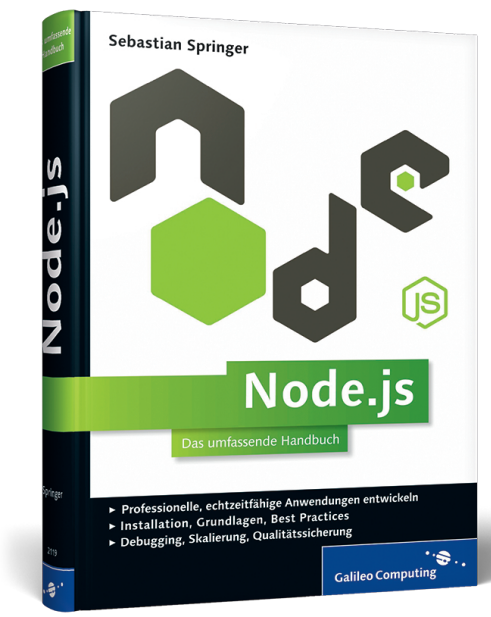
Sebastian Springer

Node.js

Das umfassende Handbuch



Galileo Press

Auf einen Blick

Inhalt

Kapitel 1
Grundlagen

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

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

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

1.1 Die Geschichte von Node.js

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

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

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

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

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

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

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

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

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

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

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

1.2 Die Vorteile von Node.js

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

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

1.3 Einsatzgebiete von Node.js

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

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

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

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

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

1.4 Das Herzstück – die V8-Engine

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

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

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

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

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

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

1.4.1 Das Speichermodell

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

1.4.2 Zugriff auf Eigenschaften

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

Prototypen in JavaScript

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

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

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

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

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

Listing 1.1 Funktionsweise von Hidden Classes

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

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

1.4.3 Maschinencodegenerierung

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

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

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

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

1.4.4 Garbage Collection

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

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

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

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

1.5 Bibliotheken um die Engine

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

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

1.5.1 Eventloop

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

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

Listing 1.2 Eventgetriebene Entwicklung in Node.js

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

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

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

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

1.5.2 Eingabe und Ausgabe

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

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

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

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

1.5.3 libuv

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

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

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

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

1.5.4 DNS

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

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

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

1.5.5 Crypto

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

1.5.6 Zlib

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

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

1.5.7 HTTP-Parser

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

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

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

1.6 Zusammenfassung

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

Kapitel 8
Anbindung von Datenbanken

Phantasie ist wichtiger als Wissen, denn Wissen ist begrenzt.
– Albert Einstein

Wie Sie bereits in den vergangenen Kapiteln gesehen haben, müssen Sie bei der Implementierung von Applikationen auch immer wieder Daten speichern. Node.js stellt Ihnen zur Persistierung von Daten lediglich das `fs`-Modul zur Verfügung. Der Nachteil dieser Methode ist, dass der Zugriff auf die Daten problematisch ist. Im schlechtesten Fall müssen Sie die gesamte Datei nach den gewünschten Informationen durchsuchen, was je nach Dateigröße sehr zeitintensiv sein kann. Dies ist ein Grund, warum Sie zur Persistierung von Informationen in Ihrer Applikation Datenbanken verwenden sollten.

Der Zugriff auf Datenbanken erfolgt in der Regel nicht direkt. In den meisten Fällen benötigen Sie eine Art von Treiber, der Ihnen als Schnittstelle zur Datenbank dient. Diese Treiber sind kein fester Bestandteil der Node.js-Plattform, sondern liegen als NPM-Module vor, die Sie je nach Bedarf installieren können.

Durch diese Struktur ist die Unterstützung von Datenbanken nicht nur auf einige wenige Systeme beschränkt, stattdessen können Sie beispielsweise die relationalen Datenbanken MySQL, MSSQL oder SQLite verwenden. Auch viele nicht-relationale Datenbanken wie Redis, CouchDB oder MongoDB werden mittlerweile unterstützt.

In diesem Kapitel erfahren Sie mehr über die Anbindung verschiedener Datenbanken an Ihre Node.js-Applikation. Konkret werden Sie MySQL, SQLite, Redis und MongoDB einsetzen. Für jede dieser Datenbanken werden Sie die grundlegenden Datenbankoperationen, die im Akronym CRUD zusammengefasst sind, umsetzen. CRUD bedeutet Create, Read, Update und Delete. Sie erstellen also neue Datensätze, lesen diese aus, aktualisieren Werte in der Datenbank und entfernen bestehende Datensätze wieder aus der Datenbank.

Außerdem erfahren Sie mehr über die Einsatzgebiete und Vor- und Nachteile der einzelnen Datenbanken. Das Kapitel startet mit dem Einsatz von relationalen Datenbanken.

8.1 Node.js und relationale Datenbanken

Relationale Datenbanken haben sich über Jahre zu einem Standard für Datenbanken in der Informationstechnologie etabliert. Dieser Typ von Datenbanken basiert auf einer Struktur von Tabellen, in denen die Daten vorgehalten werden. Eine Tabelle definiert dabei den Aufbau einzelner Datensätze.

Sie können sich eine Tabelle in einer relationalen Datenbank wie eine ganz gewöhnliche Tabelle vorstellen. Die Zeilen stellen die einzelnen Datensätze dar und die Spalten die jeweiligen Eigenschaften. Das bedeutet, dass sämtliche Datensätze einer Tabelle die gleiche Struktur aufweisen.

In relationalen Datenbanken haben Sie die Möglichkeit, nicht nur eine Tabelle, sondern nahezu beliebig viele zu definieren. Diese Tabellen können Sie dann über sogenannte *foreign keys*, auf Deutsch *Fremdschlüssel*, in Verbindung zueinander setzen. So können Sie beispielsweise eine Tabelle definieren, in der Sie die Daten Ihrer Benutzer wie den Namen oder das Geburtsdatum speichern. In einer zweiten Tabelle können Sie dann die Adressinformationen speichern. Über einen Fremdschlüssel können Sie zwischen den Datensätzen aus beiden Tabellen eine Beziehung herstellen.

Als Sprache zur Formulierung von Abfragen der Datenbank hat sich die Structured Query Language, kurz SQL, durchgesetzt. SQL wurde durch die ISO und IEC standardisiert, wobei jede konkrete Implementierung eigene Erweiterungen zu diesem Standard hinzufügt. Weitere Informationen finden Sie unter *http://de.wikipedia.org/wiki/SQL*.

Die Abschnitte zu den einzelnen Datenbanken folgen dabei stets dem gleichen Aufbau. Sie erfahren einige Detailinformationen über die verschiedenen Datenbanken und die jeweiligen Vor- und Nachteile. Darauf folgen die Installation des jeweiligen Treibers für die Datenbank und ein konkretes Beispiel zur Verwendung der Datenbank innerhalb einer Node.js-Applikation. Als Beispiel dient eine Datenbank für Adressdaten.

Der erste Teil dieses Kapitels beschäftigt sich mit der Anbindung einer MySQL-Datenbank.

8.1.1 MySQL

In den vorangegangenen Kapiteln haben Sie bereits mehrmals mit MySQL als Datenbank in Node.js-Applikationen zu tun gehabt. Diese Abschnitte wiederholen die Konzepte, die Sie bereits angewandt haben.

MySQL ist eine der am weitesten verbreiteten Datenbanken im Web. Mit nahezu allen bedeutenden Programmiersprachen können Sie auf diese Datenbank zugreifen, so auch von Node.js aus.

MySQL hat sich bereits seit vielen Jahren auch in sehr großen Applikationen als Datenbank bewährt. Das System verfügt nicht nur über eine einfache Serverkomponente, sondern erlaubt es auch, dass Sie eine Datenbank auf mehreren Servern im Master-Slave-Verbund betreiben. Das hat den Vorteil, dass Sie die Anfragen auf mehrere Systeme verteilen und so auch Ausfallsicherheit gewährleisten können.

Für sehr große Datenmengen lässt sich eine Datenbank auch partitionieren und auf mehrere Systeme verteilen, was weitere Möglichkeiten hinsichtlich Performancesteigerungen bietet.

MySQL bietet viele nützliche Features. Diese umfassen beispielsweise Trigger, also Funktionen, die bei bestimmten Operationen ausgeführt werden, beziehungsweise Transaktionen. Dies ist eine Gruppe von Operationen, die nur in ihrer Gesamtheit oder überhaupt nicht ausgeführt werden dürfen.

Ein weiterer Vorteil von MySQL ist, dass diese Datenbank auf sehr vielen verschiedenen Betriebssystemen wie beispielsweise Linux, Windows oder Mac OS X verfügbar ist. Diese Verfügbarkeit und die weite Verbreitung haben zu einer sehr aktiven Community geführt, auf die Sie bei Fragen oder Problemen zurückgreifen können.

Ein Nachteil von MySQL ist, dass es hinsichtlich des Featuresets noch nicht ganz mit den großen SQL-Datenbanken wie Oracle oder DB2 mithalten kann. Dieser Nachteil wird aber immer mehr durch die Weiterentwicklung der Datenbank ausgeglichen.

Generell bestehen zwei verschiedene Ansätze zur Verbindung mit einer MySQL-Datenbank. Zum einen existiert ein Treiber, der das MySQL-Protokoll selbst implementiert und komplett in JavaScript geschrieben ist. Für diesen Treiber wird keine weitere Software benötigt, und er kann direkt verwendet werden, um mit einer Datenbank zu arbeiten.

Eine weitere Variante von MySQL-Treibern basiert auf den MySQL-Clientbibliotheken. Diese haben den Vorteil, dass sie etwas performanter als die komplett in JavaScript implementierten Treiber arbeiten. Sie weisen allerdings den Nachteil auf, dass die MySQL-Clientbibliotheken auf dem System, auf dem die Node.js-Applikation ausgeführt wird, installiert sein müssen.

Bisher haben Sie lediglich den Einsatz des MySQL-Treibers mitverfolgen können, der komplett in JavaScript geschrieben wurde. In den nächsten Abschnitten sehen Sie, wie Sie einen Treiber verwenden können, der auf den MySQL-Clientbibliotheken basiert.

Installation

Der auf den MySQL-Clientbibliotheken basierende Treiber ist als NPM-Modul verfügbar und kann über die Kommandozeile installiert werden. Wie bereits erwähnt, müssen Sie jedoch darauf achten, dass auf Ihrem System die entsprechenden

Bibliotheken installiert sind. In Listing 8.1 sehen Sie die Kommandos, die erforderlich sind, um das Modul `db-mysql` auf einem Linux-System zu installieren.

```
apt-get install libmysqlclient-dev
npm install db-mysql
```

Listing 8.1 Installation von db-mysql

Nachdem Sie diese Kommandos abgesetzt haben, können Sie von Ihrer Applikation aus auf eine MySQL-Datenbank zugreifen.

Datenbankstruktur

Damit Sie die nachfolgenden Beispiele nachvollziehen können, benötigen Sie eine lauffähige Instanz einer MySQL-Datenbank auf einem Server. Im Falle dieses Beispiels wird angenommen, dass sich Applikation und Datenbank auf dem gleichen Rechner befinden.

```
CREATE DATABASE `node`;
USE `node`;
CREATE TABLE `Addresses` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `street` varchar(255) DEFAULT NULL,
  `place` varchar(255) DEFAULT NULL,
  `country` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Listing 8.2 Struktur der MySQL-Datenbank

Haben Sie die Statements aus Listing 8.2 ausgeführt, können Sie im nächsten Schritt mit der Umsetzung der Beispielapplikation beginnen.

Verbindung aufbauen

In jeder Applikation, die mit einer Datenbank arbeitet, muss vor der Verwendung dieser Datenbank zuerst eine Verbindung aufgebaut werden, über die einerseits die Kommandos an das Datenbanksystem gesendet werden und über die auf der anderen Seite die Informationen der Datenbank an die Applikation zurückfließen können. Listing 8.3 zeigt Ihnen, wie Sie eine solche Verbindung aufbauen.

```
var mysql = require('db-mysql');

new mysql.Database({
    hostname: 'localhost',
```

```
    user: 'root',
    password: '',
    database: 'node'
}).on('error', function (err) {
    console.log('An error occured: ' + err);
}).connect(function (err) {
    console.log('connection established');
});
```

Listing 8.3 Aufbau einer Datenbankverbindung

Im Quellcode von Listing 8.3 sehen Sie, dass Sie der `Database`-Methode eine Objektstruktur übergeben können, die die Konfiguration der Verbindung enthält. Auf dem zurückgegebenen Objekt können Sie verschiedene Methoden aufrufen. Über die `on`-Methode können Sie beispielsweise direkt Callback-Funktionen an Ereignisse binden, die im Verlauf des Verbindungsaufbaus auftreten können. Konkret sind dies die Ereignisse `error`, das im Fehlerfall auftritt, und `ready`, das ausgelöst wird, sobald die Datenbankverbindung hergestellt wurde.

Die wichtigste Methode im Zusammenhang mit der Datenbankverbindung ist `connect`. Diese Methode akzeptiert eine Callback-Funktion als Wert, die aufgerufen wird, sobald eine Verbindung hergestellt wurde. Sämtliche Datenbankoperationen finden also innerhalb dieser Callback-Funktion statt.

Neue Datensätze anlegen

Noch verfügt Ihre Datenbank lediglich über eine leere Tabelle. Bevor Sie also Datensätze auslesen oder diese verändern können, müssen Sie zuerst neue Datensätze anlegen. In Listing 8.4 sehen Sie, wie Sie dies bewerkstelligen können. Den Code dieses und der folgenden Listings sollten Sie innerhalb der Callback-Funktion der `connect`-Methode platzieren.

```
this.query().insert('Addresses',
    ['street', 'place', 'country'],
    ['Broadway 1', 'New York', 'United States of America']
).execute(function (err, result) {
    if (err) throw err;
    console.log(result);
});
```

Listing 8.4 Neue Datensätze in MySQL anlegen

Sie legen einen neuen Datensatz durch eine Kombination der Methoden `query`, `insert` und `execute` an. Dabei stellt Ihnen die Query-Methode das `Query`-Objekt zur

Verfügung, auf dem Sie die `insert`-Methode aufrufen. Dieser Methode müssen Sie als ersten Wert den Namen der Tabelle übergeben, in die Sie die Daten einfügen möchten. Das zweite Argument besteht aus einem Array von Spaltennamen, denen im dritten Argument durch ein Array Werte zugewiesen werden.

Die `execute`-Methode, die Sie auf dem Rückgabewert der `insert`-Methode aufrufen, führt schließlich die Abfrage aus. Diese Methode akzeptiert wiederum eine Callback-Funktion, die ausgeführt wird, sobald das Ergebnis der Abfrage vorliegt. Als erstes Argument erhält diese Callback-Funktion ein Fehlerobjekt, das einen Wert beinhaltet, falls bei der Abfrage ein Fehler aufgetreten ist. Das zweite Argument beinhaltet das Resultat der Abfrage. Im Falle eines Inserts sind dies die ID des neuen Datensatzes, die Anzahl der Datensätze, die betroffen waren, und die Anzahl an Warnungen, die bei der Operation aufgetreten sind.

Datensätze auslesen

Nachdem Sie Datensätze in die Datenbank eingefügt haben, werden Sie diese in den meisten Fällen auch wieder auslesen wollen. Der MySQL-Treiber bietet Ihnen hierfür die `select`-Methode des `Query`-Objekts. In Listing 8.5 sehen Sie den konkreten Einsatz dieser Methode.

```
this.query().select('*').from('Addresses').
execute(function(err, rows, cols) {
    if (err) throw err;
    console.log(rows);
    console.log(cols);
});
```

Listing 8.5 Datensätze aus einer MySQL-Tabelle auslesen

Wie Sie in Listing 8.5 sehen können, geschieht die Abfrage der Datenbank wiederum über eine Kombination von Methoden. Am Anfang steht die `query`-Methode, die das `Query`-Objekt zur Verfügung stellt. Danach verketten Sie die verschiedenen Methoden, die schließlich die Abfrage bilden, die dann mithilfe von `execute` ausgeführt wird. Die `execute`-Methode akzeptiert eine Callback-Funktion als Wert. Diese Callback-Funktion erhält drei Argumente. Das erste ist, wie in Node.js üblich, ein Fehlerobjekt. Das zweite Argument ist das eigentliche Ergebnis der Abfrage. Im Falle des Beispiels ist dies ein Array von Objekten, die die gefundenen Datensätze darstellen. Das dritte Argument enthält schließlich ein Array mit Objekten, die die gefundenen Spalten repräsentieren.

Zum Aufbau einer Abfrage können Sie auf verschiedene Methoden zurückgreifen. Die Basis bilden die Methoden `select` und `from`. Zur Einschränkung der gefundenen Datensätze können Sie mit der `where`-Methode Bedingungen definieren, die die Datensätze

erfüllen müssen. Mit `join` können Sie mehrere Tabellen innerhalb einer Abfrage verbinden. Um den Rahmen dieses Kapitels nicht zu sprengen, werde ich hier nicht auf weiterführende Operationen wie die Verbindung mehrerer Tabellen eingehen.

Datensätze aktualisieren

Der Vorteil einer Datenbank ist, dass Sie sehr schnell auf Daten zugreifen und diese bei Bedarf auch ändern können. Diese Aktualisierung erreichen Sie über die `update`-Methode. Listing 8.6 zeigt Ihnen, wie Sie bei Datenaktualisierungen vorgehen.

```
this.query().update('Addresses').
set({'street': 'Tower Hill', 'place': 'London', 'country': 'United Kingdom'}).
where('id = ?', [1]).
execute(function(err, result) {
    if (err) throw err;
     console.log(result);
});
```

Listing 8.6 Datensätze in einer MySQL-Datenbank ändern

In Listing 8.6 formulieren Sie die Abfrage zum Ändern von Datensätzen ähnlich, wie Sie es auch direkt in SQL machen würden, also mit einer Kombination aus `query`, `update`, `set`, `where` und `execute`. Mit `update` geben Sie an, welche Tabelle betroffen ist. `set` erhält die Werte, die geändert werden sollen, in einer Objektstruktur als Argument. Mit `where` schränken Sie die Datensätze ein, die aktualisiert werden sollen. Das Fragezeichen in der Zeichenkette dient als Platzhalter, der bei der Ausführung durch den Wert im Array, das Sie als zweites Argument übergeben, ersetzt wird. Dabei wird dieser Wert korrekt escaped, um Angriffe mit SQL-Injections zu vermeiden. `execute` führt schließlich die Abfrage aus. Auch diesmal erhält diese Methode wieder eine Callback-Funktion mit einem Fehlerobjekt und einem Resultat-Objekt. Dieses Objekt enthält eine ID-Eigenschaft, die allerdings den Wert `0` aufweist, die Anzahl der betroffenen Zeilen und die Anzahl der Warnungen.

Datensätze entfernen

Die letzte Operation, die Sie hier kennenlernen, ist das Löschen von Datensätzen aus der Datenbank. Das Löschen von Datensätzen lässt sich nicht ohne Weiteres rückgängig machen. Arbeiten Sie mit referenzieller Integrität über Fremdschlüssel, kann das Löschen eines Datensatzes eine Kaskade von Löschungen anderer Datensätze nach sich ziehen, die auf diesem Datensatz aufbauen. Eine weiterführende Erklärung zu Fremdschlüsseln unter MySQL finden Sie unter *http://dev.mysql.com/doc/refman/5.5/en/create-table-foreign-keys.html*. Beim Löschen von Datensätzen sollten Sie also stets Vorsicht walten lassen. Listing 8.7 zeigt Ihnen, wie Sie Datensätze aus Ihrer Datenbank entfernen können.

```
this.query().delete().from('Addresses').
where('id = ?', [ 1 ]).
execute(function(err, result) {
    if (err) throw err;
    console.log(result);
});
```

Listing 8.7 Löschen von Datensätzen aus einer MySQL-Datenbank

Auch in Listing 8.7 sehen Sie die gewohnte Kombination von Methoden zum Formulieren der Abfrage. Die Callback-Funktion der `execute`-Methode erhält zwei Argumente. Das erste Argument enthält eventuelle Fehler, die bei der Löschoperation aufgetreten sind. Das zweite Argument enthält das Ergebnis der Abfrage. Im Fall des Löschens eines Datensatzes ist dies ein Objekt, das insgesamt drei Eigenschaften enthält. Die Eigenschaft `id` enthält immer den Wert `0`, `affected` gibt an, wie viele Zeilen gelöscht wurden und `warnings` enthält die Anzahl der Warnungen, die während der Abfrage aufgetreten sind.

Eine Alternative zum endgültigen Löschen von Datensätzen ist das Markieren von Datensätzen. In der Datenbank wird dies durch ein weiteres Feld innerhalb der Tabelle repräsentiert. Dieses Feld enthält den Wert `0` für aktive Datensätze und den Wert `1` für gelöschte Datensätze. Der Nachteil dieser Variante ist, dass Sie sich selbst um die referenzielle Integrität Ihrer Datenbank kümmern müssen, also abhängige Datensätze selbst als gelöscht markieren müssen.

Eine leichtgewichtigere Alternative zu MySQL als Datenbank für Ihre Node.js-Applikation ist SQLite.

8.1.2 SQLite

SQLite ist im Vergleich zu MySQL ein Leichtgewicht. Einer der wichtigsten Unterschiede ist, dass bei SQLite kein Serverprozess benötigt wird. Die Clientsoftware greift bei Abfragen direkt auf die in einer Datei gespeicherten Datenbank zu. SQLite erfordert außerdem keine Konfiguration. Sie können die Datenbank nach der Initialisierung sofort verwenden.

SQLite steht Ihnen auf den verschiedensten Systemen zur Verfügung. So gibt es vorkompilierte Binärpakete für Linux, Mac OS X und auch für Windows.

Da SQLite die Datenbankdaten in einer Datei speichert, können diese auf sehr einfache Weise kopiert und gesichert werden.

Im Gegensatz zu MySQL verfügt SQLite über keine eigene Benutzerverwaltung. Sie haben also keine Möglichkeit, Berechtigungen zu vergeben, was ein Mehrbenutzersystem unmöglich macht.

Installation

Für SQLite existieren verschiedene Treiber, mit deren Hilfe Sie auf die Datenbank zugreifen können. In den folgenden Abschnitten lernen Sie den `sqlite3`-Treiber kennen. Dieser liegt als NPM-Paket vor und kann mit dem Kommando, das Sie in Listing 8.8 sehen, installiert werden.

```
$ npm install sqlite3
```

Listing 8.8 Installation des sqlite3-Treibers

Sobald Sie das Paket installiert haben, können Sie eine SQLite-Datenbank an Ihre Applikation anbinden und diese zur Persistierung von Daten verwenden.

Datenbankstruktur

Bevor Sie jedoch mit der Datenbank interagieren können, müssen Sie diese anlegen. Listing 8.9 führt Sie durch diesen Prozess, an dessen Ende eine funktionierende Datenbank mit einer Adresstabelle steht, die Sie in den folgenden Abschnitten als Ausgangssituation verwenden werden.

```
$ sqlite3 node.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE `Addresses` (
   `street` varchar(255) DEFAULT NULL,
   `place` varchar(255) DEFAULT NULL,
   `country` varchar(255) DEFAULT NULL
 );
sqlite>
```

Listing 8.9 Struktur der SQLite-Datenbank

Beim Aufbau der Tabelle fällt Ihnen bestimmt das Fehlen der `ID`-Spalte im Vergleich zu MySQL auf. Ein solches Feld ist bei SQLite unnötig, da jeder Datensatz automatisch eine Zahl zugeordnet bekommt, die in der Tabelle eindeutig ist und über die der Datensatz identifiziert werden kann. Zugreifen können Sie auf diesen Wert über den Spaltennamen `rowid`.

Verbindung aufbauen

Da SQLite wie bereits erwähnt über keine interne Benutzerverwaltung verfügt, ist der Aufbau einer Verbindung relativ problemlos. Listing 8.10 zeigt Ihnen, wie Sie die Verbindung zu einer SQLite-Datenbank aufbauen können, die in der Datei `node.db` liegt.

```
var sqlite = require('sqlite3');
var db = new sqlite.Database('node.db');
```

Listing 8.10 Aufbau einer Verbindung zu einer SQLite-Datenbank

Um eine Verbindung zu Ihrer Datenbank aufzubauen, reichen zwei Statements aus. Im ersten Schritt binden Sie das NPM-Paket ein, das Ihnen die Verbindung zur Datenbank ermöglicht. Im zweiten Schritt initialisieren Sie schließlich die eigentliche Datenbankverbindung, indem Sie ein Datenbank-Objekt erzeugen und dabei den Namen der Datei angeben, in der sich die Datenbank befindet. Im weiteren Verlauf Ihrer Applikation können Sie mit diesem Datenbank-Objekt Ihre Abfragen an die Datenbank senden.

Zunächst nutzen Sie nun die bestehende Verbindung zur Datenbank, um einen neuen Datensatz anzulegen.

Neue Datensätze anlegen

Noch verfügt die Datenbank in der Datei `node.db` lediglich über eine leere Tabelle `Addresses`. Im ersten Schritt fügen Sie nun einen neuen Datensatz ein. Listing 8.11 zeigt Ihnen den dazu erforderlichen Quellcode. Dabei handelt es sich um eine Erweiterung von Listing 8.10, und es wird davon ausgegangen, dass `db` das Datenbank-Objekt enthält.

```
db.run('INSERT INTO Addresses VALUES (?, ?, ?)',
    ['Broadway 1', 'New York', 'United States of America'],
    function (err) {
        if (err) throw err;
        console.log('ID: ' + this.lastID);
    });
```

Listing 8.11 Neue Datensätze in eine SQLite-Tabelle einfügen

Mit der `run`-Methode des Datenbank-Objekts können Sie Abfragen zum Anlegen neuer Datensätze an die Datenbank absetzen. Diese Methode akzeptiert drei Argumente, von denen das zweite optional ist. Das erste Argument besteht aus der SQL-Abfrage, die an die Datenbank gesendet werden soll. Das zweite Argument enthält Parameter, die in die Abfrage eingebunden werden sollen. Sie sehen in Listing 8.11, dass die SQL-Abfrage drei Fragezeichen enthält. Diese werden durch das Array im zweiten Argument mit konkreten Werten ersetzt. Dabei kümmert sich der Datenbanktreiber um das korrekte Escaping der Werte. Das dritte Argument ist schließlich eine Callback-Funktion, die aufgerufen wird, sobald die Antwort des Servers vorliegt. Das einzige Argument, auf das Sie in dieser Callback-Funktion zugreifen können, ist ein Fehlerobjekt, das existiert, falls bei der Abfrage Probleme aufgetreten sein sollten.

Über `this` können Sie innerhalb der Callback-Funktion auf weitere Informationen zugreifen. Im Falle einer `INSERT`-Abfrage steht Ihnen hier die Eigenschaft `lastID` zur Verfügung. Sie enthält die `rowid` des Datensatzes, den Sie mit diesem Kommando eingefügt haben.

Das Resultat der Operation können Sie entweder direkt über einen Kommandozeilenclient für SQLite prüfen, oder Sie erweitern Ihren Code um eine weitere Abfrage, die die entsprechenden Werte wieder aus Ihrer Datenbank ausliest. Der nächste Abschnitt beschäftigt sich mit genau dieser Problemstellung.

Datensätze auslesen

Haben Sie erst einmal Daten in Ihre Datenbank geschrieben, möchten Sie diese zu einem bestimmten Zeitpunkt auch wieder auslesen. Der von Ihnen bisher verwendete SQLite-Treiber verfügt über mehrere Methoden, mit deren Hilfe Sie Daten aus Ihrer Datenbank auslesen können.

Die einfachste Variante, Daten aus Ihrer Datenbank auszulesen, ist die Verwendung der `get`-Methode. Mit ihr kann ein einzelner Datensatz abgefragt werden. Wie Sie hier genau vorgehen sollten, sehen Sie in Listing 8.12.

```
db.get('SELECT * FROM Addresses WHERE rowid = ?',
    [1],
    function (err, row) {
        if (err) throw err;
        console.log(row);
});
```

Listing 8.12 Einen Datensatz aus einer SQLite-Datenbank auslesen

Die Signatur ist sehr ähnlich zur `run`-Methode. Der erste Wert, den Sie an den Aufruf dieser Methode übergeben, ist ebenfalls eine Zeichenkette, die die SQL-Abfrage enthält, die Sie ausführen möchten. Auch hier können Sie wieder Fragezeichen als Platzhalter verwenden, die durch die Werte im zweiten Argument ersetzt werden. Das dritte Argument besteht aus der Callback-Funktion, die ausgeführt wird, sobald das Ergebnis der Abfrage vorliegt. Das erste Argument dieser Funktion ist wie bei Node.js üblich ein Fehlerobjekt. Das zweite Argument besteht aus einem Objekt, das den ausgelesenen Datensatz repräsentiert. Die Spaltennamen sind dabei die Eigenschaften. Die jeweiligen Werte der Felder des Datensatzes werden dann diesen Eigenschaften zugewiesen.

Zusätzlich zur `get`-Methode bietet Ihnen der SQLite-Treiber zwei weitere Methoden. Mithilfe der `all`-Methode können Sie nicht nur einen einzelnen Datensatz, sondern sämtliche Datensätze, die eine `SELECT`-Abfrage zurückliefert, verarbeiten. Die Verwendung der `all`-Methode gleicht der der `get`-Methode, bis auf die Tatsache, dass Sie

in der Callback-Funktion nicht ein Objekt, sondern ein Array von Objekten als zweites Argument erhalten.

Die dritte Methode `each` ermöglicht es Ihnen schließlich, eine Callback-Funktion zu definieren, die für jeden Datensatz, den eine Abfrage zurückliefert, ausgeführt wird. Diese Methode verhält sich wie die `get`-Methode, außer dass sie ein viertes Argument akzeptiert. Dieses Argument ist eine weitere Callback-Funktion, die aufgerufen wird, sobald sämtliche Callbacks für die einzelnen Datensätze abgearbeitet wurden.

Der folgende Abschnitt zeigt Ihnen, wie Sie vorgehen können, falls Sie Datensätze, die sich bereits in Ihrer Datenbank befinden, ändern möchten.

Datensätze aktualisieren

In SQLite können Sie wie auch schon in MySQL die Informationen, die als Datensätze in Ihrer Datenbank liegen, über bestimmte Abfragen ändern. Der SQL-Standard sieht für diesen Fall die `UPDATE`-Abfrage vor. Um diese Art von Abfragen abzusetzen, müssen Sie auf die `run`-Methode zurückgreifen, die Sie bereits zum Anlegen von Datensätzen verwendet haben. Listing 8.13 stellt Ihnen vor, wie Sie bei der Aktualisierung von Datensätzen vorgehen sollten.

```
db.run('UPDATE Addresses SET street = ?, place = ?, country = ? WHERE rowid =
 ?',
    ['Tower Hill', 'London', 'United Kingdom', 1],
    function (err) {
        if (err) throw err;
        console.log(this.changes);
});
```

Listing 8.13 Datensätze in einer SQLite-Datenbank aktualisieren

Der Quellcode dieses Listings funktioniert ähnlich zu dem, den Sie implementiert haben, als Sie Datensätze angelegt haben. Die einzigen Unterschiede sind die geänderte SQL-Abfrage im ersten Argument der `run`-Methode und die Tatsache, dass in der Callback-Funktion im dritten Argument statt auf `this.lastID` auf `this.changes` zugegriffen wird. Diese Eigenschaft enthält die Anzahl der betroffenen Zeilen.

Die letzte Operation, die Sie hier zu SQLite kennenlernen, ist das Entfernen von Datensätzen.

Datensätze entfernen

Um Datensätze aus Ihrer Datenbank wieder zu entfernen, können Sie wie beim Einfügen und Aktualisieren auch auf die `run`-Methode zurückgreifen. In Listing 8.14 sehen Sie, wie Sie Ihr Ziel mit ein paar einfachen Anpassungen des Quellcodes aus Listing 8.13 erreichen können.

```
db.run('DELETE FROM Addresses WHERE rowid = ?',
    [1],
    function (err) {
        if (err) throw err;
        console.log(this.changes);
});
```

Listing 8.14 Datensätze aus einer SQLite-Datenbank löschen

In der `run`-Methode müssen Sie im Vergleich zur Aktualisierung von Datensätzen lediglich die SQL-Abfrage anpassen und die Parameter, die Sie verwenden möchten, entsprechend ändern. In der Callback-Funktion erfahren Sie wiederum über `this.changes`, wie viele Datensätze von der Löschung betroffen waren.

Wie schon in den Abschnitten über MySQL ist auch hier ein Wort der Warnung angebracht. Das Löschen von Datensätzen kann nicht umgekehrt werden. Das bedeutet, dass einmal gelöschte Datensätze unwiederbringlich verloren sind, es sei denn, Sie haben eine funktionierende Datensicherung, aus der Sie die Datensätze wiederherstellen können.

Die hier vorgestellten Features von SQLite und des SQLite-Treibers sind lediglich ein kleiner Ausschnitt des vollen Funktionsumfangs. Möchten Sie mehr darüber erfahren, was Sie mit dieser Datenbank und dem zugehörigen Treiber umsetzen können, verweise ich Sie an dieser Stelle auf die jeweilige Online-Dokumentation unter *http://www.sqlite.org/docs.html* für SQLite und *https://github.com/developmentseed/node-sqlite3/wiki* für den SQLite-Treiber.

Mit MySQL und SQLite haben Sie zwei Vertreter relationaler Datenbanken kennengelernt. In den folgenden Abschnitten erfahren Sie mehr über eine andere Kategorie von Datenbanken, die nicht-relationalen Datenbanken.

8.2 Node.js und nicht-relationale Datenbanken

Lange Zeit waren relationale Datenbanken die einzige weitverbreitete Art, Daten strukturiert zu speichern und sie für schnellen Zugriff wieder zur Verfügung zu stellen. Mittlerweile hat sich allerdings auch noch eine weitere Art von Datenbanken auf dem Markt etabliert. Sie setzen bewusst auf einen anderen Ansatz. Im Gegensatz zu relationalen Datenbanken mit ihren fest definierten Tabellenstrukturen wird bei nicht-relationalen Datenbanken keine derartige Struktur vorgeschrieben.

Häufig werden in nicht-relationalen Datenbanken Objekt- oder Dokumentstrukturen gespeichert, auf die Sie über bestimmte Schlüssel- beziehungsweise Indexwerte zugreifen können. Diese werden unter dem Sammelbegriff NoSQL zusammenge-

fasst. Weiterführende Informationen finden Sie unter *http://de.wikipedia.org/wiki/NoSQL*. In den folgenden Abschnitten lernen Sie mit Redis und MongoDB zwei Vertreter der Kategorie der nicht-relationalen Datenbanken kennen.

8.2.1 Redis

Im Kern ist Redis ein einfacher Key-Value-Store. Sie haben hier die Möglichkeit, Werte zu speichern und sie mit einem Schlüssel zu versehen. Über diesen Schlüssel können Sie wieder auf diesen Wert zugreifen.

Redis speichert die Werte im Arbeitsspeicher, was zu einer sehr hohen Leistung bei Lese- und Schreiboperationen führt. Der Nachteil dieser Technologie ist, dass der Inhalt des Arbeitsspeichers nach einem Systemabsturz unwiederbringlich verloren ist. Die Entwickler von Redis haben für dieses Problem eine sehr elegante Lösung gefunden. Das System kann zu definierbaren Zeitpunkten eine Sicherheitskopie der Datenbank auf die Festplatte schreiben, von der aus eine Wiederherstellung der Inhalte problemlos möglich ist.

Redis ist primär für POSIX-Systeme wie Linux, BSD oder Mac OS X verfügbar und wird auch auf diesen Systemen entwickelt. Für Windows existiert zwar eine Version des Redis-Servers, diese ist allerdings noch experimentell.

Installation

Der Redis-Client für Node.js liegt wie die übrigen Treiber für Datenbanken auch als NPM-Paket vor und kann über die Kommandozeile installiert werden. Das dafür notwendige Kommando können Sie Listing 8.15 entnehmen.

```
$ npm install redis
```

Listing 8.15 Installation des Redis-Clients

Sobald Sie den Redis-Server und das NPM-Paket mit dem Redis-Client auf Ihrem System installiert haben, können Sie Redis in Ihrer Node.js-Applikation verwenden.

Da Redis auf keine festen Strukturen setzt, müssen Sie keine weiteren Operationen zur Initialisierung einer Datenbank unternehmen, sondern können sich direkt um die Verbindung mit dem Server kümmern.

Verbindung aufbauen

Der Verbindungsaufbau zu einem Redis-Server ist ähnlich einfach wie der zu einer SQLite-Datenbank. In Listing 8.16 sehen Sie, wie Sie die Verbindung in nur wenigen Zeilen Quellcode herstellen können.

```
var redis = require('redis'),
    client = redis.createClient();

client.on('error', function (err) {
    console.log('An error occured: ' + err);
});

client.quit();
```

Listing 8.16 Verbindung zum Redis-Server aufbauen

Wie Sie Listing 8.16 entnehmen können, benötigen Sie für die eigentliche Verbindung lediglich zwei Kommandos. Das erste, `require`, dient dazu, das NPM-Paket in Ihre Applikation einzubinden. Mit einem Aufruf von `createClient` erzeugen Sie ein Objekt, über das Sie mit dem Redis-Server kommunizieren können.

Die Kommunikation erfolgt asynchron und eventbasiert. Das bedeutet konkret, dass Sie Callback-Funktionen auf verschiedene Ereignisse binden und damit jeweils auf das Ereignis reagieren können. In Listing 8.16 sehen Sie dies anhand des `error`-Events, das ausgelöst wird, falls ein Fehler bei der Verbindung mit dem Redis-Server auftritt. Weitere Ereignisse, auf die Sie reagieren können, sind `ready`, `connect`, `end`, `drain` und `idle`.

Sobald Sie die Verbindung zum Server nicht mehr benötigen, sollten Sie diese mit einem Aufruf der Methode `exit` oder `quit` beenden. Andernfalls wird die Verbindung offen gehalten und die Applikation nicht beendet. Der Unterschied zwischen beiden Methoden liegt darin, dass `exit` die Verbindung sofort beendet, egal ob noch Antworten vom Server ausstehen oder nicht. `quit` dagegen wartet, bis sämtliche Antworten vom Server vorliegen und beendet dann die Verbindung.

Datensätze anlegen

Mit der Methode `set` können Sie neue Datensätze in Redis anlegen. In Listing 8.17 sehen Sie, wie diese Methode konkret verwendet wird. Dieses Listing ist eine direkte Erweiterung von Listing 8.16. Achten Sie darauf, dass Sie den Aufruf von `quit` am Ende von Listing 8.16 auf jeden Fall entfernen, da ansonsten die Verbindung in der Callback-Funktion beendet wird.

```
client.set('node.js', 'Hello World', function (err, res) {
    if (err) throw err;
    console.log(res);
    client.quit();
});
```

Listing 8.17 Einen Datensatz in Redis anlegen

Die `set`-Methode erhält insgesamt drei Argumente. Das erste Argument ist eine Zeichenkette, die den Schlüssel angibt, über den Sie später wieder auf die Daten, die Sie im zweiten Argument angeben, zugreifen können. Das dritte Argument ist schließlich eine Callback-Funktion, die ausgeführt wird, sobald der Server eine Rückmeldung sendet. Diese Callback-Funktion erhält zwei Werte, der erste ist ein Fehlerobjekt, das im Standardfall den Wert `null` aufweist und nur im Fehlerfall weitere Informationen enthält. Der zweite Wert ist die Antwort des Servers. Gab es keine Probleme beim Einfügen der Daten, erhalten Sie hier die Zeichenkette `OK`.

In Listing 8.17 sehen Sie außerdem, dass innerhalb der Callback-Funktion die Methode `quit` aufgerufen wird, um die Verbindung nach erfolgtem Einfügen der Daten zu beenden.

Haben Sie Daten eingefügt, möchten Sie diese irgendwann auch wieder auslesen. Sie sehen im folgenden Abschnitt, wie Sie mit wenig Aufwand auf Ihre Daten zugreifen können.

Datensätze auslesen

Ähnlich einfach wie das Erstellen ist auch das Auslesen von Datensätzen. Mit der `get`-Methode können Sie über einen Schlüssel auf einen gespeicherten Wert zugreifen, wie Sie auch in Listing 8.18 sehen können.

```
client.get('node.js', function (err, res) {
    if (err) throw err;
    console.log(res);
    client.quit();
});
```

Listing 8.18 Einen Wert aus Redis auslesen

Im Gegensatz zur `set`-Methode akzeptiert `get` lediglich zwei Eingabewerte. Der erste ist der Schlüssel des Werts, den Sie auslesen möchten, und der zweite eine Callback-Funktion, die ausgeführt werden soll, sobald das Ergebnis vom Server vorliegt.

Die Callback-Funktion erhält zwei Argumente. Das erste besteht aus einem Fehlerobjekt, und das zweite enthält die Antwort des Servers, bestehend aus dem Wert für den zuvor angegebenen Schlüssel. Innerhalb der Callback-Funktion können Sie wiederum die Verbindung beenden.

Datensätze aktualisieren

Möchten Sie Datensätze, die in Ihrer Datenbank liegen, modifizieren, können Sie hierbei auf zwei verschiedene Arten vorgehen. Wollen Sie den Datensatz grundlegend verändern, erreichen Sie dies, indem Sie den neuen Wert einfach per `set` speichern. Dies überschreibt den bereits bestehenden Datensatz in der Datenbank.

Möchten Sie allerdings lediglich Informationen an den Datensatz anhängen, können Sie die `append`-Methode verwenden. Listing 8.19 zeigt Ihnen in einem stark verkürzten Beispiel, wie Sie in einem solchen Fall vorgehen müssen.

```
client.set('node.js', 'Hello', function (err, res) {
    client.append('node.js', ' World', function (err, res) {
        client.get('node.js', function (err, res) {
            console.log(res);
            client.quit();
        });
    });
});
```

Listing 8.19 Informationen an einen bestehenden Datensatz anhängen

Die `append`-Methode wird wie `set` aufgerufen, nur dass sie den bestehenden Datensatz nicht ersetzt, sondern lediglich die Informationen anhängt.

Im Verlauf Ihrer Applikation kann es immer wieder vorkommen, dass Sie Daten aus Ihrer Datenbank wieder entfernen möchten. Der folgende Abschnitt beschäftigt sich damit, wie Sie Datensätze aus Ihrer Datenbank entfernen.

Datensätze entfernen

Der Befehlssatz des Redis-Servers enthält das Kommando `del`, mit dem Sie Einträge aus Ihrer Datenbank entfernen können. Listing 8.20 beschreibt den Einsatz der `del`-Methode anhand eines einfachen Beispiels.

```
client.del('node.js', function (err, res) {
    console.log(res);
});
```

Listing 8.20 Daten aus einer Redis-Datenbank löschen

Um einen Datensatz aus Ihrer Datenbank zu entfernen, übergeben Sie der `del`-Methode den Schlüssel des Datensatzes, den Sie löschen möchten. Als zweites Argument akzeptiert diese Methode eine Callback-Funktion, die ausgeführt wird, sobald die Löschung erfolgt ist. Sie erhalten hier Zugriff sowohl auf ein Fehlerobjekt als auch auf die Anzahl der Datensätze, die bei dieser Operation gelöscht wurden.

Diese Betrachtung von Redis und des Redis-Clients für Node.js ist lediglich ein kleiner Ausschnitt des Featuresets, das Ihnen in diesem Zusammenhang zur Verfügung steht. Über den Redis-Client können Sie auf den gesamten Befehlssatz, den Ihnen der Redis-Server bietet, zugreifen. So können Sie beispielsweise Hashes und Listen erstellen und mit diesen arbeiten. Sollten Sie sich näher mit Redis beschäftigen wollen, ist *http://redis.io* dafür ein guter Ausgangspunkt.

Mit MongoDB lernen Sie in den nächsten Abschnitten einen weiteren Vertreter nicht-relationaler Datenbanken kennen.

8.2.2 MongoDB

MongoDB ist ein Vertreter der dokumentenorientierten Datenbanken. Das bedeutet, dass die Grundlage der Speicherung von Informationen in dieser Datenbank auf Dokumenten basiert, die im BSON-Format vorliegen. Weitere Informationen zu diesem Format finden Sie unter *http://bsonspec.org/*.

MongoDB eignet sich aufgrund seiner guten Performance auch für größere Applikationen und bringt darüber hinaus einige weitere Features mit, die gerade bei sehr großen Datenmengen entscheidende Vorteile bringen.

MongoDB ist für die verschiedensten Betriebssysteme wie Linux, Windows, Solaris und Mac OS X verfügbar. Sie können MongoDB also auf allen Systemen installieren, auf denen auch Node.js lauffähig ist.

Installation

Sobald Sie die Serversoftware von MongoDB auf Ihrem System installiert haben, benötigen Sie lediglich noch einen Treiber, über den Sie auf Ihre Datenbank zugreifen können.

Für Node.js existiert ein in JavaScript implementierter Treiber als NPM-Paket, das Sie über das Kommando installieren können, das Sie in Listing 8.21 sehen.

```
$ npm install mongodb
```

Listing 8.21 Installation des MongoDB-Treibers

Im ersten Schritt erfahren Sie nun wie bei den vorangegangenen Datenbanken auch, wie Sie eine Verbindung zu Ihrer Datenbank herstellen können, über die Sie später dann Datensätze erzeugen, auslesen, modifizieren und wieder löschen können.

Verbindung aufbauen

Die Verbindung zur Datenbank basiert auf dem MongoDB-Client. Diesen müssen Sie korrekt initialisieren. Listing 8.22 beschreibt, wie Sie hierbei vorgehen müssen.

```
var Db = require('mongodb').Db,
    Server = require('mongodb').Server;

var client = new Db('test', new Server("127.0.0.1", 27017, {}), {w: 1});

client.open(function (err, client) {
```

```
    client.createCollection('Addresses', function (err, collection) {
    });
});
```

Listing 8.22 Verbindung zur Datenbank aufbauen

Zum Aufbau der Verbindung benötigen Sie sowohl die `Db`-Klasse als auch die `Server`-Klasse des MongoDB-Clients. Das Client-Objekt erhalten Sie, indem Sie eine neue Instanz der `Db`-Klasse erzeugen. Der Konstruktor erhält den Namen der Datenbank, das Serverobjekt und die Optionen für die Verbindung. Rufen Sie schließlich die `open`-Methode des Clients auf, wird die Verbindung hergestellt. Eine weitere Besonderheit von MongoDB ist, dass die Dokumente, die Sie anlegen, nicht direkt in der Datenbank abgelegt werden, sondern in sogenannten Collections. Das sind Sammlungen von Dokumenten, die sich im weitesten Sinne mit Tabellen vergleichen lassen.

Eine Collection können Sie über das Client-Objekt mit einem Aufruf der Methode `createCollection` erzeugen. Als ersten Wert akzeptiert diese Methode den Namen der Collection. Der zweite Wert besteht aus einer Callback-Funktion, die für diese Collection ausgeführt werden soll. Alle im Folgenden vorgestellten Operationen werden innerhalb dieser Callback-Funktion ausgeführt.

Im nächsten Schritt sehen Sie, wie Sie neue Datensätze anlegen können.

Datensätze anlegen

Die Datensätze in MongoDB liegen in einem JSON-ähnlichen Format vor. Das bedeutet, dass Sie mit Node.js auf sehr einfache Art neue Datensätze erzeugen können. Der Client stellt Ihnen zu diesem Zweck die Methode `insert` zur Verfügung. Die Verwendung dieser Methode können Sie Listing 8.23 entnehmen.

```
var address = {
    street: 'Broadway 1',
    place: 'New York',
    country: 'United States of America'
};
collection.insert(address, {safe: true}, function (err, res) {
    if (err) throw err;
    console.log(res);
});
```

Listing 8.23 Datensätze in eine MongoDB einfügen

Die `insert`-Methode erhält als erstes Argument eine Objektstruktur, die den neuen Datensatz repräsentiert. Im Falle des Beispiels in Listing 8.23 ist dies eine Adresse. Das zweite Argument besteht aus einem Objekt mit Konfigurationsoptionen. Möchten

Sie im dritten Argument eine Callback-Funktion angeben, die nach erfolgter Einfügung ausgeführt werden soll, müssen Sie hier das Schlüssel-Werte-Paar `safe: true` angeben. Nur dann wird die Callback-Funktion erst nach dem Einfügen ausgeführt.

Die Callback-Funktion erhält als erstes Argument ein Fehlerobjekt und als zweites das Ergebnis der Einfügung. Das zweite Objekt besteht aus den Werten, die eingefügt wurden, plus der eindeutigen ID, die MongoDB diesem Datensatz zugewiesen hat.

Datensätze auslesen

Haben Sie erst einmal Datensätze in der Datenbank angelegt, können Sie diese auch wieder mit der `find`-Methode auslesen.

```
collection.find().toArray(function (err, docs) {
    if (err) throw err;
    console.log(docs);
});
```

Listing 8.24 Datensätze aus einer MongoDB auslesen

Auf die mit `createCollection` erstellte Sammlung von Dokumenten können Sie die `find`-Methode aufrufen, um dort nach Dokumenten zu suchen, die bestimmten Kriterien genügen. Übergeben Sie der `find`-Methode keinerlei Werte, werden sämtliche Dokumente ausgelesen. Sie können hier allerdings auch Abfragen definieren und so nur bestimmte Dokumente auswählen. So ist es beispielsweise möglich, nur Dokumente auszulesen, deren `street`-Schlüssel einen bestimmten Wert aufweist.

Auf das von `find` zurückgegebene Objekt können Sie die `toArray`-Methode aufrufen. Diese sorgt dafür, dass Ihnen in einer Callback-Funktion neben einem Fehlerobjekt ein Array mit allen gefundenen Dokumenten zur Verfügung steht.

Haben Sie auf diese Weise Dokumente gefunden, kann es sein, dass Sie eines dieser Dokumente anpassen möchten. Der nächste Abschnitt zeigt Ihnen, wie Sie hierbei vorgehen müssen.

Datensätze aktualisieren

Wie bei vielen anderen Datenbanken haben Sie auch bei MongoDB die Möglichkeit, die Werte bestehender Datensätze anzupassen. Mit der `update`-Methode des MongoDB-Clients können Sie dies auf eine sehr einfache Weise erreichen, wie Ihnen Listing 8.25 zeigt.

```
var newAddress =
 {street: 'Tower Hill', place: 'London', country: 'United Kingdom'};
collection.update({street: 'Broadway 1'}, {$set: newAddress}, {safe: true}, fu
```

```
nction (err) {
    if (err) throw err;
});
```

Listing 8.25 Datensätze in einer MongoDB aktualisieren

Die `update`-Methode rufen Sie wie auch schon die `find`-Methode auf eine Collection auf. Im ersten Argument formulieren Sie mit einem Objekt eine Bedingung, die ein Datensatz erfüllen muss, damit er aktualisiert wird. Das zweite Argument gibt an, welche Werte ersetzt werden müssen. Mit dem Konfigurationsobjekt im dritten Argument können Sie mithilfe von `safe: true` die Callback-Funktion im vierten Argument aktivieren, die ausgeführt wird, sobald die Daten aktualisiert sind.

Datensätze entfernen

Sind Datensätze in Ihrer Datenbank veraltet oder müssen sie aus einem anderen Grund entfernt werden, können Sie dies durch einen Aufruf der `remove`-Methode erreichen. Listing 8.26 zeigt Ihnen die Verwendung dieser Methode.

```
collection.remove({street: 'Tower Hill'}, {safe: true}, function (err, res) {
    if (err) throw err;
});
```

Listing 8.26 Entfernen von Datensätzen aus der Datenbank

Die `remove`-Methode erhält als ersten Wert die Bedingung, mit der die Datensätze identifiziert werden, die gelöscht werden sollen.

MongoDB bietet eine Vielzahl weiterer Operationen, mit denen Sie die Datenspeicherung innerhalb Ihrer Applikation unterstützen können. Damit wird MongoDB zu einer hochperformanten Datenbank, wenn es darum geht, viele Daten in Form von Dokumenten zu speichern.

8.3 Zusammenfassung

Wie Sie im Verlauf dieses Kapitels sehen konnten, unterstützt Node.js durch die als NPM-Pakete verfügbaren Datenbanktreiber eine Vielzahl von Datenbanken. Dabei ist die Unterstützung nicht nur auf relationale Datenbanken beschränkt. Sie können stattdessen auch verschiedenste nicht-relationale Datenbanken wie Redis oder MongoDB verwenden.

Mithilfe der Treiber können Sie neben den grundlegenden Datenbankfunktionen wie dem Anlegen von Datensätzen und dem Auslesen, Modifizieren und Löschen von Datensätzen auch auf erweiterte Features der Datenbanken zurückgreifen.

Je nachdem, welche Aufgabe Sie mit Ihrer Applikation lösen müssen, haben Sie die freie Wahl, auf die dafür passende Datenbank zurückzugreifen.

Im nächsten Kapitel erfahren Sie mehr über verschiedene Qualitätssicherungsmethoden in Node.js-Applikationen wie beispielsweise die statische Codeanalyse oder die Absicherung Ihrer Software durch Unittests.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z