch die verkürzte Schreibweise Schreibarbeit bei der Erstellung von HTML-Seiten. In Listing 14.12 sehen Sie den Einsatz von Jade im Zusammenspiel mit dem Node.js- HTTP-Server.

```
var jade = require('jade'),
    http = require('http');

var fn = jade.compile('html\n  head\n    title Hello Client\n  body\n    div Hello Client');

http.createServer(function (req, res) {
  res.end(fn());
}).listen('8080');
```

Listing 14.12 Jade im HTTP-Server

Anhand dieses Beispiels sehen Sie eines der wichtigsten Merkmale der Template-Engine Jade. Sie geben die HTML-Tags nicht mehr in der Form von `<html></html>` an, sondern lediglich mit `html`. Sie können also auf die spitzen Klammern verzichten genauso wie auf das schließende Tag. Von dieser Vereinfachung und der klaren Strukturierung haben Sie in diesem Beispiel noch nichts, da Jade sehr viel auf Zeilenumbrüche setzt und diese direkt in JavaScript-Code nur umständlich umsetzbar sind. Eine weitere Schwierigkeit ist, dass Sie die verwendeten Elemente korrekt einrücken müssen, um die Struktur der Seite aufzubauen. Verzichteten Sie auf Einrückungen, werden die einzelnen Elemente nicht ineinander verschachtelt, sondern stattdessen hintereinandergestellt. Außerdem widerspricht HTML-Code in Form von Jade-Templates im JavaScript-Quellcode der Regel der Trennung von Logik und Darstellung. Das bedeutet, dass Sie im nächsten Schritt die Templates vom eigentlichen Applikationscode trennen. Listing 14.13 zeigt Ihnen diesen nächsten Schritt.

```
var jade = require('jade'),
    fs = require('fs'),
    http = require('http');

fs.readFile('index.jade', function (err, data) {
  var fn = jade.compile(data.toString());

  http.createServer(function (req, res) {
```

```

        res.end(fn());
    }).listen('8080');
});

```

Listing 14.13 Herauslösen der Templates aus der Applikation

Wie Sie im Quellcode von Listing 14.13 sehen können, setzt dieses Beispiel auf das `fs`-Modul zum Auslesen der Template-Datei. Sobald das Jade-Template ausgelesen ist, übergeben Sie die daraus resultierende Zeichenkette an die `compile`-Methode von Jade. Diese kompiliert, wie der Name schon andeutet, die Zeichenkette. Das bedeutet, dass die Zeichenkette dann zur weiteren Verarbeitung vorbereitet ist. Im nächsten Schritt erstellen Sie den HTTP-Server mithilfe von `createServer` und binden ihn auf Port 8080. Innerhalb des `requestListener`-Callbacks nutzen Sie dann das vorkompilierte Template, um daraus die Antwort an den Client zu erstellen. Der Rückgabewert der `compile`-Methode ist wiederum eine Funktion. Rufen Sie diese auf, erhalten Sie die HTML-Repräsentation Ihres Templates, das Sie an den Client als Antwort auf die Anfrage zurücksenden können. Dies geschieht über die `end`-Methode. Damit Listing 14.13 korrekt funktionieren kann, benötigen Sie außerdem eine Datei, die das eigentliche Template enthält. Im Beispiel ist die Datei `index.jade` benannt. Listing 14.14 zeigt Ihnen den Inhalt dieser Datei.

```

!!!
html
  head
    title Hello Client
  body
    #content Client

```

Listing 14.14 Jade-Template

Eine Erweiterung des initialen Beispiels stellen die drei Ausrufezeichen zu Beginn des Templates dar. Geben Sie diese an, sorgt Jade dafür, dass der Doctype angegeben wird. In diesem Fall ist dies der für HTML5 gültige Doctype. Sie können das Ergebnis dieses Beispiel sehen, wenn Sie Ihre Applikation auf der Kommandozeile starten und dann mit Ihrem Browser auf die URL `http://localhost:8080` zugreifen. Den HTML-Code der Antwort sehen Sie in Listing 14.15.

```

<!DOCTYPE html><html><head><title>hello</title></head>
<body><div>Hello Client</div></body></html>

```

Listing 14.15 HTML-Code der Antwort

Nach diesem einleitenden Beispiel erfahren Sie nun, welche weiteren Möglichkeiten Ihnen Jade als Template-Engine für Ihre Webapplikationen bietet.

14.2.3 Verwendung von Jade

In den folgenden Abschnitten lernen Sie die Verwendung von Tags, Klassen und Attributen kennen. Wenn Sie die Beispiele nachvollziehen oder auch eigene Experimente durchführen möchten, können Sie als Basis Listing 14.13 und Listing 14.14 verwenden, wobei Sie das Template in Listing 14.14 einfach erweitern können.

Die Basis einer HTML-Seite bilden natürlich die HTML-Tags und deren Inhalt. Das bedeutet, dass Sie in einem ersten Schritt ein `div`-Element erstellen sollten, in dem Sie Ihren Nutzern Text präsentieren. In Listing 14.16 erfahren Sie, wie Sie ein solches Element umsetzen können.

```
div Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
  | sed diam nonumy eirmod tempor invidunt ut labore et dolore
  | magna aliquyam erat, sed diam voluptua.
```

Listing 14.16 `div`-Element mit mehreren Zeilen Inhalt

In diesem Beispiel sehen Sie, dass Jade das erste Wort am Zeilenanfang als HTML-Tag interpretiert. Hier ist dies ein `div`-Element. Der Inhalt dieses Elements, der als Text angezeigt wird, wird durch ein Leerzeichen von diesem Tag getrennt. Sobald dieser Text allerdings länger wird, stehen Sie vor dem nächsten Problem. Ein einfacher Zeilenumbruch führt dazu, dass das erste Zeichen wiederum als separates Tag interpretiert werden würde. Jade bietet Ihnen zu diesem Zweck die Verwendung des Pipe-Symbols `|`. Dabei müssen Sie darauf achten, dass Sie die korrekte Einrückung einhalten. Jede weitere Zeile innerhalb des `div`-Containers muss zwei Leerzeichen weiter eingerückt sein als das `div` selbst, ansonsten wird das `div` bereits nach der ersten Zeile geschlossen, und die übrigen Zeilen liegen außerhalb des Elements.

Damit Sie die Elemente auf Ihren Seiten korrekt über CSS und JavaScript ansprechen können, benötigen Sie Klassen- und ID-Attribute. Für diese beiden Fälle bietet Ihnen Jade eine Abkürzung, indem Sie eine Raute beziehungsweise einen Punkt verwenden. Wie das konkret aussieht, können Sie in Listing 14.17 sehen.

```
div.someClass This is the content of a div-element with the class someClass
div#someId This is the content of a div-element with the id someId
.myClass This is the content of a div-element with the class myClass
#myId This is the content of a div-element with the id myId
.anotherClass#anotherId This is a div-
element with a combination of id and class
```

Listing 14.17 `id`- und `class`-Attribute in Jade

Beim Aufbau von Webseiten werden sehr häufig `div`-Elemente verwendet, denen Klassen und IDs zugewiesen werden müssen. Dieser Tatsache trägt Jade mit einer Kurzschreibweise Rechnung. Schreiben Sie nur `.myClass` oder `#myId`, wird automatisch ein

div vorangestellt. Auch eine Kombination von Klasse und ID ist möglich. Zu diesem Zweck müssen Sie lediglich beide Direktiven hintereinanderstellen. Dies ist sowohl bei der langen als auch bei der kurzen Schreibweise möglich. Bei der Verwendung von Klassen und IDs sind Sie nicht auf div-Elemente beschränkt. Sie können auch beliebige andere Elemente verwenden. Möchten Sie beispielsweise einen Button mit dem Button-Element erzeugen und diesem die Klasse `btn` und die ID `cancel` zuweisen, erreichen Sie dies über die Zeichenkette `button.btn#cancel`. Bei Klassen und IDs enden jedoch die Möglichkeiten von Jade noch nicht, wenn es um die Verwendung von HTML-Attributen geht. Listing 14.18 zeigt Ihnen die Verwendung von Attributen in Jade.

```
form(action='whatsyourname', method='get')
  input(type='text', value='John')#name
  input(type='submit',
    value='OK')
```

Listing 14.18 Attribute in Jade

Die Attribute eines HTML-Elements schreiben Sie in Klammern direkt im Anschluss an das Element. Die Bezeichnung des Attributs wird dabei nicht in Anführungszeichen gesetzt, ganz im Gegensatz zum Wert, den Sie stets in Anführungszeichen setzen sollten. Außerdem unterstützt Jade auch mehrzeilige Attributangaben.

Bisher haben Sie gesehen, wie Sie mit Jade statische HTML-Seiten generieren können. Einen wichtigen Aspekt einer Template-Engine macht allerdings auch die Fähigkeit aus, dynamische Inhalte in Templates anzuzeigen. In den meisten Template-Engines wird dies über Platzhalter erreicht, die durch konkrete Werte ersetzt werden können. Auch Jade geht diesen Weg. Grundsätzlich haben die Platzhalter in Jade die Form `#{name}`, wobei `name` eine beliebige Zeichenkette zur Bezeichnung des Platzhalters sein kann. Verwenden Sie diese Art von Platzhaltern, werden sämtliche HTML-Sonderzeichen korrekt escaped. Das bedeutet, dass Sie bei der Verwendung dieser Variante einen grundlegenden Schutz vor Injections haben. Wollen Sie die Ersetzung der Platzhalter ohne Escaping verwenden, müssen Sie statt der Raute ein Ausrufezeichen verwenden. Konkret sieht der Platzhalter dann so aus: `!{name}`. Listing 14.19 verdeutlicht den Einsatz von Platzhaltern in einem Template.

```
div hello #{name}
input(type='text', value=name )#name
```

Listing 14.19 Platzhalter in Jade-Templates

Die erste Zeile aus Listing 14.19 zeigt, wie Sie Platzhalter im Text Ihrer Seite verwenden können. Die zweite Zeile demonstriert einen weiteren Sonderfall. Falls Sie Platzhalter in Attributen wie hier im Beispiel im `value`-Attribut eines `input`-Elements verwenden möchten, können Sie nicht auf die übliche Schreibweise zurückgreifen.

Stattdessen greifen Sie auf die Werte, die Sie Ihrem Template zur Ersetzung übergeben, als reguläre JavaScript-Variablen zu. Übergeben Sie also für den Platzhalter `name` einen Wert, greifen Sie auf diesen in Attributen direkt über `name` zu. Im nächsten Schritt erfahren Sie, wie Sie Ihren Platzhaltern konkrete Werte übergeben können.

```
var fn = jade.compile(templateString);
var renderedTemplate = fn({name: 'John'});
```

Listing 14.20 Übergabe von Werten an das Template

Wie gewohnt, übergeben Sie der `compile`-Methode von Jade das Template als Zeichenkette, die Sie im Idealfall aus einer separaten Datei eingelesen haben. Aus diesem Methodenaufruf erhalten Sie ein Funktionsobjekt. Diese Funktion akzeptiert als Argument eine Objektstruktur, die als Eigenschaften die Namen der Platzhalter im Template enthält, denen Sie Werte zuweisen können. Diese Werte werden dann im Template anstelle der Platzhalter eingesetzt. Bei der Verwendung von Platzhaltern sollten Sie allerdings darauf achten, dass sämtliche Platzhalter einen Wert erhalten, da Sie ansonsten einen `ReferenceError` bekommen, der im schlimmsten Fall dafür sorgt, dass Ihre Applikation abgebrochen wird.

Eine sehr hilfreiche Erweiterung der Möglichkeiten von Jade stellen Mixins und Includes dar. Mixins sind im weitesten Sinne mit Funktionen vergleichbar. Sie definieren einen Block und geben ihm einen Namen. Über diesen können Sie im weiteren Verlauf Ihrer Applikation wieder auf den Block zugreifen. Sinnvoll ist dies vor allem bei Elementen, die einen sehr ähnlichen Aufbau haben. Mithilfe von Parametern können Sie die Mixins anpassen. Listing 14.21 stellt Ihnen ein Beispiel eines Mixins vor.

```
mixin inputElement(name, label)
  div
    label(for='#{name}') #{label}
    input(type='text', id='#{name}')

html
  head
    title lala
  body
    div Bitte geben Sie Ihren Namen ein:

    form
      mixin inputElement('firstname', 'Vorname: ')
      mixin inputElement('surname', 'Nachname:')
      input(type='submit', value='OK')
```

Listing 14.21 Mixins in Jade

Das Mixin stellt Ihnen ein `input`-Element und das dazugehörige `label`-Element innerhalb eines `div`-Containers zur Verfügung. Sie können dieses Element durch zwei Argumente konfigurieren. Das erste ist die Bezeichnung des Feldes. Dieser Wert wird als ID für das Feld verwendet. Der zweite Wert, den Sie an das Mixin übergeben, ist die Bezeichnung, die Ihren Benutzern innerhalb des Labels angezeigt wird. Die Verwendung des Mixins geschieht innerhalb des Templates über die Zeile `mixin inputElement('surname', 'Nachname:')`. Sie geben also das Schlüsselwort `mixin`, gefolgt vom Namen Ihres Mixins und den Argumenten, in Klammern an. Die zweite Erweiterung besteht aus den sogenannten Includes. Mit Includes können Sie Ihre Templates so aufteilen, dass Sie Gemeinsamkeiten zwischen verschiedenen Seiten entsprechend auslagern und wiederverwenden können. In Listing 14.22 sehen Sie den Einsatz von Includes am konkreten Beispiel.

```
html
  include include/head

  body
    div this is the content of the page
```

Listing 14.22 Includes in Jade

Im Fall dieses Beispiels sehen Sie, dass statt eines regulären Seitenkopfs die Zeichenkette `include include/head` steht. Dieses Kommando weist Jade an, die Datei *head.jade* im Verzeichnis *include* in das Template einzufügen. Dabei müssen Sie darauf achten, dass das `include`-Statement innerhalb der Datei korrekt eingerückt ist. Im einzufügenden Template beginnt die Einrückung jedoch wieder am Zeilenanfang.

Neben Mixins und Includes existiert in Jade ein weiteres sehr hilfreiches, Konzept. Nutzen Sie Jade zur Erstellung Ihrer Templates, können Sie auf Vererbung innerhalb Ihrer Templates zurückgreifen. Das bedeutet, dass Sie beispielsweise Ihr Basislayout definieren und davon abgeleitet für Ihre verschiedenen Seiten jeweils Spezialisierungen implementieren. In der Praxis bedeutet dies, dass Ihnen in der abgeleiteten Seite das Basis-Template zur Verfügung steht und Sie verschiedene Blöcke ersetzen oder konkretisieren. Listing 14.23 enthält ein Beispiel für ein Basislayout.

```
html
  head
    title Hello World
  body
    block head
      h1 Hello Client
    block content
```

Listing 14.23 Basislayout

Das Layout in Listing 14.23 weist eine Besonderheit auf. Das Schlüsselwort `block` steht in Jade für einen Block im Layout. Dieses Element taucht allerdings nicht im HTML-Quellcode der Webseite auf, sondern ist stattdessen als Strukturelement in Jade vorgesehen. Ein `block`-Element reicht bis zur Definition des nächsten `block`-Elements. Mithilfe dieses Features können Sie bestimmte Teile Ihrer Webseite markieren und durch Vererbung später ersetzen. Im Falle des Beispiels aus Listing 14.23 sind dies die `block`-Elemente mit der Bezeichnung `head` und `content`. In Listing 14.24 sehen Sie, wie Sie mit dem Basislayout, das in der Datei *base.jade* gespeichert ist, eine weitere Seite erstellen.

```
extends base
```

```
block content
```

```
  h2 Lorem ipsum dolor sit amet
```

```
  div consetetur sadipscing elitr, sed diam nonumy eirmod tempor
```

```
    | invidunt ut labore et dolore magna aliquyam erat, sed diam
```

```
    | voluptua. At vero eos et accusam et justo duo dolores et ea
```

```
    | rebum. Stet clita kasd gubergren, no sea takimata sanctus est
```

Listing 14.24 Erweiterung des Basislayouts

Mit dem Kommando `extends base` weisen Sie Jade an, die Datei *base.jade* als Basis zu verwenden. Im Anschluss geben Sie an, welche Blöcke ersetzt werden sollen. In diesem Beispiel ersetzen Sie den Block `content` mit einem `div`-Element mit mehreren Zeilen Text. Der erste Block des ursprünglichen Layouts bleibt unverändert. Starten Sie nun Ihre Applikation und greifen mit einem Webbrowser darauf zu, sehen Sie, dass sowohl der erste als auch der zweite Block korrekt angezeigt werden.

Jade verfügt noch über zahlreiche weitere Features. Ein guter Einstieg in Jade ist die Online-Dokumentation, die unter <http://jade-lang.com/> verfügbar ist. In den folgenden Abschnitten erfahren Sie, wie Sie Jade und `express.js` miteinander verbinden können.

14.2.4 Integration in `express.js`

Voraussetzung für das Funktionieren des Zusammenspiels zwischen Jade und `express.js` ist, dass beide Module installiert sind. Danach können Sie dazu übergehen, Ihre Applikation zu entwickeln. In Listing 14.25 finden Sie ein Beispiel für die Umsetzung einer sehr einfachen Applikation auf Basis von `express.js`, in der Jade als Template-Engine zum Einsatz kommt.

```
var express = require('express');
```

```
var app = express();
```

```

app.set('views', __dirname);
app.set('view engine', 'jade');

app.get('/', function(req, res){
  res.render('index', { name: 'Charly' });
});

app.listen(8080);

```

Listing 14.25 express.js-Applikation mit Jade als Template-Engine

Die Applikation in Listing 14.25 beginnt mit dem Einbinden der erforderlichen Pakete. In diesem Fall benötigen Sie lediglich `express`. Das Einbinden von Jade findet im Hintergrund statt. Auch die Initialisierung Ihrer Applikation folgt dem Standard. Durch den Aufruf von `express` erhalten Sie Ihr Applikations-Objekt. Auf diesem Objekt müssen Sie nun verschiedene Werte über die `set`-Methode setzen. Das ist zum einen die Einstellung `views` und zum anderen `view engine`. Mithilfe von `views` geben Sie an, wo Ihre Templates abgelegt sind. Im Beispiel liegen die Dateien mit den Templates im Verzeichnis, in dem sich auch die Applikation befindet. Im Normalfall werden Sie eine eigene Verzeichnishierarchie zur Ablage und Organisation Ihrer Templates erstellen. Wichtig ist hier, dass Sie zur Angabe von `views` eine absolute Pfadangabe verwenden müssen. Über die globale Variable `__dirname` erhalten Sie den dazu erforderlichen absoluten Pfad. Mit der zweiten Einstellung `view engine` geben Sie an, dass Jade für das Rendering der Templates verwendet werden soll. Durch diese Definition vermeiden Sie, dass Sie erst umständlich die Template-Dateien auslesen und dann an Jade übergeben müssen. Die Kombination aus `express.js` und Jade nimmt Ihnen diese Aufgabe ab. Um ein Template zu rendern, reicht ein Aufruf der `render`-Methode des Response-Objekts in Ihrer Applikation. Diese Methode akzeptiert als erstes Argument eine Zeichenkette, die den Speicherort des entsprechenden Templates angibt. Die Erweiterung `.jade` wird automatisch angehängt, sodass aus `index` im JavaScript-Quellcode der Dateiname `index.jade` wird. Das zweite Argument der `render`-Methode ist eine Objektstruktur mit den Variablen, die die Platzhalter im Template ersetzen sollen. In Listing 14.26 sehen Sie das Template, das mit der Applikation aus Listing 14.25 zum Einsatz kommt.

```

!!!
html
  head
    title hello
  body
    #greeting hello #{name}

```

Listing 14.26 Jade-Template für die express.js-Applikation

Sie haben gesehen, wie es mit nur wenigen Zeilen möglich ist, aus der Kombination von `express.js` und `Jade` sehr schnell eine funktionsfähige Webapplikation zu erstellen. Sie haben hier Zugriff auf den vollen Funktionsumfang auf beide Bibliotheken. Im folgenden Abschnitt sehen Sie, wie Sie verschiedene Middleware von `express.js` dazu verwenden können, um Ihre Applikation besser und flexibler zu machen.

14.3 Auslieferung von statischen Inhalten

Eine sehr häufige Aufgabe bei der Erstellung von Webapplikationen ist die Auslieferung statischer Inhalte. Unter die Bezeichnung statischer Inhalte fallen sämtliche Dateien, die direkt vom Server an den Client ausgeliefert werden und nicht erst durch eine Engine verarbeitet werden müssen. Klassische Beispiele sind Mediendaten wie Bilder, Videos und Audio-Dateien, aber auch Quellcode, der erst vom Client verarbeitet wird, wie zum Beispiel JavaScript. Im folgenden Beispiel sehen Sie, wie Sie verschiedene statische Inhalte in Ihre Webseite einbinden können. Dabei werden Sie ganz konkret eine Seite mit `express.js` und `Jade` umsetzen und in diese die JavaScript-Bibliothek `jQuery`, eine eigene CSS-Datei und ein Bild einbinden. Als Grundlage für diese Applikation sollten Sie zunächst drei Verzeichnisse mit den Namen *views*, *style* und *images* anlegen. In dieser Struktur können Sie dann Ihre Templates, die Stylesheets und Bilder ablegen. Auf gleicher Ebene wie diese Verzeichnisse legen Sie nun den Einstiegspunkt in Ihre Applikation, die Datei *index.js*, an. Den Aufbau dieser Datei sehen Sie in Listing 14.27.

```
var express = require('express');

var app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'jade');

app.get('/', function (req, res) {
  res.render('index');
});

app.listen(8080);
```

Listing 14.27 Einstieg in die Applikation

Soweit bietet dieses Beispiel nichts Neues im Vergleich zum Sourcecode, den Sie bisher gesehen haben. Schritt für Schritt werden Sie nun erfahren, wie Sie die verschiedenen statischen Inhalte in Ihre Seite einbetten können. Als Ausgang für die folgenden Erweiterungen dient ein Basislayout, das Sie in der Datei *layout.jade* im

Verzeichnis *views* speichern. In Listing 14.28 finden Sie den Quellcode dieses Templates.

```
html
  head
    title Hello Client
  body
    #headline
    #content Hello Client
```

Listing 14.28 Das Basislayout

Die erste Aufgabe besteht nun darin, innerhalb des *div*-Elements mit der ID *headline* das Node.js-Logo anzuzeigen. Dieses Logo liegt als Datei *nodejs.png* im Verzeichnis *images* vor. Unterhalb des *div*-Elements fügen Sie die Zeile `img(src="/images/nodejs.png")` ein. Dies ist allerdings nur der erste Teil der Lösung. Noch können Sie nicht auf die Datei zugreifen. Versuchen Sie das Bild direkt anzuzeigen, erhalten Sie die Fehlermeldung `Cannot GET /images/nodejs.png`. Damit Sie die Datei korrekt ausliefern, benötigen Sie die *static*-Middleware von *express.js*. Sie dient dazu, statische Dateien auszuliefern. Damit Sie das Bild korrekt anzeigen lassen können, müssen Sie den Einstieg in Ihre Applikation etwas anpassen. In Listing 14.29 finden Sie die dafür erforderliche Zeile.

```
app.use('/images', express.static(__dirname + '/images'));
```

Listing 14.29 static-Middleware

Starten Sie nach diesen Änderungen Ihre Applikation neu, können Sie das Bild in Ihrer Seite sehen. Da es sich bei diesem Bild um ein PNG mit transparentem Hintergrund und weißer Schrift handelt, müssen Sie im nächsten Schritt das Styling des *div*-Elements anpassen, das das Bild enthält. Zur Lösung dieser Aufgabe existieren zwei Ansätze. Grundlage für beide Wege bildet die Datei *style.css* im Verzeichnis *style*. Den Inhalt des Stylesheets sehen Sie in Listing 14.30.

```
div#headline {
  background-color: black;
}
```

Listing 14.30 Stylesheet

Die erste Möglichkeit zur Einbindung des Stylesheets in Ihre Applikation besteht darin, den Inhalt direkt in die Seite einzubetten. Jade bietet Ihnen hierfür das *include*-Kommando, das Sie in diesem Kapitel bereits kennengelernt haben, als es darum ging, Jade-Dateien in ein Template einzubinden. Fügen Sie die Zeile *include*

`../style/style.css` unterhalb der `head`-Direktive ein, erkennt Jade anhand der Dateiendung, dass es sich hierbei um ein Stylesheet handelt, und bindet den Inhalt in die Webseite ein. Dabei werden die Angaben aus diesem Stylesheet in `style`-Tags gekapselt, was dafür sorgt, dass sie vom Browser korrekt interpretiert werden. Die zweite Variante besteht darin, dass Sie das `include`- durch ein `link`-Element ersetzen. Listing 14.31 zeigt Ihnen den Aufbau dieser Zeile.

```
//include ../style/style.css
link(href='../style/style.css', rel='stylesheet', type='text/css')
```

Listing 14.31 link-Element für Stylesheets

Das `link`-Element erhält als Attribute die Angabe `href` und damit den Speicherort des Stylesheets. Des Weiteren geben Sie mit `rel` und `type` zusätzliche Informationen für den Browser an, damit dieser das Stylesheet korrekt interpretieren kann. Bei dieser Variante stehen Sie nun wieder vor dem Problem der Auslieferung. Greifen Sie auf das Stylesheet über die URL `http://localhost:8080/style/style.css` direkt zu, erhalten Sie vom Server die Rückmeldung `Cannot GET /style/style.css`. Auch hier müssen Sie also dafür sorgen, dass die Datei korrekt ausgeliefert wird. Damit dies problemlos funktionieren kann, müssen Sie hier wie auch schon bei der Darstellung des Bildes auf die `static`-Middleware von `express.js` zurückgreifen. Listing 14.32 zeigt Ihnen den Block mit der Konfiguration.

```
app.use('/style', express.static(__dirname + '/style'));
```

Listing 14.32 static-Middleware für die Auslieferung von Stylesheets

Die Zeile gleicht der, die Sie auch für die Bilder verwendet haben, bis auf den Punkt, dass die Pfade hier angepasst werden müssen. Der Nachteil dieser Variante ist, dass für das Stylesheet ein separater Request erforderlich ist. Das bedeutet für den Server zusätzlichen Overhead durch das Ausliefern der Datei, und für den Client heißt das zusätzliche Wartezeit, bis das Stylesheet heruntergeladen ist. Bei der Auswahl der Mittel sollten Sie auf jeden Fall die Variante wählen, die für Ihre Zwecke am geeignetsten ist. Die endgültige Applikation, wie sie im Browser Ihrer Benutzer aussehen sollte, sehen Sie in Abbildung 14.1.

Sie sehen in Listing 14.31 außerdem, wie einfach es möglich ist, beide Varianten parallel zu testen, ohne dass Sie die jeweils andere Zeile löschen müssen. Stellen Sie einer Zeile zwei Schrägstriche voran, wird diese Zeile, wie Sie es aus JavaScript kennen, auskommentiert. Jade sorgt in diesem Fall dafür, dass aus diesem Kommentar ein korrekt formatierter HTML-Kommentar wird, also mit `<!--` zu Beginn und abgeschlossen durch `-->`. Sie können mit diesem Mittel auch ganze Blöcke in Ihrer Seite auskommentieren. Dies erreichen Sie ganz einfach, indem Sie ein Element in der Hierarchie Ihrer Seite auskommentieren. Sämtliche untergeordneten Elemente werden

dann ebenfalls in den Kommentar mit eingeschlossen und nicht vom Browser interpretiert.

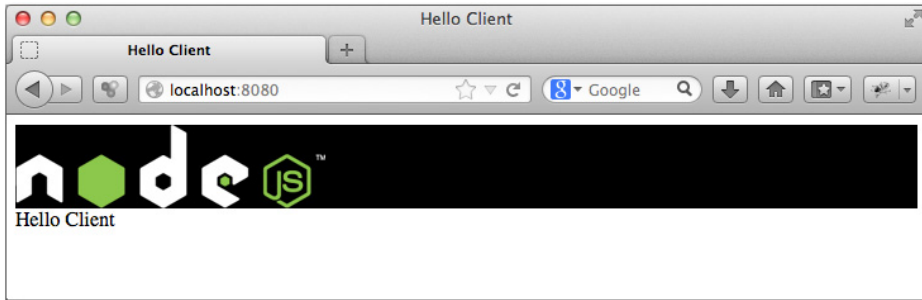


Abbildung 14.1 Webapplikation auf Basis von express.js und Jade

14.4 Zusammenfassung

In diesem Kapitel wurden Ihnen mit express.js und Jade zwei Hilfsmittel vorgestellt, mit denen Sie sehr schnell vorzeigbare Ergebnisse erzielen können. express.js nimmt Ihnen sehr viele Aufgaben ab, die häufig bei der Entwicklung von Webapplikationen auftreten. Dies wird noch deutlicher, wenn Sie sich vor Augen führen, auf welcher Ebene express.js agiert. Node.js liefert mit seinem http-Modul die Basis für Ihre Webapplikation. Auf diesem Modul baut Connect auf und bietet Ihnen zahlreiche Middleware-Komponenten für Ihre Applikation. Eine Komponente davon haben Sie mit der static-Middleware in den vorangegangenen Abschnitten eingesetzt. Auf den Möglichkeiten von Connect setzt schließlich express.js auf und bietet Ihnen neben dem Umgang mit Anfragen und Antworten an Clients und der Einbindung von Middleware außerdem beispielsweise Routing für Ihre Applikation. Haben Sie express.js in Ihrer Applikation im Einsatz, ist es ein Leichtes, die Template-Engine Jade einzubinden. Diese Template-Engine nimmt Ihnen viel Schreibarbeit ab, da Sie Ihr HTML sehr verkürzt schreiben können. Außerdem erzwingt es durch die verwendete Einrückung eine klare Struktur in Ihrer Seite. Sie können Ihre Templates mit den Features von Jade sehr gut organisieren und strukturieren und dabei unter anderem auf Vererbung, Includes und Mixins zurückgreifen. Im nächsten Kapitel gehen Sie mit dem bereits erworbenen Wissen über Node.js und Webapplikationen noch einen Schritt weiter und erfahren, wie Sie Schritt für Schritt eine voll funktionsfähige Single-Page Webapplikation aufbauen können.

Kapitel 15

Single-Page Webapplikationen

Jedes Werkzeug erhält erst seine Vollendung dadurch, dass es nicht mehreren, sondern nur einem Zwecke dient.

– Aristoteles

Das Ziel der Webentwicklung ist es, einem Benutzer den Zugriff auf Informationen beziehungsweise auf eine Applikation über das Netzwerk zu ermöglichen. Dabei wird die Applikation über einen Webbrowser bedient. Die Vorteile dieser Vorgehensweise liegen auf der Hand. Die meisten Geräte verfügen heutzutage über einen Webbrowser, seien es nun Desktop-Rechner, Notebooks, Mobiltelefone oder Fernsehgeräte. Mit all diesen Geräten können Sie Ihre Applikation verwenden und müssen keine zusätzliche Software installieren. Traditionelle Webapplikationen bestehen aus einer Sammlung verschiedener Webseiten, mit denen ein Benutzer über Hyperlinks und Formulare kommuniziert. Single-Page Webapplikationen gehen hier einen anderen Weg. Statt komplette Seiten zu laden, werden hier nur die Teile der Seite neu geladen. Diese Vorgehensweise sorgt dafür, dass die Applikation responsiver wirkt und schneller auf Eingaben eines Benutzers reagiert. Das bedeutet für den Benutzer, dass der Client und der Server näher zusammenrücken und er unmittelbarer mit den Informationen der Applikation arbeiten kann. Eine solche Single-Page-Webapplikation fühlt sich schon eher wie eine Desktop-Applikation an. Da nicht mehr komplette Seiten neu geladen werden, sondern nur noch bestimmte Teile, reduziert sich die zu übertragende Datenmenge. Dies wiederum beschleunigt die Darstellung der Informationen beim Client. In diesem Kapitel sehen Sie, wie Sie mithilfe einiger Bibliotheken eine solche Single-Page Webapplikation erstellen können.

15.1 Die Aufgabenstellung

Die Aufgabe, die es zu lösen gilt, besteht darin, eine Verwaltungsoberfläche für eine Filmdatenbank zu erstellen. Dabei sollen Sie in einem ersten Schritt die Möglichkeit schaffen, dass sich ein Benutzer anmelden kann. Einmal angemeldet, kann sich der Benutzer dann eine Liste von Filmen anzeigen lassen und neue Filme hinzufügen. Haben Sie erst einmal diese Aufgaben gelöst, sollten Sie in der Lage sein, die Applikation eigenständig zu erweitern und zusätzliche Features wie das Editieren und Löschen von Datensätzen zu implementieren.

15.2 Setup

Bevor Sie mit der Umsetzung der Applikation beginnen können, müssen Sie einige Voraussetzungen schaffen. Die erste Bedingung, die erfüllt sein muss, ist, dass Node.js auf Ihrem System installiert ist. Die nun folgenden Abschnitte geleiten Sie durch den Setup-Prozess, an dessen Ende Sie sämtliche benötigten Komponenten so platziert haben werden, dass Sie mit der Implementierung starten können.

15.2.1 Ordnerstruktur

Die MovieDB-Applikation ist etwas umfangreicher als die bisherigen Beispiele. Aus diesem Grund bietet sich hier auch eine sinnvolle Strukturierung der Dateien an, damit Sie den Überblick nicht verlieren, auch wenn Sie die Applikation weiterentwickeln. Listing 15.1 stellt Ihnen eine einfache Verzeichnisstruktur vor, die Sie für Ihre Applikation verwenden können.

```
$ tree .
.
├─ app
│   └─ views
├─ db
├─ node_modules
└─ public
    ├─ js
    │   ├─ app
    │   │   └─ models
    │   │       └─ views
    │   └─ lib
    └─ style
```

Listing 15.1 Verzeichnisstruktur der MovieDB-Applikation

Das Verzeichnis *app* enthält die Dateien mit dem Quellcode der Backend-Applikation inklusive der Template-Dateien, die in einem separaten *views*-Unterverzeichnis abgelegt werden. Im *db*-Ordner wird die Datenbank gespeichert. Um das *node_modules*-Verzeichnis müssen Sie sich nicht weiter kümmern, da es automatisch erstellt wird, wenn Sie die erforderlichen NPM-Pakete installieren. Im *public*-Verzeichnis liegen schließlich alle Dateien und Verzeichnisse für die Frontend-Logik. Auf die Dateien in diesem Verzeichnis haben die Benutzer Ihrer Applikation direkten Zugriff über ihren Browser. In der ersten Ebene enthält das *public*-Verzeichnis zwei Unterverzeichnisse. Im *style*-Ordner liegen die Stylesheets für die Applikation, und unter *js* finden Sie sämtliche für das Frontend relevanten JavaScript-Dateien. Diese werden weiter untergliedert in *lib* für die verschiedenen Bibliotheken und *app* für die Applikationslogik im

Frontend mit den Verzeichnissen *models* und *views*. Dazu aber später mehr. Zunächst sollten Sie sich erst einmal der Grundlage der Applikation widmen.

15.2.2 Die Datenbank

Die Basis für Ihre Applikation bildet eine Datenbank, in der Sie sämtliche Informationen vorhalten. Für die vorliegende Aufgabenstellung kommen verschiedene Datenbanktypen infrage. Zur Modellierung der Datenstrukturen eignen sich sowohl dokumentenbasierte Datenbanken wie MongoDB als auch relationale Datenbanken wie MySQL. Für die hier angestrebte Lösung kommt eine relationale Datenbank zum Einsatz. Damit Ihre Applikation leichtgewichtig wird und mit möglichst wenigen Abhängigkeiten auskommt, sollten Sie SQLite als Datenbanksystem einsetzen. Um die Datenbank zu erstellen, müssen Sie die SQLite3-Shell verwenden. Wechseln Sie zu diesem Zweck in das *db*-Verzeichnis Ihrer Applikation und setzen das Kommando `sqlite3 movieDB ab`. In Listing 15.2 sehen Sie die Kommandos, die benötigt werden, um die Datenbank aufzusetzen.

```
CREATE TABLE users (
    username VARCHAR(255),
    password VARCHAR(255)
);

CREATE TABLE movies (
    name VARCHAR(255),
    year INTEGER(4)
);

INSERT INTO users VALUES ('admin', 'test');
INSERT INTO movies VALUES ('Iron Man', 2008);
```

Listing 15.2 Setup der Datenbank

Mit den Kommandos aus Listing 15.2 erstellen Sie zwei Tabellen und legen jeweils einen ersten Datensatz eines Benutzers und eines Films an.

15.2.3 Abhängigkeiten

Nachdem Sie mit der Datenbank die Grundlage für die Datenhaltung Ihrer Applikation geschaffen haben, müssen Sie sich als Nächstes um die Abhängigkeiten Ihrer Applikation kümmern. Die einfachste Lösung hierfür besteht in der Erstellung einer *package.json*-Datei im Wurzelverzeichnis Ihrer Applikation. Diese Datei enthält neben einer Vielzahl weiterer Informationen die Beschreibung der Abhängigkeiten. Für die Entwicklung reicht es allerdings aus, wenn Sie lediglich die Abhängigkeiten wie in Listing 15.3 definieren.


```

{
  "dependencies": {
    "express": ">=3.2.x",
    "jade": ">=0.30.x",
    "sqlite3": ">=2.1.x"
  }
}

```

Listing 15.3 Auflösung der Abhängigkeiten über die `package.json`-Datei

Die in der `package.json`-Datei aufgelisteten Abhängigkeiten können Sie ganz einfach über den Aufruf des Kommandos `npm install` im Wurzelverzeichnis Ihrer Applikation auflösen lassen. Der Paketmanager lädt daraufhin sämtliche Pakete und ihre Abhängigkeiten herunter und installiert sie im `node_modules`-Verzeichnis.

15.2.4 Clientbibliotheken

Sie greifen allerdings nicht nur serverseitig auf etablierte Bibliotheken zurück, sondern setzen auch clientseitig auf solche Unterstützung. Der Vorteil besteht darin, dass Ihnen Bibliotheken und Frameworks häufig auftretende Aufgaben abnehmen und Ihnen dadurch viel Schreiarbeit ersparen. Das Frontend Ihrer Filmdatenbank soll auf dem Strukturframework Backbone.js (<http://backbonejs.org>) aufbauen.

Backbone.js Models

Backbone.js sorgt dafür, dass die Daten Ihrer Applikation in Form von Objekten verwaltet werden. Hierfür kommen sogenannte Models zum Einsatz. In Backbone.js besteht ein Model in erster Linie aus einer Reihe von Attributen, den Eigenschaften des Models. Des Weiteren wird ein Model durch eine eindeutige ID identifiziert. Das Framework stellt Ihnen außerdem zahlreiche Methoden zur Verfügung, mit denen Sie den gesamten Lebenszyklus eines Models verwalten können. Dabei wird sowohl der Erstellungsprozess erheblich vereinfacht als auch das Speichern des Models auf einem Server. Als Schnittstelle zum Server kommt REST zum Einsatz. Backbone.js sorgt beispielsweise dafür, dass beim Erstellen eines neuen Models korrekterweise POST als HTTP-Methode verwendet wird, wohingegen bei der Aktualisierung bestehender Daten PUT benutzt wird. Das Handling der Daten ist jedoch nur ein Teil des Funktionsumfangs von Backbone.js.

Backbone.js Collections

Dieses Framework bietet Ihnen außerdem die Möglichkeit, mehrere gleichartige Models zu einer sogenannten Collection zusammenzufassen. Eine Collection ist am ehesten mit einem Array von Objekten vergleichbar, bis auf die Tatsache, dass Back-

bone.js über zahlreiche Erweiterungen verfügt. So können Sie beispielsweise nach einem Model in der Collection suchen, die Collection sortieren oder nach bestimmten Kriterien filtern. Es existiert außerdem eine komfortable Schnittstelle, mit der Sie über sämtliche Models einer Collection iterieren können. Beim Einfügen und Entfernen von Elementen einer Collection unterstützt Sie Backbone.js mit Methoden wie `add` und `remove`. Die Models und Collections dienen nur der Organisation von Datenstrukturen. Die Benutzer Ihrer Webseiten bekommen von diesen Konstrukten auf der Webseite nichts mit.

Backbone.js Views

Für die Darstellung der einzelnen Konstrukte verfügt Backbone.js über die sogenannten Views. Das sind wiederum Objekte, die eng an Models geknüpft sind und für deren Anzeige innerhalb der Applikation sorgen. Dabei sollten Sie beachten, dass für ein Model ein oder mehrere Views existieren können, allerdings in einer View in den meisten Fällen nur ein Model dargestellt wird. Die Kopplung zwischen Model und View findet statt, indem die View ihr Model kennt und über eine Eigenschaft `model` verfügt, die eine Referenz auf das Model enthält. Das Model wiederum weiß nichts von der Existenz seiner Views. Der Vorteil dieser Vorgehensweise liegt darin, dass sich das Model nicht darum kümmern muss, wo und wie häufig es in der Applikation dargestellt wird. Ändern sich die Eigenschaften eines Models, müssen Sie innerhalb der jeweiligen View-Objekte dafür sorgen, dass die Informationen korrekt dargestellt werden.

Eventgetriebener Ansatz von Backbone.js

Backbone.js basiert auf einem eventgetriebenen System. Das hat zur Konsequenz, dass viele Aktionen, die in Ihrer Applikation stattfinden, entweder dafür sorgen, dass ein oder mehrere Events ausgelöst werden, oder aber, dass bestimmte Aktionen aufgrund von Events durchgeführt werden. Im Falle der Model-View-Verbindung kommen Events konkret zum Einsatz, wenn sich die Struktur des Models ändert, Sie also ändernd auf die Eigenschaften des Models zugreifen. In diesem Fall wird ein `change`-Event ausgelöst, auf das die View reagieren und sich komplett oder teilweise neu darstellen kann. Klar wird der Vorteil dieses Systems vor allem, wenn Sie sich vor Augen führen, dass Sie ein Model innerhalb einer Applikation über verschiedene Views mehrmals darstellen und dass sämtliche dieser Darstellungen bei einer Änderung des Models angepasst werden, ohne dass Sie die Seite neu laden müssen. Beim Einsatz von Backbone.js sind Sie nicht darauf beschränkt, Views nur mit Models zu verwenden. Sie können sie auch mit Collections benutzen. Im Zuge der Umsetzung der Filmdatenbank werden Sie diese Mechanismen noch genauer kennenlernen.

Backbone.js Router

Der Funktionsumfang von Backbone.js endet allerdings noch nicht bei den Models, Views und Collections. Backbone.js bietet Ihnen mit dem Router ein Hilfsmittel, mit dem Sie innerhalb Ihrer Applikation navigieren können. Der Router verfolgt ein ähnliches Konzept wie auch express.js. Sie definieren bestimmte Routen, an die Funktionen gebunden sind. Verwendet ein Benutzer Ihrer Applikation nun diese Route, wird die Funktion ausgeführt. Im Gegensatz zu express.js werden die Routen bei Backbone.js jedoch nicht über URL-Aufrufe ausgeführt, sondern entweder über Änderungen im Hash-Teil der URL beziehungsweise mit HTML5 dann über PushState. Über diesen Mechanismus werden Single-Page Webapplikationen erst wirklich möglich.

Lodash

Backbone.js existiert hauptsächlich, damit es Struktur in Ihre Applikation bringt, und auf diese Aufgabe konzentriert sich die Bibliothek. Wegen dieser Fokussierung auf einen bestimmten Zweck benötigen Sie beim Einsatz von Backbone.js in Ihrer Applikation noch weitere Bibliotheken, die andere Aufgaben übernehmen. Für DOM-Manipulationen und die Verbindung zum Server nutzt Backbone.js beispielsweise die Schnittstelle von jQuery. Für viele Aufgaben im Bereich der Collections kommt mit Lodash eine weitere Bibliothek zum Einsatz. Diese sorgt beispielsweise dafür, dass Sie mit der `forEach`-Methode einer Collection über die einzelnen Elemente iterieren oder mithilfe von `find` ein bestimmtes Element der Collection suchen können. Ursprünglich kam für diesen Zweck die Bibliothek Underscore.js zum Einsatz. Lodash stellt eine schnittstellenkompatible Verbesserung und Modernisierung dieser Bibliothek dar.

RequireJS

Je größer und komplexer Ihre Applikation wird, desto unübersichtlicher wird Ihr Quellcode. Backbone.js geht bereits einen Schritt in die richtige Richtung und sorgt dafür, dass die Elemente Ihrer Applikation in verschiedenen Models, Views und Collections organisiert werden. Es verfügt allerdings nicht über die Funktionalität eines Modulloaders. In Node.js kennen Sie für diesen Fall das Modulsystem und die `require`-Funktion, die auf einem Spezifikationsvorschlag von CommonJS aufbauen. Weitere Details zu CommonJS finden Sie unter <http://www.commonjs.org/>. Für clientseitiges JavaScript existieren ebenfalls Implementierungen von Modulloadern auf Basis von CommonJS. Ein häufig eingesetzter und von anderen Bibliotheken unabhängiger Modulloader ist RequireJS. Die Webseite dieses Projekts finden Sie unter <http://requirejs.org/>. In Ihrer Filmdatenbank-Applikation werden Sie RequireJS dafür einsetzen, um die verschiedenen Komponenten Ihrer Applikation zu laden und die Abhängigkeiten zwischen den einzelnen Modulen und zu den externen Bibliotheken aufzulösen.

Twitter Bootstrap

Die View-Komponente von Backbone.js sorgt zwar für die Darstellung der Models im Frontend, bietet Ihnen jedoch keinerlei Layoutunterstützung. Für einen schnellen und einfachen Start bei der Umsetzung von Layouts in Webapplikationen können Sie Twitter Bootstrap (<http://twitter.github.io/bootstrap/>) einsetzen. Bootstrap ist eine Sammlung von Komponenten, die Sie bei Standardaufgaben wie dem Layout von Navigationselementen, Formularen oder Dialogen unterstützen. Bootstrap besteht aus einem JavaScript- und einem CSS-Teil, wobei der JavaScript-Teil die Logik der einzelnen Komponenten enthält und der CSS-Teil dem Styling der Elemente dient.

Alle vorgestellten Bibliotheken werden im Verzeichnis `/public/js/lib` abgelegt. Listing 15.4 zeigt Ihnen die Liste der Dateien in diesem Verzeichnis.

```
.
├─ backbone-min.js
├─ bootstrap.min.js
├─ jquery-1.10.1.min.js
├─ lodash.min.js
└─ require.js
```

Listing 15.4 Clientseitige Bibliotheken unter `/public/js/lib`

Alle diese Komponenten werden Sie im Verlauf dieses Kapitels zu einer Single-Page Webapplikation zusammenfügen und dabei erfahren, dass vor allem bei umfangreicheren Applikationen der Einsatz von Bibliotheken und Frameworks zur Strukturierung unerlässlich ist.

15.3 Die Applikation

Sie erfahren nun Schritt für Schritt, wie Sie beim Aufbau einer Single-Page Webapplikation auf Basis der bereits vorgestellten Bibliotheken vorgehen müssen. Im Zuge der ersten Teilaufgabe, dem Login, bauen Sie zunächst die Infrastruktur für die weiteren Lösungen auf.

15.3.1 Login

Bevor Sie sich an die Umsetzung der einzelnen Features machen können, müssen Sie die wichtigste Komponente Ihrer Applikation, den Einstieg, implementieren. Diese Datei übergeben Sie als Option dem Aufruf von Node.js. Sie enthält sämtliche Anweisungen, die für die Funktionsfähigkeit Ihrer Applikation nötig sind. Erstellen Sie zu diesem Zweck im Wurzelverzeichnis eine Datei mit dem Namen `index.js`. In Listing 15.5 sehen Sie den Inhalt dieser Datei.

```

var express = require('express');
var MDB = require('./app/mdb.js');
var app = express();

app.set('views', __dirname + '/app/views');
app.set('view engine', 'jade');

app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.cookieSession({secret: 'mySecret'}));

app.use('/js', express.static(__dirname + '/public/js'));
app.use('/style', express.static(__dirname + '/public/style'));

app.get('/', MDB.login);

app.post('/login', MDB.processLogin);

app.get('/logout', MDB.logout);

app.get('/mdb', MDB.checkAuth, MDB.mdb);

app.listen(8080);

```

Listing 15.5 Der Einstieg in die Applikation – index.js

Damit der Einstieg auch bei umfangreicheren Applikationen übersichtlich bleibt, sollen Sie stets darauf bedacht sein, so wenig Logik wie möglich in dieser Datei abzuliegen. In Listing 15.5 liegt die meiste Logik der Applikation im `MDB`-Modul, was für »Movie Database-Modul« steht. Dieses Modul und `express.js` werden in dieser Applikation eingebunden. Im Anschluss konfigurieren Sie Jade als View Engine für Ihre Applikation und binden eine Reihe von Middleware-Komponenten ein, die Sie im Verlauf der Umsetzung des Login-Prozesses noch benötigen werden. Mithilfe der `static`-Middleware liefern Sie sowohl die Stylesheets aus dem Verzeichnis `/public/style` als auch die JavaScript-Dateien für das Frontend unter `/public/js` aus. Als Nächstes müssen Sie nun dafür sorgen, dass die einzelnen Funktionen, die durch die verschiedenen Routen aufgerufen werden, auch implementiert sind. Listing 15.6 enthält den JavaScript-Quellcode des `MDB`-Moduls, der für den Login-Prozess erforderlich ist.

```

var MDB = {
  _db: null,

  getDb: function () {

```

```

    if (!MDB._db) {
        var sqlite3 = require('sqlite3').verbose();
        MDB._db = new sqlite3.Database('db/movieDB');
    }
    return MDB._db;
},

checkAuth: function (req, res, next) {
    if (!req.session.user) {
        res.redirect('/');
    } else {
        next();
    }
},

login: function (req, res, next) {
    if (req.session.user) {
        res.redirect('/mdb');
    }
    res.render('login');
},

processLogin: function (req, res, next) {
    var data = [req.body.username, req.body.password];
    var db = MDB.getDb();

    db.get('SELECT * FROM users WHERE username = ? AND password = ?
', data,
    function (err, row) {
        if (row) {
            req.session.user = row.username;
            res.redirect('/mdb');
        } else {
            res.redirect('/');
        }
    });
},

logout: function (req, res, next) {
    delete req.session.user;
    res.redirect('/');
},

mdb: function (req, res, next) {

```

```

        res.send('successfully logged in');
    }

};

module.exports = MDB;

```

Listing 15.6 Komponenten des Login-Prozesses – `app/mdb.js`

Da die Benutzerdaten in der Datenbank liegen, müssen Sie zunächst eine Verbindung zur Datenbank aufbauen. Dies geschieht in der `getDB`-Methode. Die `checkAuth`-Methode stellt eine Middleware-Komponente dar, mit deren Hilfe Sie prüfen können, ob sich der aktuelle Benutzer bereits angemeldet hat. Die zuvor eingebundene `cookie-session`-Middleware stellt Ihnen zu diesem Zweck die Eigenschaft `req.session` zur Verfügung, mit der eine Session simuliert wird, die über die gesamte Verbindungszeit des Clients bestehen bleibt und somit Seitenreloads überdauert. In der Eigenschaft `req.session.user` wird der Name des aktuellen Nutzers gespeichert. Ist dieser nicht gesetzt, ist dies ein Zeichen dafür, dass sich der Benutzer nicht korrekt angemeldet hat. In diesem Fall leiten Sie die Anfrage zurück auf das Anmeldeformular. Die `login`-Methode sorgt schließlich dafür, dass dem Benutzer das Anmeldeformular präsentiert wird, allerdings nur, wenn er sich noch nicht angemeldet hat. Ist er bereits angemeldet, wird er gleich auf die `/mdb`-Route weitergeleitet. Innerhalb der `processLogin`-Methode, die durch das Absenden des Anmeldeformulars aufgerufen wird, prüfen Sie, ob die Kombination aus Benutzername und Passwort in der Datenbank existiert. Ist sie vorhanden, wird der Benutzer angemeldet, falls nicht, wird wiederum das Anmeldeformular angezeigt. Die `logout`-Methode dient Ihren Benutzern dazu, sich wieder korrekt von der Applikation abzumelden. Innerhalb dieser Methode müssen Sie lediglich dafür sorgen, dass die Eigenschaft `req.session.user` geleert und die Anfrage auf die Anmeldeseite weitergeleitet wird. Die `mdb`-Methode stellt schließlich das Ziel des Anmeldeprozesses dar. In dieser Methode sorgen Sie später dafür, dass die Liste der Filme angezeigt wird. Was nun noch fehlt, ist die Umsetzung des Anmeldeformulars, damit sich ein Benutzer an der Applikation anmelden kann. Listing 15.7 zeigt Ihnen den Inhalt der Datei `login.jade`, die Sie im Verzeichnis `app/views` speichern müssen, damit das Template durch die `login`-Methode korrekt gerendert werden kann.

```

!!!

html
  head
    link(rel="stylesheet", type="text/css", href="/style/bootstrap.min.css")

  body
    div.row-fluid

```

```

div.span3 &nbsp;
div.well.span6
  form(method="post", action="login", class="form-horizontal")
    div.control-group
      label(for="username", class="control-label") Username:
      div.controls
        input(type="text", name="username")
    div.control-group
      label(for="password", class="control-label") Passwort:
      div.controls
        input(type="password", name="password")
    div.control-group
      div.controls
        input(type="submit", value="OK", class="btn")
div.span3 &nbsp;

```

Listing 15.7 Template des Anmeldeformulars – app/views/login.jade

Die Datei *bootstrap.min.css* stammt wie die Datei *bootstrap.min.js* aus dem ZIP-Archiv von Twitter Bootstrap und bietet Ihnen eine Reihe von Standard-Layoutkomponenten, die Sie auf Ihrer Seite einsetzen können.

Starten Sie nun Ihre Applikation mit dem Kommando `node index.js` im Wurzelverzeichnis Ihrer Applikation und öffnen in Ihrem Browser die URL *http://localhost:8080*, sehen Sie eine Seite, die der in Abbildung 15.1 gleicht.

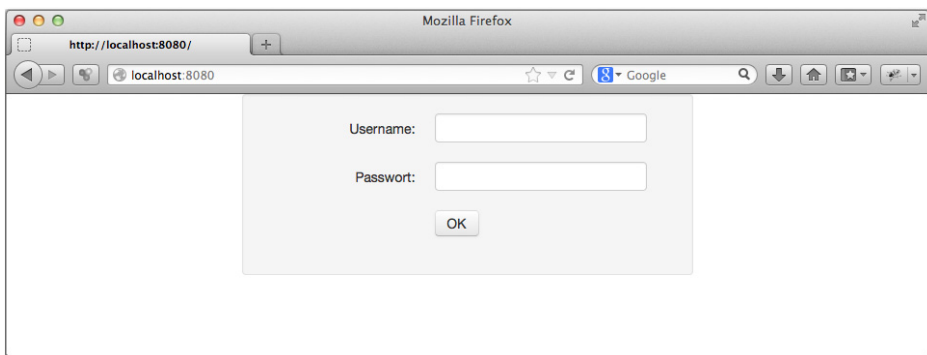


Abbildung 15.1 Anmeldemaske

Beim Aufbau der Datenbank in Listing 15.2 haben Sie bereits einen ersten Benutzer angelegt. Das bedeutet, dass Sie sich mit der Kombination aus dem Benutzernamen `admin` und dem Passwort `test` schon an Ihrer Applikation anmelden können. Danach werden Sie zur eigentlichen Applikation weitergeleitet, was auch der einzige erforder-

derliche Seitenreload bleiben sollte. Aktuell wird hier noch die Zeichenkette `successfully logged in` ausgegeben.

15.3.2 Liste der vorhandenen Datensätze

Nun können Sie die Methode `mdb` in der Datei `/app/mdb.js` dahingehend erweitern, dass Ihren Benutzern die Liste der aktuell existierenden Filme angezeigt wird. In Listing 15.8 finden Sie den entsprechenden Quellcode.

```
mdb: function (req, res, next) {
  var db = MDB.getDb();

  db.all('SELECT rowid, * FROM movies', function (err, data) {
    res.render('mdb', {"data": JSON.stringify(data)});
  });
}
```

Listing 15.8 Erweiterung der `mdb`-Methode – `app/mdb.js`

Die `mdb`-Methode sorgt nun dafür, dass sämtliche bestehenden Datensätze aus der Datenbank ausgelesen werden und als Zeichenkette im JSON-Format an das Template `mdb.jade` übergeben werden. Dieses Template befindet sich im gleichen Verzeichnis wie auch schon das `login.jade`-Template, also unter `/app/views`. Das Template sorgt dafür, dass Ihre Applikation im Frontend aufgebaut wird. Listing 15.9 enthält den Quellcode dieses Templates.

```
!!!

html
  head
    link(rel="stylesheet", type="text/css", href="/style/bootstrap.min.css")

  body
    div#main

      a#newMovie neuen Film anlegen

    script(data-main="/js/app/config", src="/js/lib/require.js")

    script
      var data = !{data}
```

Listing 15.9 Einstieg in die Applikation im Frontend – `app/views/mdb.jade`

Für die Tatsache, dass dieses Template eigentlich Ihre gesamte Applikation beinhalten sollte, weist es sehr wenige Zeilen Quellcode auf. Der Grund hierfür ist, dass die Kontrolle an das Modulsystem von RequireJS übergeben wird. Zunächst binden Sie allerdings das Stylesheet von Bootstrap ein, das Sie beim Styling Ihrer Applikation unterstützen soll. Außerdem definieren Sie den Applikations-Container, in den später die Inhalte platziert werden sollen. Die Zeile `script(data-main="/js/app/config", src="/js/lib/require.js")` ist dafür verantwortlich, dass die Applikation gestartet wird. In den letzten beiden Zeilen des Templates sorgen Sie dafür, dass die bestehenden Daten über die Filme in der globalen Variablen `data` verfügbar sind.

Mit der Angabe `data-main` übergeben Sie die Konfiguration Ihrer Applikation an RequireJS. Die Angabe in diesem Attribut enthält den Dateinamen der Konfigurationsdatei, also `/js/app/config.js`. Im `src`-Attribut laden Sie RequireJS. Listing 15.10 zeigt Ihnen die Konfiguration der Applikation.

```
require.config({
  "deps": ["main"],

  "paths": {
    "jquery": "../lib/jquery-1.10.1.min",
    "lodash": "../lib/lodash.min",
    "backbone": "../lib/backbone-min",
    "bootstrap": "../lib/bootstrap.min"
  },

  "shim": {
    "backbone": {
      "deps": [
        "jquery",
        "lodash"
      ],
      exports: "Backbone"
    },

    "bootstrap": {
      "deps": [
        "jquery"
      ]
    }
  }
});
```

Listing 15.10 Frontend-Konfiguration der Applikation – `public/js/app/config.js`

Über die Konfiguration machen Sie RequireJS die einzelnen Bibliotheken bekannt, die Sie in Ihrer Applikation verwenden. So definieren Sie über die Direktive *paths* beispielsweise, dass jQuery unter *../lib/jquery-1.10.1-min.js* zu finden ist. Im weiteren Verlauf Ihrer Applikation können Sie über *jquery* auf diese Bibliothek zugreifen, und RequireJS lädt die Datei beziehungsweise liefert den Inhalt bei mehrmaligem Aufruf aus dem internen Cache aus. Über die Direktive *shim* können Sie, wie Sie in Listing 15.10 sehen, Abhängigkeiten zu anderen Bibliotheken definieren. Backbone.js basiert beispielsweise auf jQuery und lodash. Ohne diese beiden Bibliotheken ist Backbone.js nicht funktionsfähig. Sie müssen also dafür sorgen, dass zuerst die beiden anderen Bibliotheken geladen werden. Über die Angabe von *deps* sorgen Sie dafür, dass Backbone.js erst geladen wird, sobald die anderen beiden vorhanden sind. Ähnlich verfahren Sie mit dem JavaScript-Teil von Bootstrap, der ebenfalls erfordert, dass jQuery geladen ist. Wichtig ist ebenfalls die Angabe *"deps": ["main"]*. Sie lädt die Datei */public/js/app/main.js* und startet damit die Applikation. In dieser Datei instanziiieren Sie den Router und starten die Navigation über PushState. In Listing 15.11 finden Sie den Quellcode der *main.js*-Datei.

```
require(["app", "router"], function (app, Router) {
    app.router = new Router();
    Backbone.history.start({ pushState: true, root: app.root });
    app.router.navigate(app.root, {trigger: true});
});
```

Listing 15.11 Bootstrapping der Applikation – *public/js/app/main.js*

In Listing 15.11 sehen Sie den typischen Aufbau eines RequireJS-Moduls. Es beginnt immer mit einem Aufruf der *require*- beziehungsweise der *define*-Methode. Bei *require* geben Sie im ersten Argument in einem Array die Module an, die geladen werden sollen. Das zweite Argument besteht aus einer Callback-Funktion, die ihrerseits wiederum die Rückgabewerte der Module als Argumente erhält. Mit dem Aufruf von *app.router.navigate* sorgen Sie dafür, dass die Standardroute aufgerufen wird. Als Abhängigkeiten der *main.js*-Datei geben Sie hier die Dateien *app.js* und *router.js* an. Das ist zum einen das globale Scope-Objekt, das Sie in Ihrer Applikation dazu verwenden können, um den aktuellen Status modulübergreifend festzuhalten. Zum anderen benötigen Sie in der *main.js*-Datei den Router, damit Sie das Routing Ihrer Applikation initialisieren können. Beide Module sehen Sie im Folgenden im Detail. In der Datei *app.js* erstellen Sie ein Objekt, in dem Sie später beispielsweise die Collection vorhalten, in der die einzelnen Filme gespeichert sind. Listing 15.12 zeigt Ihnen den Aufbau dieser Datei.

```
define(["backbone", "bootstrap"], function() {
    return {
        root: "/"
    };
});
```

Listing 15.12 Der globale Applikations-Namespace – `public/js/app/app.js`

Die zweite Datei, die im Rahmen der *main.js*-Datei eingebunden wird, ist *router.js*. Der Router ist dafür zuständig, auf die Aktionen des Benutzers zu reagieren. Listing 15.13 enthält den Quellcode des Routers.

```
define(["backbone", "app", "models/movies", "views/movies", "models/
movie", "views/movie"],
    function(Backbone, app, Collection, CollectionView, Model, View) {
    var Router = Backbone.Router.extend({
        routes: {
            "": "index",
        },

        index: function() {
            if (!app.collection) {
                app.collection = new Collection();
                app.collection.reset(data);
            }

            var cv = new CollectionView({model: app.collection});
            cv.render();

            app.collection.each(function (model) {
                new View({model: model});
                model.trigger('change');
            });

            $('#newMovie').off('click');
            $('#newMovie').on('click', function () {
                app.router.navigate('/new', {trigger: true});
            });
        }
    });
});
```

```
    return Router;
  });
```

Listing 15.13 Der Router – `public/js/app/router.js`

Das Kernstück des Routers ist die Eigenschaft `routes`. Diese verbindet die Route, die Sie über `PushState` aktivieren, mit einer Methode des Routers. Im ersten Schritt benötigen Sie lediglich eine Route. Diese verbindet eine leere Zeichenkette mit der Methode `index`. In dieser Methode wird zuerst die Collection, die die Filme enthält, mit dem globalen `data`-Objekt befüllt. Danach wird für die Collection eine eigene View erstellt und durch einen Aufruf der `render`-Methode angezeigt. Schließlich iterieren Sie mit der `each`-Methode der Collection über jedes Model der Collection, erzeugen für jedes Model eine View, weisen dieser das jeweilige Model zu und sorgen durch das Auslösen des `change`-Events dafür, dass die View dargestellt wird. Außerdem bereiten Sie schon das Anlegen neuer Datensätze vor, indem Sie den Link mit der ID `newMovie` mit der Route `/new` verbinden. Doch dazu später mehr. Zuerst erfahren Sie mehr über die einzelnen Komponenten Ihrer Applikation. Im Folgenden sollten Sie das Model implementieren, das die Repräsentation eines Films übernimmt und in der Datei `/public/js/app/models/movie.js` liegt. In Listing 15.14 sehen Sie den zugehörigen Quellcode.

```
define("models/movie", ['backbone'], function (Backbone) {
  return Backbone.Model.extend({
    idAttribute: "rowid",
    url: '/movie'
  });
});
```

Listing 15.14 Das movie-Model – `public/js/app/models/movie.js`

Die Standardimplementierung eines Backbone-Models ist beinahe ausreichend für die Lösung der Aufgabe. Im Aufruf der `define`-Methode von `RequireJS` geben Sie dem Modul zunächst einen Namen, in diesem Fall `models/movie`. Danach definieren Sie in einem Array die Abhängigkeiten, und schließlich geben Sie eine Callback-Funktion an, die ausgeführt wird, sobald die Abhängigkeiten geladen sind. Innerhalb dieser Funktion leiten Sie von `Backbone.Model` ab und überschreiben die Eigenschaft `idAttribute` mit dem Wert `rowid`. Diese Eigenschaft gibt an, über welches Feld ein Datensatz identifiziert wird. Des Weiteren geben Sie über die Eigenschaft `url` an, über welche URL die Daten ausgelesen beziehungsweise gespeichert werden können. Sehr eng an das Model gekoppelt ist die View, die in der Datei `/public/js/app/views/movie.js` gespeichert wird. Den dafür erforderlichen Quellcode enthält Listing 15.15.

```

define(['text!views/movie.html'], function (template) {
  return Backbone.View.extend({

    tagName: 'tr',

    initialize: function () {
      this.listenTo(this.model, "change", this.render);
    },

    render: function () {
      var data = {
        name: this.model.get('name'),
        year: this.model.get('year')
      };

      this.$el.html(_.template(template, data));
      $('#movieTable tbody').append(this.$el);
    }
  });
});

```

Listing 15.15 Die movie-View – `public/js/app/views/movie.js`

Eine Backbone-View besteht im Normalfall aus zwei Teilen. Den ersten Teil sehen Sie in Listing 15.15. Er besteht aus einem Objekt, das dafür zuständig ist, das Model darzustellen. Der zweite Teil ist ein Template, das die HTML-Elemente zur eigentlichen Darstellung enthält. Über das `text`-Plugin von RequireJS können Sie die Templates einbinden und über eine Variable darauf zugreifen. Um dieses Plugin zu aktivieren, müssen Sie, wie Sie in Listing 15.15 sehen, dem Dateinamen des Templates die Zeichenkette `text!` voranstellen. Das Plugin ist auf GitHub unter <https://github.com/requirejs/text> verfügbar. Sie müssen die Datei im `app`-Verzeichnis Ihrer Applikation speichern. Eine weitere Konfiguration ist nicht erforderlich.

Jedes View-Objekt unterstützt die Eigenschaften `id` und `className`, die angeben, welchen Wert das `id`-beziehungsweise `class`-Attribut der View-Repräsentation enthalten soll. Über die Eigenschaft `tagName` geben Sie schließlich an, welche Art von Element die View sein soll. Im Falle der `movie`-View ist dies eine Tabellenzeile. In der `initialize`-Methode sorgen Sie dafür, dass sich die View immer darstellt, wenn das `change`-Event des Models ausgelöst wird. Die `render`-Methode holt die Daten des Models ab, rendert das HTML-Template über die `lodash`-Template-Methode und hängt die View-Repräsentation anschließend in die Collection-View ein, die durch den Selektor `.movieTable tbody` lokalisiert wird. Listing 15.16 enthält das zur `movie`-View gehörende Template, das in der Datei `/public/js/app/views/movie.html` gespeichert ist.

```
<td><%= name %></td>
<td><%= year %></td>
```

Listing 15.16 Das movie-Template – public/js/app/views/movie.html

Das Template besteht nur aus zwei Tabellenzellen, die jeweils einen Platzhalter im lodash-Template-Format enthalten. Die Models werden in einer Collection gesammelt. Backbone.js definiert zu diesem Zweck eine separate Klasse, die bereits einen umfangreichen Satz an Methoden mitbringt. Aus diesem Grund ist die Implementierung dieser Collection in Ihrer Applikation auch recht übersichtlich, wie Listing 15.17 zeigt.

```
define(['backbone', 'models/movie'], function (Backbone, Movie) {
  return Backbone.Collection.extend({
    model: Movie
  });
});
```

Listing 15.17 Die movie-Collection – public/js/app/models/movies.js

In Listing 15.17 sehen Sie, dass Sie nicht zwangsläufig einen Namen für das Modul vergeben müssen. Für die Collection benötigen Sie lediglich Backbone.js selbst und das movie-Model als Abhängigkeit. Sie müssen dann nur noch von der Backbone.Collection ableiten und die Eigenschaft `model` überschreiben, die angibt, welche Art von Models die Collection enthält. Die Darstellung der Collection ähnelt der View des movie-Models. Die View ist auch hier wieder in die View-Klasse und das Template unterteilt. Listing 15.18 enthält den Quellcode für die Klasse.

```
define(['text!views/movies.html'], function (template) {
  return Backbone.View.extend({
    tagName: 'table',
    className: 'movieTable table table-striped table-hover',

    render: function () {
      this.$el.html(_.template(template, {}));
      $('#main').append(this.$el);
    }
  });
});
```

Listing 15.18 Die Collection-View – public/js/app/views/movies.js

Auch hier binden Sie das Template ein, erstellen über die Eigenschaften `tagName` und `className` ein View-Element und füllen dieses in der `render`-Methode mit Inhalt und platzieren es schließlich im `main`-Container Ihrer Seite. Die View-Klasse der Collection liegt in der Datei `/public/js/app/views/movies.js`. Das zugehörige Template speichern

Sie unter `/public/js/app/views/movies.html`. Der Inhalt dieses Templates ist ähnlich einfach wie schon der `movie-View`, wie Ihnen Listing 15.19 veranschaulicht.

```
<thead>
  <tr>
    <th>Titel</th>
    <th>Jahr</th>
  </tr>
</thead>
<tbody></tbody>
```

Listing 15.19 Das `movies`-Template – `public/js/app/views/movies.html`

Mit diesen Anpassungen am Quellcode Ihrer Applikation haben Sie nun den nächsten Teil der Aufgabe gelöst. Sie können nach einem Neustart Ihrer Applikation das Resultat testen, indem Sie die URL `http://localhost:8080` aufrufen und sich anmelden. Das Ergebnis sehen Sie in Abbildung 15.2.

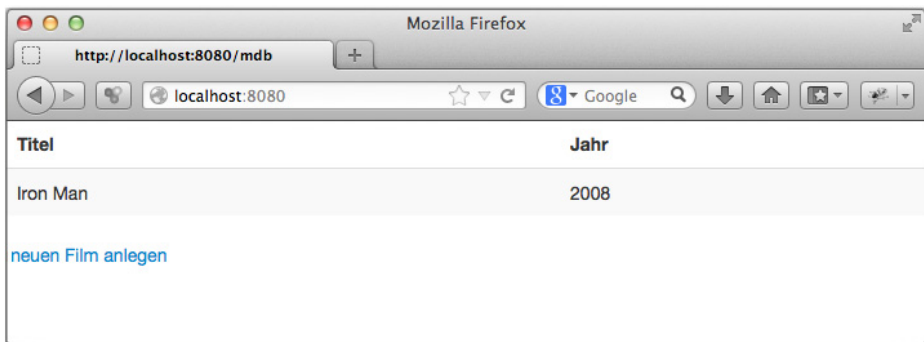


Abbildung 15.2 Liste der vorhandenen Filme

Der letzte Teil der Aufgabe besteht aus dem Anlegen neuer Datensätze. Im folgenden Abschnitt erfahren Sie, wie Sie hier vorgehen müssen.

15.3.3 Neue Datensätze anlegen

Für das Anlegen neuer Datensätze haben Sie bereits die Vorarbeit geleistet, indem Sie im Template `mdb.jade` den Link angelegt und diesen im Router bereits mit der Route `/new` verbunden haben. Im nächsten Schritt müssen Sie nun dafür sorgen, dass der Router diese Route auch kennt und eine entsprechende Funktion daran geknüpft ist. Listing 15.20 enthält die Erweiterung des Routers.

```
define([..., "views/new"],
  function(..., NewView) {
    var Router = Backbone.Router.extend({
```



```

    routes: {
...
        "new": "addMovie"
    },
...
    addMovie: function () {
        var movie = new Model();
        var newView = new NewView({model: movie});
        newView.render();
    }
    });
...

```

Listing 15.20 Erweiterung des Routers – `public/js/app/router.js`

Für das Anlegen neuer Filme in der Applikation benötigen Sie eine weitere View, die Sie im Aufruf der `define`-Funktion einbinden müssen. Außerdem müssen Sie die Eigenschaft `routes` um die Route `new` erweitern und diese mit der Methode `addMovie` verbinden. In dieser Methode instanziiieren Sie ein neues `movie`-Model und weisen es einer Instanz der neu erstellten View zu. Anschließend rufen Sie die `render`-Methode auf, um das Formular zum Anlegen des neuen Datensatzes anzuzeigen. Die View-Klasse liegt in der Datei `/public/js/app/views/new.js`, deren Quellcode Sie in Listing 15.21 finden.

```

define(['text!views/new.html', 'app', 'views/movie']
, function (template, app, View) {
    return Backbone.View.extend({
        tagName: 'div',
        className: 'modal hide fade newDialog',

        events: {
            'click #save': 'save'
        },

        render: function () {
            var data = {
                title: 'Filmdaten bearbeiten',
                name: this.model.get('name'),
                year: this.model.get('year')
            }

            $('#main .newDialog').remove();

```

```

        this.$el.html(_.template(template, data));
        $('#main').append(this.$el);

        $('.newDialog').modal('show');
    },

    save: function () {
        var data = {
            name: $('#name').val(),
            year: $('#year').val()
        };

        this.model.save(data);

        this.model.on("sync", _.bind(function () {
            app.collection.add(this.model);
            var view = new View({model: this.model});
            view.render();
            $('.newDialog').modal('hide');
            app.router.navigate('/');
        }, this));
    }
});
});

```

Listing 15.21 Die new-View – `public/js/app/views/new.js`

Diese View-Klasse ist um einiges umfangreicher als die, die Sie bisher kennengelernt haben. Eine Neuerung besteht in der Eigenschaft `events`. Mit dieser können Sie direkt Events, die im Template ausgelöst werden können, mit entsprechenden Callback-Funktionen verknüpfen. In diesem Fall wird die Methode `save` aufgerufen, wenn Sie auf den Link `save` klicken. Die `render`-Methode liest eventuell bestehende Werte aus dem Model aus und stellt das Template, das Sie in Listing 15.22 finden, dar. Dabei kommt der Modale Dialog von Bootstrap zum Einsatz. In der `save`-Methode lesen Sie die Werte der Formularfelder aus und schreiben diese durch einen Aufruf der `save`-Methode ins Model. `save` löst außerdem einen Ajax-Request zum Server aus, der dafür sorgt, dass die Daten gespeichert werden. Sobald der Datensatz gespeichert ist, wird ein `sync`-Event auf dem Model ausgelöst, und Sie können das Model zur Collection hinzufügen und es anzeigen lassen. Danach schließen Sie den Dialog und kehren zur Standardroute zurück.

```

<div class="modal-header">
    <button type="button" class="close" data-dismiss="modal"

```

```

        aria-hidden="true">&times;</button>
    <h3><%= title %></h3>
</div>

<div class="modal-body">
    <label for="name">Filmtitel: </label>
    <input type="text" id="name" value="<%= name %>" />
    <label for="year">Erscheinungsjahr: </label>
    <input type="text" id="year" value="<%= year %>" />
</div>

<div class="modal-footer">
    <a href="#" id="save" class="btn">save</a>
</div>

```

Listing 15.22 Das new-Template – public/js/app/views/new.html

Serverseitig müssen Sie Ihren Router, der sich in der *index.js*-Datei auf der obersten Ebene Ihrer Applikation befindet, ebenfalls um eine neue Route erweitern. Diese lautet `app.post('/movie', MDB.checkAuth, MDB.addMovie);`. Das bedeutet allerdings auch, dass Sie die Klasse *MDB* in der Datei *mdb.js* um die Methode `addMovie` erweitern müssen. Listing 15.23 zeigt Ihnen den entsprechenden Quellcode.

```

addMovie: function (req, res, next) {
    var db = MDB.getDb();
    var data = [req.body.name, parseInt(req.body.year)];

    db.run('INSERT INTO movies VALUES (?, ?)', data, function () {
        res.end('{"rowid": ' + this.lastID + '}');
    });
}

```

Listing 15.23 Die addMovie-Methode – app/mdb.js

Die `addMovie`-Methode liest die Felder `name` und `year` aus der Anfrage aus und erstellt mit diesen Werten einen neuen Eintrag in der Tabelle `movies`. Nachdem die Datenbankabfrage ausgeführt ist, wird die eingefügte ID zum Client zurückgesendet und automatisch dem Model zugewiesen. Starten Sie nun Ihre Applikation neu, können Sie mit einem Klick auf den Link **NEUEN FILM ANLEGEN** einen neuen Film in Ihrer Applikation anlegen. Das Ergebnis sehen Sie in Abbildung 15.3.

Tragen Sie im Formular Werte ein und klicken auf **SAVE**, wird der Dialog ausgeblendet und der Datensatz in die Liste eingefügt. Diese Aktionen erfolgen ohne einen Seitenreload.

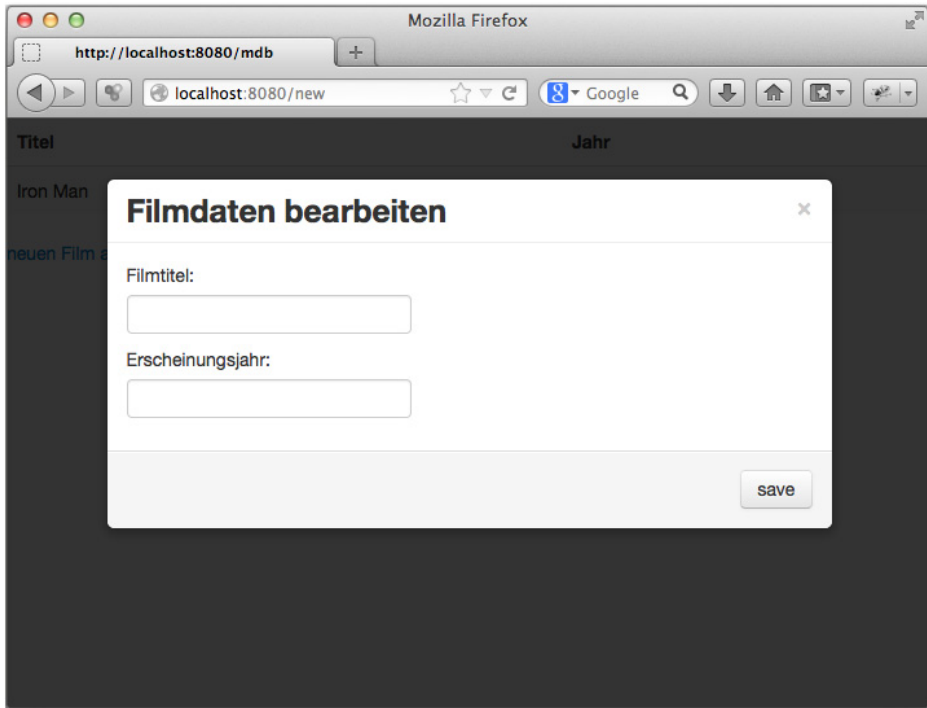


Abbildung 15.3 Neue Datensätze anlegen

15.4 Zusammenfassung

In diesem Kapitel haben Sie gesehen, wie Sie ein Backend mit Node.js mit einer Frontend-Applikation in JavaScript kombinieren können. Mit dem Beispiel der Filmdatenbank haben Sie einen soliden Grundstein für eine erweiterbare Applikation geschaffen. Die nächsten möglichen Schritte sind nun das Editieren und Löschen bestehender Datensätze oder die Erweiterung der Liste um Features wie Suchen oder Sortieren. Auch Bewertungen der Filme oder eine Detailansicht sind denkbar. Alles, was Sie hierfür tun müssen, ist, die bestehende Infrastruktur entsprechend um neue Routen im Backend und Frontend zu erweitern und die notwendigen Datenstrukturen zu schaffen. Der einzige Nachteil dieser Art von Applikation besteht darin, dass Sie die Inhalte aktiv vom Server abholen müssen und nicht benachrichtigt werden, wenn ein anderer Benutzer einen neuen Datensatz anlegt. Aber auch für diese Problemstellung existiert in der Welt von Node.js mit Websockets und Socket.IO ein Lösungsansatz. Wie Sie echtzeitfähige Applikationen in Node.js erstellen, erfahren Sie im nächsten Kapitel.

Kapitel 16

Echtzeit-Webapplikationen

Die Welt ist nicht größer als das Fenster, das du ihr öffnest.
– Sprichwort

Die ursprünglichen Webapplikationen bestanden aus einer Vielzahl einzelner HTML-Seiten, die mehr oder weniger eng aneinandergekoppelt waren. Die Nutzerinteraktion erfolgt bei solchen Applikationen über klassische Formulare zur Dateneingabe und über Links, die den Nutzer auf eine andere Seite weiterleiten. Jede dieser Aktionen hatte einen kompletten Seitenreload zur Folge. Dieses Verhalten fühlt sich wenig dynamisch und flexibel an. Eine klare Verbesserung lag daher im Auftreten der sogenannten Single-Page-Applikationen, bei denen nach einer Nutzerinteraktion nicht die gesamte Seite, sondern lediglich Teile davon neu geladen werden mussten. Auch die Kommunikationsschnittstelle des Benutzers änderte sich weg von klar sichtbaren Formularelementen hin zu intuitiven Bedienkonzepten, bei denen die Elemente der Nutzerinteraktion nicht mehr nach reinen HTML-Vorgaben, sondern nach der Bedienbarkeit der Applikation modelliert werden. Auch wenn durch dieses Konzept Webapplikationen bezogen auf ihr Verhalten ein großes Stück näher an das von Desktop-Applikationen herangerückt wurden, bleibt immer noch ein großes Problem bestehen: Wodurch erfährt ein Benutzer, wenn sich die Daten der Applikation geändert haben? Webapplikationen bauen meist auf HTTP auf. Dieses Übertragungsprotokoll basiert allerdings auf einem klassischen Client-Server-Ansatz. Das bedeutet, dass ein Client beim Server anfragt, dieser die Anfrage verarbeitet und eine Antwort sendet. Ein Rückkanal vom Server zum Client ist nicht vorgesehen. Das heißt, dass ein Client zwar nach aktuellen Daten fragen kann, der Server allerdings über keine Möglichkeit verfügt, seine Clients zu benachrichtigen, sobald neue Daten vorhanden sind. Neue Technologien wie die WebSocket-API erlauben es Ihnen allerdings bei der Entwicklung Ihrer Applikation, einen solchen Kommunikationskanal vom Server zum Client hin zu öffnen und so eine bidirektionale Kommunikation zwischen dem Client und Server zu erlauben. Neben Websockets existieren allerdings noch weitere Mechanismen, die Ihnen echtzeitfähige Webapplikation erlauben. In diesem Kapitel lernen Sie einige Möglichkeiten kennen, mit denen Sie Ihre Webapplikation so dynamisch gestalten können, dass sie vom Verhalten noch näher an klassische Desktop-Applikationen heranreichen.

16.1 Die Beispielapplikation

Als konkretes Anwendungsbeispiel für eine echtzeitfähige Webapplikation eignet sich beinahe jede Aufgabenstellung. Damit eine solche Webapplikation ihre Vorteile ausspielen kann, müssen allerdings einige Voraussetzungen erfüllt sein. Für rein statische Inhalte bringt eine echtzeitfähige Webapplikation beispielsweise keinerlei Mehrwert. Das bedeutet, dass innerhalb Ihrer Applikation Daten entstehen müssen. Wird Ihre Applikation gleichzeitig von mehreren Nutzern verwendet und entstehen durch diese Verwendung neue Informationen, ist dies eine potenzielle Aufgabenstellung, zu deren Lösung Sie eine echtzeitfähige Webapplikation verwenden können. Neue Informationen in Ihrer Applikation müssen allerdings nicht zwangsläufig von den Benutzern direkt kommen. Eine weitere Möglichkeit sind vom Server selbst gesteuerte Vorgänge. Denkbar sind hier zum Beispiel zeitgesteuerte Berechnungen, die dazu dienen, den vorhandenen Datenbestand aufzubereiten. Die dritte Variante, über die neue Informationen in Ihrer Applikation entstehen können, ist die Abfrage externer Quellen. So können Sie in regelmäßigen Abständen die aktuellen Börsenkurse oder Nachrichten von einem Webservice abrufen. Diese Daten können Sie dann innerhalb Ihrer Applikation aufbereiten und speichern und dann Ihren Benutzern zugänglich machen. Je aktueller die Informationen in Ihrer Applikation sind und je schneller Ihre Benutzer von diesen Informationen erfahren, desto wertvoller ist Ihre Applikation für diese Benutzer.

Ein klassisches Beispiel für echtzeitfähige Webapplikationen ist ein Multi-Client-Chat. Diese Aufgabenstellung erfüllt sämtliche Anforderungen, sodass eine Echtzeit-Webapplikation ihre Stärken ausspielen kann. Die Benutzer melden sich über eine Anmeldemaske an Ihrer Applikation an. Nach der Anmeldung kann sich der Benutzer in einem Chatraum mit anderen Teilnehmern unterhalten.

16.2 Setup

Der grundlegende Aufbau Ihrer Applikation besteht aus zwei Dateien und drei Verzeichnissen. Die wichtigste Datei ist wie immer der Einstieg in Ihre Applikation. Diese *index.js*-Datei liegt im Basisverzeichnis der Applikation. Neben dieser Datei benötigen Sie außerdem die Datei *package.json*, mit deren Hilfe Sie die Abhängigkeiten Ihrer Applikation vom NPM auflösen lassen. Diese Datei liegt auf der gleichen Verzeichnisebene wie die *index.js*-Datei. Das erste Verzeichnis, das Sie anlegen, heißt *app* und enthält sämtliche Dateien, die für die korrekte Funktionsweise des Backends Ihrer Applikation erforderlich sind. Die beiden anderen Verzeichnisse enthalten die Dateien für das Frontend. Zum einen ist das das *js*-Verzeichnis, in dem der gesamte JavaScript-Quellcode für das Frontend gespeichert ist. Im *style*-Verzeichnis liegt

schließlich das Stylesheet für das Frontend. Der erste Schritt der Umsetzung Ihrer Applikation besteht im Anlegen der *package.json*-Datei. Dies geschieht allerdings nicht mit dem Hintergedanken, dass Sie die Applikation später auf <http://npmjs.org> veröffentlichen möchten, sondern vielmehr aus praktischen Gründen. In dieser Datei können Sie die Abhängigkeiten Ihrer Applikation definieren und diese dann ganz einfach mit einem Kommandozeilenbefehl installieren lassen. Listing 16.1 zeigt Ihnen den Aufbau der *package.json*-Datei.

```
{
  "dependencies": {
    "express": ">=3.2.x",
    "jade": ">=0.30.x",
    "websocket": ">=1.0.x"
  }
}
```

Listing 16.1 *package.json*-Datei

Haben Sie diese Datei gespeichert, können Sie die Kommandozeile verwenden und in das Verzeichnis Ihrer Applikation wechseln. Mit dem Befehl `npm install` werden automatisch die für die Applikation erforderlichen Pakete heruntergeladen und im Verzeichnis *node_modules* installiert. Im nächsten Schritt bauen Sie dann den Einstieg in Ihre Applikation auf. Dieser besteht in erster Linie aus dem Inhalt der *index.js*-Datei. In Listing 16.2 sehen Sie den grundsätzlichen Aufbau dieser Datei. Im Verlauf dieses Kapitels werden Sie an dieser Datei immer wieder Erweiterungen vornehmen.

```
var express = require('express');

var app = express();

app.set('views', __dirname + '/app/views');
app.set('view engine', 'jade');

app.get('/', function (req, res) {
  res.render('login');
});

app.listen(8080);
```

Listing 16.2 Einstieg in die Applikation – *index.js*

Der Inhalt der *index.js*-Datei besteht momentan nur aus dem Setup von *express.js*, also einem Aufruf von *require* zur Einbindung von *express.js* und einem anschlie-

ßenden Aufruf der zurückgegebenen Funktion. Auf dem Applikations-Objekt rufen Sie zum Schluss die `listen`-Methode auf und binden Ihre Applikation auf Port 8080. Als Template-Engine kommt Jade in Ihrer Applikation zum Einsatz. Mithilfe der `use`-Methode setzen Sie die `view engine`-Eigenschaft Ihrer Applikation auf den Wert `jade` und geben durch die `views`-Eigenschaft an, wo Ihre Templates abgelegt sind. Bevor Sie Ihre Applikation zum ersten Mal testen können, benötigen Sie außerdem eine Route. Diese legen Sie über einen Aufruf der `get`-Methode Ihres Applikations-Objekts an. In der Callback-Funktion der Route wird durch die `render`-Methode ein Template gerendert und an den anfragenden Benutzer zurückgeschickt. Dieses Template legen Sie im Verzeichnis `app/views` ab und benennen es `login.jade`. Den Inhalt des Templates sehen Sie in Listing 16.3.

!!!

```
html
  head
    title Node.js Chat
  body
    form(action="login", method="post")
      div
        label(for="username") Username:
        input(type="text", name="username")
      div
        label(for="password") Password:
        input(type="password", name="password")
      div
        input(type="submit", value="Log in")
```

Listing 16.3 Anmeldeformular – `app/views/login.jade`

Starten Sie nun Ihre Applikation mit dem Kommando `node index.js`, können Sie mit Ihrem Browser über die URL `http://localhost:8080` darauf zugreifen und sehen als Ergebnis eine Webseite ähnlich wie in Abbildung 16.1.

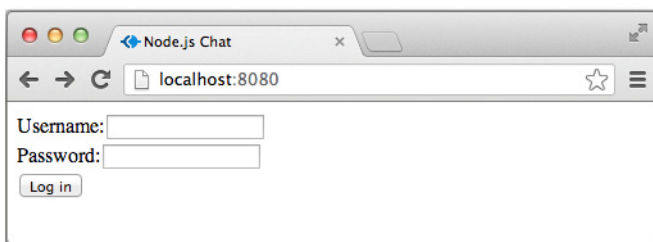


Abbildung 16.1 Anmeldemaske

Geben Sie in Ihrer Anmeldemaske einen Namen und ein Passwort ein und betätigen den LOG IN-Button, erhalten Sie die Fehlermeldung `Cannot POST /login`. Das bedeutet, dass Sie sich im nächsten Schritt darum kümmern müssen, dass sich Ihre Benutzer an Ihrer Applikation anmelden, um diese benutzen zu können. Die Fehlermeldung wird angezeigt, weil Ihre Applikation aktuell noch nicht über eine passende Route für die Anfrage verfügt. Damit der Login an Ihrer Applikation funktionieren kann, müssen Sie Ihre `index.js`-Datei anpassen. Zunächst benötigen Sie einige Middleware-Komponenten, die Ihnen die Arbeit erleichtern. Listing 16.4 zeigt Ihnen die Einbindung dieser Komponenten.

```
app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.cookieSession({secret: 'mySecret'}));
```

Listing 16.4 Middleware für den Anmeldeprozess – `index.js`

Der `bodyParser` sorgt dafür, dass Sie auf einfache Art auf die per POST übermittelten Daten, also auf Benutzername und Passwort, zugreifen können. Die `cookieSession`-Middleware stellt Ihnen eine Session zur Verfügung, mit deren Hilfe Sie den Anmeldestatus eines Benutzers für eine bestimmte Session speichern können. Damit diese Middleware ordnungsgemäß funktioniert, benötigen Sie außerdem den `cookieParser`. Mit diesen Anpassungen können Sie nun dazu übergehen, die weiteren Routen zu definieren. Insgesamt benötigen Sie für den nächsten Schritt drei Routen. Die erste dient zur unmittelbaren Anmeldung, eine weitere benötigen Sie als Ziel für angemeldete Nutzer, und über die dritte Route schließlich kann sich ein Benutzer abmelden. Listing 16.5 stellt Ihnen die Routen und die zugehörigen Callback-Funktionen vor.

```
app.post('/login', function (req, res) {
  var user = req.body.username,
      pw = req.body.password;

  if (user === 'u1' && pw === 'test') {
    req.session.user = 'u1';
  } else if (user === 'u2' && pw === 'test') {
    req.session.user = 'u2';
  }

  res.redirect('/chat');
});

app.get('/chat', checkAuth, function (req, res) {
  res.render('chat', {user: req.session.user});
});
```

```
app.get('/logout', function (req, res) {
  delete req.session.user;
  res.redirect('/');
});
```

Listing 16.5 An- und Abmeldung – index.js

Der wichtigste Teil dieser Routen ist die `post`-Methode für die `/login`-Route. Das Formular aus Listing 16.3 sendet die Anmeldeinformationen direkt an diese Route. Durch den `bodyParser` können Sie innerhalb der Callback-Funktion über das `req.body`-Objekt auf den Benutzernamen und das Passwort zugreifen. Um das Beispiel einfach und übersichtlich zu halten, werden hier lediglich zwei Benutzer vorgesehen. An dieser Stelle können Sie natürlich auch eine Datenbank einbinden und prüfen, ob der Benutzer vorhanden ist. Durch die `cookieSession`-Middleware können Sie innerhalb des Request-Objekts auf die Eigenschaft `session` zugreifen und hier den Benutzernamen speichern. Nach der Prüfung des Benutzernamens wird der Benutzer an die `/chat`-Route weitergeleitet. Einmal angemeldet, sollte sich ein Benutzer auch wieder abmelden können. Diese Aufgabe erledigt die `/logout`-Route, indem die Eigenschaft `req.session.user` gelöscht wird und der Benutzer anschließend zurück auf die Login-Seite weitergeleitet wird. Die `/chat`-Route weist eine Besonderheit auf. Bei dieser Route kommt die `checkAuth`-Middleware zum Einsatz. Sie sorgt dafür, dass nur angemeldete Benutzer diese Route verwenden dürfen. Diese Middleware können Sie, wie in Listing 16.6 zu sehen ist, ganz einfach selbst implementieren.

```
function checkAuth(req, res, next) {
  if (!req.session.user) {
    res.redirect('/');
  } else {
    next();
  }
}
```

Listing 16.6 Die `checkAuth`-Middleware – index.js

Diese Middleware prüft, ob die Eigenschaft `req.session.user` gesetzt ist. Ist dies der Fall, wird die nächste Callback-Funktion der Routing-Kette ausgeführt. Hat sich der Benutzer nicht angemeldet, wird er auf das Anmeldeformular weitergeleitet. Die Callback-Funktion der `/chat`-Route sorgt dafür, dass das `chat.jade`-Template aus dem `app/views`-Verzeichnis gerendert wird. Den Quellcode für dieses Template finden Sie in Abschnitt 16.3.2, »Die Clientseite«. Dabei übergeben Sie den Namen des aktuell angemeldeten Benutzers. Dieser Stand der Applikation bildet die Grundlage für die nun folgenden Anpassungen, an deren Ende ein voll funktionsfähiger Multi-Client-Chat steht.

16.3 Websockets

Die zentrale Komponente der Chat-Applikation ist die Kommunikationsverbindung zwischen Client und Server. Über diese kann der Benutzer Nachrichten zum Server senden und Nachrichten vom Server empfangen. Die WebSocket-API ist noch eine relativ junge Technologie, die im Zuge von HTML5 entwickelt wurde und das Problem der bidirektionalen Kommunikation in Webapplikationen lösen sollte. Im Browser steht Ihnen als Programmierer eine Schnittstelle in JavaScript zur Verfügung, über die Sie mithilfe des Websockets mit der Gegenseite kommunizieren können. Die eigentliche Verbindung, also der WebSocket, ist unabhängig von HTTP. Stattdessen beschreibt die WebSocket-Spezifikation ein eigenes Protokoll, das parallel zu HTTP existiert. Wie HTTP basiert auch das WebSocket-Protokoll auf TCP. Ein großer Vorteil der WebSocket-API liegt in ihrer Einfachheit. Die Programmierschnittstelle umfasst lediglich wenige Methoden und einige Ereignisse, die im Umgang mit Websockets auftreten können. In Listing 16.7 sehen Sie ein einfaches Beispiel, das die Verwendung von Websockets auf Clientseite verdeutlicht.

```
var socket = new WebSocket('ws://localhost:8181/');

socket.send('Hello Server');

socket.onmessage = function (msg) {
    console.log(msg.data);
}
```

Listing 16.7 Methoden der WebSocket-API im Einsatz

Wie Sie im Beispiel von Listing 16.7 sehen, müssen Sie, bevor Sie die WebSocket-API nutzen können, zuerst eine Instanz eines Websockets erzeugen. Dies geschieht über den Konstruktor, der Ihnen durch Ihren Browser zur Verfügung gestellt wird. Diese Konstruktor-Funktion akzeptiert als erstes Argument die URL, auf die Sie sich verbinden möchten. Da Websockets ein eigenes Protokoll darstellen, beginnt die URL auch nicht wie gewohnt mit `http://`, sondern mit `ws://`, was für WebSocket steht. Analog zu HTTP, bei dem es mit HTTPS eine verschlüsselte und damit sichere Variante des Protokolls gibt, existiert auch bei Websockets mit Secure Websockets eine sichere Version des Protokolls. Das Präfix für sichere Websockets lautet `wss://`. Als zweites optionales Argument können Sie dem Konstruktor eine Zeichenkette übergeben, die das sogenannte Subprotokoll spezifiziert. Das Subprotokoll kommt zum Einsatz, wenn Sie sicherstellen möchten, dass sich Client und Server Nachrichten senden, die beide verstehen. So können Sie beispielsweise für Ihre eigene Applikation ein bestimmtes Nachrichtenformat definieren. Haben Sie erst einmal eine derartige Spezifikation, können Sie den Namen Ihrer Applikation oder noch besser den Namen

Ihres Kommunikationsprotokolls als Subprotokoll angeben. Client und Server akzeptieren dann nur Verbindungen auf das entsprechende Subprotokoll.

Beim Senden von Nachrichten kommt die `send`-Methode zum Einsatz. Da das `Websocket`-Objekt, auf dem Sie die Methode aufrufen, an eine bestimmte Verbindung geknüpft ist, müssen Sie außer der Nachricht, die Sie übermitteln möchten, nichts weiter angeben.

Die `onmessage`-Eigenschaft des `Websocket`-Objekts erhält, wie Sie in Listing 16.7 sehen, eine Funktion als Wert zugewiesen. Diese Funktion dient als Callback, der aufgerufen wird, sobald das Ereignis auftritt, in diesem Fall die Ankunft einer neuen Nachricht. Als Argument erhält diese Funktion eine Repräsentation der ursprünglichen Nachricht der Gegenstelle.

Neben diesen beiden wichtigsten Mechanismen zur Kommunikation über `Websockets` existieren noch weitere Methoden und Events, die bei der Entwicklung einer Applikation nützlich sein können. Haben Sie erst einmal eine `Websocket`-Verbindung aufgebaut, muss diese auch irgendwann wieder geschlossen werden. Dies geschieht mithilfe der `close`-Methode des `Websocket`-Objekts. Diese Methode erhält keine Parameter. Wird die `Websocket`-Verbindung von der Gegenseite geschlossen, wird ein Ereignis ausgelöst. Dieses führt dazu, dass die Callback-Funktion, die Sie der Eigenschaft `onclose` zuweisen, ausgeführt wird. Das letzte Ereignis, das Sie im Rahmen der `Websocket`-API kennen sollten, ist schließlich das `open`-Event. Der Eigenschaft `onopen` des `Websocket`-Objekts können Sie, wie schon bei `onclose`, eine Callback-Funktion zuweisen, die ausgeführt wird, sobald die `Websocket`-Verbindung hergestellt ist.

16.3.1 Die Serverseite

Mit diesem Wissen können Sie nun dazu übergehen, Ihre Applikation um eine `Websocket`-Verbindung zu erweitern. In Listing 16.8 sehen Sie den ersten Schritt der erforderlichen Anpassungen. Sie müssen diesen Quellcode lediglich in Ihre `index.js`-Datei einfügen.

```
var WSS = require('websocket').server,
    http = require('http');

var server = http.createServer();
server.listen(8181)

var wss = new WSS({
  httpServer: server,
  autoAcceptConnections: false
});
```

```

var connections = [];

wss.on('request', function (request) {
    var connection = request.accept('chat', request.origin);

    connections.push(connection);

    connection.on('message', function(message) {
        for (var i = 0; i < connections.length; i++) {
            if (connections[i] && connections[i].send) {
                connections[i].send(message.utf8Data);
            }
        }
    });
});
});

```

Listing 16.8 WebSocket-Backend – index.js

Bei der Erstellung der *package.json*-Datei für Ihre Applikation haben Sie bereits die Grundlage für die Erweiterung um Websockets geschaffen, indem Sie das *websocket*-Paket als Abhängigkeit definiert haben. Mit dem `npm install`-Kommando haben Sie anschließend die erforderlichen Pakete installiert. Das *websocket*-Paket bietet Ihnen Klassen, mit deren Hilfe Sie sowohl einen WebSocket-Server als auch einen -Client umsetzen können. Für dieses Beispiel benötigen Sie jedoch nur die Serverkomponente des Pakets. Über die Eigenschaft `server` des *websocket*-Pakets erhalten Sie Zugriff auf die Konstruktor-Funktion des Servers. Dieses Paket nutzt allerdings einige Features des *http*-Moduls von Node.js. Aus diesem Grund müssen Sie in Listing 16.8 auch dafür sorgen, dass Sie eine lauffähige Instanz eines unabhängigen HTTP-Servers haben. Diesen erstellen Sie am besten über die `createServer`-Methode des *http*-Moduls. In diesem Fall müssen Sie keine weitere Logik im Webserver implementieren. Es reicht aus, wenn Sie ihn über einen Aufruf der `listen`-Methode auf einen TCP-Port binden. Bei der Instanziierung des WebSocket-Servers übergeben Sie dann die Referenz auf den HTTP-Server als Teil eines Konfigurationsobjekts an die Konstruktor-Funktion. Der Name der Eigenschaft für diese Referenz lautet `httpServer`. Mit diesem Quellcode haben Sie nun die Voraussetzungen erfüllt, die zur Implementierung Ihres Chat-Servers erforderlich sind. Was nun folgt, ist die eigentliche Logik des Servers.

Wie Sie bereits in Listing 16.7 sehen konnten, ist die WebSocket-API grundsätzlich eventbasiert aufgebaut. Das bedeutet, dass jede Aktion auf dem WebSocket zu einem Ereignis führt, an das Sie Callback-Funktionen binden können. Dieses Konzept setzt das *websocket*-Paket für Node.js konsequent fort. Den Einstieg in die WebSocket-Kommunikation bildet das Ereignis `request`. Dieses wird ausgelöst, sobald ein Benutzer versucht, eine WebSocket-Verbindung mit dem Server herzustellen. Das *websocket*-Paket bietet Ihnen mehrere Möglichkeiten, wie Sie mit eingehenden

Verbindungen verfahren können. In Listing 16.8 sehen Sie, dass Sie der Konstruktor-Funktion des WebSocket-Servers die Option `autoAcceptConnections` übergeben können. Setzen Sie den Wert dieser Eigenschaft auf den Wert `true`, bedeutet das, dass der Server sämtliche eingehenden Verbindungen akzeptiert, eine WebSocket-Verbindung aufbaut und diese offen hält. Alternativ können Sie den Wert der Eigenschaft `autoAcceptConnections`, wie Sie es im Beispiel sehen, auf `false` setzen. Das bedeutet, dass Sie die eingehenden Verbindungen selbst akzeptieren beziehungsweise ablehnen müssen. Gedacht ist dieses Feature vor allem dazu, dass Sie innerhalb Ihrer Applikation eine Möglichkeit erhalten, mit der Sie steuern können, von woher sich Ihre Benutzer mit dem Server verbinden können. Das Objekt, das Sie in der Callback-Funktion des `request`-Events als Argument erhalten und das die Anfrage des Clients repräsentiert, weist eine Eigenschaft `origin` auf. Diese trägt als Wert die URL der Webseite, von der aus die Verbindung aufgebaut wird. Im Falle des Chatbeispiels ist dies der Wert `http://localhost:8080`. Erhält Ihr Server nun eine Anfrage eines Clients, können Sie anhand der `origin`-Eigenschaft entscheiden, ob Sie die Verbindung akzeptieren möchten oder ob Sie die Anfrage einfach abweisen. In Listing 16.8 werden alle Anfragen mit der `accept`-Methode des `request`-Objekts akzeptiert, ohne dass Sie sie auf die Herkunft der Anfrage prüfen. Dieser Methode geben Sie als Werte zum einen das Subprotokoll mit, das akzeptiert werden soll, und zum anderen eine Referenz auf `request.origin`. Das bedeutet in diesem konkreten Fall, dass Sie die Verbindung von `http://localhost:8080` auf das `chat`-Subprotokoll akzeptieren. Als Rückgabewert dieses Methodenaufrufs erhalten Sie eine Referenz auf die Verbindung. Alternativ haben Sie außerdem die Möglichkeit, bestimmte Verbindungen nicht zu akzeptieren. Das Abweisen einer solchen Verbindung erreichen Sie mit der `reject`-Methode. Im Firefox führt dies zum Beispiel zur Meldung `Firefox can't establish a connection to the server at ws://localhost:8181/..`

Das Verbindungsobjekt, das Sie als Rückgabewert vom Aufruf der `accept`-Methode erhalten, können Sie nun verwenden, um mit dem Client zu kommunizieren. Hier können Sie, wie Sie es auch schon bei der clientseitigen Umsetzung in Listing 16.7 sehen konnten, auf verschiedene Methoden und Ereignisse zurückgreifen. Die wichtigsten Schnittstellen sind hier die `send`-Methode des Verbindungsobjekts und das `message`-Event. Über das `connection`-Objekt haben sie jeweils nur Zugriff auf die aktuelle Verbindung. Im Falle der Chat-Applikation reicht es nicht aus, auf diesen einen Kanal zuzugreifen. Deswegen müssen Sie sich ein Konstrukt schaffen, das Ihnen erlaubt, sämtliche aktiven Kanäle zu verwenden. Für diese Struktur eignet sich beispielsweise ein Array. In Listing 16.8 ist dies das Array mit dem Namen `connections`. Bei jedem Verbindungsaufbau wird die aktuelle Verbindung mit der `push`-Methode in dieses Array eingefügt. Da es sich dabei um eine globale Variable handelt, haben Sie von überall in Ihrer Applikation Zugriff auf sämtliche Verbindungen.

Sendet ein Client eine Nachricht an den Server, wird das `message`-Event ausgelöst. Mit der `on`-Methode des `connection`-Objekts können Sie eine Callback-Funktion auf die-

ses Event binden. Innerhalb dieser Funktion müssen Sie dafür sorgen, dass die eingehenden Nachrichten an alle angeschlossenen Clients verteilt werden. Dies erreichen Sie am besten, indem Sie über alle aktiven Sockets iterieren und mithilfe der `send`-Methode die Nachricht weiterleiten. Nun, da die serverseitige Umsetzung abgeschlossen ist, können Sie sich im nächsten Schritt dem Client widmen.

16.3.2 Die Clientseite

Das Frontend Ihrer Chat-Applikation besteht im Wesentlichen aus einer Datei, dem Template *chat.jade*. In Listing 16.9 sehen Sie den Aufbau dieser Datei.

```
html
  head
    script(src="/js/jquery-1.9.1.min.js")

  body
    div#msgs(style="height:400; width:400; overflow: scroll;")

    form#chatForm
      input#name(type="text")
      input#msg(type="text")
      button#sendBtn Send

    script
      var socket = new WebSocket('ws://localhost:8181/', 'chat');

      $('#sendBtn').on('click', function (e) {
        e.preventDefault();
        var name = $('#name').val(),
            msg = $('#msg').val();

        socket.send('{"name": "' + name + '", "msg": "' + msg + '"}');
        $('#msg').val('');
      });

      socket.onmessage = function (msg) {
        var data = JSON.parse(msg.data);

        var msg = $('<div>' + data.name + ': ' + data.msg + '</div>');

        $('#msgs').append(msg);
      };
```

Listing 16.9 Das *chat.jade*-Template – *app/views/chat.jade*

Damit Sie sich Schreibarbeit sparen, binden Sie im ersten Schritt die JavaScript-Bibliothek jQuery ein. Dafür müssen Sie zuerst die Quelldatei von <http://www.jquery.com> herunterladen und im Verzeichnis *js* in Ihrer Applikation speichern. Im nächsten Schritt müssen Sie dafür sorgen, dass die Datei von Node.js ausgeliefert wird. Am einfachsten erreichen Sie dies, indem Sie die *static*-Middleware von *express.js* verwenden. Die entsprechende Anpassung Ihrer *index.js*-Datei finden Sie in Listing 16.10.

```
app.use('/js', express.static(__dirname + '/js'));
```

Listing 16.10 static-Middleware – index.js

Mit der Erweiterung aus Listing 16.10 haben Sie bereits den ersten Teil Ihres Templates lauffähig gemacht. Der Quellcode der jQuery-Bibliotheken wird durch das Jade-Template eingebunden und korrekt durch die *express.js*-Middleware ausgeliefert. Damit steht der Rahmen des Templates. Das nächste Problem in der Umsetzung liegt jetzt darin, dass ein Benutzer der Seite noch keine Inhalte sehen kann. Aus diesem Grund sollten Sie sich im nächsten Schritt nun darum kümmern, eine Schnittstelle für Ihre Benutzer zu schaffen. Diese besteht aus zwei Teilen. Der erste Teil besteht aus einem Container, in dem die erhaltenen Nachrichten angezeigt werden. Diesen Container erstellen Sie im einfachsten Fall mit einem *div*-Element. Wichtig ist, dass Sie diesem Element eine eindeutige ID zuweisen, damit Sie später einfacher darauf zugreifen können. Der zweite Teil der Benutzerschnittstelle besteht aus einem Formular, über das ein Benutzer seine Nachricht verfassen und sie anschließend versenden kann. Das Formular besteht aus insgesamt drei Elementen. Mit zwei Eingabefeldern kann der Benutzer seinen Namen festlegen und Nachrichten formulieren. Über einen Button können diese Informationen dann zum Server gesendet werden. Nun müssen Sie lediglich noch dafür sorgen, dass die Nachrichten zum Server gelangen und vom Server empfangene Nachrichten dem Benutzer auch angezeigt werden. Diese Aufgabe übernimmt eine Reihe von JavaScript-Funktionen. Wie Sie in Listing 16.7 schon gesehen haben, können Sie über die Konstruktor-Funktion *WebSocket* eine neue Verbindung aufbauen. Dieser Funktion übergeben Sie zum einen die Adresse und zum anderen das Subprotokoll, in diesem Fall den Wert *chat*, als Werte. Die daraus entstehende Instanz eines Websockets können Sie wie auch schon beim Server dazu verwenden, Nachrichten zu versenden und zu empfangen.

Für den Versand einer Nachricht registrieren Sie nun eine Callback-Funktion auf das *click*-Event des Buttons. Innerhalb dieser Funktion sammeln Sie die Werte der beiden Eingabefelder und versenden sie als Eigenschaften in einem Objekt über die *send*-Methode des *WebSocket*-Objekts. Im Anschluss daran leeren Sie das Eingabefeld für die Nachricht und bereiten die Eingabe so für die nächste Nachricht vor. Über die *onmessage*-Eigenschaft des *WebSocket*-Objekts können Sie dann mit den eingehenden Nachrichten umgehen. Hierfür müssen Sie lediglich eine Callback-Funktion definieren. Sie erhält als Argument eine Zeichenkette, die die eingehende Nachricht

repräsentiert. Sie können aus dieser Zeichenkette mit einem Aufruf von `JSON.parse` ein Objekt erstellen, mit dem Sie weiterarbeiten können. Für jede eingegangene Nachricht generieren Sie einen eigenen `div`-Container, den Sie an den Inhalt des Anzeigecontainers anfügen.

Sobald Sie diese Anpassungen am Quellcode von Client und Server durchgeführt haben, können Sie Ihre Applikation mit dem Kommando `node index.js` starten. Sie können Ihre Applikation testen, indem Sie über Ihren Browser auf die Adresse `http://localhost:8080` zugreifen. Im ersten Schritt müssen Sie sich anmelden, anschließend werden Sie an den eigentlichen Chat weitergeleitet. Mit diesem einen Client können Sie Nachrichten an sich selbst versenden. Die Auswirkungen der Realtime Webapplikation sehen Sie allerdings erst, wenn Sie sich mit einem zweiten Browser-Tab oder einem zweiten Browser mit dem Server verbinden und beginnen, Nachrichten zu versenden. Dann sehen Sie, dass in beiden Browserfenstern der Anzeigecontainer aktualisiert wurde, egal, von welchem Browser Sie die Nachricht versendet haben. Das Ergebnis sehen Sie in Abbildung 16.2.

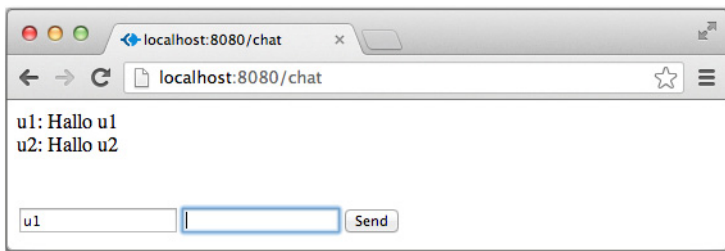


Abbildung 16.2 Chat zwischen zwei Benutzern

In den folgenden Abschnitten gehen Sie nun dazu über, die bestehende Funktionalität Ihrer Applikation zu erweitern.

16.3.3 Userliste

Bis jetzt können Sie sich an Ihrer Chat-Applikation anmelden und in einem globalen Chatraum Nachrichten an andere Benutzer senden. Die erste Erweiterung Ihrer Applikation besteht darin, dass Sie eine Liste der aktiven Benutzer einfügen, damit jeder Benutzer weiß, mit wem er sich unterhalten kann. Mit dieser Erweiterung entsteht auch die Notwendigkeit, verschiedene Arten von Nachrichten zu etablieren. Neben den reinen Textnachrichten kommen nun auch Statusnachrichten hinzu. Sobald sich ein Benutzer am Chat-Server anmeldet, muss eine solche Statusnachricht versendet werden, damit einerseits der Server den Namen des neuen Benutzers erfährt und andererseits auch jeder andere Benutzer des Chats von der Existenz des neuen Teilnehmers Kenntnis erhält. In Listing 16.11 sehen Sie das angepasste `chat.jade`-Template für das Frontend.

```

html
  head
    script(src="/js/jquery-1.9.1.min.js")

  body
    div#msgs(style="height:400px; width:400px; overflow: scroll;
      float:left;")
    div#users(style="height:400px; width:100px; overflow: scroll;")

    form#chatForm
      label(for="msg") #{user}&nbsp;
      input#msg(type="text")
      button#sendBtn Send

    script
      var socket = new WebSocket('ws://localhost:8181/', 'chat');

      socket.onopen = function () {
        socket.send('{"type": "join", "name": "#{user}"}');
      }

      $('#sendBtn').on('click', function (e) {
        e.preventDefault();
        var name = '#{user}',
            msg = $('#msg').val();

        socket.send('{"type": "msg", "msg": "' + msg + '"}');
        $('#msg').val('');
      });

      socket.onmessage = function (msg) {
        var data = JSON.parse(msg.data);

        switch (data.type) {
          case 'msg':
            var msg = $('<div>' + data.name + ': ' + data.msg +
              '</div>');
            $('#msgs').append(msg);
            break;
          case 'join':
            $('#users').empty();
            for (var i = 0; i < data.names.length; i++) {
              var user = $('<div>' + data.names[i] + '</div>');
              $('#users').append(user);
            }

```

```

    }
    break;
  }
};

```

Listing 16.11 Frontend mit Unterstützung von Nachrichtentypen – `app/views/chat.jade`

An der Benutzerschnittstelle führen Sie lediglich zwei Änderungen durch. Die erste Anpassung besteht darin, dass Sie einen Container definieren, der die Benutzerliste enthält. Dieser Container trägt die ID `users` und liegt direkt neben dem Container zur Anzeige der Nachrichten. Außerdem entfällt das Eingabefeld für den Namen. Statt eines eingegebenen Namens sollte der AnmeldeName des Benutzers verwendet werden. Der Name des Benutzers wird als `label` für das Eingabefeld der Nachricht verwendet. Die wirklichen Änderungen finden Sie innerhalb des `script`-Tags. Der Verbindungsaufbau bleibt unverändert. Danach nutzen Sie allerdings die Eigenschaft `onopen`, um in einer Callback-Funktion eine Nachricht vom Typ `join` und unter dem Namen des Benutzers zu senden. Beim Versand der Nachrichten fügen Sie ebenfalls eine `type`-Eigenschaft ein. Des Weiteren sollten Sie in der Callback-Funktion, die auf dem `click`-Event des Buttons registriert ist, die Referenz auf das Eingabefeld des Benutzernamens entfernen.

Beim Empfang von Nachrichten müssen Sie nun auch mit verschiedenen Typen von Nachrichten umgehen können. Aus diesem Grund müssen Sie die Callback-Funktion, die Sie der `onmessage`-Eigenschaft der Verbindung zuweisen, entsprechend anpassen. Die Zeichenkette, die Sie vom Server erhalten, liegt im JSON-Format vor und weist ein Feld mit dem Namen `type` auf. Aktuell müssen Sie zwei Typen von Nachrichten unterscheiden. Das ist einmal der Typ `msg`. Dieser wird verwendet, wenn Sie neue Nachrichten erhalten. Hier bleibt die ursprüngliche Logik vorhanden. Daneben kommt ein neuer Typ `join` hinzu. Bei Nachrichten dieses Typs erhalten Sie neben dem Typ ein Array mit allen angemeldeten Benutzern. Sie müssen dann lediglich noch dafür sorgen, dass der Inhalt der Benutzerliste entfernt wird und die geänderte Benutzerliste, in `div`-Elementen gekapselt, eingefügt wird. Im nächsten Schritt müssen Sie die Serverseite anpassen, sodass auch sie mit den neuen Nachrichtentypen umgehen kann. Listing 16.12 enthält den angepassten Quellcode des Chat-Servers.

```

var connections = {};

wss.on('request', function (request) {
  var connection = request.accept('chat', request.origin);

  connection.on('message', function(message) {

    var name = '';

```

```

    for (var key in connections) {
        if (connection === connections[key]) {
            name = key;
        }
    }

    var data = JSON.parse(message.utf8Data);

    switch (data.type) {
        case 'join':
            connections[data.name] = connection;
            var msg = '{"type": "join", "names": [' +
                Object.keys(connections).join(',') + ']}';
            break;
        case 'msg':
            var msg = '{"type": "msg", "name": "' + name + '", "msg":"' +
                data.msg + '"}';
            break;
    }

    for (var key in connections) {
        if (connections[key] && connections[key].send) {
            connections[key].send(msg);
        }
    }
});
});

```

Listing 16.12 Backend mit Unterstützung von Nachrichtentypen – index.js

Serverseitig beschränken sich die Änderungen auf den Bereich der Websockets. Der `express.js`-Teil der Applikation bleibt unverändert. Eine wichtige Änderung ist, dass statt eines Arrays zum Vorhalten der Referenzen auf die einzelnen Socket-Verbindungen nun ein Objekt zu diesem Zweck eingesetzt wird. Jede Socket-Verbindung wird mit dem Namen des jeweils angemeldeten Benutzers in diesem Objekt gespeichert. Durch diesen Mechanismus können Sie später dann auch herausfinden, von welchem Benutzer eine Nachricht versendet wurde, sodass die Notwendigkeit der Namensübermittlung entfällt. Innerhalb der Callback-Funktion für das `message`-Event können Sie nun den Namen ermitteln. Sie erhalten den Namen, indem Sie die aktuelle Verbindung mit allen vorgehaltenen Referenzen vergleichen. Liegt eine Übereinstimmung vor, können Sie den Schlüssel, unter dem die Verbindung gespeichert ist, als Namen des Versenders verwenden. Anschließend müssen Sie die unterschiedlichen Nachrichtentypen behandeln. Die eingehenden Daten liegen als JSON-Daten vor und müssen vor ihrer Verwendung in ein Objekt gewandelt werden. Danach können Sie

auf die `type`-Eigenschaft der Nachricht zugreifen und entsprechend entscheiden, wie Sie mit den Daten verfahren. Im Falle einer Nachricht vom Typ `msg`, einer Textnachricht also, müssen Sie die Nachricht nur entsprechend formatieren. Bei Nachrichten vom Typ `join`, also von dem Beitritt eines neuen Nutzers in den Chat, sorgen Sie zuerst dafür, dass die Verbindung unter dem Namen des Nutzers in das `connections`-Objekt eingefügt wird. Zusätzlich bereiten Sie hier ebenfalls die Nachricht an alle Chat-Teilnehmer mit dem Typ `join` vor und fügen die Namen sämtlicher Benutzer als JSON-Array ein. Nach dieser Fallunterscheidung sorgt eine Schleife dafür, dass die vorbereitete Nachricht an alle Teilnehmer gesendet wird. Mit dieser Anpassung des Quellcodes verfügen Sie nun über eine echtzeitfähige Liste der aktuell angemeldeten Benutzer Ihrer Chat-Applikation. Das Resultat sehen Sie in Abbildung 16.3.

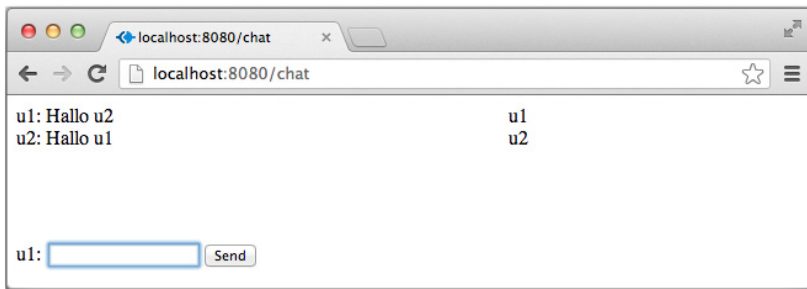


Abbildung 16.3 Chat-Applikation mit Teilnehmerliste

Die Anpassungen, die Sie an der Applikation durchgeführt haben, ermöglichen es Ihnen nun, ein weiteres Feature umzusetzen. Ein Benutzer soll sich, nachdem er die Applikation verwendet hat, auch wieder ordnungsgemäß abmelden können.

16.3.4 Logout

Mit allen Vorbereitungen, die Sie bisher getroffen haben, ist die Umsetzung des Logout-Prozesses nun kein Problem mehr. Die einzige Erweiterung im Frontend besteht aus dem Einfügen des Logout-Links. Diesen platzieren Sie am besten über dem Anzeige-Container. Listing 16.13 zeigt Ihnen die erforderliche Zeile im Template.

```
div(style="width:500px; text-align:right;")
  a(href="/logout") Logout
```

Listing 16.13 Logout-Link – `app/views/chat.jade`

Klickt ein angemeldeter Benutzer auf diesen Link, wird er auf die `/logout`-Route Ihrer Applikation weitergeleitet. Diese Route haben Sie bereits im Rahmen der Basisapplikation erstellt. Momentan sorgt sie dafür, dass der Benutzername aus der Session gelöscht und der Browser im Anschluss auf die Startseite weitergeleitet wird. Diese

Route müssen Sie nun noch etwas erweitern. In Listing 16.14 sehen Sie den entsprechenden Codeblock.

```
app.get('/logout', function (req, res) {
  connections[req.session.user].close();
  delete connections[req.session.user];

  delete req.session.user;

  var msg = '{"type": "join", "names": [' +
    Object.keys(connections).join(',') + ']}';

  for (var key in connections) {
    if (connections[key] && connections[key].send) {
      connections[key].send(msg);
    }
  }
  res.redirect('/');
});
```

Listing 16.14 Die /logout-Route – index.js

Das Löschen der Session-Daten und die Weiterleitung bleiben wie bisher bestehen. Was hinzukommt, ist die Logik, die dafür sorgt, dass die Websocket-Verbindung getrennt wird. Dies erreichen Sie über einen Aufruf der `close`-Methode. Außerdem sollten Sie die getrennte Verbindung dann aus dem `connections`-Objekt entfernen. Dies erspart Ihnen die Iteration über getrennte Verbindungen beim Versand von Nachrichten. Außerdem müssen Sie eine Nachricht vom Typ `join` an jeden Client absetzen, damit die Teilnehmerliste aktualisiert wird und der abgemeldete Benutzer dort nicht mehr erscheint. Die bis jetzt entwickelte Chat-Applikation kann Ihnen nun für eigene Weiterentwicklungen als Basis dienen. So können Sie beispielsweise das Layout der Applikation verbessern oder weitere Features wie verschiedene Chat-räume, private Konversationen zwischen einzelnen Teilnehmern oder Ähnliches implementieren. Den Möglichkeiten sind hier nur wenige Grenzen gesetzt. In den folgenden Abschnitten werden Sie sich auch noch weiterhin mit der Chat-Applikation beschäftigen. Sie werden jedoch keine weiteren Features hinzufügen, sondern stattdessen den aktuell verwendeten Kommunikationsmechanismus austauschen. Statt Websockets kommt nun eine andere Technologie, Socket.IO, zum Einsatz.

16.4 Socket.IO

Der größte Nachteil in Ihrer Chat-Applikation besteht darin, dass sie nicht mit allen Browsern benutzt werden kann. Die Websocket-API ist noch relativ neu. Das bedeu-

tet, dass ältere Browser sie nicht unterstützen, was wiederum zu einer Fehlermeldung beim Aufbau der Verbindung führt, da die Konstruktor-Funktion `WebSocket` nicht bekannt ist. Gerade ältere Versionen des Internet Explorers und einige mobile Browser unterstützen das WebSocket-Protokoll noch nicht. Abhilfe schafft hier die Bibliothek Socket.IO. Das Ziel dieser Bibliothek ist es, browserübergreifend Realtime-Webapplikationen zu ermöglichen, die nicht an ein bestimmtes Kommunikationsprotokoll gebunden sind. Zu diesem Zweck unterstützt Socket.IO eine ganze Reihe von Kommunikationsmöglichkeiten, mit denen eine bidirektionale Verbindung zwischen Client und Server möglich wird. Socket.IO besteht aus zwei Komponenten. Zum einen existiert eine Serverkomponente und zum anderen eine Datei, die der Client laden muss, um die Bibliothek benutzen zu können. Socket.IO ist komplett in JavaScript umgesetzt. Das heißt, dass Sie keine zusätzlichen Plugins in Ihrem Browser benötigen. Serverseitig liegt Socket.IO als NPM-Paket vor. Einmal installiert und eingebunden, kümmert sich Socket.IO selbst um die Aushandlung der Art der Verbindung. Socket.IO arbeitet hier mit einer Reihe von Technologien, die nacheinander getestet werden. So wird beispielsweise geprüft, ob der Browser Websockets unterstützt. Ist dies nicht der Fall, wird alternativ versucht, die Verbindung über JSONP Polling, Flash Sockets oder AJAX Long Polling aufzubauen. In den meisten Fällen wird zumindest eine dieser Technologien von einem Browser unterstützt, und Sie können dann mit dieser Verbindung Ihre Realtime Webapplikation umsetzen. Im folgenden Schritt bereiten Sie zunächst Ihre Chat-Applikation auf den Einsatz von Socket.IO vor.

16.4.1 Installation und Einbindung

Wie bereits erwähnt, liegt Socket.IO serverseitig als NPM-Paket vor. Das bedeutet, dass Sie diese Bibliothek ganz einfach mit dem Kommando `npm install socket.io` installieren können. Socket.IO ist eine recht umfangreiche Bibliothek mit einigen Abhängigkeiten, die der NPM bei der Installation automatisch auflöst. Um Socket.IO korrekt in Ihre Applikation aufzunehmen, sollten Sie wiederum Ihre *package.json*-Datei anpassen. In Listing 16.15 sehen Sie das Resultat.

```
{
  "dependencies": {
    "express": ">=3.2.x",
    "jade": ">=0.30.x",
    "socket.io": ">=0.9.x"
  }
}
```

Listing 16.15 *package.json* mit Socket.IO

In dieser Version der *package.json*-Datei ersetzen Sie das *websocket*-Paket, das Sie zuvor für die Echtzeitkommunikation mit dem Client verwendet haben, durch die

Zeichenkette `socket.io`. Mit dem Kommando `npm update` wird die Bibliothek im `node_modules`-Verzeichnis Ihrer Applikation installiert. Das nun nicht weiter benötigte `websocket`-Paket können Sie mit dem Befehl `npm uninstall websocket` wieder entfernen. Die Anpassungen Ihrer Applikation beschränken sich wie schon bei der Integration der Websockets vor allem auf den Teil außerhalb der `express.js`-Applikation. Nach erfolgter Installation können Sie sich an die Einbindung der Bibliothek auf Serverseite machen. Dazu müssen Sie die Zeile `app.listen(8080);` in der `index.js`-Datei durch den Quellcode aus Listing 16.16 ersetzen. Außerdem sollten Sie die bisherigen WebSocket-Komponenten entfernen.

```
var http = require('http');
var server = http.createServer(app);
server.listen(8080);
var io = require('socket.io').listen(server);
```

Listing 16.16 Serverseitige Integration von Socket.IO – `index.js`

Mit dieser Zeile schaffen Sie nicht nur die Grundlage für eine funktionierende Serverimplementierung, sondern bauen damit auch die Basis für die Umsetzung der Clientseite. Socket.IO stellt Ihnen nicht nur den Quellcode der Bibliothek für den Server zur Verfügung, sondern sorgt gleichzeitig auch für die Auslieferung der Quelldatei für den Client. Dieser Mechanismus funktioniert automatisch, sodass Sie lediglich eine Zeile in Ihr Template einfügen müssen, um Socket.IO dort zu verwenden. Die entsprechende Erweiterung des Templates finden Sie in Listing 16.17.

```
script(src="http://localhost:8080/socket.io/socket.io.js")
```

Listing 16.17 Clientseitige Integration von Socket.IO

Mit diesen beiden Änderungen haben Sie bereits alle Voraussetzungen für die weiteren Anpassungen erfüllt. In den nächsten Abschnitten sehen Sie, welche Teile Ihrer Applikation Sie modifizieren müssen, um von Websockets auf Socket.IO umzusteigen.

16.4.2 Socket.IO-API

Die Programmierschnittstelle von Socket.IO ähnelt der der WebSocket-API. In einigen Feinheiten unterscheiden sich beide Technologien jedoch. Die Anpassungen der Frontends beschränken sich auf den JavaScript-Teil des Templates, den Sie in Listing 16.18 sehen können.

```
script
  var socket = io.connect('http://localhost:8080');

  socket.on('connect', function () {
```

```

        socket.emit("join", '{"name":"#{user}"}');
    });

$('#sendBtn').on('click', function (e) {
    e.preventDefault();
    var name = '#{user}',
        msg = $('#msg').val();

    socket.emit("msg", '{"msg": "' + msg + '"}');
    $('#msg').val('');
});

socket.on('msg', function (msg) {
    var data = JSON.parse(msg);
    var msg = $('<div>' + data.name + ': ' + data.msg + '</div>');
    $('#msgs').append(msg);
});

socket.on('join', function (msg) {
    var data = JSON.parse(msg);
    $('#users').empty();
    for (var i = 0; i < data.names.length; i++) {
        var user = $('<div>' + data.names[i] + '</div>');
        $('#users').append(user);
    }
});

```

Listing 16.18 Anpassung des Frontends an Socket.IO – `app/views/chat.jade`

Die Verbindung zum Server bauen Sie nun nicht mehr über die Konstruktor-Funktion `WebSocket`, sondern über die `connect`-Methode des `io`-Objekts auf. Diese liefert Ihnen eine Instanz der Verbindung zum Socket.IO-Server, mit der Sie im weiteren Verlauf arbeiten können. Die Benachrichtigung des Servers über einen neuen Benutzer implementieren Sie innerhalb der Callback-Funktion des `connect`-Events, das ausgelöst wird, sobald die Verbindung hergestellt ist. Statt der `send`-Methode der Websockets kommt bei Socket.IO die `emit`-Methode des Verbindungsobjekts zum Einsatz. Socket.IO unterstützt bereits nativ verschiedene Nachrichtentypen, die Sie bei den Websockets über eine Eigenschaft der Nachricht selbst emuliert haben. So können Sie in diesem Fall direkt eine Nachricht vom Typ `join` versenden. Die Callback-Funktion, die Sie auf das `click`-Event des Buttons binden, bleibt unverändert bis auf die Verwendung der `emit`-Methode und die Anpassung des Nachrichtentyps auf `msg`. Auch bei der Behandlung eingehender Nachrichten können Sie nun auf die verschiedenen Nachrichtentypen von Socket.IO zurückgreifen und müssen nicht die

beiden verschiedenen Typen in einer Callback-Funktion behandeln. Eingehende Nachrichten sorgen dafür, dass ein `msg`-Event ausgelöst wird. An dieses Event binden Sie eine Callback-Funktion, die dafür sorgt, dass die übermittelten Daten in den Anzeige-Container eingefügt werden. Auch die Logik bei eingehenden Nachrichten des Typs `join` bleibt nahezu unverändert. In der Callback-Funktion des `join`-Events leeren Sie zunächst den Container, der die Benutzernamen enthält, und fügen anschließend die neue Liste der Benutzer wieder ein.

Die Umbauten auf Serverseite gestalten sich ähnlich einfach wie schon auf Client-seite. Listing 16.19 zeigt Ihnen die erforderlichen Maßnahmen.

```
var connections = {};
```

```
function getName (connections, socket) {
  var name;

  for (var key in connections) {
    if (socket === connections[key]) {
      name = key;
    }
  }

  return name;
}
```

```
io.sockets.on('connection', function (socket) {

  socket.on('msg', function(message) {
    var data = JSON.parse(message),
        name = getName(connections, socket);

    var msg = '{"name": "' + name + '", "msg":"' + data.msg + '"}';

    socket.emit('msg', msg);
    socket.broadcast.emit('msg', msg);
  });

  socket.on('join', function(message) {
    var data = JSON.parse(message),
        name = getName(connections, socket);

    connections[data.name] = socket;

    var msg = '{"names": [' + Object.keys(connections).join(',') +
```

```

        '"]}}';

        socket.emit('join', msg);
        socket.broadcast.emit('join', msg);
    });

});

```

Listing 16.19 Anpassung des Backends an Socket.IO – index.js

Die Hilfsfunktion `getName` dient dazu, den Namen des Benutzers herauszufinden, der für den Versand einer bestimmten Nachricht verantwortlich ist. Im Gegensatz zum `websocket`-Paket heißt das Event, das ausgelöst wird, sobald ein Client eine Verbindung zum Server aufbaut, nun nicht mehr `request`, sondern `connection`. Innerhalb der Callback-Funktion, die Sie auf dieses Ereignis binden, behandeln Sie dann die beiden Nachrichtentypen `msg` und `join`. In der Callback-Funktion des `msg`-Events formatieren Sie die Nachricht entsprechend und senden sie dann mit der `emit`-Methode an sämtliche angemeldeten Benutzer. Hier kommen zwei verschiedene Arten dieser Methode zum Einsatz: Die Methode `socket.emit` sorgt dafür, dass der Client, zu dem die aktuelle Verbindung besteht, die Nachricht erhält. Die `socket.broadcast.emit`-Methode leitet die Nachricht an alle übrigen Clients weiter. Ähnlich verfahren Sie beim `join`-Event, mit dem Unterschied, dass Sie hier wie auch schon bei den Websockets eine Datenstruktur aufbauen, die sämtliche Referenzen auf die bestehenden Verbindungen hält. Schließlich müssen Sie nur noch die `/logout`-Route Ihrer Applikation anpassen. Listing 16.20 enthält den entsprechenden Quellcode hierfür.

```

app.get('/logout', function (req, res) {
    var names = Object.keys(connections);
    names.splice(names.indexOf(req.session.user), 1);

    var msg = '{"names": [' + names.join(',') + ']}';

    connections[req.session.user].broadcast.emit('join', msg);

    connections[req.session.user].disconnect();
    delete connections[req.session.user];

    delete req.session.user;

    res.redirect('/');
});

```

Listing 16.20 Die `/logout`-Route für Socket.IO – index.js

In der Callback-Funktion der `/logout`-Route holen Sie sich sämtliche Nutzer anhand des `connections`-Objekts und entfernen den aktuellen Nutzer. Diese Information versenden Sie mit der `broadcast.emit`-Methode der Verbindung des abzumeldenden Nutzers an alle übrigen Verbindungen. Anschließend trennen Sie die Verbindung, löschen die Referenz aus dem `connections`-Objekt, leeren die Benutzer-Session und leiten den Benutzer auf die Anmeldeseite. Mit diesen Anpassungen haben Sie Ihre Applikation komplett von Websockets auf Socket.IO umgestellt und damit erreicht, dass auch ältere Browser Ihre Applikation nutzen können.

Die Features von Socket.IO, die Sie im Verlauf dieses Kapitels kennengelernt haben, bilden nur einen sehr kleinen Teil des Gesamtumfangs dieser Bibliothek. Falls Sie mehr über Socket.IO lernen möchten, empfehle ich Ihnen die Webseite dieses Projekts unter <http://socket.io> als Startpunkt für weitere Recherchen.

16.5 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie mit nur wenigen Codezeilen aus Ihrer Webapplikation eine echtzeitfähige Anwendung machen können. Diese Art von Software rückt Webapplikationen noch ein Stück näher an traditionelle Applikationen heran. Der Server verfügt hier über die Möglichkeit, aktiv mit den Clients zu kommunizieren und Nachrichten zu versenden, falls relevante Informationen vorliegen. Zur Umsetzung einer Realtime Webapplikation existieren mittlerweile sehr viele Technologien. Vor allem die WebSocket-API wurde in der letzten Zeit immer populärer, da sie auf einem eigenen, sehr leichtgewichtigen Protokoll aufbaut und im Rahmen von HTML5 bei immer mehr Browsern Einzug hält. Mit Socket.IO haben Sie außerdem eine Bibliothek kennengelernt, die einen der größten Nachteile der WebSocket-API umgeht. Websockets sind auf vielen älteren Browsern nicht verfügbar. Socket.IO verfügt über einen Fallback-Mechanismus auf andere Technologien wie beispielsweise AJAX Long Polling, mit dem ein Server auch mit älteren Browsern kommunizieren kann. Am praktischen Beispiel einer einfachen Chat-Applikation haben Sie zum einen gesehen, wie Sie Websockets zur bidirektionalen Kommunikation einsetzen können, und zum anderen, wie sich statt dieser Technologie auch Socket.IO einbinden lässt.

Index

.bom	52
.msi-Paket	45
/doc-Verzeichnis	73
/etc/profile	42, 56
/opt	55
/usr/local	54
/usr/local/lib/node_modules	151
_dirname	94
_filename	94
_read	377
_write	377
1.500 Byte	372
65.535 Byte	372

A

Abfrage	145
Abfrage einer Tabelle	147
Abhängigkeiten	139, 141, 153, 294, 317, 320, 411
Abkürzungen	188, 385
Ableitung	172
Abmelden	438
abort	105
abort-Methode	207
Absicherung des Servers	336
Absoluter Pfad	118, 161
Accept-Header	195, 360
accept-Methode	442
Account	333
ACK-Paket	373
AddressBook	352
addTrailers	201
Administrator	63, 141
Administratorberechtigungen	336
Adressbuch	351
Adresse	379
affected	264
afterEach	289
AJAX Long Polling	451
Ajax-Request	429
Aktualisierung	312, 358
Aktualisierungsmechanismus	149
Alert-Fenster	347
Algorithmus	330
all-Methode	267, 389

Anfragen	208, 325, 330
Anfrage-Optionen	203
Angriffe	335
Angriffspotenzial	343
Anmeldeformular	438
Antipatterns	298
Antwort	208
Antwortheader	199
Anwendung beenden	179
Anzahl von Kindprozessen	340
API frozen	74
API-Dokumentation	72
appendFile	174
append-Methode	273
application/json	361
Applikation	61
Applikationen mit dem Node Package Manager installieren	318
Applikations-Container	421
Applikations-Namespace	422
apt-get	42
Arbeitsspeicher	325
Arbeitsverzeichnis	178
argv	124
Arrange, Act, Assert	281, 285
Assertions	76, 77, 279, 281, 282, 287, 292
assert-Modul	76, 280
Asynchron	147, 173, 271
Asynchrone Funktionsaufrufe	244
Asynchrone Programmierung	223
Attribute	399
Aufbau eines Moduls	152
Auflistung der installierten Pakete	141
Ausfallsicherheit	330
Ausführungskontext	82
Ausgabe	285
Ausgabemethoden	97
Aushandlung	451
Auslagerung	223, 339
Auslieferung	452
Auslieferung von statischen Inhalten	404
Ausspionieren	335
Austauschformat	361
Auszeichnung	396
autoAcceptConnections	442
Autovervollständigung	59
Azure	44

B

Backbone.js	412
Backbone.js Collections	412
Backbone.js Models	412
Backbone.js Router	414
Backbone.js Views	413
Backbone-Model	424
Backbone-View	425
Backend-Applikation	410
Backtraces	305
Base64-Codierung	375
basicAuth	394
Basislayout	401
Basismodule	75, 94
Basisverzeichnis	434
Baumstruktur	142
BDD	284
Bedienbarkeit	433
Beenden des Workerprozesses	239
Befehlsausführung	227
beforeEach	289
Behavior-Driven Developments	284
Benennungsschema	121
Benutzerberechtigungen	336
Benutzer-ID	106, 183
Benutzername	210
Berechnung	338
Berechtigungen	161
Betriebssystem	39, 84, 237
Betriebssystemparameter	101
Bibliotheken	31
Bidirektionale Kommunikation	236, 327, 369
Binärpaket	44
Binary-Paket	48
bin-Verzeichnis	155
block-Element	402
bodyParser	437
Breakpoints	307
BSO-Format	274
Buffer	77, 95, 107
Buffer-Objekt	66, 88, 167, 228
Buildtool	320
Callback-Funktion	145, 163, 224
Cannot GET	405
C-Ares	35, 81
Chakra	25
change	182
Character Set	352
chat-Subprotokoll	442
chdir	179
checkAuth-Middleware	438
checkContinue	191
Child Process	78, 225, 232, 326
chmod	184
chown	184
Chrome	25
Chunk	65, 197, 207, 356
Client	369
Clientbibliotheken	412
clientError	193
ClientRequest	205
Client-Server-Ansatz	433
Client-Server-Kommunikation	218
close	165
close-Event	215
close-Methode	440, 450
Cloud Computing	332
Cloud Storage	332
Cloudbasierte Lösung	312
Cluster	78
cluster.worker	241
Cluster-Ereignisse	240
cluster-Modul	78, 237, 329
Codierung	209
Collection	275, 426
CommonJS	247, 414
compile-Methode	397
Compiler	54
configurable	126
configure	54
CONNECT	192
Connect	392
connect	145
connect-Event	206
Connect-Framework	385
connection-Eigenschaft	195
connection-Event	218
Connection-Pooling	205
connections-Objekt	449
connect-Methode	218, 261
console	78, 95
Console-Modul	78
Content-Type	65, 67, 352, 361

C

C++	39
Cache	120, 325
Caching	29, 195
Caching-Mechanismus	121

content-type	210
continue-Event	206
cookieParser	437
cookieSession	418, 437
Copy-and-Paste-Detection	301
CouchDB	318
createClient	271
createServer	189, 217, 366
createSocket	380
Cross-Site-Scripting	346
CRUD	257
Crypto-Modul	36, 79, 80
CSS	395
cURL	188, 214, 351, 390
Currying	172
Cygwin	44

D

Daemons	315
Dahl, Ryan	20
Darstellung	413
data-Event	209
Datagram	80
Datagramme empfangen	382
data-main	421
Date-Header	199
Dateien	161
Dateien beobachten	182
Dateien lesen	163
Dateien schreiben	169
Dateiendung	122
Dateigröße	167
Datei-Handle	163, 170
Datei-Handle schließen	167
Dateiinformationen	166, 180
Dateisynchronisierung mit rsync	314
Dateisystem	83, 128
Dateisystembasierte Kommunikation	371
Dateisystemberechtigungen	183, 368
Dateisystembrowser	178
Dateisystemoperationen	224, 255
Dateitransfer	374
Dateiübertragung	374
Daten	412
Datenbank	143, 257, 341, 411
Datenbank-Abstraktionslayer	325
Datenbankstruktur	260, 265
Datenbanktreiber	257
Datenbankverbindung	145
Datenbasis	354
Datenmenge	325
Datenpaket	194
Datensätze aktualisieren	263, 268, 272, 276
Datensätze anlegen	271
Datensätze auslesen	262, 267, 272, 276
Datensätze entfernen	263, 268, 273, 277
Datenstreams	107
Datenströme	88, 187, 212, 218, 230, 365
Datenübermittlung	236
Datenübertragung	373
Db-Klasse	275
db-mysql	260
Debugger	80, 303
debugger-Statements	308
Debugging in der Entwicklungsumgebung	309
Debug-Modus	304
defer-Methode	249
Deferred	249, 251
Deflate	36, 93
Defragmentierung	31
Deinstallieren	43, 47, 52
DELETE	359
del-Methode	273
Denial of Service	340
dependencies	153
Dependency Injection	294
Deployment	312, 318
describe-Method	284
Desktop-Applikation	409, 433
destroy-Methode	243
destroySoon	216
devDependencies	153, 321
dgram-Modul	379
Dienste unter UNIX	315
Dienste unter Windows	316
Dienstverwaltung	317
disconnect	226, 239, 243, 244
div-Element	398
DNS	31, 35, 80, 379
dns.lookup	35
DNS-Modul	81
Doctype	397
Dokumentenorientierte Datenbank	274
Domain	81
Domain-Modul	81
done-Methode	290, 293
DOS-Attacken	341
Douglas Crockford	298
drain-Event	377
Duplex-Stream	377

Duplikate	301
Durchlaufzeiten	79
Durchsatz	325
Dynamische URLs	388
Dynamische Webapplikationen	62
Dynamischer Inhalt	395

E

EACCES	175
each	268
EADDRINUSE	190, 369
Echtzeitfähige Webapplikation	434
Eclipse	309
ECMAScript	22, 126
ECMAScript-Standard	100
Eigene Klassen	110
Eigene Matcher	288
eigene Middleware	393
Eigene Module	95, 115
Eigene Pakete	155
Eigenschaften	26
Eigentümer	184
Einbinden von Plugins	322
Eindeutige ID	276
Eindeutige Kennung	241
Einfaches Deployment	312
Eingabe und Ausgabe	33
Eingabedatei	61, 229
Einrückungen	396
Einsatzgebiete	23
Einstiegsdatei	386
Einstiegspunkt	117, 119, 124, 153
Elternklasse	99
Elternmodul	117
emit-Methode	453
Encoding	88, 200
end	200
end-Event	197, 375
end-Methode	202, 206, 216
entities	347
Entwicklungsumgebung	309
enumerable	126
Ereignisse	101, 243
Erfolgsfall	360
Erfolgsmeldung	296
Erfüllung	247
error-Event	213, 215
Erweiterbarkeit	301
Escape Output	335

Escapesequenz	173
Escaping	266, 342, 399
Eval	343
Evaluierten	91
EventEmitter	81, 82, 172, 205, 356
Eventgetriebener Ansatz	82, 172, 173, 188, 224, 238, 271, 413, 441
EventListener	100
Eventloop	32
Events	82, 100, 413, 429
events.EventEmitter	32
Exception	225
Exception Handling	247
execFile-Methode	229
exec-Methode	227
execute	261
exit	104
exit-Event	239
expect-Methode	285
experimental	74
Explorer-Klasse	178
exports	95, 115, 117, 124
exports-Objekt	290
express.js	386, 402, 435
express.logger	392
ExtendedTimeTracker	131
extends	402
extensions	122
Externe Kommandos	227

F

fail-Methode	252
Fallback-Route	388
fcall	248
Fehler	175
Fehlerbehandlung	253
Fehlercode	104
Fehlerfall	252
Fehlermeldung	225, 394
Fehler-Objekt	163, 178, 180, 228, 262, 282
Fehlersuche	303
Fehlschlagender Test	295
File System-Modul	83
Filedescriptor	181, 216
Filesystem	83
Filesystem Hierarchy Standard	41
Filesystem-Modul	34
Filter Input	335

Filterprozess	336
finally-Statement.....	253
find-Methode	276
fin-Methode.....	253
Firewall.....	341
Flash Sockets.....	451
Flusssteuerung	376
foreman	333
Fork.....	232, 240
format	96, 212
Fragmentierung	372
Fremdschlüssel.....	258, 263
from	262
Frontend.....	452
fs-Modul.....	254, 375
FTP	313
Funktionen exportieren	135
Funktionsebene	294

G

Garbage Collection.....	25, 30
GCC Runtime Library	32
Gesamtpaket	197
Geschichte	19
GET – lesender Zugriff.....	351
getaddrinfo	35, 81
GET-Anfrage.....	390
get-Methode	267, 272, 387
Getter-Methode.....	126, 133, 172
Gewichtung.....	330
Git Publishing	333
Globale Objekte	94
Globaler Kontext	61, 94
Globe Installation.....	139, 153, 157
Größenbeschränkung.....	383
Grundlagen eines UDP-Servers.....	379
grunt	142
Grunt ausführen	322
grunt watch.....	325
grunt-cli.....	142
grunt-contrib-watch-Plugin.....	324
Gruntfile.....	142
Gruntfile.js.....	320
Grunt-Plugins.....	321
Gruppen	293
Gruppen-ID	106, 183
Gzip	36

H

Handshake.....	372, 379
HAproxy	330
Hash	414
Hashes	273
Häufige Ausführung.....	297
Header	193
Header X-Response-Time	393
Header-Felder.....	372
Header-Informationen.....	197, 205
Heroku Toolbelt	332
Hidden Class	26, 27
Hilfsvariablen.....	133
Historie	60
Hochperformanter Bibliotheken	311
Hostname	210, 218
HTML	361
HTML5	439
HTML-Code	395
HTML-Injection	347
HTML-Seite.....	385, 394
HTML-Tags.....	396
HTML-Zeichenkette	67
HTTP.....	83
http.Agent.....	203
http.STATUS_CODES	198
HTTP-Body.....	65
HTTP-Client.....	84, 201
HTTP-Header	65
HTTP-Kommandos	351
HTTP-Methoden.....	192, 194, 360, 389
http-Modul	83, 349, 365
HTTP-Parser.....	32, 36
HTTP-Protokoll	36
HttpProxyModule	331
HTTPS	84
HTTP-Server.....	62, 188, 239, 349, 441
HTTP-Statuscode.....	65
HTTP-Version	195

I

I/O-Operationen	337
ID-Attribut	398
idAttribute.....	424
Include	401
index.jade	397
Informationen im Debugger.....	305
inherits.....	99

ini-Parser.....	164
Initialisierung.....	111
Injectons.....	399
inotify.....	182
insert.....	261, 275
inspect.....	98
Installation.....	39, 259, 265, 270, 274, 285, 290, 312, 386, 395
Installation und Einbindung.....	451
Installer-Paket.....	48
Integration in express.js.....	402
Interaktion.....	187
Interaktive Modus.....	57
Internes Modul.....	118
Interprozesskommunikation.....	226
Intervall.....	89
IOCP.....	34
io-Objekt.....	453
IP-Adresse.....	63
IP-Hash.....	331
isMaster.....	238
Isoliert280.....	
isWorker.....	238
it-Methode.....	284

J

Jade.....	402, 436
Jade-Template.....	444
JägerMonkey.....	25
Jasmine.....	284
jasmine-node.....	284
join.....	263
Joyent.....	22, 137
jQuery.....	444
jsconf.eu.....	21
JSLint.....	298, 323
JSLint-Plugin.....	323
JSON.parse.....	445
JSON-Format.....	352, 420
JSON-Objekt.....	207
JSONP Polling.....	451
Just-in-time-Kompilierung.....	25, 28

K

Kapselung.....	351
Kategorien der Statuscodes.....	198
keine Verbindung.....	379

Kernel-Puffer.....	215
Key-Value-Store.....	270
kill.....	105, 226
Kindprozess.....	78, 225, 233, 326
Kindprozess beenden.....	226, 234
Klassen exportieren.....	136
Klassen-Attribut.....	398
Kommando.....	59, 228, 304
Kommandos im Überblick.....	151
Kommandozeile.....	86, 122, 157, 232, 281, 298
Kommandozeilenoptionen.....	106, 111, 302
Kommandozeilenwerkzeug.....	24, 106, 110, 141
Kommentar.....	406
Kommunikation.....	100, 187, 365
Kommunikationsendpunkte.....	217
Kommunikationsverbindung.....	439
Kompilieren und installieren.....	54
Kompiliervorgangs.....	27
Komprimieren.....	93
Konfiguration.....	322
Konfigurationsdatei.....	330, 421
Konfigurationsinformationen.....	161
Konfigurationsobjekt.....	145, 237
Konfigurationsoptionen.....	55, 392
Konfliktauflösung.....	327
Konstruktor.....	99, 112, 126, 130, 172, 352
Kontext.....	91, 172
Kopieren.....	312
Kopplung.....	115, 413
kqueue.....	182
Kris Kowal.....	248
Kris Zyp.....	248
Kurzschreibweise.....	152, 398

L

Ladeoperationen.....	121
lastID.....	267
Layoutunterstützung.....	415
Least Connections.....	331
Lesbarer Datenstrom.....	378
Lesebuffer.....	377
libeio.....	34
libev.....	33, 34
libevent.....	33
libevent2.....	33
libuv.....	34
lib-Verzeichnis.....	72, 155
Link-Shortener.....	347
Linux.....	39, 41

Linux Binaries 41
 Liste der vorhandenen Datensätze 420
 Liste der wichtigsten Module 137
 Listen 273
 listening-Event 239, 244
 listen-Methode 189, 219, 367
 list-Kommando 305
 LiveScript 19
 Lizenzbedingungen 49
 Lizenzinformationen 45
 Loadbalancer 78, 237, 329
 Location 355
 Location-Headers 357
 Lock-Dateien 327
 locked 74
 Locking-Mechanismus 327
 Lodash 414
 Logger 170, 392
 Login 415
 Login-Prozess 416
 Logout 449
 Logout-Prozess 449
 Lokale Installation 139
 Loopback-Schnittstelle 63
 Löschen 359

M

Mac OS X 39, 48
 main-Container 426
 Major-Version 154
 make 54
 make install 55
 Makefiles 54
 makeNodeResolver 250
 Mark-and-Sweep 31
 Maschinencode 25, 28
 Massendaten 381
 Maßzahlen 298
 Masterprozess 237
 Master-Slave-Verbund 259
 Matcher 287
 maxconn 330
 maxHeadersCount 190
 md5 80
 Mediendaten 404
 Mehrere Clients 371
 Mehrere parallele Prozesse 328
 Mehrzeilige Attributangaben 399
 message-Event 236, 244, 380, 442

Metainformationen 61, 196
 Method Invocation 344
 Methodenausführung 344
 Metriken 279
 Middleware 386, 392, 418, 437
 MIME-Type 77
 Minor-Version 154
 Mixin 400
 mkdir 179
 Model View Controller-Pattern 395
 Modularer Ansatz 71
 Modulcache 121, 130
 Module 62, 71, 75, 84
 module.exports 95
 module.filename 117
 module.id 117
 Module-Modul 84, 117
 Modulloader 96, 115, 118, 414
 Modulsuche 118
 Modulsystem 71, 84, 311, 421
 MongoDB 274, 411
 MongoDB-Client 274
 Movie Database-Modul 416
 Multi-Client-Chat 434
 Multi-Page Webapplikationen 385
 MySQL 143, 258, 411
 mysql-Modul 144
 MySQL-Protokoll 259

N

Nachrichten 236, 243, 440
 Nachrichtenkörper 193
 Nachrichtentypen 453
 Namensauflösung 81
 Namenskonvention 220, 371
 Navigation 175, 422
 Navigation im Debugger 304
 Navigationshilfe 109
 Nebenläufigkeit 223
 Negierungen 283, 287
 net.Socket 370
 Net-Modul 84, 365
 Net-Server 239
 Network Time Protocol 379
 Netzwerk 187
 Netzwerkbasierte Sockets 372
 Netzwerkverbindung 372
 Neue Datensätze anlegen 261, 266, 427
 Neue Route 430

Neustart.....	314	Oktalnotation.....	184
next-Funktion.....	391, 393	online-Event.....	239, 243
nfbind.....	250	open.....	165
nfcall.....	250	open-Methode.....	275
nfinvoke.....	250	Open-Source-Projekte.....	318
Nginx.....	331	OpenSSL.....	36
Nicht-relationale Datenbanken.....	269	Optimierung.....	120, 311
Nitro.....	25	Ordnerstruktur.....	410
Node in der Cloud.....	332	origin-Eigenschaft.....	442
Node Package Manager.....	136	OS.....	84, 101
node.exe.....	44	OSI-Modell.....	217, 365
Node.js.....		OS-Modul.....	85
<i>Deinstallation</i>	47	Overhead.....	373
<i>Installation</i>	39		
<i>Versionierung</i>	39	P	
Node.js und Promises.....	250		
Node.js-Prozess.....	63, 104	package.json.....	119, 152, 156, 318, 320, 411, 434, 451
Node.js-Shell.....	87	Pakete aktualisieren.....	149
node_modules.....	118, 138, 142	Pakete anzeigen.....	142
node_modules beim Deployment.....	317	Pakete entfernen.....	151
NODE_PATH.....	118	Pakete installieren.....	138
NODE_UNIQUE_ID.....	238	Pakete lokal installieren.....	319
nodeunit.....	289	Pakete suchen.....	137
nodeunit-Kommando.....	290	Pakete verwenden.....	142
Nonblocking I/O.....	21	Paketmanager.....	40, 42
Normalisierung.....	108	Paketverwalter.....	137
NoSQL.....	269	Parallelisierung.....	329
NPM.....	45, 136	params-Eigenschaft.....	389
npm install.....	138	parse-Methode.....	202
npm list.....	142	Partitionieren.....	259
npm publish.....	158	Passwort.....	210
npm search.....	138	Patch Level.....	154
npm uninstall.....	151	Path.....	85, 108
npm update.....	150	pause.....	210, 214
npm useradd.....	158	pause-Methode.....	376
npmjs.org-Repository.....	152	Performanceaspekt.....	325
NPM-Module.....	257	Performancemessungen.....	393
NPM-Paket.....	332, 451	Persistierung.....	127, 161, 257
NPM-Repository.....	317	Pfad.....	161
Nutzdaten.....	355	Pfadangabe.....	108, 403
Nutzeraccount.....	158	Pfadnamen.....	85
Nutzerbasis.....	311	Pfadtrenner.....	108, 162
		Pipe.....	214, 215, 378, 398
O		pkg-Datei.....	48
		pkgutil.....	52
Object.create.....	27	Platzhalter.....	154, 399
Object.defineProperty.....	126	Plugins.....	321
Objekte exportieren.....	136	PMD CPD.....	300
Objektstruktur.....	98		
Octet-Streams.....	107		

Port.....	210, 379
Port 5858.....	304
Portnummer.....	218
POST – Anlegen neuer Ressourcen.....	354
POST-Anfrage.....	390
Primzahlen.....	235
printf.....	96
Prioritäten.....	170
Process.....	96, 103
process.argv.....	234
process.exit.....	180
process.getuid.....	106
process.setuid.....	106
process-Objekt.....	114
Procfile.....	332
Programmierschnittstelle.....	452
PromisedIO.....	253
Promises.....	244, 249, 250
Protokoll.....	210
Protokolldatei.....	129
Protokolleinträge.....	172
Prototyp.....	27
proxy_pass.....	331
Proxyserver.....	192
Prozess.....	227
Prozess-ID.....	242
Prozessor.....	325
Prozessorkern.....	237
Prozessorressourcen.....	326
Prozesssteuerung.....	104
Prüfsumme.....	372
Publish-Subscribe.....	32
Puffer.....	215
Punycode.....	85
PushState.....	414, 422, 424
Push-Technologien.....	354
PUT – Aktualisierung bestehender Daten.....	357

Q

Q.....	248
Qualitätsmaßstäbe.....	280
Qualitätssicherung.....	279
Quellcodeformatierung.....	300
Quelldateien.....	54
query.....	69, 211, 261
Query String.....	85
query-Methode.....	145
quit.....	271
QUnit.....	284

R

RangeError.....	126
read.....	165
Readable Stream.....	213
readable-Event.....	367
readdir.....	178
ReadDirectoryChangesW.....	183
Read-Eval-Print Loop.....	57, 87
readFile.....	130, 168
Readline.....	86
read-Methode.....	367, 375
Rechenintensive Operationen.....	224
Redis.....	270
Redis-Client.....	270
ReferenceError.....	400
Referenzen.....	448
Regelsätzen.....	300
registerTask.....	324
Registry.....	317
Reguläre Ausdrücke.....	388
Reihenfolge.....	394
reject-Methode.....	250, 442
Rekursiv.....	314
Relationale Datenbanken.....	257, 258
Relativer Pfad.....	161
Releasezyklen.....	312
remove-Methode.....	277
rename.....	182
Rendering.....	386
render-Methode.....	403, 428
REPL.....	57, 87, 305
REPL verlassen.....	59
REPL-Befehle.....	59
Replizieren.....	318
REPL-Modul.....	87
Repository.....	137, 318
request-Ereignis.....	190
request-Event.....	387
Request-Objekt.....	65, 193
require.....	72, 84, 96, 117, 121, 124, 143
RequireJS.....	414
resolve.....	121
resolve-Methode.....	250
response-Event.....	205
Response-Objekt.....	65, 197
responseTime.....	393
Ressourcen.....	234
Ressourcenforderungen.....	329
Ressourcenzugriff.....	327

REST..... 350, 412
 REST-Service 349
 resume 210, 214
 resume-Methode 376
 Round Robin..... 330, 331
 Route..... 436
 Router..... 422, 424, 427
 routes-Eigenschaft 392
 Routing 386, 387
 rowid..... 265
 Roy Fielding..... 350
 rsync-Kommandozeilenbefehl 314
 Rückgabe 294
 Rückgabewert..... 245
 Rückkanal 433
 Runlevel..... 315
 run-Methode 266, 268

S

Sandbox..... 335
 save-Methode..... 429
 Scavenge Collector 30
 Schadcode..... 336, 346
 Schadsoftware..... 335
 Schlueter, Isaac 22, 136
 Schnittmenge..... 143
 Schnittstelle..... 110, 116, 187
 Schreibbarer Datenstrom..... 378
 Schreibberechtigung..... 337
 Schreibpuffer..... 377
 Schutz des Clients..... 346
 scp 312
 SecondsFormatter 111
 Secure Shell-Protokoll 312
 Seiteneffekte..... 294
 Seitenreload..... 433
 SELECT-Abfrage 267
 select-Methode 262
 Sencha Labs..... 385
 send-Methode..... 236, 381
 Separate Datei 115, 400
 Sequenznummer..... 372, 379
 Server..... 369
 server.js..... 61
 Server-Events 190
 Serverimplementierung..... 452
 Server-Klasse 275
 Serverkomponente..... 441, 451
 Server-Objekt..... 188
 Serverprozess 264
 session..... 438
 Session-Daten 450
 set 263
 setBreakpoint 307
 setHeader 199
 setInterval..... 89
 set-Methode..... 272, 403
 setNoDelay 207
 setSocketKeepAlive..... 208
 Setter-Methode..... 126, 133, 172
 setTimeout 89
 Setup 410
 setUp..... 293
 Setup und Initialisierung der Applikation .. 386
 setup-Event 239
 setupMaster 237
 sha1 80
 Shebang 143
 Shellscripate 232
 shim 422
 Sichere Websockets..... 439
 Sicherheit 335
 Sicherheitsaspekte 335
 Sicherheitsrisiko 313
 Sicherungsmaßnahmen 379
 SIGHUP 239
 SIGKILL..... 106, 226
 Signatur 79
 SIGTERM..... 105, 226
 Simulieren 314
 Single-Page Applikationen..... 409, 433
 Single-Threaded-Ansatz 20, 33, 78, 223, 337
 Singletons 136
 Skalierbarkeit und Deployment 311
 Skalierung..... 325
 Socket 208
 socket.broadcast.emit 455
 socket.emit 455
 Socket.IO..... 450
 Socket-Client..... 370
 Socket-Datei..... 367
 Socket-Lösungen..... 372
 Socket-Objekt 195
 Socketpool..... 203
 Socket-Server..... 365
 Socket-Verbindungen 448
 Softwarequalität 279
 Sonderzeichen maskieren..... 347
 Spannen von Versionen..... 154
 spawn-Methode 230

spec..... 285
 Speichermodell 26
 Spy 288
 spyOn-Methode 289
 SQL..... 258
 SQL-Abfrage 266
 SQL-Injections..... 341
 SQLite 264, 411
 sqlite3-Treiber 265
 SSH..... 315
 Stabilitätsindex 73
 stable..... 74
 Stacktrace..... 286, 292
 Standard C Bibliothek 32
 Standard Library 72
 Standardaufgaben 385
 Standardausgabe 87, 90, 97, 225, 228, 231
 Standardeingabe 87, 90, 187, 225, 231
 Standardfehlerausgabe 97, 225, 228
 Standardrepository..... 42
 Standardroute..... 422
 Standardtask..... 324
 static-Middleware..... 405, 416, 444
 Statische Codeanalyse 298
 Statische Inhalte..... 404
 Stats-Objekt..... 180
 Statuscode 104, 192, 197, 208, 352, 360, 391
 Statusnachrichten 445
 stderr 104
 stdin 104
 stdout 104
 Strategien..... 311
 Stream-Modul..... 88, 365, 377
 Streams 90, 212
 Strict Mode..... 300, 344
 String Decoder-Modul 88
 Structured Query Language 258
 Struktur 298
 Strukturierung..... 115, 175, 293, 396
 Stylesheet..... 405
 Styling 415
 Subklasse..... 99
 Subprotokoll..... 439
 Subshell 228
 Suchoperationen 175
 Suchpfad..... 42, 45, 52, 56
 Suchprozess..... 119
 Suchstring..... 147
 Suchvorgang..... 141
 sudo..... 41, 141
 SYN/ACK-Paket 373

Synchronisierung 313, 314, 329
 SYN-Paket..... 372
 Syntaktische Korrektheit..... 300
 Systeminformationen 103
 Systemmonitor 103
 Systemstart 315
 Systemweite Installation..... 139

T

Tabelle 258
 Tagged Pointers..... 26
 Tar-Archiv 320
 Tasks..... 324
 TCP..... 84, 216, 365
 TCP-Client 218, 374
 TCP-Portnummer 63, 217
 TCP-Server 218, 373
 TCP-Sockets 371
 TCP-Verbindungen..... 330
 tearDown 293
 Technologieevaluierung..... 24
 Template 425, 436
 Template-Dateien..... 410
 Template-Engine 395
 Templates mit Jade 394
 Test
 fehlgeschlagener 286
 organisieren 292
 Testabdeckung..... 295
 Testfall..... 280
 Testframework..... 281
 Testfunktionen 284
 Testgetriebene Entwicklung..... 295
 Textnachrichten..... 445
 text-Plugin..... 425
 then 247
 this.changes 268, 269
 this.lastID 268
 Thread 227
 Thread-Pool..... 237
 Timeout 89
 Timeoutfehler 64
 Timeoutspanne 207
 Timers 89
 Time-Tracker..... 122
 TLS..... 89, 192
 TLS-Modul..... 89
 Toolunterstützung mit Grunt 320
 toString 108, 113

Trailer	195, 198
Transaktionen.....	259
Transfer-Encoding.....	207
Transportschicht.....	217
Trigger.....	259
TTY.....	90
Twitter Bootstrap.....	415
Typ des Objekts	180
Typenprüfung	98

U

Übertragungsprotokoll	217
Ubuntu.....	54
UDP	365
UDP-Client.....	380
UDP-Protokoll.....	80
UDP-Server	380
UDP-Sockets	379
UglifyJS.....	323
Utilities-Modul.....	91
Umgebung zur Laufzeit	303
Underscore.js.....	143, 414
Undone tests	293
UND-Verknüpfung.....	154
Unittests.....	76, 280
Universität Aarhus.....	25
UNIX.....	39
UNIX Domain Socket	190, 219
UNIX-Sockets.....	84, 366, 376
Unkontrollierte Ausführung.....	343
Unstable	74, 182
unwatchFile	182
unwatch-Funktion	307
UPDATE-Abfrage.....	268
update-Methode	263, 276
Updates.....	315
Upgrade.....	192
upgrade-Event	206
URL	69, 85, 90, 194, 210, 351, 365
url.parse.....	212
url-Eigenschaft.....	210
URL-Modul	90
url-Modul.....	211
URL-Pfad.....	211, 388
use-Methode.....	392
Userliste.....	445
util.puts	114
Utilities	91
Utility.....	96

V

V8-Engine.....	22, 24
Verantwortung	335
Verbindung	442
<i>aufbauen</i>	191, 260, 265, 270, 274, 372
<i>beenden</i>	197
<i>trennen</i>	456
Verbindungsinformationen.....	193
Verbindungsloses Protokoll.....	379
Vererbung.....	99, 131, 172, 401
Verkettung	247
Verschachtelung	245, 287, 396
Verschlüsselte Verbindungen	89
Verschlüsselung	31, 79
Version	
<i>asynchron</i>	162
<i>synchron</i>	162
Version eines Moduls	149
Versionierung	39
Versionsinformationen	43, 47, 52, 106
Versionskonflikte	119, 139
Versionsnummer	61, 154
Verteilung.....	312
Verwendung von Jade.....	398
Verzeichnis	285
Verzeichnisfreigaben	313
Verzeichnishierarchie	150
Verzeichnisoperationen	175
Verzeichnisse manipulieren	175
Verzeichnisstruktur.....	118, 280, 410
view engine	403, 436
View-Repräsentation.....	425
Visuelle Gruppierung.....	292
VM.....	91
Vorteile	22

W

warnings.....	264
Wartbarkeit	301
Warteschlange	190
watch	182
watch-Funktion	306
Webapplikation	385
Webserver.....	62, 188, 331, 349, 385
WebSocket	193, 354, 433, 439, 444, 451
Websocket-Paket.....	441
Websocket-Protokoll.....	439

WebStorm..... 309
 Webworker 78
 Weiterentwicklung 153
 Weiterführende Operationen..... 180
 Werkzeuge 142
 Werkzeugsammlung..... 135
 wget..... 41
 when-Methode 253
 where-Methode..... 262
 Whitelist..... 345
 Wildcards 390
 Windows..... 34, 39, 44
 Windows Azure 333
 Windows Pipes..... 220, 371, 376
 wordCount 116
 Workerprozess 78, 237, 241, 328
 workers 241
 Wrapper..... 387
 Wrapper-Funktion..... 250
 writable..... 214
 Writable Stream 214
 write 170, 200
 writeFile..... 174, 179
 writeHead 199

write-Methode 206, 215
 ws:// 439
 wss:// 439

X

XML 361

Z

Zeichencodierung..... 66, 213, 215
 Zeilenumbrüche..... 396
 Zeitabhängigkeit 89
 Zeitgesteuerte Berechnungen 434
 Zielsysteme 312, 336
 Zirkuläre Abhängigkeit..... 134
 zlib 32, 36, 93
 ZLIB-Modul..... 93
 Zugriff auf den Socket 367
 Zugriff auf die Umgebung..... 305
 Zugriffsberechtigungen 175, 183, 336
 Zugriffsflags 166