# Kapitel 1
# Initialisierung und Setup

*Eine wichtige Grundlage für die effektive Entwicklung unter Node.js ist zum einen die richtige Konfiguration von Node.js und dem Node.js Package Manager (npm) sowie zum anderen das richtige Setup von Node.js-Projekten.*

In diesem Kapitel zeige ich Ihnen, wie Sie Node.js und den Paketmanager von Node.js (npm) installieren und wie Sie schnell zwischen verschiedenen Node.js-Versionen hin und her wechseln (Rezepte 1 und 2). Außerdem zeige ich Ihnen, wie Sie Node.js-Projekte richtig aufsetzen, organisieren und konfigurieren (Rezepte 3 bis 7).

- Rezept 1: Node.js installieren
- Rezept 2: Mehrere Node.js-Versionen parallel betreiben
- Rezept 3: Ein neues Node.js-Package manuell erstellen
- Rezept 4: Ein neues Node.js-Package automatisch erstellen
- Rezept 5: Den Kommandozeilenwizard von npm anpassen
- Rezept 6: Abhängigkeiten richtig installieren und verwalten
- Rezept 7: Packages in Mono-Repositorys organisieren

## 1.1 Rezept 1: Node.js installieren

Sie möchten Node.js unter macOS, Linux oder Windows installieren und wissen, welche unterschiedlichen Möglichkeiten zur Installation zur Verfügung stehen.

### 1.1.1 Arten der Installation

Die Installation von Node.js ist unabhängig vom Betriebssystem relativ einfach und auf der Website von Node.js sehr gut dokumentiert. Ich möchte mich im Folgenden daher etwas kürzer halten und mich darauf konzentrieren, welche verschiedenen Möglichkeiten der Installation Sie haben und worin sich diese unterscheiden.

Prinzipiell kann Node.js auf verschiedene Arten installiert werden: über eine Installationsdatei, über ein Binärpaket, über Paketmanager des jeweiligen Betriebssystems,

über einen sogenannten Node.js Version Manager und durch Kompilieren der Quelltextdateien von Node.js. Schauen Sie sich diese Möglichkeiten der Reihe nach an.

- über eine **Installationsdatei**: Installationsdateien stehen unter *https://nodejs.org/en/download/* derzeit nur für macOS und Windows zum Download zur Verfügung (Abbildung 1.1). Für die meisten Anwendungsfälle reicht es dabei, die sogenannte LTS-Version, also die Version mit »Long Term Support« (siehe Kasten) zu verwenden. Möchten Sie für Ihre Applikation neuere Features nutzen, können Sie sich alternativ unter dem Register »Current« auch die Installationsdateien der jeweils letzten Node.js-Version herunterladen. Für den Produktiveinsatz empfiehlt sich der Einsatz der »Current«-Version in den meisten Fällen allerdings nicht, bzw. sollten Sie dann darauf achten, in Ihrer Applikation nur stabile Features von Node.js zu nutzen.


**Abbildung 1.1** Node.js-Versionen zum Download

- über ein **Binärpaket**: Die Binärpakete können Sie ebenfalls unter *https://nodejs.org/en/download/* herunterladen und anschließend ohne weitere Installation ausführen. Allerdings hat diese Variante den Nachteil, dass Node.js erst mal nicht systemweit in der Kommandozeile zur Verfügung steht. Um dem entgegenzuwirken, müssen Sie den Suchpfad des Systems manuell erweitern bzw. die Binärdateien entsprechend an einen Ort kopieren, der bereits im Suchpfad enthalten ist.
- über den **Paketmanager** des jeweiligen Betriebssystems: Für viele Betriebssysteme wie bspw. die unterschiedlichen Linux-Distributionen, aber auch für macOS und Windows stehen spezielle Paketmanager zur Verfügung, über die sich Software zentral installieren bzw. verwalten lässt. Auch Node.js lässt sich für die meisten Betriebssysteme auf diese Weise installieren. Dabei ist allerdings zu beachten, dass die jeweiligen Packages nicht vom Node.js Core Team gepflegt werden und es







gegebenenfalls sein kann, dass Sie auf diesem Weg nicht die aktuellste Version von Node.js installieren können. Weitere Informationen zu dieser Installationsform finden Sie unter *https://nodejs.org/en/download/package-manager/*.

- über einen sogenannten **Node.js Version Manager**: Der Vorteil bei dieser Installationsart ist, dass Sie hierüber relativ komfortabel mehrere Versionen von Node.js parallel installieren und bei Bedarf von einer zur anderen Version wechseln können (Stichwort »Kompatibilitätstests«). Ich persönlich verwende diese Art der Installation und kann Ihnen das auch nur empfehlen – insbesondere, wenn Sie verschiedene Projekte verwalten oder wenn Sie schnell prüfen möchten, wie sich eine Applikation unter einer bestimmten Version verhält. Details zu der Installation über einen Node.js Version Manager finden Sie in Rezept 2.
- durch **Kompilieren der Quelltextdateien**: Dies ist sicherlich die aufwendigste Installationsmöglichkeit, die nur in Ausnahmefällen sinnvoll ist, bspw. wenn Sie sich aktiv an der Entwicklung von Node.js beteiligen möchten. Auf diese Art der Installation werde ich aus Platzgründen nicht weiter eingehen und verweise daher auf die entsprechende Dokumentation von Node.js unter *https://github.com/nodejs/node/blob/master/BUILDING.md*.

**LTS-Versionen**

Der Release-Plan von Node.js sieht alle sechs Monate eine neue Major-Version vor, wobei die Releases mit geraden Versionsnummern im April und die Releases mit ungeraden Versionsnummern im Oktober veröffentlicht werden. Bei Veröffentlichung eines Release mit ungerader Versionsnummer geht die zuvor veröffentlichte Version in den LTS-Plan (»Long Term Support«) über. Mit anderen Worten: Jede LTS-Version hat immer eine gerade Versionsnummer, jede »Nicht-LTS-Version« eine ungerade Versionsnummer. Im LTS-Plan wird die jeweilige (gerade) Version 18 Monate lang gewartet (»maintained«).

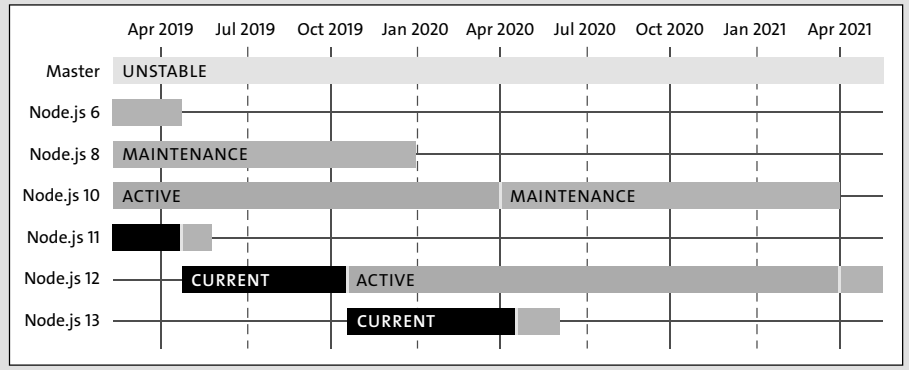


**Abbildung 1.2** Release-Plan von Node.js

### 1.1.2 Lösung: Installation über Installationsdatei unter macOS

Für die Installation unter macOS laden Sie die *pkg*-Datei von der Downloadseite *https://nodejs.org/en/download* herunter. Wenn Sie diese Datei durch Doppelklick starten, öffnet sich der in Abbildung 1.3 gezeigte Installationswizard, der Sie durch die Installation führt (Abbildung 1.4 bis Abbildung 1.6).

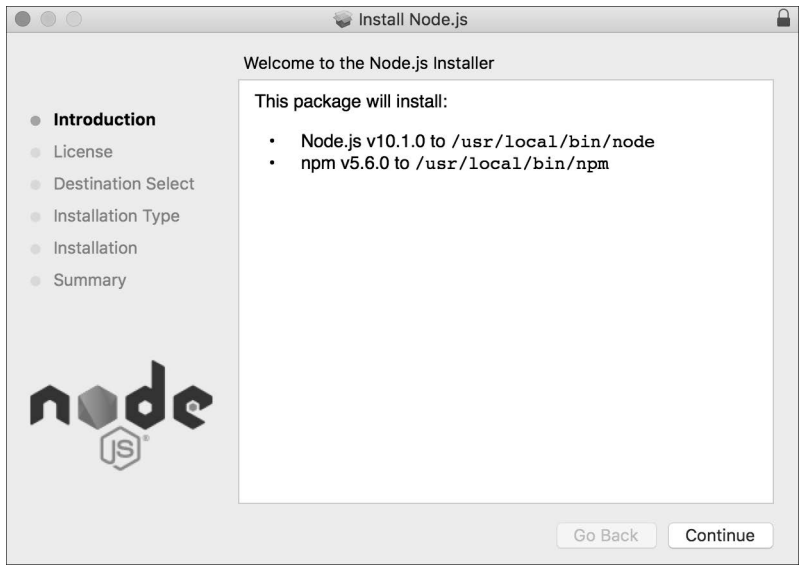


**Abbildung 1.3** Begrüßungsdialog der Node.js-Installation unter macOS

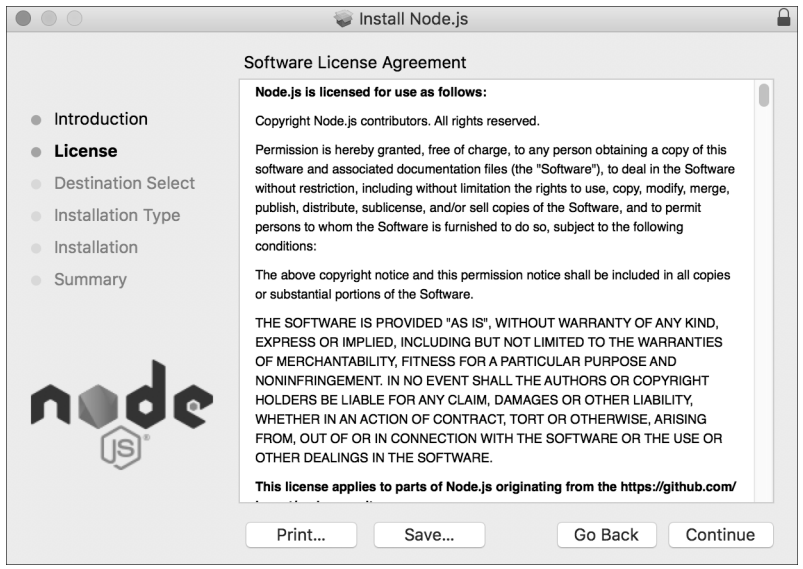


**Abbildung 1.4** Lizenzinformationen für Node.js unter macOS







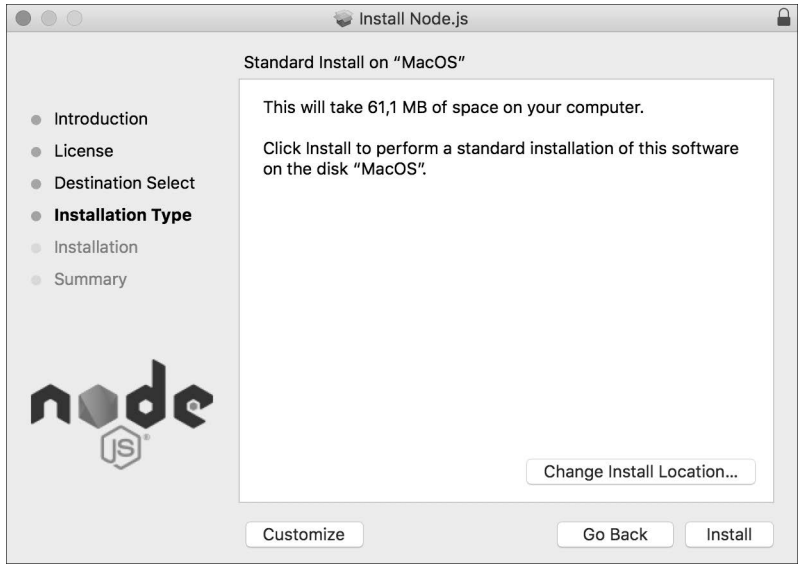


**Abbildung 1.5** Beginn der Installation von Node.js unter macOS

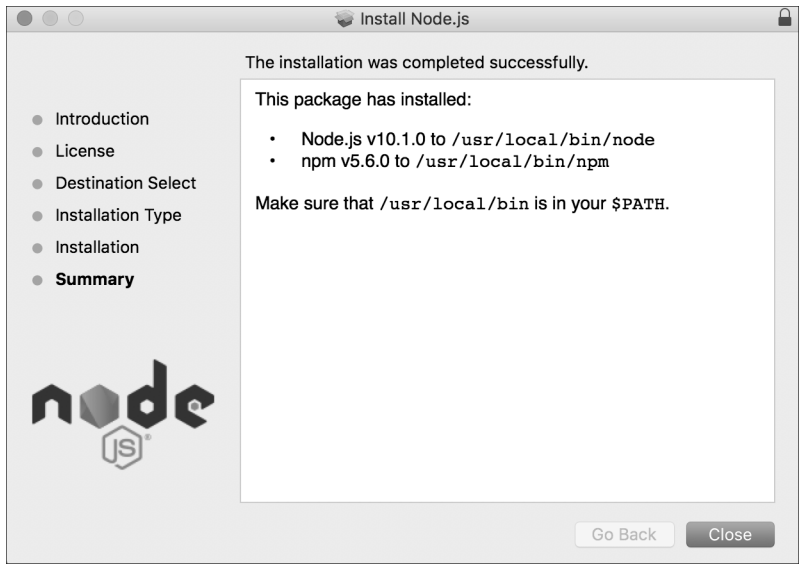


**Abbildung 1.6** Bestätigungsdialog zur Installation von Node.js unter macOS

Nach erfolgreicher Installation steht Ihnen auf Ihrem System global der Befehl `node` zur Verfügung, über den Sie Node.js-Applikationen starten können. Um die Installation zu überprüfen, können Sie sich bspw. mithilfe von `node -v` die installierte Node.js-Version ausgeben lassen:

```
$ node -v
v12.3.1
```

**Listing 1.1** Ausgabe der aktuell installierten Node.js-Version

Durch die Installation von Node.js werden noch zwei weitere wichtige Tools installiert: zum einen (seit Node.js-Version 0.6.3) der *Node.js Package Manager* (kurz npm), zum anderen (seit npm-Version 5.2.0) der *npm Package Runner* (kurz npx). Auf beides werde ich noch gesondert an anderer Stelle genauer eingehen, trotzdem können Sie schon jetzt prüfen, ob beide Tools erfolgreich installiert wurden:

```
$ npm -v
6.9.0
$ npx -v
6.9.0
```

**Listing 1.2** Ausgabe der aktuell installierten Versionen von npm und npx

### 1.1.3 Lösung: Installation über Installationsdatei unter Windows

Für die Installation unter Windows verwenden Sie die *msi*-Datei, die ebenfalls auf der Downloadseite *https://nodejs.org/en/download/* zum Download angeboten wird. Ein Starten dieser Datei öffnet den in Abbildung 1.7 gezeigten Installationswizard, wobei auch hier das meiste selbsterklärend und ähnlich wie bei der Installation unter macOS sein dürfte, sodass ich auf eine detaillierte Beschreibung und Abbildung entsprechender Screenshots verzichte.


**Abbildung 1.7** Begrüßungsdialog der Node.js-Installation unter Windows







Wie schon bei der Installation unter macOS werden durch die Installation von Node.js auch die Tools npm und npx installiert. Testen können Sie die erfolgreiche Installation über die folgenden Befehle:

```
$ node -v
v8.11.3
$ npm -v
5.6.0
$ npx -v
9.7.1
```

**Listing 1.3** Ausgabe der aktuell installierten Versionen von Node.js, npm und npx

### 1.1.4 Lösung: Installation über Binärpaket unter macOS

Das Binärpaket für macOS steht als gezipptes Tar-Archiv für 64 Bit auf der Downloadseite zur Verfügung. Wenn Sie diese Datei herunterladen und entpacken, finden Sie die ausführbare Datei *node* in dem entpackten Ordner unterhalb des Verzeichnisses *bin*.

```
$ bin/node -v
v8.11.3
$ bin/npm -v
5.6.0
$ bin/npx -v
9.7.1
```

**Listing 1.4** Ausgabe der aktuell installierten Versionen nach Installation des Binärpakets unter macOS

### 1.1.5 Lösung: Installation über Binärpaket unter Windows

Das Binärpaket für Windows steht als ZIP-Datei sowohl für 32 Bit als auch für 64 Bit zur Verfügung. Laden Sie die passende Datei herunter und entpacken Sie diese, finden Sie anschließend in dem entpackten Ordner u. a. die Datei *node.exe*. Diese Datei können Sie direkt per Doppelklick oder über die Kommandozeile ausführen:

```
$ node.exe -v
v8.11.3
$ npm -v
5.6.0
$ npx -v
9.7.1
```

**Listing 1.5** Ausgabe der aktuell installierten Versionen nach Installation des Binärpakets unter Windows

### 1.1.6 Lösung: Installation über Binärpaket unter Linux

Das Binärpaket für Linux steht als Archiv-Datei für 32 Bit und 64 Bit zur Verfügung. Nach dem Download und dem anschließenden Entpacken dieser Datei finden Sie die ausführbare Datei *node* in dem entpackten Ordner unterhalb des Verzeichnisses *bin*:

```
$ bin/node -v
v8.11.3
$ bin/npm -v
5.6.0
$ bin/npx -v
9.7.1
```

**Listing 1.6** Ausgabe der aktuell installierten Versionen nach Installation des Binärpakets unter Linux

### 1.1.7 Lösung: Installation über Paketmanager

Node.js steht für viele verschiedene Betriebssysteme bzw. Paketmanager als Package zur Verfügung, wobei – wie eingangs erwähnt – die Packages nicht vom Node.js Core Team verwaltet werden und damit die Zuverlässigkeit und Aktualität vom Verwalter des Packages abhängt. Eine zumindest halbwegs offizielle Liste von vertrauenswürdigen Quellen finden Sie unter *https://nodejs.org/en/download/package-manager/*. So stehen dort u. a. Packages für Debian, Ubuntu, FreeBSD, OpenBSD, openSUSE, Android (experimentell) und viele weitere zur Verfügung.

Ebenfalls einen Blick wert in diesem Zusammenhang ist das Git-Repository unter *https://github.com/nodesource/distributions*, in dem verschiedene Shell-Scripts angeboten werden, um Node.js auf unterschiedlichen Linux-Distributionen zu installieren. Auch für den Fall, dass Sie Node.js auf einem Raspberry Pi installieren möchten (bei dem oft das auf Debian basierende Betriebssystem Raspbian zum Einsatz kommt), ist das genannte Git-Repository eine gute Einstiegshilfe. So reichen bspw. die folgenden beiden Befehle, um Node.js auf einem Raspberry Pi zu installieren:

```
curl -sL https://deb.nodesource.com/setup_11.x | sudo -E bash -
sudo apt install -y nodejs
```

**Paketmanager für macOS und Windows**

Alternativ zu den in den vorherigen Abschnitten gezeigten Installationsmöglichkeiten für macOS und Windows können Sie auch die entsprechenden Paketmanager dieser Betriebssysteme nutzen, bspw. im Fall von macOS die Paketmanager Homebrew (*https://brew.sh/index_de*) oder MacPorts (*https://www.macports.org/*) und im Fall von Windows den Paketmanager Chocolatey (*https://chocolatey.org/*).







### 1.1.8 Ausblick

In diesem Rezept haben Sie gesehen, auf welche unterschiedlichen Weisen sich Node.js für verschiedene Betriebssysteme installieren lässt. Eine weitere Möglichkeit, die insbesondere dann interessant ist, wenn Sie parallel an verschiedenen Projekten arbeiten, die jeweils eine andere Node.js-Version voraussetzen, habe ich schon genannt, aber noch nicht im Detail gezeigt. Die Rede ist von einem Node.js Version Manager, dessen Verwendung ich Ihnen in folgendem Rezept im Detail vorstellen werde.

**Verwandte Rezepte**

- Rezept 2: Mehrere Node.js-Versionen parallel betreiben

## 1.2 Rezept 2: Mehrere Node.js-Versionen parallel betreiben

Sie möchten mehrere Versionen von Node.js parallel betreiben, um während der Entwicklung schnell zwischen einzelnen Versionen wechseln zu können.

### 1.2.1 Lösung

Bei der Entwicklung von Node.js-Applikationen ist es in vielen Fällen hilfreich, wenn man ohne großen Aufwand schnell zwischen verschiedenen Versionen von Node.js wechseln kann. Beispielsweise, wenn man in unterschiedliche Projekte involviert ist, die jeweils andere Versionen von Node.js erfordern, oder aber, wenn man die Kompatibilität einer Applikation bezüglich verschiedener Node.js-Versionen testen möchte. Ich empfehle Ihnen daher, langfristig unbedingt einen *Node.js Version Manager* zu verwenden.

**Einen Node.js Version Manager installieren**

Einer der bekanntesten Node.js Version Manager ist nvm (*https://github.com/creationix/nvm*), der für Linux und macOS zur Verfügung steht (Alternativen für Windows finden Sie in Abschnitt 1.2.2, »Alternativen«.

Als Voraussetzung für die Installation muss allerdings ein C++-Compiler installiert sein. Unter Linux installieren Sie dazu das »build-essential«-Package:

```
$ sudo apt-get update
$ sudo apt-get install build-essential
```

Für macOS reicht es, hierfür die Xcode Command Line Tools zu installieren:

```
$ xcode-select --install
```

Anschließend installieren Sie nvm wie folgt entweder über cURL oder Wget:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/
install.sh | bash
```

bzw.

```
$ wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/
install.sh | bash
```

**Hinweis**

Durch die oben gezeigten Script-Aufrufe wird zum einen das Git-Repository von nvm lokal in das Verzeichnis */.nvm* geklont, zum anderen das lokale Nutzerprofil (*/.bash_profile*, */.zshrc*, */.profile* oder */.bashrc*) angepasst, sodass nvm anschließend global verwendet werden kann. Alternativ können Sie diese Schritte auch manuell durchführen (siehe *https://github.com/creationix/nvm#git-install*).

### Node.js-Versionen installieren

Nach erfolgreicher Installation von nvm lassen sich einzelne Node.js-Versionen gezielt über den Befehl `nvm install` installieren. Um bspw. die aktuellste Version von Node.js zu installieren, verwenden Sie folgenden Befehl:

```
$ nvm install node
```

Die aktuellste LTS-Version installieren Sie dagegen wie folgt:

```
$ nvm install --lts
```

Alternativ können Sie dem Befehl auch eine exakte Versionsnummer übergeben. Um bspw. die Version 8.1.1 zu installieren, verwenden Sie folgenden Befehl:

```
$ nvm install 8.1.1
```

Da sich nvm an der *semantischen Versionierung* orientiert, können Sie dabei auch einzelne Teile der Versionsnummer weglassen. Der Befehl

```
$ nvm install 8.1
```

bspw. installiert die aktuellste Version 8.1, sprich die Version 8.1.x, wobei x für die aktuellste Patch-Version steht.

### Node.js-Versionen wechseln

Wenn Sie über `nvm install` eine neue Node.js-Version installieren, wird diese implizit auch als Standardversion gesetzt. Um explizit zu einer bereits installierten Version zu wechseln, können Sie den Befehl `nvm use` verwenden:







```
$ nvm use 8.9.4  # Wechsel zu Version 8.9.4
$ nvm use 9.3    # Wechsel zu Version 9.3.0
$ nvm use node   # Wechsel zur aktuellsten Version
```

Intern erstellt nvm bei einem Wechsel der Node.js-Version einen symbolischen Link vom `node`-Befehl zu der jeweiligen Node.js-Instanz.

**Hinweis**

Über die Konfigurationsdatei *.nvmrc* (*https://github.com/creationix/nvm#nvmrc*, nicht zu verwechseln mit der npm-Konfigurationsdatei *.npmrc*) können Sie die zu verwendende Node.js-Version auch pro Projekt definieren. Ein anschließender Aufruf von `npm use` wechselt dann die Node.js-Version basierend auf dieser Konfigurationsdatei.

Ebenfalls nützlich: Über den Befehl `nvm alias` haben Sie die Möglichkeit, Aliasse zu erzeugen, die Sie dann in Kombination mit `nvm use` verwenden können:

```
$ nvm alias development-version 8.9.4 # Erstellen eines Alias
$ nvm use development-version         # Wechsel zu Version 8.9.4
$ nvm unalias development-version     # Löschen des Alias
```

### Verfügbare Versionen ermitteln

Welche Versionen von Node.js auf Ihrem Rechner installiert sind bzw. über nvm verwaltet werden, können Sie über den Befehl `nvm ls` ermitteln:

```
$ nvm ls
  v7.10.0
   v8.1.3
   v8.4.0
->      v9.11.1
   system
default -> node (-> v9.11.1)
node -> stable (-> v9.11.1) (default)
stable -> 9.11 (-> v9.11.1) (default)
iojs -> N/A (default)
lts/* -> lts/carbon (-> N/A)
lts/argon -> v4.9.1 (-> N/A)
lts/boron -> v6.14.1 (-> N/A)
lts/carbon -> v8.11.1 (-> N/A)
```

**Listing 1.7** Auflistung der verfügbaren Node.js-Versionen

Um dagegen zu ermitteln, welche Versionen generell zur Verfügung stehen, verwenden Sie den Befehl `nvm ls-remote`:

```
$ nvm ls-remote
  v0.1.14
  v0.1.15
  v0.1.16
  v0.1.17
  v0.1.18
  v0.1.19
  v0.1.20
  ...
  v8.11.0   (LTS: Carbon)
  v8.11.1   (Latest LTS: Carbon)
  v9.0.0
  ...
  v9.10.0
  v9.10.1
  v9.11.0
->  v9.11.1
```

**Listing 1.8** Auflistung der installierten und zur Verfügung stehenden Node.js-Versionen (gekürzte Ausgabe)

#### Node.js-Versionen deinstallieren

Wenn Sie eine einzelne Node.js-Version nicht länger benötigen, können Sie diese über den Befehl `nvm uninstall` wieder deinstallieren:

```
$ nvm uninstall 0.11
```

#### Globale Packages beim Versionswechsel aktualisieren

Wie Sie in Rezept 1 gesehen haben, wird bei der Installation von Node.js direkt auch der Node.js Package Manager npm installiert. Wenn Sie mithilfe von nvm verschiedene Versionen von Node.js installieren, werden folglich auch verschiedene Versionen von npm installiert. Dabei ist zu beachten, dass globale Packages nicht wie bei der »Single-Node.js-Installation« in einem einzigen Verzeichnis installiert werden, sondern jeweils getrennt pro Node.js-Version unter »~/.nvm/versions/node/<version>/lib/node_modules«. Dies hat zwar den netten Nebeneffekt, dass zur Installation von Packages keine sudo-Rechte notwendig sind (siehe auch Abschnitt 1.6.5, »Lösung: Installieren von globalen Packages ohne sudo-Rechte«), allerdings bedeutet dies auch, dass globale Packages nicht automatisch global für alle Node.js-Versionen zur Verfügung stehen: Wenn Sie für eine spezielle Node.js-Version ein globales Package installieren und dann zu einer anderen Node.js-Version wechseln, ist das jeweilige Package







für diese Version nicht installiert. Doch nvm hat diesbezüglich vorgesorgt: Über den Parameter `--reinstall-packages-from` können Sie bei der Installation einer neuen Node.js-Version angeben, von welcher bereits installierten Version die globalen Packages übernommen (und nachinstalliert) werden sollen:

```
$ nvm install v9.0.0 --reinstall-packages-from=8.9
```

Tabelle 1.1 gibt Ihnen eine Übersicht über die wichtigsten Befehle, die nvm zur Verfügung stellt.

| Befehl | Beschreibung |
|---|---|
| `nvm install <version>` | Installiert die angegebene Node.js-Version. |
| `nvm uninstall <version>` | Deinstalliert die angegebene Node.js-Version. |
| `nvm use <version>` | Verwendet die angegebene Node.js-Version. |
| `nvm current` | Gibt die aktuell verwendete Node.js-Version aus. |
| `nvm alias <aliasname> <version>` | Erzeugt ein Alias für die angegebene Node.js-Version. |
| `nvm ls` | Listet die lokal verfügbaren Node.js-Versionen auf. |
| `nvm ls-remote` | Listet die verfügbaren Node.js-Versionen auf, die generell für die Installation zur Verfügung stehen. |

**Tabelle 1.1** Übersicht über die wichtigsten Befehle von nvm

### 1.2.2 Alternativen

Neben nvm gibt es noch einige andere Alternativen, um verschiedene Versionen von Node.js zu verwenden, die ich Ihnen im Folgenden vorstellen möchte.

#### Alternative Node.js Version Manager

Alternativ zu nvm stehen eine Reihe weiterer Node.js Version Manager zur Verfügung. Die prominenteste Alternative dürfte »n« (*https://github.com/tj/n*) von TJ Holowaychuk sein, die Sie direkt als Node.js-Anwendung über npm installieren können (allerdings funktioniert »n« ebenfalls nicht unter Windows).

Ebenfalls interessant ist das Tool avn (für »Automatic Version Switching«, *https://github.com/wbyoung/avn*), welches das automatische Wechseln der Node.js-Version abhängig von einer *.node-version*-Datei vornimmt. Wechselt man in ein Projekt mit solch einer Datei, wird automatisch die in dieser Datei enthaltene Version verwendet. Als Node.js Version Manager unterstützt avn dabei sowohl nvm als auch »n«.

### Node.js Version Manager für Windows

Der Node.js Version Manager nvm funktioniert, wie bereits gesagt, nicht unter Windows. Es steht mit nvm-windows (*https://github.com/coreybutler/nvm-windows*) allerdings eine Alternative zur Verfügung (die übrigens trotz des ähnlichen Namens nicht von dem nvm-Team gewartet wird). nvm-windows ist in Go geschrieben und kann als Installationsdatei unter *https://github.com/coreybutler/nvm-windows/releases* heruntergeladen werden. Die Befehle von nvm-windows sind im Großen und Ganzen ähnlich den Befehlen von nvm (Tabelle 1.2).

| Befehl | Beschreibung |
|---|---|
| `nvm install <version>` | Installiert die angegebene Node.js-Version. |
| `nvm uninstall <version>` | Deinstalliert die angegebene Node.js-Version. |
| `nvm use <version>` | Verwendet die angegebene Node.js-Version. |
| `nvm list` | Listet die verfügbaren Node.js-Versionen auf. |
| `nvm list available` | Listet die verfügbaren Node.js-Versionen auf, die generell für die Installation zur Verfügung stehen. |

**Tabelle 1.2** Übersicht über die wichtigsten Befehle von nvm-windows

### CI-Systeme und Docker

Ein Node.js Version Manager eignet sich sehr gut für das schnelle Wechseln zwischen verschiedenen Versionen auf einem Entwicklungsrechner. Für das automatische Testen einer Applikation unter verschiedenen Versionen empfiehlt es sich natürlich, langfristig auf automatisierte Techniken zurückzugreifen wie bspw. die Verwendung von CI-Systemen (Travis CI, Circle CI, GitLab CI, Jenkins etc.).

Auch der Einsatz von Docker-Images, die jeweils eine spezielle Node.js-Version verwenden, ist sinnvoll. Docker-Images können Sie ohne viel Aufwand lokal erstellen und dann Ihre Node.js-Anwendung innerhalb eines Docker-Containers laufen lassen. Wie das geht, zeige ich Ihnen in Kapitel 13, »Publishing, Deployment und Microservices«.

# Kapitel 3
# Logging und Debugging

*In diesem Kapitel zeige ich Ihnen, wie Sie Node.js-Anwendungen debuggen und für das Logging konfigurieren.*

Im Folgenden schauen wir uns zwei wichtige Aspekte an, die bei der Fehlersuche in Node.js-Applikationen helfen können: In den Rezepten 16 bis 18 zeige ich Ihnen, wie Sie das Logging einrichten, und in den Rezepten 19 bis 21 stelle ich Ihnen verschiedene Möglichkeiten vor, Node.js-Anwendungen zu debuggen.

- Rezept 16: Logging für Node.js-Packages einrichten
- Rezept 17: Logging für Node.js-Applikationen einrichten
- Rezept 18: Logging über Adapter-Packages einrichten
- Rezept 19: Applikationen mit Chrome Developer Tools debuggen
- Rezept 20: Applikationen mit Visual Studio Code debuggen
- Rezept 21: Applikationen über die Kommandozeile debuggen

## 3.1 Rezept 16: Logging für Node.js-Packages einrichten

Sie möchten dafür sorgen, dass in Ihren Node.js-Packages Logging-Informationen ausgegeben werden.

### 3.1.1 Exkurs: Logging

Während des Lebenszyklus einer Applikation ist es hilfreich, gewisse Informationen zu protokollieren, z. B. Zustände von Objekten, Werte von Variablen, Fehlermeldungen, Nutzerinformationen oder andere Informationen bezüglich des aktuellen Programmzustands. Solche Informationen können Ihnen dann bspw. dabei helfen, in Produktivsystemen Fehlerursachen zu finden oder generell einen Überblick über die Abläufe einer Applikation zu erhalten.

Node.js und andere JavaScript-Laufzeitumgebungen, wie sie z. B. in Browsern zum Einsatz kommen, stellen standardmäßig das `console`-Objekt zur Verfügung, mit dessen Hilfe Sie auf die Kommandozeile zugreifen und Informationen darauf ausgeben können. Das `console`-Objekt stellt dabei u. a. die Methoden `info()`, `warn()` und `error()`

zur Verfügung, um allgemeine Informationen, Warnungen oder Fehler auszugeben (Listing 3.1). Je nach Laufzeitumgebung und verwendeter Methode wird die Ausgabe dann gegebenenfalls farbig hervorgehoben und/oder mit einem entsprechenden Icon versehen, damit Sie schnell zwischen dem Typ der Ausgabe unterscheiden können.

```
console.log('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  console.error(error.message);
}
```

**Listing 3.1** Logging über das »console«-Objekt

Auch wenn man während der Entwicklung relativ schnell dazu neigt, das Logging über das `console`-Objekt zu implementieren, rate ich Ihnen dringend davon ab, sobald ein Package oder eine Applikation etwas umfangreicher wird. Das wesentliche Problem bei der Verwendung des `console`-Objekts ist nämlich, dass sich einzelne Typen von Ausgaben nicht gezielt an- bzw. abschalten lassen. So ist es bspw. nicht ohne Weiteres möglich, für Ihre gesamte Applikation nur Fehlermeldungen auszugeben, normale Meldungen aber zu unterdrücken. Auch das gezielte An- bzw. Abschalten der Log-Ausgabe bestimmter Komponenten einer Applikation oder das Umleiten von Log-Ausgaben in eine Datei oder auf einen Logging-Server ist auf diese Weise nicht möglich.

Besser ist es daher, auf professionellere Lösungen zurückzugreifen. Welche Sie dabei verwenden, hängt im Wesentlichen davon ab, welche Art von Node.js-Projekt Sie entwickeln:

- Entwickeln Sie ein **Node.js-Package**, das von anderen Node.js-Packages oder Node.js-Applikationen eingebunden werden soll, greifen Sie am besten auf das »debug«-Package (*https://github.com/visionmedia/debug*) zurück (dieses Rezept), weil dies eine besonders schlanke Logging-Bibliothek ist.
- Entwickeln Sie eine komplexere **Node.js-Applikation**, verwenden Sie am besten eine Logging-Bibliothek wie »winston« (*https://github.com/winstonjs/winston*), »bunyan« (*https://github.com/trentm/node-bunyan*) und »log4js-node« (*https://github.com/log4js-node/log4js-node*), einen Port von log4js (*https://github.com/stritti/log4js*) für Node.js (Rezept 17).







- Entwickeln Sie eine **Microservice-Anwendung** oder generell eine komplexere Node.js-Anwendung, ist es zudem sinnvoll, die Logging-Ausgabe zur besseren Analyse zentral zu verwalten bzw. an zentraler Stelle zu sammeln, bspw. über Tools wie den ELK-Stack (bestehend aus der Suchmaschine Elasticsearch, dem Log-Verarbeitungs-Tool Logstash und der Weboberfläche Kibana, *https://www.elastic.co/de/elk-stack*). Übrigens: Wie Sie mithilfe von Node.js Microservice-Anwendungen konzipieren und implementieren, zeige ich Ihnen in Kapitel 13, »Publishing, Deployment und Microservices«, in den Rezepten 102 (Microservice-Architekturen verstehen) und 103 (Microservice-Architekturen aufsetzen mit Docker Compose).

Die Verwendung der genannten Logging-Lösungen bietet verschiedene Vorteile:

- **Timestamps**: Logging-Meldungen können mit einem Zeitstempel versehen werden, was später bei der Analyse der Log-Daten hilfreich ist.
- **Log-Levels**: Über sogenannte *Log-Levels* können Sie gezielt – auch zur Laufzeit einer Applikation – die Art einer Meldung definieren, z. B. ob es sich um einen Fehler, eine Warnung, eine Debug-Ausgabe oder eine generelle Information handelt (siehe Tabelle 3.1).
- **Namespaces**: Bestimmte Komponenten einer Applikation lassen sich zu Namespaces bzw. Logging-Gruppen zusammenfassen, sodass sich gezielt definieren lässt, für welche Komponenten das Logging aktiviert sein soll und für welche nicht.
- **Log-Analyse**: Das Ziel der protokollierten Meldungen lässt sich nahezu frei wählen. So ist es bspw. möglich, die Meldungen in Dateien, Datenbanken, über HTTP an spezielle Webservices oder über UDP an Logging-Dienste wie Logstash (*https://www.elastic.co/de/products/logstash*) zu übertragen. Dies hilft Ihnen bei der Analyse von Logs und beim Finden von Fehlern. Analysetools wie der erwähnte ELK-Stack sind in diesem Zusammenhang auf jeden Fall einen Blick wert.

| Log-Level | Beschreibung |
|---|---|
| `fatal` | Die Anwendung wird aufgrund eines Fehlers gestoppt und steht nicht mehr zur Verfügung. |
| `error` | Die aktuelle Anfrage konnte nicht bearbeitet werden bzw. erzeugte einen Fehler, die Anwendung als Ganzes wird aber weiterhin ausgeführt. |
| `warn` | Hinweis über eine Auffälligkeit im Programmablauf |
| `info` | Informationen zu einer normalen Operation |

**Tabelle 3.1** Typische Levels für das Logging

| Log-Level | Beschreibung |
|---|---|
| `debug` | detaillierte Informationen zu einer normalen Operation |
| `trace` | sehr detaillierte Informationen |

**Tabelle 3.1** Typische Levels für das Logging (Forts.)

### 3.1.2 Lösung: das »debug«-Package

Falls Sie ein Package entwickeln, dessen Ziel es ist, von anderen Packages oder Node.js-Applikationen verwendet zu werden, empfehle ich Ihnen den Einsatz des »debug«-Packages (*https://github.com/visionmedia/debug*), das Sie wie folgt installieren:

```
$ npm install debug
```

Anschließend binden Sie das Package wie gewohnt über `require()` ein und erstellen über den Aufruf von `debug()` eine Logger-Funktion, wobei Sie als Parameter einen *Namespace* übergeben, über den Sie später den Logger an- oder ausschalten können. Anschließend rufen Sie die Logger-Funktion wie folgt auf:

```
// src/start-debug.js
const debug = require('debug');
const logger = debug('my-application');

logger('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  logger(error.message);
}
```

**Listing 3.2** Logging mit dem »debug«-Package

Standardmäßig produziert das »debug«-Package keine Ausgabe, d. h., wenn Sie das obige Programm über `node src/start-debug.js` starten, sehen Sie auf der Konsole erst mal nichts. Um das Logging zu aktivieren, müssen Sie den verwendeten Namespace als Wert für die Umgebungsvariable `DEBUG` hinzufügen. Diese können Sie über entsprechende Kommandozeilenbefehle definieren, am einfachsten ist es aber, diese beim Start des Programms wie folgt mitzugeben (siehe auch Rezept 22 in Kapitel 4, »Konfiguration und Internationalisierung«):







```
$ DEBUG=my-application node src/start-debug.js
  my-application Program started +0ms
  my-application Example error +3ms
```

**Hinweis**

Da das »debug«-Package sehr weit verbreitet ist und von vielen anderen bekannten Packages verwendet wird (laut npm-Registry derzeit von mehr als 27.000), können Sie das Logging dieser Packages ebenfalls über die Umgebungsvariable `DEBUG` steuern. Für eine Applikation, die das Webframework Express verwendet können Sie das Logging bspw. wie folgt aktivieren:

```
$ DEBUG=express* node server.js
```

Auf die weiteren Möglichkeiten von Namespaces und die Funktion von Wildcards gehe ich im Folgenden genauer ein.

#### Wildcards

Über Namespaces können Sie exakt steuern, welche Log-Meldungen ausgegeben werden sollen und welche nicht. Um das noch besser nachvollziehen zu können, passen Sie das obige Programm wie in Listing 3.3 zu sehen an.

```
const debug = require('debug');
const logger = debug('my-application');
const errorLogger = debug('my-application:error');
const function1Logger = debug('my-application:function1');
const function2Logger = debug('my-application:function2');

logger('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  errorLogger(error.message);
}
```

```
function function1() {
  function1Logger('function1() executing');
  setTimeout(function1, 500);
}

function1();

function function2() {
  function2Logger('function2() executing');
  setTimeout(function2, 500);
}

function2();
```

**Listing 3.3** Verwenden von hierarchischen Namespaces mit dem »debug«-Package

Das Programm wurde hier um zwei Funktionen (`function1()` und `function2()`) sowie um drei weitere Logger (`function1Logger`, `function2Logger` und `errorLogger`) ergänzt: Der Logger `function1Logger` wird dabei von der Funktion `function1()` verwendet, der Logger `function2Logger` von der Funktion `function2()` und der Logger `errorLogger` für das Logging von Fehlermeldungen.

Wenn Sie nun das Programm über den gleichen Befehl wie eben starten, werden Sie feststellen, dass nur Folgendes ausgegeben wird:

```
my-application Program started +0ms
```

Der Grund: Die drei neu hinzugefügten Logger werden durch die Namespace-Angabe `my-application` nicht abgedeckt. Um alle Log-Meldungen unterhalb dieses Namespaces zu aktivieren, müssen Sie daher *Wildcards* einsetzen. So können Sie bspw. über `DEBUG=my-application.*` *nur* die drei neu hinzugefügten Logger aktivieren und über `DEBUG=my-application*` *alle* der insgesamt vier Logger:

```
$ DEBUG=my-application* node start-debug.js
  my-application Program started +0ms
  my-application:error Example error +0ms
  my-application:function1 function1() executing +0ms
  my-application:function2 function2() executing +0ms
  my-application:function1 function1() executing +505ms
  my-application:function2 function2() executing +505ms
```

### 3.1.3 Ausblick

Das »debug«-Package eignet sich insbesondere dann, wenn Sie ein Node.js-Package implementieren. Wenn Sie dagegen eine Node.js-Applikation implementieren, gibt





es noch einige andere Logging-Bibliotheken, die Sie in Betracht ziehen sollten. Welche das sind und welche zusätzlichen Features sie anbieten, zeige ich in dem folgenden Rezept.

Allerdings spricht auch nichts dagegen, das »debug«-Package auch für das Logging in Node.js-Applikationen zu verwenden und umgekehrt die im Folgenden beschriebenen Bibliotheken auch für das Logging in Node.js-Packages. Die in diesem Buch vollzogene Unterscheidung gibt Ihnen nur eine grobe Richtlinie, was ich für sinnvoll halte.



**Verwandte Rezepte**

- Rezept 17: Logging für Node.js-Applikationen einrichten
- Rezept 18: Logging über Adapter-Packages einrichten

## 3.2 Rezept 17: Logging für Node.js-Applikationen einrichten

Sie möchten dafür sorgen, dass Ihre Applikation Logging-Informationen speichert.

### 3.2.1 Lösung: Logging mit »winston«

Ein Package, das sich für das Logging in Node.js-Applikationen anbietet, ist das Package »winston« (*https://github.com/winstonjs/winston*), das Sie über folgenden Befehl installieren können:

```
$ npm install winston`
```

Um »winston« verwenden zu können, erstellen Sie, wie in Listing 3.4 zu sehen, über die Methode `createLogger()` zunächst eine Logger-Instanz. Als Parameter übergeben Sie dabei ein Konfigurationsobjekt, über das Sie bspw. das Log-Level, das Format der Log-Meldungen sowie sogenannte Transports definieren können. Über Letztere lässt sich die Ausgabe der Log-Meldungen z. B. in Dateien, in Datenbanken oder an Webservices weiterleiten. »winston« liefert von Haus aus schon einige Transport-Möglichkeiten mit, lässt sich dank Plugin-System aber um beliebige weitere Transports erweitern. Eine Übersicht über existierende Transports finden Sie unter *https://github.com/winstonjs/winston/blob/master/docs/transports.md*. Dazu zählen bspw. Transports für MongoDB (Rezept 49), Cassandra (Rezept 53) und Elasticsearch. Transports lassen sich zudem kombinieren, sodass Sie z. B. die Log-Ausgabe gleichzeitig auf die Konsole ausgeben, in eine Datei schreiben und per UDP an Logstash zur Log-Analyse senden können. In Listing 3.4 sehen Sie zudem, dass ein Transport-Typ auch mehrfach verwendet werden kann. Hier werden bspw. zwei Transports vom Typ

`winston.transports.File` definiert: Der eine schreibt nur die Fehlermeldungen in die Datei *error.log*, der andere alle Arten von Meldungen in die Datei *all.log*.

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({
      filename: 'error.log',
      level: 'error'
    }),
    new winston.transports.File({
      filename: 'all.log'
    })
  ]
});

// Ausgabe auf Konsole
logger.info('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  // Ausgabe in Datei
  logger.error(error.message);
}
```

**Listing 3.4** Logging mit »winston«

In Listing 3.4 wird als Ausgabeformat für die Meldungen das JSON-Format verwendet. Neben JSON unterstützt »winston« aber auch weitere Formate und erlaubt über die Methode `winston.format()` sogar die Definition individueller Ausgabeformate.

**Browser-Support**

Beachten Sie: Derzeit kann »winston« noch nicht wie andere JavaScript-Logging-Bibliotheken im Browser verwendet werden, sondern nur unter Node.js. Zum Zeitpunkt der Drucklegung dieses Buches ist laut offizieller Roadmap (*https://github.com/winstonjs/winston/blob/master/CONTRIBUTING.md#roadmap*) Browser-Support erst mit einer der nächsten Versionen geplant. Sollten Sie also ein Package entwickeln, das sowohl unter Node.js als auch in Browsern funktionieren soll (Stichwort *Isormophic JavaScript*), ist »winston« momentan keine Option.







### 3.2.2 Lösung: Logging mit »bunyan«

Das Package »bunyan« (*https://github.com/trentm/node-bunyan*) gilt als besonders schlanke und schnelle Logging-Bibliothek, bietet allerdings im Gegensatz zu »winston« auch nur JSON als Logging-Format an. Installieren können Sie »bunyan« über folgenden Befehl:

```
$ npm install bunyan
```

Wie schon bei »winston« erstellen Sie auch bei »bunyan« über `createLogger()` zunächst eine Logger-Instanz (Listing 3.5) und definieren dabei über ein Konfigurationsobjekt Details wie das Log-Level und das Ziel für die Log-Meldungen. Letzteres definieren Sie über die Eigenschaft `streams` (wollen Sie nur eine einzelne Ausgabe definieren, können Sie dies auch über Eigenschaft `stream` machen). Streams funktionieren dabei ähnlich wie die Transports bei »winston« und lassen sich ebenfalls kombinieren. So ist es auch bei »bunyan« relativ einfach möglich, die Log-Ausgabe parallel an mehrere Streams weiterzuleiten. In Listing 3.5 bspw. werden zwei Streams verwendet: Meldungen vom Typ »info« werden an die Standardausgabe weitergeleitet und Meldungen vom Typ »error« in die Datei *error.log*.

```
const bunyan = require('bunyan');

const logger = bunyan.createLogger({
  name: 'example-logger',
  level: 'info',
  streams: [
    {
      level: 'info',
      stream: process.stdout
    },
    {
      level: 'error',
      path: 'error.log'
    }
  ]
});
```

```
// Ausgabe auf Konsole
logger.info('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  // Ausgabe in Datei
  logger.error(error.message);
}
```

**Listing 3.5** Logging mit »bunyan«

**Browser-Support**

Im Unterschied zu »winston« kann »bunyan« sowohl unter Node.js als auch – dank Browserify (*http://browserify.org/*) und Webpack (*https://webpack.js.org/*) – im Browser verwendet werden. Für Packages, die sowohl unter Node.js als auch im Browser funktionieren sollen, ist »bunyan« als Logging-Bibliothek daher momentan die bessere Wahl.

### 3.2.3 Lösung: Logging mit »log4js-node«

Ein weiteres Package für das Logging unter Node.js ist das Package »log4js«. Die Installation des Packages geschieht über folgenden Befehl:

```
$ npm install log4js`
```

Anschließend binden Sie das Package über `require('log4js')` in Ihre Applikation ein. Das Konfigurieren und Erzeugen einer Logger-Instanz geschieht im Gegensatz zu dem bei »winston« und »bunyan« in zwei Schritten: Wie in Listing 3.6 zu sehen, konfigurieren Sie zunächst über die Methode `configure()` u. a., wohin die Meldungen geschrieben werden sollen. Auch hierbei übergeben Sie ein Konfigurationsobjekt, über das Sie u. a. über die Eigenschaft `appenders` verschiedene Ausgaben definieren können (analog zu den Transports in »winston« und den Streams in »bunyan«). Anschließend erzeugen Sie über einen Aufruf von `getLogger()` eine entsprechend konfigurierte Logger-Instanz.

```
const log4js = require('log4js');

log4js.configure({
```





```
  appenders: {
    file: {
      type: 'file',
      filename: 'error.log'
    }
  },
  categories: {
    default: {
      appenders: ['file'],
      level: 'info'
    }
  }
});

const logger = log4js.getLogger('file');

// Ausgabe auf Konsole
logger.info('Program started');

const throwError = () => {
  throw new Error('Example error');
};

try {
  throwError();
} catch (error) {
  // Ausgabe in Datei
  logger.error(error.message);
}
```

**Listing 3.6** Logging mit »log4js-node«

### 3.2.4 Ausblick

Ob Sie für das Logging »winston«, »bunyan« oder »log4js-node« verwenden, ist letztendlich wie so oft bei Third-Party-Node.js-Packages eine Frage des persönlichen Geschmacks. Alle drei Bibliotheken sind mehr oder weniger gleich stark vertreten und unterscheiden sich nur in den Details (Ausnahme: Browser-Support). Wenn Sie sich nicht festlegen möchten, bietet sich die Verwendung einer sogenannten *Adapter-Klasse* bzw. eines *Adapter-Packages* an. Was dahintersteckt, zeige ich Ihnen im nächsten Rezept.

# Kapitel 7
# Persistenz

*Ob relational oder nicht relational: Node.js macht in jeder Hinsicht eine gute Figur in Bezug auf die Integration von Datenbanken.*



Für alle bekannten Datenbanksysteme stehen mittlerweile entsprechende Treiber für Node.js zur Verfügung. In diesem Kapitel zeige ich Ihnen anhand bekannter Datenbanksysteme, wie die Integration in eine Node.js-Anwendung funktioniert. Dabei gehe ich zunächst auf *relationale Datenbanken* (auch: *SQL-Datenbanken*) ein (Rezepte 48 bis 50), im Anschluss (Rezepte 51 bis 53) dann auf *nicht relationale Datenbanken* (auch: *NoSQL-Datenbanken*). In jedem Rezept zeige ich Ihnen dabei, wie Sie sich aus einer Node.js-Applikation mit der jeweiligen Datenbank verbinden und wie Sie die sogenannten *CRUD*-Methoden (Create, Read, Update, Delete) ausführen können.

- Rezept 48: Auf eine MySQL-Datenbank zugreifen
- Rezept 49: Auf eine PostgreSQL-Datenbank zugreifen
- Rezept 50: Objektrelationale Mappings definieren
- Rezept 51: Auf eine MongoDB-Datenbank zugreifen
- Rezept 52: Auf eine Redis-Datenbank zugreifen
- Rezept 53: Auf eine Cassandra-Datenbank zugreifen

## 7.1 Rezept 48: Auf eine MySQL-Datenbank zugreifen

Sie möchten auf eine MySQL-Datenbank zugreifen und Datensätze lesen, anlegen, aktualisieren oder löschen.

### 7.1.1 Lösung

Auch wenn für die Neuentwicklung einer Node.js-Anwendung der Trend eher zu der Verwendung einer NoSQL-Datenbank geht, sind *relationale Datenbanksysteme* (*RDBMS*) wie MySQL nach wie vor weit verbreitet. Insbesondere hinsichtlich komplexer Anfragen und bspw. Transaktionsmanagement sind relationale Datenbanken NoSQL-Datenbanken vorzuziehen.

### MySQL installieren

Damit Sie ohne viel Aufwand lokal auf Ihrem Rechner eine MySQL-Installation zum Laufen bringen, habe ich Ihnen eine entsprechende Docker-Compose-Datei vorbereitet (Listing 7.1). Neben der eigentlichen Datenbank wird zudem das Datenbankverwaltungssystem Adminer (*https://www.adminer.org/*) gestartet, das Sie anschließend unter *http://localhost:8082/* aufrufen können. Damit haben Sie direkt eine grafische (Web)Oberfläche, mit der Sie auf die MySQL-Datenbank zugreifen können.

```
version: '3.1'
services:
  db:
    image: mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 1234
    ports:
      - 3306:3306
    expose:
      - 3306
    volumes:
      - ./init-db-sql:/docker-entrypoint-initdb.d
  adminer:
    image: adminer
    restart: always
    ports:
      - 8082:8080
```

**Listing 7.1** Docker-Compose-Datei für MySQL

Das Schema für die im Folgenden verwendete Beispieldatenbank sowie die Beispieldaten, die wir im Folgenden verwenden wollen, finden Sie unter *https://dev.mysql.com/doc/index-other.html*. Nach erfolgreichem Download legen Sie die beiden entsprechenden Dateien *sakila-schema.sql* und *sakila-data.sql* einfach innerhalb Ihres Node.js-Projektes in das Verzeichnis *init-db-sql* (bzw. in das Verzeichnis, in dem sich die Docker-Compose-Datei befindet). Dieses Verzeichnis ist in der Konfiguration von Docker Compose auf den Ordner */docker-entrypoint-initdb.d* innerhalb des resultierenden Docker-Containers gemountet. Alle Dateien mit den Endungen *.sh*, *.sql* und *.sql.gz*, die innerhalb dieses Verzeichnisses liegen (bzw. eben in dem Verzeichnis, das dorthin gemountet ist), werden beim Erstellen des Containers ausgeführt und dienen dazu, den initialen Aufbau der Datenbank zu konfigurieren. Beachten Sie dabei: Da das Ausführen der Dateien in alphabetischer Reihenfolge geschieht, müssen Sie die beiden heruntergeladenen Dateien so umbenennen, dass die Schema-Datei in dieser Reihenfolge vorn steht und die Daten-Datei hinten. Ansonsten kommt es bei





dem Ausführen der Dateien zu einem Fehler, weil beim Anlegen der Daten die durch das Schema definierte Struktur nicht gefunden wurde.

### Client für MySQL installieren

Der bekannteste und wahrscheinlich meistgenutzte MySQL-Client für Node.js ist das Package »mysql« (*https://github.com/mysqljs/mysql*), das Sie wie folgt für Ihr Projekt installieren können:

```
$ npm install mysql
```



### Verbindung herstellen

Um über das Package »mysql« eine Verbindung zu der Datenbank herzustellen, erstellen Sie, wie in Listing 7.2 zu sehen, zunächst über die Methode `createConnection()` ein Objekt, das die Verbindung repräsentiert, wobei über ein Konfigurationsobjekt die Verbindungseinstellungen wie Nutzername und Passwort sowie der Host und der Name der Datenbank angegeben werden können.

Auf dem Verbindungsobjekt rufen Sie anschließend die Methode `connect()` auf. Die Callback-Funktion, die Sie hierbei als Parameter übergeben, wird aufgerufen, sobald die Verbindung hergestellt wurde oder beim Herstellen der Verbindung ein Fehler auftrat. Ist Letzteres der Fall, haben Sie über den Parameter `error` die Möglichkeit, auf detaillierte Informationen zu dem jeweiligen Fehler zuzugreifen.

Um die Verbindung zu der Datenbank wieder zu trennen, verwenden Sie die Methode `end()`.

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'root',
  password : '1234',
  database : 'sakila'
});
connection.connect((error) => {
  if (error) {
    console.error(error);
  } else {
    console.log('Connected');
  }
});
connection.end();
```

**Listing 7.2** Herstellen einer Verbindung zu MySQL

### Lesen von Datensätzen

Für das Lesen von Datensätzen und generell das Ausführen von *Datenbankanfragen* (*Queries*) stellt das Verbindungsobjekt (hier: `connection`) die Methode `query()` zur Verfügung (Listing 7.3). Diese Methode kann mit einer unterschiedlichen Anzahl an Parametern aufgerufen werden: Bei zwei Parametern bezeichnet der erste Parameter die Datenbankanfrage und der zweite Parameter die Callback-Funktion, die aufgerufen wird, sobald die Anfrage abgeschlossen wurde. Optional lässt sich zwischen diesen beiden Parametern aber noch ein dritter Parameter verwenden, über den spezielle Werte definiert werden können, die im Rahmen der Datenbankanfrage benötigt werden, bspw. die einzusetzenden Werte für Platzhalter (ein Beispiel hierzu folgt weiter unten im nächsten Abschnitt beim Erstellen von Datensätzen).

Beim Lesen von Datensätzen sind lediglich zwei Parameter zu übergeben: zum einen die Datenbankanfrage (in diesem Fall eine `SELECT`-Anfrage), zum anderen die Callback-Funktion. Die gefundenen Datensätze sind in entsprechender Callback-Funktion in Form eines Arrays als Parameter `rows` aufgeführt, über das sich mit `forEach()` iterieren lässt. Jedes in diesem Array enthaltene Objekt hat dabei Eigenschaften, die den Spaltennamen der entsprechenden Tabelle entsprechen (im Beispiel sind dies `actor_id`, `first_name`, `last_name` und `last_update`; siehe auch die Kommandozeilenausgabe des Programms weiter unten). Möchten Sie innerhalb Ihres JavaScript-Codes aber mit anders benannten Eigenschaften arbeiten (bspw. in üblicher Camel-Case-Schreibweise), können Sie, wie in Listing 7.3 zu sehen, dank Objekt-Destructuring relativ einfach ein entsprechendes Objekt erzeugen.

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '1234',
  database: 'sakila'
});
connection.connect(error => {
  if (error) {
    console.error(error);
  } else {
    console.log('Connected');
  }
});
const QUERY = 'SELECT * FROM actor';
connection.query(QUERY, (error, rows, fields) => {
  if (error) {
    console.error(error);
  }
  console.log(rows);
  rows.forEach(row => {
    const { first_name: firstName, last_name: lastName } = row;
    // ...
  });
});
connection.end();
```





**Listing 7.3** Lesen von Datensätzen



Die Ausgabe des Programms lautet wie folgt:

```
[ {
    actor_id: 1,
    first_name: 'PENELOPE',
    last_name: 'GUINESS',
    last_update: 2006-02-15T03:34:33.000Z },
  {
    actor_id: 2,
    first_name: 'NICK',
    last_name: 'WAHLBERG',
    last_update: 2006-02-15T03:34:33.000Z },
  {
    actor_id: 3,
    first_name: 'ED',
    last_name: 'CHASE',
    last_update: 2006-02-15T03:34:33.000Z },
  {
    actor_id: 4,
    first_name: 'JENNIFER',
    last_name: 'DAVIS',
    last_update: 2006-02-15T03:34:33.000Z },
  {
    actor_id: 5,
    first_name: 'JOHNNY',
    last_name: 'LOLLOBRIGIDA',
    last_update: 2006-02-15T03:34:33.000Z },
  ...
]
```

### Erstellen von Datensätzen

Um einen Datensatz zu erstellen, übergeben Sie der Methode `query()`, wie in Listing 7.4 gezeigt, eine entsprechende `INSERT`-Query (`INSERT INTO actor SET ?`). Als zweiten

Parameter übergeben Sie außerdem ein Objekt, dessen Eigenschaften entsprechend für die gleichnamigen Datenbankfelder eingesetzt werden und das intern für den Platzhalter in der Anfrage verwendet wird. Der Vorteil: »mysql« nimmt Ihnen die Arbeit ab und bildet automatisch die Eigenschaften des Objekts auf die passenden Datenbankfelder ab. Darüber hinaus verhindert es durch *Escaping*, dass die Anfragen syntaktisch nicht korrekt sind.

Wurde der Datensatz erfolgreich erstellt, erhält das `result`-Objekt in der Callback-Methode die entsprechende ID des neuen Datensatzes über die Eigenschaft `insertId` (im Beispiel der Wert 201, weil in der Beispieldatenbank zuvor genau 200 Datensätze in der Datenbank waren).

```
const mysql = require('mysql');
// Erstellen der Verbindung wie zuvor
// ...
const QUERY = 'INSERT INTO actor SET ?';
const actor = {
  first_name: 'MAX',
  last_name: 'MUSTERMANN',
  last_update: new Date()
};
connection.query(QUERY, actor, (error, result) => {
  if (error) {
    console.error(error);
  }
});
connection.end();
```

**Listing 7.4** Erstellen von Datensätzen

### Aktualisieren von Datensätzen

Das Aktualisieren von Datensätzen funktioniert in MySQL über die `UPDATE`-Query. Über das `?`-Zeichen können Sie dabei wieder Platzhalter definieren. Die konkret einzusetzenden Werte übergeben Sie der Methode `query()` in Form eines Arrays als zweiten Parameter. In Listing 7.5 wird auf diese Weise bspw. eine `UPDATE`-Query definiert, über die der Vorname des Schauspielers mit der ID 201 auf den Wert »MORITZ« gesetzt wird.

Da von einer `UPDATE`-Query prinzipiell auch mehrere Datensätze betroffen sein können, enthält das `result`-Objekt in der Callback-Funktion die Eigenschaft `changedRows`, über die genau die Anzahl an geänderten Datensätzen ermittelt werden kann.

```
const mysql = require('mysql');
// Erstellen der Verbindung wie zuvor
// ...
```





```
const QUERY = 'UPDATE actor SET first_name = ? WHERE actor_id = ?';
const updateData = ['MORITZ', 201];
connection.query(QUERY, updateData, (error, result) => {
  if (error) {
    console.error(error);
  }
  console.log(`Changed ${result.changedRows} row(s)`);
});
connection.end();
```

**Listing 7.5** Aktualisieren von Datensätzen

### Löschen von Datensätzen

Für das Löschen von Datensätzen verwenden Sie die `DELETE`-Query. Eventuelle Platzhalter in der Query können Sie wie gewohnt in Form eines Arrays als zweiten Parameter für die `query()`-Methode übergeben. In Listing 7.6 bspw. wird der Datensatz mit der ID 201 aus der Datenbank gelöscht. Da auch beim Löschen von Datensätzen mehrere Datensätze betroffen sein können, lässt sich die genaue Anzahl in der Callback-Funktion über die Eigenschaft `affectedRows` des `result`-Objekts ermitteln.

```
const mysql = require('mysql');
// Erstellen der Verbindung wie zuvor
// ...
const QUERY = 'DELETE FROM actor WHERE actor_id = ?';
const deleteData = [201];
connection.query(QUERY, deleteData, (error, result) => {
  if (error) {
    console.error(error);
  }
  console.log(`Deleted ${result.affectedRows} row(s)`);
});
connection.end();
```

**Listing 7.6** Löschen von Datensätzen

## 7.1.2 Ausblick

In diesem Rezept haben Sie gesehen, wie Sie mithilfe des »mqysql«-Packages auf MySQL-Datenbanken zugreifen und Datensätze erstellen, lesen, aktualisieren und löschen können (mit anderen Worten: wie Sie die CRUD-Operationen durchführen können). Im nächsten Rezept zeige ich Ihnen, wie Sie Gleiches in Verbindung mit einer PostgreSQL-Datenbank machen. Im daran anschließenden Rezept 50 werde ich Ihnen schließlich zeigen, wie Sie den Zugriff auf relationale Datenbanken über sogenanntes *objektrelationales Mapping* vereinfachen.

# Kapitel 10
# Testing und TypeScript

*In diesem Kapitel stelle ich Ihnen zwei Ansätze vor, mit deren Hilfe Sie Fehler im Code vermeiden. Über Unit-Tests sichern Sie Ihren Code durch automatisierte Tests ab. Mithilfe von TypeScript machen Sie Ihren Code typsicherer.*



In diesem Kapitel möchte ich Ihnen verschiedene Rezepte vorstellen, die Ihnen dabei helfen, Ihren Code robuster gegen Fehler zu machen. Dies ist zum einen das automatisierte Testen von Node.js-Applikationen (Rezepte 74 bis 77) und zum anderen die Verwendung von TypeScript (Rezepte 78 und 79).

- Rezept 74: Unit-Tests schreiben
- Rezept 75: Unit-Tests automatisch neu ausführen
- Rezept 76: Die Testabdeckung ermitteln
- Rezept 77: Unit-Tests für REST-APIs implementieren
- Rezept 78: Eine Node.js-Applikation in TypeScript implementieren
- Rezept 79: TypeScript-basierte Applikationen automatisch neu kompilieren

## 10.1 Rezept 74: Unit-Tests schreiben

Sie möchten einen Unit-Test erstellen, um die Funktionalität Ihrer Anwendung automatisiert testen zu können.

### 10.1.1 Exkurs: Unit-Tests und testgetriebene Entwicklung

Wenn Sie professionell Software entwickeln, führt kein Weg daran vorbei, den Code automatisiert zu testen. Über sogenannte *Unit-Tests* (auch *Modultests*) stellen Sie sicher, dass der eigentliche Anwendungs-Code, den Sie programmieren, auch so funktioniert, wie Sie sich das vorstellen. Zugleich beugen Sie durch Unit-Tests vor, dass bei Änderungen am Anwendungs-Code (bspw. bei vermeintlichen Bug-Fixes) nicht versehentlich neue Bugs in den Code gelangen.

Oft wird im Zusammenhang mit Unit-Tests auch der Begriff der *testgetriebenen Entwicklung* genannt (kurz *TDD* für *Test Driven Development*). Ihren Ursprung hat die

testgetriebene Entwicklung im sogenannten *Extreme Programming*, einer Methode der *agilen Softwareentwicklung*.

Die grundlegende Idee von TDD ist es, bei der Entwicklung iterativ vorzugehen: Bevor Sie mit der Implementierung einer neuen Komponente beginnen, legen Sie zunächst über Unit-Tests die Anforderungen an diese Komponente fest.

Dieses Vorgehen ist auch unter dem Begriff *Red, Green, Refactor* bekannt und besteht aus folgenden drei gleichnamigen Phasen (siehe auch Abbildung 10.1):

1. In der *Red-Phase* definiert man, was genau entwickelt bzw. welche Funktionalität einer Komponente entwickelt werden soll. Diese Anforderungen schreibt man in Form eines Unit-Tests nieder, der in dieser Phase zunächst fehlschlägt (den Namen hat diese Phase aufgrund der Tatsache, dass fehlschlagende Unit-Tests üblicherweise durch die Farbe Rot gekennzeichnet werden).
2. In der *Green-Phase* implementiert man nun die entsprechende Komponente, sodass sie die im Test definierten Anforderungen erfüllt und der Test nicht mehr fehlschlägt (nicht fehlschlagende Tests werden in der Regel mit der Farbe Grün gekennzeichnet, daher der Name dieser Phase).
3. In der *Refactor-Phase* hat man die Möglichkeit, die Implementierung der Komponente zu optimieren, gegebenenfalls unter Verwendung der Refactoring-Techniken, die Martin Fowler in seinem Buch »Refactoring: Improving the Design of Existing Code« beschreibt.

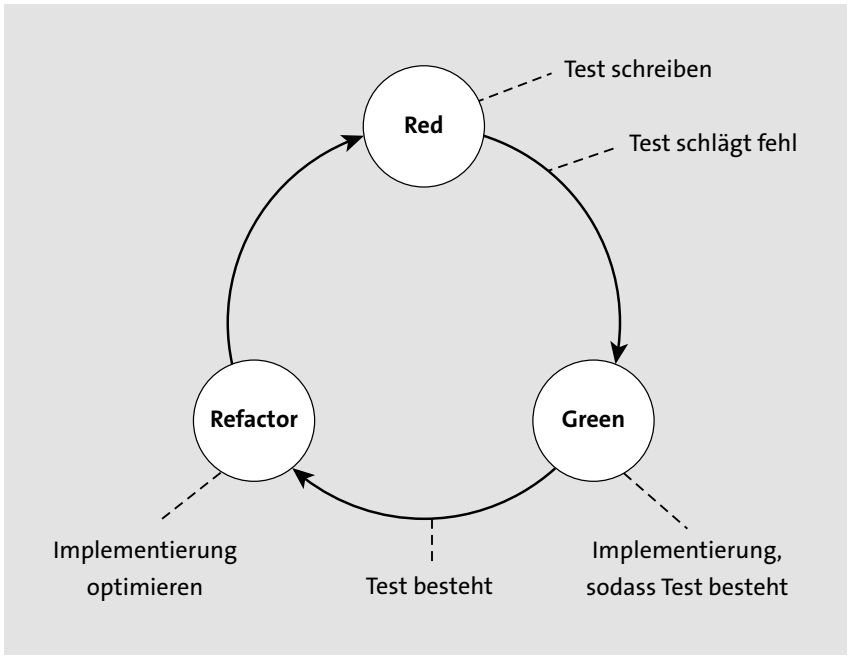


**Abbildung 10.1** Workflow der testgetriebenen Entwicklung





**Hinweis**

Das Buch »Refactoring: Improving the Design of Existing Code« von Martin Fowler gilt als einer der Klassiker der Softwareliteratur. Interessante Randnotiz: Während die erste Ausgabe von 1999 die Refactoring-Techniken noch in Java beschreibt, setzt die zweite Ausgabe von 2018 vollständig auf JavaScript. Ein weiterer Beleg dafür, wie wichtig die Sprache JavaScript in den vergangenen Jahren geworden ist.

Ein einzelner Unit-Test wiederum ist aus einem oder mehreren *Testfällen* (*Test Cases*) aufgebaut. Testfälle können außerdem über sogenannte *Test-Suites* zusammengefasst werden. Innerhalb eines Test Cases formulieren Sie über sogenannte *Assertions* die Anforderungen an die zu implementierende Komponente. Hierüber können Sie bestimmte Aspekte prüfen, bspw., welches Ergebnis eine Funktion für gegebene Parameter zurückgeben soll. Genauer gesagt, bestehen einzelne Test Cases aus drei verschiedenen Phasen: der *Arrange-Phase*, der *Act-Phase* und der *Assert-Phase* (auf diese Phasen komme ich gleich in dem Praxisbeispiel noch mal zu sprechen).



#### Test-Frameworks für Node.js

Für die Implementierung von Tests unter Node.js steht Ihnen eine ganze Reihe von Tools und Frameworks zur Verfügung. Zu den bekanntesten zählen »Jest« (*https://jestjs.io/*), »mocha« (*https://mochajs.org/*) »Jasmine« (*https://jasmine.github.io/*), »node-tap« (*http://www.node-tap.org/*), AVA (*https://github.com/avajs/ava*) und »tape« (*https://github.com/substack/tape*).

In der Praxis verwende ich persönlich in den meisten Fällen »Jest«, weil es zum einen sehr schnell ist und zum anderen auch Support für die Ermittlung der Testabdeckung mit sich bringt (siehe auch Rezept 76). Da ich außerdem frontendseitig hauptsächlich mit »React« arbeite (»React« und »Jest« werden beide von Facebook entwickelt), gehört »Jest« quasi sowieso zu meinem Werkzeugkasten.

### 10.1.2 Lösung: Unit-Test mit »Jest« schreiben

Um »Jest« in Aktion zu sehen, legen Sie ein Beispielprojekt an, und installieren Sie das Package als Entwicklungsabhängigkeit:

```
$ mkdir testing-example
$ cd testing-example
$ npm init -y
$ npm i -D jest
```

Legen Sie außerdem ein separates Verzeichnis für die Testdateien an. Üblicherweise sollte sich das Verzeichnis auf gleicher Ebene wie das *src*-Verzeichnis befinden und den Namen *test* verwenden:

```
$ mkdir test
$ mkdir src
```

**Speicherort von Testdateien**

Bezüglich des Speicherorts von Testdateien sind vor allem zwei Ansätze populär. Der eine Ansatz sieht – wie oben beschrieben – vor, dass die Tests in einem separaten Verzeichnis gespeichert werden und dabei die Verzeichnisstruktur von dem entsprechenden Quelltextverzeichnis eingehalten wird. Ein Test für eine Datei *ConnectionManager.js*, die in dem Verzeichnis *src/services/connection* liegt, würde also in dem Verzeichnis *test/services/connection* gespeichert. Der zweite Ansatz dagegen sieht vor, die Testdateien in dem gleichen Verzeichnis wie die jeweils getestete Datei zu speichern. Im obigen Beispiel würde die Testdatei also in das Verzeichnis *src/service/connection* gelegt.

### Einen Test erstellen

Wenn Sie so diszipliniert sind und strikt der testgetriebenen Entwicklung folgen, müssen Sie *vor* der Implementierung einer neuen Komponente den Unit-Test schreiben (zugegeben, auch ich mache das nicht immer). Wenn Sie also bspw. die Aufgabe haben, eine Klasse zu implementieren, über die Nutzer verwaltet werden können, könnte ein Unit-Test in »Jest« etwa wie in Listing 10.1 aussehen.

```
const UserRepository = require('../src/UserRepository');

describe('UserRepository', () => {
  describe('#add()', () => {
    it('should add the user and increase the numer of all users', () => {
      const userRepository = new UserRepository();

      userRepository.add({ name: 'Max' });
      userRepository.add({ name: 'Moritz' });

      expect(userRepository.getAll().length).toBe(2);
    });
    it('should add the user only if it is not already there', () => {
      const userRepository = new UserRepository();

      userRepository.add({ name: 'Max' });
      userRepository.add({ name: 'Max' });

      expect(userRepository.getAll().length).toBe(1);
    });
  });
```





```
  describe('#clearAll()', () => {
    it('should clear all users', () => {
      const userRepository = new UserRepository();
      userRepository.add({ name: 'Moritz' });
      expect(userRepository.getAll().length).toBe(1);

      userRepository.clearAll();

      expect(userRepository.getAll().length).toBe(0);
    });
  });
});
```

**Listing 10.1** Unit-Test in »Jest«



Über die von »Jest« zur Verfügung gestellte Funktion `describe()` können Sie zusammengehörige Tests sinnvoll zu Test Suites zusammenfassen (beim späteren Ausführen der Tests werden entsprechende Tests zusammen gruppiert und entsprechend eingerückt). Gruppierungen können dabei beliebig geschachtelt werden, was hilfreich ist, um eine gewisse Struktur in die definierten Tests zu bekommen.

Einzelne Tests Cases dagegen definieren Sie über die Methode `it()`. Hierbei übergeben Sie eine entsprechende Beschreibung des Tests in Form einer Zeichenkette und den eigentlichen Test in Form einer Funktion.

Prinzipiell bestehen Test Cases, wie eingangs erwähnt, aus drei Phasen, auch bezeichnet als *Arrange*, *Act*, *Assert* (kurz *AAA*). In der Arrange-Phase werden Initialisierungen durchgeführt und das Test-Setup aufgebaut, bspw.:

```
it('should add the user and increase the numer of all users', () => {
  // 1.) Arrange-Phase
  const userRepository = new UserRepository();
  // ...
});
```

In der Act-Phase werden die zu testenden Methoden aufgerufen:

```
it('should add the user and increase the numer of all users', () => {
  // ...
  // 2.) Act-Phase
  userRepository.add({ name: 'Max' });
  userRepository.add({ name: 'Moritz' });
});
```

In der Assert-Phase wird schließlich das zu erwartende Ergebnis bzw. der zu erwartende Zustand überprüft:

```
it('should add the user and increase the numer of all users', () => {
  // ...
  // 3.) Assert-Phase
  expect(userRepository.getAll().length).toBe(2);
});
```

Insgesamt werden auf diese Weise im Beispiel drei Tests definiert:

- **Test 1**: Die Klasse `UserRepository` soll über eine Methode `add()` verfügen, über die Nutzerobjekte hinzugefügt werden können. Über die Methode `getAll()` soll eine Liste dieser Nutzer zurückgegeben werden. Fügt man über die Methode `add()` zwei Nutzerobjekte hinzu, soll `getAll()` genau diese Objekte als Liste zurückgeben.
- **Test 2**: Die Methode `add()` soll doppelte Nutzer ignorieren. Fügt man über die Methode `add()` zweimal einen Nutzer mit gleichem Namen hinzu, soll die Liste anschließend nur einmal den Nutzer enthalten.
- **Test 3**: Über die Methode `clearAll()` soll es möglich sein, alle Nutzer aus dem Repository zu löschen.

**Tipp**

Lassen Sie zwischen den einzelnen Phasen Arrange, Act und Assert jeweils eine Leerzeile. Auf diese Weise können Sie beim späteren Durchsehen der Tests schneller zuordnen, welche Code-Zeile zu welcher Phase gehört.

**Hinweis**

Übrigens müssen Sie »Jest«, wie Sie in Listing 10.1 sehen konnten, nicht explizit als Abhängigkeit über `require()` einbinden, um die Funktionen `describe()`, `it()` und `expect()` nutzen zu können. Beim Ausführen von »Jest« werden diese Bibliotheksfunktionen implizit geladen und stehen daher innerhalb des Tests zur Verfügung.

Speichern Sie den Test in einer Datei *UserRepository.test.js* in dem eben angelegten Verzeichnis *test*.

### Einen Test ausführen

Fügen Sie nun in das *src*-Verzeichnis die Klasse `UserRepository` als Datei *UserRepository.js* hinzu, und ergänzen Sie folgenden Code (die konkrete Implementierung der Klasse bleibt dabei zunächst noch bewusst leer):

```
module.exports = class UserRepository {};
```

Um nun den oben erstellten Test mit »Jest« auszuführen, verwenden Sie folgenden Befehl (dazu muss »Jest« global installiert sein):





```
$ jest -verbose
```

Oder alternativ – wenn Sie wie ich npx nutzen – folgenden Befehl:

```
$ npx jest --verbose
```

Die Angabe `--verbose` sorgt dabei dafür, dass die Konsolenausgabe von »Jest« etwas umfangreicher ist und mehr Informationen zu den ausgeführten Tests liefert. Wie Sie anhand dieser Konsolenausgabe in Listing 10.2 sehen, schlagen momentan alle Tests fehl. Das ist nicht weiter verwunderlich, denn es gibt ja auch noch keine Implementierung der Klasse `UserRepository` bzw. keine Methode `add()`. Sie befinden sich also gerade in der Red-Phase.

```
FAIL  test/UserRepository.test.js
  UserRepository
    #add()
      × should add the user and increase the numer of all users (12ms)
      × should add the user only if it is not already there (1ms)
    #clearAll()
      × should clear all users (1ms)

  ● UserRepository › #add() › should add the user and increase the numer
                      #of all users

    TypeError: userRepository.add is not a function

       5 |   it('should add the user and increase the numer of all users', () => {
       6 |     const userRepository = new UserRepository();
       7 |
    >  8 |     userRepository.add({ name: 'Max' });
       9 |     userRepository.add({ name: 'Moritz' });
      10 |
      11 |     expect(userRepository.getAll().length).toBe(2);
      12 |   });

      at Object.add (test/UserRepository.test.js:7:22)
...
Test Suites: 1 failed, 1 total
Tests:       3 failed, 3 total
Snapshots:   0 total
Time:        2.166s
Ran all test suites.
```

**Listing 10.2** Ausgabe der fehlgeschlagenen Tests

Erweitern Sie daher – im Rahmen der Green-Phase – die Klasse `UserRepository` wie folgt um die Methoden `add()`, `contains()`, `getAll()` und `clearAll()` (die alle selbsterklärend sein dürften, weswegen ich auf eine detaillierte Beschreibung an dieser Stelle verzichte).

```
module.exports = class UserRepository {
  constructor() {
    this.users = [];
  }

  add(user) {
    if (!this.contains(user)) {
      if (user && user.name) {
        this.users.push(user);
      } else {
        throw new Error('Wrong user format.');
      }
    }
  }

  contains(newUser) {
    return this.users.filter((user) => user.name === newUser.name).length > 0;
  }

  getAll(user) {
    return this.users;
  }

  clearAll() {
    this.users = [];
  }
};
```

**Listing 10.3** Implementierung der »UserRepository«-Klasse

Führen Sie nun die Tests erneut aus, schlägt keiner der definierten Tests fehl:

```
$ npx jest --verbose
 PASS  test/UserRepository.test.js
  UserRepository
    #add()
      ✓ should add the user and increase the numer of all users (5ms)
      ✓ should add the user only if it is not already there (1ms)
```





```
    #clearAll()
      ✓ should clear all users (2ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.785s
Ran all test suites.
```

**Tipp**

Wenn Sie den Testbefehl unter dem Skript `test` in die *package.json*-Konfiguration übernehmen, lassen sich die Tests anschließend sowohl über `npm run test` als auch über die Kurzform `npm test` ausführen.

```
{
  "name": "testing-example",
  "version": "1.0.0",
  "main": "./src/start.js",
  "scripts": {
    "test": "npx jest --verbose"
  },
  "keywords": [
    "javascript",
    "nodejs"
  ],
  "author": "Philip Ackermann",
  "license": "MIT",
  "devDependencies": {
    "jest": "^23.6.0"
  }
}
```

### 10.1.3 Ausblick

Sie haben jetzt in aller Kürze gesehen, welche Vorteile das Implementieren von Unit-Tests mit sich bringt und wie Sie Unit-Tests unter Node.js erstellen. Damit sind die Voraussetzungen für die nächsten beiden Rezepte geschaffen, in denen ich Ihnen zeige, wie Sie den Workflow rund um das Testen optimieren, indem Sie Tests während der Entwicklung automatisch neu ausführen (Rezept 75) und ermitteln, welche Stellen Ihres Applikations-Codes durch Tests abgedeckt werden und welche noch nicht (Rezept 76).