



Leseprobe

Mit diesem Buch haben Sie alles an der Hand, um mit Node.js die Vorteile von JavaScript auf die Serverseite zu bringen. Die Leseprobe zeigt Ihnen die Grundlagen von Node.js, und Sie entwickeln exemplarisch eine Single-Page-Applikation mit Express.js und Angular. Außerdem können Sie einen Blick in das Inhaltsverzeichnis und Stichwortverzeichnis des Buchs werfen.



»Grundlagen«
»Single-Page-Webapplikationen mit
Express.js und Angular.js«



Inhaltsverzeichnis



Index



Der Autor



Leseprobe weiterempfehlen

Sebastian Springer

Node.js – Das Praxisbuch

560 Seiten, gebunden, 2. Auflage 2016
39,90 Euro, ISBN 978-3-8362-4003-1



www.rheinwerk-verlag.de/4037

Kapitel 1

Grundlagen

Aller Anfang ist schwer.
– Ovid

Mehr Dynamik in Webseiten zu bringen, das war die ursprüngliche Idee hinter JavaScript. Die Scriptsprache sollte die Schwachstellen von HTML ausgleichen, wenn es darum ging, auf Benutzereingaben zu reagieren. Die Geschichte von JavaScript geht zurück auf das Jahr 1995, wo es unter dem Codenamen Mocha von Brendan Eich, einem Entwickler von Netscape, entwickelt wurde. Eine der bemerkenswertesten Tatsachen über JavaScript ist, dass der erste Prototyp dieser erfolgreichen und weltweit verbreiteten Sprache in nur 10 Tagen entwickelt wurde. Noch im Jahr der Entstehung wurde Mocha in LiveScript und schließlich in einer Kooperation zwischen Netscape und Sun in JavaScript umbenannt. Dies diente vor allem dem Marketing, da zu diesem Zeitpunkt davon ausgegangen wurde, dass sich Java als führende Sprache in der clientseitigen Webentwicklung durchsetzen würde.

Vom Erfolg von JavaScript überzeugt, integrierte auch Microsoft 1996 eine Scriptsprache in den Internet Explorer 3. Das war die Geburtsstunde von JScript, welches größtenteils kompatibel mit JavaScript war, allerdings um weitere Features ergänzt wurde.

Das gegenseitige Wettfeiern der beiden Unternehmen ist heute bekannt als die Browserkriege. Die Entwicklung sorgte dafür, dass die beiden JavaScript-Engines sowohl im Featureumfang als auch in der Performance stetig verbessert wurden, was zu einem Großteil für den heutigen Erfolg von JavaScript verantwortlich ist.

Im Jahr 1997 entstand der erste Entwurf des Sprachstandards bei der ECMA International. Unter der kryptischen Bezeichnung ECMA-262 beziehungsweise ISO/IEC 16262 ist der gesamte Sprachkern der Scriptsprache festgehalten. Den aktuellen Standard finden Sie unter www.ecma-international.org/publications/standards/Ecma-262.htm. Die Hersteller der verschiedenen JavaScript-Engines implementieren die ältere Version 5 des Standards nahezu vollständig und integrieren auch bereits zahlreiche Features des aktuellen Standards. Der Erfolg von JavaScript lässt sich auch gut grafisch darstellen. So ist JavaScript in den Sprachtrends von Github seit dem Jahr 2008 immer auf den vorderen beiden Plätzen zu finden.

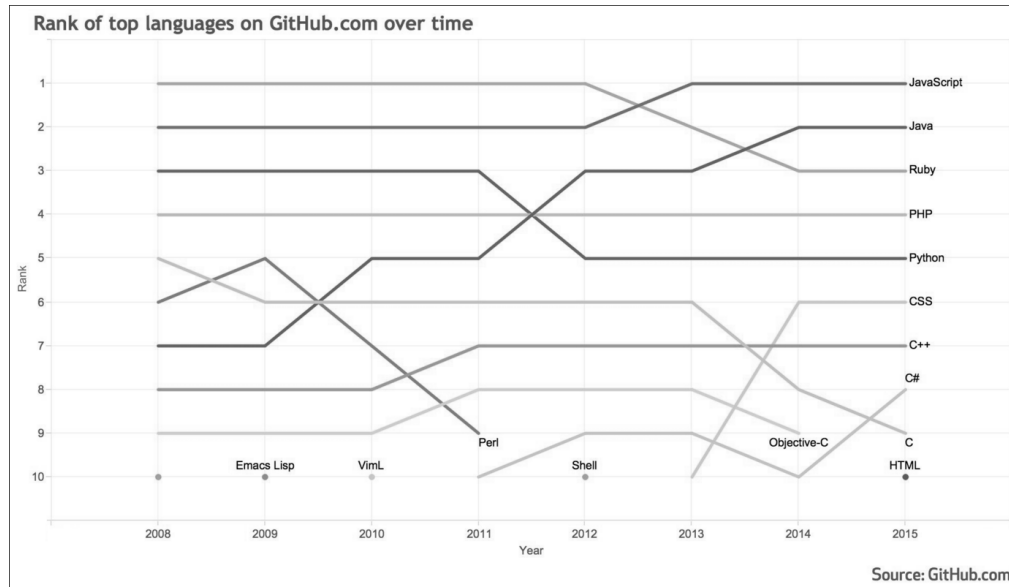


Abbildung 1.1 Topsprachen in Github

Node.js basiert auf dieser erfolgreichen Scriptsprache und hat selbst einen kometenhaften Aufstieg hingelegt. Dieses Kapitel soll Ihnen als Einführung in die Welt von Node.js dienen und Ihnen einen ersten Eindruck vermitteln, wo Sie die Plattform einsetzen können. Im ersten Schritt erfahren Sie zunächst einmal mehr über die Entwicklungsgeschichte von Node.js.

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.

1.1.1 Die Ursprünge

Node.js wurde ursprünglich von Ryan Dahl entwickelt, einem Doktoranden der Mathematik, der sich eines Besseren besann, seine Bemühungen abbrach und stattdessen lieber mit einem One-Way-Ticket und nur sehr wenig Geld in der Tasche nach Südamerika reiste, wo er sich mit Englischunterricht über Wasser hielt. In dieser Zeit kam er sowohl mit PHP als auch mit Ruby in Berührung und entdeckte darüber seine Liebe zur Webentwicklung. Das Problem bei der Arbeit mit dem Ruby-Framework Rails war, dass es nicht ohne Workarounds möglich war, mit konkurrierenden Anfragen umzugehen. Die Applikationen waren zu langsam und lasteten die CPU vollstän-

dig aus. Eine Lösung für seine Probleme fand Ryan Dahl in Mongrel. Dabei handelt es sich um einen 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 damalige Single-Threaded-Ansatz von Ruby und die große Anzahl an Requests, die benötigt wurden.

Einen weiteren vielversprechenden Ansatz bot eine Umsetzung in C. Hier war Ryan Dahl nicht auf einen Thread begrenzt. C als Programmiersprache 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 treten auch keine Problemstellungen auf, 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.

1.1.2 Die Geburt von Node.js

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.

Bereits im September 2009 begann Isaac Schlüter seine Arbeit an einem Paketmanager für Node.js, dem Node Package Manager, auch bekannt als NPM.

1.1.3 Der Durchbruch von Node.js

Nachdem Ryan Dahl sämtliche Komponenten integriert hatte und erste lauffähige Beispiele auf der neuen Node.js-Plattform erstellt waren, benötigte er 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. Die Aufzeichnung dieses Vortrags finden Sie unter www.youtube.com/watch?v=EeYvFl7li9E. Die Präsentation verfehlte ihr Ziel nicht und führte dazu, dass Joyent als Sponsor für Node.js einstieg. Joyent ist ein Anbieter für Software und Ser-

vice 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.

1.1.4 Node.js erobert Windows

Einen bedeutenden Schritt in Richtung Verbreitung von Node.js machten die Entwickler, indem sie im November 2011 in der Version 0.6 die native Unterstützung für Windows einführten. Bis zu diesem Zeitpunkt konnte Node.js nur umständlich über Cygwin unter Windows installiert werden.

Seit der Version 0.6.3 im November 2011 ist der Node Package Manager fester Bestandteil der Node.js-Pakete und wird dadurch bei der Installation von Node.js automatisch ausgeliefert.

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 aktiv an der Entwicklung des Kerns von Node.js beteiligt. Dieser Wechsel verunsicherte die Community, da nicht klar war, ob die Weiterentwicklung der Plattform auch ohne Ryan Dahl weiterlaufen würde. Ein Signal, dass die Node.js-Community stark genug für eine solide Weiterentwicklung war, gab die Veröffentlichung der Version 0.8 im Juni 2012, die vor allem die Performance und Stabilität von Node.js entscheidend verbessern sollte.

Mit der Version 0.10 im März 2013 veränderte sich eine der zentralen Schnittstellen von Node.js: die Stream-API. Mit dieser Änderung wurde das aktive Pullen von Daten von einem Stream möglich. Da sich die bisherige API schon sehr weit verbreitet hatte, wurden beide Schnittstellen weiter unterstützt.

1.1.5 io.js – der Fork von Node.js

Im Januar 2014 gab es erneut eine Änderung in der Projektleitung von Node.js. Auf Isaac Schlüter, der die Maintenance von Node.js zugunsten seines eigenen Unternehmens npmjs, des Hosters des NPM-Repositorys, aufgab, folgte TJ Fontaine. Unter seiner Regie wurde im Februar 2014 die Version 0.12 veröffentlicht. Ein weit verbreiteter Kritikpunkt an Node.js war zu diesem Zeitpunkt, dass das Framework immer noch nicht die vermeintlich stabile Version 1.0 erreicht hatte, was zahlreiche Unternehmen davon abhielt, Node.js für kritische Applikationen einzusetzen.

Viele Entwickler waren unzufrieden mit Joyent, das seit Ryan Dahl die Maintainer für Node.js stellte, und so kam es im Dezember 2014 zum Bruch in der Community. Das

Resultat war io.js, ein Fork von Node.js, der getrennt von Node.js weiterentwickelt wurde. Daraufhin wurde im Februar 2015 die unabhängige Node.js Foundation gegründet, die für die Weiterentwicklung von io.js zuständig war. Zeitgleich erschien die Version 0.12 des ursprünglichen Node.js-Projekts.

1.1.6 Node.js wieder vereint

Im Juni 2015 wurden die beiden Projekte io.js und Node.js in der Node.js Foundation zusammengeführt. Mit der Version 4 des Projekts wurde die Zusammenführung abgeschlossen. Die weitere Entwicklung der Node.js-Plattform wird nun von einem Komitee innerhalb der Node.js Foundation und nicht mehr von einzelnen Personen koordiniert. Das Resultat sind häufigere Releases und eine stabile Version mit Langzeit-Support.

Nachdem Sie nun wissen, wie die Node.js-Plattform entstanden ist, stellt sich die Frage, in welchen Bereichen sie zum Einsatz kommt.

1.2 Vorteile von Node.js

Die Entwicklungsgeschichte von Node.js zeigt eine Sache sehr deutlich: Sie 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 können Sie auch während der Entwicklung von Webapplikationen sehr flexibel auf sich ändernde Anforderungen 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 Fragen und Problemstellungen recht schnell und kompetent über verschiedenste Kommunikationskanäle beziehungsweise die Community gelöst werden können.

1.3 Einsatzgebiete von Node.js

Vom einfachen Kommandozeilenwerkzeug bis hin zum Applikationsserver für Webapplikationen, der auf einem Cluster mit mehreren Knoten läuft, kann Node.js überall eingesetzt werden. Der Einsatz einer Technologie hängt stark von der Problemstellung, den persönlichen Präferenzen und dem Wissensstand der Entwickler ab.

Aus diesem Grund sollten Sie sowohl die wichtigsten Eckdaten von Node.js kennen wie auch ein Gefühl für die Arbeit mit der Plattform haben. Den zweiten Punkt können Sie nur erfüllen, wenn Sie entweder die Möglichkeit haben, in ein bestehendes Node.js-Projekt einzusteigen, oder die Erfahrung im besten Fall mit kleineren Projekten sammeln, die Sie umsetzen.

Nun aber zu den wichtigsten Rahmendaten:

- **Reines JavaScript:** Bei der Arbeit mit Node.js müssen Sie keinen neuen Sprachdialekt lernen, sondern können auf den Sprachkern von JavaScript zurückgreifen. Für den Zugriff auf Systemressourcen stehen Ihnen standardisierte und gut dokumentierte Schnittstellen zur Verfügung.
- **Optimierte Engine:** Node.js baut auf der JavaScript-Engine V8 von Google auf. Sie profitieren hier vor allem von der stetigen Weiterentwicklung der Engine, bei der nach kürzester Zeit die neuesten Sprachfeatures unterstützt werden.
- **Nonblocking-IO:** Sämtliche Operationen, die nicht direkt in Node.js stattfinden, blockieren die Ausführung Ihrer Applikation nicht. Der Grundsatz von Node.js lautet, alles, was die Plattform nicht direkt erledigen muss, wird an das Betriebssystem oder andere Applikationen ausgelagert. Die Applikation erhält damit die Möglichkeit, auf weitere Anfragen zu reagieren. Ist die Bearbeitung der Aufgabe erledigt, erhält der Node.js-Prozess eine Rückmeldung und kann die Informationen weiter verarbeiten.

- **Single-Threaded:** Eine typische Node.js-Applikation läuft in einem einzigen Prozess ab. Es gibt kein Multi-Threading, und Nebenläufigkeit ist zunächst nur in Form des bereits beschriebenen Nonblocking-IO vorgesehen. Sämtlicher Code, den Sie selbst schreiben, blockiert also potenziell Ihre Applikation. Sie sollten daher auf eine ressourcenschonende Entwicklung achten. Falls es dennoch erforderlich wird, Aufgaben parallel abzuarbeiten, bietet Ihnen Node.js hierfür Lösungen in Form des `child_process`-Moduls, mit dem Sie eigene Kindprozesse erzeugen können.

Damit Sie Ihre Applikation optimal entwickeln können, sollten Sie zumindest einen groben Überblick über die Komponenten und deren Funktionsweise haben. Die wichtigste dieser Komponenten ist die V8-Engine.

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 Lauf 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 mittlerweile den ECMAScript-Standard ECMA-262 in der fünften Version komplett und große Teile der sechsten 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 Bits 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.

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 Kompilervorgangs, 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 `first-name` als auch für `lastname` enthält und eine Transition mit dem Hinweis einfügt, 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. Listing 1.2 verdeutlicht diesen Zusammenhang.

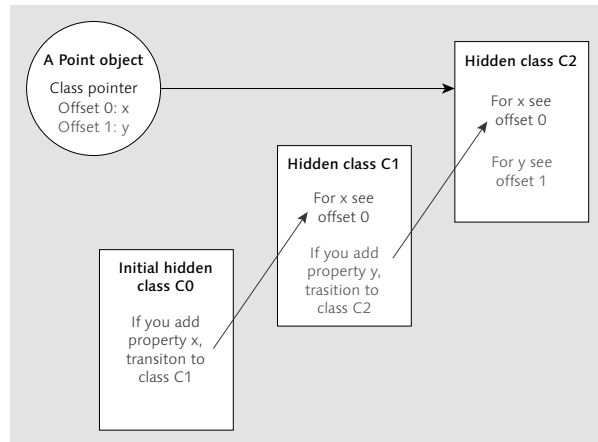


Abbildung 1.2 Hidden Classes in der V8-Engine
(»<https://developers.google.com/v8/design>«)

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 (JIT) in nativem 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 darüber, 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, 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-Funktionen 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. Wie der Eventloop funktioniert, können Sie in Abbildung 1.3 sehen.

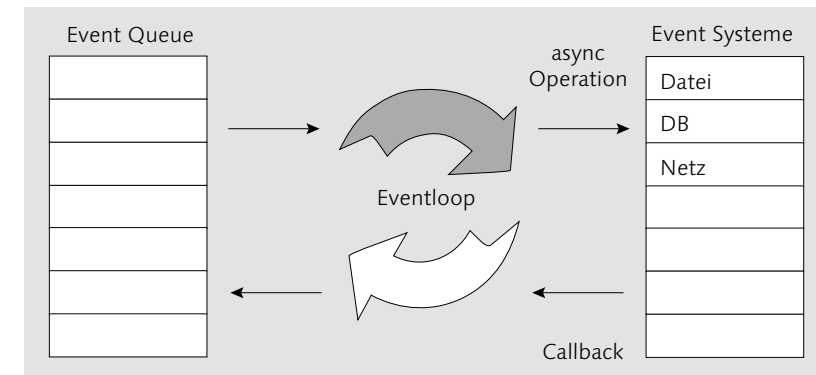


Abbildung 1.3 Der Eventloop

Der ursprüngliche Eventloop, der bei Node.js zum Einsatz kommt, basiert auf `libev`, einer 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 Windows-Systemen 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 Entwickler, 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 darunter liegende 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 seinen 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 des HTTP-Parsers. 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

Seit einigen Jahren ist Node.js nicht mehr aus der Webentwicklung wegzudenken. Dabei wird Node.js nicht nur zur Erstellung von Serverapplikationen verwendet, sondern ist auch Grundlage für eine Vielzahl von Hilfsmitteln vom Build-System wie Grunt oder Gulp bis hin zum Compiler für CSS-Präprozessoren. Der Erfolg der Plattform beruht auf einigen sehr einfachen Konzepten. Die Plattform basiert auf einer Sammlung von etablierten Bibliotheken, die zusammengefasst eine sehr flexible Arbeitsumgebung schaffen. Der Kern der Plattform wurde über die Jahre hinweg stets kompakt gehalten und bietet lediglich einen Satz an Grundfunktionalität. Für alle weiteren Anforderungen gibt es den Node Package Manager, über den Sie die verschiedensten Pakete in Ihre Applikation einbinden können.

Obwohl sich Node.js mittlerweile seit einigen Jahren in der Praxis bewährt hat, wird immer noch häufig die Frage gestellt: Kann ich Node.js bedenkenlos für meine Applikation einsetzen. In den Versionen vor 0.6 ließ sich diese Frage nicht guten Gewissens mit Ja beantworten, da die Schnittstellen der Plattform häufigen Änderungen unterlagen. Mittlerweile ist Node.js den Kinderschuhen entwachsen. Die Schnittstel-

len werden von den Entwicklern stabil gehalten. Für den Einsatz in Unternehmen wurde die LTS-Version geschaffen. Hierbei handelt es sich um eine Node.js-Version, die 18 Monate durch Updates unterstützt wird. Dies erhöht die Verlässlichkeit der Plattform und nimmt Unternehmen den Druck, immer auf die neueste Version zu aktualisieren.

Ein durchaus spannendes Kapitel in der Entwicklungsgeschichte war die Abspaltung von io.js. Der Grund dafür war, dass die Entwicklung von Node.js an Dynamik verlor und lange Zeit keine Neuerungen in die Plattform Einzug hielten. Dieses Ereignis war ein entscheidender Wendepunkt für die Entwicklung für Node.js. Die Node.js Foundation wurde gegründet, und die Verantwortung für die Entwicklung wurde von Einzelpersonen auf eine Gruppe übertragen. Daraufhin wurden die Releasezyklen und die Versionierung standardisiert, was den Verwendern der Plattform einerseits Verlässlichkeit und andererseits kontinuierliche Weiterentwicklung signalisiert.

Mit Ihrer Entscheidung, sich eingehender mit Node.js beschäftigen zu wollen, befinden Sie sich in guter Gesellschaft mit zahlreichen großen und kleinen Unternehmen weltweit, die Node.js mittlerweile strategisch für die Entwicklung von Applikationen einsetzen.

Kapitel 12

Single-Page-Webapplikationen mit Express.js und Angular.js

*Wir suchen niemals die Dinge, sondern das Suchen nach ihnen.
– Blaise Pascal*

Dieses Kapitel soll Ihnen in einer Art Workshop das Thema Single-Page-Webapplikationen näherbringen. Dazu werden zahlreiche Konzepte aus den vorangegangenen Kapiteln wiederholt und vertieft, aber auch einige Themen berührt, die erst in späteren Kapiteln behandelt werden. Das Ziel ist, dass Sie sowohl das Frontend als auch das Backend einer solchen Single-Page-Applikation erstellen und miteinander verbinden. Eine Single-Page-Webapplikation besteht im Gegensatz zu einer traditionellen Webapplikation nicht aus einer Sammlung einzelner Seiten, die über Hyperlinks verbunden sind, sondern aus einer Seite, auf der verschiedene Informationen per asynchrone Anfragen nachgeladen und Teile bei Bedarf ein- beziehungsweise ausgeblendet werden. Solche Applikationen wirken auf den Benutzer wesentlich responsiver und reagieren schneller und direkter auf die Eingabe eines Benutzers. Client und Server rücken bei solchen Applikationen enger zusammen. Die ausgetauschten Datenpakete werden kleiner und zugleich wird die Kommunikation intensiviert. Das hat wiederum zur Folge, dass die Daten einem Benutzer schneller und aktueller präsentiert werden können. Bei der Implementierung können Sie auf zahlreiche Bibliotheken und Frameworks zurückgreifen, die Ihnen viele Arbeitsschritte abnehmen.

12.1 Die Aufgabenstellung

Die Aufgabe, die es in diesem Kapitel zu lösen gilt, besteht darin, eine Verwaltungsoberfläche für eine Filmdatenbank zu erstellen. Um das Beispiel übersichtlich zu halten, wird auf eine Anmeldung und Benutzerverwaltung verzichtet. Außerdem beschränken Sie sich bei der Eingabe lediglich auf den Titel und das Erscheinungsjahr des Films. Die Applikation besteht also im Kern aus einer Liste von Datensätzen, der Sie neue Einträge hinzufügen und bestehende verändern oder löschen können. Mit dem Wissen, das Sie in diesem Kapitel erwerben, können Sie diese Applikation dann selbstständig um weitere Funktionen erweitern.

12.2 Setup

Der erste Schritt bei der Applikationsentwicklung ist die Installation der Kernkomponenten und die Erstellung einer Verzeichnisstruktur. Auf dieser Basis aufbauend, fügen Sie dann Schritt für Schritt Ihre eigenen Module hinzu.

12.2.1 Ordnerstruktur

Bei der MovieDB-Applikation handelt es sich um eine sehr überschaubare Anwendung. In der ersten Ausbaustufe verfügt diese serverseitig über einen Controller, ein Model und einen Router. Die Applikation wird über eine zentrale Index-Datei gestartet. Zusätzlich dazu gibt es ein Verzeichnis mit dem Namen *public*, das die Dateien beinhaltet, die für die Benutzer freigegeben sind. Den Einstiegspunkt in die Applikation für die Clientseite stellt die *index.html*-Datei im *public*-Verzeichnis dar. Sie ist damit quasi das Gegenstück der serverseitigen *index.js*-Datei. Die datenbankrelevanten Dateien liegen im Verzeichnis *db*. Wie Sie wissen, liegen die Node.js-Bibliotheken im *node_modules*-Verzeichnis. Darum müssen Sie sich also auch nicht weiter kümmern. Dieses Verzeichnis befüllen Sie in den nächsten Abschnitten mit Inhalt. Die Struktur für das Frontend ist ähnlich einfach aufgebaut wie das Backend. Unterhalb des *public*-Verzeichnisses finden Sie ein *app*-Verzeichnis, das den Quellcode Ihrer Applikation enthält. Die Templates liegen nicht wie gewohnt auf der Serverseite, sondern werden mit dem übrigen Clientcode ausgeliefert und befinden sich im *partials*-Verzeichnis. Das *lib*-Verzeichnis schließlich enthält die Clientbibliotheken. In Abbildung 12.1 sehen Sie einen Überblick über diese initiale Verzeichnisstruktur.

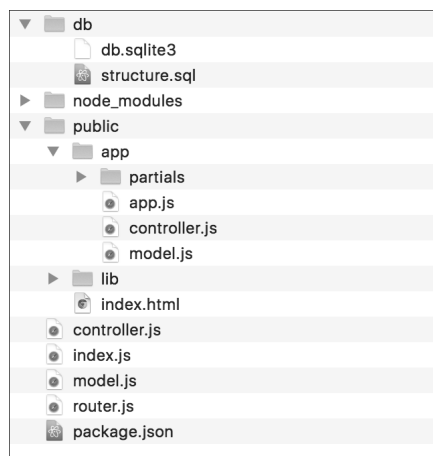


Abbildung 12.1 Verzeichnisstruktur

Sie müssen jetzt nicht alle Verzeichnisse und Dateien gleich anlegen. Im Zuge dieses Kapitels werden Sie die Applikation Schritt für Schritt generieren und dazu die Struk-

tur erstellen und den Quellcode erzeugen. Diese Struktur soll Ihnen zunächst als erster Überblick dienen, damit Sie einen Eindruck von der Aufteilung erhalten.

12.2.2 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 Sie das Kommando `sqlite3 db.sqlite3` ab. In Listing 12.1 sehen Sie die Kommandos, die benötigt werden, um die Datenbank aufzusetzen.

```
CREATE TABLE movies (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title VARCHAR,
  year INTEGER
);

INSERT INTO movies (title, year) VALUES
('Iron Man', 2008),
('The Avengers', 2012);
```

Listing 12.1 Setup der Datenbank

Mit den Kommandos aus Listing 12.1 erstellen Sie Ihre movieDB mit einer Tabelle und zwei Datensätzen.

12.2.3 Abhängigkeiten

Der erste Schritt bei der Entwicklung einer Node.js-Applikation ist immer die Erstellung einer *package.json*-Datei. Diese beschreibt Ihre Applikation und enthält zahlreiche wichtige Informationen für Benutzer und Entwickler. Im Falle der movieDB sind hier vor allem die zu installierenden Bibliotheken in Form von NPM-Paketen relevant. Sie sollten nun also diese Datei von NPM generieren lassen, indem Sie den Befehl `npm init` auf der Kommandozeile absetzen. Im nächsten Schritt installieren Sie die Bibliotheken, die Sie für die serverseitige Entwicklung verwenden. In diesem Fall sind das Express.js, der Body-Parser von Express.js und der SQLite3-Treiber für den

Datenbankzugriff. Die Installation erfolgt über das Kommando `npm install --save express body-parser sqlite3`.

```
{
  "name": "moviedb",
  "version": "0.0.1",
  "description": "Awesome movie database",
  "main": "index.js",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.15.0",
    "express": "^4.13.4",
    "sqlite3": "^3.1.1"
  }
}
```

Listing 12.2 »package.json«-Datei der movieDB-Applikation

In Listing 12.2 sehen Sie die *package.json*-Datei für die Applikation. Mit diesen Schritten haben Sie die Grundlage für die serverseitige Implementierung Ihrer Applikation geschaffen. Nun liegt es an Ihnen, die Komponenten zu einer funktionierenden Applikation zusammenzufügen.

12.2.4 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 Schreibarbeit ersparen. Das Frontend Ihrer Filmdatenbank soll auf dem bekannten Framework Angular.js aufbauen. Für dieses Beispiel wird noch die erste Version des Frameworks eingesetzt. Es handelt sich allerdings auch nicht um eine Einführung in Angular.js, stattdessen soll in diesem Kapitel nur demonstriert werden, wie Sie mit wenigen Schritten ein Node.js-Backend mit einem etablierten Frontend-Framework verbinden können. Statt Angular.js können Sie auch ein beliebiges anderes Framework wie beispielsweise React oder Ember.js verwenden. Durch das HTTP-Protokoll als Schnittstelle zwischen Client und Server sind Sie in der Auswahl der Werkzeuge sehr flexibel.

In diesem Beispiel kommen einige Komponenten von Angular.js zum Einsatz, die einer kurzen Erklärung bedürfen:

Die Architektur von Angular.js

Angular.js ist ein MVVM-Framework, also ein Strukturframework, das vor allem aus den Komponenten Model, View und ViewModel besteht. Die Models beinhalten die Daten der Applikation und die clientseitige Businesslogik. Die Views sind für die Darstellung verantwortlich. Angular setzt auf HTML mit spezifischen Auszeichnungen zur Darstellung und verzichtet auf eine eigene Template-Engine. Die ViewModels schließlich verbinden Models und Views und kümmern sich um die Synchronisierung der Daten. Damit wird das sogenannte Two-Way-Databinding ermöglicht. Dabei werden die Daten zwischen Model und View synchron gehalten. Wird eine Änderung an einer der beiden Stellen vorgenommen, wird sie auf der anderen Stelle reflektiert. Noch interessanter wird die Situation, wenn Sie mehrere Repräsentationen eines Models auf einer Seite haben. Angular.js sorgt dann dafür, dass die Daten an sämtlichen Stellen aktualisiert werden.

Controller

Der Controller wird zwar in MVVM nicht explizit erwähnt, spielt in Angular.js jedoch eine wichtige Rolle. Ein Controller ist eine Funktion, die das ViewModel kapselt. In einer solchen Controller-Funktion können Sie alle Funktionen definieren, auf die Sie in der View zurückgreifen möchten. Im weitesten Sinne ist der Controller der Ausführungskontext Ihrer View. Sie können mehrere Controller ineinander verschachteln, wobei die inneren Controller stets Zugriff auf die äußeren haben. Um die Komplexität Ihrer Applikation nicht unnötig zu erhöhen, sollten Sie eine zu tiefe Verschachtelung vermeiden. Das hat auch einen Performancehintergrund, da Angular.js das Two-Way-Databinding manuell durch Benachrichtigungen löst. Je mehr dynamische Ausdrücke sich in Ihrer Applikation befinden, desto performancekritischer ist dies für Ihre Applikation. Ab 2.000 bis 3.000 solcher als Watcher bezeichneten dynamischen Ausdrücke wird Ihre Applikation spürbar langsamer. Um nicht an diese Grenze zu stoßen, sollten Sie die Architektur Ihrer Applikation entsprechend auslegen und durch eine geschickte Aufteilung der Datenmenge dieses Problem umgehen.

Services in Angular.js

Zur Auslagerung mehrfach genutzter Programmlogik kennt Angular.js das Konzept von Services. Dabei handelt es sich meist um Funktionen, die Sie an mehreren Stellen einbinden können. Bei Angular.js gibt es drei unterschiedliche Arten von Services:

- **Service:** Diese Art von Services liefert ein Objekt zurück, das aus einer Konstruktor-Funktion erzeugt wurde. Durch die Erweiterung des Prototyps können Sie auf dem Objekt zusätzliche Methoden definieren.

- **Factory:** Eine Factory ist ein Objekt, das über Eigenschaften und Methoden verfügt.
- **Provider:** Die flexibelste Variante eines Service in Angular.js ist ein Provider. Diesen können Sie zum Startzeitpunkt einer Applikation konfigurieren. Ansonsten verhält er sich wie eine Factory.

Ein wichtiges Merkmal der Services, die intern aufeinander aufbauen, ist, dass es sich um Singletons handelt. Das bedeutet, dass Sie immer auf demselben Objekt arbeiten. Diesen Umstand können Sie sich zunutze machen und einen Service zur Kommunikation über Komponentengrenzen einer Applikation verwenden.

Dependency Injection

Angular.js ist ein Framework, das Sie bei der Erstellung von Tests für Ihre Applikation unterstützt. Die genaue Betrachtung dieser Facette der Applikationsentwicklung würde allerdings den Rahmen dieses Kapitels sprengen. Ein Teil des Frameworks, das die Testbarkeit von Angular.js-Applikationen gewährleistet, ist der Umgang mit internen Abhängigkeiten. Mit dem Dependency-Injection-Mechanismus von Angular.js können Sie Services und andere Komponenten Ihrer Applikation laden lassen, ohne dass Sie sich selbst um die Erzeugung des jeweiligen Elements kümmern müssen.

Neben der Dependency Injection verfügt Angular.js über ein sehr einfaches Modulsystem, das es Ihnen erlaubt, Module von Drittanbietern in Ihre Applikation einzubinden und deren Funktionalität durch Dependency Injection in Ihren Quellcode zu integrieren. Die definierten Schnittstellen können Sie dann für die Erstellung von Tests durch Stubs und Mocks ersetzen, sodass Sie einzelne Bestandteile Ihrer Applikation alleinstehend prüfen können.

Direktiven

Wenn Sie mit Angular.js arbeiten, kommen Sie früher oder später mit dem Begriff der Direktiven in Berührung. Eine Direktive ist, einfach ausgedrückt, ein Marker in Ihrer HTML-Struktur, den Angular.js durch etwas anderes ersetzt. Damit lassen sich der Wortschatz und der Funktionsumfang von HTML beträchtlich erweitern. Häufig verwenden Sie in Angular.js Direktiven, ohne dass Sie sich darüber bewusst sind. Die Angular.js-eigenen Direktiven erkennen Sie am vorangestellten `ng-`. So handelt es sich bei `ng-app` oder `ng-model` um Direktiven. Auch das `ui-sref`-Attribut, das Sie später noch im Zuge der Implementierung Ihrer Applikation verwenden werden, ist eine Direktive. In diesem Beispiel werden Sie allerdings keine eigenen Direktiven erstellen, sondern lediglich mit den bereits bestehenden arbeiten.

Angular-Router

Ein typischer Vertreter eines solchen externen Moduls, das recht häufig eingebunden wird, ist der Router von Angular.js. Der Router sorgt dafür, dass Sie in der Single-Page-Applikation über die Adressleiste Ihres Browsers navigieren können, ohne die Seite neu zu laden. Zu diesem Zweck wird entweder die Hash-Navigation oder Push-State verwendet. In beiden Fällen wird die aktuelle Adresse verändert, ein Neuladen der Seite jedoch verhindert. Bei der Hash-Navigation wird die Anker-Syntax verwendet, indem an die URL ein `#`-Symbol angehängt wird. Alle Werte, die diesem Zeichen folgen, führen nicht zu einem Seitenreload, sondern dienen ursprünglich zur Navigation zu lokalen Sprungmarken. Bei Push-State handelt es sich um eine JavaScript-API, mit der ein State in der Applikation aktiviert werden kann. Zusätzlich zu einer URL-Änderung, die nicht zum Seitenreload führt, können Sie der `history.pushState`-Methode weitere Informationen mitgeben. Wenn Sie die Möglichkeit haben, sollten Sie `pushState` verwenden, da Sie hier wesentlich flexibler sind. Anhand der übergebenen Informationen wird entschieden, welche Komponenten dargestellt werden und welche Informationen vom Server geholt werden müssen. Der am weitesten verbreitete Router für Angular.js ist der Angular-UI-Router, den Sie in Ihrem Beispiel einsetzen werden. Neben einfachen Routen bietet diese Bibliothek noch zahlreiche weitere Features wie beispielsweise verschachtelte oder benannte Routen.

ngResource

Die Kommunikation mit dem Server findet in einer Single-Page-Applikation im Normalfall asynchron statt. Zu diesem Zweck kommt das `xmlHttpRequest`-Objekt des Browsers zum Einsatz. Mit dieser nativen Browserschnittstelle werden Sie jedoch in den seltensten Fällen direkt zu tun haben. Angular.js selbst bietet den `$http`-Service, der diese Schnittstelle umschließt und komfortable Funktionen bietet. Noch einen Schritt weiter gehen Dienste wie `ngResource`, die Ihnen die Kommunikation in Form von REST mit dem Server ermöglichen. Der Vorteil dabei besteht darin, dass Sie nur sehr wenig Quellcode selbst schreiben müssen, um die Verbindung zum Server zu implementieren.

12.3 Die Applikation

Nach dieser kurzen theoretischen Einführung in Angular.js sind Sie nun bereit, Ihre Applikation Schritt für Schritt umzusetzen. Sie entwickeln zunächst die Anzeige der Liste von Datensätzen und schaffen damit einen ersten Querschnitt durch sämtliche Schichten der Applikation. Danach erweitern Sie diese erste Implementierung nach und nach zu einer vollwertigen Applikation.

12.3.1 Liste von Datensätzen

An erster Stelle steht Ihre Express.js-Applikation. Ohne sie können Sie keine Dateien zum Benutzer ausliefern und damit auch nichts anzeigen. Diese erste Implementierung besteht aus nur einer Datei, der *index.js*, die den Einstieg in Ihre Applikation darstellt. In Listing 12.3 sehen Sie den entsprechenden Quellcode.

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

var app = express();

app.use(express.static('public'));

app.listen(8080, function () {
  console.log('Application is listening on http://localhost:8080');
});
```

Listing 12.3 Erste Implementierung des Backends

Dieser Quellcode sorgt dafür, dass alle Inhalte des *public*-Verzeichnisses zum Benutzer ausgeliefert werden. Der Server wird an Port 8080 gebunden. Damit verfügen Sie über einen funktionierenden Webserver, der bereits den clientseitigen Quellcode ausliefern kann, den Sie im nächsten Schritt umsetzen werden.

Den Einstieg für Ihre Benutzer bildet die Datei *index.html* im *public*-Verzeichnis, deren Struktur Sie in Listing 12.4 finden.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>MovieDB</title>
  <script src="lib/angular.js"></script>
  <script src="lib/angular-ui-router.js"></script>
  <script src="lib/angular-resource.js"></script>
  <script src="app/app.js"></script>
</head>
<body ng-app="movieDB">
</body>
</html>
```

Listing 12.4 Die Struktur der »index.html«-Datei

Die zentrale Datei des Frontends ist recht übersichtlich. Im Kern besteht sie aus einer Reihe von *script*-Tags, die dafür sorgen, dass die JavaScript-Dateien geladen werden, und einem *body*-Tag mit dem Attribut *ng-app*, das dem Angular.js-Compiler angibt,

welche Elemente zu einer Applikation transformiert werden sollen. Bevor Sie nun Ihren Webserver starten und Ihre Applikation testen können, benötigen Sie noch den Quellcode von Angular.js. Diesen erhalten Sie direkt über die Webseite des Projekts www.angularjs.org. Für den weiteren Verlauf dieses Beispiels wird davon ausgegangen, dass Sie die Datei *angular.js* im Verzeichnis *public/lib* abgelegt haben. Starten Sie nun den Serverprozess und rufen die URL *http://localhost:8080* auf, sehen Sie nur eine weiße Seite. Um dies zu ändern, implementieren Sie im nächsten Schritt die clientseitige Logik für die Darstellung der Liste. Zu diesem Zweck benötigen Sie zunächst zwei weitere Bibliotheken: *angular-ui-router* und *ngResource*. Den Router können Sie unter <https://github.com/angular-ui/ui-router> herunterladen. Das *ngResource*-Paket finden Sie unter <https://docs.angularjs.org/api/ngResource>. Natürlich können Sie auch zur Installation der Bibliotheken einen Paketmanager wie JSPM, Bower oder NPM verwenden. Die Beschreibung deren Verwendung würde allerdings den Rahmen dieses Kapitels sprengen. Nachdem Sie sowohl die Datei *angular-ui-router.js* als auch *angular-resource.js* im *public/lib*-Verzeichnis gespeichert haben, können Sie die Datei *app.js* im *public*-Verzeichnis erstellen. Diese stellt den Kern Ihrer Angular.js-Applikation dar, in den Sie wiederum die einzelnen Module integrieren. Im ersten Schritt sollte Ihre *app.js*-Datei aussehen wie in Listing 12.5.

```
angular.module('movieDB', ['ui.router', 'ngResource'])
.config(configFn);

configFn.$inject = ['$stateProvider', '$urlRouterProvider'];

function configFn($stateProvider, $urlRouterProvider) {
  $urlRouterProvider.otherwise("/list");

  $stateProvider
    .state('list', {
      url: "/list",
      templateUrl: "app/partials/list.html",
      controller: 'ListController',
      controllerAs: 'listController'
    });
};
```

Listing 12.5 Einstieg in die Angular.js-Applikation

Mit der Anweisung *angular.module* erzeugen Sie Ihre Applikation, auf die Sie im *ng-app*-Attribut im HTML verweisen. In den eckigen Klammern geben Sie die externen Module an, die Sie in Ihre Applikation einbinden möchten. In diesem Fall sind das der Angular-UI-Router und *ngResource*. Die *config*-Funktion sorgt dafür, dass der Router zum Startzeitpunkt Ihrer Applikation mit den korrekten Werten versorgt wird. Eine Besonderheit ist die *\$inject*-Eigenschaft der Konfigurationsfunktion. Hier sehen Sie,

wie die Dependency Injection von Angular.js funktioniert. Sie übergeben ein Array von Objekten, auf die Sie zugreifen möchten, und Angular.js sorgt dafür, dass diese Objekte als Argumente an die Funktion übergeben werden. Der `$urlRouterProvider` und der `$stateProvider` sind Eigenschaften des Angular-UI-Routers, mit denen Sie die clientseitigen Routen festlegen können. Für die Umsetzung der Liste benötigen Sie die `/list-Route`, die mit dem `ListController` verbunden wird. Jede Route verfügt neben dem Controller über eine View, also ein HTML-Template, das dargestellt wird. Die Template-Datei für die Listendarstellung speichern Sie im Verzeichnis `public/partials` unter dem Namen `list.html`. Den Quellcode des Templates finden Sie in Listing 12.6.

```
<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Year</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="movie in listController.movies">
      <td>{{movie.title}}</td>
      <td>{{movie.year}}</td>
      <td>
        <a ui-sref="form({id: movie.id})">
          Edit
        </a>
      </td>
      <td>
        <a ui-sref="delete({id: movie.id})">
          Delete
        </a>
      </td>
    </tr>
  </tbody>
</table>

<a ui-sref="form()">New</a>
```

Listing 12.6 Template für die Listendarstellung

Bei dem `ng-repeat`-Attribut handelt es sich um eine Angular.js-Direktive, die dafür sorgt, dass der ausgezeichnete Block wiederholt wird. Auf diese Weise lassen sich einfach Schleifen in Templates realisieren. Die doppelten geschweiften Klammern ste-

hen für dynamische Ausdrücke, die der Angular.js-Compiler umwandelt und durch Zeichenketten ersetzt. Diese werden über die Laufzeit der Applikation stets aktuell gehalten. Die `ui-sref`-Attribute stammen vom Angular-UI-Router und dienen dazu, Routen innerhalb der Applikation zu aktivieren. Im Falle des Beispiels sind dies die Routen zum Erstellen, Editieren und Löschen von Datensätzen, die sogenannten CRUD-Operationen.

Neben dem Template benötigen Sie noch die Controller-Implementierung. Diese liegt in der Datei `public/app/controller.js`, deren Inhalt in Listing 12.7 abgebildet ist.

```
angular.module('movieDB')
  .controller('ListController', ListController);

ListController.$inject = ['dataFactory'];
function ListController (dataFactory) {
  this.movies = dataFactory.getAll();
}
```

Listing 12.7 Controller-Implementierung

Zunächst laden Sie die bereits existierende Instanz des `movieDB`-Moduls und fügen den neuen `ListController` hinzu. Im Anschluss sehen Sie, wie die Dependency Injection in Angular.js funktioniert. Sie übergeben Ihrem Controller ein `dataFactory`-Objekt, mit dem Sie die Datensätze vom Server laden können. Haben Sie den Quellcode Ihres Controllers fertiggestellt, müssen Sie dafür sorgen, dass dieser auch im Browser geladen wird, indem Sie in Ihrer `index.html`-Datei ein neues Script-Tag einfügen. Dieses fügen Sie am besten im Anschluss an das Tag, das die `app.js`-Datei lädt, ein. Listing 12.8 enthält den entsprechenden Ausschnitt der `index.html`-Datei.

```
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <script src="lib/angular.js"></script>
  <script src="lib/angular-ui-router.js"></script>
  <script src="lib/angular-resource.js"></script>
  <script src="app/app.js"></script>
  <script src="app/controller.js"></script>
  <script src="app/model.js"></script>
</head>
```

Listing 12.8 Erweiterung der »index.html«-Datei

Neben dem Laden der `controller.js`-Datei sehen Sie, dass außerdem die Datei `public/app/model.js` eingebunden wird. Diese enthält die `dataFactory`, die Sie bereits im Con-

troller verwenden, allerdings noch nicht implementiert haben. Daraus besteht nun auch die nächste Aufgabe. Listing 12.9 enthält den Quellcode für den Datenservice.

```
angular.module('movieDB')
.factory('dataFactory', dataFactory);

dataFactory.$inject = ['$resource'];

function dataFactory($resource) {
    return $resource(
        '/movie/:id',
        {id: '@id'},
        {
            getAll: {method: 'GET', isArray: true},
            create: {method: 'POST'},
            read: {method: 'GET'},
            update: {method: 'PUT'},
            delete: {method: 'DELETE'}
        }
    );
}
```

Listing 12.9 Die »dataFactory« der movieDB

Im Gegensatz zu den übrigen Stellen in der Applikation ist die `dataFactory` damit schon komplett implementiert und unterstützt sowohl lesende als auch schreibende Operationen. Die `dataFactory` besteht im weitesten Sinne aus einer Erweiterung des `$resource-Service`, der aus dem `ngResource-Modul` stammt. Die erste Information, die Sie hier übergeben, ist die URL, die auf der Serverseite verwendet werden soll. Das zweite Argument ordnet die dynamische Komponente, die ID in diesem Fall, zu. Das dritte Argument ordnet schließlich die verfügbaren Methoden des resultierenden `dataFactory-Objekts` den entsprechenden HTTP-Methoden zu. Mit diesen Anpassungen haben Sie alle Komponenten auf der Clientseite implementiert, sodass Sie sich nun der Serverseite widmen können.

Die Clientseite greift auf der Serverseite auf den URL-Pfad `/movie` zu. Um diese bedienen zu können, benötigen Sie einen Router für Ihre Express.js-Applikation. Den Quellcode aus Listing 12.10 speichern Sie in der Datei `router.js` im Wurzelverzeichnis Ihrer Applikation.

```
var controller = require('./controller.js');

module.exports = function(app) {
```

```
    app.get('/movie', controller.fetchAll);
};
```

Listing 12.10 Der Router der movieDB

Bevor Sie den Controller implementieren, der die Funktionalität hinter der Route zur Verfügung stellt, binden Sie den Router zunächst in die `index.js`-Datei Ihrer Applikation ein.

```
var express = require('express');
var router = require('./router');

var app = express();

app.use(express.static('public'));

router(app);

app.listen(8080, function () {
    console.log('Application is listening on http://localhost:8080');
});
```

Listing 12.11 Einbindung des Express.js-Routers

Listing 12.11 zeigt Ihnen, wie Sie den Router einbinden. Danach geht es nun daran, den Controller zu implementieren. Diesen speichern Sie in der Datei `controller.js`, die wie auch der Router im Wurzelverzeichnis der Applikation liegt und den Quellcode aus Listing 12.12 enthält.

```
var movie = require('./model');

function fetchAll(req, res) {
    movie.fetchAll().then(function success(rows) {
        res.send(rows);
    }, function failure(err) {
        res.send(err);
    })
}

module.exports = {
    fetchAll: fetchAll
};
```

Listing 12.12 Initiale Implementierung des Controllers

Auch der Controller ist noch nicht der Endpunkt des Backend-Workflows. Im Controller sorgen Sie dafür, dass die Daten über das Model aus der Datenbank ausgelesen werden. Die `fetchAll`-Methode des Models gibt ein `Promise`-Objekt zurück. Im Erfolgsfall senden Sie die ausgelesenen Datensätze an den Client.

Tritt ein Fehler auf, werden die Informationen an den Client geschickt. Um die Lesbarkeit des Controllers zu erhöhen, findet der Export gesammelt am Ende der Datei statt. Dafür trennen Sie den Export von der Implementierung der Funktion. Sie müssen jetzt nur noch das Model und damit die Datenbankverbindung implementieren. Den Quellcode des Models zeigt Ihnen Listing 12.13.

```
var sqlite3 = require('sqlite3');

var db = new sqlite3.Database('db/db.sqlite3');

function fetchAll() {
  return new Promise(function (resolve, reject) {
    db.all('SELECT * FROM movies', function (err, rows) {
      if (err) {
        reject(err);
      } else {
        resolve(rows);
      }
    });
  });
}

module.exports = {
  fetchAll: fetchAll
};
```

Listing 12.13 Abfrage der SQLite-Datenbank

Die Implementierung des Models greift etwas dem Kapitel 18 vor. Die Verbindung zur Datenbank besteht aus drei Teilen. Sie bauen die Verbindung auf, formulieren die Abfrage und gehen mit dem Ergebnis um. Damit die Verarbeitung des Ergebnisses komfortabler durchgeführt werden kann, setzen Sie hier Promises ein, die Sie bereits aus Kapitel 6 kennen. Auch im Model setzen Sie den Export für die Übersichtlichkeit an das Ende der Datei. Mit diesem Stand des Quellcodes können Sie jetzt Ihre Applikation starten und sich die Daten aus der Datenbank ansehen. Das Ergebnis sollte aussehen wie in Abbildung 12.2.

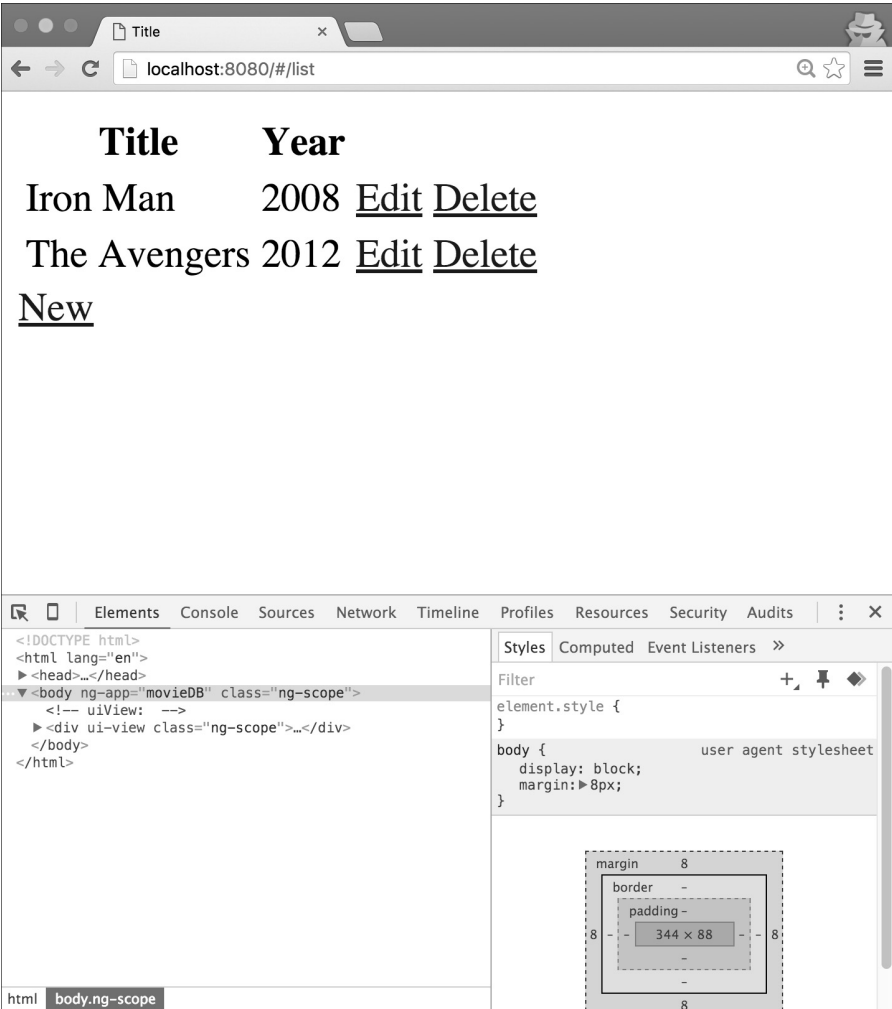


Abbildung 12.2 Die Liste der Datensätze

Datensätze zu betrachten reicht für die meisten Anwendungsfälle nicht aus. Aus diesem Grund implementieren Sie als Nächstes die Möglichkeit, neue Datensätze einzufügen.

12.3.2 Anlegen neuer Datensätze

Bei der Entwicklung eines neuen Features müssen Sie nicht zwingend im Frontend beginnen. Für das Anlegen neuer Datensätze beginnen Sie deshalb mit der Umsetzung des Express.js-Backends. Die erste Voraussetzung für die Erweiterung haben Sie schon erfüllt, indem Sie den `Body-Parser` installiert haben. Mit diesem Modul nehmen Sie die eingehenden Daten entgegen und übergeben sie an die Datenbank. Um

die Daten empfangen zu können, müssen Sie zunächst den Body-Parser einbinden und Ihre Applikation entsprechend konfigurieren. Zu diesem Zweck fügen Sie die beiden Zeilen aus Listing 12.14 in Ihre *index.js* im Wurzelverzeichnis Ihrer Applikation ein. Achten Sie dabei darauf, dass Sie den Aufruf von `app.use` an einer Stelle einfügen, an der das `app`-Objekt bereits existiert.

```
var bodyParser = require('body-parser');
app.use(bodyParser.json());
```

Listing 12.14 Einbindung von »body-parser«

Mit dieser Anpassung können Sie auf die Daten zugreifen, die sich im Body einer eingehenden Anfrage befinden. Zum Anlegen neuer Datensätze benötigen Sie außerdem eine neue Route. Ihre Benutzer werden dafür mit der HTTP-POST-Methode auf den URL-Pfad `/movie` zugreifen. Zusätzlich zur `get`-Route erweitern Sie nun also Ihren Router in der *router.js*-Datei um eine `post`-Route. Die Zeile aus Listing 12.15 sollten Sie direkt unter der bestehenden Route einfügen.

```
app.post('/movie', controller.create);
```

Listing 12.15 »POST«-Route

Der Router definiert lediglich die Schnittstellen, die einem Benutzer zur Verfügung stehen, und reicht die Anfrage weiter an den Controller. Innerhalb des Controllers definieren Sie zunächst die `create`-Funktion, die Sie dann über `module.exports` verfügbar machen. Wie der angepasste Quellcode aussieht, sehen Sie in Listing 12.16.

```
function create(req, res) {
  var movieData = {
    title: req.body.title,
    year: req.body.year
  };

  movie.insert(movieData).then(function(id) {
    res.send(JSON.stringify({id: id}));
  });
}

module.exports = {
  fetchAll: fetchAll,
  create: create
};
```

Listing 12.16 Erstellen von Datensätzen im Controller

Der Controller aus der Datei *controller.js* entnimmt der eingehenden Anfrage die Daten, die in die Datenbank geschrieben werden sollen, und sorgt dafür, dass die Antwort an den Benutzer versendet wird, sobald die Daten in die Datenbank geschrieben sind und die Promise erfüllt ist. Das Model, das Sie in der Datei *model.js* abgelegt haben, erweitern Sie um eine `insert`-Funktion, wie in Listing 12.17 zu sehen ist.

```
function insert(data) {
  return new Promise(function (resolve, reject) {
    db.run('INSERT INTO movies (title, year) VALUES (?, ?)',
      [data.title, data.year], function (err) {
        if (err) {
          reject(err);
        } else {
          resolve(this.lastID);
        }
      });
  });
}

module.exports = {
  fetchAll: fetchAll,
  insert: insert
};
```

Listing 12.17 Daten in die Datenbank schreiben

Wichtig beim Model ist auch, dass Sie die `insert`-Funktion über `module.exports` für andere Module verfügbar machen. Die `insert`-Funktion selbst gibt eine Promise zurück und sorgt dafür, dass die Daten über eine `INSERT`-Query korrekt in die Datenbank gespeichert werden. Weitere Informationen zum Umgang mit Datenbanken und der Verarbeitung von Werten finden Sie in Kapitel 18. Starten Sie jetzt Ihren Node.js-Prozess mit der Applikation neu, haben Sie die Möglichkeit, bereits Daten in die Datenbank einzufügen. Um die Bedienung für einen Benutzer angenehmer zu gestalten, sollten Sie sich nun um die Umsetzung des Frontends kümmern.

Der Router im Frontend verrichtet im Endeffekt ähnliche Arbeit wie im Backend: Er sorgt dafür, dass Anfragen des Benutzers an die korrekte Stelle geleitet werden. Um neue Datensätze anzulegen, benötigen Sie ein Formular. Ihr Router soll Ihre Anfragen also an eine Funktion weiterleiten, die ein Formular darstellt und mit den eingegebenen Werten umgehen kann. Zu diesem Zweck fügen Sie in der `config`-Funktion in der *public/app/app.js*-Datei einen neuen State zum `$stateProvider` hinzu.

```
$stateProvider.state('form', {
  url: "/form ",
  templateUrl: "app/partials/form.html",
```

```

    controller: 'FormController',
    controllerAs: 'formController'
  });

```

Listing 12.18 Route für das Formular

Wie Sie in Listing 12.18 sehen, geht der neue State davon aus, dass es sowohl ein Template für die neue Route als auch einen Controller gibt. Zunächst zum Formular, das Sie im Verzeichnis *public/app/partials* unter dem Namen *form.html* speichern. Den Quelltext dafür entnehmen Sie Listing 12.19.

```

<div>
  <label>Title</label>
  <input type="text" ng-model="formController.title">
</div>
<div>
  <label>Year</label>
  <input type="text" ng-model="formController.year">
</div>
<div>
  <button ng-click="formController.save()">save</button>
  <button ui-sref="list">cancel</button>
</div>

```

Listing 12.19 Formular zum Anlegen von Datensätzen

Die Eingabefelder werden mit der *ng-model*-Direktive über Two-Way-Databinding direkt mit dem Model im Controller verbunden. Eine Änderung des Werts wirkt sich direkt auf das Model aus. Ein Klick auf den Button ruft die *save*-Methode des Controllers auf, die das Speichern der Daten im Backend veranlasst. Zum Abschluss des Speichervorgangs fehlt Ihnen nun lediglich noch die Controller-Implementierung. Den Service haben Sie ja bereits im ersten Schritt fertiggestellt.

```

angular.module('movieDB')
  .controller('FormController', FormController)
FormController.$inject = ['$state', 'dataFactory'];
function FormController ($state, dataFactory) {
  this.title = '';
  this.year = '';

  this.save = function () {
    var data = {
      title: this.title,
      year: this.year
    };

```

```

    };
    dataFactory.create(data).$promise.then($state.go.bind($state, 'list'));

  }.bind(this);
}

```

Listing 12.20 Implementierung des FormControllers

Die notwendigen Erweiterungen der *public/app/controller.js*-Datei finden Sie in Listing 12.20. Die beiden Repräsentationen der Eingabefelder werden direkt auf der Controller-Instanz als Eigenschaften definiert. Ebenso verfahren Sie mit der *save*-Methode. Diese nutzt die *dataFactory*, um die Daten ans Backend zu senden. Nachdem der Speichervorgang erfolgreich war, wird über den Router wieder die Liste geladen. Starten Sie Ihre Applikation neu und nutzen den *New*-Link auf der Listenseite, werden Sie zu Ihrem Formular weitergeleitet. Diese Weiterleitung wirkt zwar, als ob die Seite neu geladen wird, im Hintergrund tauscht Angular.js jedoch lediglich Teile der Seite aus. Das Ergebnis sehen Sie in Abbildung 12.3.

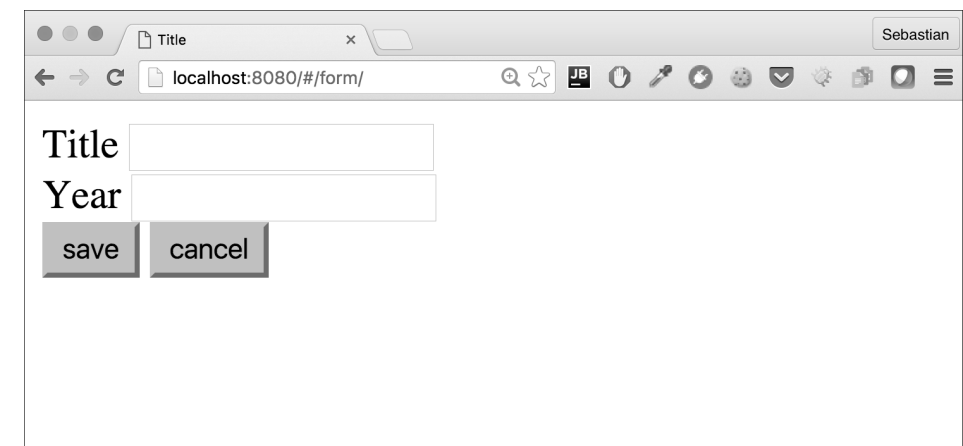


Abbildung 12.3 Formular zur Eingabe von Daten

Nachdem Sie sich schon im Frontend bewegen, können Sie Ihr Formular gleich dahingehend anpassen, dass es auch in der Lage ist, Datensätze zu editieren.

12.3.3 Datensätze verändern

Im Template der Liste haben Sie bereits Links vorgesehen, die zum Bearbeiten der Datensätze verwendet werden können. Zu diesem Zweck lassen Sie ebenfalls das Formular anzeigen, laden jedoch die Informationen eines Datensatzes. Das bedeutet, dass Sie in erster Linie Ihre Frontend-Route dahingehend anpassen müssen, dass sie

eine optionale ID eines Datensatzes verarbeiten kann. In der Datei *public/app/app.js* müssen Sie nur die *url*-Eigenschaft der *form*-Route etwas verändern, sodass sie folgenden Wert beinhaltet: *'/form/:id'*. Auf diesen ID-Wert können Sie dann in Ihrem Controller über den *\$stateParams*-Service des Routers zugreifen. Listing 12.21 zeigt Ihnen den angepassten Quellcode des Controllers.

```
FormController.$inject = ['$state', '$stateParams', 'dataFactory'];
function FormController ($state, $stateParams, dataFactory) {
    this.title = '';
    this.year = '';

    if($stateParams.id) {
        dataFactory.read({id: $stateParams.id}).$promise.then(function(movie)
        {
            this.title = movie.title;
            this.year = movie.year;
        }).bind(this));
    }

    this.save = function () {
        var data = {
            title: this.title,
            year: this.year
        };
        if ($stateParams.id) {
            data.id = $stateParams.id;
            dataFactory.update(data).$promise.then($state.go.bind($state, 'list'));
        } else {
            dataFactory.create(data).$promise.then($state.go.bind($state, 'list'));
        }
    }.bind(this);
}
```

Listing 12.21 Angepasster FormController

Haben Sie eine ID übergeben, verwenden Sie die *dataFactory*, um aktualisierte Informationen über den Datensatz vom Server abzurufen. Auch der Speichervorgang hängt davon ab, ob eine ID vorhanden ist oder nicht. Ist dies der Fall, nutzen Sie die *update*-Methode der *dataFactory*, ansonsten wie bisher die *create*-Methode. Mehr Anpassungen sind im Frontend nicht nötig, um auch das Editieren zu unterstützen. Damit können Sie sich wieder um die Serverseite kümmern. Da Sie beim Bearbeiten eines Datensatzes zunächst noch einmal die Informationen des Datensatzes vom

Server holen, um sicherzustellen, dass Sie mit der neuesten Version arbeiten, müssen Sie zwei Routen anlegen: eine zum Auslesen eines einzelnen Datensatzes und eine weitere zum Speichern der geänderten Daten. Die zusätzlichen Zeilen, die Sie in die Datei *router.js* einfügen müssen, sehen Sie in Listing 12.22.

```
app.get('/movie/:id', controller.fetch);
app.put('/movie/:id', controller.update);
```

Listing 12.22 Erweiterung des Routers für Updates

Die Erweiterung des Routers bedeutet, dass Sie auch Ihrem Controller in der Datei *controller.js* zwei Funktionen hinzufügen müssen, wobei die *fetch*-Funktion sehr ähnlich zur *fetchAll*-Funktion ist, und *update* und *create* ähneln einander ebenfalls. Sollten Sie duplizierten Quellcode einsparen wollen, können Sie die Funktionen auch zusammenlegen und parametrisieren. Die einfachere Variante mit dem duplizierten Quellcode finden Sie in Listing 12.23.

```
function fetch(req, res) {
    movie.fetch(req.params.id).then(function success(row) {
        res.send(row);
    }, function failure(err) {
        res.send(err);
    })
}

function fetch(req, res) {
    movie.fetch(req.params.id).then(function success(row) {
        res.send(row);
    }, function failure(err) {
        res.send(err);
    })
}

module.exports = {
    fetchAll: fetchAll,
    fetch: fetch,
    create: create,
    update: update
};
```

Listing 12.23 Anpassungen an der Datei »controller.js«

Der letzte Schritt der Implementierung besteht, wie auch schon zuvor, aus der Umsetzung des Models in der Datei *model.js*. Analog zum Controller müssen Sie auch hier die Funktionen *fetch* und *update* umsetzen. Das Prinzip dieser beiden Funktio-

nen ist ebenfalls nichts Neues für Sie. Zunächst erzeugen Sie eine Promise, danach generieren Sie die Abfrage an die Datenbank und warten, bis Ihnen das Ergebnis vorliegt. Je nachdem, ob die Abfrage erfolgreich war oder ein Fehler aufgetreten ist, lösen Sie die Promise auf oder weisen sie ab. Bei der Implementierung sollten Sie darauf achten, dass Sie nicht vergessen, beide Funktionen auch zu exportieren, da Sie ansonsten eine Fehlermeldung erhalten, weil nicht auf die Funktionen zugegriffen werden kann. Den Quellcode können Sie Listing 12.24 entnehmen.

```
function fetch(id) {
  return new Promise(function (resolve, reject) {
    db.get('SELECT * FROM movies WHERE id = ?', [id], function (err, rows) {
      if (err) {
        reject(err);
      } else {
        resolve(rows);
      }
    });
  });
}

function update(data, id) {
  return new Promise(function (resolve, reject) {
    db.run('UPDATE movies SET title = ?, year = ? WHERE id = ?',
    [data.title, data.year, id], function (err) {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
}

module.exports = {
  fetchAll: fetchAll,
  fetch: fetch,
  insert: insert,
  update: update
};
```

Listing 12.24 Erweiterung des Models um »fetch« und »update«

Mit diesen Anpassungen des Quellcodes sind Sie nun in der Lage, Ihre erstellten Datensätze auch nachträglich noch zu bearbeiten. Hierfür müssen Sie lediglich Ihre

Applikation neu starten, damit Node.js den angepassten Quellcode einliest. Verwenden Sie nun den *Edit*-Link eines Datensatzes in der Liste, werden die Informationen dieses Datensatzes in das Formular eingelesen, und Sie können Änderungen durchführen und diese speichern. Was nun noch fehlt, ist die Möglichkeit, bestehende Datensätze zu löschen.

12.3.4 Löschen von Datensätzen

Auch in diesem Fall liegt es bei Ihnen, wo Sie mit der Umsetzung beginnen möchten. Nachdem Sie die letzten Änderungen im Backend durchgeführt haben, können Sie hier auch gleich die Arbeit fortsetzen. Die Implementierung der Löschfunktion besteht wie bisher aus einer zusätzlichen Route, einer Controller-Funktion und der Anpassung des Models. Doch ein Schritt nach dem anderen.

Der Router, den Sie in der Datei *router.js* gespeichert haben, erhält noch eine weitere Zeile Quellcode, die die HTTP-Methode DELETE für den URL-Pfad */movie* unterstützt. Außerdem soll ein Benutzer die ID des zu löschenden Datensatzes übergeben können. Listing 12.25 enthält den Quellcode für die delete-Route.

```
app.delete('/movie/:id', controller.remove);
```

Listing 12.25 Die »delete«-Route

Eine häufig angewandte Best Practice besagt, dass Sie es vermeiden sollten, reservierte Wörter als Funktionsnamen zu verwenden. Also versehen Sie die Funktion des Controllers nicht mit dem Namen *delete*, sondern nennen sie *remove*. Den Quellcode dieser Funktion entnehmen Sie Listing 12.26. Und vergessen Sie auch hier nicht, die Funktion in Ihr *module.exports* aufzunehmen.

```
function remove(req, res) {
  movie.remove(req.params.id).then(function() {
    res.send(JSON.stringify(true));
  });
}

module.exports = {
  fetchAll: fetchAll,
  fetch: fetch,
  create: create,
  update: update,
  remove: remove
};
```

Listing 12.26 Die »remove«-Funktion des Controllers

Den Abschluss der serverseitigen Implementierung stellt die Umsetzung der `remove`-Funktion im Model dar. Hier setzen Sie wieder auf die bereits bekannte Struktur aus einer Kombination von Promise und Datenbankabfrage. Den entsprechenden Quellcode können Sie Listing 12.27 entnehmen. Diesen fügen Sie in die Datei *model.js* ein.

```
function remove(id) {
  return new Promise(function (resolve, reject) {
    db.run('DELETE FROM movies WHERE id = ?', [id], function (err) {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
}

module.exports = {
  fetchAll: fetchAll,
  fetch: fetch,
  insert: insert,
  update: update,
  remove: remove
};
```

Listing 12.27 »remove«-Funktion im Model

Der clientseitige Einstieg in die Löschoperation ist der *Delete*-Link in der Listendarstellung. Für diesen benötigen Sie eine Erweiterung der Angular.js-Route, die in der Datei *public/app/app.js* liegt. Auch hier rufen Sie wieder die `state`-Methode des `$stateProvider`-Objekts auf. Diese Route ist allerdings etwas einfacher als die bisherigen, da Sie weder ein Template noch die `controllerAs`-Angabe benötigen. Den Quellcode zeigt Ihnen Listing 12.28.

```
$stateProvider.state('delete', {
  url: "/delete/:id",
  controller: 'DeleteController'
});
```

Listing 12.28 »delete«-Route im Frontend

Wie bereits erwähnt, benötigen Sie nur noch die Implementierung des Controllers, um Datensätze löschen zu können.

```
angular.module('movieDB')
.controller('DeleteController', DeleteController);

DeleteController.$inject = ['$state', '$stateParams', 'dataFactory'];
function DeleteController ($state, $stateParams, dataFactory) {
  dataFactory.delete({id: $stateParams.id}).$promise.then(function() {
    $state.go('list');
  });
}
```

Listing 12.29 Der DeleteController

Mit dem Quellcode aus Listing 12.29, den Sie in die Datei *public/app/controller.js* einfügen müssen, schließen Sie die Implementierung Ihrer Single-Page-Applikation ab. Wenn Sie jetzt den Node.js-Prozess noch einmal neu starten, um sämtliche Änderungen zu aktivieren, können Sie Datensätze anlegen, sie anzeigen und bearbeiten und auch wieder löschen. Sämtliche dieser Schritte werden in der angeschlossenen Datenbank persistiert.

12.4 Zusammenfassung

Single-Page-Applikationen vermitteln dem Benutzer den Eindruck, dass sie eher mit einer Applikation und weniger mit einer Webseite arbeiten. Statt kompletter Seitenreloads werden nur Teile der Seite ausgetauscht. Im Idealfall übermitteln Sie sämtliche statischen Daten Ihrer Webapplikation nur einmal zum Client. Verwendet ein Benutzer Ihre Applikation, tauscht er nur noch dynamische Informationen mit dem Server aus. Dies geschieht in den meisten Fällen im JSON-Format, da diese Daten am schnellsten und einfachsten vom Browser verarbeitet werden. Basiert Ihr Webserver auf Node.js, entsteht durch das Datenformat kein größerer Zusatzaufwand, da die V8-Engine die JSON-Encodierung nativ beherrscht. Diese Form der Informationsübertragung spart im Gegensatz zur Übermittlung vollständiger HTML-Seiten zum einen Übertragungsbandbreite, und zum anderen ermöglicht es dem Browser, die statischen Inhalte im lokalen Cache vorzuhalten, was die Ladezeit zusätzlich verbessert. Wann immer Sie die Möglichkeit haben, sollten Sie eine Single-Page-Applikation einer Multi-Page-Applikation vorziehen, da die Nutzerführung und das Bedienerlebnis wesentlich angenehmer für einen Benutzer ist.

Bei der Entwicklung einer solchen Applikation sollten Sie sowohl client- als auch serverseitig auf eines der zahlreichen bewährten Frameworks zurückgreifen, da Ihnen dieses viel Arbeit abnimmt. In diesem Kapitel haben Sie gesehen, dass eine Kombination aus Express.js und Angular.js eine gute Wahl ist. Sowohl die Auslieferung der Daten als auch die Übertragung von Informationen ist weitestgehend standardisiert,

sodass die Umsetzung von Standardaufgaben wenig Schreibarbeit für Sie bedeutet. Und selbst für exotischere Anforderungen existieren in den meisten Fällen schon Module oder Plugins, die Sie entweder direkt verwenden können oder nur leicht modifizieren müssen.

Auf einen Blick

1	Grundlagen	25
2	Installation	47
3	Ein erstes Beispiel	67
4	Node-Module	83
5	NPM	109
6	ECMAScript 6	129
7	Node auf der Kommandozeile	155
8	HTTP	173
9	Express.js	201
10	Template-Engines	229
11	REST Server	259
12	Single-Page-Webapplikationen mit Express.js und Angular.js	275
13	Echtzeit-Webapplikationen	301
14	Asynchrone Programmierung	327
15	Streams	355
16	Arbeiten mit Dateien	383
17	Socket-Server	405
18	Anbindung von Datenbanken	427
19	Qualitätssicherung	453
20	Sicherheitsaspekte	491
21	Skalierbarkeit und Deployment	519

Inhalt

Geleitwort des Fachgutachters	19
Vorwort	21

1 Grundlagen 25

1.1 Die Geschichte von Node.js	26
1.1.1 Die Ursprünge	26
1.1.2 Die Geburt von Node.js	27
1.1.3 Der Durchbruch von Node.js	28
1.1.4 Node.js erobert Windows	29
1.1.5 io.js – der Fork von Node.js	29
1.1.6 Node.js wieder vereint	30
1.2 Vorteile von Node.js	30
1.3 Einsatzgebiete von Node.js	31
1.4 Das Herzstück – die V8-Engine	32
1.4.1 Das Speichermodell	33
1.4.2 Zugriff auf Eigenschaften	34
1.4.3 Maschinencodgenerierung	36
1.4.4 Garbage Collection	37
1.5 Bibliotheken um die Engine	39
1.5.1 Eventloop	40
1.5.2 Eingabe und Ausgabe	41
1.5.3 libuv	42
1.5.4 DNS	43
1.5.5 Crypto	44
1.5.6 Zlib	44
1.5.7 HTTP-Parser	45
1.6 Zusammenfassung	45

2 Installation 47

2.1 Installation von Paketen	48
2.1.1 Linux	49

2.1.2	Windows	52
2.1.3	OS X	57
2.2	Kompilieren und Installieren	63
2.3	Zusammenfassung	65

3 Ein erstes Beispiel 67

3.1	Der interaktive Modus	67
3.1.1	Generelle Benutzung	67
3.1.2	Zusätzliche REPL-Befehle	69
3.1.3	Speichern und Laden im REPL	70
3.1.4	Kontext der REPL	71
3.1.5	REPL-Historie	71
3.1.6	REPL-Modus	72
3.2	Die erste Applikation	72
3.2.1	Ein Webserver in Node.js	73
3.2.2	Erweiterung des Webserver	76
3.2.3	Erstellen einer HTML-Antwort	78
3.2.4	Dynamische Antworten generieren	79
3.3	Zusammenfassung	81

4 Node-Module 83

4.1	Modularer Ansatz	83
4.2	Stabilität	86
4.2.1	Stabilitätsindex	86
4.2.2	Verfügbare Module	87
4.3	Basis-Module	90
4.3.1	Globale Objekte	90
4.3.2	Utility	93
4.3.3	Events	95
4.3.4	OS	97
4.3.5	Path	97
4.4	Eigene Module erstellen und einbinden	98
4.4.1	Eigene Module in Node.js	98
4.4.2	Eigene Node.js-Module	99

4.4.3	Verschiedene Datentypen exportieren	101
4.4.4	Das modules-Modul	102
4.4.5	Der Modulloader	104
4.4.6	Die »require«-Funktionalität	107
4.5	Zusammenfassung	108

5 NPM 109

5.1	Pakete suchen	109
5.1.1	Pakete installieren	110
5.1.2	Installierte Pakete anzeigen	114
5.1.3	Pakete verwenden	115
5.1.4	Pakete aktualisieren	116
5.1.5	Pakete entfernen	117
5.1.6	Die wichtigsten Kommandos im Überblick	118
5.1.7	Der Aufbau eines Moduls	119
5.1.8	Eigene Pakete erstellen	123
5.1.9	NPM Scripts	125
5.2	Werkzeuge für NPM	126
5.2.1	Node License Finder	126
5.2.2	Sinopia	127
5.3	Zusammenfassung	128

6 ECMAScript 6 129

6.1	Ein kurzer Rückblick	129
6.1.1	Die wichtigsten ES5 Features	130
6.2	Node.js und ECMAScript 6	131
6.3	String- und Array-Erweiterungen	132
6.4	Scoping	133
6.4.1	Gültigkeitsbereiche in JavaScript	133
6.4.2	Module Scope in Node.js	134
6.4.3	Block Scoping	134
6.4.4	Konstante Werte	136
6.4.5	Function in Blocks	136

6.5	Klassen	137
6.5.1	Eine Klasse definieren	137
6.5.2	Vererbung	139
6.6	Template Strings	140
6.6.1	Standard Template Strings	140
6.6.2	Tagged Template Strings	141
6.7	Collections	142
6.7.1	Map	142
6.7.2	Set	143
6.7.3	WeakMaps und WeakSets	143
6.8	Arrow Functions	144
6.9	Generators	145
6.10	Promises	146
6.11	Symbols	148
6.12	Typed Arrays	150
6.13	Spread-Operator	150
6.14	Rest Parameters	151
6.15	Destructuring	151
6.16	Binary und Octal Literals	152
6.17	Erweiterung der Object Literals	152
6.18	»Object.assign«	153
6.19	»new.target«	153
6.20	Zusammenfassung	153

7 Node auf der Kommandozeile 155

7.1	Grundlagen	155
7.1.1	Aufbau	156
7.1.2	Ausführbarkeit	157
7.2	Der Aufbau einer Kommandozeilenapplikation	158
7.2.1	Datei und Verzeichnisstruktur	159
7.2.2	Paketdefinition	160
7.3	Zugriff auf Ein- und Ausgabe	161
7.3.1	Ausgabe	161

7.3.2	Eingabe	162
7.3.3	Optionen und Argumente	164
7.3.4	Signale und Exit Codes	166
7.4	Werkzeuge	169
7.4.1	Commander.js	169
7.4.2	Chalk	171
7.5	Zusammenfassung	172

8 HTTP 173

8.1	Der Webserver	173
8.1.1	Das »Server«-Objekt	173
8.1.2	Server-Events	175
8.1.3	Das »Request«-Objekt	177
8.1.4	Das »Response«-Objekt	180
8.1.5	Ausliefern von HTML	184
8.1.6	Ausliefern von statischen Dateien	184
8.1.7	Umgang mit »GET« und »POST«	185
8.1.8	Dateiupload	186
8.2	HTTP-Client mit Node.js	187
8.2.1	Die »get«-Methode	188
8.2.2	Der »http.Agent«	188
8.2.3	Die Anfrage-Optionen	189
8.2.4	Die Klasse »ClientRequest«	191
8.2.5	Die Antwort des Servers	193
8.2.6	HTML-Parser	195
8.3	Umgang mit URLs	196
8.4	Sichere Kommunikation mit HTTPS	198
8.5	Zusammenfassung	199

9 Express.js 201

9.1	Aufbau	201
9.2	Installation	203
9.3	Grundlagen	204

9.3.1	»Request«	204
9.3.2	»Response«	205
9.4	Setup	206
9.4.1	Verzeichnisstruktur	207
9.4.2	Router	210
9.4.3	Controller	211
9.4.4	Models	212
9.4.5	View	213
9.5	Routing	215
9.5.1	HTTP-Methoden	215
9.5.2	Muster in Routen	216
9.5.3	Variablen in Routen	217
9.5.4	Routing-Callbacks	218
9.5.5	»Route«-Objekt	219
9.5.6	Das »Router«-Objekt	220
9.6	Middleware	220
9.6.1	Eigene Middleware	221
9.6.2	Morgan	222
9.6.3	Serve-static	223
9.6.4	Body-Parser	224
9.7	Zusammenfassung	227
10	Template-Engines	229
10.1	Eine eigene Template-Engine	230
10.2	Template-Engines in der Praxis – Jade	231
10.2.1	Installation und grundsätzliche Verwendung	232
10.2.2	Jade und Express.js	233
10.2.3	Variablen in Jade	236
10.2.4	Die Besonderheiten von Jade	238
10.2.5	Bedingungen und Schleifen	242
10.2.6	Extends und Includes	245
10.2.7	Mixins	248
10.2.8	Compiling	250
10.3	Handlebars	251
10.3.1	Installation und erstes Beispiel	251
10.3.2	Bedingungen und Schleifen	253
10.3.3	Partials	254

10.3.4	Eigene Helper	255
10.3.5	Integration in Express.js	256
10.4	Zusammenfassung	258
11	REST Server	259
11.1	»GET« – lesender Zugriff	261
11.2	»POST« – Anlegen neuer Ressourcen	265
11.3	»PUT« – Aktualisierung bestehender Daten	267
11.4	»DELETE« – Löschen vorhandener Daten	269
11.5	»Accept«-Header	271
11.6	Zusammenfassung	273
12	Single-Page-Webapplikationen mit Express.js und Angular.js	275
12.1	Die Aufgabenstellung	275
12.2	Setup	276
12.2.1	Ordnerstruktur	276
12.2.2	Die Datenbank	277
12.2.3	Abhängigkeiten	277
12.2.4	Clientbibliotheken	278
12.3	Die Applikation	281
12.3.1	Liste von Datensätzen	282
12.3.2	Anlegen neuer Datensätze	289
12.3.3	Datensätze verändern	293
12.3.4	Löschen von Datensätzen	297
12.4	Zusammenfassung	299
13	Echtzeit-Webapplikationen	301
13.1	Die Beispielapplikation	302
13.2	Setup	303

13.3 Websockets	308
13.3.1 Die Serverseite	309
13.3.2 Die Clientseite	312
13.3.3 Userliste	314
13.3.4 Logout	319
13.4 Socket.IO	320
13.4.1 Installation und Einbindung	321
13.4.2 Socket.IO-API	321
13.5 Zusammenfassung	325
 14 Asynchrone Programmierung	 327
14.1 Grundlagen asynchroner Programmierung	327
14.1.1 Das child_process-Modul	330
14.2 Externe Kommandos asynchron ausführen	332
14.2.1 Die »exec«-Methode	332
14.2.2 Die »spawn«-Methode	335
14.3 »fork«	337
14.4 Das cluster-Modul	341
14.4.1 Der Masterprozess	342
14.4.2 Die Workerprozesse	346
14.5 Promises in Node.js	349
14.5.1 Flusssteuerung mit Promises	351
14.6 Zusammenfassung	353
 15 Streams	 355
15.1 Einleitung	355
15.1.1 Was ist ein Stream?	355
15.1.2 Wozu verwendet man Streams?	356
15.1.3 Welche Streams gibt es?	357
15.1.4 Stream-Versionen in Node.js	357
15.1.5 Streams sind EventEmitter	358
15.2 Readable Streams	358
15.2.1 Einen Readable Stream erstellen	359

15.2.2 Die Readable-Stream-Schnittstelle	360
15.2.3 Die Events eines Readable Streams	361
15.2.4 Fehlerbehandlung in Readable Streams	362
15.2.5 Methoden	363
15.2.6 Piping	363
15.2.7 Readable-Stream-Modi	364
15.2.8 Wechsel in den Flowing Mode	364
15.2.9 Wechsel in den Paused Mode	365
15.2.10 Eigene Readable Streams	365
15.2.11 Beispiel für einen Readable Stream	366
15.2.12 Readable-Shortcut	368
15.3 Writable Streams	369
15.3.1 Einen Writable Stream erstellen	369
15.3.2 Die Writable-Stream-Schnittstelle	370
15.3.3 Events	370
15.3.4 Fehlerbehandlung in Writable Streams	372
15.3.5 Methoden	372
15.3.6 Schreiboperationen puffern	373
15.3.7 Flusssteuerung	374
15.3.8 Eigene Writable Streams	375
15.3.9 Writable-Shortcut	376
15.4 Duplex Streams	376
15.4.1 Duplex Streams im Einsatz	377
15.4.2 Eigene Duplex Streams	377
15.4.3 Duplex-Shortcut	378
15.5 Transform Streams	378
15.5.1 Eigene Transform Streams	379
15.5.2 Transform-Shortcut	380
15.6 Gulp	380
15.6.1 Installation	381
15.6.2 Beispiel für einen Build-Prozess mit Gulp	381
15.7 Zusammenfassung	382
 16 Arbeiten mit Dateien	 383
16.1 Synchron und asynchrone Funktionen	383
16.2 Existenz von Dateien	384

16.3	Dateien lesen	385
16.4	Fehlerbehandlung	390
16.5	In Dateien schreiben	391
16.6	Verzeichnisoperationen	394
16.7	Weiterführende Operationen	398
16.7.1	»watch«	400
16.7.2	Zugriffsberechtigungen	401
16.8	Zusammenfassung	402

17 Socket-Server 405

17.1	UNIX-Sockets	406
17.1.1	Zugriff auf den Socket	408
17.1.2	Bidirektionale Kommunikation	410
17.2	Windows Pipes	413
17.3	TCP-Sockets	414
17.3.1	Datenübertragung	416
17.3.2	Dateiübertragung	417
17.3.3	Flusssteuerung	418
17.3.4	Duplex	420
17.3.5	Pipe	420
17.4	UDP-Sockets	421
17.4.1	Grundlagen eines UDP-Servers	422
17.4.2	Beispiel zum UDP-Server	424
17.5	Zusammenfassung	426

18 Anbindung von Datenbanken 427

18.1	Node.js und relationale Datenbanken	428
18.1.1	MySQL	428
18.1.2	SQLite	434
18.2	Node.js und nicht relationale Datenbanken	439
18.2.1	Redis	440
18.2.2	MongoDB	444

18.3	Datenbanken und Promises	449
18.4	Datenbanken und Streams	450
18.5	Zusammenfassung	451

19 Qualitätssicherung 453

19.1	Unittesting	453
19.1.1	Verzeichnisstruktur	454
19.1.2	Unittests und Node.js	455
19.1.3	Tripple-A	455
19.2	Assertion Testing	456
19.3	Jasmine	459
19.3.1	Installation	460
19.3.2	Konfiguration	460
19.3.3	Tests in Jasmine	461
19.3.4	Assertions	462
19.3.5	Spies	464
19.3.6	»beforeEach« und »afterEach«	465
19.4	nodeunit	465
19.4.1	Installation	465
19.4.2	Ein erster Test	466
19.4.3	Assertions	468
19.4.4	Gruppierung	468
19.4.5	»setUp« und »tearDown«	470
19.5	Praktisches Beispiel von Unittests mit nodeunit	470
19.5.1	Der Test	471
19.5.2	Die Implementierung	472
19.5.3	Der zweite Test	472
19.5.4	Verbesserung der Implementierung	473
19.6	Statische Codeanalyse	474
19.6.1	ESLint	475
19.6.2	PMD CPD	478
19.6.3	Plato	480
19.7	Node.js Debugger	482
19.7.1	Navigation im Debugger	483
19.7.2	Informationen im Debugger	484
19.7.3	Breakpoints	486

19.8 Node Inspector	487
19.8.1 Installation	488
19.8.2 Features	488
19.9 Debugging in der Entwicklungsumgebung	489
19.10 Zusammenfassung	489

20 Sicherheitsaspekte 491

20.1 Filter Input und Escape Output	492
20.1.1 Filter Input	492
20.1.2 Black- und Whitelisting	492
20.1.3 Escape Output	493
20.2 Absicherung des Servers	494
20.2.1 Benutzerberechtigungen	495
20.2.2 Single-Threaded-Ansatz	496
20.2.3 Denial of Service	499
20.2.4 Reguläre Ausdrücke	500
20.2.5 HTTP-Header	501
20.2.6 Fehlermeldungen	504
20.2.7 SQL-Injections	504
20.2.8 »eval«	506
20.2.9 Method Invocation	508
20.2.10 Überschreiben von Built-ins	510
20.3 NPM-Sicherheit	511
20.3.1 Berechtigungen	512
20.3.2 Qualitätsaspekt	512
20.3.3 NPM Scripts	513
20.4 Schutz des Clients	514
20.4.1 Cross-Site-Scripting	515
20.4.2 Cross-Site-Request-Forgery	516
20.5 Zusammenfassung	518

21 Skalierbarkeit und Deployment 519

21.1 Deployment	519
21.1.1 Einfaches Deployment	520
21.1.2 Dateisynchronisierung mit rsync	521
21.1.3 Die Applikation als Dienst	522
21.1.4 node_modules beim Deployment	525
21.1.5 Applikationen mit dem Node Package Manager installieren	526
21.1.6 Pakete lokal installieren	527
21.2 Toolunterstützung	528
21.2.1 Grunt	528
21.2.2 Gulp	533
21.2.3 NPM	534
21.3 Skalierung	534
21.3.1 Kindprozesse	535
21.3.2 Loadbalancer	538
21.3.3 Node in der Cloud	541
21.4 pm2 – Prozessmanagement	543
21.5 Zusammenfassung	544
 Index	 545

Index

--prod	120	Accessor-Methoden	131
__dirname	90	Account	542
__express	235, 256	Account-Informationen	505
__filename	90	ACK-Paket	415
_flush-Methode	380	Act	456
_read	420	AddressBook	262
_read-Methode	369	addTrailers	184
_transform-Methode	379	Administrator	74, 113
_write	420	Administratorberechtigungen	507
_write-Methode	376	Administratorkonto	495
.bash_history	71	Adressbuch	261
.bom	61	Adresse	422
.break	70	affectedRows	433
.clear	70	afterEach	465
.eslintrc	475	AJAX Long Polling	320
.exit	69	Aktualisierung	269
.handlebars	257	Aktualisierungsmechanismus	116
.hbs	252	Alert-Fenster	515
.load	70	Algorithmus	539
.msi-Paket	53	all-Methode	438
.node_repl_history	71	andReturn	465
.save	70	Anfragen	534, 539
.spec.js	461	Anfrage-Optionen	189
/etc/profile	50, 65	Angriffspotenzial	506
/opt	64	Angriffsszenarien	491
/usr/local	64	Angular.js	278
/usr/local/bin	61	Angular-Router	281
/usr/local/lib/node_modules	118	angular-ui-router	283
=>	144	Anmeldeformular	307
\$http	281	Antipattern	241, 475
\$inject	283	Antwort	193
\$stateProvider	284	Antwort-Header	182
\$urlRouterProvider	284	Anzahl von Kindprozessen	498
A		Anzeige	214
Aarhus	32	appendFile	394
Abhängigkeiten	111, 113, 120, 277, 351, 470, 525, 527	append-Methode	443
Abkürzungen	173	application/json	272
Abmelden	307	Applikation	72
abort-Methode	193	Applikationen mit dem Node Package Manager installieren	526
Absicherung des Servers	494	Applikationsentwicklung	276
Absoluter Pfad	104	Applikationslebenszyklus	519
Abstraktionsschicht	448	apt-get	51
Accept-Header	178, 271	Arbeitsspeicher	440, 535
accept-Methode	311	Argumente	156, 157, 164
		arguments	151
		Arrange	456

Arrange, Act, Assert	455, 460
Array-Methoden	132
Arrow Functions	144
Assert	456
Assertions	453, 456, 457, 458, 462, 468
assert-Modul	455
Asynchron	441
Asynchrone Operationen	147
Asynchrone Programmierung	327
asyncQuery	449
Atomare Operationen	212
Aufbau	156
Aufbau eines Moduls	119
Ausfallsicherheit	540
Ausführbarkeit	157
Ausführungsberechtigungen	158
Ausführungskontext	145
Ausgabe	461
Ausgabekanal	162
Aushandlung	320
Auslagerung	327, 497
Auslieferung	321
Austauschbarkeit	85
Austauschformat	271
Authentifizierung	201
autoAcceptConnections	311
Autovervollständigung	69
Azure	53
B	
Backend-Workflow	288
Backtraces	484
Base64-Codierung	418
Basis-Module	90
Basis-Template	245
Basisverzeichnis	303
Baumstruktur	114
BDD	459
Bedienbarkeit	301
Bedingungen	242, 253
Beenden des Workerprozesses	344
Befehl	156
Befehlsausführung	332
beforeEach	465
Behavior-Driven Developments	459
Benennungsschema	107
Benutzerberechtigungen	495
Benutzer-ID	93, 401
Benutzername	196
Berechnung	497
Berechtigungen	383, 512
Betriebssystem	47, 342
Bezeichner	148
Bibliotheken	39, 275
Bidirektionale Kommunikation	341, 410, 536
Bilder	184
bin	159
Binärpaket	53, 434
Binärversion	53
Binary Literals	152
Binary-Paket	57
bin-Verzeichnis	122
Blacklist	492
Block Scoping	134
block-Auszeichnung	246
Blockierende Operationen	328
Body-Parser	204, 224, 289, 306
Bower	283
Breaking Changes	86
Breakpoints	486
Browserübergreifend	320
BSON-Format	444
Buffer	91
Buffer-Objekt	77, 333, 388
Bugtracker	122
Build-Prozess	155, 380
Build-System	380
Buildtool	528
Built-ins	510
Businesslogik	279

C

C++	47
Cache	106
Caching	37, 230
Caching-Mechanismus	107
Callback-Funktion	329, 384
C-Ares	44
catch	351
cert.pem	198
Chakra	32
Chalk	171
change	400
Character Set	263
chat-Subprotokoll	311
checkAuth-Middleware	307
cheerio	195
child_process	330, 535
ChildProcess	330, 337
chmod	401

chown	402
Chrome	32
Chrome-Entwicklertools	488
Chunk	76, 180, 183, 192, 266
class	137
Client	410
Clientbibliotheken	278
clientError	176
ClientRequest	191
Clientseitige Sicherheitsmechanismen	494
Client-Server-Ansatz	301
close	387
close-Methode	309, 320
Closure Scope	134
Cloud Computing	541
Cloud Storage	541
Cloudbasierte Lösung	519
cluster.worker	346
Cluster-Ereignisse	345
Clustering	444
cluster-Modul	341, 538
cmd.exe	56
Codefragmente	337
Codequalität	474
Codierung	194, 417
Collections	142, 445
Commander.js	169
Commit-Statistik	513
compile	231, 252
compileFile	250
Compiler	63
Compiling	250
config-Funktion	283
configure	63
Connect.js	202
connection	176
Connection-Pooling	191
connections-Objekt	318
console	91
const	136
Content-Type	76, 77, 78, 205, 263, 272
content-type	182, 195
Controller	207, 211, 279
controllerAs	298
Cookie-Parser	204
cookie-session	306
Copy-and-Paste-Detection	478
Copy-and-Paste-Detector	478
cork-Methode	373
CORS	517
CouchDB	427, 527
CPD	478
createClient	441
createConnection	431
createServer	174, 407
createSocket	423
createWriteStream	370
Crockford, Douglas	475
Cross-Origin Resource Sharing	517
Cross-Site-Request-Forgery	516
Cross-Site-Scripting	491, 515
CRUD	213, 427
CRUD-Operationen	285
crypto	44
Crypto-Modul	44
CSRF	516
CSS	224
cssmin-Plugin	530
CSS-Optimierung	530
csurf-Middleware	518
cURL	173, 216, 260
Cygwin	29, 52, 539
D	
Daemons	523
Dahl, Ryan	26
Darstellung	213
Datagramme empfangen	425
Date-Header	183
Dateien	383
Dateien beobachten	400
Dateien lesen	385
Dateien schreiben	391
Dateiendung	108
Dateigröße	389
Datei-Handle	385, 388, 391
Datei-Handle schließen	389
Dateiinformationen	388, 398
Dateisynchronisierung mit rsync	521
Dateisystembasierte Kommuni- kation	406, 413
Dateisystemberechtigungen	401, 409
Dateisystembrowser	397
Dateisystemoperationen	384
Dateitransfer	417
Dateiübertragung	417
Dateiuploads	173, 186, 201
Datenbank	277, 427, 504
Datenbank-Objekt	436
Datenbankstruktur	430, 435
Datenbanktreiber	427

Datenbankverbindung	288
Datenbasis	265
Datenfluss	418
Datenhaltung	212
Datenmengen	177, 534
Datenpakete	183, 275
Datenquelle	355
Datensätze	212
Datensätze aktualisieren	433, 438, 443, 447
Datensätze anlegen	441
Datensätze auslesen	432, 437, 442, 446
Datensätze entfernen	433, 439, 443, 447
Datenservice	286
Datenspeicher	142
Datenströme	335, 355, 405
Datenstrukturen	241
Datentypen	101
Datenübermittlung	341
Datenübertragung	416
db-mysql	430
DDOS-Attacken	491
Debugger	453, 482
debugger-Statements	487
Debugging in der Entwicklungs- umgebung	489
Debug-Modus	483, 484
Deflate	44
Defragmentierung	39
Deinstallieren	52, 56, 61
DELETE	269, 297
del-Methode	443
Denial of Service	499
dependencies	120
Dependency Injection	280, 470
Deployment	203, 519, 526
Deployment-Prozess	528
Deployment-Unterstützung	534
Deprecated	86
--depth	115
describe-Methode	460
Desktop-Applikation	301
destroy-Methode	348
Destructuring	151
devDependencies	120, 529
devDependency	381
dgram-Modul	422
Dienst	522
Dienste unter UNIX	523
Dienste unter Windows	524
Dienstverwaltung	524
Direktiven	280
disconnect	331, 344, 348, 349
Django	201
DNS	39, 43, 422
dns.lookup	44
Dokumentation	85, 512
Dokumentenorientierte Datenbank	444
Dokumentieren	454
done-Methode	467, 470
DOS-Attacken	500
Download-Zahlen	512
drain-Event	183, 374, 419
Drei-Wege-Handshake	415
Duplex	420
Duplex Stream	357, 420
Duplex Streams	376
Duplex-Shortcut	378
Duplikate	478, 479
Durchsatz	535
Dynamische Antworten	79
Dynamische Ausdrücke	285
Dynamische Elemente	237
Dynamische Webapplikationen	73

E

EACCES	391
each	438
each-Helper	254
each-Schleife	244
EADDRINUSE	175, 410
Echtzeitfähige Webapplikation	301, 302
Eclipse	477, 489
ECMA-262	25
ECMAScript	30
ECMAScript 2015	130
ECMAScript 6	95, 129
Eigene Duplex Streams	377
Eigene Matcher	464
Eigene Middleware	221
Eigene Module	91, 98
Eigene Pakete	123
Eigene Readable Streams	365
Eigene Transform Streams	379
Eigene Writable Streams	375
Eigenschaften	34
Eigenschaften des Request-Objekts	177
Eigentümer	402
Ein- und Ausgabe	161
Einbinden von Plugins	530
Eindeutige ID	446
Eindeutige Kennung	346

Einfaches Deployment	520
Einfärben	171
Eingabe	358
Eingabe und Ausgabe	41
Eingabedatei	334
Einrückung	239
Einsatzgebiete	31
Einstiegsdatei	211
Einstiegspunkt	103, 105, 120, 124
Elternklasse	140
Elternmodul	103
Elternprozess	538
Ember.js	278
emit	358
emit-Methode	323
Encoding	183, 359, 390
end-Event	180, 417
end-Methode	187, 192
ENOENT	391
entities	516
Entkoppeln	208
Entwicklungsumgebung	489
Ereignisse	97, 348
Erfolg	181
Erfolgsfall	270
Erfolgsmeldung	472
error-Event	362, 372
Erweiterbarkeit	478
ES5 Features	130
Escape Output	492, 493
Escapesequenz	393
Escaping	432, 437, 506
eslintConfig	475
--es-staging	151
Estimated errors	481
eval	506
Eventarchitektur	350
EventEmitter	95, 266, 358, 393
Eventgetriebene Architektur	40, 175
Eventgetriebener Ansatz	310, 343, 441
EventListener	96
Eventloop	40
Events	95
Events von Writable Streams	370
events.EventEmitter	40
Eventschnittstellen	370
Exceptions	330, 362, 398
execFile-Methode	334
exec-Methode	332
Existenz von Dateien	384
Exit Codes	166, 167
exit-Event	344
expect-Methode	460
Experimental	86
Export	212, 288
exports	91, 99, 103
exports-Objekt	466
Express.js	111, 201, 233, 256, 303, 304, 480, 503
Extends	245
extends	140
extensions	108
Externe Kommandos	332
Externer Dienstleister	541

F

Factory	280
Fehler	181
Fehleranfälligkeit	136
Fehlerbehandlung	362, 372, 390
Fehlerlevel	393
Fehlermeldungen	296, 330, 504
Fehler-Objekt	333, 362, 384, 391, 398, 432, 457
Fehlersuche	482
Fehlschläge	461
Fehlschlagender Test	472
Fielding, Roy	260
Filedescriptor	399
Filesystem Hierarchy Standard	49
Filesystem-Modul	42
Filmdatenbank	275
Filter Input	492
Filterprozess	492
find-Methode	446
finish-Event	371
Firewall	500
First-Class-Funktionen	40
Flash Sockets	320
Flowing Mode	361, 364
Flusssteuerung	351, 374, 418
foreman	542
fork	330, 337, 345
format	94, 197
formidable	186
Formular	225, 291
for-of	142
Fragmentierung	414
Frameworks	275
Fremdschlüssel	428, 434
Frontend	282, 322
Frontend-Route	293
Frontend-Testing	465

fs.access	384
fs.stat	384
fs-Modul	417
FTP	521
Function Scope	134
Function-Konstruktor	507
Funktionsebene	470
Funktionsreferenzen	219
G	
Garbage Collection	33, 37
Garbage Collector	133, 143
<i>Mark-and-Sweep</i>	38
<i>Scavenge Collector</i>	38
GCC	39
GCC Runtime Library	39
Gegenmaßnahme	515
Generator-Funktion	145
Generator-Objekt	145
Generators	145
Gesamtpaket	180
Geschichte	26
GET	185
GET – lesender Zugriff	261
getaddrinfo	44
get-Methode	437, 442
Getter/Setter	139
Gewichtung	540
Git Publishing	542
Github-Repository	85
Global Scope	134
globalAgent	188
Globale Installation	120
Globale Objekte	90
Globale Scope	508
Globaler Kontext	71, 90
Globle Installation	112
Große Applikationen	209
Große Datenmengen	450
Größenbeschränkung	425
Grundlagen eines UDP-Servers	422
Grunt	528
grunt	115
Grunt ausführen	530
grunt watch	533
grunt.initConfig	532
grunt-cli	115
grunt-contrib-watch-Plugin	532
Gruntfile	115
Gruntfile.js	528, 529
Grunt-Plugins	529
Grunt-Tasks	532
Gruppen	469
Gruppen-ID	93, 401
Gruppierung und Wiederholung	500
Gulp	380, 533
gulpfile.js	381
Gulp-Konfiguration	382
Gzip	44

H

Halstead-Formel	481
Handlebars	230, 251
Handlebars-Modul	252
Handshake	415, 421
Haproxy	539
--harmony_destructuring	152
Hashes	444
Hash-Navigation	281
Häufige Ausführung	474
Haupt-Thread	329, 497
Header	177
Header-Felder	414
Header-Informationen	191
Helmet	503
Helper	255
Heroku	541
Heroku Toolbelt	541
Hidden Class	33, 35, 37
highWaterMark	359
Hilfsprogramme	155
Historie	70
Hochperformante Bibliotheken	519
Hohe Frequenz	425
Hoisting	135, 212
Holowaychuck, TJ	201
Hostname	175, 196
HTML	184, 224, 271
HTML5	308
HTML-Block	248
HTML-Injection	515
HTML-Knoten	243
HTML-Parser	195
HTML-Strukturen	173, 184, 195
HTML-Tags	232, 238
HTML-Views	208
HTML-Zeichenkette	78
HTTP	173
http_parser	39
http.Agent	188

http.STATUS_CODES	182
HTTP-Body	76
HTTP-Client	187
HTTP-Header	76, 178, 405, 501
HTTP-Kommandos	260
HTTP-Methoden	178, 215, 271, 286
http-Modul	202, 259, 405
HTTP-Parser	39, 45
HTTP-Protokoll	45
HttpProxyModule	540
HTTPS	198
https	173
HTTP-Server	74, 173, 310, 344
HTTP-Statuscode	76
httpVersion	178
Hyperlinks	275

I

I/O-Operationen	43, 496
IBM DB2	427
ID	239
if-Statement	242
In Progress	132
Includes	245, 254
index.jade	235
indexAction	211
Index-Datei	210
Information	181
Informationen im Debugger	484
ini-Parser	386
--init	475
init-Option	460
inotify	400
INSERT	432
insert	446
inspect	94
Installation	47, 429, 435, 440, 444, 460, 466
<i>Zielverzeichnis</i>	55
Installation und Einbindung	321
Installer-Paket	57
Interaktiver Modus	67
Internes Modul	104
Interpreter	156
Interprozesskommunikation	331
io.js	29
IOCP	43
io-Objekt	322
IP-Adresse	74
IP-Hash	541
IPv4	175, 423
IPv6	174, 423
isMaster	343
ISO/IEC-Standard	130
Isoliert	453
Issues	513
isWorker	343
Iteratoren	149
it-Methode	460
J	
Jade	230, 231, 303, 304
Jade-Elemente	240
Jade-Template	313
JägerMonkey	32
Jasmine	459
JavaScript	224
JavaScript-Engines	
<i>Chakra</i>	32
<i>JägerMonkey</i>	32
<i>Nitro</i>	32
<i>V8</i>	32
JIT	33, 36
Joyent	28
jQuery	195, 313
jsconf.eu	28
JSLint	475, 531, 532
JSLint-Plugin	531
JSON	131, 225
JSON.parse	314
JSON-Format	263
JSON-Objekt	192
JSONP Polling	320
JSPM	283
Just-in-time-Kompilierung	33, 36
K	
Kapselung	261
Kategorien der Statuscodes	182
Keine Ausgabe	457
Keine Verbindung	421
Kette	219, 356
Kettenglieder	378
key.pem	198
Key-Value-Store	142, 440
kill	167, 331
Kindklasse	140
Kindprozess	330, 331, 338, 535, 538
Kindprozess beenden	339
Klartext	184

Klassen	137, 213, 239
Klassendefinition	138, 366
Kleine Applikationen	207
Kommando	69, 156, 333, 483
Kommandoprompt	69
Kommandos im Überblick	118
Kommandozeile	155, 337, 457, 475
Kommandozeilenoptionen	169, 479
Kommandozeilenwerkzeuge	113, 123, 155
Kommunikation	405
Kommunikationsverbindung	308
Kompilieren und installieren	63
Kompilervorgang	35
Komplexität	480
Konfiguration	460, 530
Konfigurationsdatei	539
Konfigurationsobjekt	310, 342
Konfigurationsoptionen	64
Konfigurationsscript	525
Konfliktauflösung	537
Konstante Werte	136
Konstruktor	138, 262
Konstruktor-Funktion	90, 138
Konsumenten	365
Kontinuierlicher Datenfluss	356
Kopieren	520
kqueue	400
Kurzschreibweise	119, 368, 376, 378
Linux	47, 49
Linux Binaries	49
Listen	282, 444
listen	174
listening-Event	344, 349
listen-Methode	407
list-Kommando	484
LiveScript	25
Lizenzbedingungen	58
Lizenzen	126
Lizenzinformationen	54
Loadbalancer	538, 540
Loadbalancing	342
Location	265
Location-Header	267
Lock-Dateien	536
Locked	87
Locking-Mechanismus	536
Logger	221, 391
Logikblöcke	241
Logout	319
Logout-Prozess	319
Lokale Installation	112
Loopback-Schnittstelle	74
Loose Coupling	99
Löschen	269
Lose Kopplung	99

M

Ladeoperationen	107
lastID	437
Least Connections	541
Lebenszyklus	133, 360
Lesbarer Datenstrom	420
Lesebuffer	420
lessc	113
let	135
lib	159
libeio	42
libev	41, 42
libevent	41
libevent2	41
libuv	42, 84
lib-Verzeichnis	123
License-Datei	127
Lines of code	481
Link-Shortener	515
Lint errors	481
Mac OS X	47
magic	72
Maintainability	480
Major-Version	121
make	63
make install	64
Makefile	63
Manuelle Eingriffe	454
Map	142
MariaDB	429
Mark-and-Sweep	38
Marker	230
Markup	229
Markup-Sprache	251
Maschinencode	33, 36
Massendaten	424
Masterprozess	342
Master-Slave-Verbund	429
Maßzahlen	474
Matcher	463

maxconn	539
Mehrere Clients	413
Mehrere parallele Prozesse	537
Memory Leaks	488
Menge	143
message-Event	311, 340, 349, 423
Messwert	366
Metainformationen	71, 177, 179, 521
Method Invocation	508
Methoden	138
Methoden eines Writable Streams	372
Methodenausführung	508
METHODS	215
Metriken	453
Middleware	201, 220, 306
Middleware-Funktionen	220
Middleware-Komponenten	204
Minifizieren	380, 531
Minor-Version	121
MIT-Lizenz	203
Mittelgroße Applikationen	208
Mixins	248, 255
Model	206, 212, 279, 288
Modularer Ansatz	83
Modularisierung	84
Modulcache	107
Module	73, 87
Module Scope	134
module.exports	92
module.filename	103
module.id	103
Modules-Modul	102
Modulloader	99, 104
Modulsuche	104
Modulsystem	72, 519
MongoClient-Klasse	445
MongoDB	277, 427, 444
MongoDB-Client	445
Mongoose	448
Mongrel	27
Monolithische Architektur	84
Morgan	222
MSSQL	427
Multi-Client-Chat	302
Muster	216
MVC	206
MVVM	279
MySQL	260, 277, 427, 428
MySQL-Clientbibliotheken	429
MySQL-Protokoll	429

N

Nachrichten	309, 340, 348
Nachrichtenkörper	177, 178
Nachrichtentypen	323
Namenskonvention	413
Navigation	395
Navigation im Debugger	483
Navigationshilfe	98
Nebenläufigkeit	327
Negieren	458
Negierungen	458, 463
net.Socket	411
net-Modul	405
Net-Server	344
Network Time Protocol	422
Netzwerkbasierende Sockets	414
Netzwerkverbindung	414
Neue Datensätze anlegen	431, 436
Neustart	521
new.target	153
next-Funktion	219
ng-	280
Nginx	540
ng-model	292
ngResource	281, 283
Nicht relationale Datenbanken	439
Nitro	32
Node Bindings	84
Node in der Cloud	541
Node Inspector	487
Node License Finder	126
Node Package Manager	109
node_modules	104, 111, 114
node_modules beim Deployment	525
NODE_PATH	104, 105
NODE_REPL_HISTORY	71
NODE_REPL_HISTORY_SIZE	71
NODE_UNIQUE_ID	343
node.exe	53
Node.js	
<i>Deinstallation</i>	56
<i>Installation</i>	47
<i>Versionierung</i>	47
Node.js Foundation	30
Node.js-Prozess	74
node-linux	524
node-mac	524
node-static	184
nodeunit	465, 530

nodeunit-Kommando	466
node-windows	524
Nonblocking I/O	27
Normalisierung	97
NoSQL	440
NPM	28, 53, 109, 283
npm	
dependencies	116
globale Module	117
Versionsnummer	117
Werkzeuge	126
npm init	116, 160, 277, 303
npm install	110, 161
npm link	161
npm list	114
npm publish	125
NPM Scripts	125, 513
npm Scripts	
Install	125
Publish	125
Restart	125
Start	125
Stop	125
Test	125
Uninstall	125
Version	125
npm search	110
npm uninstall	118
npm update	117
npm useradd	124
npmjs.org-Repository	119
NPM-Module	427
NPM-Paket	321, 541
NPM-Repository	525
Nutzdaten	265
Nutzeraccount	124

O

Object Literals	152
Object Mode	367, 368
Object.assign	153
Object.create	34
Objektstruktur	94
Octal Literals	152
Oktalnotation	401
on	358
Onlinedokumentation	439, 448
online-Event	344, 348
open	387
Open-Source-Projekte	526
OpenSSL	44
openssl	84, 198
Optimierung	106, 519, 531
Optionen	156, 157, 164
Options-Objekt	223
Oracle DB	427
Ordnerstruktur	276
origin-Eigenschaft	311
OS	97
OS X	57
OSI-Modell	405
Overhead	415
owasp	504

P

package.json	105, 119, 123, 159, 277, 303, 513, 526, 528
Paketdefinition	160
Pakete aktualisieren	116
Pakete anzeigen	114
Pakete entfernen	117
Pakete installieren	110
Pakete lokal installieren	527
Pakete suchen	109
Pakete verwenden	115
Paketmanager	48, 51, 283
Paketverwalter	109
Parallele Operationen	352
Parallelisierbarkeit	85, 229
Parallelisieren	449
Parallelisierung	538
params-Objekt	218
parse-Methode	187
Parsen	169
Partials	229, 254
Partitionieren	429
Passwort	196
Patch Level	121
PATH	158
Path	97
Paused Mode	361, 364
pause-Methode	195, 365, 418
peerDependency	112
Pending	147
Performanceaspekt	534
Persistierung	427
Pfadangabe	97
Pfade	383
Pfadtrenner	97, 383
Pipe	420, 421

pipe-Methode	363, 421, 451
Pipe-Symbol	163
Piping	163, 363, 371
pkg-Datei	57
pkgutil	61, 62
Platform as a Service	541
Plato	480
plato-output	480
Platzhalter	121, 250
Plugins	380, 477, 529
pm2	543
PMD CPD	478
Polyfills	131
Port	196, 422
Port 5858	483
POSIX-Systeme	440
POST	185, 290
POST – Anlegen neuer Ressourcen	265
Postman	216
POST-Requests	224
Precompiling	230
preferGlobal	160
preinstall	514
Primzahlen	340
printf	94
Prioritäten	391
private	160
process	92
process.argv	165, 339
process.getuid	93
process.setuid	93
process.stderr	162
process.stdin	162
process.stdout	162
Procfile	542
Programmierschnittstelle	321
Programmlogik	242
Promise	288
Promise.all	148, 352
Promise.race	148, 352
Promise-Konstruktor	351
Promises	146, 349, 449
Protokoll	196, 301
Protokollwechsel	302
Prototyp	34, 137
Provider	280
Proxy	127
proxy_pass	541
Proxy-Server	527
Prozess	332
Prozess-ID	167, 347
Prozessmanager	543
Prozessor	535
Prozessorkern	342
Prozessorressourcen	535
Prüfsumme	414
Pseudo-Array	151
Publish-Subscribe	40
Publizieren	526
Puffern	373
Pug	231
pull-Streams	357
push- und pull-Streams	358
Push-State	281
push-Streams	357
Push-Technologien	265
PUT – Aktualisierung bestehender Daten	267

Q

Q	146
Qualitätsaspekt	512
Qualitätsmaßstäbe	454
Qualitätssicherung	453
Quellcode einschleusen	511
Quelldateien	63
query	80, 197
quit	441
JUnit	459

R

React	278
read	387
readable	360
Readable Stream	162, 357, 358, 451
Readable Stream erstellen	359
Readable und Writable Streams	377
readable-Event	408
Readable-Shortcut	368
Readable-Stream-Modi	364
Readable-Stream-Schnittstelle	360
ReadDirectoryChangesW	400
Read-Eval-Print Loop	67
readFile	330, 390
readFileSync	330
readline	163
README.md	512
Readme.md	127
read-Methode	408, 417
Redis	427, 440

Redis-Client	440
Redis-Server	441
ReDOS	500
Referenz	143, 318
Referenzielle Integrität	434
Regelkonfiguration	476
registerPartial	255
registerTask	532
Registry	524
Regular Expression Denial of Service	500
Reguläre Ausdrücke	217, 500
Rejected	147
reject-Methode	311
Rekursiv	522
Relationale Datenbanken	427, 428
Releasezyklen	131
Relevanter Code	454
remove	297
remove-Methode	447
rename	400
render	232
renderFile	232
REPL	67, 484
REPL verlassen	69
REPL-Befehle	69
REPL-Historie	71
Replizieren	527
REPL-Modus	72
Repository	109, 512, 526
Request	177, 202, 204
Requestbasiert	176
Request-Body	179
Request-Objekt	76, 80, 177
require	103, 107, 116
resolve	107
Resolved	147
Response	202, 205
Response-Body	183
Response-Bodys	218
Response-Objekt	76, 180
Responsiv	275
Ressource	413
Ressourcen	339
Ressourcenforderungen	538
Ressourcenzugriff	536
REST	259, 281
Rest Parameter	151
REST-Service	259
resume	195
resume-Methode	418
rimrafall	511, 513
Round Robin	539, 541
Route	204, 304
Routendefinition	210
Route-Objekt	219
Router	210, 291
Router-Objekt	220
Routing	215
Routing-Callbacks	218
Routing-Funktionen	205
rowid	435
RSVP	146
rsync	522
rsync-Kommandozeilenbefehl	522
Ruby on Rails	201
Rückgabe	470
Rückgabewert	168, 341
Rückkanal	301
Runlevel	523
run-Methode	436, 438, 439
run-script	126

S

safe	
<i>true</i>	446
Sandbox	491
Scavenge Collector	38
Schadcode	238, 493, 510, 515
Schadenersatzforderungen	492
Schema	428
Schemalos	440
Schleifen	242, 253
Schleifendurchläufe	240
Schlueter, Isaac	29, 109
Schlüsseldatei	199
Schlüssel-Wert-Paar	237
Schlüter, Isaac	28
Schnelle Tests	454
Schnittmenge	116
Schnittstelle	98, 101, 210
Schreibbarer Datenstrom	420
Schreibberechtigung	495
Schreibpuffer	419
Schriftfarbe	171
Schutz des Clients	514
Scoping	133
scp	520
Secure-Shell-Protokoll	520
Seiteneffekte	470
Seitenreload	301
SELECT-Abfrage	438

semver	47
send-Methode	340, 424
Sequenznummer	414, 422
Server	410
server.js	73
Server-Events	175
Serverimplementierung	321
Serverkomponente	310, 320
Server-Objekt	173
Serverprozess	156, 408, 434
ServerRequest	270
ServerResponse	270
Serve-static	223
Service	279
session	307
Session Hijacking	491
Session-Daten	319
Session-Handling	201
Set	143
setBreakpoint	486
setEncoding	194, 408
setHeader	182
set-Methode	442
setNoDelay	193
setSocketKeepAlive	193
Settled	147
Setup	276
setUp	470
setup-Event	344
setupMaster	342
Sharding	444
Shebang	115, 159
Shell-Kommandos	156
Shellscripte	337
Shipping	131
Shortcut-Notation	451
Shortcut-Schreibweise	380
Sichere Websockets	308
Sicherheit	491
Sicherheitsaspekte	491
Sicherheitsmechanismen	506
Sicherheitsproblem	511
Sicherheitsrisiko	495, 521
Sicherungsmaßnahmen	421
SIGHUP	344
SIGINT	70, 167
SIGKILL	167, 331
Signale	166
SIGTERM	331
Simulieren	522
Single-Page-Applikationen	275, 301
Single-Threaded-Ansatz	40, 327, 496
Singletons	280
Sinopia	127, 527
size-Eigenschaft	389
Skalierbarkeit und Deployment	519
Skalierung	534
sloppy	72
Socket	193
socket.broadcast.emit	324
socket.emit	324
Socket.IO	320
Socket-Client	411
Socket-Datei	408
Socket-Lösungen	414
Socketpool	188
Socket-Server	405
Socket-Verbindungen	318
Softwarequalität	453
Sonderzeichen maskieren	515
Spannen von Versionen	121
spawn-Methode	335
spec-Verzeichnis	461
Speichermodell	33
Speicherstrukturen	150
Sprachstandard	349
Spread-Operator	150
Spy	464
spyOn-Methode	465
SQL	428
SQL-Abfrage	436
SQL-Injections	504
SQLite	277, 427, 434
sqlite3-Treiber	435
SSH	522
Stabilitätsindex	86
Stable	86
Stack	332
Stacktrace	462, 468
Staged	131
Standard Library	84
Standard Template Strings	140
Standardausgabe	92, 162, 331, 333, 336
Standard-C-Bibliothek	39
Standardeingabe	331, 336
Standardevents	361
Standardfehlerausgabe	162, 331, 333
Standard-Logger	222
Standardrepository	51
Standardtask	532
static-Middleware	313
Statische Codeanalyse	453, 474

Statische Dateien	184, 223
Statische Methoden	139
Stats-Objekt	398
Statuscode	181, 193, 263, 270
Statusnachrichten	315
Strategien	519
Stream-API	355, 419
Streamen	356
stream-Modul	405
Streams	29, 355, 450
Stream-Versionen	357
strict	72
Strict Mode	131, 507
String-Methoden	133
Structured Query Language	428
Struktur	474
Strukturframework	279
Strukturierung	395, 469
Stylesheets	184
Subapplikationen	220
Submodul	219
Subprotokoll	308
Subshell	333
Subtemplates	248
Suchoperationen	395
Suchpfad	50, 53, 61, 65, 157
Suchprozess	105
Suchvorgang	113
sudo	50, 113
super	140
Symbolischer Link	398
Symbol-Registry	149
Symbols	148
Symfony	201
SYN/ACK-Paket	415
Synchronisierung	520, 522, 539
SYN-Paket	415
Systemabsturz	523
Systempfade	504
Systemstart	523
Systemweite Installation	112
T	
Tabelle	428
Tag-Funktion	141
Tagged Pointers	33
Tagged Template Strings	140
Tar-Archiv	527
Tasks	532
TCP	405
TCP-Client	416
TCP-Port	174
TCP-Port 8080	203
TCP-Portnummer	74
TCP-Server	416
TCP-Sockets	414
TCP-Verbindung	356, 539
TDD	471
tearDown	470
Teil-Template	245, 255
Template	304
Template Strings	140, 213
Template-Datei	235, 284
Template-Engines	229
templates	235
Template-Verarbeitung	250
Temporärer Speicherplatz	187
Ternäroperator	245
Test	380, 513
<i>fehlgeschlagener</i>	462
<i>organisieren</i>	468
Testbarkeit	85
Testfall	454
Testframework	456
Testfunktionen	460
Testgetriebene Entwicklung	471
Text	225
Textnachrichten	315
then	351
this.changes	438, 439
this.lastID	438
Thread	332
Thread-Pool	342
tight cohesion	99
Timeout	183
Timeoutfehler	75
Timeoutspanne	193
TJ Fontaine	29
Tokens	517
Toolunterstützung	528
Toolunterstützung mit Grunt	534
Trailer	178
Transaktionen	429
Transfer-Encoding	192
Transform Stream	357, 378
Transform-Shortcut	380
TravisCI	513
Trigger	429
Tripple-A	455

try-catch-Block	330, 457
Two-Way-Databinding	279
Typ des Objekts	399
Typed Arrays	150

U

Ubuntu	63
UDP	405
UDP-Client	423
UDP-Server	422
UDP-Sockets	421, 423
UglifyJS	531
Umgebung zur Laufzeit	482
Umgebungsvariablen	93
uncork-Methode	374
Undone tests	470
UND-Verknüpfung	121
Unerreichbarkeit	499
Ungepuffert	374
Unittests	453, 530
Universität Aarhus	32
UNIX	47
UNIX Domain Socket	175
Unix Pipes	355
Unix-Philosophie	83
UNIX-Socket-Client	408
UNIX-Sockets	406, 418
UNIX-Socket-Server	407
Unkontrollierte Ausführung	506
Unperformant	501
unwatchFile	400
unwatch-Funktion	486
UPDATE-Abfrage	438
update-Methode	433, 447
Updates	522
URL	80, 178, 196, 261, 405
url.parse	197
url-Eigenschaft	196
url-Modul	197
URL-Pfad	197, 215
Userliste	314
Utility	93

V

V8	32
V8-Engine	30, 32, 84, 129
Variablen	236
Variablen analysieren	488

Variablen in Routen	217
Verantwortung	491
Verbindung	311
<i>aufbauen</i>	415, 430, 436, 441, 445
<i>beenden</i>	180
<i>trennen</i>	325
Verbindungsinformationen	177
Verbindungsloses Protokoll	422
Vereinfachung	379
Vererbung	137, 139
Vergleiche	459
Verkettung	364
Verschachtelung	462
Verschlüsselte Kommunikation	198
Verschlüsselung	39
Version	
<i>asynchron</i>	383
<i>synchron</i>	383
Version eines Moduls	116
Versionen	93
Versionierung	47
Versionshistorie	129
Versionsinformationen	52, 56, 61
Versionskonflikt	104, 112
Versionskontrollsystem	520
Versionsnummer	71, 121
Vertrauenswürdige Pakete	512
Verzeichnisfreigaben	521
Verzeichnishierarchie	117, 206
Verzeichnisoperationen	394
Verzeichnisse manipulieren	394
Verzeichnisstruktur	104, 159, 454
View	207, 279
view engine	235, 304
ViewModel	279
Visuelle Gruppierung	468
Vorteile	30

W

Wartbarkeit	478
watch	400
Watch Expressions	488
Watcher	279
watch-Funktion	485
Watch-Task	532
wc	158
WeakMaps	143
WeakSets	143
Webapplikationen	201

Webframeworks	211	writeHead	182
Webserver	73, 173, 233, 259, 282, 540	write-Methode	192
Websocket	265, 301, 308, 320	ws://	308
WebSocket (Funktion)	313	wss://	308
Websocket-API	309		
websocket-Paket	310	X	
Websocket-Protokoll	308, 359	XML	271
Websocket-Spezifikation	308	xmlHttpRequest	281
WebStorm	477, 489	XML-Parser	110
Wechsel in den Flowing Mode	364	XML-Struktur	272
Wechsel in den Paused Mode	365	X-Powered-By-Header	504
Weiterentwicklung	120		
Weiterführende Operationen	398	Y	
Weiterleitung	226, 293	yield	145
Werkzeuge	115		
wget	50	Z	
while-Schleife	245	Zeichencodierung	77
Whitelist	492, 509	Zeitgesteuerte Berechnungen	302
Wiederverwendbarkeit	85, 211, 248	Zertifikat	198
Wiederverwendung	136	Zertifikatsdatei	199
Windows	29, 42, 47, 52	Zielsysteme	493
Windows Azure	542	zlib	39, 44
Windows Pipes	413, 418	Zufallszahlen	410
wordCount	100	Zugriff	408
Worker-Objekt	346	Zugriff auf den Socket	408
Workerprozess	341, 346, 537	Zugriff auf die Umgebung	484
workers	346	Zugriffsberechtigungen	401, 495
Wrapper	464	Zugriffsflag	387, 393
Writable Stream	162, 357, 369, 451	Zuordnungstabellen	504
Writable Stream erstellen	369	Zustandsänderung	350
Writable-Shortcut	376	Zyklomatische Komplexität	480
Writable-Stream-Schnittstelle	370		
write	183, 391		
writeFile	394		



Sebastian Springer

Node.js – Das Praxisbuch

560 Seiten, gebunden, 2. Auflage 2016

39,90 Euro, ISBN 978-3-8362-4003-1

 www.rheinwerk-verlag.de/4037



Sebastian Springer ist als JavaScript Engineer bei Mai-bornWolff tätig. Neben der Entwicklung und Konzeption von Applikationen liegt sein Fokus auf der Vermittlung von Wissen. Als Dozent für JavaScript, Sprecher auf zahlreichen Konferenzen und Autor versucht er die Begeisterung für professionelle Entwicklung mit JavaScript zu wecken.

Wir hoffen sehr, dass Ihnen diese Leseprobe gefallen hat. Sie dürfen sie gerne empfehlen und weitergeben, allerdings nur vollständig mit allen Seiten. Bitte beachten Sie, dass der Funktionsumfang dieser Leseprobe sowie ihre Darstellung von der E-Book-Fassung des vorgestellten Buches abweichen können. Diese Leseprobe ist in all ihren Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen beim Autor und beim Verlag.

Teilen Sie Ihre Leseerfahrung mit uns!

