ExpressJS

Fast, unopinionated, minimalist web framework for Node.js

Materialien

Source code & slides: https://github.com/philippkrauss/js-express-workshop

Ziele

Express.js kennenlernen

RESTful webservices implementieren

Datenbankanbindung

Project Setup

```
$ mkdir myapp

$ cd myapp

$ npm init # Achtung bei entry point: (index.js)

$ npm install express
```

Project Setup

Demo: minimales setup

Project Setup

Aufgabe: port konfigurierbar machen

Hinweise:

- CLI-Parameter stehen in array process.argv
  - Beispiel: process.argv[2]
- Umgebungsvariablen stehen in process.env
  - Beispiel: process.env.MY_ENV_VARIABLE

Simple CRUD

Was bedeutet CRUD?

HTTP Verbs

HTTP status codes

Daten-Input

Daten-Output

Simple CRUD

CRUD steht für:

Simple CRUD

CRUD steht für: Create/Read/Update/Delete

HTTP Verbs:

Simple CRUD

CRUD steht für: Create/Read/Update/Delete

HTTP Verbs: POST, GET, PUT, DELETE

HTTP status codes:

Simple CRUD

CRUD steht für: Create/Read/Update/Delete

HTTP Verbs: POST, GET, PUT, DELETE

HTTP status codes:

- 2xx - success
- 4xx - user error
- 5xx - server error

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Simple CRUD

Daten Input:

Simple CRUD

Daten Input:

- path
- query string
- request body
- cookies
- session
- request headers

Simple CRUD

Daten Output:

Simple CRUD

Daten Output:

- response status
- response body
- cookies
- session
- response headers

Simple CRUD

Demo: simple CRUD

Express.js API docs: https://expressjs.com/en/4x/api.html

Simple CRUD

Ein User hat die Eigentschaften id, name und email.

Aufgabe: Implementieren der folgenden Endpunkte

- GET /users
- GET /users/1

Server responses am besten in JSON

Wer noch Zeit hat gerne auch POST /users

Hinweise:

- users in einem globalen Array speichern
- Array.find
- Array.push

Database Connectivity

Asynchrone Programmierung

- Node.js ist single threaded
- I/O darf nicht blockieren
  - webservice calls, filesystem, database, …
- Lösung: callbacks
  - Semantik: Wenn du fertig damit bist, dann tue all das hier.
- Konvention: erster Parameter ist immer ein Fehler oder null

Database Connectivity

Beispiel:

```
const {readFile} = require('fs')
readFile('/etc/passwd', 'utf-8', (err, data) => {
 if (err) {
   console.log('an error occurred', err)
 } else {
   console.log(data)
 }
})
```

Database Connectivity

Node.js mysql Modul: https://github.com/mysqljs/mysql

```
var mysql = require('mysql')
var connection = mysql.createConnection({
 host: 'localhost',
 user: 'me',
 password: 'secret',
 database: 'my_db'
})

connection.query('SELECT 1 + 1 AS solution', function (error, results, fields)
{
 if (error) throw error
 console.log('The solution is: ', results[0].solution)
})
```

Database Connectivity

Node Clients für alle beliebten Datenbanken:

- mysql
- postgres
- mariadb
- redis
- mongodb
- sql server
- oracle
- elasticsearch
- …

Database Connectivity

Demo: Database Connectivity

Database Connectivity

Aufgabe: Umsetzen der bereits implementierten Endpoints mit mysql

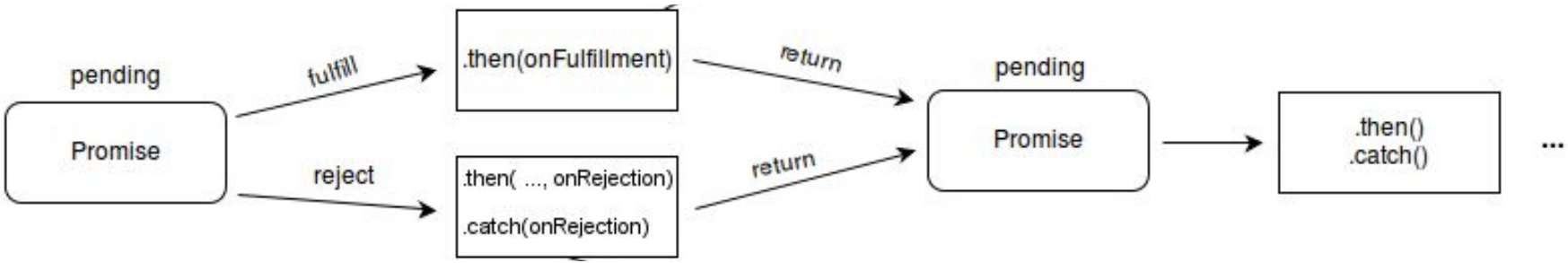
Promises

```
a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});
```

Promises

- Alternative zu callbacks
- 3 Zustände:
  - pending
  - rejected
  - resolved
- Heute: Fokus auf Benutzung, nicht Erzeugung

Promises


pending
Promise
fulfill
reject
.then(onFulfillment)
.then(..., onRejection)
.catch(onRejection)
return
return
pending
Promise
.then()
.catch()
...

Promises

Promise chaining:

```
const myPromise = new Promise((resolve, reject) => {
 setTimeout(() => {
   resolve('foo')
 }, 300)
})

myPromise
   .then(handleResolvedA)
   .then(handleResolvedB)
   .then(handleResolvedC)
```

Promises

Promise error handling:

```
myPromise
    .then(handleResolvedA,handleRejectedA)
    .then(handleResolvedB,handleRejectedB)
    .then(handleResolvedC,handleRejectedC)
```

Promises

Promise error handling:

```
myPromise
    .then(handleResolvedA)
    .then(handleResolvedB)
    .then(handleResolvedC)
    .catch(handleRejectedAny)
```

Promises

Node.js mysql-promises Modul:
https://github.com/CodeFoodPixels/node-promise-mysql

Promises

Demo: Promises

Promises

Aufgabe: Umsetzen der bereits implementierten Endpoints mit promise-mysql

async/await

Promises:

```
myPromise
    .then(handleResolvedA)
    .then(handleResolvedB)
    .then(handleResolvedC)
```

async/await:

```
const result = await myPromise
const resultA = await handleResolvedA(result)
const resultB = await handleResolvedB(resultA)
const resultC = await handleResolvedC(resultB)
```

async/await

Achtung: um await zu verwenden müssen Funktionen async deklariert sein

```
async function doSomethingAsync() {
  …
}
```

Nicht alle Funktionen können async sein, z.B. Konstruktoren, Array.find

async/await

Error handling:

```
try {
 await myPromise
 await handleResolvedA()
 await handleResolvedB()
 await handleResolvedC()
} catch (error) {
 handleError()
}
```

async/await

Achtung, beliebter Fehler: await vergessen!

async/await

Demo: async/await

async/await

Aufgabe: Umsetzen der bereits implementierten Endpoints mit async/await

Error Handling

Error handler haben **immer** 4 Parameter!

```
app.use((error, req, res, next) => {
 return res.status(500).send({error: error.message})
})
```

Error Handling

Verwendung in synchronem Code:

```
app.get('/error-not-async', () => {
 throw new Error('oh oh...')
})
```

Verwendung in asynchronem Code:

```
app.get('/error-async', async (req, res, next) => {
 next(new Error('oh oh...'))
})
```

try/catch nicht vergessen…

Error Handling

Hilfsfunktion für async handlers

```
function asyncHandler (callback) {
 return (req, res, next) => {
   callback(req, res, next)
       .catch(next)
 }
}
app.get('/endpoint', asyncHandler(async (req, res) => {
 …
}))
```

Error Handling

Demo: Error Handling

Error Handling

Aufgabe: Error Handler in bereits implementierten Endpoints umsetzen

Middlewares

Alle handler-Funktionen, die wir bisher geschrieben haben, sind middleware:

```
const middleware = (req, res, next) => {}
```

Middleware kann

- Code ausführen
- Änderungen an request oder response Objekt machen
- Request-response-Zyklus beenden (= Antwort geben)
- die nächste Middleware aufrufen

Middlewares

Middleware wird ausgeführt wenn die Basis der URL auf den Pfad passt:

app.use([path,] callback [, callback...])

Beispiel: path=/base funktioniert auch für GET /base/path

Middleware wird ausgeführt wenn HTTP Verb und Path genau passen:

app.get(path, callback [, callback ...])

app.put(path, callback [, callback ...])

…

Middlewares

Beispiele für path patterns:

‘/abcd’	/abcd
‘/abc?d’	/abcd und /abd
‘/ab+cd’	/abcd, /abbcd, /abbbbbcd, …
‘/ab*cd’	/abcd, /abxcd, /abFOOcd, /abbArcd
‘/a(bc)?d’	/ad /abcd

Middlewares

Beispiel für path regex:

/\/abc|\/xyz/	/abc /xyz

Beispiel für path array:

[/\/abc|\/xyz/, ‘/def’]	/abc /xyz /def

Middlewares

Demo: Middlewares

‘Offizielle’ Middlewares: https://expressjs.com/en/resources/middleware.html

und viele mehr…

Logging

Winston js: https://github.com/winstonjs/winston

Konzept: Transports

- File
  - log rotation
- Console
- …

Unterschiedliche Transports für log levels möglich

Logging

```
const logger = winston.createLogger({
 transports: [
   new winston.transports.File({
     filename: 'combined.log',
     level: 'info'
   }),
   new winston.transports.File({
     filename: 'errors.log',
     level: 'error'
   })
 ]});
```

Logging

Konzept: Formats

- simple
- json
- splat (string interpolation mit %d und %s)
- …

```
const logger = winston.createLogger({
 transports: [new winston.transports.Console()],
 format: winston.format.json()
});
```

Logging

Konzept: metadata + child loggers

```
logger.info('Hello, Logger!')

//loggt zusätzliche Metadaten
logger.info('Hello, Logger!', {key1: 'value1', key2: 'value2'})

//loggt zusätzlich die error message und den stack trace
logger.info('Hello, Logger!', new Error('oh oh...'))

//erzeugt einen child logger, der zusätzliche Metadaten loggt
const childLogger = logger.child({requestId: uuid()})
```

Logging

Demo: Logging

Static File Serving

Eingebaute middleware: express.static

Demo: Static File Serving

Sonstiges

- möglichst keine synchronen methoden verwenden
  - console.log ist synchron…
- NODE_ENV=’production’
- Prozess überwachen und automatisch neu starten
  - Prozessmanager
  - OS init system
  - Kubernetes
- Node ist Singlethreaded - einen Prozess pro Kern starten
  - Vorsicht, kein shared memory state
  - Alternative: DBMS, Redis
- x-powered-by header abschalten
  - `app.set('x-powered-by', false)`

Sonstiges

- Graceful shutdown

```
const server = app.listen(3000)
process.on('SIGTERM', () => {
 logger.info('SIGTERM signal received: closing HTTP server')
 server.close(() => {
   logger.info('HTTP server closed')
 })
})
```