Aktuell ab Version 24

Eventloop, Buffer, TypeS
Module, Expre
async, npm,
Streams, Path
WebSockets

```
import { open } from 'node:fs/promises';

async function printFileContents(path) {
  const fileHandle = await open(path);
  const stream = fileHandle.readableWebStream()
    .pipeThrough(new TextDecoderStream());

  for await (const chunk of stream) {
    process.stdout.write(chunk);
  }

  await fileHandle.close();
}

await printFile
```

Sebastian Springer

Node.js

Das umfassende Handbuch

- Professionelle Backends entwickeln
- Installation, Grundlagen, Best Practices
- Debugging, Skalierung, Qualitätssicherung

5., aktualisierte und erweiterte Auflage

Mit allen Codebeispielen zum Download

Rheinwerk Computing

Kapitel 1
Grundlagen

Aller Anfang ist schwer.
– Ovid

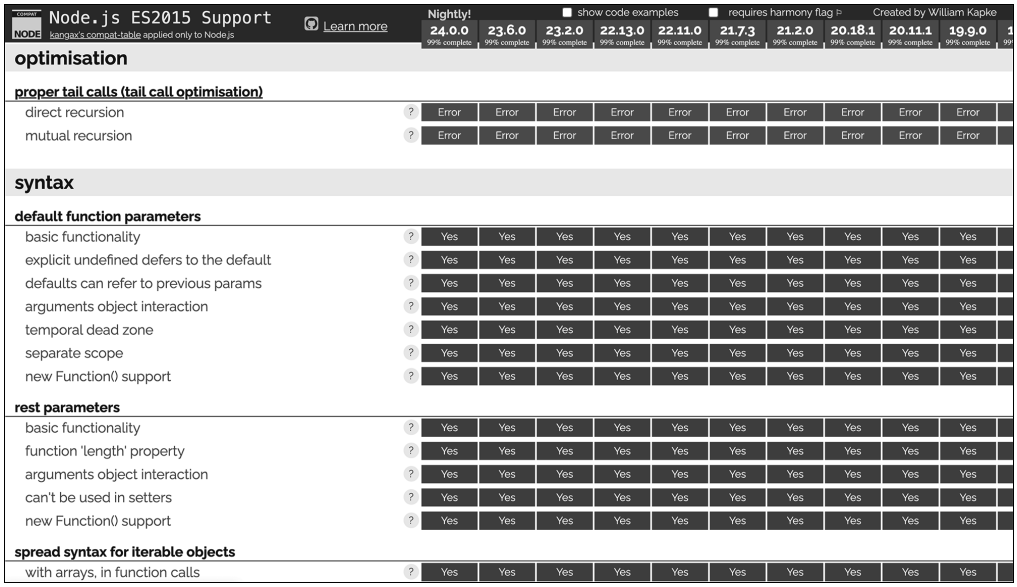
Mehr Dynamik in Webseiten zu bringen, das war die ursprüngliche Idee hinter JavaScript. Die Skriptsprache sollte die Schwachstellen von HTML ausgleichen, wenn es darum ging, auf Benutzereingaben zu reagieren. Die Geschichte von JavaScript geht zurück auf das Jahr 1995, da es unter dem Codenamen Mocha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

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

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

Im Jahr 1997 entstand der erste Entwurf des Sprachstandards bei der ECMA International. Unter der kryptischen Bezeichnung ECMA-262 beziehungsweise ISO/IEC 16262 ist der gesamte Sprachkern der Skriptsprache festgehalten. Den aktuellen Standard finden Sie unter *www.ecma-international.org/publications/standards/Ecma-262.htm*. Herstellerunabhängig wird JavaScript aufgrund dieser Standardisierung auch als ECMAScript bezeichnet. Bis vor einigen Jahren wurde der ECMAScript-Standard in Ganzzahlen beginnend bei 1 versioniert. Seit Version 6 werden die Versionen außerdem mit Jahreszahlen versehen. ECMAScript in Version 16 wird daher als ECMAScript 2025 bezeichnet. In der Regel können Sie davon ausgehen, dass die Hersteller die älteren Versionen des Standards gut unterstützen. Neuere Features müssen Sie entweder durch Konfigurationsflags im Browser freischalten oder durch Polyfills, also den

Nachbau der Features in JavaScript, simulieren. Eine gute Übersicht der aktuell unterstützten Features bietet die Compat-Table von kangax, die Sie unter *https://compat-table.github.io/compat-table/es2016plus/* finden. Eine für Node.js angepasste Version erreichen Sie unter *http://node.green/*.



Node.js ES2015 Support — kangax's compat-table applied only to Node.js — Learn more — Nightly! — show code examples — requires harmony flag — Created by William Kapke

	24.0.0 99% complete	23.6.0 99% complete	23.2.0 99% complete	22.13.0 99% complete	22.11.0 99% complete	21.7.3 99% complete	21.2.0 99% complete	20.18.1 99% complete	20.11.1 99% complete	19.9.0 99% complete
optimisation										
proper tail calls (tail call optimisation)										
direct recursion	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error
mutual recursion	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error
syntax										
default function parameters										
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
explicit undefined defers to the default	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
defaults can refer to previous params	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
temporal dead zone	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
separate scope	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
rest parameters										
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
function 'length' property	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
can't be used in setters	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
spread syntax for iterable objects										
with arrays, in function calls	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Abbildung 1.1 Unterstützung von JavaScript-Features in Node.js (http://node.green)

JavaScript ist leichtgewichtig, relativ einfach zu erlernen und verfügt über ein riesiges Ökosystem von Frameworks und Bibliotheken. Aus diesen Gründen ist JavaScript eine der erfolgreichsten Programmiersprachen der Welt. Dieser Erfolg lässt sich sogar durch Zahlen belegen: Seit 2008 ist JavaScript bei den Sprachtrends von GitHub immer auf den beiden vorderen Plätzen zu finden. Bis 2022 belegte JavaScript sogar jahrelang in Folge den ersten Platz. Erst 2024 wurde es von Python als Nummer 1 abgelöst. Das liegt vor allem an der Popularität von Python für KI-Applikationen. Zur Beruhigung für alle JavaScript-Entwickler: GitHub hat sich dafür entschieden, JavaScript und TypeScript als zwei getrennte Sprachen zu betrachten. JavaScript belegt in der Statistik den zweiten und TypeScript den dritten Platz. Nimmt man beide zusammen, liegen sie deutlich vor Python.

Node.js basiert auf dieser erfolgreichen Skriptsprache und arbeitet aktuell an einer nativen Unterstützung für TypeScript. Node.js hat als Plattform selbst einen kometenhaften Aufstieg hingelegt. Dieses Kapitel soll Ihnen als Einführung in die Welt von Node.js dienen und Ihnen zeigen, wie die Plattform aufgebaut ist und wo Sie Node.js überall einsetzen können.

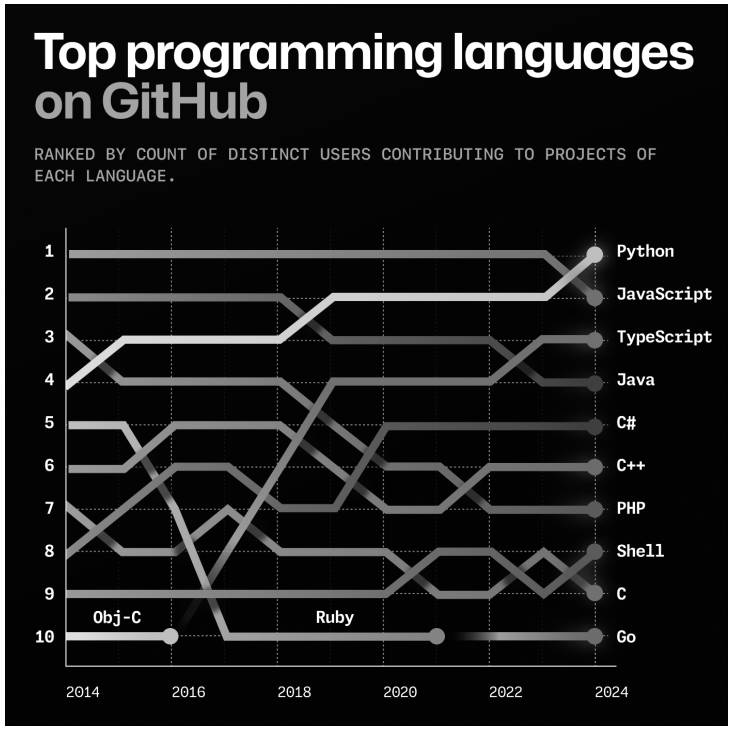


Abbildung 1.2 Topsprachen in GitHub nach Pull Requests (octoverse.github.com)

1.1 Die Geschichte von Node.js

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

1.1.1 Die Ursprünge

Die Geschichte von Node.js beginnt mit Ryan Dahl, einem Doktoranden der Mathematik, der sich eines Besseren besann, seine wissenschaftlichen Bemühungen abbrach und stattdessen lieber mit einem One-Way-Ticket und nur wenig Geld in der Tasche nach Südamerika reiste, wo er sich mit Englischunterricht über Wasser hielt. In dieser Zeit kam er mit den Programmiersprachen PHP und Ruby in Berührung und entdeckte darüber seine Liebe zur Webentwicklung. Das Problem bei der Arbeit mit dem Ruby-Framework Rails war, dass es nicht ohne Workarounds möglich war, mit konkurrierenden Anfragen umzugehen. Die Applikationen waren für seinen Ge-

schmack zu langsam und lasteten die CPU vollständig aus. Eine Lösung für seine Probleme fand Ryan Dahl in Mongrel. Dabei handelt es sich um einen Webserver für Applikationen, die auf Ruby basieren.

Im Gegensatz zu klassischen Webservern reagiert Mongrel auf Anfragen von Nutzern und generiert die Antworten dynamisch, wo sonst lediglich statische HTML-Seiten ausgeliefert werden.

Die Aufgabe, die eigentlich zur Entstehung von Node.js führte, ist vom heutigen Standpunkt aus betrachtet recht trivial. Im Jahr 2005 suchte Ryan Dahl nach einer eleganten Möglichkeit, einen Fortschrittsbalken für Dateiuploads zu implementieren. Mit den damals verfügbaren Technologien waren nur unbefriedigende Lösungen möglich. Zur Übertragung der Dateien wurde für kleinere Dateien das HTTP- 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. Einen ersten Versuch zur Umsetzung eines Fortschrittsbalkens unternahm Ryan Dahl mit Mongrel. Nach dem Absenden der Datei an den Server prüfte er mithilfe einer Vielzahl von asynchronen Anfragen den Status des Uploads und stellte diesen in einem Fortschrittsbalken grafisch dar. Störend an dieser Umsetzung waren allerdings der damalige Single-threaded-Ansatz von Ruby und die große Anzahl an Anfragen, die benötigt wurden.

Einen weiteren vielversprechenden Ansatz bot eine Umsetzung in C. Hier war Ryan Dahl nicht auf einen Thread begrenzt. C als Programmiersprache für das Web hat allerdings einen entscheidenden Nachteil: Es lassen sich recht wenige Entwickler für dieses Einsatzgebiet begeistern. Mit diesem Problem sah sich auch Ryan Dahl konfrontiert und verwarf diesen Ansatz nach kurzer Zeit ebenfalls wieder.

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

1.1.2 Die Geburt von Node.js

Die Lösung, mit der er schließlich zufrieden war, fand Ryan Dahl dann im Januar 2009 mit JavaScript. Hier wurde ihm klar, dass diese Skriptsprache sämtliche seiner Anfor-

derungen 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, die Sprache sah keine Schnittstellen zur Arbeit mit Dateien in einem Dateisystem vor, und es gab keine Implementierung von Sockets zur Kommunikation mit anderen Anwendungen oder Systemen. All diese Punkte sprachen für JavaScript als Grundlage für eine Plattform für interaktive Webapplikationen, da es noch keine Vorgaben in diesem Bereich gab und demzufolge auch noch keine Fehler begangen worden waren (ganz fehlerfrei war jedoch auch der Ansatz von Ryan Dahl nicht, wie sich im Laufe der Zeit herausstellte). Auch die Architektur von JavaScript sprach für den Einsatz in einer serverseitigen Plattform. Der Ansatz der Top-Level-Functions, also der Funktionen, die mit keinem Objekt verknüpft und daher frei verfügbar sind und zudem Variablen zugeordnet werden können, bietet eine hohe Flexibilität in der Entwicklung und ermöglicht funktionale Lösungsansätze.

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.

Ebenfalls im Jahr 2009, genauer gesagt im September 2009, begann Isaac Schlueter seine Arbeit an einem Paketmanager für Node.js, dem *Node Package Manager*, auch bekannt als *NPM*, einem weiteren Grund für den Erfolg von Node.js und dem gesamten JavaScript-Ökosystem.

1.1.3 Der Durchbruch von Node.js

Nachdem Ryan Dahl sämtliche Komponenten integriert hatte und erste lauffähige Beispielapplikationen auf Basis von Node.js erstellt hatte, benötigte er eine Möglichkeit, Node.js der Öffentlichkeit vorzustellen. Dies wurde auch nötig, da seine finanziellen Mittel durch die Entwicklung an Node.js beträchtlich schrumpften und er, falls er keine Sponsoren finden sollte, die Arbeit an Node.js hätte einstellen müssen. Als Präsentationsplattform wählte er die JavaScript-Konferenz jsconf.eu im November 2009 in Berlin. Ryan Dahl setzte alles auf eine Karte. Eine erfolgreiche Präsentation würde ihm Sponsoren einbringen, die seine Arbeit an Node.js unterstützten. Dann könnte er sein Engagement fortsetzen; falls er keine Sponsoren fände, wäre die Arbeit von fast einem Jahr umsonst gewesen. In einem mitreißenden Vortrag stellte er Node.js dem Publikum vor und zeigte, wie man mit nur wenigen Zeilen JavaScript-Code einen voll funktionsfähigen Webserver erstellen kann. Als weiteres Beispiel

brachte er eine Implementierung eines IRC-Chatservers mit. Der Quellcode dieser Demonstration umfasste etwa 400 Zeilen. Anhand dieser Beispiele demonstrierte er die Architektur der Plattform und damit die Stärken von Node.js und machte es gleichzeitig für die Zuschauer greifbar. Die Aufzeichnung dieses Vortrags finden Sie unter *www.youtube.com/watch?v=EeYvFl7li9E*. Die Präsentation verfehlte ihr Ziel nicht und führte dazu, dass das Unternehmen Joyent als Sponsor für Node.js einstieg. 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 in Vollzeit als Maintainer für Node.js eingesetzt.

1.1.4 Node.js erobert Windows

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

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

Überraschend war Anfang 2012 die Ankündigung Ryan Dahls, sich nach drei Jahren der Arbeit an Node.js schließlich aus der aktiven Weiterentwicklung zurückzuziehen. Er übergab die Leitung der Entwicklung an Isaac Schlueter. Dieser war, wie auch Ryan Dahl, Angestellter bei Joyent und aktiv an der Entwicklung des Kerns von Node.js beteiligt. Der Wechsel verunsicherte die Community, da nicht klar war, ob die Plattform auch ohne Ryan Dahl weiterentwickelt würde. Ein Signal, dass die Node.js-Community stark genug für eine solide Weiterentwicklung war, gab die Veröffentlichung der Version 0.8 im Juni 2012, die vor allem die Performance und Stabilität von Node.js entscheidend verbessern sollte.

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

1.1.5 io.js – der Fork von Node.js

Im Januar 2014 gab es erneut eine Änderung in der Projektleitung von Node.js. Auf Isaac Schlüter, der die Maintenance von Node.js zugunsten seines eigenen Unterneh-

mens npmjs, des Hosters der zentralen NPM-Registry, aufgab, folgte TJ Fontaine. Ein weitverbreiteter Kritikpunkt an Node.js war zu diesem Zeitpunkt, dass das Framework immer noch nicht die vermeintlich stabile Version 1.0 erreicht hatte, was zahlreiche Unternehmen davon abhielt, Node.js für kritische Applikationen einzusetzen.

Viele Entwickler waren unzufrieden mit Joyent, das seit Ryan Dahl die Maintainer für Node.js stellte, und so kam es im Dezember 2014 zum Bruch in der Community. Das Resultat war das Projekt io.js, ein Fork von Node.js, der getrennt von der ursprünglichen Plattform weiterentwickelt wurde. Daraufhin wurde im Februar 2015 die unabhängige Node.js Foundation gegründet, die für die Weiterentwicklung von io.js zuständig war. Nahezu zeitgleich erschien die Version 0.12 des Node.js-Projekts.

1.1.6 Node.js wieder vereint

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

1.1.7 Deno – 10 Things I Regret about Node.js

Seit der Zusammenführung von io.js und Node.js ist es ruhiger um Node.js geworden. Die geplanten Releases, die Stabilität, aber auch die Integration neuer Features, wie beispielsweise Worker-Threads, HTTP/2 oder Performance-Hooks, sorgen für gute Stimmung in der Community. Und gerade als es begann, schon fast zu ruhig um Node.js zu werden, trat 2018 mit Ryan Dahl ein alter Bekannter auf die Bühne und stellte im Zuge seines Vortrags »10 Things I Regret about Node.js« eine neue serverseitige JavaScript-Plattform mit dem Namen Deno vor.

Die Idee hinter Deno besteht darin, ein besseres Node.js zu erschaffen, und zwar gelöst von den Zwängen der Rückwärtskompatibilität, die revolutionäre Sprünge in der Entwicklung verhindern. So setzt Deno beispielsweise standardmäßig auf TypeScript und fügt ein grundlegend anderes Modulsystem ein. Auch der Kern von Deno unterscheidet sich deutlich von dem von Node.js, da er nahezu vollständig in Rust geschrieben ist. Das Ziel von Ryan Dahl war, mit Deno die Fehler, die sich bei Node.js im Laufe der Zeit eingeschlichen hatten, zu beheben.

Dennoch gibt es eine Reihe von Gemeinsamkeiten. So setzt Deno auf die altbewährte V8-Engine, die auch das Herz von Node.js bildet. Und auf die riesige Anzahl von NPM-Paketen müssen Sie ebenfalls nicht verzichten. Deno bietet hierfür eine Kompatibilitätsschicht. Mehr zum Thema Deno erfahren Sie in Kapitel 25, »JavaScript Runtimes«.

1.1.8 Die OpenJS Foundation

Im Jahr 2015 wurde die Node.js Foundation gegründet, um die Entwicklung der Plattform zu koordinieren. Die Foundation ist ein der Linux Foundation untergeordnetes Projekt. Im Jahr 2019 haben sich die JS Foundation und die Node.js Foundation dann zur OpenJS Foundation zusammengeschlossen. Sie umfasst neben Node.js eine Reihe weiterer populärer Projekte wie beispielsweise webpack, ESLint oder Electron.

Die wichtigsten Aufgaben der OpenJS Foundation sind:

- Verwaltung und Förderung von JavaScript-Projekten
- Sicherstellen von Stabilität und langfristiger Unterstützung
- Fördern offener Standards und Interoperabilität
- Bereitstellen von Ressourcen für Entwickler und Unternehmen

1.1.9 Bun – die nächste Generation der JavaScript-Runtimes

Im Jahr 2021 begann Jarred Sumner mit der Arbeit an einem Konkurrenzprojekt zu Node.js und Deno. Das Projekt trägt den Namen Bun und ist ebenfalls eine Plattform für serverseitiges JavaScript. Im Gegensatz zu Node.js, das im Kern hauptsächlich auf C und C++ setzt, und Deno, das Rust als Programmiersprache verwendet, nutzt Bun die Programmiersprache Zig, die hauptsächlich in der systemnahen Programmierung eingesetzt wird. Auch bei der JavaScript-Engine gibt es einen bedeutenden Unterschied. Node.js und Deno nutzen hier die V8-Engine von Google. Bun nutzt stattdessen JavaScriptCore, die JavaScript-Engine von Apple, bekannt aus dem Safari-Browser.

Bun legt seinen Fokus auf Performance und integrierte Werkzeuge. So ist der Paketmanager fester Bestandteil der Plattform und nicht austauschbar wie beispielsweise bei Node.js. Bei allen Unterschieden zu Node.js strebt Bun jedoch eine möglichst hohe Kompatibilität mit Node.js an, sodass Entwickler ihre Applikationen mit nur minimalen Änderungen von Node.js auf Bun umziehen können. Weitere Informationen zu Bun finden Sie in Kapitel 25, »JavaScript Runtimes«.

1.2 Die Organisation von Node.js

Die Community hinter Node.js hat aus der Vergangenheit gelernt. Immer wieder gab es große Verunsicherung, als es einen Wechsel des Maintainers der Plattform gab. Aus diesem Grund gibt es an der Spitze von Node.js keine Einzelpersonen mehr, sondern

ein Komitee aus mehreren Personen, die die Entwicklung der Plattform steuern. Die Steuerung und Entwicklung ist auch ganz bewusst an eine Foundation ausgelagert und hängt nicht von einem großen Unternehmen ab, das mit der Plattform Eigeninteressen verfolgt.

Die einzelnen Steuerungselemente von Node.js sind:

- **Technical Steering Committee**: Es ist für die Weiterentwicklung der Plattform zuständig und umfasst normalerweise sechs bis zwölf Mitglieder. Es macht Vorgaben für die technische Entwicklung, steuert das Projekt, definiert Richtlinien und ist für das Repository verantwortlich. Weitere Informationen finden Sie unter *https://github.com/nodejs/TSC*.
- **Collaborators**: Eine Gruppe von Personen, die schreibend auf das GitHub-Repository von Node.js zugreifen können.
- **Arbeitsgruppen**: Es gibt verschiedene Arbeitsgruppen für bestimmte Themen wie beispielsweise:
  - **Release**: Verwaltet den Releaseprozess von Node.js, definiert die Inhalte und kümmert sich um den Long Term Support.
  - **Streams**: Arbeitet an der Verbesserung der Stream-API der Plattform.
  - **Docker**: Verwaltet die offiziellen Docker-Images von Node.js und sorgt dafür, dass diese aktuell gehalten werden.
- **OpenJS Foundation**: Bildet das Dach der Entwicklung von Node.js. Sie wurde als unabhängiges Gremium zur Weiterentwicklung von Node.js gegründet und finanziert sich aus Spenden und Beiträgen von Unternehmen und individuellen Mitgliedern.

1.3 Versionierung von Node.js

Einer der größten Kritikpunkte an Node.js vor dem Fork von io.js war, dass die Entwicklung sehr langsam voranschritt. Gerade im Unternehmenseinsatz sind regelmäßige und planbare Releases ein wichtiges Auswahlkriterium. Aus diesem Grund vereinbarten die Entwickler von Node.js nach der Zusammenführung von Node.js und io.js einen transparenten Releaseplan mit regelmäßigen Releases und einer Long-Term-Support-Version (LTS), die über einen längeren Zeitraum mit Updates versorgt wird. Der Releaseplan sieht ein Major Release pro Halbjahr vor.

Tabelle 1.1 zeigt den Releaseplan von Node.js.

Release	Status	Code-name	Initial Release	Active LTS Start	Maintenance Start	End-of-life
18.x	Maintenance	Hydrogen	19.04.22	25.10.22	18.10.23	30.04.25
20.x	Maintenance	Iron	18.04.23	24.10.23	22.10.24	30.04.26
22.x	LTS	Jod	24.04.24	29.10.24	21.10.25	30.04.27
23.x	Current		15.10.24	–	01.04.25	01.06.25
24.x	Pending		22.04.25	28.10.25	20.10.26	30.04.28

Tabelle 1.1 Node.js-Releaseplan (https://github.com/nodejs/release#release-schedule)

Wie Sie dem Releaseplan entnehmen können, handelt es sich bei Versionen mit einer geraden Versionsnummer um LTS-Releases und bei ungeraden um Releases mit einer verkürzten Supportzeit. In der grafischen Repräsentation des Releaseplans sehen Sie auch die Überlappung der verschiedenen Versionen.

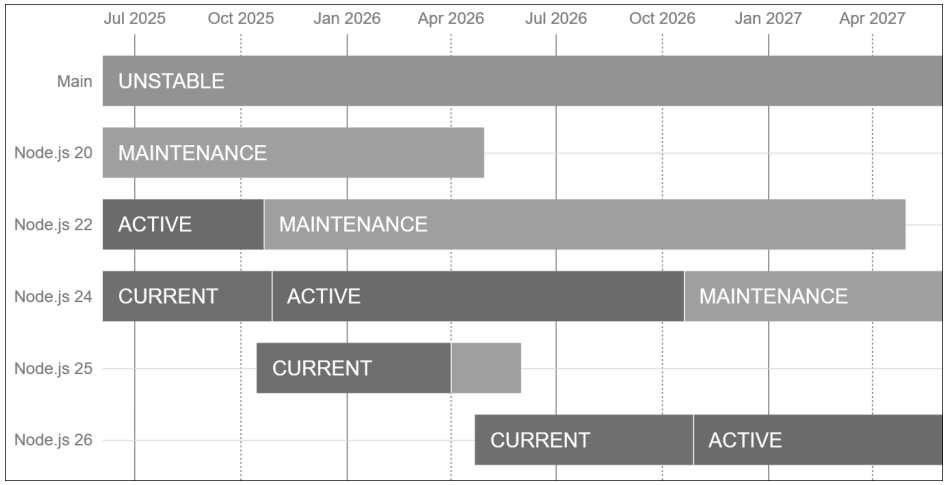


Abbildung 1.3 Grafische Darstellung des Node.js-Releaseplans (https://github.com/nodejs/release#release-schedule)

Die Releases von Node.js bewegen sich in unterschiedlichen Zuständen:

- **Unstable**: Die Main-Entwicklungslinie ist der aktuelle Entwicklungsstand, hier finden Sie immer die neuesten Features. Allerdings ist diese Version nicht für den produktiven Einsatz gedacht.
- **Current**: Die aktuell verfügbare Version von Node.js wird als »Current« bezeichnet. Bis auf seltene Überlappungen gibt es immer nur eine aktuelle Version. Diese wechselt zwischen geraden und ungeraden Versionsnummern hin und her. Egal,

ob es sich bei einer Version um eine LTS-Version oder um ein gewöhnliches Release handelt, die Version verbleibt für ein halbes Jahr in der »Current«-Phase.

- **Active**: Der Active-Zustand ist den LTS-Versionen vorbehalten. Eine Version befindet sich normalerweise für 12 Monate in diesem Zustand und erhält während dieser Zeit:
  - Bugfixes
  - Sicherheitsupdates
  - Aktualisierungen von NPM, die keine Major-Updates sind
  - Aktualisierungen der Dokumentation
  - Performanceverbesserungen, die keine existierenden Applikationen gefährden
  - Veränderungen am Quellcode, die die Integration zukünftiger Verbesserungen vereinfachen
- **Maintenance**: Nach der Active-Phase folgt eine weitere, 18-monatige Maintenance-Phase, in der die Version noch mit Sicherheitsupdates versorgt wird, die kritische Fehler und Sicherheitslücken beheben.

LTS-Releases

Node.js stellt LTS-Releases zur Verfügung, damit die Entwickler von Applikationen mehr Zeit für die Aktualisierung auf neuere Versionen bekommen. Ein LTS-Release sichert Ihnen Stabilität für einen Zeitraum von 36 Monaten zu. Innerhalb dieser Zeit sollten Sie sich jedoch darum kümmern, auf das nachfolgende LTS-Release zu aktualisieren.

Das Node.js-Team empfiehlt, dass produktive Applikationen stets auf LTS-Releases setzen sollten.

1.4 Stärken von Node.js

Node.js bietet Ihnen als eine etablierte Plattform im Web eine ganze Reihe von Vorteilen für einen Einsatz selbst in umfangreichen Applikationen:

- **Leichtgewichtige Plattform**: Der Kern von Node.js ist verhältnismäßig klein, und so benötigt die Plattform nur wenig Ressourcen. Das betrifft sowohl den benötigten Speicherplatz auf der Festplatte als auch den Arbeitsspeicher.
- **Gute Verfügbarkeit**: Node.js ist auf allen verbreiteten Systemen verfügbar. Das gilt sowohl für Desktop- und Serversysteme mit Windows, Linux und macOS als auch für Cloud-Instanzen. Alle großen Infrastrukturanbieter unterstützen Node.js und ermöglichen es Ihnen damit, skalierbare Applikationen mit geringem Aufwand zu veröffentlichen.

- **Standardkompatibel**: Node.js nutzt modernes JavaScript als Basis und orientiert sich auch sonst an den Webstandards, wie die `fetch`-API oder auch die Web Crypto und die Web Streams API zeigen.
- **Weite Verbreitung**: Sowohl JavaScript als Programmiersprache als auch Node.js sind weltweit verbreitet, sodass eine große Zahl potenzieller Entwickler verfügbar ist.
- **Performant**: Node.js basiert im Kern auf der V8-Engine. Diese kommt auch in Chrome und weiteren Browsern zum Einsatz. Google hat viele Jahre Entwicklungszeit in diese Engine investiert und die Ausführung von JavaScript über diese Zeit zu einem hohen Grad optimiert. Da eine Node.js-Applikation üblicherweise über eine längere Zeit im Speicher verbleibt, greifen einige Features der Engine, die für langlaufende Applikationen gedacht sind.
- **Open Source**: Node.js ist durch und durch ein Open-Source-Projekt, an dem sich die Community sehr rege beteiligt. Das gilt sowohl für die Beiträge in Form von Tickets als auch für die Entwicklung neuer Features und das Beisteuern von Bugfixes. Neben dem Kernprojekt verfügt Node.js über eine aktive Community, die den Funktionsumfang der Plattform durch zusätzliche Pakete erweitert.

Mit dieser Charakteristik lässt sich Node.js in einer breiten Palette von Anwendungsfällen einsetzen.

1.5 Einsatzgebiete von Node.js

Node.js ist besonders für Anwendungen geeignet, die eine hohe Performance, Skalierbarkeit oder Echtzeitkommunikation erfordern. Außerdem eignet sich die Plattform sehr gut für Webapplikationen, da Sie die wichtigsten Protokolle, Technologien und Datenstrukturen nativ verwenden können. Typische Anwendungsfälle für Node.js sind:

- **Web-APIs und Backend-Services**: Seinen Ursprung hat Node.js als Backend von Webapplikationen. Hier setzen Sie es typischerweise für die Implementierung von RESTful APIs oder GraphQL-Schnittstellen ein.
- **Echtzeitanwendungen**: Mit Node.js können Sie neben der klassischen Kommunikation über das HTTP-Protokoll auch bidirektionale Kommunikation über das WebSocket-Protokoll umsetzen und damit Chat- oder ähnliche echtzeitbasierte Applikationen umsetzen.
- **Streaming-Anwendungen**: Node.js ist in der Lage, Audio, Video und Daten in einem kontinuierlichen Strom zu verarbeiten und zu senden, was die Plattform zu einer guten Basis für verschiedene Streaming-Applikationen macht.

- **Serverless- und Cloud-basierte Anwendungen**: Durch seine leichtgewichtige, nicht blockierende Architektur und Unterstützung bei nahezu allen Providern eignet sich Node.js als Basis für verschiedenste Cloud-Applikationen und Microservices. Auch im Bereich des Edge Computings ist Node.js weit verbreitet.
- **DevOps und CLI-Werkzeuge**: Node.js ist häufig als Backend für Webapplikationen anzutreffen, eignet sich jedoch auch hervorragend für Kommandozeilenwerkzeuge im Webbereich, zur Automatisierung von Prozessen und als Unterstützung von Deployment-Prozessen.
- **Internet-of-Things-Anwendungen (IoT)**: Im IoT-Umfeld können Sie Node.js nutzen, um die Daten von Sensoren und Smart Devices zu verarbeiten und mit den Geräten über Protokolle wie MQTT zu kommunizieren.
- **Unternehmensanwendungen und E-Commerce**: In großen Applikationen im Unternehmens- und E-Commerce-Bereich können Sie Node.js als Microservice nutzen, Systeme integrieren und Daten verarbeiten.
- **Cross-Platform-Anwendungen**: Sie können Node.js als Backend-Schicht in Webapplikationen, als Basis für Kommandozeilenwerkzeuge, aber auch im Zuge von Desktop-Applikationen beispielsweise mit Electron verwenden.

Sie können Node.js in vielen modernen Anwendungsfällen und auch im Verbund mit anderen Plattformen, Technologien und Programmiersprachen einsetzen.

1.6 Die wichtigsten Merkmale von Node.js

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

Nun aber zu den wichtigsten Rahmendaten:

- **Reines JavaScript**: Bei der Arbeit mit Node.js müssen Sie keinen neuen Sprachdialekt lernen, sondern können auf den Sprachkern von JavaScript zurückgreifen. Für den Zugriff auf Systemressourcen stehen Ihnen standardisierte und gut dokumentierte Schnittstellen zur Verfügung. Alternativ zu JavaScript können Sie Ihre Node.js-Applikation jedoch auch in TypeScript verfassen, den Quellcode in JavaScript übersetzen und mit Node.js ausführen lassen. Mehr zu diesem Thema erfahren Sie in Kapitel 13, »Typsichere Applikationen in Node.js«.
- **Optimierte Engine**: Node.js baut auf der JavaScript-Engine V8 von Google auf. Sie profitieren hier vor allem von der stetigen Weiterentwicklung der Engine, bei der nach kürzester Zeit die neuesten Sprachfeatures unterstützt werden.

- **Nonblocking I/O**: Sämtliche Operationen, die nicht direkt in Node.js stattfinden, blockieren die Ausführung Ihrer Applikation nicht. Der Grundsatz von Node.js lautet: Alles, was die Plattform nicht direkt erledigen muss, wird an das Betriebssystem, andere Applikationen oder Systeme ausgelagert. Die Applikation erhält damit die Möglichkeit, auf weitere Anfragen zu reagieren oder Aufgaben parallel abzuarbeiten. Ist die Bearbeitung der Aufgabe erledigt, erhält der Node.js-Prozess eine Rückmeldung und kann die Informationen weiterverarbeiten.
- **Single-threaded**: Eine typische Node.js-Applikation läuft in einem einzigen Prozess ab. Es gab lange Zeit kein Multi-Threading, und Nebenläufigkeit war zunächst nur in Form des bereits beschriebenen Nonblocking I/O vorgesehen. Sämtlicher Code, den Sie selbst schreiben, blockiert also potenziell Ihre Applikation. Sie sollten daher auf eine ressourcenschonende Entwicklung achten. Falls es dennoch erforderlich wird, Aufgaben parallel abzuarbeiten, bietet Ihnen Node.js hierfür Lösungen in Form des `child_process`-Moduls, mit dem Sie eigene Kindprozesse erzeugen können.

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

1.7 Das Herzstück – die V8-Engine

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

Der JavaScript-Quellcode wird vor der Ausführung nicht kompiliert, stattdessen werden die Dateien mit dem Quellcode 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, sodass die angepassten Quellcodedateien erneut eingelesen werden.

1.7.1 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, was 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 Datenwörtern. 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 dahin gehend optimiert, dass ein sehr schneller Zugriff auf die Elemente im Speicher erfolgen kann und hier wenig Wartezeiten durch das Suchen von Objekten entstehen.

1.7.2 Zugriff auf Eigenschaften

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 auch in anderen Sprachen 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 Konstruktorfunktionen oder die Methode `Object.create` nutzen. Diese Methoden haben gemein, dass Sie ein Objekt erstellen und den Prototyp zuweisen. Der Prototyp ist ein Objekt, von dem ein anderes Objekt seine Eigenschaften erbt. Ein weiteres Merkmal von Prototypen ist, dass sie zur Laufzeit der Applikation modifiziert werden können und Sie so neue Eigenschaften und Methoden hinzufügen können. Durch die Verwendung von Prototypen können Sie in JavaScript eine Vererbungshierarchie aufbauen.

Im Normalfall geschieht der Zugriff auf Eigenschaften in einer JavaScript-Engine über ein Verzeichnis im Arbeitsspeicher. Greifen Sie also auf eine Eigenschaft zu, wird in diesem Verzeichnis nach der Speicherstelle der jeweiligen Eigenschaft gesucht, danach kann dann auf den Wert zugegriffen werden. Stellen Sie sich nun eine große Applikation vor, die auf der Clientseite ihre Geschäftslogik in JavaScript abbildet und in der parallel eine Vielzahl von Objekten im Speicher gehalten werden, die ständig miteinander kommunizieren – diese Art des Zugriffs auf Eigenschaften wird 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, in der 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.

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}
const johnDoe = new Person('John', 'Doe');
```

Listing 1.1 Zugriff auf Eigenschaften in einer Klasse

Im Beispiel erstellen Sie eine neue Konstruktorfunktion für die Gruppe der `Person`-Objekte. Dieser Konstruktor besitzt zwei Parameter, den Vor- und den 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. Erfolgt die erste Zuweisung, also das Setzen des Vornamens, 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 einfügt, dass Class 2 verwendet werden soll, wenn die Eigenschaft `lastname` verwendet wird. Werden Eigenschaften abseits des Konstruktors hinzugefügt und erfolgt dies in unterschiedlicher Reihenfolge, werden jeweils neue Hidden Classes erzeugt. Abbildung 1.4 verdeutlicht diesen Zusammenhang.

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.

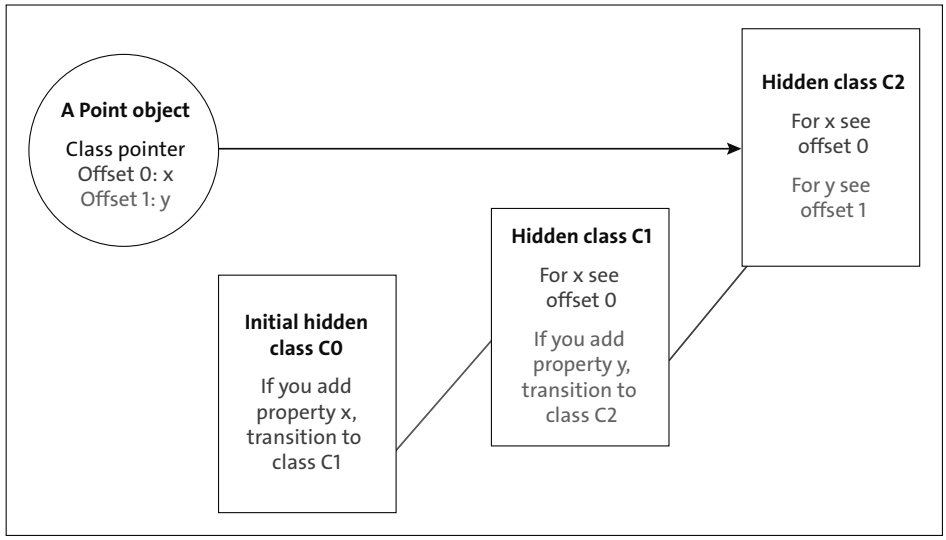


Abbildung 1.4 Hidden Classes in der V8-Engine (https://v8.dev/blog/fast-properties)

1.7.3 Maschinencodegenerierung

Wie Sie bereits wissen, interpretiert die V8-Engine den Quellcode der JavaScript-Applikation nicht direkt, sondern führt eine Just-in-time-Kompilierung (JIT) 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 eins zu eins umgewandelt. Die V8-Engine besitzt neben diesem Just-in-time-Compiler einen weiteren Compiler, der in der Lage ist, den Maschinencode zu optimieren. Zur Entscheidung, welche Codefragmente zu optimieren sind, führt die Engine eine interne Statistik über die Anzahl der Funktionsaufrufe und darüber, wie lange die jeweilige Funktion ausgeführt wird. Aufgrund dieser Daten wird die Entscheidung getroffen, ob der Maschinencode einer Funktion optimiert werden muss oder nicht.

Nun stellen Sie sich bestimmt die Frage, warum denn nicht der gesamte Quellcode der Applikation mit dem zweiten, viel besseren Compiler kompiliert wird. Das hat einen ganz einfachen Grund: Der Compiler, der keine Optimierungen vornimmt, ist wesentlich schneller. Da die Kompilierung des Quellcodes just in time stattfindet, ist

dieser Vorgang sehr zeitkritisch, weil sich eventuelle Wartezeiten durch einen zu lange dauernden Kompilierungsvorgang direkt auf den Nutzer auswirken können. Daher 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 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 hätten eine Codestelle, an der im Wechsel immer zwei verschiedene Objekte mit unterschiedlichen Hidden Classes verwendet werden. Dann würde die Optimierung mit der Vorhersage der Hidden Class bei der nächsten Ausführung niemals greifen. Für diesen Fall kommen verschiedene Codefragmente zum Einsatz, anhand deren 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.

1.7.4 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 Java-

Script 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 Collector 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 eine längere Laufzeit einer Applikation zu Problemen führt. Aus diesem Grund existiert ein zweiter Algorithmus, der ebenfalls die Elemente des Speichers nach solchen durchsucht, die nicht mehr benötigt werden, diese markiert und löscht.

Der wichtigste Unterschied zwischen beiden ist, dass der zweite Algorithmus den Speicher defragmentiert, also die verbleibenden Objekte im Speicher so umordnet, dass der Speicher danach möglichst wenige Lücken aufweist. Diese Defragmentie-

rung 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 das Mark-and-sweep ohne Optimierungen mit 50 Millisekunden und schließlich das 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.

1.8 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 sie 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. Das Gleiche gilt für Fehlerbehebungen. Tritt ein Fehler in der DNS-Auflösung auf, wird er einmal behoben, und die Lösung wirkt an allen Stellen, an denen die Bibliothek eingesetzt wird. Das führt aber gleich auch zur Kehrseite 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.

1.8.1 Event-Loop

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 anonymer 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.

```
import { EventEmitter } from 'events';

const myObj = new EventEmitter();
myObj.on('myEvent', (data) => {
  console.log(data); // Ausgabe: Hello World
});
myObj.emit('myEvent', 'Hello World');
```

Listing 1.2 Eventgetriebene Entwicklung in Node.js

Mit der `on`-Methode eines Objekts, das Sie von `events.EventEmitter`, einem Bestandteil der Node.js-Plattform, 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 in diesem Fall als Arrow-Funktion umgesetzt ist, 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 Skripts wird auf `myObj` die `emit`-Methode aufgerufen. Diese sorgt dafür, dass sämtliche durch die `on`-Methode registrierten Callback-Funktionen ausgeführt werden.

Was in diesem Beispiel mit einem selbst erstellten Objekt funktioniert, verwendet Node.js, um eine Vielzahl asynchroner Aufgaben zu erledigen. Die Callback-Funktionen werden allerdings nicht parallel ausgeführt, sondern sequenziell. Durch den Single-threaded-Ansatz von Node.js entsteht das Problem, dass nur eine Operation zu einem Zeitpunkt ausgeführt werden kann. Vor allem zeitintensive Lese- oder Schreiboperationen würden die gesamte Ausführung der Anwendung blockieren. Aus diesem Grund werden sämtliche Lese- und Schreiboperationen mithilfe des Event-Loops

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 sie an den Event-Loop 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 Event-Loop zurückübermittelt. Es tritt ein Event auf, und der Event-Loop sorgt dafür, dass die zugehörigen Callback-Funktionen ausgeführt werden. Wie der Event-Loop funktioniert, sehen Sie in Abbildung 1.5.

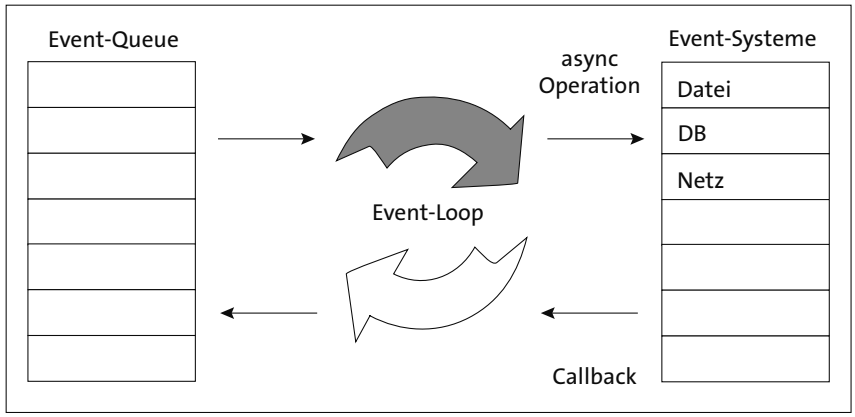


Abbildung 1.5 Der Event-Loop

Der ursprüngliche Event-Loop, der bei Node.js zum Einsatz kommt, basiert auf `libev`, einer Bibliothek, die in C geschrieben ist und für eine hohe Performance und einen großen Umfang an Features steht. `libev` baut auf den Ansätzen von `libevent` auf, verfügt aber 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 Event-Loop allerdings abstrahiert, und damit wurde eine bessere Portierbarkeit auf andere Plattformen erreicht.

1.8.2 Eingabe und Ausgabe

Der Event-Loop 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 dem Dateisystem zu interagieren. Bei der Implementierung serverseitiger Anwendungen spielen Zugriffe auf das Dateisystem eine wichtige 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 Event-Loop 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 Event-Loop, 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.

1.8.3 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 Version 0.6 von Node.js die Bibliothek `libuv` eingeführt. Sie dient primär der Abstraktion von Unterschieden zwischen verschiedenen Betriebssystemen. Der Einsatz von `libuv` macht es also möglich, dass Node.js auch auf Windows-Systemen lauffähig ist. Der Aufbau ohne `libuv`, wie er bis 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 Event-Loop und den 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 vielen Jahren ohne Probleme unter Windows. Schwieriger wird die Situation beim Event-Loop und bei den asynchronen Dateisystemoperationen. Einige Komponenten von `libev` müssten beim Einsatz unter Windows umgeschrieben werden. Außerdem basiert `libev` auf nativen Implementierungen des Betriebssystems der `select`-Funktion, unter Windows steht allerdings mit `IOCP` eine für das Betriebssystem optimierte Variante zur Verfügung. Um nicht verschiedene Versionen von Node.js für die unterschiedlichen Betriebssysteme erstellen zu müssen, entschieden sich die Entwickler, mit `libuv` eine Abstraktionsschicht einzufügen, die es erlaubt, für Linux-Systeme `libev` und für Windows `IOCP` zu verwenden. Mit `libuv` wurden einige Kernkonzepte von Node.js angepasst. Es wird beispielsweise nicht mehr von Events, sondern von Operationen gesprochen. Eine Operation wird an die `libuv`-Komponente weitergegeben, innerhalb von `libuv` wird die Operation an die darunterliegende Infrastruktur, also `libev` beziehungsweise `IOCP`, weitergereicht. So bleibt die Schnittstelle von Node.js unverändert, unabhängig davon, welches Betriebssystem eingesetzt 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` komplexe Features wie das Erstellen und das Spawnen von Kindprozessen sowie 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.

1.8.4 DNS

Die Wurzeln von Node.js liegen im Internet. 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 Protocol ist die Adresse eine 32-Bit-Zahl, die in vier Blöcken mit je 8 Bit dargestellt wird. In der sechsten Version des Protokolls haben die Adressen eine Größe von 128 Bit und werden in acht Blöcke mit Hexadezimalzahlen aufgeteilt. Mit diesen kryptischen Adressen will man in den seltensten Fällen direkt arbeiten, vor allem wenn eine dynamische Vergabe über DHCP hinzukommt. Die Lösung hierfür besteht im Domain Name System, kurz DNS. Das DNS ist ein Dienst zur Namensauflösung im Netz. Es sorgt dafür, dass Domainnamen in IP-Adressen gewandelt werden. Außerdem gibt

es die Möglichkeit der Reverse-Auflösung, bei der eine IP-Adresse in einen Domainnamen übersetzt wird. Falls Sie in Ihrer Node.js-Applikation einen Webservice anbinden oder eine Webseite auslesen möchten, kommt auch hier das DNS zum Einsatz.

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

1.8.5 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.

1.8.6 Zlib

Bei der Erstellung von Webapplikationen müssen Sie als Entwickler 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.

1.8.7 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, das sowohl Anfragen als auch Antworten des HTTP-Protokolls ausliest. Das bedeutet für Sie als Entwickler konkret, dass Sie mit dem HTTP-Parser beispielsweise die verschiedenen Informationen des HTTP-Headers oder den Text der Nachricht selbst auslesen können.

Das primäre Entwicklungsziel von Node.js ist es, eine performante Plattform für Webapplikationen zur Verfügung zu stellen. Um diese Anforderung zu erfüllen, baut Node.js auf einem modularen Ansatz auf. Dieser erlaubt die Einbindung externer Bibliotheken wie beispielsweise der bereits beschriebenen `libuv` oder des HTTP-Parsers. Der modulare Ansatz wird durch die internen Module der Node.js-Plattform weitergeführt und reicht bis zu den Erweiterungen, die Sie für Ihre eigene Applikation erstellen. Im Laufe dieses Buchs 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.

1.9 Zusammenfassung

In diesem Kapitel haben Sie die Grundlagen von Node.js kennengelernt:

- **Geschichte von Node.js**: Sie haben mehr über die Entwicklungsgeschichte der Plattform gelernt, wie Node.js entstanden ist und wie es sich entwickelt hat.
- **Organisation von Node.js**: Sie wissen nun, wie die Plattform entwickelt wird und welche wichtigen Prozesse und Abstimmungen im Hintergrund ablaufen.
- **Versionierung**: Sie kennen das Versionsschema von Node.js und wissen, wie der Releasezyklus aufgebaut ist.
- **Einsatzgebiete**: Sie wissen, wofür Sie Node.js benutzen können und was die wichtigsten Merkmale der Plattform sind.
- **Architektur**: Sie kennen die wichtigsten Elemente der Plattform und wissen, wie sie miteinander interagieren.

In den folgenden Kapiteln lernen sie aufbauend auf diesen Grundlagen, wie Sie verschiedene Arten von Applikationen implementieren können.

Kapitel 6
Express

Die Zukunft war früher auch besser!
– Karl Valentin

Express ist seit Jahren das populärste Web-Application-Framework für Node.js. TJ Holowaychuck hat das Open-Source-Projekt im Juni 2009 ins Leben gerufen. Es hat sich zur Aufgabe gemacht, Ihnen die Entwicklung von Webapplikationen zu erleichtern. Der Fokus von Express liegt auf Geschwindigkeit, überschaubarem Umfang des Kern-Frameworks und einer leicht zu erweiternden Schnittstelle. Die durchdachte Architektur macht es möglich, dass dies bis heute durchgehalten werden kann, und damit wurde das Framework zu einem nahezu unverzichtbaren Begleiter, wenn es um die Entwicklung von Webserverapplikationen auf Basis von Node.js geht. Der Grund dafür, dass es Frameworks wie Express gibt, ist, dass bei der Webentwicklung häufig Standardaufgaben zu lösen sind. So gibt es beispielsweise in PHP das Symfony-Framework, in Python können Sie auf Django zurückgreifen, und Ruby on Rails bietet eine Lösung für Webapplikationen unter Ruby. Sie können Ihre Applikation zwar vollständig in der jeweiligen Sprache, in diesem Fall in Node.js, ohne Zuhilfenahme von weiteren Bibliotheken und Frameworks umsetzen, allerdings verlieren Sie sehr viel Zeit mit der Umsetzung der Basisinfrastruktur. Denken Sie nur an die `createServer`-Callback-Funktion im vorangegangenen Kapitel. Hier mussten Sie sich selbst um das Parsen der URL und die Ausführung der entsprechenden Aktion kümmern. Neben dem Umgang mit Anfragen und dem Auflösen von URLs fallen weitere Standardaufgaben wie Sessionhandling, Authentifizierung oder Dateiuploads an. Für all diese Aufgaben gibt es bereits etablierte Lösungen, die unter Federführung von Express zu einem Framework zusammengefügt werden. Aufgrund seiner Stabilität über die letzten Jahre hinweg und seiner erweiterbaren Architektur dient Express als Grundlage für eine Vielzahl weiterer Bibliotheken und Frameworks wie beispielsweise Nest, das wir uns in Kapitel 14, »Webapplikationen mit Nest«, näher ansehen werden.

6.1 Aufbau

Express ist ein kompaktes Framework mit einem überschaubaren Funktionsumfang. Es lässt sich jedoch mit sogenannten Middleware-Komponenten gut erweitern. Der

Aufbau von Express weist, ähnlich wie Node.js selbst, einen mehrschichtigen Aufbau auf, wie Sie Abbildung 6.1 entnehmen können.

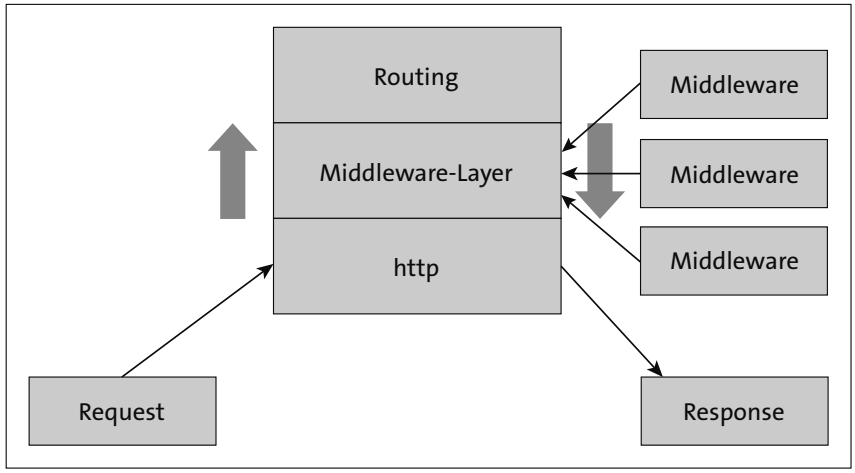


Abbildung 6.1 Der Aufbau von Express

Das `http`-Modul von Node.js bildet die Grundlage für Express. Mit dem `http`-Modul wird der Serverprozess erstellt, auf dem Express im Weiteren aufsetzt. Außerdem stehen Ihnen die `Request`- und `Response`-Objekte zur Verfügung, um auf die Anfrageinformationen zurückzugreifen und die Antwort an den Client zu erstellen. Intern bedient sich Express.js beispielsweise der URL, die der Benutzer im Browser eingegeben hat, um das Routing innerhalb der Applikation umzusetzen.

Die zweite Ebene der Architektur von Express stellt der Middleware-Layer dar. Im Kontext von Express ist eine Middleware eine Funktion, die zwischen der eingehenden Anfrage und der Antwort des Servers an den Client steht. Mehrere Middleware-Funktionen können zu einer Kette zusammengesetzt werden, um basierend auf der Anfrage des Clients bestimmte Aktionen auszuführen. Bis zur dritten Version von Express bildete Connect diesen Middleware-Layer. Mit der vierten Version verzichteten die Entwickler auf diese zusätzliche Abhängigkeit und entwickelten einen eigenständigen Layer, der allerdings größtenteils kompatibel bleibt.

Die dritte Ebene der Architektur von Express bildet der Router. Diese Komponente von Express steuert, welche Funktion abhängig von der aufgerufenen URL ausgeführt werden soll, um eine Antwort an den Client zu generieren. Beim Routing werden sowohl die HTTP-Methode als auch der URL-Pfad betrachtet.

Im Zuge dieses Kapitels erstellen Sie eine Webapplikation, mit der Sie eine Filmdatenbank verwalten können. Bevor Sie mit der Arbeit an der Applikation beginnen, müssen Sie sie zunächst initialisieren und Express installieren.

6.2 Installation

Express folgt dem Standard von NPM-Paketen: Es ist als Open-Source-Projekt frei verfügbar, unterliegt der MIT-Lizenz und wird auf GitHub entwickelt. Das Express-Paket ist über einen Paketmanager Ihrer Wahl, also beispielsweise NPM, verfügbar. Für die Beispiel-Applikation legen Sie ein neues Verzeichnis an und generieren mithilfe von `npm init -y` auf der Kommandozeile die *package.json*-Datei. Um das ECMAScript-Modulsystem korrekt zu unterstützen, fügen Sie das Feld `type` mit dem Wert `module` ein. Anschließend installieren Sie Express mit dem Kommando `npm install express`. Danach können Sie mit einer einfachen Applikation die Funktionsfähigkeit des Frameworks testen. In Listing 6.1 sehen Sie den Quellcode dieses ersten Schritts. Sie speichern ihn in einer Datei mit dem Namen *index.js*.

```
import express from 'express';

const app = express();

app.get('/', (request, response) => {
  response.send('My first express application');
});

app.listen(8080, () => {
  console.log('Filmdatenbank erreichbar unter http://localhost:8080');
});
```

Listing 6.1 Die erste Express.js-Applikation (»index.js«)

Im ersten Schritt binden Sie das `express`-Paket ein. Der Standardexport des Pakets ist eine Funktion, mit der Sie mit dem `app`-Objekt die Basis für Ihre Applikation erstellen. Die `get`-Methode des `app`-Objekts erzeugt eine Route, in diesem Fall für den Pfad */*. Die Callback-Funktion, die Sie als zweites Argument übergeben, ist der Request-Handler für diese Route. Sie kümmert sich darum, für die eingehende Anfrage eine passende Antwort zu generieren. Hierfür haben Sie über die Parameter `request` und `response` Zugriff auf die eingehende Anfrage und die ausgehende Antwort. Mit der `send`-Methode des `response`-Objekts senden Sie die Antwort an den Client. Im letzten Schritt binden Sie Ihre Applikation an den TCP-Port `8080` und erzeugen eine Konsolenausgabe, sobald der Server bereit ist. Intern wird an dieser Stelle ein Server mit dem `http`-Modul von Node.js erstellt und an den angegebenen Port gebunden. Nachdem Sie die Applikation mit dem Befehl `node index.js` gestartet haben, können Sie sie im Browser über die URL *http://localhost:8080* erreichen und testen. Das Ergebnis sehen Sie in Abbildung 6.2.

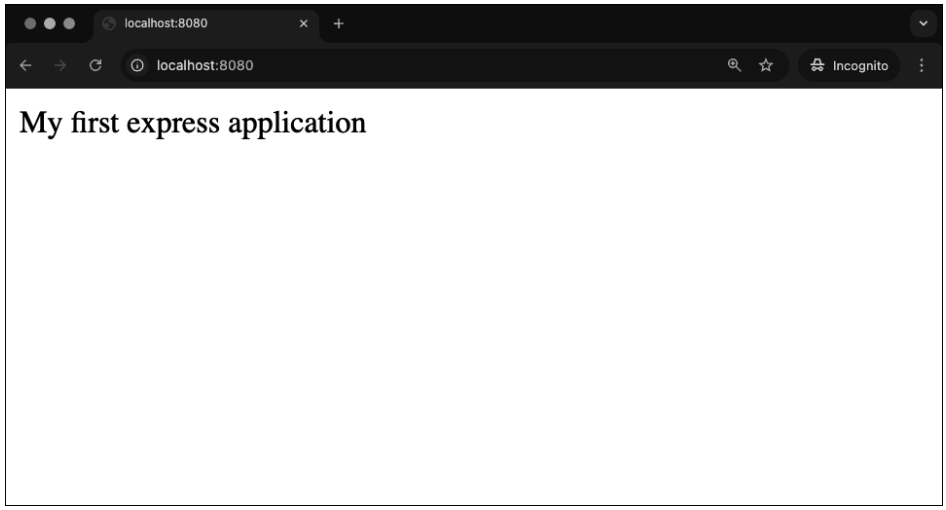


Abbildung 6.2 Ausgabe der Express-Applikation im Browser

Auf Basis dieser funktionierenden Grundlage können Sie nun dazu übergehen, dieses Beispiel Schritt für Schritt zu einer voll funktionsfähigen Applikation zu erweitern.

6.3 Grundlagen

Der Ablauf, nach dem eine Express-Applikation arbeitet, weist stets das gleiche Schema auf. Ein Request eines Clients geht beim Express-Server ein. Anhand der verwendeten `http`-Methode und des URL-Pfads wählt Express eine passende Route und führt eine oder mehrere Callback-Funktionen aus. Innerhalb dieser Callback-Funktion können Sie, wie beim `http`-Modul, auf das `Request`- und das `Response`-Objekt zugreifen. Diese beiden Objekte sind neben dem Router und den Middleware-Komponenten die Herzstücke einer Applikation.

6.3.1 Request

Das `Request`-Objekt ist das erste Argument der Routing-Callback-Funktionen von Express und steht für die Anfrage des Benutzers. Sie können dieses Objekt durch den Einsatz von Middleware-Komponenten wie dem Body-Parser und dem Cookie-Parser erweitern, damit Sie beispielsweise Cookies oder andere Aspekte in der Kommunikation mit dem Client komfortabler verwalten können. Aber auch im Standardmodus enthält es viele hilfreiche Informationen. Tabelle 6.1 stellt Ihnen einige der wichtigsten Eigenschaften des `Request`-Objekts vor.

Eigenschaft	Bedeutung
`method`	Enthält die HTTP-Methode, mit der die Anfrage an den Server gesendet wurde.
`originalUrl`	Diese Eigenschaft enthält die ursprüngliche Anfrage-URL. Damit kann die `url`-Eigenschaft, die die gleichen Informationen enthält, für applikationsinterne Zwecke verändert werden.
`params`	Der Wert der `params`-Eigenschaft sind die variablen Anteile der URL. Wie Sie diese in Express.js definieren und nutzen können, erfahren Sie weiter unten in diesem Kapitel.
`path`	Mit dieser Eigenschaft können Sie auf den URL-Pfad zugreifen.
`protocol`	Das Protokoll der Anfrage kann beispielsweise HTTP oder HTTPS sein.
`query`	Der Query-String ist ein Teil der URL. Sie können auf ihn mit der `query`-Eigenschaft zugreifen.

Tabelle 6.1 Die wichtigsten Eigenschaften des »Request«-Objekts

Neben den Eigenschaften haben Sie außerdem Zugriff auf einige Methoden, mit denen Sie weitere Informationen zur eingehenden Anfrage auslesen können. Mit der `get`-Methode sind Sie in der Lage, Header-Felder aus der Anfrage auszulesen. Sind Sie beispielsweise am `Accept`-Feld interessiert, lautet der Aufruf `request.get('Accept')`. Die Groß- und Kleinschreibung des Feldnamens spielt bei dieser Methode keine Rolle. In Listing 6.2 sehen Sie, wie Sie auf die verschiedenen Eigenschaften des Request-Objekts zugreifen können.

```
import express from 'express';

const app = express();

app.get('/', (request, response) => {
  console.log('Method:', request.method);
  console.log('URL:', request.originalUrl);
  console.log('Parameter:', request.params);
  console.log('Path:', request.path);
  console.log('Protocol:', request.protocol);
  console.log('Query:', request.query);
  console.log('Content-Type:', request.get('Accept'));
  response.send('My first express application');
});
```

```
app.listen(8080, () => {
  console.log('Filmdatenbank erreichbar unter http://localhost:8080');
});
```

Listing 6.2 Zugriff auf verschiedene Informationen des Request-Objekts (»index.js«)

Starten Sie die Applikation neu und greifen mit Ihrem Browser auf die Adresse *http://localhost:8080/?name=Jane* zu, erhalten Sie eine Ausgabe wie in Listing 6.3.

```
$ node index.js
Filmdatenbank erreichbar unter http://localhost:8080
Method: GET
URL: /?name=Jane
Parameter: [Object: null prototype] {}
Path: /
Protocol: http
Query: [Object: null prototype] { name: 'Jane' }
Content-Type: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
```

Listing 6.3 Konsolenausgabe mit den Anfragedetails

Hilfreiche NPM-Skripte

Sie können sich Ihre Arbeit erleichtern, wenn Sie sich für häufig wiederkehrende Aufgaben kleine Hilfsskripte schreiben. So können Sie beispielsweise ein Startskript definieren, das Ihre Applikation regulär startet, und ein weiteres, das die Applikation in den Watch-Modus versetzt und sie bei Änderungen am Dateisystem automatisch neu startet. In Listing 6.4 sehen Sie die angepasste Version der *package.json*-Datei.

```
{
  …
  "scripts": {
    "start": "node index.js",
    "start:dev": "node --watch index.js"
  },
  …
}
```

Listing 6.4 »package.json«-Datei mit Hilfsskripten

Mit dieser Datei können Sie Ihre Applikation mit dem Kommando `npm start` regulär starten. Mit `npm run start:dev` starten Sie sie im Entwicklungsmodus. Hier müssen Sie sich nicht mehr um den manuellen Neustart Ihres Servers bei Änderungen am Code kümmern.

6.3.2 Response

Das `Response`-Objekt, auf das Sie über das zweite Argument der Routing-Funktion Zugriff haben, steht für die Antwort an den Client. Da Sie überwiegend schreibend auf dieses Objekt zugreifen, weist es auch deutlich mehr Methoden als Eigenschaften auf. Die wichtigste Eigenschaft des Objekts ist `headersSent`. Mit diesem booleschen Wert erfahren Sie, ob die HTTP-Header der Antwort bereits gesendet wurden. Ist dies der Fall, können Sie sie nicht mehr modifizieren. Tabelle 6.2 fasst die wichtigsten Methoden des `Response`-Objekts für Sie zusammen.

Methode	Bedeutung
`get(field)`	Liest das angegebene Header-Feld der Antwort aus.
`set(field[, value])`	Setzt den Wert des angegebenen Header-Felds.
`cookie(name, value[, options])`	Setzt einen Cookie-Wert.
`redirect([status, ]path)`	Leitet die Anfrage weiter.
`status(code)`	Setzt den Statuscode der Antwort.
`send([body])`	Sendet die HTTP-Antwort.
`json([body])`	Sendet die HTTP-Antwort. Das übergebene Objekt wird in ein JSON-Objekt umgewandelt, und die korrekten Response-Header werden gesetzt.
`end([data][, encoding])`	Sendet die HTTP-Antwort. Diese Methode sollten Sie vorrangig nutzen, wenn Sie keine Nutzdaten wie HTML-Strukturen versenden. Andernfalls sollten Sie besser die `send`-Methode einsetzen.

Tabelle 6.2 Die wichtigsten Methoden des »Response«-Objekts

Im nachfolgenden Listing 6.5 sehen Sie, wie Sie mit dem `Response`-Objekt arbeiten können und statt der bisher verwendeten `send`-Methode mit der `json`-Methode eine Objektstruktur zum Client senden können.

```
import express from 'express';

const app = express();

app.get('/', (request, response) => {
```

```
  const movie = { id: 1, title: 'Inception' };
  response.set('Last-Modified', new Date().toUTCString());
  response.set('Cache-Control', 'no-cache');
  response.set('X-Movie-Id', movie.id);
  response.status(200).json(movie);
});

app.listen(8080, () => {
  console.log('Filmdatenbank erreichbar unter http://localhost:8080');
});
```

Listing 6.5 Arbeiten mit dem »Response«-Objekt (»index.js«)

Die Handler-Funktion für eine GET-Anfrage auf den Pfad / soll ein einfaches Movie-Objekt mit den Eigenschaften `id` und `title` zurückliefern, das hier exemplarisch erzeugt wird, und setzt anschließend drei Header-Felder:

- `Last-Modified`: Dieses Feld gibt an, wann die Ressource zuletzt verändert wurde.
- `Cache-Control`: Mit diesem Header-Feld können Sie das Caching-Verhalten eines Clients beeinflussen. Der Wert `no-cache` gibt beispielsweise an, dass der Client die Ressource nicht cachen soll.
- `X-Movie-Id`: Bei diesem Header-Feld handelt es sich um einen sogenannten Custom-Header. Solche selbst definierten Felder beginnen in der Regel mit einem `X-`. Sie können diese Felder nutzen, um Informationen zwischen Client und Server auszutauschen.

Im letzten Schritt setzt die `status`-Methode den HTTP-Statuscode der Antwort auf `200` (OK) und nutzt die `json`-Methode, um die Antwort zu senden. Das Setzen des Status ist eigentlich überflüssig, da Express den Code `200` als Standard verwendet. Diese Methode wird vor allem interessant für Sie, wenn Sie von diesem Standard abweichen müssen, um beispielsweise einen Fehler bei der Verarbeitung zu melden.

Nach dieser kurzen Einführung in die Elemente zur Behandlung von Anfrage und Antwort beschäftigen sich die folgenden Abschnitte mit dem Setup und der Architektur einer Express-Applikation.

6.4 Architektur einer Express-Applikation

Als generelle Best Practice im Umgang mit Express hat sich herausgebildet, dass eine Applikation in möglichst separate Bestandteile unterteilt werden sollte, die Sie jeweils in eigenen Dateien ablegen. Mit dieser Strategie erstellen Sie, je nach Größe Ihrer Applikation, zwar viele Dateien, durch eine gut strukturierte Verzeichnishierarchie können Sie die Dateien allerdings schnell lokalisieren. Eine Datei enthält

lediglich eine Komponente und somit eine abgeschlossene Einheit. Zur Strukturierung einer Express-Applikation bietet sich ein klassischer MVC-Ansatz an.

MVC – das Model-View-Controller-Pattern

Das MVC-Pattern dient der Strukturierung von Applikationen. Gerade in der Webentwicklung hat es sich zu einem wichtigen Standard entwickelt. Dieses Muster beschreibt, wie Sie Ihre Applikation strukturieren, welche Aufgaben die jeweiligen Teile haben und wie sie zusammenwirken. Der Name MVC enthält bereits die drei Bestandteile des Musters:

- **Model**: Die Models einer MVC-Applikation dienen der Datenhaltung. Models kapseln sämtliche Operationen, die mit den Daten Ihrer Applikation zu tun haben. Das betrifft sowohl das Erstellen wie auch das Modifizieren oder Löschen von Informationen. Typischerweise kapselt ein Model Datenbankzugriffe. Neben den reinen Daten und der dazugehörigen Logik für den Umgang mit ihnen kapseln Models auch die Businesslogik Ihrer Applikation.
- **View**: Die Aufgabe der Views ist die Darstellung von Informationen. Die Views einer Express.js-Applikation sind meistens HTML-Templates, die vor der Auslieferung an den Client mit den dynamischen Daten Ihrer Applikation befüllt werden. In modernen API-lastigen Applikationen tritt dieser Aspekt der MVC-Architektur zunehmend in den Hintergrund, da selten Templates gerendert werden und stattdessen häufig JSON-Objekte vom Server an den Client gesendet werden.
- **Controller**: Ein Controller enthält die Steuerungslogik Ihrer Applikation. Der Controller bringt Models und Views zusammen. Sie sollten darauf achten, dass Ihre Controller nicht zu umfangreich werden. Enthält der Controller zu viel Logik, sollten Sie diese in Hilfsfunktionen oder Models auslagern.

Beim Aufbau Ihrer Applikation ist es von großer Bedeutung, dass Sie bei der Strukturierung der Datei- und Verzeichnishierarchie einer einheitlichen Konvention folgen, um die Wartbarkeit und Erweiterbarkeit über den Lebenszyklus der Applikation zu erhalten.

6.4.1 Struktur einer Applikation

Die Wahl der Verzeichnisstruktur hängt sehr stark vom Umfang Ihrer Applikation ab. Vermeiden Sie, dass die Struktur schon zu Beginn unnötig komplex wird, da Sie am Anfang der Arbeit an einer Applikation meist noch nicht genau wissen, wie diese am Ende aussehen wird. Je komplexer die Struktur wird, desto aufwendiger werden Anpassungen daran. Sie starten also mit einer möglichst flachen Hierarchie und strukturieren die Verzeichnisse und Dateien je nach Anforderung um. Das Modulsystem von Node.js und moderne Entwicklungsumgebungen unterstützen Sie bei diesem kontinuierlichen Refactoring-Prozess, indem Dateien und Verzeichnisse einfach ver-

schoben und die `import`-Statements automatisiert angepasst werden können. Eine Applikation besteht in der Regel aus folgenden Bestandteilen: aus Models, Views, Controllern, Routern und Helpern.

Neben den bereits vorgestellten Models, Views und Controllern übernehmen Router die Definition der verfügbaren Methoden- und Pfadkombinationen und verbinden sie mit den Methoden der Controller. Helper sind üblicherweise Sammlungen von Hilfsfunktionen.

Struktur für kleine Applikationen

Bei sehr kleinen Applikationen mit nur zwei oder drei Endpunkten und insgesamt wenig Business-Logik erstellen Sie pro Komponente eine Datei und legen diese in einem Verzeichnis ab.

Dieser Strukturierungsansatz funktioniert nur für wirklich sehr überschaubare Applikationen oder kleine Prototypen. Sobald Ihr Projekt über mehr als drei oder vier separate Endpunkte verfügt, sollten Sie auf die nächstgrößere Variante wechseln oder direkt mit dieser beginnen, da die Migration der Struktur in diesem Fall nur unnötig aufwendig wird.

Struktur für mittelgroße Applikationen

In die Kategorie der mittelgroßen Applikationen passen Webanwendungen, die 10 bis 15 eigenständige Endpunkte, also separate Routen, aufweisen. Diese Variante der Strukturierung ist ein guter Startpunkt für normale Webanwendungen. Bei dieser Struktur werden die verschiedenen Bestandteile – also Models, Views und Controller – in separaten Verzeichnissen abgelegt. Alle Strukturen liegen in eigenen Dateien und sind sprechend benannt. Damit sich die Strukturen besser in der Entwicklungsumgebung lokalisieren lassen, hat sich als Best Practice herausgebildet, die Art der Struktur in den Dateinamen einfließen zu lassen. Ein Controller, der für den Login-Prozess Ihrer Applikation verantwortlich ist, liegt dann beispielsweise in einer Datei mit dem Namen *login.controller.js* im Verzeichnis *controllers*.

Der Vorteil dieser Strukturierungsvariante ist, dass alle Strukturen getrennt voneinander abgelegt werden und auch innerhalb der jeweiligen Komponente, beispielsweise bei Models, eine thematische Trennung auf Dateiebene stattfindet. Das entkoppelt die verschiedenen Bereiche der Applikation und hält die Anzahl der Dateien pro Verzeichnis gering. Für kleinere und mittelgroße Applikationen ist dies ein sehr guter Ansatz.

Struktur für große Applikationen

Was den Umfang Ihrer Applikation angeht, sind Sie bei der Verwendung von Express.js kaum beschränkt. Bei der Entwicklung der Datei- und Verzeichnisstruktur

von umfangreichen Applikationen sollten Sie jedoch einen Ansatz wählen, der Ihnen eine thematische Trennung der Module Ihrer Applikation erlaubt. Eine solche Strategie ermöglicht Ihnen, mit mehreren Teams parallel Ihre Applikation weiterzuentwickeln und die einzelnen Module unabhängig voneinander zu verwalten. Auch die Lokalisierung von Codestellen wird dadurch erheblich vereinfacht.

In dieser Variante gibt es für die einzelnen fachlichen Bereiche der Applikation separate Verzeichnisse. Die Auswahl und Abgrenzung dieser Bereiche hängt ganz von Ihnen und den Anforderungen Ihrer Applikation ab. Typischerweise werden Module so gewählt, dass sie ein Thema komplett abdecken. Diese Vorgehensweise ermöglicht saubere Schnittstellen zur übrigen Applikation. Die Möglichkeiten reichen hier von logischen Einheiten innerhalb eines Endpunkts bis hin zu Gruppen von mehreren themenverwandten Endpunkten. Jedes Modul beinhaltet wiederum Unterverzeichnisse, die jeweils die Models, Views und Controller des Moduls enthalten. Außerdem definiert jedes Modul seinen eigenen Router. Nachdem Sie nun einen Überblick über die Strukturierungsmöglichkeiten Ihrer Applikation haben, können Sie im nächsten Schritt dazu übergehen, Ihre erste Beispielapplikation zu erweitern.

6.5 Filmdatenbank

Als Beispielapplikation für Express soll eine Filmdatenbank dienen. Diese deckt alle wesentlichen Aspekte einer typischen Applikation ab. Sie können Filme der Datenbank hinzufügen, die bestehenden Datensätze ansehen, sie bei Bedarf aktualisieren und auch wieder löschen. Außerdem lassen sich weitere Funktionalitäten wie beispielsweise Bewertungen realisieren. Diese Applikation dient auch den folgenden Kapiteln als Grundlage, wenn es darum geht, Template-Engines zu integrieren, verschiedene Datenbanken anzubinden oder Benutzer zu authentifizieren.

Für die Beispielapplikation wählen wir die Struktur einer großen Applikation, obwohl diese zunächst aus nur einem Modul bestehen wird. Der Grund dafür ist, dass diese Struktur die größte Flexibilität bietet und Sie so mehr Features von Express kennenlernen können.

Zu Beginn dieses Kapitels haben Sie erfahren, wie Sie Ihre Express-Applikation initialisieren. Auf dieser Grundlage setzen wir an dieser Stelle auf. Sie sollten also über ein neues Verzeichnis mit einer *package.json*-Datei verfügen, in der Sie das ES-Modulsystem aktiviert haben. Außerdem sollte Express installiert sein, und Sie sollten über eine Einstiegsdatei mit dem Namen *index.js* verfügen. Achten Sie während der Entwicklung darauf, dass diese Datei nur für die Initialisierung der Applikation verantwortlich ist und nicht noch weitere Aufgaben übernimmt. Sollten Sie bereits Request-Handler-Funktionen definiert haben, entfernen Sie diese nun wieder, sodass der Code der *index.js*-Datei wie in Listing 6.6 aussieht.

```
import express from 'express';

const port = 8080;

const app = express();

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.6 Einstieg in die Applikation (»index.js«)

In der Einstiegsdatei erzeugen Sie das App-Objekt und binden es an einen TCP-Port. Hier ist es empfehlenswert, den Port in einer Variablen zu speichern, sodass Sie ihn bei Bedarf leicht ändern können. Mit diesem Stand ist die Applikation noch nicht in der Lage, mit eingehenden Anfragen umzugehen.

6.5.1 Routing

Im ersten Schritt soll Ihre Applikation eine einfache Liste von Filmtiteln ausgeben. Zu diesem Zweck definieren Sie zunächst ein Modul, also eine Einheit in Ihrer Applikation, die alle Aspekte zum Thema Filme in der Applikation abdeckt. Den Vorgaben für die Struktur folgend, benötigen Sie zunächst ein Verzeichnis mit dem Namen des Moduls, hier *movies*. In diesem Verzeichnis legen Sie eine Datei mit dem Namen *index.js* an. Die Datei bildet den Einstieg in das Modul. Binden Sie es zu einem späteren Zeitpunkt in Ihre Applikation ein, sorgt diese Datei dafür, dass alle weiteren relevanten Teile des Moduls geladen werden, sodass die Integration möglichst wenig Aufwand verursacht.

Mit der Router-Funktion aus dem Express-Paket erzeugen Sie eine neue `Router`-Instanz. Die Einstiegsdatei des Moduls exportiert das `Router`-Objekt, das Sie später in die Einstiegsdatei Ihrer Applikation einbinden, um das Modul zu aktivieren. Neben der Erzeugung des `Router`-Objekts und der Routendefinition kümmert sich diese Datei aktuell noch um die Datenhaltung und die Beantwortung der Anfrage in der Callback-Funktion der Route. Listing 6.7 enthält den Quellcode der Datei.

```
import { Router } from 'express';

const router = Router();

const data = [
  { id: 1, title: 'Iron Man', year: '2008' },
  { id: 2, title: 'Thor', year: '2011' },
```

```
  { id: 3, title: 'Captain America', year: '2011' },
];

router.get('/', (request, response) => {
  response.json(data);
});

export default router;
```

Listing 6.7 Routerdatei des Movie-Moduls (»movies/index.js«)

Bevor Sie sich die Daten im Browser anzeigen lassen können, müssen Sie den Router in Ihre Applikation einbinden. Dies geschieht in der *index.js*-Datei im Wurzelverzeichnis Ihrer Applikation, wie in Listing 6.8 zu sehen.

```
import express from 'express';
import moviesRouter from './movies/index.js';

const port = 8080;

const app = express();

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.8 Einbindung des Routers (»index.js«)

Damit der Code etwas sprechender ist, benennen Sie den `router` beim Import in `moviesRouter` um. Mit der `use`-Methode geben Sie an, dass der `moviesRouter` für den Pfad `/movie` verantwortlich ist. Die Pfadangaben des Routers sind immer relativ zu diesem Basispfad. Ein `/` im `moviesRouter` wird damit zu `/movies`. Damit die Benutzer Ihrer Applikation nicht wissen müssen, dass die Filmliste unter der URL *http://localhost:8080/movies* zu finden ist, erzeugen Sie mit der `get`-Route für den Pfad `/` eine Weiterleitung auf `/movies` und ermöglichen damit den Benutzern, die Liste auch über die Adresse *http://localhost:8080* zu erreichen. Somit haben Sie den Einstieg in Ihre Applikation festgelegt. Starten Sie Ihre Applikation mit dem Kommando `npm start`, `node index.js` oder `npm run start:dev`, wobei Letzteres den Entwicklungsmodus aktiviert, können Sie sie in Ihrem Browser öffnen und sehen eine Ansicht ähnlich wie die in Abbildung 6.3.

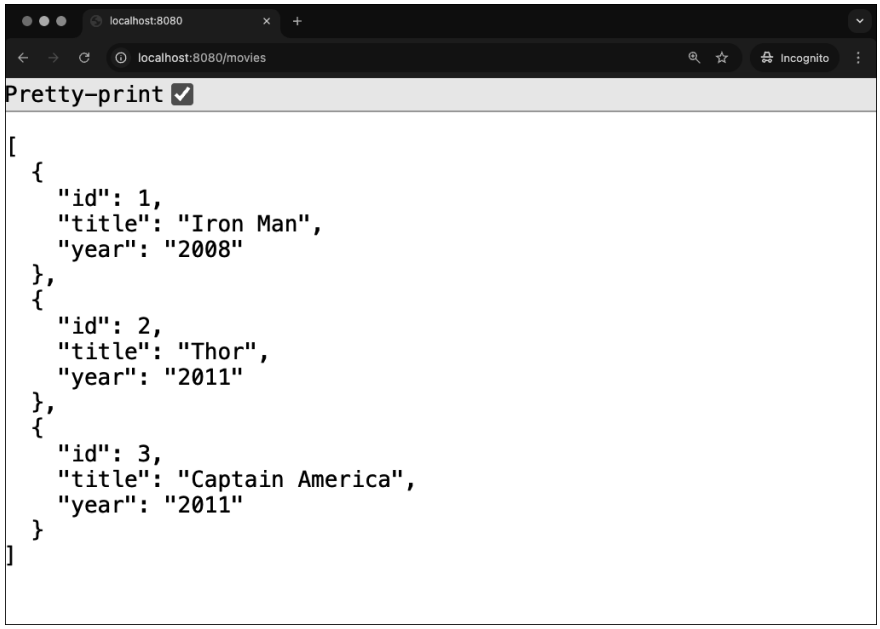


Abbildung 6.3 Die Filmliste im Browser

Muster in Routen

Statische Routen, wie Sie sie bisher kennengelernt haben, decken die meisten Anwendungsfälle in einer Webapplikation ab. Es gibt jedoch Anwendungsfälle, bei denen Sie mit so wenig Flexibilität an die Grenzen stoßen. Aus diesem Grund ist es in Express möglich, dynamische Routen zu formulieren.

Muster	Beispiel	Pfad	Bedeutung
?	`/ab?c`	*/abc /ac*	Zeichen kann vorkommen, muss aber nicht.
+	`/ab+c`	*/abc /abbc*	Zeichen kommt einmal oder mehrfach vor.
*	`/a*c`	*/ac /aABCc*	Beliebige Zeichen.

Tabelle 6.3 Muster in Routen

Zusätzlich zu den Mustern aus Tabelle 6.3 können Sie Gruppen von Zeichen mithilfe von Klammern gruppieren und auf diese Gruppen die Multiplikatoren anwenden. So trifft eine Route von `/a(bc)?d` sowohl auf `/ad` als auch auf `/abcd` zu. Reichen Ihnen diese Mittel nicht aus, um Ihren Anwendungsfall abzudecken, können Sie Routen auch als reguläre Ausdrücke angeben. In Listing 6.9 sehen Sie ein Beispiel für eine solche Route, die für alle Pfade zuständig ist, die die Zeichenkette `movie` irgendwo im Pfad enthalten.

Abhängigkeit von Routen

Beachten Sie, dass eine Route in Express, je nachdem, an welcher Stelle Sie sie in Ihrer Applikation platzieren, dafür sorgen kann, dass andere Routen nicht mehr ausgeführt werden. Express verwendet immer die erste passende Route, die es finden kann. Sendet diese die Antwort an den Client, wird die Middleware-Kette unterbrochen und Express führt keine weiteren Funktionen für diese Anfrage aus.

```
import express from 'express';

const port = 8080;

const app = express();

app.get(/.*\movie.*$/, (request, response) => {
  response.send('Movie Route');
});

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.9 Reguläre Ausdrücke als Routen (»index.js«)

Die hier definierte Route gilt sowohl für die Adresse *http://localhost:8080/movie*, also nur die Zeichenkette `movie`, als auch für *http://localhost:8080/x/movie/x*, bei der `movie` im Pfad vorkommt, oder *http://localhost:8080/xmoviex*, wo `movie` innerhalb eines Pfadteils enthalten ist.

Formulieren Sie Ihre Routen mit regulären Ausdrücken, haben Sie ein Maximum an Flexibilität. Dies sorgt allerdings häufig dafür, dass die Routen schlechter lesbar werden, was die Fehlersuche erschwert. Sie sollten reguläre Ausdrücke in diesem Fall also sparsam und eher als Ausnahme einsetzen, wenn die normalen Routendefinitionen nicht mehr ausreichen.

6.5.2 Controller

Momentan implementiert Ihr Router noch das komplette MVC-Pattern allein. Das ist nicht erstrebenswert, da die Lesbarkeit bei zunehmendem Funktionsumfang leidet, und so lagern Sie im nächsten Schritt alles bis auf die eigentliche Routendefinition in einen Controller aus. Dieser hat die Aufgabe, die View und das Model zusammenzuführen. Außerdem werden an dieser Stelle üblicherweise die Informationen aus der Anfrage extrahiert, und die Antwort an den Client wird formuliert. Für die Benen-

nung der Routing-Callback-Funktionen hat sich in vielen Web-Frameworks als Standard herauskristallisiert, dass diese Funktionen als Actions bezeichnet werden. Ein Controller kann mehrere dieser Actions-Funktionen enthalten. Listing 6.10 zeigt Ihnen die Implementierung des Controllers für die Ausgabe aller Datensätze.

```
const data = [
  { id: 1, title: 'Iron Man', year: '2008' },
  …
];

export function getAllAction(request, response) {
  response.json(data);
}
```

Listing 6.10 Controller (»movies/controller.js«)

Export und Implementierung sind in diesem Fall miteinander verbunden. Sie können an dieser Stelle allerdings genauso gut alle Exporte am Ende der Datei sammeln. Es gibt hier weder richtig noch falsch, wichtig ist nur, dass Sie konsistent bleiben und immer die gleiche Art des Exports verwenden. Üblicherweise nutzen Sie die Kombination aus Export und Definition in Dateien, die nur wenige Exporte beinhalten. Generell sollten Sie darauf achten, dass eine Datei nicht zu viele Strukturen exportiert, da dies schnell unübersichtlich werden kann und darauf hindeutet, dass die Datei zu viele Strukturen beherbergt.

Eingebunden wird der Controller im Router, wo statt der Routing-Callback-Funktion eine Referenz auf die Methode `getAllAction` des Controllers eingetragen wird. Den Quellcode des angepassten Routers finden Sie in Listing 6.11.

```
import { Router } from 'express';
import { getAllAction } from './controller.js';

const router = Router();

router.get('/', getAllAction);

export default router;
```

Listing 6.11 Einbinden der Controller-Action in den Router (»movies/index.js«)

Bei der Erweiterung des Routers importieren Sie die `getAllAction` und verwenden diese in der `get`-Methode des Routers. Mit diesem Umbau haben Sie das Routing Ihres Moduls sowie den Umgang mit Request und Response voneinander getrennt. Model, View und Controller sind allerdings immer noch recht eng miteinander verbunden. Dies gilt es im nächsten Schritt aufzulösen.

6.5.3 Model

Die Daten der Filmdatenbank liegen in Form eines einfachen Arrays aus Objekten vor. Das Model kapselt dieses Array und stellt eine Funktion zum Auslesen der Daten sowie später weitere Methoden zum Modifizieren dieser Datenstruktur zur Verfügung. Noch arbeiten Sie nicht mit einer Datenbank oder einem anderen externen System zur Datenhaltung in Ihrer Applikation. Damit der Quellcode der Applikation dennoch etwas realitätsnäher wird, setzen Sie die Schnittstellen des Models mit Promises, und damit asynchron, um.

Promises

Eine Promise ist ein Objekt, das für die Erfüllung einer asynchronen Operation in JavaScript steht. Im Gegensatz zu Callback-Funktionen können Sie mit Promises deutlich besser arbeiten und auch mehrere Operationen an die Erfüllung eines solchen Promise-Objekts binden.

Promises sind ein Sprachfeature von JavaScript und Bestandteil des Standards, sodass sie sowohl client- als auch serverseitig unterstützt werden. Ein Promise-Objekt können Sie entweder mit dem Promise-Konstruktor oder mit `Promise.resolve` beziehungsweise `Promise.reject` erzeugen. In Listing 6.12 sehen Sie ein einfaches Beispiel einer Funktion, die mit Promises arbeitet.

```
function asyncFunction() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Hallo Welt!');
    }, 1000);
  });
}

const promise = asyncFunction();
promise.then((value) => {
  console.log(value); // Ausgabe: Hallo Welt! (nach 1 Sekunde)
});
```

Listing 6.12 Arbeiten mit Promises

Dem Promise-Konstruktor in der Funktion `asyncFunction` übergeben Sie eine Callback-Funktion. Diese hat Zugriff auf die Argumente `resolve` und `reject`. Mit diesen Funktionen können Sie einen Erfolg beziehungsweise einen Fehlschlag einer asynchronen Operation anzeigen, indem Sie die jeweilige Funktion aufrufen. Hierbei können Sie einen beliebigen Wert übergeben, der so etwas wie der Rückgabewert der asynchronen Operation ist.

Das Promise-Objekt, das die `asyncFunction` zurückgibt, verfügt über die Methoden `then`, `catch` und `finally`. Jede dieser Methoden akzeptiert eine Callback-Funktion, die

im Erfolgsfall, im Fehlerfall beziehungsweise in beiden Fällen ausgeführt wird und der der Wert übergeben wird, den Sie beim Aufruf von `resolve` oder `reject` übergeben haben. Die `then`-Methode stellt einen Sonderfall dar, da Sie hier noch eine zweite Callback-Funktion für den Fehlerfall angeben können.

Alternativ zu Promises können Sie auch mit dem `async`-Schlüsselwort arbeiten. Kennzeichnen Sie eine Funktion als `async`, gibt diese immer ein Promise-Objekt zurück. Arbeiten Sie mit einem gewöhnlichen Rückgabewert, sorgt die JavaScript-Engine dafür, dass dieser in ein Promise-Objekt verpackt wird. Ein positiver Effekt des `async`-Schlüsselworts ist außerdem, dass Sie in einer solchen Funktion mit dem `await`-Schlüsselwort auf die Auflösung eines Promise-Objekts warten können, ohne dass Sie die `then`-Methode und eine Callback-Funktion benötigen. Für die Fehlerbehandlung können Sie ein `try-catch`-Statement verwenden. Das Promise-Beispiel sieht mit `async-await` wie in Listing 6.13 aus.

```
import { setTimeout } from 'node:timers/promises';

async function asyncFunction() {
  return await setTimeout(1000, 'Hallo Welt!');
}

try {
  const value = await asyncFunction();
  console.log(value); // Ausgabe: Hallo Welt! (nach 1 Sekunde)
} catch (error) {
  console.error(error);
}
```

Listing 6.13 Arbeiten mit »async-await«

Dieses Codebeispiel arbeitet mit der promisebasierten Variante von `setTimeout` aus dem `timers`-Modul von Node.js. Die Funktion gibt Ihnen ein Promise-Objekt zurück, das nach einer bestimmten Anzahl von Millisekunden mit dem angegebenen Wert aufgelöst wird. Das `await` vor dem `setTimeout` ist optional, da die `asyncFunction` durch das vorangestellte `async`-Schlüsselwort auf jeden Fall ein Promise-Objekt zurückgibt.

```
class Model {
  #data = [
    { id: 1, title: 'Iron Man', year: '2008' },
    …
  ];
  async getAll() {
    return this.#data;
  }
```

```
}

export default new Model();
```

Listing 6.14 Umsetzung des Models (»movies/model.js«)

Das Model ist in diesem Fall als JavaScript-Klasse implementiert. Die Daten befinden sich in einem Private Field, das einen Zugriff von außerhalb der Klasse verhindert. Die asynchrone `getAll`-Methode gibt ein Promise-Objekt zurück, das die Daten des Models enthält und direkt aufgelöst wird. Das stellt zunächst ein Problem dar, da über diese Referenz die Informationen verändert werden könnten. Zu einem späteren Zeitpunkt tauschen Sie diese Datenstruktur gegen eine vollwertige Datenbank aus, sodass Sie sich aktuell nicht um dieses Problem kümmern müssen. Nachdem Sie die Grundlage für das Model geschaffen haben, müssen Sie es noch in Ihre Applikation integrieren. Der Controller ist die Stelle, die hierfür infrage kommt. Listing 6.15 zeigt Ihnen die Anpassungen, die Sie an Ihrem Code vornehmen müssen.

```
import model from './model.js';

export async function getAllAction(request, response) {
  const data = await model.getAll();
  response.json(data);
}
```

Listing 6.15 Anpassungen des Controllers zur Integration des Models (»movies/controller.js«)

Implementieren Sie die `getAllAction`-Funktion als `async`-Funktion wie im Beispiel, können Sie innerhalb dieser Funktion das `await`-Schlüsselwort nutzen, um auf die Auflösung der Promise zu warten. Das Ganze geschieht asynchron, also nicht blockierend, der Quellcode wird damit deutlich leserlicher, als würden Sie mit Callback-Funktionen arbeiten. Express unterstützt diese Art der Asynchronität direkt und ohne zusätzliche Konfiguration.

6.5.4 View

Die letzte Komponente Ihrer MVC-Applikation mit Express.js ist die View. Dieser Teil ist für die Darstellung verantwortlich. Die Ausgabe eines JavaScript-Objekts ist keine sonderlich ansprechende Darstellungsform für menschliche Benutzer. Daher sollten Sie an dieser Stelle besser auf HTML als Ausgabeformat zurückgreifen. Für die Anzeige haben Sie mehrere Möglichkeiten. Sie können beispielsweise den HTML-Code in einer separaten HTML-Datei speichern, diese mit JavaScript einlesen und die entsprechenden Stellen ersetzen. Diese Ersetzung können Sie entweder mit der `replace`-Me-

thode der HTML-Zeichenkette direkt vornehmen, oder Sie nutzen einen HTML-Parser wie Cheerio. Eine einfachere Variante ist der Einsatz von JavaScript-Template-Strings, wie Sie im Folgenden sehen werden. Eine weitere Möglichkeit ist die Verwendung einer Template-Engine. Diesem Thema widmet sich das nächste Kapitel. Listing 6.16 enthält die View, die die Filmliste anzeigt.

```
export default function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
</head>
<body>
  <table>
    <thead><tr><th>Id</th><th>Title</th></tr></thead>
    <tbody>
      ${movies
        .map((movie) => `<tr><td>${movie.id}</td><td>${movie.title}</td>
</tr>`)
        .join('')}
    </tbody>
  </table>
</body>
</html>
  `;
};
```

Listing 6.16 View zur Anzeige der Filmliste (»movies/view.js«)

Die View besteht in diesem Fall aus einer `render`-Funktion, die die anzuzeigenden Daten als Argument erhält. Innerhalb der HTML-Struktur wird jeder Eintrag mithilfe der `map`-Methode in eine Tabellenzeile transformiert. Diese Struktur verbinden Sie mit der `join`-Methode zu einer Zeichenkette, die dann innerhalb des Template-Strings an der korrekten Stelle ausgegeben wird. Der Controller hat die Aufgabe, Model und View zusammenzubringen. Listing 6.17 enthält den angepassten Code des Controllers.

```
import render from './view.js';
import model from './model.js';

export async function getAllAction(request, response) {
  const data = await model.getAll();
```

```
  const body = render(data);
  response.send(body);
}
```

Listing 6.17 View-Integration in den Controller (»movies/controller.js«)

Die `listAction`-Funktion lädt im ersten Schritt die Liste der Filme. Diese wird im zweiten Schritt an die View übergeben. Im letzten Schritt senden Sie die HTML-Struktur an den Client. Ist der Watch-Modus Ihrer Applikation über das Kommando `npm run start:dev` aktiviert, startet der Server beim Speichern automatisch neu, ansonsten müssen Sie an dieser Stelle Ihre Applikation manuell neu starten und erhalten dann eine Ausgabe wie die in Abbildung 6.4.

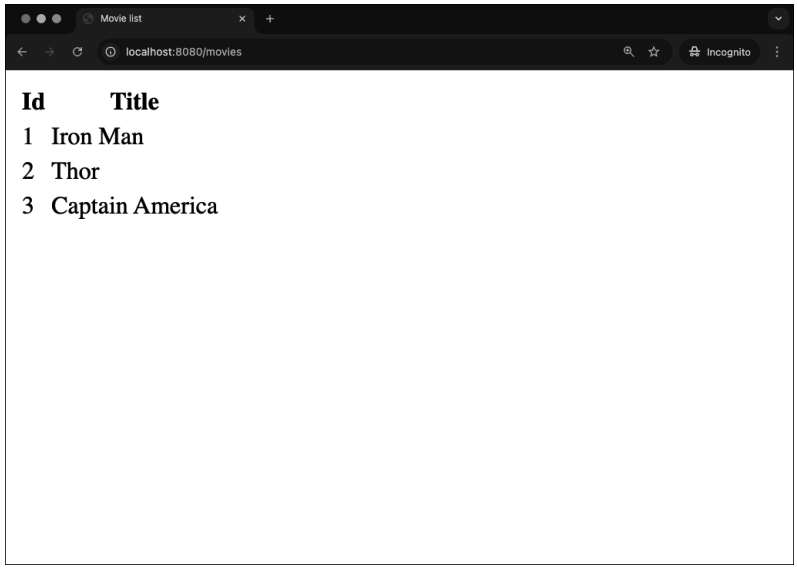


Abbildung 6.4 Anzeige der Filmliste nach dem View-Umbau

Achten Sie bei der Implementierung Ihrer Applikation darauf, dass Sie so wenig Logik wie möglich in die View verlagern. Eine Iteration, wie Sie sie hier umgesetzt haben, oder einfache Bedingungen, um Teile aus- oder einzublenden, sind in Ordnung. Umfangreichere Logik sollten Sie auf jeden Fall in Ihre Models auslagern.

6.6 Middleware

Ein wesentliches Designmerkmal von Express ist das Middleware-Konzept. Es macht das Framework zu einem sehr flexiblen Werkzeug. Einfach ausgedrückt, bezeichnet eine Middleware hier eine Funktion, die zwischen der eingehenden Anfrage und der ausgehenden Antwort steht. Sie können beliebig viele dieser Funktionen hinterein-

anderschalten. Es existieren bereits zahlreiche vordefinierte Middleware-Funktionen, die Sie in Ihre Applikation einbinden können und die für Sie bestimmte Aufgaben erledigen. Existiert noch keine Lösung für Ihr Problem, können Sie eine solche Funktion auch selbst schreiben.

6.6.1 Eigene Middleware

Eine Middleware-Funktion können Sie entweder dazu verwenden, Informationen aus der eingehenden Anfrage zu extrahieren und sie zu speichern, wie beispielsweise bei einem Logger. Sie können aber auch die Response basierend auf Informationen aus dem Request anreichern. Eine Middleware-Funktion muss eine bestimmte Signatur aufweisen, damit sie die Kette der Middleware-Funktionen nicht unterbricht. Angenommen, Sie wollten für Ihre Applikation einen Logger definieren, der bei jeder Anfrage den jeweiligen URL-Pfad auf die Konsole schreibt. Hierfür erzeugen Sie im Wurzelverzeichnis eine neue Datei mit dem Namen *logger.js*. Diese exportiert die gleichnamige Middleware-Funktion, deren Code Sie in Listing 6.18 sehen können.

```
export default function logger(request, response, next) {
  console.log(request.url);
  next();
}
```

Listing 6.18 Implementierung einer einfachen Logger-Middleware (»logger.js«)

Wie Sie sehen, ähnelt die Signatur einer Middleware-Funktion der einer gewöhnlichen Routing-Funktion. Eine Middleware-Funktion weist immer die drei Parameter Request, Response und eine Callback-Funktion auf. Über Request und Response haben Sie, wie in den Controller-Actions auch, Zugriff auf Anfrage und Antwort. Das `Request`-Objekt nutzen Sie im Beispiel, um die angefragte URL auf der Konsole auszugeben. Die übergebene Callback-Funktion, die laut Konvention den Namen `next` trägt, sorgt bei ihrem Aufruf dafür, dass die nächste Middleware-Funktion in der Kette aufgerufen wird. Führen Sie diese Callback-Funktion nicht aus, wird in der Regel keine Antwort an den Client gesendet, und dieser erhält einen Time-out-Fehler. Damit die Middleware funktionieren kann, müssen Sie sie mit der `use`-Methode des `app`-Objekts in Ihrer Applikation registrieren. Diese Methode akzeptiert optional als erstes Argument einen URL-Pfad, auf den die Middleware angewendet werden soll. Übergeben Sie lediglich eine Callback-Funktion, wird die Middleware bei jeder Anfrage ausgeführt. Die Reihenfolge, in der Sie Ihre Funktionen registrieren, ist wichtig. Registrieren Sie zuerst eine reguläre Routing-Methode, die nicht die `next`-Callback-Funktion aufruft, wird Ihre Middleware-Komponente, die danach registriert wird, nicht ausgeführt. Soll Ihre Middleware Berechnungen oder Modifikationen durchführen, die Sie im späteren Verlauf der Aufrufkette der Funktionen benötigen, können Sie diese In-

formationen im `Request`- oder, besser noch, im `Response`-Objekt in einer Eigenschaft zwischenspeichern. Diese Objekte stehen als Referenz allen Funktionen zur Verfügung und können deshalb auch für den Transport von Informationen verwendet werden.

Doch zurück zur Logger-Middleware, die nur lesend auf das Response-Objekt zugreifen soll. Die Middleware soll für jede Anfrage ausgeführt werden, also müssen Sie sie in der Einstiegsdatei Ihrer Applikation zu einem frühen Zeitpunkt, am besten direkt nach der Erstellung des `app`-Objekts, einbinden. Den dafür erforderlichen Code finden Sie in Listing 6.19.

```
import express from 'express';
import moviesRouter from './movies/index.js';
import logger from './logger.js';

const port = 8080;

const app = express();

app.use(logger);

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.19 Integration der Logger-Middleware (»index.js«)

Neben der Positionierung der Middleware ist hier auch wichtig, dass Sie der `use`-Methode die Referenz auf die Funktion übergeben und sie nicht versehentlich selbst aufrufen.

Gerade für Standardaufgaben wie beispielsweise die Erstellung eines Accesslogs gibt es vorgefertigte Komponenten, die Sie nur noch installieren und einbinden müssen. Eine Liste von Middleware-Komponenten finden Sie unter *http://expressjs.com/en/resources/middleware.html*.

6.6.2 Morgan – Logging-Middleware für Express

Beim Protokollieren von eingehenden Anfragen handelt es sich um ein Standardproblem, für das es etablierte Lösungen gibt. Morgan ist eine der populärsten Middle-

ware-Komponenten und nimmt Ihnen diese Arbeit ab. Mit dem Befehl `npm install morgan` installieren Sie das Paket in Ihrer Applikation. Morgan besteht im Kern aus einer Funktion, die ein Format für die Log-Einträge und zusätzliche Optionen akzeptiert. Das bedeutet, dass Sie Ihre bisherige Logger-Implementierung im nächsten Schritt durch Morgan ersetzen können, wie in Listing 6.20 dargestellt.

```
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';

const port = 8080;

const app = express();

app.use(morgan('common', { immediate: true }));

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.20 Die Morgan-Middleware (»index.js«)

Für das Format steht Ihnen eine Reihe vordefinierter Formate wie `combined`, `short` oder `dev` zur Verfügung. Das im Beispiel verwendete Format `common` gleicht dem Format, das beim Apache-Webserver im Accesslog zum Einsatz kommt. Alternativ zu den vordefinierten Formaten können Sie auch selbst ein Format definieren. Wollen Sie beispielsweise nur die Informationen Datum, HTTP-Methode, URL und den Statuscode festhalten, sieht das entsprechende Format folgendermaßen aus: `':date :method :url :status'`. Die dritte Variante zur Definition eines Formats ist die mit einer Funktion statt einer Zeichenkette. Jede dieser drei Varianten übergeben Sie der `morgan`-Funktion als erstes Argument. Mit dem `Option`-Objekt, das Sie als zweites Argument an Morgan übergeben, können Sie das Verhalten des Loggers beeinflussen. Mit dem im Beispiel verwendeten Schlüssel `immediate` legen Sie fest, ob der Log-Eintrag sofort oder erst bei der Antwort an den Client geschrieben werden soll. Mit der Eigenschaft `stream` können Sie einen Writeable Stream angeben, in den die Log-Einträge geschrieben werden. Damit können Sie die Log-Einträge nicht nur auf die Konsole, sondern auch in eine Datei schreiben.

```
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';
import { createWriteStream } from 'node:fs';

const port = 8080;

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(
  morgan('common', {
    immediate: true,
    stream: accessLogStream,
  })
);

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.21 Log-Einträge in eine Datei schreiben (»index.js«)

In Listing 6.21 sehen Sie, wie Sie mit der `createWriteStream`-Funktion aus dem `fs`-Modul einen Datenstrom öffnen können, der in die Datei *access.log* schreibt, und ihn über die `stream`-Eigenschaft des Konfigurationsobjekts an Morgan weiterreichen. Das führt dazu, dass bei jedem Zugriff ein neuer Eintrag in die Datei geschrieben wird. Das Konfigurationsobjekt der `createWriteStream`-Funktion mit der Eigenschaft `flags` und dem Wert `a` sorgt dafür, dass neue Einträge angehängt und nicht der bestehende Inhalt überschrieben wird; außerdem wird die Datei angelegt, sollte sie noch nicht existieren.

Mit der Eigenschaft `skip` können Sie eine Funktion angeben, die auf den Request und die Response zugreifen kann und deren Rückgabewert festlegt, ob ein Eintrag in das Log geschrieben wird oder nicht. Gibt die `skip`-Funktion `false` zurück, wird der Eintrag nicht geschrieben.

6.6.3 Statische Inhalte ausliefern

Webapplikationen, die Sie mit Express.js umsetzen, bestehen in der Regel nicht nur aus dynamischen Inhalten, sondern benötigen auch statische Dateien. So müssen HTML-, JavaScript-, CSS- und Bilddateien geladen werden. Sie können zwar mit dem `fs`-Modul die Inhalte dieser Dateien auslesen und sie als Response an den Client senden, viel einfacher wird diese Aufgabe jedoch, wenn Sie die `static`-Middleware einsetzen. Im Gegensatz zu Morgan ist sie ein Teil von Express, sodass Sie keine zusätzlichen Pakete installieren müssen. Werfen Sie einen Blick auf die Filmliste in Ihrem Browser, bietet sich Ihnen kein besonders ansehnlicher Anblick. Das lässt sich mit ein wenig CSS jedoch ändern. Der CSS-Code in Listing 6.22 sorgt dafür, dass die Tabelle etwas ansprechender aussieht.

```
table {
  border-spacing: 0;
}
th {
  background-color: black;
  font-weight: bold;
  color: lightgrey;
  text-align: left;
  padding: 5px;
}
td {
  border-top: 1px solid darkgrey;
  padding: 5px;
}
tbody tr:hover {
  background-color: lightgrey;
}
```

Listing 6.22 Styling der Liste (»public/style.css«)

Um das Styling auf die Filmliste anzuwenden, passen Sie Ihre Applikation an zwei Stellen an. Zunächst müssen Sie dafür sorgen, dass der Server die CSS-Datei ausliefert und der Browser diese auch lädt. Die Auslieferung erfolgt über die bereits erwähnte `static`-Middleware. Diese binden Sie, wie auch schon den Logger oder den Router, über die `use`-Methode des `app`-Objekts ein. In Listing 6.23 sehen Sie die angepasste Einstiegsdatei Ihrer Applikation.

```
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';
import { createWriteStream } from 'node:fs';
```

```
import { dirname } from 'node:path';
import { fileURLToPath } from 'node:url';

const port = 8080;

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(
  morgan('common', {
    immediate: true,
    stream: accessLogStream,
  })
);

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});
```

Listing 6.23 Einbindung der »static«-Middleware (»index.js«)

Beim Aufruf der `static`-Middleware übergeben Sie den Namen des Verzeichnisses, in dem die statischen Inhalte liegen. Den Pfad zu diesem Verzeichnis können Sie über die Kombination von `import.meta.url`, `fileURLToPath` und `dirname` von Node.js berechnen lassen und ihn relativ zur Einstiegsdatei generieren. Im Fall der Movie-Liste ist dies das *public*-Verzeichnis im Wurzelverzeichnis Ihrer Applikation. Im nächsten Schritt können Sie nun im HTML-Code der Liste die CSS-Datei über ein `link`-Tag referenzieren.

```
export default function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
  <link rel="stylesheet" href="style.css" />
```

```
</head>
<body>
  <table>…</table>
</body>
</html>
  `;
}
```

Listing 6.24 Einbetten der CSS-Datei in den HTML-Code (»movies/view.js«)

Verändern Sie den Quellcode der Listenansicht entsprechend Listing 6.24 und starten Ihre Applikation neu, erhalten Sie eine Ansicht wie die in Abbildung 6.5.

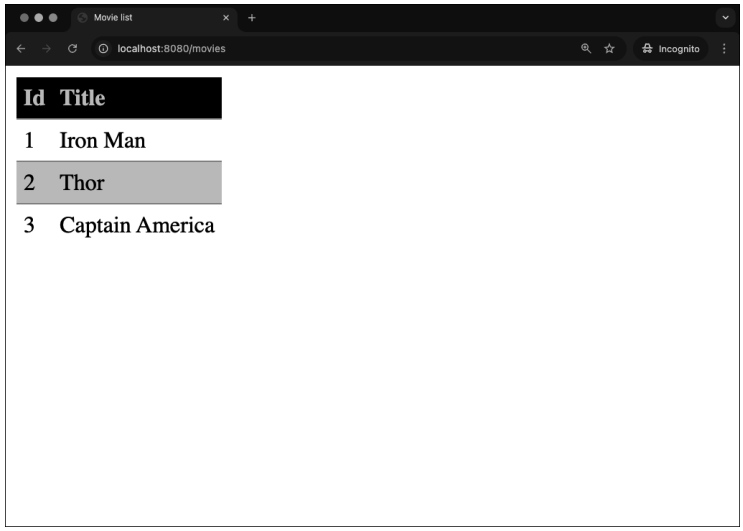


Abbildung 6.5 Listenansicht mit CSS

6.7 Dynamische Routen – Löschen von Datensätzen

Bis jetzt beschränkt sich die Funktionalität Ihrer Applikation auf die Anzeige der Daten. Im nächsten Schritt bieten Sie Ihren Benutzern die Möglichkeit, Daten zu löschen. Hierfür fügen Sie zunächst im Frontend einen Link pro Datensatz ein. Dieser führt zu einer Route auf dem Server, die sich um das Löschen des Datensatzes kümmert. Nach erfolgter Löschung leiten Sie den Benutzer auf die Liste weiter, sodass die Ansicht aktualisiert wird.

```
export default function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <table>
    <thead><tr><th>Id</th><th>Title</th><th></th></tr></thead>
    <tbody>
      ${movies
        .map(
          (movie) => `
          <tr>
            <td>${movie.id}</td>
            <td>${movie.title}</td>
            <td><a href="/movies/delete/${movie.id}">löschen</a></td>
          </tr>`
        )
        .join('')}
    </tbody>
  </table>
</body>
</html>
  `;
}
```

Listing 6.25 Erweiterung des Templates um einen Löschen-Link (»movies/view.js«)

Die Routing-Funktionen von Express unterstützen alle verfügbaren HTTP-Methoden, also auch beispielsweise die `DELETE`-Methode. Dies sollte im Normalfall die bevorzugte Variante zum Löschen von Informationen sein, da jede HTTP-Methode eine bestimmte Bedeutung hat. In klassischen rein HTML-basierten Webapplikationen haben Sie allerdings keine Möglichkeit, über einen Link eine andere Methode als `GET` aufzurufen. Aus diesem Grund müssen Sie Ihrem Router für das Löschen von Datensätzen eine entsprechende `get`-Route hinzufügen. In Kapitel 10, wenn Sie einen REST-Server implementieren, sieht die Situation anders aus. Werfen Sie einen Blick auf den ausgelieferten Quellcode, werden Sie feststellen, dass jeder Link in der Tabelle unterschiedlich aussieht, da Sie die ID des zu löschenden Datensatzes angeben. Diese Tatsache führt zu einer Besonderheit bei der Angabe des Pfads im Router-Link. In diesem Fall enthält der Link einen variablen Anteil, der für die ID steht und auf den Sie über das `Request`-Objekt zugreifen können. Eine solche Variable kennzeichnen Sie in der Angabe des Pfads der Routing-Funktion durch einen Doppelpunkt gefolgt vom Namen der Variablen. Listing 6.26 können Sie die Modifikation am Router entnehmen.

```
import { Router } from 'express';
import { getAllAction, removeAction } from './controller.js';

const router = Router();

router.get('/', getAllAction);
router.get('/delete/:id', removeAction);

export default router;
```

Listing 6.26 Erweiterung des Routers um eine Löschen-Route (»movies/index.js«)

In der `removeAction` des Controllers greifen Sie über das `params`-Objekt des Requests auf die Variablen der Route zu. In diesem Fall erreichen Sie die vom Benutzer übergebene ID über `request.params.id`. Hier sollten Sie beachten, dass Express die übermittelten Werte als Zeichenketten interpretiert. Um im weiteren Verlauf mit der ID weiterarbeiten zu können, sollten Sie sie mit der `parseInt`-Funktion in eine Zahl umwandeln. Listing 6.27 enthält den Quellcode des Controllers.

```
import render from './view.js';
import model from './model.js';

export async function getAllAction(request, response) {
  const data = await model.getAll();
  const body = render(data);
  response.send(body);
}

export async function removeAction(request, response) {
  const id = parseInt(request.params.id, 10);
  await model.remove(id);
  response.redirect(request.baseUrl);
}
```

Listing 6.27 »removeAction« des Controllers (»movies/controller.js«)

Nachdem Sie die ID umgewandelt haben, können Sie die `remove`-Methode des Models aufrufen, das dafür sorgt, dass der entsprechende Datensatz in der Datenquelle gelöscht wird. Diese Operation ist ebenfalls asynchron und arbeitet mit Promises, daher setzen Sie auch hier `async`/`await` ein. Nachdem die Operation beendet ist, leiten Sie den Benutzer mithilfe der `response.redirect`-Methode auf die Liste weiter. Hier sehen Sie eine weitere Besonderheit des modularen Aufbaus einer Applikation: Statt in der `redirect`-Methode direkt auf `"/movies/"` weiterzuleiten, kommt in diesem Fall die

`baseUrl`-Eigenschaft des `Request`-Objekts zum Einsatz. Der Grund hierfür ist, dass das Movie-Modul selbst die Basis-URL nicht kennt, für die es zuständig ist. Um hier eine enge Kopplung zwischen der einbindenden Stelle und dem Modul zu vermeiden, verwenden Sie die `baseUrl`-Eigenschaft, die die Information enthält. So ist es Ihnen zu einem späteren Zeitpunkt möglich, in der *index.js*-Datei die `BaseUrl` zu ändern, ohne dass Sie das Modul anpassen müssen.

Die `remove`-Methode des Models schließlich filtert den zu löschenden Datensatz aus der Datenquelle heraus und überschreibt die Datenquelle mit den aktualisierten Informationen. In Listing 6.28 finden Sie die entsprechende Implementierung.

```
class Model {
  #data = […];
  async getAll() {
    return this.#data;
  }

  async remove(id) {
    this.#data = this.#data.filter((item) => item.id !== id);
  }
}

export default new Model();
```

Listing 6.28 Anpassung am Model (»movie/model.js«)

Da die Daten in einem Private Field liegen, können Sie die Daten von außerhalb nicht manipulieren und sind auf Schnittstellen wie die `remove`-Methode angewiesen. Mit solchen Methoden können Sie sicherstellen, dass die Daten konsistent bleiben.

6.8 Anlegen und Bearbeiten von Datensätzen – Zugriff auf den Request-Body

Über Variablen in der URL können Sie nur wenige Daten übermitteln, gerade bei Formularen ist das keine praktikable Lösung. In der Regel werden diese auch mit einem HTTP-`POST` versendet. Diesen Umstand machen Sie sich bei der Erweiterung Ihrer Applikation zunutze. Im nächsten Schritt implementieren Sie eine Möglichkeit, neue Datensätze anzulegen und bestehende zu editieren. Diese Anpassung starten Sie wieder im Frontend Ihrer Applikation, indem Sie die Listenanzeige um einen Link erweitern, der es Ihren Benutzern ermöglicht, neue Datensätze anzulegen. Außerdem fügen Sie in der Tabelle pro Eintrag einen Link zum Editieren der Datensätze ein. Listing 6.29 enthält den Quellcode der View.

```
export default function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <table>
    <thead><tr><th>Id</th><th>Title</th><th></th><th></th></tr></thead>
    <tbody>
      ${movies
        .map(
          (movie) => `
          <tr>
            <td>${movie.id}</td>
            <td>${movie.title}</td>
            <td><a href="/movies/delete/${movie.id}">löschen</a></td>
            <td><a href="/movies/form/${movie.id}">bearbeiten</a></td>
          </tr>`
        )
        .join('')}
    </tbody>
  </table>
  <a href="/movies/form">neu</a>
</body>
</html>
  `;
}
```

Listing 6.29 Anpassung der View (»movies/view.js«)

Beide Links der View zeigen auf eine ähnliche Route, einmal auf `/movie/form` und auf `/movie/form/:id`. In Express haben Sie die Möglichkeit, optionale Parameter zu definieren. Dazu fassen Sie den Parameter in geschweifte Klammern und müssen damit nicht zwei separate Routen definieren. Die aktualisierte Routerkonfiguration sehen Sie in Listing 6.30.

```
import { Router } from 'express';
import { getAllAction, removeAction, formAction } from './controller.js';

const router = Router();
```

```
router.get('/', getAllAction);
router.get('/delete/:id', removeAction);
router.get('/form{/:id}', formAction);

export default router;
```

Listing 6.30 Anpassung der Routerkonfiguration (»movie/index.js«)

Der Controller ist dafür verantwortlich, auf Basis der Informationen aus dem Request gegebenenfalls Informationen aus dem Model auszulesen und die View zu rendern. Der entscheidende Punkt ist hier die Unterscheidung, ob der Benutzer eine ID übergeben hat, um einen Datensatz zu editieren, oder ob ein neuer Datensatz angelegt werden soll.

```
import render from './view.js';
import model from './model.js';
import renderForm from './form.js';

export async function getAllAction(request, response) {…}

export async function removeAction(request, response) {…}

export async function formAction(request, response) {
  let movie = { id: '', title: '', year: '' };

  if (request.params.id) {
    const id = parseInt(request.params.id, 10);
    movie = await model.getById(id);
  }

  const body = renderForm(movie);
  response.send(body);
}
```

Listing 6.31 »formAction« im Controller (»movie/controller.js«)

Listing 6.31 enthält die Implementierung der neuen Controller-Funktion mit dem Namen `formAction`. Sie definiert zunächst einen leeren Datensatz, den Sie zum Rendern des Formulars für einen neuen Datensatz übergeben. Anschließend überprüfen Sie, ob bei der Anfrage eine ID übergeben wurde; ist dies der Fall, nutzen Sie die `getById`-Methode des Models, um den passenden Datensatz zu laden und ihn als neuen Wert der `movie`-Variablen zu übergeben. Im letzten Schritt erzeugen Sie mithilfe der `renderForm` Funktion, also der View, eine HTML-Zeichenkette, die Sie zum Client senden. Den

Code der View, die Sie in der Datei *form.js* im *movies*-Verzeichnis speichern, finden Sie in Listing 6.32.

```
export default function render(movie) {
  return `
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Movie list</title>
    <link rel="stylesheet" href=" /style.css" />
  </head>
  <body>
    <form action="/movies/save" method="post">
      <input type="hidden" id="id" name="id" value="${movie.id}" />
      <div>
        <label for="title">Titel:</label>
        <input type="text" id="title" name="title" value="${movie.title}" />
      </div>
      <div>
        <label for="id">Jahr:</label>
        <input type="text" id="year" name="year" value="${movie.year}" />
      </div>
      <div>
        <button type="submit">speichern</button>
      </div>
    </form>
  </body>
  </html>
  `;
}
```

Listing 6.32 Formular-View (»movies/form.js«)

Das Formular sorgt mit den Attributen `action="/movies/save"` und `method="post"` dafür, dass die vom Benutzer eingegebenen Daten der HTTP-`POST`-Methode an den Server gesendet werden. Anhand des Vorhandenseins eines ID-Werts kann der Server unterscheiden, ob es sich bei der Anfrage um eine erstellende oder eine modifizierende Operation handelt. Die Werte, die beim Bearbeiten übergeben werden, werden über Template-Ersetzungen als `value`-Attribute in die einzelnen Formularfelder eingefügt. Damit Sie einen Datensatz editieren können, müssen Sie im Model noch die `getById`-Methode implementieren, mit deren Hilfe Sie den Datensatz aus der Datenquelle laden. In dieser Applikation reicht ein Aufruf der `find`-Methode des `data`-Arrays

aus, um den Datensatz anhand seiner ID zu laden. Listing 6.33 zeigt Ihnen den entsprechenden Quellcode.

```
class Model {
  #data = […];

  async getAll() {…}

  async getById(id) {
    return this.#data.find((item) => item.id === id);
  }

  async remove(id) {…}
}

export default new Model();
```

Listing 6.33 Erweiterung des Models um die »get«-Methode (»movies/model.js«)

Laden Sie nun Ihre Applikation neu, erreichen Sie das Formular entweder über den Neu- oder die Bearbeiten-Links in der Liste. Das Ergebnis sehen Sie in Abbildung 6.6.

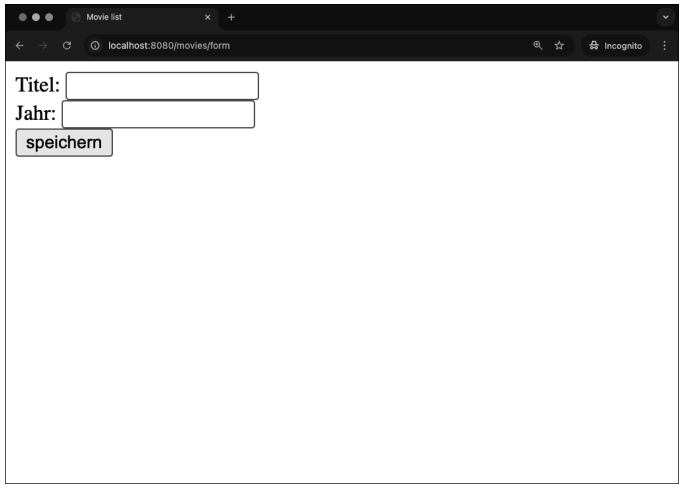


Abbildung 6.6 Formular zum Anlegen von Datensätzen

6.8.1 Umgang mit Formulareingaben – die Body-Parser-Middleware

Damit Sie die Eingaben Ihrer Benutzer speichern können, müssen Sie auf die Formulardaten zugreifen können. Die komfortabelste Variante ist die Nutzung der Body-Parser-Middleware. Das Body-Parser-Paket hat eine bewegte Vergangenheit hinter sich. Ursprünglich war die Middleware, ähnlich wie die `static`-Middleware, fester Be-

standteil von Express. Mit Version 4 wurde sie allerdings in ein eigenes Paket ausgelagert und mit Version 4.16 wieder zurück in den Kern von Express eingegliedert. Nutzen Sie also eine aktuelle Version von Express, müssen Sie für die Verarbeitung von Request-Bodys keine zusätzlichen Pakete installieren.

Der Body-Parser stellt Ihnen die zwei Funktionen `json` und `urlencoded` für JSON- beziehungsweise URL-encodierte Anfragen zur Verfügung, auf die Sie direkt über das `express`-Objekt zugreifen können. Die Integration der Middleware in die *index.js*-Datei Ihrer Applikation finden Sie in Listing 6.34.

```
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';
import { createWriteStream } from 'node:fs';
import { dirname } from 'node:path';
import { fileURLToPath } from 'node:url';

const port = 8080;

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(morgan('common', {…}));

app.use(express.urlencoded({ extended: false }));

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

app.listen(port, () => {…});
```

Listing 6.34 Einbindung der Body-Parser-Middleware (»index.js«)

Die Body-Parser-Middleware unterstützt, wie schon erwähnt, verschiedene Parser. Falls Sie wie im Beispiel ein gewöhnliches HTML-Formular verarbeiten, nutzen Sie den `urlencoded`-Parser. Sollte Ihr Frontend stattdessen die Informationen im JSON-Format senden, kommt der JSON-Parser zum Einsatz. Weitere Parser sind der `raw`-Parser, der den Body als Buffer parst, und der `text`-Parser, der den Body der Anfrage als Text interpretiert. Diese beiden sind allerdings erst seit der Express-Version 4.17 verfügbar. Die verschiedenen Parser können Sie auch parallel verwenden, also beispielsweise sowohl den `urlencoded`- als auch den JSON-Parser gleichzeitig.

Das Verhalten des `urlencoded`-Parsers können Sie durch zusätzliche Optionen steuern. Im Codebeispiel nutzen Sie die `extended`-Eigenschaft des Optionsobjekts, um durch den Wert `false` die `querystring`-Bibliothek zum Parsen zu nutzen. Setzen Sie den Wert auf `true`, nutzt Express die `qs`-Bibliothek und kann damit besser mit komplexen Objekten und Arrays umgehen. Nach der Einbindung erweitern Sie den Router Ihres Movie-Moduls, damit der URL-Pfad */movies/save* unterstützt wird.

```
import { Router } from 'express';
import {
  getAllAction,
  removeAction,
  formAction,
  saveAction,
} from './controller.js';

const router = Router();

router.get('/', getAllAction);
router.get('/delete/:id', removeAction);
router.get('/form{/:id}', formAction);
router.post('/save', saveAction);

export default router;
```

Listing 6.35 Erweiterung des Routers um die Save-Route (»movies/index.js«)

Wie Sie in Listing 6.35 sehen, folgt die Erweiterung dem bisherigen Schema. Die `saveAction` des Controllers reicht die Informationen an das Model weiter. Zur besseren Kontrolle extrahiert der Controller die benötigten Eigenschaften aus der `request.body`-Eigenschaft, die vom Body-Parser zur Verfügung gestellt wird. Diese Eigenschaft enthält sämtliche Daten des Formulars als Objekteigenschaften.

Eine separate View wird im Controller nicht benötigt, da die `saveAction` auf die Liste weiterleitet. Auch hier kommt, wie beim Löschen von Datensätzen, die `baseUrl`-Eigenschaft des `Request`-Objekts zum Einsatz. Listing 6.36 zeigt Ihnen den angepassten Controller.

```
import render from './view.js';
import model from './model.js';
import renderForm from './form.js';

export async function getAllAction(request, response) {…}
export async function removeAction(request, response) {…}
export async function formAction(request, response) {…}
```

```
export async function saveAction(request, response) {
  const { id, title, year } = request.body;
  const movie = { id: parseInt(id, 10), title, year };

  await model.save(movie);

  response.redirect(request.baseUrl);
}
```

Listing 6.36 Die »saveAction« des Controllers (»movies/controller.js«)

In der `saveAction` des Controllers extrahieren Sie die Eigenschaften `id`, `title` und `year` aus dem `request.body`-Objekt. So stellen Sie sicher, dass nur die erlaubten Eigenschaften an das Model weitergegeben werden. Mit diesen Werten erzeugen Sie ein neues `movie`-Objekt und reichen es ans Model zum Speichern weiter. Nachdem der Speichervorgang abgeschlossen ist, leiten Sie den Client mittels `response.redirect` zur Listenansicht weiter, die hierdurch automatisch aktualisiert wird.

Der Controller weiß nicht, ob es sich bei der aktuellen Operation um die Neuanlage oder eine Aktualisierung eines Datensatzes handelt. Diese Entscheidung bleibt dem Model überlassen. Aus diesem Grund ist der Model-Code an dieser Stelle auch etwas umfangreicher, wie Sie in Listing 6.37 sehen.

```
class Model {
  #data = [
    { id: 1, title: 'Iron Man', year: '2008' },
    …
  ];

  #getNextId() {
    return Math.max(...this.#data.map((item) => item.id)) + 1;
  }

  async getAll() {…}
  async getById(id) {…}
  async remove(id) {…}

  async save(movie) {
    if (movie.id) {
      const index = this.#data.findIndex((item) => item.id === movie.id);
      this.#data[index] = movie;
    } else {
      movie.id = this.#getNextId();
      this.#data.push(movie);
```

```
    }
  }
}

export default new Model();
```

Listing 6.37 Implementierung der Speichern-Methode im Model (»movies/model.js«)

In der `save`-Funktion des Models wird anhand der `id`-Eigenschaft der übermittelten Daten entschieden, ob es sich um einen neuen oder einen bestehenden Datensatz handelt. Bei neuen Datensätzen ist das `hidden`-Input-Feld des Formulars nicht gefüllt beziehungsweise hat eine leere Zeichenkette als Wert. Beim Erzeugen eines neuen Datensatzes ermittelt die private `#getNextId`-Methode die nächste freie ID in der Datenquelle. Im Normalfall wird diese Aufgabe von der Datenbank übernommen, in unserem Fall wird die höchste ID der Daten im Array gesucht und um eins erhöht. Anschließend wird die neue ID dem Datensatz zugewiesen und die Information in das Datenarray gepusht. Bei der Aktualisierung der Daten suchen Sie den Index des betroffenen Datensatzes im Daten-Array und überschreiben diesen Eintrag mit den Daten, die Sie über die Anfrage vom Client erhalten haben.

Nach einem Neustart des Prozesses können Sie nun Filme in Ihrer Datenbank anzeigen lassen, löschen, bearbeiten und neu anlegen.

6.9 Validierung von Eingaben mit Zod

Die bestehende Codebasis können Sie nun noch beliebig erweitern. Ein häufiger Anwendungsfall ist die Validierung der Eingaben. Dies dient zum einen der Konsistenz der Daten auf dem Server und zum anderen auch der Sicherheit, da Sie damit verschiedene Angriffsvektoren ausschließen können. Für Validierung stehen Ihnen eine ganze Reihe von Bibliotheken zur Verfügung. Eine der populärsten ist Zod. Detaillierte Informationen zu dieser Bibliothek finden Sie auf der Webseite des Projekts unter *https://zod.dev/*, die Installation von Zod erfolgt über Ihren Paketmanager, also beispielsweise NPM, mit dem Kommando `npm install zod`.

Die Grundlage von Zod ist das Validierungsschema, eine Struktur, mit der Sie Regeln für einzelne Felder oder ganze Objektstrukturen definieren. Für die Movie-Struktur erzeugen Sie eine neue Datei mit dem Namen *schema.js* im *movies*-Verzeichnis mit dem Inhalt aus Listing 6.38.

```
import { z } from 'zod/v4';

const schema = z.object({
  id: z
```

```
    .string()
    .optional()
    .transform((val) => (val ? parseInt(val, 10) : undefined)),
  title: z.string().trim().min(1, 'Titel darf nicht leer sein.'),
  year: z.coerce
    .number('Jahr muss eine Zahl sein.')
    .int('Jahr muss eine ganze Zahl sein.')
    .min(1888, 'Jahr muss nach 1887 liegen.')
    .max(
      new Date().getFullYear() + 5,
      `Jahr darf nicht in ferner Zukunft liegen (max. ${
        new Date().getFullYear() + 5
      }).`
    ),
});

export default schema;
```

Listing 6.38 Definition eines Zod-Schemas für die Movie-Struktur (»movies/schema.js«)

Wie Sie sehen, arbeitet Zod mit verschiedenen Methoden. Mit `z.object` definieren Sie eine Objektstruktur, in der Sie verschiedene Eigenschaften festlegen können. Die `id`-Eigenschaft ist beispielsweise ein optionaler String, kann also gesetzt sein oder – bei neuen Datensätzen – auch nicht. Doch falls ein Wert übermittelt wurde, wandeln Sie ihn mit der `transform`-Methode in eine Zahl um. Bei der `title`-Eigenschaft, die ebenfalls eine Zeichenkette ist, werden mit der `trim`-Methode alle Leerzeichen vorne und hinten entfernt, das Ergebnis muss anschließend immer noch mindestens ein Zeichen lang sein, was Sie mit der `min`-Methode definieren. Den Regelmethoden können Sie benutzerdefinierte Fehlermeldungen übergeben. Beim `year`, das Sie ebenfalls als Zeichenkette erhalten, versuchen Sie den Wert mittels `coerce` in eine Zahl umzuwandeln, die eine Ganzzahl (`int`) sein muss, die größer oder gleich 1888 (`min`) und kleiner oder gleich dem aktuellen Jahr plus 5 Jahre sein muss.

Die Validierung findet im Controller statt, da dies die frühestmögliche Stelle ist, an der Sie mit dem `Request`-Objekt arbeiten können. Um die Validierung zu integrieren, konzentrieren Sie sich auf die `saveAction`-Funktion. Die angepasste Version finden Sie in Listing 6.39

```
import render from './view.js';
import model from './model.js';
import renderForm from './form.js';
import schema from './schema.js';

export async function getAllAction(request, response) {…}
```

```
export async function removeAction(request, response) {…}
export async function formAction(request, response) {…}

export async function saveAction(request, response) {
  const parseResult = schema.safeParse(request.body);

  if (!parseResult.success) {
    const errors = parseResult.error.flatten().fieldErrors;
    const body = renderForm(request.body, errors);
    response.send(body);
    return;
  }
  const validatedData = parseResult.data;

  await model.save(validatedData);

  response.redirect(request.baseUrl);
}
```

Listing 6.39 »saveAction« im Controller mit Validierung (»movies/controller.js«)

Mit der `safeParse`-Funktion können Sie eine Objektstruktur gegen die Regeln des Schemas überprüfen lassen. Im Gegensatz zur `parse`-Funktion wirft die `safeParse` keine Fehler, falls die Validierung fehlschlägt. Stattdessen erhalten Sie in der `success`-Eigenschaft des Ergebnisses die Information, ob die Validierung erfolgreich war. Ist dies nicht der Fall, extrahieren Sie die Fehlermeldungen und übergeben sie an die Form-View, um sie zu rendern und zum Client zu senden.

War die Validierung erfolgreich, speichern Sie die Daten mit der `save`-Methode des Models und leiten den Benutzer auf die Listenansicht weiter.

Der letzte Schritt besteht aus einer Anpassung der *form.js*-Datei, um die Fehlermeldungen korrekt anzuzeigen. Den geänderten Quellcode dieser Datei sehen Sie in Listing 6.40.

```
function displayError(field, errors) {
  if (errors[field]) {
    return `<span class="error">${errors[field][0]}</span>`;
  }
  return '';
}

export default function render(movie, errors = {}) {
  return `
  <!DOCTYPE html>
  <html lang="en">
```

```
  <head>
    <meta charset="UTF-8">
    <title>Movie list</title>
    <link rel="stylesheet" href="/style.css" />
  </head>
  <body>
    <form action="/movies/save" method="post">
      <input type="hidden" id="id" name="id" value="${movie.id}" />
      <div>
        <label for="title">Titel:</label>
        <input type="text" id="title" name="title" value="${movie.title}" />
        ${displayError('title', errors)}
      </div>
      <div>
        <label for="year">Jahr:</label>
        <input type="text" id="year" name="year" value="${movie.year}" />
        ${displayError('year', errors)}
      </div>
      <div>
        <button type="submit">speichern</button>
      </div>
    </form>
  </body>
  </html>
  `;
}
```

Listing 6.40 Integration der Fehleranzeige in die Formularstruktur (»movies/form.js«)

In die *form.js*-Datei fügen Sie zunächst eine lokale Hilfsfunktion `displayError` ein. Sie erhält den Namen eines Felds und alle Fehlermeldungen des Formulars. Die Funktion überprüft, ob es für das übergebene Feld einen Fehler gibt, und generiert ein `span`-Element mit der Fehlermeldung.

Die `render`-Funktion für das Formular erweitern Sie um einen optionalen `errors`-Parameter, der die Fehlermeldungen enthält. Nach jedem Feld rufen Sie die `displayError`-Funktion mit dem Feldnamen und dem `errors`-Objekt auf. Das sorgt für die Anzeige des Fehlers. Als letzten Schritt können Sie nun noch Styles für die Fehlermeldungen in der *public/style.css*-Datei definieren.

```
table {…}
th {…}
td {…}
tbody tr:hover {…}
```

```
.error {
  color: red;
  display: block;
  margin-top: 2px 0 5px 0;
}
```

Listing 6.41 Erweiterung des Stylesheets um Fehler-Styles (»public/style.css«)

Die Style-Definition in Listing 6.41 sorgt beispielsweise dafür, dass die Fehlermeldungen in roter Schrift unterhalb der Formularfelder mit etwas Abstand nach oben und unten angezeigt werden. Wie das im Browser aussieht, sehen Sie in Abbildung 6.7.

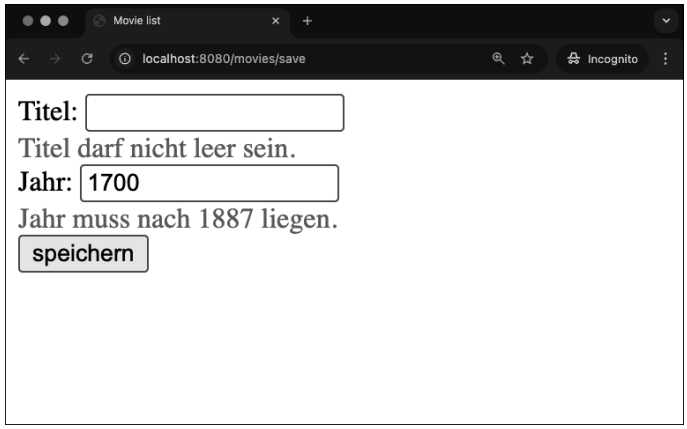


Abbildung 6.7 Fehlermeldungen im Formular

6.10 Express 5

Ein großer Kritikpunkt an Express war, dass die Entwicklung über viele Jahre ins Stocken geraten war. Die letzte Major-Version wurde mit Version 4 im April 2014 veröffentlicht. Bis September 2024, also mehr als zehn Jahre, befand sich das Framework in dieser Version. Lange Zeit wurde die Version 5 zwar angekündigt, es bewegte sich jedoch wenig. Seit September 2024 tut sich hier jedoch wieder deutlich mehr. Mit dem Erscheinen von Version 5 hat sich zwar am Framework nichts gravierend geändert, was auch ein Ziel der Entwickler war. Dieses Release soll jedoch ein klares Zeichen an die Community sein, dass wieder deutlich aktiver an Express gearbeitet wird.

Die Version 5.0.0 musste noch mit expliziter Versionsangabe über das Kommando `npm install express@5` installiert werden. Seit Version 5.1 ist der Entwicklungszweig mit der Version 5 das Latest Release, Sie können es also auch ohne Versionsnummer direkt installieren.

6.10.1 Änderungen in Express 5

Im Vergleich zur Version 4 hat sich bei Express 5 kaum etwas geändert. Es wurden lediglich einige Methoden und Eigenschaften wie beispielsweise die `app.del`-Methode entfernt. Außerdem haben die Entwickler einige Änderungen an existierenden Features vorgenommen. So konnten Sie in Version 4 Routenparameter mit einem `?` als optional kennzeichnen. Ab Version 5 müssen Sie diese in geschweifte Klammern fassen. Alle Änderungen finden Sie zusammengefasst im Migrationsguide unter *https://expressjs.com/en/guide/migrating-5.html*. Zusätzlich zu dieser ausführlichen Beschreibung stellt Ihnen das Express-Team sogenannte Codemods zur Verfügung, mit denen Sie den Code Ihrer Express 4-Applikation automatisch auf Version 5 migrieren können. Hierfür müssen Sie lediglich das Kommando `npx @expressjs/codemod upgrade` ausführen. Anschließend sollten Sie Ihre Applikation ausgiebig testen.

6.10.2 LTS Timeline

Mit dem Wechsel von Version 4 auf 5 ist Express auch auf eine neue Versionierung umgestiegen. Sie soll Entwicklern und Unternehmen mehr Planungssicherheit und Stabilität geben. Der Versionsplan orientiert sich an dem LTS-Plan von Node.js. Express unterstützt dabei insgesamt drei Versionen:

- CURRENT: Diese Version ist speziell für neue Releases gedacht. Eine neue Version bleibt in dieser Version für mindestens drei Monate, wird aber nicht als »latest« Version gekennzeichnet. Das bedeutet, dass Sie sie nicht mit `npm install express` ohne weitere Versionsnummer installieren können.
- ACTIVE: Nach mindestens drei Monaten und wenn das Technical Committee von Express beschließt, dass die Version stabil genug ist, wird das »latest«-Tag vergeben. Dies bleibt für mindestens zwölf Monate bestehen.
- MAINTENANCE: Wechselt eine Version in den ACTIVE-Zustand, wird die vorherige Version für mindestens zwölf Monate in die MAINTENANCE-Phase versetzt.

Für die Versionen 4, 5 und 6 bedeutet diese Versionierungsstrategie Folgendes:

Version	CURRENT	ACTIVE	MAINTENANCE	EOL
4.x			01.04.2025	nicht vor 01.10.2026
5.x	11.09.2024	31.03.2025	nicht vor 01.04.2026	nicht vor 01.04.2027
6.x	nicht vor 01.01.2026			

Tabelle 6.4 Versionen und Phasen von Express

6.11 HTTPS und HTTP/2

Da Express auf dem HTTP-Modul von Node.js aufbaut, ist das Framework recht flexibel, wenn es um den Wechsel zwischen unverschlüsselter und verschlüsselter Kommunikation geht.

6.11.1 HTTPS

Statt HTTP können Sie auch die gesicherte und für produktive Applikationen empfohlene HTTPS-Variante nutzen. Hierfür müssen Sie nichts weiter tun, als das `app`-Objekt, das Sie durch den Aufruf der `express`-Funktion erzeugt haben, an die `createServer`-Methode des `https`-Moduls zu übergeben. Achten Sie bei dieser Umstellung darauf, dass Sie den Aufruf der `listen`-Methode des `app`-Objekts entfernen. An seine Stelle tritt die `listen`-Methode des HTTPS-Server-Objekts.

```
import { createServer } from 'node:https';
import { readFileSync } from 'node:fs';
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';
import { createWriteStream } from 'node:fs';
import { dirname } from 'node:path';
import { fileURLToPath } from 'node:url';

const port = 8080;

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(morgan('common', {…}));

app.use(express.urlencoded({ extended: false }));

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

const options = {
  key: readFileSync('./key.pem'),
  cert: readFileSync('./cert.pem'),
};
```

```
const server = createServer(options, app);

server.listen(port, () => {
  console.log(`Server is listening to https://localhost:${port}`);
});
```

Listing 6.42 Express mit HTTPS (»index.js«)

In Listing 6.42 finden Sie die erforderlichen Anpassungen, um Ihre Filmdatenbank mit HTTPS auszuliefern. Das Beispiel geht davon aus, dass Sie sich wie in Abschnitt 5.4, »Sichere Kommunikation mit HTTPS«, ein selbst signiertes Zertifikat ausgestellt und die Dateien im Wurzelverzeichnis Ihrer Applikation gespeichert haben. Starten Sie nach diesen Anpassungen den Server neu, erreichen Sie Ihre Applikation unter *https://localhost:8080*.

6.11.2 HTTP/2

So schön und einfach wie der Wechsel zwischen HTTP und HTTPS funktioniert die Änderung der Protokollversion von 1 auf 2 leider nicht. Versuchen Sie, von der `createServer`-Methode des `https`-Moduls auf die `createSecureServer`-Methode des `http2`-Moduls zu wechseln, erhalten Sie einen `TypeError`, wenn Sie eine Anfrage an den Server senden, und Ihre Applikation stürzt ab.

Es gibt zwar eine Möglichkeit, dass Sie mit dem `spdy`-Paket eine alternative Implementierung des `http/2`-Protokolls nutzen. Allerdings wird dieses Paket seit mehreren Jahren nicht weiterentwickelt, sodass Sie beim Einsatz in einer Produktivapplikation vorsichtig sein sollten.

Das `spdy`-Modul installieren Sie mit dem Kommando `npm install spdy`. Der Name dieses Moduls ist etwas irreführend, da es nicht nur Spdy, sondern auch HTTP/2 unterstützt. Spdy ist ein Protokoll, das von Google entwickelt wurde und HTTP in der Version 1 ersetzen sollte. Das HTTP/2-Protokoll greift einige Konzepte des Spdy-Protokolls auf.

Das `spdy`-Modul ist API-kompatibel mit dem `http`- und dem `https`-Modul von Node.js und lässt sich aus diesem Grund gut mit Express kombinieren. Zur Integration gehen Sie ähnlich vor wie zuvor schon bei der HTTPS-Integration. Wie Sie in Listing 6.43 sehen können, importieren Sie statt des `https`-Moduls das `spdy`-Paket und rufen statt der `createServer`-Funktion des `https`-Moduls die `spdy.createServer`-Funktion auf.

```
import spdy from 'spdy';
import { readFileSync } from 'node:fs';
import express from 'express';
import moviesRouter from './movies/index.js';
import morgan from 'morgan';
```

```
import { createWriteStream } from 'node:fs';
import { dirname } from 'node:path';
import { fileURLToPath } from 'node:url';

const port = 8080;

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(morgan('common', {…}));

app.use(express.urlencoded({ extended: false }));

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use('/movies', moviesRouter);

app.get('/', (request, response) => response.redirect('/movies'));

const options = {
  key: readFileSync('./key.pem'),
  cert: readFileSync('./cert.pem'),
};

const server = spdy.createServer(options, app);

server.listen(port, () => {
  console.log(`Server is listening to https://localhost:${port}`);
});
```

Listing 6.43 HTTP/2 in Express (»index.js«)

Sie können überprüfen, ob die Umstellung auf das HTTP/2-Protokoll funktioniert hat, indem Sie die Entwicklertools Ihres Browsers öffnen und sicherstellen, dass das `h2`-Protokoll für das Laden der einzelnen Ressourcen verwendet wird.

6.12 Zusammenfassung

In diesem Kapitel haben Sie mit Express das populärste und am weitesten verbreitete Webapplikations-Framework für Node.js kennengelernt. Sie konnten sehen, wie Express die Entwicklung von Webanwendungen durch Abstraktion von Standardaufgaben vereinfacht und auf Geschwindigkeit, einen schlanken Kern sowie eine gute Erweiterbarkeit setzt.

Die wichtigsten Themen in diesem Kapitel waren:

- **Grundlagen und Architektur von Express**: Sie haben den mehrschichtigen Aufbau von Express kennengelernt, der auf dem Node.js-`http`-Modul basiert und durch einen Middleware-Layer sowie einen Router ergänzt wird. Mit der Installation und Initialisierung einer einfachen Express-Anwendung haben Sie einen Einstieg in die Entwicklung mit dem Framework erhalten.
- **Request- und Response-Objekte**: Die zentralen Objekte in Express zur Handhabung von Anfragen und Antworten sind die Objekte Request und Response, auf die Sie in allen Routing-Handlern und Middleware-Funktionen Zugriff haben.
- **Anwendungsstruktur (MVC)**: Eine strukturierte Architektur ist von großer Bedeutung für eine Applikation, vor allem wenn es darum geht, dass sie zukunftssicher und erweiterbar sein soll. Eine mögliche und weit verbreitete Architektur ist das Model-View-Controller-(MVC-)Muster. Sie haben außerdem verschiedene Ansätze zur Verzeichnisstruktur für kleine, mittlere und große (modulare) Applikationen gesehen.
- **Routing**: Sie haben gelernt, wie der Express-Router funktioniert, um Anfragen basierend auf HTTP-Methode und URL-Pfad an Controller-Funktionen weiterzuleiten. Dies umfasste statische Pfade, dynamische Routen mit Parametern, optionale Parameter und Muster. Für die Modularisierung einer Applikation können Sie den Express-Router ebenfalls nutzen.
- **Middleware**: Ein wichtiges Konzept in Express ist die Middleware, das sind Funktionen in der Request-Response-Kette. Sie haben gesehen, wie Sie eine eigene Middleware schreiben und die integrierten Middleware-Funktionen wie `express.static`, `express.urlencoded` oder `express.json` sowie externe Middleware wie Morgan für Logging verwenden. Die Middleware-Funktionen erweitern Ihre Applikation um allgemeine Features wie das Ausliefern statischer Dateien und das Parsen von Request-Bodys.
- **Praktische Beispielanwendung**: Anhand eines konkreten Beispiels haben Sie den Aufbau einer CRUD-Applikation Schritt für Schritt nachvollzogen. Dabei haben Sie Routing, Controller, Models und Views implementiert und miteinander verbunden.
- **Validierung von Eingaben**: Die Überprüfung von Daten, die Ihr System von außen erhält, hat eine große Bedeutung für die Konsistenz und Sicherheit Ihrer Applikation. Mit Zod haben Sie eine Bibliothek kennengelernt, die Ihnen diese Aufgabe erleichtert. Sie wissen jetzt, wie Sie ein Validierungsschema definieren und es zur Überprüfung von Benutzereingaben im Controller einsetzen. Die daraus entstehenden Fehlermeldungen haben Sie in der View Ihrer Applikation integriert und angezeigt.

- **Express 5**: Das Release von Express 5 markiert einen Wendepunkt in der Entwicklung des Frameworks. Lange Zeit hat sich bei Express nichts mehr verändert. Mit diesem Release verfolgt Express eine LTS-Strategie mit den Phasen CURRENT, ACTIVE und MAINTENANCE für mehr Planungssicherheit. Die Entwickler stellen Ihnen außerdem Migrationshilfen (Codemods) für einen einfacheren Umstieg zur Verfügung.
- **HTTPS und HTTP/2**: Sie haben gesehen, wie Sie eine Express-Applikation einfach auf HTTPS umstellen können und wie Sie HTTP/2 mithilfe des (allerdings nicht mehr aktiv entwickelten) spdy-Pakets als Workaround integrieren können.

Nach diesem Kapitel haben Sie ein solides Verständnis für die Funktionsweise von Express.js und dessen Kernkomponenten. Sie sind nun in der Lage, strukturierte Webanwendungen mit Routing, Middleware, Datenverarbeitung und Validierung zu erstellen. Express bietet eine flexible und leistungsstarke Grundlage, die Ihnen die Entwicklung komplexer Node.js-basierter Webserver erheblich erleichtert. Im nächsten Kapitel nutzen Sie Template-Engines, um die Inhalte Ihrer Applikation noch komfortabler zu rendern.

Auf einen Blick

Inhalt

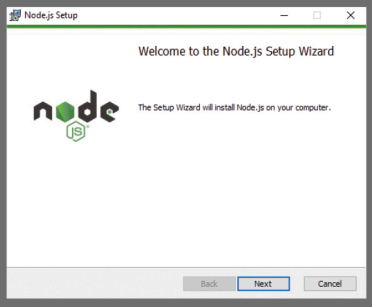
26 Künstliche Intelligenz mit Node.js nutzen 927

Node.js

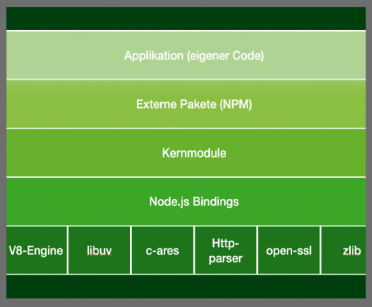
»Idealer Einstieg und verlässliches Nachschlagewerk«

Hochperformante Webapplikationen entwickeln

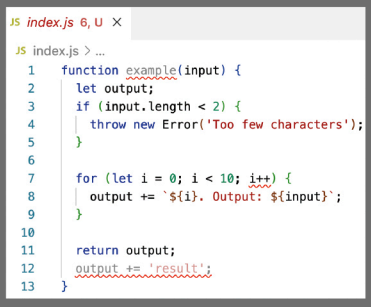
Nutzen Sie mit Node.js die Stärke von JavaScript auch serverseitig. Dieses Handbuch zeigt Ihnen, wie Sie die Laufzeitumgebung praxisnah einsetzen. Sebastian Springer führt Sie kompetent und verständlich durch alle wichtigen Themen: von den Grundlagen auf der Kommandozeile über moderne Web-APIs bis hin zur eigenen professionellen Webanwendung.



Einrichten und konfigurieren



Die Node.js-Plattform verstehen



Profi-Funktionen nutzen

Node.js von A bis Z

Dieses Handbuch deckt alle Aspekte von Node.js ab: von der Installation und den Kernkonzepten bis zu den wichtigsten Modulen. Verstehen Sie die Plattform bis ins Detail und nutzen Sie das Buch als zuverlässiges Nachschlagewerk für Ihre tägliche Arbeit.

Praxiswissen für die Webentwicklung

Steigen Sie mit zahlreichen Praxisbeispielen direkt in die Entwicklung eigener Anwendungen ein. Meistern Sie auch fortgeschrittene Themen wie Codeanalyse, Serversicherheit und Cloud-Deployment. Immer mit konkreten Code-Bausteinen.

Eigene Lösungen finden

Profitieren Sie vom modularen Aufbau von Node.js: Mit NPM, dem Node Package Manager, erweitern Sie Ihre Anwendungen gezielt und schaffen maßgeschneiderte Lösungen.

Alle Codebeispiele aus dem Buch zum Download

Sebastian Springer ist JavaScript Engineer und setzt Node.js in der täglichen Arbeit in seinen Projekten ein. Als Dozent für JavaScript, Sprecher auf zahlreichen Konferenzen und Autor ist es ihm ein Anliegen, Begeisterung für die professionelle Programmierung zu wecken.

Aus dem Inhalt

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 49,90 [D] € 51,30 [A]
Für Windows, macOS und Linux
Programmierung
ISBN 978-3-367-10834-3