

Software Engineering I (IB)

Node.js

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 15.11.2018 08:01

Inhaltsverzeichnis

JavaScript auf dem Server!?	1
Hello World in der Node Shell	2
Node ist eine Scripting Engine	2
Und jetzt ein erster Server	3
HTTP Status Code Definitions	3
Nicht blockierende IO und asynchrone Programmierung	4
Blockierend und synchron	4
Nichtblockierend und asynchron: <code>setTimeout</code> in JavaScript	4
Beispiel: Lesen aus einer Datei	5
Single-threaded event-loop	5
Fehlerbehandlung	6
Aber ich habe doch etwas Synchrones gefunden	6
Ein Node.js-TypeScript-Beispiel	6

JavaScript auf dem Server!?

- JavaScript bisher nur im Browser (vor Node.js)
- für Web-Anwendung typischerweise JavaScript auf dem Client (Browser)
 - und irgendeine andere Programmiersprache auf dem Server
- mit Node.js *Full-Stack-JavaScript* möglich



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

<https://nodejs.org/>

Hello World in der Node Shell

- die Node Shell ist ein REPL¹:
 - node
 - > console.log("Hello, world!")
 - Hello, world!
 - undefined
 - >
- das bedeutet:
 - console.log gibt die Zeichenkette Hello, world! **aus**
 - und den Wert undefined **zurück**
- Verlassen der REPL mit Ctrl+D

Node ist eine Scripting Engine

- speichern wir folgende Zeile in einer Datei und geben ihr z.B. den Namen hello.js:

```
console.log("hello, world!")
```
- können wir sie mit dem Kommando node hello.js ausführen:

¹REPL = **R**ead **E**valuate **P**rint **L**oop

```
→ node hello.js
Hello, world!
```

Und jetzt ein erster Server

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

const s = http.createServer((req, res) => {
  const body = "Thanks for calling!\n";
  const content_length = body.length;
  res.writeHead(200, {
    "Content-Length": content_length,
    "Content-Type": "text/plain"
  });
  res.end(body);
});

s.listen(8080);
```

- in Datei web.js speichern
- starten mit node web.js
- Öffnen im Browser oder Anzeigen mit `curl`

```
→ curl -i http://localhost:8080
HTTP/1.1 200 OK
Content-Length: 20
Content-Type: text/plain
Date: Wed, 07 Nov 2018 15:44:00 GMT
Connection: keep-alive
```

Thanks for calling!

HTTP Status Code Definitions

- siehe <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
- Informational 1xx
- Successful 2xx
 - 200 OK
 - ...
- Redirection 3xx
- Client Error 4xx
 - ...

- 404 Not Found
- ...
- 500 Internal Server Error
- Halten Sie sich an diese Codes und basteln Sie nichts eigenes!

Nicht blockierende IO und asynchrone Programmierung

Blockierend und synchron

- Beispiel:
 - öffne eine Datei (dauert bis sie offen ist)
 - lese den Inhalt (dauert bis der Inhalt gelesen ist)
 - mache etwas mit dem Inhalt...
- damit ein Server mehrere Clients gleichzeitig bedienen kann, müssen mehrerer solche Abläufe parallel laufen, z.B. in Threads oder Prozessen
- Problem: Jeder Thread/Prozess hat einen Overhead
 - Beispiel: Apache und PHP ca. 10-15 MB Overhead im RAM pro Prozess

Nichtblockierend und asynchron: `setTimeout` in JavaScript

- Beispiel: Datei `timeout.js`

```
setTimeout(() => {  
  console.log("Arbeit erledigt!");  
}, 2000);  
  
console.log("ich warte ...");
```
- was wird ausgegeben beim Aufruf
 - `node timeout.js`
 - und warum?
- `setTimeout` hat als ersten Parameter eine **Callback-Funktion**
- `setTimeout(f, 2000)` heißt also
 1. warte 2000 Millisekunden und
 2. führe im Anschluss daran die Funktion `f` aus
- diese Idee setzt Node.js grundsätzlich um
 - statt auf eine Aktion zu warten, wird eine Callback-Funktion mitgegeben, die im Anschluss ausgeführt wird

Beispiel: Lesen aus einer Datei

- 1. Versuch

```
const fs = require("fs");

let file;
const buf = Buffer.alloc(100000);

fs.open("readfile.js", "r", (err, handle) => {
  file = handle;
});

fs.read(file, buf, 0, 100000, null, (err, length) => {
  console.log(buf.toString());
  fs.close(file, () => {});
});
```

- Ausführen:

```
→ node readfile.js
internal/validators.js:113
  throw err;
  ^
```

```
TypeError [ERR_INVALID_ARG_TYPE]:
  The "fd" argument must be of type number.
  Received type undefined
  ...
```

- und warum?
- Lösungsmöglichkeit

```
const fs = require("fs");

fs.open("readfile.js", "r", (err, handle) => {
  const buf = Buffer.alloc(100000);
  fs.read(handle, buf, 0, 100000, null, (err, length) => {
    console.log(buf.toString());
    fs.close(handle, () => {});
  });
});
```

Single-threaded event-loop

- Node.js läuft in einem einzigen Thread

- Callback-Funktionen werden in einem Event-Stack verwaltet und sind erst ausführbar, wenn alle Parameter vorliegen
- der eine Eventloop (single-thread) führt dann eine Callback-Funktion nach der anderen aus
- Achtung: “teure” Berechnungen können die gesamte App “einfrieren”
- Lösungsmöglichkeit:
 - globales `process`-Objekt hat eine `nextTick`-Methode²
 - mit dieser kann man die aktuelle Berechnung stoppen und eine weitere Funktion aufrufen lassen

Fehlerbehandlung

- durch die asynchrone Ausführung macht `try` und `catch` in Node.js keinen Sinn
- Lösung:
- die Callback-Funktionen bekommen in der Regel als ersten Parameter einen Fehlerwert übergeben
- Beispiel:

```
fs.open("readfile.js", "r", (err, handle) => { ...
```

- Fehlerfreiheit, wenn `err` den Wert `null` hat

Aber ich habe doch etwas Synchrones gefunden

- Node.js hat von manchen Funktionen, synchrone Versionen, z.B.

```
const fs = require("fs");

const handle = fs.openSync("readfile.js", "r");
const buf = Buffer.alloc(100000);
const read = fs.readSync(handle, buf, 0, 100000, null);
console.log(buf.toString("utf8", 0, read));
fs.closeSync(handle);
```

- diese sind z.B. für Kommandozeilentools, die Sie mit Node.js schreiben können
- **nicht** für Webserver!!!

Ein Node.js-TypeScript-Beispiel

- Mini-Server für die Angular Tour of Heroes
- <https://github.com/swenib/nodeWithTypeScript>

²Wenn Sie das brauchen, können wir uns das gemeinsam anschauen