Node.js

Software Engineering I (IB)

Prof. Dr. Oliver Braun

Letzte Änderung: 15.11.2018 08:01

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 REPL[1]:

```
↝ 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`

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

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:

```
↝ node hello.js
Hello, world!
```

Und jetzt ein erster Server (1/2)

```
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/plan"
    });
    res.end(body);
});

s.listen(8080);
```

Und jetzt ein erster Server (2/2)

- 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/plan
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 mehrer 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 (1/2)

- Beispiel: Datei `timeout.js`

```
setTimeout(() ⇒ {
  console.log("Arbeit erledigt!");
}, 2000);

console.log("ich warte ...");
```

- was wird ausgegeben beim Aufruf

  ```
  ⇝ node timeout.js
  ```

  und warum?

Nichtblockierend und asynchron: `setTimeout` in JavaScript (2/2)

- `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 (1/3)

- 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, () => {});
});
```

Beispiel: Lesen aus einer Datei (2/3)

- 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?

Beispiel: Lesen aus einer Datei (3/3)

- 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`-Methode[2]
  - mit dieser kann man die aktuelle Berechnung stoppen und eine weitere Funktion aufrufen lassen

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

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