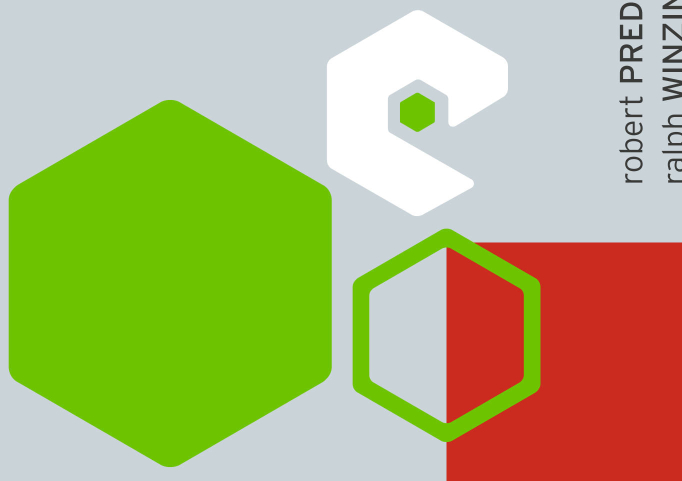
robert PREDIGER
ralph WINZINGER



NODE.js

Professionell hochperformante Software entwickeln

HANSER

Prediger/Winzinger

Node.js

Bleiben Sie auf dem Laufenden!

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

www.hanser-fachbuch.de/newsletter

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

www.hanser-fachbuch.de/update

Robert Prediger
Ralph Winzinger

Node.js

Professionell hochperformante Software entwickeln

HANSER

Die Autoren:
Robert Prediger, Wang
Ralph Winzinger, Nürnberg

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

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

Bibliografische Information der Deutschen Nationalbibliothek:

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

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

© 2015 Carl Hanser Verlag München, www.hanser-fachbuch.de
Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Herstellung: Irene Weilhart
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Stephan Rönigk
Gesamtherstellung: Kösel, Krugzell
Printed in Germany

Print-ISBN: 978-3-446-43722-7

E-Book-ISBN: 978-3-446-43758-6

Inhalt

Vorwort XI
... und ihre Motivation XIII
Das Zielpublikum XIII
Das Buch XIV
Die Welt von JavaScript XV

1 Hello, Node.js 1
1.1 Einführung in Node.js 1
1.2 Installation 8
1.2.1 Windows 8
1.2.2 Mac OS X 8
1.2.3 Debian 9
1.2.4 Ubuntu 9
1.2.5 openSUSE und SLE 10
1.2.6 Fedora 11
1.2.7 RHEL und CentOS 11
1.3 IDEs 11
1.3.1 cloud9 12
1.3.2 WebStorm 14
1.3.3 Nodeclipse 15
1.3.4 WebMatrix/VisualStudio 16
1.3.5 Atom 16
1.4 nvm & nodist – mit Node-Versionen jonglieren 17
1.4.1 *ix-Systeme 18
1.4.2 Windows 19
1.5 npm – Node Packaged Modules 21
1.5.1 npm install – ein Modul laden 22
1.5.2 Global? Lokal? 23
1.5.3 package.json 24
1.5.4 Module patchen 24
1.5.5 Browserify 28
1.6 Kein Code? 30

2 You build it ... 35

2.1 File I/O ... 36
2.1.1 Dateifunktionen in Node.js ... 36
2.1.2 Permissions ... 39
2.1.3 „watch“ – Änderungen im Auge behalten ... 40
2.1.4 Erweiterungen ... 41
2.1.4.1 Modul „fs-extra“ ... 42
2.1.4.2 Modul „file“ ... 43
2.1.4.3 Modul „find“ ... 43
2.1.4.4 Modul „properties“ ... 44
2.1.4.5 Modul „token-filter“ ... 45

2.2 Streams ... 46
2.2.1 Aus Streams lesen ... 47
2.2.1.1 Objekte und Strings ... 48
2.2.2 ... und in Streams schreiben ... 49
2.2.2.1 Streams verknüpfen ... 49
2.2.3 Eigene Streams implementieren ... 50
2.2.3.1 Ein Random-Number-Generator ... 51
2.2.3.2 Ein Daten-Lösch-Stream ... 53
2.2.3.3 Ein Verschlüsselungsserver für geheime Botschaften ... 54
2.2.4 Buffers and Strings ... 56

2.3 Daten für immer ... 57
2.3.1 Neo4j ... 57
2.3.1.1 Asynchron? ... 60
2.3.1.2 Querying Neo4j ... 61
2.3.1.3 Cypher für Abfragen ... 62
2.3.1.4 Indizes ... 64
2.3.1.5 Cypher für Batches ... 65
2.3.2 MongoDB ... 66
2.3.2.1 Wann sind Daten geschrieben? ... 67
2.3.2.2 _id ... 68
2.3.2.3 Die Mongo-API ... 68

2.4 Sichtbarkeit erzeugen – im Web ... 73
2.4.1 Middleware Framework Connect ... 73
2.4.1.1 Installation und einführendes Beispiel ... 74
2.4.1.2 Ausprägungen von Connect-Middleware-Typen ... 75
2.4.1.3 Integrierte Middleware-Komponenten ... 77
2.4.1.4 Middleware-Strukturen ... 85
2.4.2 Webentwicklung mit Express ... 90
2.4.2.1 Ready for take off: Installation und Einführungsbeispiel ... 91
2.4.2.2 Routing von HTTP-Anfragen ... 94
2.4.2.3 Views und Web-Templating ... 98
2.4.3 Express 4 ... 99

2.4.4 Jade . . . 101
2.4.4.1 Einbindung in Express . . . 103
2.4.4.2 Sprachelemente von Jade . . . 103
2.4.5 swig . . . 116
2.4.5.1 Grundeinstellungen . . . 116
2.4.5.2 Einbindung in Express . . . 117
2.4.5.3 Sprachelemente von swig . . . 118
2.4.5.4 Filterliste . . . 121
2.4.5.5 Verketten von Filtern . . . 124
2.4.5.6 Die swig-API . . . 124
2.4.5.7 Eigene Funktionalitäten hinzufügen . . . 126
2.4.6 Sessions & Authentifizierung . . . 127
2.4.6.1 Ich will Kekse und biete dafür eine Session . . . 128
2.4.6.2 Authentifizierung (Authentication) . . . 130
2.4.6.3 Facebook . . . 133
2.4.6.4 Twitter . . . 134
2.4.6.5 Google . . . 135
2.5 socket.io . . . 136
2.5.1 Verbindung herstellen . . . 137
2.5.2 Kommunikation . . . 138
2.5.3 Broadcast . . . 139
2.5.4 Private Daten . . . 139
2.5.5 Rückantwort und Bestätigung . . . 139
2.5.6 Namespaces . . . 140
2.5.7 Räume . . . 141
2.5.8 Autorisierung . . . 143
2.5.8.1 Globale Autorisierung . . . 143
2.5.8.2 Autorierung mit Namespaceses . . . 144
2.5.8.3 Benutzerdefinierte Variablen und Autorisierung . . . 145
2.5.9 Sessions mit „socket.io-session“ . . . 145
2.5.9.1 socket.io-bundle . . . 145
2.5.9.2 socket.io-passport . . . 146
2.5.10 Version 1.0 . . . 147
2.6 Node.js und Webservices . . . 151
2.6.1 SOAP-Services . . . 151
2.6.1.1 Von und nach SOAP . . . 153
2.6.2 REST-Services . . . 163
2.6.2.1 Von Nomen, Verben und Routen . . . 164
2.6.2.2 Ansichtssache? Verhandlungssache . . . 168
2.6.2.3 Fehlermeldungen . . . 170
2.6.2.4 Plug-ins . . . 171
2.6.2.5 Sicherheit und Authentifizierung . . . 176
2.6.3 XML-Verarbeitung . . . 183
2.6.3.1 XML-Parsing . . . 183

2.6.3.2 XML-Erzeugung und -Veränderung 189
2.6.3.3 Exkurs: Ein (selbst unterschriebenes) Zertifikat erstellen 191

2.7 Clustering 193
2.7.1 Methoden und Eigenschaften von cluster 197
2.7.1.1 isMaster/isWorker 197
2.7.1.2 fork/online – Event 197
2.7.1.3 exit – Event 198
2.7.1.4 workers 198
2.7.2 Der Master 198
2.7.2.1 setupMaster() 199
2.7.2.2 fork() 200
2.7.2.3 disconnect() 200
2.7.3 Der Worker 201
2.7.3.1 Die Attribute „id“ und „process“ 201
2.7.3.2 Das suicide-Attribut 201
2.7.3.3 kill() & disconnect() 201

2.8 Der Callback-Hölle entfliehen 202
2.8.1 async 203
2.8.1.1 Kontrollfluss 205
2.8.2 Q 212
2.8.2.1 then 214
2.8.2.2 fail 215
2.8.2.3 progress 215

2.9 Auf Herz und Nieren – Node.js-Anwendungen testen 216
2.9.1 Mocha 217
2.9.1.1 Asynchrone Aufrufe und Timeouts 220
2.9.1.2 Set-Up & Tear-Down 222
2.9.1.3 Only & Skip 223
2.9.1.4 Mocha im Browser 223
2.9.2 Assert & Chai 225
2.9.2.1 Assert 225
2.9.2.2 Chai 227
2.9.3 Sinon 232
2.9.3.1 Spies 234
2.9.3.2 Stubs 235
2.9.3.3 Mocks 236
2.9.3.4 Faked Timers 237
2.9.4 Jasmine 238
2.9.5 Continuous Test 239
2.9.5.1 Mocha & Jasmine im Überwachungsmodus 239
2.9.5.2 Travis-CI 240

3 ... you run it! 245
3.1 Eigene Module publizieren 245
3.1.1 Patterns & Style 246
3.1.1.1 package.json 247
3.1.1.2 Import & Export 248
3.1.1.3 Tests 249
3.1.1.4 Dokumentation 250
3.1.2 Ausführbare Module 252
3.1.3 Module mit nativen Abhängigkeiten 254
3.1.3.1 OS Libraries 255
3.1.3.2 Sourcecode Dependencies 256
3.1.3.3 Hands-On mit Add-On 257
3.1.4 It works on my machine – Dependency Hell 266
3.1.5 Veröffentlichung von Modulen 269
3.1.5.1 Einen Benutzer erzeugen 269
3.1.5.2 ... und das Modul publizieren 269
3.2 Private Repositories für npm 270
3.2.1 reggie 271
3.2.1.1 Inbetriebnahme 271
3.2.1.2 reggie publish 272
3.2.1.3 Laden von Modulen 272
3.2.1.4 HTTP-Abfragen 274
3.2.1.5 npm-Client 274
3.2.2 sinopia 275
3.3 Deployment 277
3.3.1 Ein eigener Server 278
3.3.1.1 Docker 278
3.3.1.2 Modul „forever“ 280
3.3.1.3 pm2 284
3.3.1.4 git-deploy 290
3.3.2 Cloud 291
3.3.2.1 PaaS-Provider 291
3.3.2.2 Server-Systeme 295
3.4 Was Node.js antreibt ... V8 Engine 296
3.4.1 Architektur 297
3.4.2 Die Performance-Tricks 299
3.4.2.1 „Fast Property Access“ 300
3.4.2.2 Arrays 301
3.4.2.3 Kein Interpretationsspielraum 302
3.4.2.4 Garbage Collection 302
3.4.2.5 Caching Modules 303
3.5 Logging 304
3.5.1 debug 304
3.5.2 winston 307

3.5.2.1 Transportmechanismen ... 307
3.5.2.2 Logger-Instanz ... 308
3.5.2.3 Logging Levels ... 309
3.5.2.4 Strukturierte Daten loggen ... 309
3.5.2.5 Profiling ... 310
3.5.3 Bunyan ... 311
3.5.3.1 Konfiguration ... 312
3.5.3.2 Child Logger ... 313
3.5.3.3 Die „src“-Option ... 314
3.5.3.4 Streams ... 314
3.6 Debugging ... 315
3.6.1 Der Node-Debugger ... 315
3.6.2 Node-Inspector ... 318
3.7 Monitoring ... 321
3.7.1 Kommerzielle Monitoring-Services ... 323
3.7.1.1 New Relic ... 323
3.7.1.2 Nodetime ... 325
3.7.1.3 StrongOps ... 329
3.8 Alternativen zu Node.js ... 334
3.8.1 Vert.x – die polyglotte JVM-Alternative ... 335
3.8.1.1 Architektur ... 335
3.8.1.2 Hands-On ... 342
3.8.1.3 Node.js oder Vert.x? ... 347

Index ... 349

Vorwort

Als Erstes möchten wir uns an dieser Stelle für Ihr Interesse an unserem Buch bedanken und die Gelegenheit nutzen, ein paar Worte über unser Buch zu verlieren. Das heißt, wir wollen uns, die Autoren, kurz vorstellen und unsere Motivation erklären, dieses Buch zu schreiben. Außerdem gibt es natürlich das eine oder andere über den Aufbau und das Format zu sagen, über die enthaltenen Beispiele und über JavaScript.

Über die Autoren ...

Es war von Anfang an klar, dass dieses Buch ein Gemeinschaftsprojekt sein soll. JavaScript – und damit auch Node.js – leben mit dem Ruf, keine professionelle Entwicklungsumgebung darzustellen. Insbesondere stößt man in der „Enterprise"-Welt auf diese Meinung. Um das objektiv zu hinterfragen, braucht es aber Beitragende aus beiden Lagern. So sind wir – Robert und Ralph – letztendlich aufeinandergetroffen, der eine seit erstaunlich langer Zeit professionell im Node.js-Umfeld tätig, der andere seit vielen Jahren als Software-Architekt in der Java-Enterprise-Welt unterwegs. Einer, der eine gewisse Erwartungshaltung mitbringt, der andere, der diese Erwartungshaltung mit den richtigen Bibliotheken, Tools und Methoden adressieren muss. Für uns war es eine fruchtbare Zusammenarbeit und bedeutete obendrein noch großen Spaß. Wir hoffen, beides in den kommenden Kapiteln weitergeben zu können.

Robert Prediger

Ich bin Robert, der Node.js-Freak. Mein größtes Projekt vor meiner Node.js-Zeit war die Entwicklung eines Accounting-Systems für internationale Hotelketten in Progress. Mit der Erstellung von Webapplikationen beschäftige ich mich seit gut 15 Jahren.

Vor knapp vier Jahren bin ich auf Node.js gestoßen und mich hat die Umgebung von Anfang an, vor allem aufgrund ihrer Stabilität und Geschwindigkeit, überzeugt und fasziniert. Mittlerweile sind diverse Projekte mit Node.js im Rahmen meiner web4biz Consulting ans Laufen gebracht worden.

Seit kurzem bin ich als Co-Founder an der whogloo Inc. beteiligt, deren Ziel es ist, eine Plattform zur Erstellung von Enterprise-Business-Applikationen zu erstellen – auf Basis von Node.js natürlich.

So habe ich in meinem täglichen Umfeld permanent mit Node.js zu tun. Das ist anstrengend, denn die Technik ist nach wie vor neu, teilweise bereits den Kinderschuhen entwachsen, dennoch sehr lernintensiv, wenn man permanent auf dem Laufenden bleiben will. Und das muss man auch, denn die Entwicklung in diesem Umfeld ist rasend schnell. Aber bereut habe ich es bisher nicht, mich komplett auf Node.js einzulassen. Es macht immer wieder Spaß und es fasziniert stets aufs Neue, mit welch teilweise einfachen Mitteln man Programme entwickeln kann, die auch einem hohen Standard genügen.

Und ich bin gespannt, wo uns die Reise mit Node.js noch hinführt.

Ralph Winzinger

Ich bin Ralph, der Java-Architekt. Ich arbeite für Senacor Technologies, ein Beratungsunternehmen mit vielen großen Kunden aus der Finanzwelt, bei denen ich Architekturen geplant, Entwicklungsprozesse eingeführt und Software erstellt habe, gerne auch mal auf der Frontend-Seite, aber hauptsächlich im Backend. EJBs, Webservices, Spring, JPA … das ist in der Regel meine Welt. Und zugegeben, ein großer Freund von JavaScript war ich lange Zeit nicht. Ich glaube, mich noch ungefähr an meine erste Reaktion auf Node.js erinnern zu können: „Serverside JavaScript? Wer braucht denn bitte eine Alert-Box auf dem Server?!?“ Wie gesagt, das war die allererste, spontane Reaktion.

Ich bin aber nicht mit der Java-Enterprise-Welt verheiratet. Es macht mir Spaß, neue Technologien zu ergründen und mir so manche Nacht mit Sourcecode um die Ohren zu schlagen. So ist es nicht verwunderlich, dass mich der Charakter von Node.js bald fasziniert hat.

„High-Scalability“ ist ein Thema, das auch im Enterprise-Umfeld immer wichtiger wird. Eine riesige Zahl von mobilen Endgeräten, die in jeder Branche zu einem extrem wichtigen Kundenkanal geworden sind, und die Entwicklung im Feld von „Internet of Things“ verbieten es geradezu, sich weiter exklusiv auf traditionelle Java Application Container und die damit verbundenen Technologien zu verlassen.

Ob sich ausgerechnet Node.js in meinen Projekten und bei meinen Kunden jemals etablieren wird, ist derzeit noch schwer zu beurteilen. Aber den Konzepten von Node.js wird sich unsere Branche sicherlich nicht verschließen können. Und ich hätte nun meine Meinung zum professionellen Einsatz von JavaScript.

… ihre Helfer …

Zum Buch haben nur wir beide beigetragen? Nein. Eigentlich sollten vier Namen auf dem Cover stehen. Zwei geschätzte Kollegen und sehr gute Freunde haben diese Erkundungstour mit uns gestartet. Teils aus privaten, teils aus beruflichen Gründen mussten aber beide unser Buchprojekt verlassen, bevor wir es zum Abschluss bringen konnten. Trotzdem haben sie wichtige Impulse gegeben und auch wichtige Inhalte beigetragen. Wir sprechen von Victor Volle und Charles-Tom Kalleppally, beide in der Java-Welt verwurzelt, sehr versiert im Umgang mit Architektur, Design und Code und ebenfalls immer bereit, ausgetretene Pfade zu verlassen und sich auch mal neuen Technologien zuzuwenden.

An dieser Stelle großen Dank an Victor und Charly!

… und ihre Motivation

Unsere Motivation haben wir in den einleitenden Sätzen ja schon skizziert: Der Node.js-Mensch wollte eine Lanze für seine Technologie brechen und der Java-Mensch „mal was Verrücktes tun“ – mit dem Ziel, die Entwicklung mit Node.js und JavaScript ein wenig ins rechte Licht zu rücken.

Wir haben die Maßstäbe aus der Java-Enterprise-Entwicklung Node.js und JavaScript angewandt, sowohl technisch als auch methodisch. Toolunterstützung für eine effiziente Arbeitsweise, Qualitätssicherung, um höchsten Ansprüchen zu genügen, State-of-the-Art Deployment und Monitoring … all das haben wir uns angesehen und die aus unserer Sicht und zum aktuellen Zeitpunkt besten oder vielversprechendsten Ansätze aus dem Node.js-Umfeld zusammengetragen.

Wir sind der Meinung, dass es für Node.js oder verwandte Technologien auch im Enterprise-Umfeld Bedarf gibt und dass sich Node hier durchaus verwenden lässt. Nichtsdestotrotz bleibt es zu einem gewissen Teil vorerst noch Kopfsache, ob man Node.js und JavaScript das nötige Vertrauen schenkt, geschäftskritische Teile der Systemlandschaft zu realisieren.

Das Zielpublikum

Wir richten uns mit diesem Buch ganz klar an Entwickler und entwickelnde Architekten. Es ist ein sehr technisches Buch und wir gehen an vielen Stellen davon aus, dass entsprechendes Wissen vorhanden ist, um den einen oder anderen Sachverhalt zu verstehen.

Es werden vor allem diejenigen Freude am Buch haben, die nicht einfach nur möglichst viel Code in möglichst kurzer Zeit produzieren wollen. Das Design des Codes, die Paradigmen der Laufzeitumgebung, aber auch der Betrieb von Enterprise-Software sind uns wichtig und sollten es auch unseren Lesern sein.

Und nicht zuletzt wünschen wir uns ein wenig Spieltrieb. JavaScript ist eine sehr flexible Sprache, Node.js ein sehr schnell wachsendes Ökosystem, in dem es immer wieder Neues zu entdecken gibt. Man kann es sich hier noch viel weniger leisten, den Blick für die aktuelle Entwicklung und aktuelle Trends zu verlieren, als das im eher trägen Enterprise-Umfeld der Fall ist.

Die Arbeit mit JavaScript, mit seinem dynamischen Typsystem und den offenen Sourcecodes führt auch ganz natürlich dazu, dass man sich immer wieder in den Tiefen von fremdem Code verliert, um herauszufinden, wie eine Bibliothek funktioniert, welche versteckten Optionen eine Funktion eventuell noch bietet.

Das alles muss man mögen. Wir mögen es und wir hoffen, unsere Leser ebenfalls.

Das Buch

Es ist nicht nur wichtig, wer ein Buch verfasst hat und weshalb ein Buch geschrieben wurde. Nachdem wir einen bestimmten internen Aufbau verfolgt haben und auch über die verwendete Formatierung die Verständlichkeit erhöhen wollten, möchten wir unsere Richtlinien an dieser Stelle kurz skizzieren.

Aufbau

Wir haben unser Buch in drei Teile gegliedert.

Der erste Teil soll dafür sorgen, dass sich unsere Leser langsam an die Welt von Node.js gewöhnen. Die Geschichte, die Idee, das Tooling stehen hier im Vordergrund. Die Entwicklung an sich ist noch kein Thema, allenfalls am Rande.

Im zweiten und weitaus größten Teil bewegen wir uns durch die Schichten und Problemfelder einer Enterprise-Anwendung. Angefangen beim Dateisystem über Webanwendungen und Service-Schnittstellen bis hin zu einer ausgefeilten Testunterstützung werden die typischen Fragestellungen anhand von vielen vorgestellten Modulen und Codebeispielen beantwortet.

Der letzte Teil kümmert sich im Wesentlichen um die Zeit nach der Entwicklung. Wie nehme ich eine solche Anwendung in Betrieb und wie halte ich sie in Betrieb? Deployment, Monitoring oder auch Performance sind die Themen in diesem Teil. Und nicht zuletzt ein Blick auf eine Alternative – ähnliche Konzepte und Performance, aber nicht (nur) JavaScript.

Formate

Bezüglich der verwendeten Formate haben wir uns zurückgehalten. Natürlich sind alle coderelevanten Passagen als solche zu erkennen. Schreiben wir beispielsweise im Text über Code, so ist das wie hier `console.log("this is code")` entsprechend gekennzeichnet. Längere Codepassagen sind natürlich nicht im Fließtext untergebracht, sondern erhalten ihren eigenen Abschnitt:

```
var fs = require("fs");
var i = 5;
console.log("this is still code")
```

Auf dieselbe Art sind auch Kommandozeileninteraktionen formatiert. Auch diese können im Fließtext (`npm install express`) oder in einem größeren Block zu finden sein.

```
$ node app.js
INFO: this is some console output
```

Und dann sind da noch Modulangaben. Beziehen wir uns im Text auf Module wie *Express* oder *restify*, so sind diese immer kursiv dargestellt. Manchmal gibt es in diesem Zusammenhang auch eine gewisse Grauzone. So kann es sein, dass man von dem Modul *npm* spricht oder aber von dem Kommando `npm`. Damit ist dasselbe Wort dann vielleicht auch im selben Abschnitt verschieden formatiert.

Beispiele

Die allermeisten unserer Beispiele sind tatsächlich funktionierender Code, der seine Korrektheit zunächst in einem Test oder wenigstens durch eine Ausführung unter Beweis stellen musste. Erst dann durfte er aus der IDE über copy & paste ins Buch springen. Natürlich können auch dabei immer wieder mal Fehler passieren, die bitten wir zu entschuldigen.

Die Codebeispiele

Die Codebeispiele in diesem Buch müssen natürlich nicht abgetippt werden; sie stehen Ihnen online auf einem GitHub Repository zur Verfügung:

https://github.com/WinzingerPrediger/node.js

Die Welt von JavaScript

Eines liegt uns noch sehr am Herzen. Wie erwähnt, ist die Welt von JavaScript und Node.js sehr kurzlebig. Ein Modul, das heute noch *das* Modul für eine bestimmte Problemstellung ist, wird morgen vielleicht schon von einem neuen Modul abgelöst, welches riesigen Zuspruch in der Community erhält und schnell zum De-facto-Standard wird.

Natürlich stellen wir im Buch ganz konkrete Module vor und laufen damit potenziell Gefahr, dass diese einige Monate nach dem Erscheinungstermin des Buchs schon nicht mehr existieren oder niemanden mehr interessieren. Wir haben aber auch immer versucht, die aus unserer Sicht relevante Funktionalität herauszustellen – welches Problem wird von dem Modul eigentlich gelöst? Mit diesen Hintergrundinformationen sollte es einfacher sein, die Eignung neuer Module zu hinterfragen oder gezielt nach Alternativen zu suchen.

„Lerne ich hier JavaScript?"

Diese Frage lässt sich einfach beantworten: nein. Vielleicht kann man sich an der einen oder anderen Stelle einen kleinen Kniff abschauen, aber wir geben hier noch nicht mal eine gezielte Einführung in JavaScript.

Es gibt sehr viel Literatur und sehr viele Quellen im Internet, die das leisten können. Wir haben uns das nicht zum Ziel gesetzt.

Viel Spaß beim Lesen, Lernen und Spielen,

Robert und Ralph

1 Hello, Node.js

Was ist Node.js eigentlich und wie arbeitet man nun damit? Das ist die zentrale Frage, die im ersten Teil des Buchs beantwortet werden soll. Hier geht es nicht darum, welche Bibliotheken bei der Erstellung großer Softwareprojekte besonders hilfreich sind oder wie man dafür sorgt, dass eine Node.js-Anwendung stabil läuft, auch wenn sie eine Unmenge von Anfragen abarbeiten muss. Nein, dieser Teil des Buchs liegt zeitlich vor den ersten Zeilen Anwendungscode, die hoffentlich bald entstehen.

Am Anfang dürfen natürlich ein paar einführende Worte zu Node.js nicht fehlen. Woher kommt dieses Node eigentlich und was ist daran besonders? Die technischen Tiefen werden erst im letzten Teil des Buchs erreicht, aber eine Idee von der Funktionsweise sollte man trotzdem von Anfang an haben.

Und neben der Idee bedarf es natürlich auch noch des notwendigen Handwerkszeugs. Node.js braucht eine Laufzeitumgebung – für welche Systeme existiert die und wo kann man sie bekommen? Mit welchen Tools kann Quellcode editiert werden? Wo findet man Bibliotheken und wie kann man diese benutzen? Wenn auch diese Fragen alle geklärt sind, hat man keine Ausreden mehr, sich die Finger am Code schmutzig zu machen. Und auch wenn man kein Enterprise-System realisieren möchte – Spaß kann man mit Node.js allemal haben. Und das ist schon mal ein ganz wichtiger Punkt!

1.1 Einführung in Node.js

Node.js hat ein unglaubliches Wachstum hingelegt. Von der ersten Ankündigung durch Ryan Dahl 2009[1] bis heute (Ende 2014) sind nur fünf Jahre vergangen. Das Node.js Repository auf GitHub ist aktuell auf Platz zwei der „most starred repositories“[2] (und unter den ersten 15 der „most forked repositories“[3]).

[1] Auf der Seite von JSConf.eu 2009 ist das Video von Ryans Vortrag leider nicht mehr verfügbar, wohl aber eine Anmerkung zum Video: „Node.js might be the most exciting single piece of software in the current JavaScript universe. Ryan received standing ovations for his talk and he really deserved it!“. Inzwischen ist der Vortrag auf YouTube zu sehen: https://www.youtube.com/watch?v=EeYvFl7li9E

[2] https://github.com/search?q=stars%3a%3E1&s=stars&type=Repositories

[3] https://github.com/search?o=desc&p=1&q=stars%3A%3E1&s=forks&type=Repositories

Auf der anderen Seite gibt es auch einen geradezu aggressiven „Backlash“[4]. Natürlich ist das „trolling“ – sinnvolle Argumente sind in solchen Fällen nur selten auszumachen. Auch Beiträge wie „Node.JS Is Stupid And If You Use It So Are You“[5] fallen ganz klar in diese Kategorie.

Warum ist das so? Node.js ist immer noch vergleichsweise neu, hat sich aber schon an vielen Stellen einen festen Platz in der Anwendungslandschaft ergattert. Nichtsdestotrotz gibt es aber immer noch eine Art missionarischen Effekt. Überzeugte Anwender möchten Node.js weiter aus seiner Nische holen und versuchen, andere von seinen Vorteilen zu überzeugen. Dann gibt es natürlich auch den gegenläufigen Effekt: Jemand probiert etwas aus, von dem er gerade Lobpreisungen über sich hat ergehen lassen. Und dann passt es nicht zu seinem Problem, seinen Erwartungen oder einfach seiner aktuellen Denkweise. Er hat Zeit investiert und ist enttäuscht. Also muss die Technologie nutzlos, unsinnig oder überflüssig sein. Unter Umständen reicht für viele Entwickler alleine schon die Tatsache, dass Node.js auf JavaScript basiert, um es als indiskutabel einzustufen.

Weshalb haben wir nun ein Buch über Node.js geschrieben? Nicht weil wir missionieren möchten und denken, dass Node.js das Beste seit geschnitten Brot ist. Es gibt Situationen, in denen Node.js eine gute Lösung darstellt, aber auch Situationen, in denen man vielleicht zu einer Alternative greifen sollte. In den nächsten Kapiteln versuchen wir nicht nur Vorteile, sondern auch Nachteile zu zeigen – insbesondere auch im Hinblick auf „professionelle Softwareentwicklung“. JavaScript hat nicht den besten Ruf in unserer Industrie und es wird schnell behauptet, dass es nicht als Basis für unternehmenskritische Anwendungen dienen kann. Es ist natürlich schwierig, hier absolute K. O.-Kriterien für oder gegen eine solche Verwendung zu finden – Node.js ist sicherlich erwachsen genug, um nicht sofort auszuscheiden. Wir versuchen aber, viele Aspekte zu betrachten, die in den Bereich der professionellen Softwareentwicklung fallen, und beleuchten, wie diese Aspekte in Node.js und JavaScript behandelt werden. Unserer Meinung nach schneiden Node.js und JavaScript hier nicht immer optimal ab. Allerdings hängt es vom konkreten Szenario ab, ob ein ganz bestimmter Aspekt nun benötigt wird oder nicht. Node.js hat es auf jeden Fall verdient, eine Chance zu bekommen. Es kann vielleicht nicht in allen Bereichen mit etablierten Sprachen und Umgebungen wie dem Java-Ökosystem mithalten, aber es gibt durchaus Situationen, in denen Node.js einfach auf der Überholspur vorbeizieht …

Also, was ist der Nutzen von Node.js? Oder besser: Welches Problem löst Node.js besser als andere Technologien?

Eines der wichtigsten Probleme für Internetanwendungen ist Skalierbarkeit. Wenn die Benutzeranzahl steigt, soll der Ressourcenverbrauch nach Möglichkeit linear steigen. Von den relevanten Ressourcen CPU, I/O und RAM soll Node.js vor allem die Skalierbarkeit bei I/O-intensiven Anwendungen verbessern.

Um dies zu erreichen, setzt Node.js vollständig auf asynchrone I/O-Zugriffe. Wann immer Node.js auf eine Datenbank, einen Webservice oder auf das Dateisystem zugreift, erfolgt der Aufruf asynchron. Was macht das für einen Unterschied?

Im synchronen Fall könnte ein Datenbankaufruf folgendermaßen aussehen:

---

[4] https://www.semitwist.com/mirror/node-js-is-cancer.html

[5] https://www.youtube.com/watch?v=1e1zzna-dNw

```
result = database.query("SELECT * FROM user");
result.get…
```

Der aktuelle Thread wartet auf das Ergebnis und wird „geweckt“, wenn das Ergebnis vorliegt (s. Bild 1.1). Dieses Verfahren funktioniert gut und ist aus konzeptioneller Sicht leicht zu begreifen. Als Entwickler kann man so programmieren, als ob man keine Nebenläufigkeit hätte. Man muss sich (meist) nicht um Synchronisation kümmern. Und das Wichtigste: Die Kosten für das Wechseln von einem Thread zum nächsten sind vernachlässigbar.

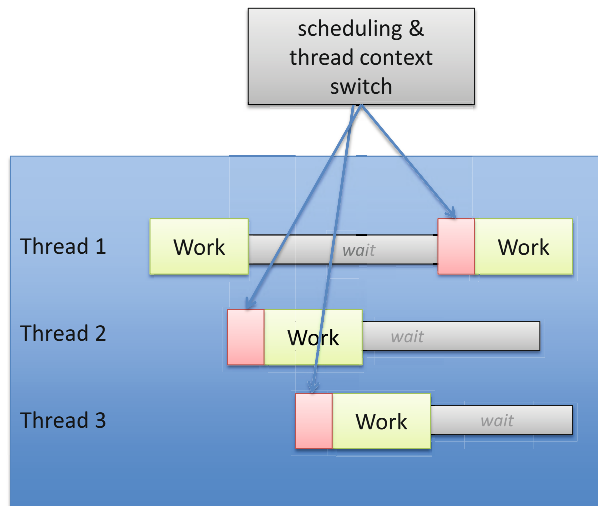


Bild 1.1 multi-threaded Server

Doch was tun, wenn die Kosten für den Wechsel nicht mehr vernachlässigbar sind? Wenn auf den „Thread Context Switch“ ein signifikanter Anteil an der Gesamtlaufzeit entfällt?

Hinzu kommt, dass Threads Hauptspeicher verbrauchen. Ein Java-Applikationsserver hat oft 200 bis 300 Threads. Auf einem 64-Bit-Linux-System wird für einen Thread 1 MB für den Stack reserviert. Somit werden ca. 250 MB verbraucht, die nur für die Threads benötigt werden. In vielen Anwendungen spielt das keine Rolle, da für den Heap allein 8 GB veranschlagt werden. Doch wenn die Anwendung

- sehr viele parallele Zugriffe hat,
- viel I/O (Datenbank, Dateisystem etc.) benötigt,

wird es eng. Was heißt in diesem Zusammenhang „viel“? 100 Zugriffe pro Minute sind (unserer *Meinung* nach) noch nicht viel. Spätestens bei 100 parallelen Zugriffen pro Sekunde wird Node.js definitiv interessant.

Keine Threads!

Was macht Node.js anders? Node.js verwendet nur einen Thread. Das ist alles. Nun ja, das ist nun etwas plakativ formuliert und technisch auch nicht ganz korrekt, aber in einer ersten Näherung ist es das, was Node.js anders macht als die meisten gewohnten Umgebungen.

Wie handhabt Node.js dann Hunderte von parallelen Zugriffen? Es arbeitet einfach einen Auftrag nach dem anderen ab – und wann immer dann auf einen langsamen I/O-Zugriff gewartet werden muss, wird der aktuelle Auftrag einfach beendet und ein neuer Auftrag für die Verarbeitung der Antwort erzeugt.

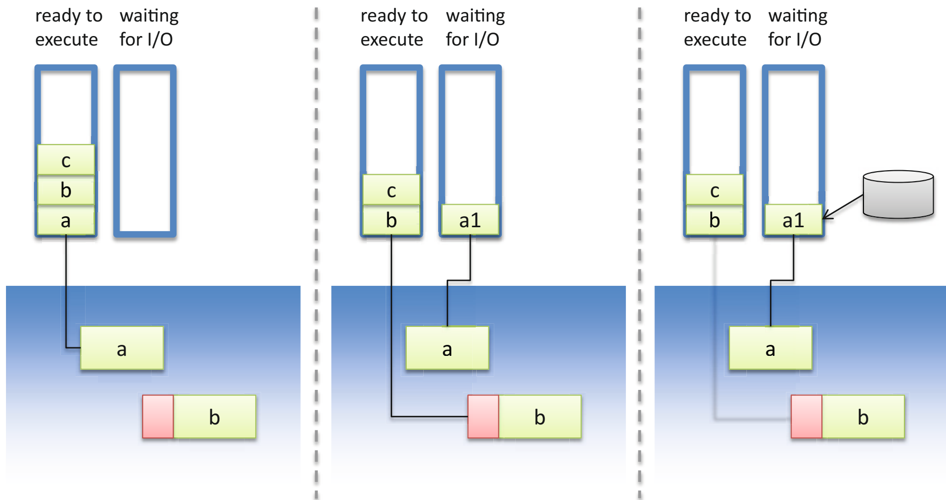


Bild 1.2 Node.js Event-Loop

In der Abbildung „Node.js Event-Loop“ ist der Ablauf – stark vereinfacht – skizziert. Es gibt eine Warteschlange mit Aufträgen, die bereit zur Verarbeitung sind („ready to execute“). Sobald der Verarbeitungs-Thread frei ist, beginnt er mit der Abarbeitung des Auftrags „a“. Wenn dann jedoch ein I/O-Zugriff erfolgt, wird die Abarbeitung des Auftrags beendet und ein Folgeauftrag „a1“ erzeugt. Dieser Auftrag wird jedoch erst dann in die „ready-to-execute“-Queue eingestellt, wenn der I/O-Zugriff abgeschlossen ist.

Bei Node.js muss der Entwickler jeden Folgeauftrag als „Callback-Funktion“ selbst definieren – ein Vorgehen, das jedem JavaScript-Entwickler sehr vertraut sein dürfte und zu Code wie dem folgenden führt:

```
database.query(
  "SELECT * FROM user",
  function(result) {
    result…
  }
);
```

Wirklich schwer zu lesen wird es, wenn bei der Verarbeitung der Antwort wiederum ein I/O-Zugriff mit der nächsten Callback-Funktion zu schreiben ist. Statt der geschachtelten anonymen Funktionen kann man natürlich auch mit explizit benannten Funktionen arbeiten:

```
function schritt1(result) {
  ...
  database.query("...", schritt2);
}
```

```
function schritt2(result) {
  ...
}

database.query("...", schritt1);
```

Damit wird es jedoch schwerer, den tatsächlichen Ablauf in den Codestrukturen nachzuvollziehen.

Da es vielen Programmierern so geht, gibt es eine große Menge an Node.js-Bibliotheken, die einen besser lesbaren Code ermöglichen. Zwei solche Bibliotheken werden im zweiten Teil im Abschnitt 2.8 gezeigt.

Wenn es keine Threads gibt, stellt sich die Frage: Wie nutzt man Multi-Prozessor und Multi-Core-Maschinen sinnvoll aus? Wenn man Node.js auf einer solchen Maschine laufen lässt, wäre gerade mal ein Core ausgelastet. Deshalb sollte man pro Core eine Node.js-Instanz laufen lassen und noch einen Core für andere Aufgaben freihalten. Das bedeutet aber auch, dass man sich um das Thema Session State kümmern muss.

Kriterien für den Einsatz von Node.js

Durch die Vermeidung von Threads benötigt Node.js tendenziell weniger Hauptspeicher und durch den Wegfall des Thread-Context-Wechsels auch weniger CPU-Zyklen; was aber – wie gesagt – erst bei einer hohen Anzahl konkurrierender Zugriffe relevant wird.

Machen aber die Kosten für die Server einen relevanten Anteil am Budget aus, sollte man Node.js wirklich in Betracht ziehen. Beispielsweise hat LinkedIn seine Mobile-App von Rails auf Node.js umgestellt und damit die Anzahl der benötigten Server von 30 auf drei reduzieren können[6]. Natürlich macht man bei einem „rewrite“ vieles besser als beim ersten Mal. Man kennt die kritischen Stellen, kann sich auch des angewachsenen Mülls entledigen: Die Reduktion lässt sich sicherlich nicht komplett Node.js zuschreiben. Dennoch sind wir (und LinkedIn) überzeugt, dass Node.js einen wichtigen Anteil an der Reduktion hat.

Neben den Vorteilen im Betrieb für Anwendungen mit hohem I/O und vielen parallelen Zugriffen gibt es noch einen Vorteil in der Entwicklungsphase: *eine* Sprache für das Frontend und das Backend. „The best tool for the job“ – also für das jeweilige Problem die passende Sprache zu verwenden – kann eine gute Idee sein. Aber dann läuft man Gefahr, eine sehr heterogene Anwendungslandschaft pflegen zu müssen.

Auf der anderen Seite führt die Trennung in Frontend- und Middleware-Team oft zu Barrieren im Austausch, Missverständnissen und doppelter Arbeit. Wenn man also *ein* Team aufbauen kann, das nur eine Programmiersprache benutzt, ist zumindest diese Barriere niedriger. Natürlich sind die benötigten Fähigkeiten für Frontend-Entwicklung anders als die für die Middleware-Entwicklung. Es sind andere Randbedingungen zu beachten. Aber ein Feature vom Frontend bis zu den Backend-Aufrufen von einem Entwickler bauen zu lassen, der – im wörtlichen Sinne – alles „von vorne bis hinten“ versteht und überblickt, reduziert Kommunikationsbarrieren immens. So meint zum Beispiel Ben Galbraith, Vice President for mobile engineering bei Walmart: „We’ve been fascinated for a long time by end-to-end JavaScript.“[7]

[6] http://ikaisays.com/2012/10/04/clearing-up-some-things-about-linkedin-mobiles-move-from-rails-to-node-js/

[7] http://venturebeat.com/2012/01/24/why-walmart-is-using-node-js/

Ab einer bestimmten Teamgröße kommen andere Effekte hinzu. Je größer das Team, desto mehr Zeit wird für Kommunikation benötigt. Ab acht bis zehn Menschen sollte man Teams aufteilen, um diesem Effekt entgegenzuwirken. Aber dennoch muss man sich gut überlegen, ob man die Aufteilung anhand der Applikationsschichten (also Frontend, Middleware, Backend) vornimmt oder entlang von Funktions-Clustern. Stimmen aus dem agilen Umfeld oder auch aus dem Umfeld von Microservices fordern vehement, dass ein Team eine funktionale Anforderung komplett umsetzen können muss.

Welche Technologie einzusetzen ist, hängt von zu vielen Einflussfaktoren ab, um eine einfache Empfehlung geben zu können. Wir möchten zwei Aspekte herausgreifen. Der eine ist eher „nichtfunktional“ und der andere eher kulturell.

Node.js ist, wie gesagt, gut geeignet für die Abarbeitung einer sehr hohen Anzahl von parallelen Requests. Aber das Abstraktionsniveau der Sprache ist eher niedrig. Die Frameworks, um beispielsweise Webanwendungen zu bauen, sind nicht so ausgereift wie man es etwa von Wicket oder JSF für Java oder Ruby on Rails gewohnt ist. Das kann ein Vorteil sein, weil man schneller einfache Anwendungen bauen kann und besser versteht, was genau passiert. Aber für Anwendungen, in denen nicht der Ressourcenverbrauch im Vordergrund steht, sondern komplexe Geschäftslogik mit vielen (oft bizarren) Ausnahmen, ist das sicherlich nicht die erste Wahl. Auch mit Rails kann man sehr schnell eine Anwendung bauen. Man kann schnell mit einer Datenbank arbeiten – bevorzugt mit einer Datenbank, die man im exklusiven Zugriff hat. Man kann Rails inzwischen robust und performant zum Laufen bekommen, aber eine große Anzahl von parallelen Zugriffen ist aufgrund der Single-Thread-Architektur und des „Globalen Interpreter Lock“ (GIL)[8] schwierig zu handeln.

Natürlich kann man auch JRuby verwenden, um mit nativen Threads zu arbeiten und den GIL zu umgehen oder auch mit Eventmachine[9], um asynchron und Event-basiert zu arbeiten wie in Node.js. Aber hier kommt ein weiterer Aspekt zum Tragen: Rails hatte im Enterprise-Umfeld lange Zeit einen schlechten Ruf, da es keine Prepared Statements und „Bind Variables“ für Oracle unterstützte. Bei der Verwendung von MySql spielt das vielleicht keine Rolle, aber Oracle-Administratoren lassen nur Anwendungen auf ihre Datenbank, die sich an die Regeln halten – und das heißt: Bind Variables und Prepared Statements verwenden[10].

Aber neben diesen – zumindest ansatzweise – objektivierbaren Kriterien spielt der kulturelle Aspekt eine Rolle: Entsteht die Anwendung in einem „Startup“ oder in einem großen Unternehmen („Enterprise“)? Auch große Unternehmen wie Walmart[11] oder LinkedIn[12] verwenden inzwischen Node.js. Ob ein Unternehmen diesen Weg einschlägt und einer neuen Technologie eine Chance gibt, ist eine Frage der Risikofreudigkeit und des Wertes von technischen Argumenten. In großen Unternehmen ist man an dieser Stelle oft sehr konservativ. Die Frage der langfristigen Pflege der Anwendung, der Abhängigkeit von ein paar wenigen Know-how-Trägern spielt oft eine größere Rolle als die Kosten für ein paar Server mehr. In einer solchen Umgebung bewegen auch wir uns sehr vorsichtig. Eine Empfehlung kann objektiv richtig sein, doch wenn der Empfänger nicht offen für eine solche Empfehlung ist,

[8] http://en.wikipedia.org/wiki/Global_Interpreter_Lock
[9] http://rubyeventmachine.com/
[10] http://www.akadia.com/services/ora_bind_variables.html
[11] http://venturebeat.com/2012/01/24/why-walmart-is-using-node-js/
[12] http://ikaisays.com/2012/10/04/clearing-up-some-things-about-linkedin-mobiles-move-from-rails-to-node-js/ „Clearing up some things about LinkedIn mobile’s move from Rails to node.js“

müssen wir uns nicht unnütz in Kämpfen aufreiben – „Choose your battles“. In einer Start-up-Kultur sind die Randbedingungen anders. Die Entscheider haben oft einen starken technischen Hintergrund und können technische Argumente selber beurteilen oder sie haben aufgrund der kleinen Teamgröße und der damit einhergehenden engen Zusammenarbeit ein größeres Vertrauen in die „Nerds“, die zudem viel stärker mit dem Erfolg oder Misserfolg des Startups verbunden sind als in großen Unternehmungen[13].

Die „Entscheidung für Node.js“ (Bild 1.3) lässt sich eigentlich nicht in ein einfaches Diagramm pressen. Es kann durchaus auch Anwendungen mit hoher „business complexity“ und „high concurrency“ geben, aber als Ausgangspunkt und Anhaltspunkt zur Einordnung von Node.js kann die Grafik durchaus herangezogen werden.

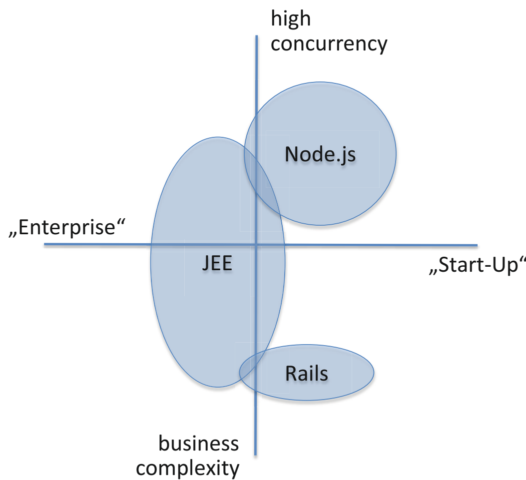


Bild 1.3 Entscheidung für Node.js

Alternativen

Es gibt natürlich noch weitere Alternativen als nur JEE und Rails oder das oben erwähnte Eventmachine. Uns erscheint vor allem Vert.x[14] interessant. Der Server läuft auf der JVM und erlaubt es in einer Vielzahl von Sprachen, unter anderem JavaScript, Python oder Java, zu programmieren. Vert.x basiert auf den gleichen Überlegungen zu Concurrency wie Node.js. Viele Libraries liegen jedoch nicht in einer asynchronen Variante vor. Datenbankbibliotheken blockieren den ausführenden Thread bis zur Antwort. Deshalb unterscheidet Vert.x zwischen normaler Eventverarbeitung durch sogenannte „Verticles“ und der Verarbeitung durch „Worker Verticles“, die jeweils einen Thread aus einem Threadpool zur Ausführung erhalten. Damit ist Vert.x zwar nicht mehr so klar und einfach wie Node.js, kann andererseits aber die riesige Palette an Java-Bibliotheken nutzen. Abschnitt 3.8 liefert mehr Informationen zu dieser interessanten Alternative.

[13] Es wäre interessant, das Gehalt von Mitarbeitern direkt mit dem Erfolg von deren Technologieempfehlungen zu verknüpfen. Und jedem Team 10 000 EUR zur Verfügung zu stellen, die sie entweder in Software, Konferenzen, Monitore etc. investieren oder sich am Jahresende auszahlen lassen können.

[14] http://vertx.io

1.2 Installation

Es gibt eigentlich keine echte Ausrede, Node.js nicht einfach mal auszuprobieren: Node.js ist für alle gängigen Betriebssysteme verfügbar und auf einfachste Art und Weise installierbar.

Für die meisten Systeme existieren bereits vorkonfigurierte Installationspakete. Diese können direkt auf der Webseite von Node.js unter

http://nodejs.org/download

heruntergeladen werden und man ist innerhalb von wenigen Minuten bereits in der Lage, das erste kleine Programm auszuführen.

Sollte für das gewünschte Betriebssystem kein Paket existieren, so lässt sich Node.js auch aus den Quelltexten heraus kompilieren.

Auf die Besonderheiten der einzelnen Betriebssysteme wird nachfolgend gesondert eingegangen.

1.2.1 Windows

Für Windows existiert sowohl ein Installer (`.msi`) als auch eine binäre Installationsdatei (`.exe`). Beide sind jeweils in einer 32- und 64-bit-Version verfügbar.

Die Installation unter Windows verläuft unter allen Versionen normalerweise reibungslos. Die Unterstützung der Node.js-Community durch Microsoft ist sehr gut. Node.js war die erste Umgebung, die mit nur einem Mausklick in Azure, der Cloud-Umgebung von Microsoft, übertragen werden konnte.

Alternativ können auch die Kommandozeilen-Installer `scoop`[15] oder `chocolatey`[16] benutzt werden. Für die Installation bedarf es in beiden Fällen jeweils nur einer knappen Befehlszeile:

```
scoop install node.js
```

```
cinst nodejs.install
```

Neben diesen Möglichkeiten gibt es auch noch das Tool `nodist`, auf das im Abschnitt 1.4.2 noch genauer eingegangen wird.

1.2.2 Mac OS X

Auch für Mac OS X können Installationspakete entweder als Installer (`.pkg`) oder als Archiv (`.tar.gz`) direkt von der Node.js-Webseite heruntergeladen werden.

[15] http://scoop.sh
[16] http://chocolatey.org

Alternativ kann homebrew oder macports benutzt werden, um die Installation und Updates per Kommandozeile zu vereinfachen. Beides sind Package-Manager, wie sie auch aus dem Linux- und Unix-Umfeld bekannt sind und die sich zum Ziel gesetzt haben, an den Stellen einzuspringen, an denen von Apple noch keine vorgefertigte Installation vorgesehen war. Ein populäres Beispiel für ein fehlendes Paket – wenigstens für einen Entwickler – dürfte wget sein. Spätestens wenn man zum ersten Mal überrascht die Zeile „-bash: wget: command not found“ gelesen hat, wird man kurze Zeit später auf homebrew oder macports stoßen.

Wie auch schon bei den Windows-Installern, ist die Installation mit den Max-OS-X-Package-Managern nahezu identisch:

```
brew install node
```

```
port install nodejs
```

Auch für Mac OS X und alle Unix-Derivate gibt es noch eine Alternative namens nvm, die im Abschnitt 1.4.1 gezeigt wird.

1.2.3 Debian

Node.js ist für Debian im offiziellen Package-Repository verfügbar. Allerdings hängt die freigegebene Version immer etwas hinterher – möchte man „bleeding edge“ arbeiten, so kommt man nicht umhin, Node.js entweder selbst zu kompilieren oder ebenfalls ein Tool wie nvm zu verwenden.

Die Installation über das Repository erfolgt ganz gewöhnlich über einen einzigen Befehl

```
apt-get install nodejs
```

Will man die aktuellste Node.js-Version installieren, wird empfohlen, das Repository um die Datenbank von „wheezy-backport“ zu erweitern. Dies wird über folgende Kommandos ausgeführt:

```
$ echo "deb http://ftp.us.debian.org/debian \
  wheezy-backports main" >> /etc/apt/sources.list
$ apt-get update
$ apt-get install nodejs-legacy
$ curl --insecure https://www.npmjs.org/install.sh | bash
```

1.2.4 Ubuntu

Node.js ist auch für Ubuntu im offiziellen Repository verfügbar. Allerdings sind für ältere Versionen von Ubuntu (12.04 bis 13.04) nur veraltete Versionen von Node.js enthalten. Die Versionsreihe 0.10.x ist erst ab Ubuntu Version 13.10 aufwärts enthalten. Es existieren aber auch für Ubuntu zusätzliche Repositories, in denen man neuere Node.js-Versionen für ältere Ubuntu-Systeme finden kann.

Die Installation erfolgt per Kommandozeile – getrennt für Node.js und den eigentlich unumgänglichen Package-Manager `npm`. Letzterer wird später noch detailliert beschrieben und man sollte gar nicht erst versuchen, ohne ihn Node.js-Software erstellen zu wollen.

```
$ sudo apt-get install nodejs
$ sudo apt-get install npm
```

Um aktuellste Node.js-Versionen zu installieren, wird auch hier empfohlen, das Repository zu erweitern. Hier empfiehlt sich die Datenbank von Chris Lea:

```
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install python-software-properties \
  python g++ make nodejs
```

Für die älteren Ubuntu-Versionen bis 13.04 ist es oft auch erforderlich, das Paket „software-properties-common“ zu installieren:

```
$ sudo apt-get install software-properties-common
```

Falls das Kommando `add-apt-repository` nicht verfügbar ist, müssen zudem noch die „python-software-properties“ installiert werden:

```
$ sudo apt-get install python-software-properties
```

Namenskonflikt „node“

Bei Ubuntu lauert zudem noch ein Namenskonflikt: Neben dem Node.js Executable `node` gibt es auch noch das Projekt „Amateur Packet Radio Node Program“, welches leider auch ein Executable namens `node` mitbringt. Deshalb wurde das Programm von Node.js hier in `nodejs` umbenannt[17]. Um eventuelle Konflikte zu umgehen, kann ein „Symlink“ von `nodejs` auf `node` gemacht werden – dann ist Node.js wie auf jedem anderen System benutzbar:

```
$ ln -s /usr/bin/nodejs node
```

1.2.5 openSUSE und SLE

Node.js für openSUSE lässt sich mit der richtigen URL mittels „openSUSE one-click“ direkt herunterladen. Die URLs gehorchen dabei einem bestimmten Muster, in dem das „project“ und das „package“ des zu ladenden Pakets verschlüsselt sind. Für Node.js müssen hier die Werte „devel:languages:nodejs“ und „nodejs“ eingetragen werden. Die resultierende URL lautet damit:

http://software.opensuse.org/download.html?project=devel%3Alanguages%3Anodejs&package=nodejs

[17] https://lists.debian.org/debian-devel-announce/2012/07/msg00002.html

Für die Versionen openSUSE 11.4, 12.1, 12.2, 12.3, 13.1, Factory und Tumbleweed sowie für SLE 11 (in allen SP1/SP2/SP3-Varianten) sind RPM-Pakete verfügbar.

Hier als Beispiel die Installation unter openSUSE 13.1 mittels des Paketmanagers „zypper“:

```
$ sudo zypper ar http://download.opensuse.org/repositories/\
  devel%3Alanguages%3Anodejs/openSUSE_13.1/ Node.js
$ sudo zypper in nodejs nodejs-devel
```

1.2.6 Fedora

Node.js und NPM sind seit Fedora Version 18 verfügbar und können mit dem favorisierten Package-Manager benutzt werden. Alternativ lassen sich beide über die Kommandozeile installieren:

```
$ sudo yum install nodejs npm
```

1.2.7 RHEL und CentOS

Node.js und *npm* sind hier mittels „Fedora Extra Packages for Enterprise Linux“ (EPEL) verfügbar. Falls noch nicht geschehen, sollte also zuerst *epel* eingebunden werden. Um zu überprüfen, ob dies bereits geschehen ist, kann man das Kommando `sudo yum repolist` benutzen. Sollte *epel* nicht in der Liste der Repositories auftauchen, so muss es zuerst noch installiert werden[18].

Die Installation von Node.js (und *npm*) selbst erfolgt dann mittels folgender Kommandos:

```
$ sudo yum install nodejs --enablerepo=epel
$ sudo yum install npm --enablerepo=epel
```

1.3 IDEs

Um die Quelltexte von Node.js zu editieren, gibt es viele verschiedene Möglichkeiten – letztendlich reicht ja schon ein einfacher Texteditor, um ein JavaScript-Programm zu schreiben. Gerade unter Linux und MacOS sind Editoren wie *vim*[19] oder *MacVim*[20] weit verbreitet und werden gerne genutzt. Da für JavaScript kein Compiler benötigt wird, sind die grundsätzlichen Anforderungen an eine „IDE“ für JavaScript auch nicht sonderlich groß.

[18] Auf der Webseite https://fedoraproject.org/wiki/EPEL#How_can_I_use_these_extra_packages kann für das betreffende System das passende Paket heruntergeladen und mittels `rpm` installiert werden.

[19] http://www.vim.org/

[20] https://code.google.com/p/macvim/

Im Detail heben sich dann doch ein paar Editoren durch spezielle Eigenschaften hervor. In den nächsten Abschnitten wird auf diese Editoren genauer eingegangen und es werden ihre Stärken und Schwächen gezeigt.

Die Aufstellung erhebt natürlich keinen Anspruch auf Vollständigkeit. Auf der einen Seite kann man nicht jeden möglichen Editor eingehend testen, auf der anderen Seite ist das Umfeld um Node.js sehr agil. Es kommt immer wieder vor, dass Tools oder Module innerhalb kürzester Zeit sehr populär werden. Daneben werden die Vorlieben bei Editoren oft von subjektiven Kriterien bestimmt. Daher ist es wirklich sinnvoll, wenn sich der Leser selbst ein Bild von der Arbeitsweise mit einem der Editoren macht.

Wir wollen hier die subjektiven Erfahrungen der Autoren wiedergeben, um dem Leser eine Entscheidungshilfe an die Hand zu geben oder auch einfach nur mit Tipps zur Seite zu stehen.

1.3.1 cloud9

Die größte Stärke von *cloud9*[21] ist auch gleichzeitig die größte Schwäche. Wie der Name schon vermuten lässt, „lebt" *cloud9* in der Cloud. Es ist ein webbasierter Editor, der komplett online im Browser ausgeführt wird. Das bedeutet aber auch, dass man hierfür eine Internetverbindung haben muss. Wer oft auf Reisen ist, könnte hier Probleme bekommen – ein Ausweg aus dieser Situation ist aber weiter unten beschrieben. Die Serverseite von *cloud9* ist übrigens in Node.js geschrieben.

Der Editor ist für ein HTML- und JavaScript-basiertes Programm unglaublich leistungsstark. Die Funktions- und Kombinationstasten (Ctrl-, Alt- und Shift-Kombinationen) sind durchaus vergleichbar mit nativen Editoren. Man muss noch nicht mal auf eine Historie der Änderungen verzichten. Im Wesentlichen sieht es so aus, als würde man durch das Browserfenster einfach nur auf einen anderen Bildschirm blicken.

Wenn man sich bei *cloud9* anmeldet, kann man beliebig viele öffentliche Projekte auf dem Server anlegen und verwalten. Öffentliche Projekte sind jedoch von jedermann auffindbar und einsehbar. Sollen die Projekte privater Natur sein, so ist lediglich ein Projekt kostenlos. Benötigt man mehrere private Projekte, so kann man hierfür ein Premium-Paket buchen.

Startet man ein Projekt, so erhält man einen „Workspace", in dem man entwickeln und testen kann. Hierfür steht auch die dedizierte PaaS-Runtime für Node.js-Anwendungen zur Verfügung. Normalerweise gibt man nun für ein Projekt an, wo sich das zugehörige Quellcode-Repository befindet. Unterstützt werden GitHub und BitBucket. Zudem ist es möglich, „Workspaces" auf einem eigenen Server anzulegen.

Die größte Stärke sind die Collaboration-Fähigkeiten, also das Arbeiten im Team. Sind mehrere Entwickler zur gleichen Zeit im selben Code, sehen sie die Änderungen der anderen Entwickler in Echtzeit. Ein nachträglicher Merge von verschiedenen Quellcode-Versionen erübrigt sich damit.

Wenn man im Team schnell entwickeln will, kann man mittels *cloud9* auch gemeinsam direkt auf einem eigenen Server arbeiten. Dazu wird das Projekt in *cloud9* als „ssh" defi-

[21] https://c9.io/

niert. Die Quellen werden direkt auf diesem Server editiert. Ein *forever*- oder *pm2*-Prozess kann die Änderungen am Quellcode überwachen und die gerade veränderte Anwendung sofort neu starten, sobald ein Entwickler eine Codeänderung abspeichert. Näheres zur Verwendung dieser beiden Tools findet man im Abschnitt 3.3.1, „Ein eigener Server".

Da *cloud9* intern eine History der Änderungen ablegt, kann man jederzeit über einen Zeitstrahl sehen, was und von wem etwas verändert wurde. Ein Aus- bzw. Einchecken von einzelnen Dateien erübrigt sich damit theoretisch, allerdings müsste man sich dann Gedanken machen, wie man anderweitig mit verschiedenen Versionsständen, Feature-Branches oder dem Thema Datensicherheit umgehen möchte.

Und genau das ist auch der größte Nachteil dieser Vorgehensweise. Programmiert man in einem größeren Team, will man seine eigenen Änderungen sehen und testen – nach Möglichkeit isoliert und ohne andere bei der Arbeit zu stören. Speziell, wenn man an verschiedenen Problemstellungen parallel arbeitet, kann man sich in einer gemeinsamen Codebasis auch schnell gegenseitig in die Quere kommen.

Das Deployment einer in *cloud9* bearbeiteten Anwendung in PaaS-Umgebungen wie Windows Azure, Heroku und RedHat Openshift ist ebenfalls direkt möglich.

Wer mit den erwähnten Nachteilen leben kann, erhält also einen außerordentlich guten und leistungsfähigen Editor. Wer jedoch nicht immer auf eine Internetverbindung zurückgreifen kann, für den ist eventuell eine lokale Installation von *cloud9* ein Ausweg.

Lokale Installation von cloud9

Der Editor von *cloud9* ist ein Open-Source-Projekt. Somit kann der Code einfach geladen und der Editor lokal installiert werden. Natürlich muss man in diesem Fall auf das eine oder andere Feature verzichten – dass dann beispielsweise Collaboration bei einer lokalen Installation nicht funktioniert, sollte nicht weiter überraschen. Für Situationen, in denen man die Online-Variante von *cloud9* nicht nutzen kann, scheint das aber ein gangbarer Weg zu sein.

Zur Installation werden *git* und eine Node.js-Installation inklusive *npm* benötigt.

Zur Installation wird der Code auf den lokalen Rechner kopiert (git clone) und die Anwendung anschließend vorbereitet und gestartet:

```
$ git clone https://github.com/ajaxorg/cloud9.git
$ cd cloud9
$ npm install
$ bin/cloud9.sh -w <dir>
```

Wobei `<dir>` dem Projektverzeichnis entspricht.

Zum Editieren selbst wird im Browser die URL *http://localhost:3131* aufgerufen. Man hat damit nahezu dieselben Möglichkeiten wie in der öffentlichen Variante über *http://c9.io*.