# Inhaltsübersicht

# Node.js



- Node.js ist eine Open-Source Laufzeitumgebung für Server-seitigen JavaScript-Code
  - Entwickelt von Ryan Dahl (erstes Release: 27. Mai 2009)
  - https://nodejs.org
- Hauptmerkmale
  - V8 JavaScript-Engine (Open Source)
  - Event-Driven, Non-Blocking I/O-Modell
  - Modular Ansatz (CommonJS) mit einigen Kernmodulen
  - Installation von Drittanbieter-Modulen via npm, dem Paketmanager für Node.js
  - Interaktives Ausführen von JavaScript-Code (REPL)

# Node.js
## Kommandozeilen-Programm



- Starten einer Anwendung

```
$ node myapp
```

```
$ node myapp.js
```

  - Führt die JavaScript-Datei `myapp.js` als Hauptmodul der Anwendung aus

- Aufruf der REPL (Read-Eval-Print-Loop)

```
$ node
```

  - In der REPL kann interaktiv JavaScript-Code ausgeführt werden
  - Befehle (Auswahl)
    - Laden und Ausführen eines Skripts: `.load script.js`
    - Speichern der REPL Session: `.save save.js`
    - Beenden der REPL: `.exit`

# Module



- Eine Node.js Anwendung besteht aus einem oder mehreren Modulen, zwischen denen Abhängigkeitsbeziehungen bestehen
  - Jede JavaScript-Datei der Anwendung ist ein Modul
  - Die beim Start der Anwendung angegebene Datei ist das **Hauptmodul**

```
$ node app.js
```

- Kernmodule (*core modules*)
  - Node.js wird mit einigen Kernmodulen ausgeliefert, die daher nicht über npm installiert werden müssen, u. a.:

```
console, events, fs, http, https, module, os, path
```

# Module

## Modul Wrapper-Funktion



- Ein Modul wird innerhalb einer Wrapper-Funktion ausgeführt

```
(function (exports, require, module, __filename, __dirname) {
  // Quellcode des Moduls
});
```

| Parameter | Erläuterung |
|---|---|
| `exports` | Verweist auf `module.exports` (ist initial ein leeres Objekt) |
| `require` | Funktion zur Definition einer Modulabhängigkeit |
| `module` | Objekt, das das aktuelle Modul repräsentiert |
| `__filename, __dirname` | Dateiname bzw. Verzeichnisname des Moduls |

- Damit sind alle Deklarationen innerhalb des Moduls lokal
- Zudem hat das Modul Zugriff auf modulspezifische Variablen

# Module

## Beziehungen zwischen Modulen



- Ein Modul exportiert den Inhalt von `module.exports` an alle Module, die von dem Modul abhängig sind
- Über den Aufruf der Funktion `require` definiert ein Modul die Abhängigkeit von einem anderen Modul
  - Rückgabewert von `require` ist der Export des Moduls

```
const PI = Math.PI;
exports.area = (r) => PI * r * r;
```

oder

```
const PI = Math.PI;
module.exports = {
  area: (r) => PI * r * r
}
```

Datei circle.js

```
const circle = require('./circle');
const a = circle.area(4);
```

Datei main.js

# Module

## Laden von Modulen



- Über `require` wird ein Modul geladen
  - Modulalternativen: JavaScript-Datei, JSON-Datei oder binäres Addon
- Alternativen für das Argument
  - Argument ist der Name eines Kernmoduls, wie z. B. `'html'`
  - Argument ist ein absoluter oder relativer Pfad zu einer Datei oder einem Verzeichnis (beginnt mit `'./'`, `'../'` oder `'/'`)
  - Argument referenziert ein Drittanbieter-Modul
- Ein Modul wird nur einmal geladen (beim ersten `require`-Aufruf)
  - Quellcode wird nur einmal ausgeführt
  - Rückgabewert von `require` ist bei allen abhängigen Modulen gleich
    - Wenn `module.exports` ein Objekt ist, dann erhalten alle dasselbe Objekt

# Module

## Laden von Modulen: Pfad



- Bei `require(X)`, wobei `X` ein Pfad ist, werden folgende Dateien nacheinander gesucht (relativ zum Verzeichnis des abhängigen Moduls)

| Datei | Aktion |
|---|---|
| `X, X.js` | Datei wird als JavaScript-Datei geladen |
| `X.json` | Datei wird als JSON-Datei geladen und geparst. Rückgabewert von `require(X)` ist ein JavaScript-Objekt. |
| `X.node` | Datei wird als binäres Addon geladen |
| `X/package.json` | Die Datei, auf die die Eigenschaft `main` der JSON-Datei verweist, wird geladen |
| `X/index.js` | Datei wird als JavaScript-Datei geladen |
| `X/index.json` | Datei wird als JSON-Datei geladen und geparst (s. o.) |
| `X/index.node` | Datei wird als binäres Addon geladen |

Für die **erste Datei**, die existiert, wird die entsprechende Aktion ausgeführt.

# Module

## Laden von Modulen: Pfad



- Beispiel: `X/index.js`

```
const circle = require('./circle');
```

Datei main.js

```
const PI = Math.PI;
exports.area = (r) => PI * r * r;
```

Datei index.js

```
main.js
circle
  index.js
```

# Module

## Laden von Modulen: Pfad



- Beispiel: `X/package.json`

```
const circle = require('./circle');
```

Datei main.js

```
{
  "name" : "circle",
  "main" : "./lib/circle.js"
}
```

Datei package.js

```
const PI = Math.PI;
exports.area = (r) => PI * r * r;
```

Datei circle.js

```
main.js
circle
  package.json
  lib
    circle.js
```

# Module

## Laden von Modulen: Drittanbieter-Modul



- Beispiel

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

Datei main.js

```
const PI = Math.PI;
exports.area = (r) => PI * r * r;
```

Datei index.js

```
folderA
  folderB
    main.js
node_modules
  circle
    index.js
```

- Ausgehend vom Verzeichnis des abhängigen Moduls wird jedes Verzeichnis auf dem Weg zum Wurzelverzeichnis überprüft, bis ein `node_modules`-Unterverzeichnis Y gefunden wird, für das gilt:
  - Bei `require(X)` kann X relativ zu Y als Modul geladen werden (X wird dabei als relativer Pfad betrachtet)

# Node.js-Standardobjekte

## global



- Das globale Objekt in Node.js heißt `global`
- Es stellt einen Namensraum für globale Werte bereit
- Auf die globalen Werte kann an jeder Stelle ohne Umweg über das globale Objekt direkt zugegriffen werden
  - z. B. `Math` statt `global.Math`
- Alle JavaScript-Standardobjekte (z. B. `Math`, `Date`, `Array`) sowie alle Node.js-Standardobjekte (z. B. `console`, `process`) sind über dieses globale Objekt erreichbar

# Node.js-Standardobjekte

## console



- Ausgabe über die Standardausgabe (`stdout`)

```
console.log('Hello World');
```

```
console.log(obj1, obj2);
```

  - Es können ein oder mehrere Werte angegeben werden
  - Bei einem Objekt wird über die Funktion `inspect` des Kernmoduls `util` eine String-Repräsentation des Objekts erzeugt

- Ausgabe über die Standardfehlerausgabe (`stderr`)

```
console.error('Fehler!');
```

```
console.log(errorObject);
```

- Ausgabe inkl. Stacktrace über die Standardausgabe

```
console.trace('message');
```

# Node.js-Standardobjekte

## console



- Ausgabe der Dauer der Ausführung eines Code-Abschnitts

```
console.time('my-code');
// Code, dessen Ausführungsdauer zu messen ist
console.timeEnd('my-code'); // Ausgabe: my-code: 0.458ms
```

  - Die Funktion `time` startet einen Timer mit der angegebenen Bezeichnung
  - Die Funktion `timeEnd` beendet den Timer mit der angegebenen Bezeichnung und gibt das Ergebnis über die Standardausgabe aus

- Anmerkung
  - Es können auch `Console`-Objekte mit eigenen Streams (für `log` und `error`) erzeugt werden, um z. B. in Dateien zu loggen (siehe Kernmodul `console`)

# Node.js-Standardobjekte

## Funktionen zur verzögerten Ausführung



- Eine Funktion einmalig zeitverzögert ausführen

```
const timeoutObj = setTimeout(param => console.log(param), 1000,
                              'Argument');
```

  - Funktion wird nach der angegebenen Zeit (in Millisekunden) aufgerufen
  - Dieser Funktion können beliebig viele Argumente mitgegeben werden

- Eine Funktion in bestimmten Zeitintervallen aufrufen

```
const intervalObj = setInterval(param => console.log(param), 1000,
                                'Argument');
```

  - Parameter wie bei `setTimeout`

- Timer abbrechen

```
clearTimeout(timeoutObj);
```

```
clearInterval(intervalObj);
```

# Node.js-Standardobjekte

## Funktionen zur verzögerten Ausführung



- Eine Funktion „als nächstes“ ausführen

```
const immediateObj = setImmediate(param => console.log(param),
                                  'Argument');
```

  - Funktion wird als nächstes, d. h. nach I/O-Event Callbacks, jedoch vor den Timer-Callbacks (`setTimer`, `setInterval`) ausgeführt

- Abbrechen der Funktionsausführung

```
clearImmediate(immediateObj);
```

# Node.js-Standardobjekte

## process



- Zugriff auf die Kommandozeilen-Argumente

```
const argArray = process.argv;
```

  - Das erste Element ist der Pfad zu node, das zweite der Pfad zur JavaScript-Datei und alle weiteren die beim Aufruf übergebenen Argumente

- Ermitteln des aktuellen Arbeitsverzeichnisses

```
const dir = process.cwd();
```

- Zugriff auf die Umgebungsvariablen

```
const userHome = process.env.HOME; // '/Users/mustermann'
```

  - Jede Umgebungsvariable ist eine Eigenschaft des Objekts `process.env`, wobei der Eigenschaftswert immer ein String ist
  - Schreibender Zugriff wirkt sich nur innerhalb des Prozesses aus

# Node.js-Standardobjekte

## process



- Beenden des Prozesses

```
process.exit(-1); // Exit-Code ist -1
```

- Die Ausführungsdauer eines Code-Blocks messen (hochauflösend)

```
const start = process.hrtime();
// Code, dessen Ausführungsdauer zu messen ist
const dauer = process.hrtime(start);
console.log(`${dauer[0]} Sekunden, ${dauer[1]} Nanosekunden }`)
```

  - `hrtime()` ohne Argument liefert die Differenz zu einem willkürlichen Zeitpunkt in der Vergangenheit; mit Argument liefert `hrtime()` die Differenz zum angegebenen Zeitpunkt (jeweils als Array mit Sekunden und Nanosekunden)

- ID des Prozesses ermitteln

```
const pid = id; // z. B. 847
```

# Mikroübung

## Richtig oder falsch?



- Mit Node.js kann JavaScript-Code auf Serverseite ausgeführt werden — ja
- Standardobjekte wie `Math` und `Date` stehen auch unter Node.js zur Verfügung — ja
- Die folgende Deklaration innerhalb eines Moduls erzeugt bei Node.js eine globale Variable — nein, sie sind nur in dem Modul verfügbar

```
var name = 'Max Mustermann';
```

- Alle Deklarationen eines Moduls stehen den anderen Modulen zur Verfügung — nein, sie müssen dafür exportiert werden

# Kernmodule

## Modul path



- Das Modul `path` bietet Hilfsfunktionen im Kontext von Datei- und Verzeichnispfaden, wie z. B. das Verbinden von Pfadsegmenten

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

const filepath = path.join(__dirname, '..', 'db', 'testdata.json');
// liefert z. B. '/Users/john/express-app/db/testdata.json'
```

- Weitere Hilfsfunktionen (Auswahl)

| Funktion | Erläuterung und Beispiel |
|---|---|
| `basename` | Liefert das letzte Pfadsegment<br>`path.basename('/foo/bar/index.html'); // => 'index.html'` |
| `dirname` | Liefert den Pfad ohne das letzte Pfadsegment<br>`path.dirname('/foo/bar/index.html'); // => '/foo/bar'` |

# Kernmodule

## Modul fs





- Asynchrones Einlesen einer Textdatei

```
const fs = require('fs');
const path = require('path');

fs.readFile(path.join(__dirname, 'foo.txt'), 'utf-8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

- Synchrones Einlesen einer Textdatei nur für Anwendungen mit einem einzigen User ok

```
// ... (wie oben)
const data = fs.readFileSync(path.join(__dirname, 'foo.txt'), 'utf-8');
console.log(data);
```

  - Im Fehlerfall (Datei existiert nicht o.ä.) wird ein Error geworfen

# Kernmodule

## Modul fs



- Asynchrones Schreiben in eine Textdatei

```
const fs = require('fs'); const path = require('path');
const filename = path.join(__dirname, 'bar.txt');

fs.writeFile(filename , 'Hello World!\n', 'utf-8', (err) => {
  if (err) throw err;
  console.log('Erfolgreich gespeichert!');
});
```

- Synchrones Schreiben in eine Textdatei

```
// ... (wie oben)
fs.writeFileSync(filename, 'Hello World!\n', 'utf-8');
```

  - Im Fehlerfall wird ein Error geworfen

# Kernmodule

## Modul http



- Beispiel: Einfacher HTTP-Server

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

const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

# Kernmodule

## Modul events



- EventEmitter erzeugen

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter { }
const myEmitter = new MyEmitter();
```

- Listener hinzufügen

```
myEmitter.on('myEventName', (a, b) => {
  console.log(a, b, this);
});
```

  - Fügt die Callback-Funktion als letzten Listener für Events mit dem angegebenen Namen hinzu (hier: `'myEventName'`)
  - Parameter der Callback-Funktion sind die Argumente, die der EventEmitter beim Emittieren des Events mit übergibt
  - Innerhalb der Callback-Funktion referenziert `this` den EventEmitter

# Kernmodule

## Modul events



- Event ermittieren

```
myEmitter.emit('myEventName', 'Argument 1', 'Argument2');
```

  - Emittiert ein Event mit dem angegebenen Namen (`'myEventName'`) und ruft alle entsprechenden Listener nacheinander auf (synchron)
  - Jeder Listener wird mit den angegebenen Argumenten aufgerufen

- One Time Listener hinzufügen

```
myEmitter.once('myEventName', (a, b) => {
  console.log(a, b, this);
});
```

  - Listener wird als letzter Listener hinzugefügt und nach dem ersten Aufruf automatisch wieder entfernt

# Kernmodule

## Modul events



- Listener entfernen

```
myEmitter.removeListener('myEventName', myListener);
```

  - Entfernt den Listener für das angegebene Event

- Alle Listener entfernen

```
myEmitter.removeAllListener('myEventName');
```

  - Entfernt alle Listener für das angegebene Event
  - Ohne Argument werden alle Listener für alle Events entfernt

- Alle Listener ermitteln

```
const listenerArray = myEmitter.listeners('myEventName');
```

  - Rückgabewert ist eine Kopie des Arrays der Listener

# Kernmodule

## Modul events



- `error`-Events
  - Wenn bei einem `EventEmitter` ein Fehler auftritt, dann emittiert er typischerweise ein Event mit dem Namen `error`
  - Solche `error`-Events werden besonders behandelt: Ist kein Listener dafür eingetragen, dann wird ein Fehler geworfen
- `newListener`-Events
  - Wenn ein neuer Listener hinzugefügt wird, wird ein Event mit dem Namen `newListener` emittiert
  - Argumente: `eventName` und `listener`
- `removeListener`-Events
  - Wird ein Listener entfernt, so wird ein Event mit dem Namen `removeListener` emittiert (Argumente wie bei `newListener`-Events)

# Event Loop
## Einleitung



- Der **Anwendungscode** wird von einem **einzigen Thread** ausgeführt
  - Für die Ausführung des **Systemcodes** wird hingegen auf einen **Thread-Pool** zugegriffen (z. B. für Datenoperationen)
- Ruft der **Anwendungsthread** eine **blockierende I/O-Operation** auf, dann wird diese **synchron** ausgeführt, d. h., der Anwendungsthread wartet auf das Ende der Operation (er wird blockiert)

```
const data = fs.readFileSync(filename, 'utf-8'); // Thread wartet
console.log(data);
```

  - Nachteil: Die Anwendung steht bis zum Abschluss der Operation. Im Falle eines Servers können z. B. keine weiteren Anfragen beantwortet werden.

# Event Loop
## Einleitung



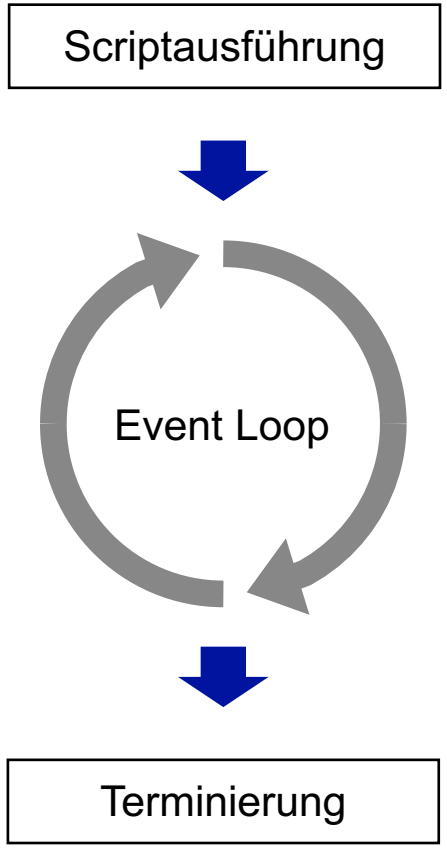
- Ruft der Anwendungsthread eine **nicht-blockierende I/O-Operation** auf, dann wird diese **asynchron** ausgeführt, d. h., der Anwendungsthread kann direkt fortfahren und muss nicht warten

```
fs.readFile(filename, 'utf-8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('Einlesen der Datei angestoßen.');
```

# Event Loop
## Überblick



- Wenn das Ende des auszuführenden Scripts erreicht ist, dann tritt der **Anwendungthread** in eine Endlosschleife, der sog. **Event Loop**
  - Das Script hat dabei i. d. R. **Callbacks** registriert, um z. B. den Code zu definieren, der bei jeder eingehenden HTTP-Anfrage auszuführen ist
  - Die Callbacks werden in der Event Loop ausgeführt, wenn das zugehörige **Ereignis** eingetreten ist (HTTP-Anfrage ist eingegangen, Zeit ist abgelaufen, Datei ist eingelesen etc.)
- Die Event Loop wird solange iteriert, bis keine Callbacks mehr vorhanden sind - erst dann terminiert die Node.js-Anwendung




*Die Event-Loop wird von der C-Bibliothek `libuv` implementiert: http://libuv.org

# Event Loop



- Beispiel 1

```
console.log('Script-Start');
setTimeout(() => { console.log('Callback'); }, 2000);
console.log('Script-Ende');
```

```
Script-Start
Script-Ende
Callback
```

  - Die Anwendung terminiert nach dem Aufruf des Callbacks

- Beispiel 2

```
const http = require('http');
http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
}).listen(8080);
```

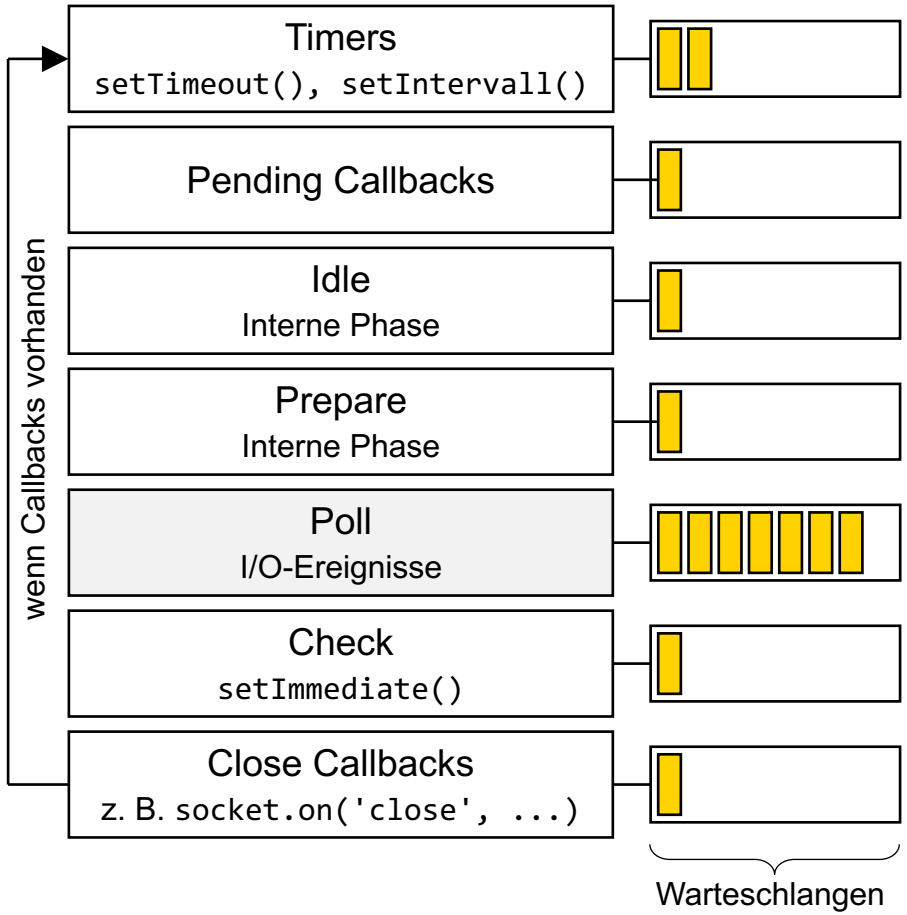
  - Die Anwendung wartet auf eingehende HTTP-Anfragen und terminiert nicht

# Event Loop

## Phasen



- Mit jeder Iteration der Event Loop werden nacheinander mehrere Phasen durchlaufen
- Pro Phase werden phasenspezifische Operationen ausgeführt und anschließend die Callbacks aus der zugehörigen **Warteschlange** synchron aufgerufen
  - Ein Callback gelangt erst dann in die Warteschlange, wenn das zugehörige Ereignis eingetreten ist (Zeit abgelaufen, Datei eingelesen etc.)
  - Die Phase wird beendet, wenn keine Callbacks mehr vorhanden sind, oder eine Obergrenze für die Anzahl an Callbacks erreicht ist
  - Nur in der Poll-Phase wird ggf. **blockierend** gewartet



# npm



- npm ist ein Online-Repository für Pakete (*packages*) sowie der Name des zugehörigen Kommandozeilen-Programms (*npm client*)
  - Webseite: https://www.npmjs.com
  - npm stand ursprünglich für Node Package Manager
- Ein Paket kapselt meist ein oder mehrere Node.js-Module, optional mit Konsolenskripten
  - Es gibt jedoch auch Pakete für den Einsatz im Browser
  - Ein Paket ist letztlich ein Verzeichnis, das mindestens die Datei `package.json` enthält, die das Paket beschreibt
- Für öffentliche Pakete ist die Nutzung des Repositories kostenlos
  - Für private Pakete wird indes eine Gebühr erhoben

# npm

## package.json: Typische Eigenschaften



```
{
  "name": "express-test",
  "version": "1.0.0",
  "author": "Max Mustermann",
  "description": "express-Test",
  "license": "MIT",
  "main": "index.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "express": "~4.13.1"
  },
  "devDependencies": {
    "jasmine-core": "^2.4.1"
  }
}
```

| Eigenschaft | Erläuterung |
|---|---|
| `name*` | Name des Pakets |
| `version*` | Version des Pakets |
| `author` | Autor des Pakets |
| `description` | Kurzbeschreibung des Pakets |
| `licence` | Lizenz des Pakets |
| `scripts` | Kommandozeilen-Skripte |
| `main` | Hauptmodul |
| `dependencies` | Pakete, die zur Laufzeit benötigt werden |
| `devDependencies` | Pakete, die zur Entwicklungszeit benötigt werden |

*Pflichteigenschaft

# npm
## package.json



- Name eines Pakets
  - Darf keine Großbuchstaben und keine Zeichen enthalten, die nicht URL-sicher sind
  - Darf nicht mit einem Punkt oder Unterstrich beginnen
- Scoped Package
  - Beginnt der Name mit einem `@`, dann ist es ein Scoped Package

```
"name": "@scope/my-package"
```

  - Der Scope ist der Teilstring zwischen `@` und `/` (im Beispiel: `scope`)
  - Ein Scope ist wie ein Namensraum für Pakete

# npm
## package.json



- Versionsnummern folgen in der Regel dem **Semantic Versioning***

```
MAJOR.MINOR.PATCH        (z. B. 1.0.0)
```

Grundstruktur einer Versionsnummer

  - Weitere Angaben werden mit - (Prerelease) oder + (Build Metadata) angehängt (z. B. `1.0.0-alpha`, `1.0.0+001`, `1.0.0-beta+exp.sha.5114f85`)

| Release | Erläuterung | Beispiel |
|---|---|---|
| Initial release | Erste stabile Version, die veröffentlicht wird | 1.0.0 |
| Patch release | Bugfixes und kleinere Änderungen; keine neuen Features<br>PATCH Version (dritte Ziffer) wird erhöht | 1.0.**1** |
| Minor release | Neue Features; Abwärtskompatibilität bleibt bewahrt<br>MINOR Version (zweite Ziffer) wird erhöht | 1.**1**.0 |
| Major release | Nicht abwärtskompatible Änderungen<br>MAJOR Version (erste Ziffer) wird erhöht | **2**.0.0 |

*Spezifikation: http://semver.org

# npm
## package.json



- Möglichkeiten zur Angabe der benötigten Version (Auswahl)

| Beispiel | Anforderung an die Version des benötigten Pakets |
|---|---|
| `1.2.3` | genau 1.2.3 |
| `>=1.2.3` | mindestens 1.2.3 |
| `*` | keine Einschränkung |
| `1.2.x` | mindestens 1.2.0 und kleiner als 1.3.0 |
| `1.x` | mindestens 1.0.0 und kleiner als 2.0.0 |
| `~1.2.3` | mindestens 1.2.3 und kleiner als 1.3.0 |
| `~1.2` | mindestens 1.2.0 und kleiner als 1.3.0 (wie `1.2.x`) |
| `~1` | mindestens 1.0.0 und kleiner als 2.0.0 (wie `1.x`) |
| `^1.2.3` | mindestens 1.2.3 und kleiner als 2.0.0 |
| `^0.2.3` | mindestens 0.2.3 und kleiner als 0.3.0 |

# npm
## Kommandozeilen-Programm



- Das Kommandozeilen-Programm `npm` wird zusammen mit der Installation von Node.js installiert
- Ermöglicht vor allem
  - Erzeugen und Upload eigener Pakete
  - Installation von Paketen aus dem Repository (lokal oder global)
  - Aktualisieren der installierten Pakete

# npm
## Installation von Paketen



- Wird ein Node.js-Projekt heruntergeladen, so enthält es typischerweise nicht die benötigten Pakete
  - Das Verzeichnis `node_modules` ist nicht vorhanden
- Anhand der Angaben in der `package.json`-Datei können diese jedoch nachinstalliert werden

```
$ npm install
```

```
$ npm i
```

- Um nur die für die Ausführung der Anwendung benötigten Pakete zu installieren (Pakete aus `devDependencies` werden dann nicht installiert)

```
$ npm install --production
```

# npm

## Installation eines einzelnen Pakets



- Aktuelle Version / bestimmte Version des Pakets lokal installieren

```
$ npm i express
```

```
$ npm i express@4.15.3
```

  - Das Paket wird im Verzeichnis `node_modules` abgelegt (inkl. Abhängigkeiten) und als **Laufzeit-Abhängigkeit** in der `package.json` festgehalten
  - Auch Installation aus lokalem Verzeichnis oder tarball-Datei/-URL möglich

- Lokale Installation als Entwicklungszeit-Abhängigkeit

```
$ npm i jasmine -D
```

```
$ npm install jasmine --save-dev
```

- Globale Installation eines Pakets

```
$ npm install typescript --global
```

aus dem npm Repository

```
$ npm i ./my-package -g
```

aus dem lokalen Verzeichnis

In **package-lock.json** wird zu jedem Paket die installierte Version nebst Hash-Wert abgelegt. Sie ermöglicht eine Reproduzierbarkeit bei unspezifischen Versionsangaben.

# npm
## Kommandozeilen-Programm



- Aktualisieren aller bzw. eines Pakets

```
$ npm update
```

```
$ npm up express
```

  - Wird `--global` mit angegeben, so werden globale Pakete aktualisiert
  - Nicht vorhandene Pakete werden installiert

- Deinstallation eines Pakets

```
$ npm uninstall express
```

```
$ npm un typescript --global
```

- Deinstallation aller nicht in `package.json` aufgeführten Pakete

```
$ npm prune
```

```
$ npm prune --production
```

  - Bei `--production` werden zusätzlich alle Pakete aus `devDependencies` entfernt

# npm
## Kommandozeilen-Programm



- Erzeugen einer initialen `package.json`-Datei (bei `--yes` ohne Fragen)

```
$ npm init
```

```
$ npm init --yes
```

- Ausführen von Skripten mit Standardnamen (Auswahl)

```
$ npm start
```

```
$ npm stop
```

```
$ npm restart
```

```
$ npm test
```

  - Z. B. führt `npm start` die Skripte `prestart`, `start` und `poststart` aus

- Ausführen eines beliebigen Skripts

```
$ npm run-script compile
```

```
$ npm run compile
```

# Mikroübung

## Richtig oder falsch?



- Jede Anfrage an einen mit Node.js realisierten Web-Server wird durch einen dedizierten Thread beantwortet, sodass Zugriffe auf gemeinsame Ressourcen synchronisiert werden müssen

  falsch

- Wenn für eine I/O-Operation eine synchron aufrufbare und eine asynchron aufrufbare Variante existieren, dann ist die synchron aufrufbare Variante stets zu bevorzugen

  ja, es sei denn, ich bin der einzige User

- Für die Bereitstellung eines Node.js-Projekts kann das `node_modules`-Verzeichnis ignoriert werden, da die benötigten Pakete jederzeit nachgeladen werden können

  ja

# Inhaltsübersicht

# Express



- Express ist ein minimalistisches Web Application Framework für Node.js
  - Entwickelt von TJ Holowaychuk (1.0 Beta: 15. Juli 2010)
  - http://expressjs.com
- Hauptmerkmale
  - Ermöglicht das Routing von HTTP-Anfragen
  - Unterstützt mehrere HTML-Template Engines, u. a.
    - Pug (früher Jade): https://github.com/pugjs/pug
    - handlebars: http://handlebarsjs.com
- Installation

```
$ npm install express
```

# Express

## Einführendes Beispiel



```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.type('text');
  res.send('Hello World!\n');
});

app.listen(3000, () => {
  console.log('listening on port 3000!');
});
```

Datei app.js

### Starten der Anwendung

```
$ node app.js
```

### Beispiel-Anfrage

```
$ curl http://localhost:3000
```

# Grundlegende Objekte

## Express-Anwendungsobjekt



- Ein Express-Anwendungsobjekt ermöglicht u. a. das
  - Routing von HTTP-Anfragen
  - Konfigurieren von Middleware-Funktionen
  - Registrieren einer Template-Engine und Rendern von HTML-Templates
  - Speichern globaler Werte (in der Eigenschaft locals)
- Es wird über die `express`-Funktion erzeugt

```
const express = require('express');
const app = express();
```

Per **Konvention** wird die Variable **app** genannt.

- Ablage globaler Werte

```
app.foo = 'bar';
```

# Grundlegende Objekte

## Request-Objekt



- Das Request-Objekt repräsentiert eine HTTP Anfrage
  - Namenskonvention für den Parameter: `req`
- Wesentliche Eigenschaften

| Eigenschaft | Erläuterung |
|---|---|
| `app` | Zugehöriges Anwendungsobjekt |
| `body` | Message Body. Erfordert eine Middleware wie `body-parser` oder `multer` (sonst `undefined`) |
| `cookies` | Objekt, das für jeden Cookie eine Eigenschaft besitzt. Erfordert die Middleware `cookie-parser` (sonst `{}`). |
| `method` | HTTP-Methode: `'GET'`, `'PUT'`, `'POST'` etc. |
| `params` | Objekt, das für jeden Route-Parameter eine Eigenschaft besitzt |
| `query` | Objekt, das für jeden Parameter des Anfragestrings eine Eigenschaft besitzt |

# Grundlegende Objekte

## Request-Objekt: Methoden (Auswahl)



- Auswahl eines zum `Accept`-Header passenden Werts

```
// Accept: text/*;q=.5, application/json
req.accepts('html', 'json'); // => 'json'
```

  - Es können ein oder mehrere Strings oder ein Array von Strings übergeben werden, von denen der **beste** Match zurückgeliefert wird (bzw. `false`, falls kein Match existiert)

- Weitere `accepts`-Methoden

| Methode | Zugehöriger HTTP-Header der Anfrage |
|---|---|
| `acceptsCharsets` | Accept-Charset |
| `acceptsEncodings` | Accept-Encoding |
| `acceptsLanguages` | Accept-Language |

Analog zur accepts-Methode, jedoch wird der **erste** Match geliefert.

# Grundlegende Objekte

## Request-Objekt: Methoden (Auswahl)



- Prüfen, ob ein MIME-Type zum Content-Type Header der Anfrage passt

```
// Bei Content-Type: text/html; charset=utf-8
req.is('html'); // => true
```

- Zugriff auf einen HTTP-Header der Anfrage

```
req.get('Content-Type'); // => "text/plain"
```

# Grundlegende Objekte

## Request-Objekt: Zugriff auf den Message Body



- Der Zugriff auf den Message Body erfordert eine Middleware, die zunächst zu installieren ist

```
$ npm i body-parser
```

- Registrierung der Middleware

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.urlencoded({ extended: true }));
```

  - Konfiguriert die Middleware für die Dekodierung von HTML-Formulardaten
  - Konfiguration für JSON

```
app.use(bodyParser.json());
```

# Grundlegende Objekte

## Request-Objekt: Zugriff auf den Message Body



- Anschließend enthält die body-Eigenschaft des Request-Objekts den dekodierten Message Body in Form eines JavaScript-Objekts

```
<form action="/tasks" method="post">
  <input name="title" type="text">
  <textarea name="description" type="text">
</form>
```

```
app.post('/tasks', (req, res) => {
  console.log('Title: ' + req.body.title);
  console.log('Description: ' + req.body.description);
  // ...
});
```

# Grundlegende Objekte

## Response-Objekt



- Das Response-Objekt repräsentiert eine HTTP Antwort
  - Namenskonvention für den Parameter: `res`
- Wesentliche Eigenschaften

| Eigenschaft | Erläuterung |
|---|---|
| `app` | Zugehöriges Anwendungsobjekt |
| `locals` | Objekt, das die Ablage Anfrage-spezifischer Informationen ermöglicht |

# Grundlegende Objekte

## Response-Objekt: Methoden (Auswahl)



- Message Body setzen und Antwort senden

```
res.send('Hello World!\n');
```

  - Ohne weitere Angaben wird `text/html` als Content Type sowie der Status Code 200 angenommen

- HTTP Header setzen (allgemein / Content Type)

```
res.set('Content-Type', 'text/html');
```

```
res.type('html');
```

- Status-Code setzen

```
res.status(200);
```

# Grundlegende Objekte

## Response-Objekt: Methoden (Auswahl)



- Status-Code setzen und Antwort ohne Message Body absenden

```
res.status(404).end();
```

- Status-Code setzen und Antwort mit Message Body absenden

```
res.status(200).json({ name: 'Max Mustermann' });
```

  - Im Beispiel wird automatisch der passende Content-Type Header gesetzt

- Cookie setzen (Name, Wert [,Optionen])

```
res.cookie('username', 'max', { secure: true , httpOnly: true });
res.cookie('cart', { items: [1,2,3] }, { maxAge: 900000 });
```

- Redirect absenden

```
res.redirect('/login');
```

# Grundlegende Objekte

## Response-Objekt: Methoden (Auswahl)





- Antwort absenden mit Content Negotiation

```
res.format({
  text: () => { res.send('Hello World!'); },
  html: () => { res.send('<p>Hello World!</p>'); },
  json: () => { res.send({ message: 'Hello World!' }); },
  default: () => { res.status(406).send('Not Acceptable'); }
});
```

- Es wird die zum `Accept`-Header passende Funktion ausgeführt und der `Content-Type` Header entsprechend gesetzt
  - Wenn der `Accept`-Header nicht gesetzt ist, wird die erste Funktion ausgeführt
- Statt der Kurzform (z. B. `json`) kann auch der entsprechende Media Type angegeben werden

```
'application/json': () => { res.send({ /* ... */ });  },
```

# Statische Dateien



- Für die Auslieferung statischer Dateien ist eine Middleware zu registrieren

```
const express = require('express');
const path = require('path');
const app = express();

app.use(express.static(path.join(__dirname, 'public')));

app.listen(3000, () => {
  console.log('listening on port 3000!');
});
```

Datei app.js

```
my-app
  src
    public
      css
        style.css
      js
        main.js
      index.html
    app.js
```

- So wird beim Aufruf der URL `http://localhost:3000` die Datei `public/index.html` ausgeliefert

# Routing



- Definition einer Route

```
app.METHOD(PATH, HANDLER)
```

  - `METHOD` ist eine **HTTP-Methode** (`get`, `put`, `post`, `delete` etc.)
  - `PATH` ist der **Route-Pfad**, mit dem der Pfadanteil einer URL verglichen wird
  - HANDLER ist der **Route Handler**, der definiert, wie auf einen entsprechenden Request zu reagieren ist
  - Die Kombination aus der HTTP-Methode und dem Route-Pfad wird **Endpunkt** (*endpoint*) genannt

- Beispiel

```
app.get('/', (req, res) => {
  // ...
});
```

# Routing
## Route-Pfad



- Ein Route-Pfad wird als String, regulärer Ausdruck oder als Array von Strings und/oder regulären Ausdrücken angegeben

| Beispiel | Erläuterung |
|---|---|
| `'/about'` | Muss genau mit dem URL-Pfad übereinstimmen |
| `'/admin/*'` | URL-Pfad muss mit `/admin/` beginnen, z. B.<br>`/admin/a/b` |
| `'/orders/:id'` | URL-Pfad mit Parameter `id`, z. B.<br>/orders/4711 |
| `/.*fly$/` | URL-Pfad muss zum regulären Ausdruck passen, z. B.<br>`/butterfly`<br>`/a/b/dragonfly` |
| `['/about',`<br>`'/orders/:id',`<br>`/.*fly$/]` | URL-Pfad muss zu einer der drei Bedingungen passen, z. B.<br>`/about`<br>`/abc`<br>`/butterfly` |

# Routing
## Route Handler



- Ein Route Handler wird in Form einer oder mehrerer Funktionen (**Route Callbacks**) und/oder als Array solcher Funktionen übergeben

```
app.get('/about', (req, res, next) => {
  next(); // zum nächsten Route Callback wechseln
}, (req, res) => {
  res.json({ name: 'John Doe' });
});
```

| Parameter | Erläuterung |
|---|---|
| `req` | Request-Objekt |
| `res` | Response-Objekt |
| `next` | • Aufruf des nächsten Route Callbacks durch `next()`<br>• Überspringen der restlichen Route Callbacks des Route Handlers durch `next('route')` |

# Routing
## Route Handler



- Es sind mehrere Route Handler zu einem Endpunkt möglich

```
app.get('/about', (req, res, next) => {
  next('route'); // direkt zum nächsten Route Handler wechseln
}, (req, res) => {
    // wird übersprungen
});
app.get('/about', (req, res) => {
  res.json({ name: 'John Doe' });
});
```

- Es kann ein Route Handler für alle HTTP-Methoden definiert werden

```
app.all('/secret', (req, res, next) => {
  console.log('Wird bei get, put, post, delete etc. aufgerufen.');
  next(); // zum nächsten Route Handler wechseln
});
```

# Mikroübung



- Was leistet das folgende Programm?

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
let data = [];

app.use(bodyParser.json());

app.post('/', (req, res) => {
  data.push(req.body);
  res.status(201).json(req.body);
});

app.get('/', (req, res) => { res.json(data); });

app.listen(3000);
```

# Routing
## Route Handler



- Kompakte Form für mehrere Route Handler mit demselben Pfad

```
app.route('/user')
  .get((req, res) => {
    // ...
  })
  .post((req, res) => {
    // ...
  })
  .put((req, res) => {
    // ...
  });
```

# Routing
## Route Handler



- Über ein **Router-Objekt** können relative Routen definiert und gemeinsam eingehängt (*mount*) werden

```
const express = require('express');
const router = express.Router();
router.get('/user', (req, res) => {
  // ...
});
router.get('/order', (req, res) => {
  // ...
});
module.exports = router;
```

Datei api.js

```
const api = require('./api');
// ...
// Router einhängen
app.use('/api', api);
```

Datei app.js

➡ Definiert Routen für die Pfade `/api/user` und `/api/order`

# Middleware Funktionen



- Eine Middleware-Funktion (kurz: Middleware) ist eine Funktion, die vor der eigentlichen Anfrageverarbeitung aufgerufen wird
  - Sie realisiert typischerweise eine Querschnittsaufgabe, wie z. B. Logging, Authentifizierung und Fehlerbehandlung
- Parameter einer Middleware-Funktion

| Parameter | Erläuterung |
|---|---|
| `req` | Request-Objekt |
| `res` | Response-Objekt |
| `next` | • Aufruf der nächsten Middleware bzw. Route Callbacks durch `next()`<br>• **Achtung:** kein Überspringen der restlichen Middleware möglich |

# Middleware Funktionen

## Registrieren



- Eine Middleware Funktion wird über die Methode `use` eines Anwendungsobjekts bzw. Router-Objekts registriert

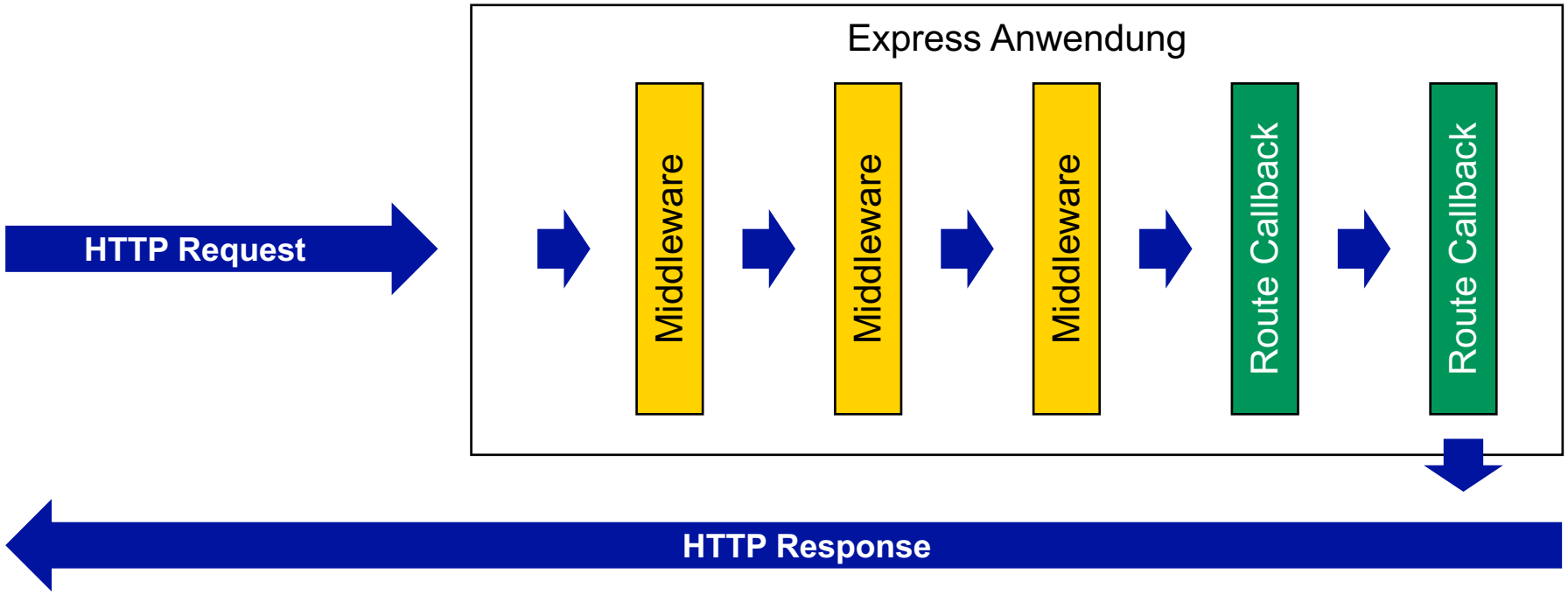
```
app.use('/', (req, res, next) => {
  console.log('middleware');
  next();
});
```

- Der erste Parameter definiert den Pfad
  - gleiche Alternativen wie beim Route-Pfad, jedoch als **Pfadpräfix**
    - z. B. passen zum Pfad `'/'` URLs mit den Pfaden `'/about'` und `'/order/4711'`
  - Parameter ist optional (Default ist `'/'`)
- Zweiter Parameter: Wie bei der Definition einer Route
  - Also ein oder mehrere Funktionen und/oder Array von Funktionen

# Middleware Funktionen



- Eine HTTP-Anfrage kann mehrere Middleware-Funktionen und Route Callbacks (in der Reihenfolge ihrer Registrierung) durchlaufen



# Middleware Funktionen

## Beispiel



```
const express = require('express');
const app = express();

app.use('/', (req, res, next) => {
  // wird bei jedem Request zuerst aufgerufen
  console.log('middleware');
  next();
});
app.get('/user', (req, res) => {
  res.json({ name: 'John Doe' });
});

app.listen(3000, () => {
  console.log('listening on port 3000!');
});
```

# Middleware Funktionen

## Durchreichen von Informationen



- Über die Eigenschaft `locals` des Response-Objekts können Anfrage-spezifische Daten an nachfolgende Middleware-Funktionen, Route Handler oder Templates weitergereicht werden
- Beispiel

```
app.use((req, res, next) => {
  // user anhand der Anfrage ermitteln
  res.locals.user = user;
  next();
});
```

  - Die Eigenschaft `locals` enthält initial ein leeres Objekt
  - Die Eigenschaften des `locals`-Objekts sind in den Templates abgreifbar

# Middleware Funktionen

## Fehlerbehandlungs-Funktionen



- Eine Fehlerbehandlungs-Funktion ist eine Middleware-Funktion mit vier statt drei Parametern
  - wird aufgerufen, wenn in vorherigen Middleware-Funktionen bzw. Route Callbacks synchron ein Fehler geworfen wird
  - Das Fehlerobjekt wird der Funktion als erstes Argument übergeben

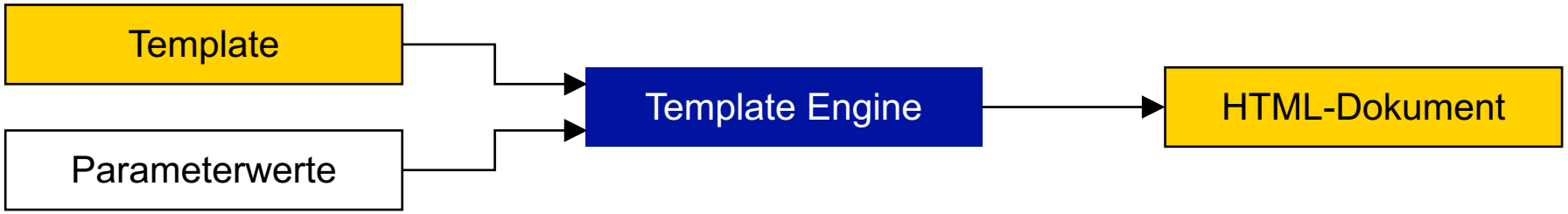
```
app.use((err, req, res, next) => {
  // wird nur im  Fehlerfall aufgerufen
  res.status(500).send('Ein Fehler ist aufgetreten!');
});
```

- Es können mehrere Fehlerbehandlungs-Funktionen definiert werden
  - Sie sollten als letzte Middleware-Funktionen registriert werden

# Template Engines



- Eine Template Engine ermöglicht es, (HTML-)Dokumente auf Basis von parametrisierten Vorlagen (*templates*) zu erzeugen




- Bekannte Template Engines
  - Handlebars (http://handlebarsjs.com)
  - Pug (https://pugjs.org/api/getting-started.html)

# Handlebars

## Installation und Konfiguration



- Installation

```
npm install express-handlebars
```

- Konfiguration als Template Engine für Express

```
const exphbs= require('express-handlebars');

app.engine('hbs', exphbs({ extname: '.hbs', defaultLayout: 'main' }));
app.set('views', path.join(__dirname, 'views')); // Templates-Verzeichnis
app.set('view engine', 'hbs'); // Registrieren der Template Engine
```

# Handlebars

## Templates



- Ein Handlebars Template ist allgemein ein Textdokument (i. d. R. ein HTML-Dokument) mit eingebetteten **Handlebars-Ausdrücken**
  - Ein Handlebars-Ausdruck wird durch geschwungene Klammern umschlossen
  - Er ermöglicht u. a. den Zugriff auf Parameterwerte
  - Beispiel: `{{header}}`
- Beim **Rendern** eines Templates werden dessen Name und die Parameterwerte übergeben

```
app.get('/', (req, res) => {
  res.render('index', {header: 'Mitarbeiter'});
});
```

```
<!doctype html>
<html>
<head><!-- ... --></head>
<body>
  <h1>{{header}}</h1>
</body>
</html>
```

Datei views/**index**.hbs

```
<!doctype html>
<html>
<head><!-- ... --></head>
<body>
  <h1>Mitarbeiter</h1>
</body>
</html>
```

# Handlebars

## Template-Ausdrücke



- Zugriff auf eine Eigenschaft des Kontextobjekts

```
<h1>{{header}}</h1>
```

```
<h1>{{this.header}}</h1>
```

  - **Kontextobjekt** ist per Default das Objekt, das beim Rendern übergeben wird (enthält die Parameterwerte). Es kann über `this` referenziert werden.

- Bedingtes Rendern

```
{{#if user}}
  <h2>Hallo {{user.name}}</h2>
{{else}}
  <h2>Willkommen</h2>
{{/if}}
```

# Handlebars

## Template-Ausdrücke



- Wiederholtes Rendern

```
<ol>
  {{#each employees}}
    <li>{{name}}</li>
  {{/each}}
</ol>
```

```
app.get('/', (req, res) => {
  const employees = [{ name: 'Max' },
                     { name: 'John' }];
  res.render('index', { employees });
});
```

```
<ol>
  <li>Max</li>
  <li>John</li>
</ol>
```

  - Es wird über das angegebene Array iteriert, wobei das jeweilige Array-Element zum Kontextobjekt für die inneren Ausdrücke wird

# Handlebars

## Template-Ausdrücke: Escaping



- Per Default wird das Ergebnis eines Template-Ausdrucks escaped

```
<h1>
  {{foo}}
</h1>
```

```
res.render('index',
  { foo: '<em>Foo</em>' }
);
```

```
<h1>
  &lt;em&gt;Foo&lt;/em&gt;
</h1>
```

- Bei dreifacher Klammerung erfolgt hingegen kein Escaping

```
<h1>
  {{{foo}}}
</h1>
```

```
res.render('index',
  { foo: '<em>Foo</em>' }
);
```

```
<h1>
  <em>Foo</em>
</h1>
```

# Handlebars

## Beispiel: Template



```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8"><title>Handlebars</title>
</head>
<body>
  <h1>{{header}}</h1>
  {{#if employees}}
    <ol>
      {{#each employees}}
        <li>{{this.name}}</li>
      {{/each}}
    </ol>
  {{/if}}
</body>
</html>
```

Datei views/index.hbs

# Handlebars

## Beispiel: Template-Nutzung



```
const express = require('express');
const exphbs= require('express-handlebars');
const app = express();

app.engine('hbs', exphbs({ extname: '.hbs' }));
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

app.get('/', (req, res) => {
  const employees = [{ name: 'Max' }, { name: 'Erika' }];
  res.render('index', {header: 'Mitarbeiter', employees: employees});
});

app.listen(3000, () => {
  console.log('listening on port 3000!');
});
```

Datei app.js

# Handlebars

## Layouts



- Ein von der Anwendung gerendertes Template wird typischerweise in ein Layout Template (kurz: **Layout**) eingebettet

```
<h1>About</h1>
```

Datei views/about.hbs

```
<!doctype html>
<html>
<head>
  <title>foo</title>
</head>
<body>
  {{{body}}}
</body>
</html>
```

Datei views/**layouts**/**main**.hbs

```
<!doctype html>
<html>
<head>
  <title>foo</title>
</head>
<body>
  <h1>About</h1>
</body>
</html>
```

```
my-app
  src
    views
      layouts
        main.hbs
      about.hbs
      index.hbs
    app.js
  ...
```

- Ein Layout ermöglicht somit, die gemeinsame Grundstruktur für mehrere Templates zu beschreiben

# Handlebars

## Layouts



- Das Verzeichnis der Layouts sowie das Default-Layout werden per Konfiguration festgelegt

```
const engineConfig = {
  extname: '.hbs',
  layoutsDir: path.join(__dirname, 'views', 'layouts'),
  defaultLayout: 'main'
}
app.engine('hbs', exphbs(engineConfig));
```

- Alternativ zum Default-Layout kann beim Rendern ein spezifisches Layout angegeben werden (als weiterer Parameter)

```
res.render('employees', { header: 'Mitarbeiter', employees: employees,
                          layout: 'main' });
```

# Handlebars

## Partials



- Ein Partial Template (kurz: **Partial**) ist ein Template, das von einem anderen Template gerendert und eingebettet werden kann

```
<!doctype html>
<html>
<head>
  <title>foo</title>
</head>
<body>
  {{{body}}}
  {{> footer}}
</body>
</html>
```

Datei views/layouts/main.hbs

```
<footer>
  Inhalt des Footers
</footer>
```

Datei views/**partials**/**footer**.hbs

```
my-app
  src
    views
      layouts
        main.hbs
      partials
        footer.hbs
      about.hbs
      index.hbs
    app.js
  ...
```

- Partials ermöglichen somit die Wiederverwendung von Template-Fragmenten

# Handlebars

## Partials





- Das Verzeichnis der Partials wird per Konfiguration festgelegt

```
const engineConfig = {
  extname: '.hbs',
  partialsDir: path.join(__dirname, 'views', 'partials')
}
app.engine('hbs', exphbs(engineConfig));
```

- Beim Aufruf eines Partials kann ein Objekt übergeben werden, das zum Kontextobjekt innerhalb des Partials wird

```
<ol>
  {{#each employees}}
  {{> employee this}}
  {{/each}}
</ol>
```

Datei views/employees.hbs

```
<li>{{name}}</li>
```

Datei views/partials/employee.hbs

# Inhaltsübersicht

# MongoDB



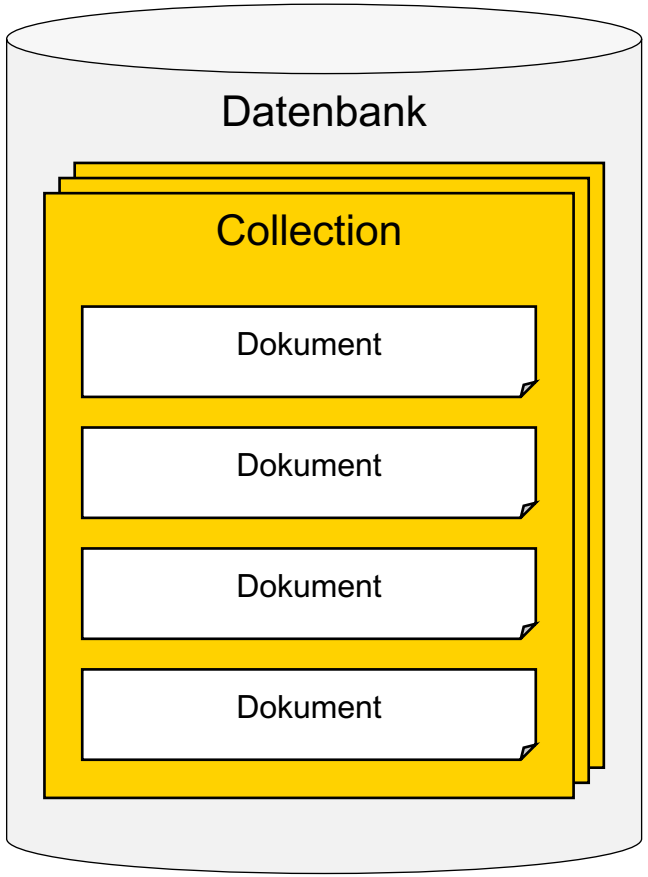
- MongoDB ist eine dokumentenorientierte NoSQL-DB
  - Open Source + kommerzielle Lizenz
    (Erstes Release: 2009)
  - https://www.mongodb.com
- Hauptmerkmale
  - Dokumente werden im BSON-Format schemalos gespeichert (Binary JSON)
  - Ausführung von JavaScript auf dem DB-Server möglich
  - Treiber für mehrere Programmiersprachen verfügbar (C#, C++, Java, JavaScript etc.)
  - Unterstützt Replikation und Sharding

# MongoDB



- Eine MongoDB-Instanz kann mehrere Datenbanken verwalten
- Eine Datenbank kann mehrere Collections enthalten
- Eine Collection enthält gleichartige Dokumente (z. B. Bestellungen)
- Ein Dokument besteht aus Feldern
- Ein Dokument kann andere Dokumente einbetten

Collection = Tabelle
Dokument = Spalte
Dokumente in einer Collection können unterschiedliche Felder haben --> kein ALTER TABLE notwendig



# MongoDB
## Grundlagen



- Starten des Datenbank-Servers

```
$ mongod –dbpath <Pfad zum Verzeichnis aller Datenbanken>
```

- Starten der MongoDB Shell

```
$ mongo
```

- Hilfe in der Shell aufrufen

```
> help
```

# MongoDB

## Shell: Beispiele für DB-Operationen



- Verbinden mit der Datenbank `human-resources`

```
use human-resources
```

- Einfügen eines Dokuments in die Collection `employees`

```
db.employees.insert({
  name: 'Max Mustermann', department: 4711
});
```

- Ermitteln von Dokumenten (mit Paging)

```
db.employees.find({department: 4711}).limit(5).skip(10);
```

- Anmerkung
  - Datenbank bzw. Collection wird mit dem ersten schreibenden Zugriff automatisch erzeugt

# MongoDB

## Shell: Beispiele für DB-Operationen



- Aktualisieren von Dokumenten (`multi: true` => *alle* Dokumente)

```
db.employees.update(
  { department: 4711 },
  { $set: { department: 4710 } },
  { multi: true }
)
```

- Index erstellen (1 => aufsteigend)

```
db.employees.createIndex( { name: 1 } )
```

- Collection löschen

```
db.employees.drop();
```

# MongoDB
## Node.js



- Populäre Möglichkeiten für den Zugriff auf eine MongoDB-Datenbank unter Node.js
  - Offizieller MongoDB-Treiber
    - Bietet eine ähnliche API wie die MongoDB Shell
  - Mongoose
    - Setzt auf dem offiziellen MongoDB-Treiber auf
    - Ermöglicht u. a. die Definition von Schemata für Collections und die entsprechende Konformitätsprüfung von Dokumenten
- Installation des offiziellen Treibers für MongoDB

```
$ npm i mongodb
```

# MongoDB

## Offizieller Treiber für Node.js



- Verbindung mit einer DB aufbauen

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost/my-db';

MongoClient.connect(url, (err, db) => {
  if (err) {
    console.log('Could not connect to MongoDB: ', err.stack);
    process.exit(1);
  } else {
    // Verbindung hergestellt
  }
});
```

- Verbindung schließen

```
db.close();
```

# MongoDB

## Offizieller Treiber für Node.js



- Einfügen eines Dokuments in eine Collection

```
let collection = db.collection('employees');
let employee = { name: 'John Doe', department: 4711 };

collection.insertOne(employee, (err, result) => {
  if (!err) { console.log('Inserted ' + result.insertedCount); }
});
```

- Einfügen mehrerer Dokumente in eine Collection

```
let employees = [ { name: 'John Doe', department: 4711 },
                  { name: 'Max Mustermann', department: 4711 } ];

collection.insertMany(employees, (err, result) => {
  if (!err) { console.log('Inserted ' + result.insertedCount); }
});
```

# MongoDB

## Offizieller Treiber für Node.js



- Ermitteln des ersten Dokuments, das zur Anfrage passt

```
collection.findOne({ department: 4711 }, (err, employee) => {
  if (!err) { console.log(employee); }
});
```

- Ermitteln aller Dokumente, die zur Anfrage passen (mit Sortierung & Limit)

```
collection.find({ department: 4711 })
  .sort({ name: 1 })
  .limit(2)
  .toArray((err, employees) => {
    if (!err) { console.log(employees); }
  });
```

  - Für eine absteigende Sortierung: `.sort({ name: -1 })`

# MongoDB

## Offizieller Treiber für Node.js



- Das erste Dokument aktualisieren, das zur Anfrage passt

```
collection.updateOne(
  { name: 'John Doe' }, { $set: { department: 4710 }},
  (err, result) => {
    if (!err) {
      console.log('Matched ' + result.matchedCount);
      console.log('Modified ' + result.modifiedCount);
    }
  });
```

- Alle Dokumente aktualisieren, die zur Anfrage passen

```
collection.updateMany(
  { name: 'John Doe' }, { $set: { department: 4710 }},
  (err, result) => { /* siehe oben */ });
```

# MongoDB

## Offizieller Treiber für Node.js



- Das erste Dokument entfernen, das zur Anfrage passt

```
collection.deleteOne({ name: 'John Doe' }, (err, result) => {
  if (!err) { console.log('Deleted ' + result.deletedCount); }
});
```

- Alle Dokumente entfernen, die zur Anfrage passen

```
collection.deleteMany({ department: 4710 }, (err, result) => {
  if (!err) { console.log('Deleted ' + result.deletedCount); }
});
```

# MongoDB
## Integration in Express



```
const express = require('express');
const MongoClient = require('mongodb').MongoClient;
// ...
const url = 'mongodb://localhost/my-db';
MongoClient.connect(url, (err, db) => {
  if (err) {
    console.log('Could not connect to MongoDB: ', err.stack);
    process.exit(1);
  } else { startServer(db); }
});

function startServer(db) {
  const app = express();
  app.locals.db = db;
  // ...
}
```

Datei app.js

# MongoDB
## Integration in Express



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

const router = express.Router();

router.get('/', (req, res) => {
  let db = req.app.locals.db;
  db.collection('employees').find().toArray((err, employees) => {
    res.render('employees', { employees });
  });
});

module.exports = router;
```

Datei routes/employees.js

# Mongoose



- Mongoose ist eine Object Data Mapping (ODM)-Lösung für MongoDB-Zugriffe unter Node.js
  - http://mongoosejs.com
- Ermöglicht u. a. die Definition von Schemata für Collections und die entsprechende Konformitätsprüfung von Dokumenten
- Installation

```
$ npm install mongoose
```

# Mongoose

## Beispiele für DB-Operationen



- Verbinden mit einer MongoDB-Datenbank

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/human-resources');
```

- Erstellen eines Schemas

```
const employeeSchema = new mongoose.Schema({
  name: { type: String, required: true },
  department: { type: Number, required: true }
});
```

- Erstellen eines Modells (Collection: `employees`)

```
const Employee = mongoose.model('Employee', employeeSchema);
```

# Mongoose
## Beispiele für DB-Operationen



- Speichern eines Dokuments

```
const employee = new Employee({
  name: 'Max Mustermann',
  department: 4711
});

employee.save((err, employee) => {
  if (err) { return handleError(err); }
  // Speichern war erfolgreich
});
```

# Mongoose
## Beispiele für DB-Operationen



- Abfragen von Dokumenten

```
Employee.find({department: 4711})
  .sort('+name')
  .exec((err, employees) => {
    if (err) { return handleError(err); }
    employees.forEach(employee => {
      console.log(JSON.stringify(employee));
    })
  });
```

# Mongoose
## Beispiele für DB-Operationen



- Aktualisieren von Dokumenten

```
Employee.update(
  { department: 4711 }, { department: 4710 },
  { multi: true }, (err, result) => {
    if (err) return handleError(err);
    console.log('Antwort von Mongo: ', result);
  }
);
```

- Entfernen von Dokumenten

```
Employee.remove({ department: 4711 }, err => {
  if (err) { return handleError(err); }
  // Remove war erfolgreich
});
```