Web-Entwicklung

Serverseitige Anwendungen I: Node.js und Express

Inhalte der Vorlesung

- Serverseitige Anwendungen: Node.js und Express
  - Grundlagen
  - Zugriff auf das Dateisystem
  - Zugriff auf Datenbanken
    - SQL
    - NoSQL
  - Erstellen eines HTTP-Servers
  - Erstellen eines WebSocket-Servers
  - Packaging

Grundlagen

Frontend - Backend

- Kommunikation der clientseitigen Browser-Anwendung ("Frontend") mit entfernten Servern ("Backends") ist ein wesentlicher Aspekt von Web-Anwendungen
- Kommunikationsprotokolle
  - HTTP
  - WebSockets
- Protokoll dient als Schnittstelle und erlaubt Realisierung des Backends unabhängig vom Frontend mithilfe verschiedener Technologien
  - Java, .NET, …
  - Perl, PHP, Python, Ruby, …
- Aber auch: JavaScript via Node.js
  - Beliebt bei erfahrenen JavaScript-Entwicklern, da kein zusätzliches "Ökosystem"
  - Standalone-Betrieb mit integriertem HTTP-Server oder hinter einem anderen HTTP-Server (*Reverse Proxy*, z.B. Apache)
  - Eignung für Produktiveinsatz unter hoher Last

Event Loop und blockierende synchrone Anweisungen

- Eine Node.js-Anwendung wird stets in nur einem Thread ausgeführt!
  - Sog. *Event Loop*
- Vorteil: Programmierung in vielen Fällen einfacher
- Nachteil: Langanhaltende Operation blockieren die Event Loop
  - I/O: Dateisystemzugriff, Netzwerkkommunikation
  - CPU-intensive Funktionen, z.B. Verschlüsselung
- Anhalten des Kontrollfluss insbesondere bei vielen Client-Anfragen kritisch

```
let data = fs.readFileSync("file.md");

// synchronous I/O function blocks
// event loop and execution waits here
// until file is read

console.log(data);
moreWork();  // will run after console.log()
```

Nicht-blockierende asynchrone Anweisungen

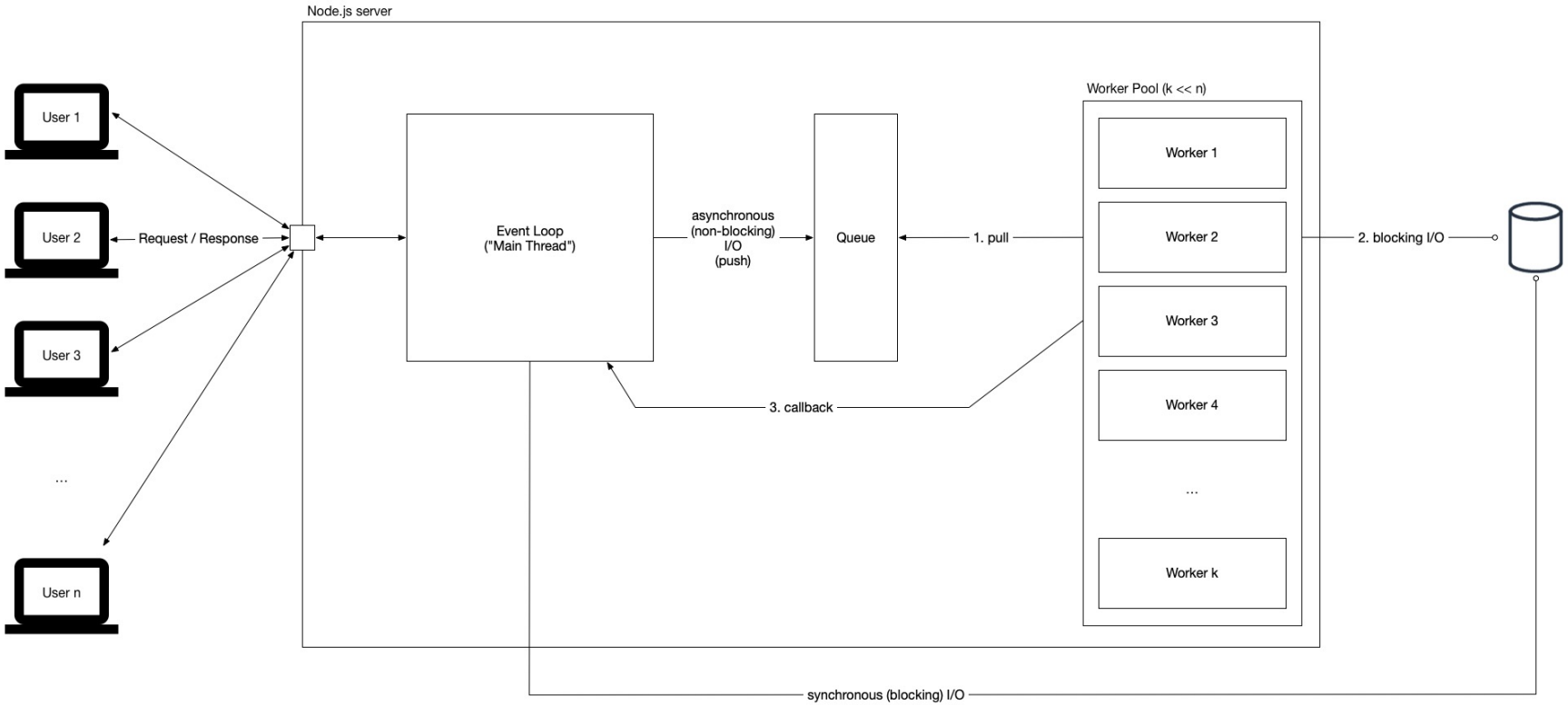
- Node.js bietet für fast alle I/O-Operationen auch nicht-blockierende, asynchrone Varianten an
- *Node.js style callback*
- *Run-to-completion*: Kontrollfluss läuft regulär weiter, d.h. die Anweisungssequenz und alle davon aufgerufenen Funktionen werden vollständig ausgeführt, bevor die Event Loop in der nächsten Iteration alle anstehenden Callbacks ausführt

```
fs.readFile("file.md",
   (error, data) => {
      if (error) {
         console.error(error.message);
         return;
      }
      console.log(data);
   }
);
moreWork(); // will run before console.log()!
```

Worker Pool

- Intern verwaltet Node.js einen Pool zusätzlicher Threads mit konstanter geringer Größe
  - Größe des Pools unabhängig von Anzahl der Client-Anfragen
- Worker bedienen sich an einer gemeinsamen Queue, übernehmen im Hintergrund die Ausführung der langanhaltenden Operationen und rufen anschließend die Callback-Funktion (in der Event Loop) auf
  - Relative wenige Kontextwechsel nötig
  - Hohe Skalierbarkeit

Node.js-Server



Events

- Erinnerung: Ereignisgesteuerte Programmierung in ein wesentlicher Faktor bei der Entkopplung von Komponenten
  - In Browser-Anwendungen via DOM-Events
- Standardmodul `event`: https://nodejs.org/api/events.html
  - Zentrale Klasse `EventEmitter` implementiert Observer-Entwurfsmuster
  - Andere Standardmodule und viele npm-Module verwenden `EventEmitter`
  - Kann in eigenen Modulen eingesetzt werden
    - Kann sogar via Browserify im Browser eingesetzt werden und somit DOM-Events ersetzen

EventEmitter

```
const EventEmitter = require("events");

class Emitter extends EventEmitter {
   constructor() {
      super();
      setInterval(this.emitEvent.bind(this),
            1000);
   }

   emitEvent() {
      this.emit(Emitter.ON_EXECUTE, new Date());
   }
}

Emitter.ON_EXECUTE = "EMITTER_ON_EXECUTE";

module.exports = Emitter;
```

```
const Emitter = require("./Emitter");

let emitter = new Emitter();
emitter.on(Emitter.ON_EXECUTE, date => {
   console.log(date.toString());
});  // execute callback on every emit

emitter.once(Emitter.ON_EXECUTE, () => {
    console.log("TADA!");
});  // execute callback only once
```

Zugriff auf das Dateisystem

Dateisystem-Zugriff

- Daten, die von Clients angefragt werden können, liegen typischerweise in einer Datenbank oder in Dateien auf dem Dateisystem des Servers
- Node.js-Anwendungen sind reguläre Anwendungen und besitzen somit Zugriffsrechte auf das Dateisystem gemäß des Nutzers, der die Anwendung ausführt
- Standardmodul `fs`: https://nodejs.org/api/fs.html
- Wiederholung: Asynchrone Funktionen nutzen Worker-Mechanismus, synchrone blockieren den Haupt-Thread
  - Bevorzuge asynchrone Funktionen bei Operationen, die während der gesamten Laufzeit und insbesondere bei Client-Anfragen angestoßen werden
  - Synchrone Funktionen für einmalige Operationen zur Initialisierung der Server-Anwendung (z.B. Auslesen von Konfigurationsdateien)

Dateien lesen

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

fs.readFile("./Luxemburg_2012.gpx", (error, data) => {
   if (error) {
      console.error(error);
      return;
   }
   console.log(data.toString());
});
```

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

try {
   let data = fs.readFileSync("./Luxemburg_2012.gpx");
   console.log(data.toString());
}
catch (error) {
   console.error(error);
}
```

Dateien schreiben

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

fs.writeFile("./outputAsync.txt", "Async\n2\n", error => {
   if (error) {
      console.error(error);
      return;
   }
});
```

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

try {
   fs.writeFileSync("./outputSync.txt", "Sync\n1\n");
}
catch (error) {
   console.error(error);
}
```

Dateien löschen

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

fs.unlink("./outputAsync.txt", error => {
   if (error) {
      console.error(error);
      return;
   }
});
```

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

try {
   fs.unlinkSync("./outputSync.txt");
}
catch (error) {
   console.error(error);
}
```

Verzeichnisse erstellen, auslesen und löschen

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


try {
   fs.mkdirSync("./newDirSync");
   fs.rmdirSync("./newDirSync");


   let files = fs.readdirSync(".");
   for (let file of files) {
      console.log(file);
   }
}
catch (error) {
   console.error(error);
}
```

Verzeichnisse erstellen, auslesen und löschen

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

fs.mkdir("./newDirAsync", error => {
   if (error) {
      console.error(error);
      return;
   }

   fs.rmdir("./newDirAsync", error => {
      if (error) {
         console.error(error);
         return;
      }
   });
});
```

```
fs.readdir(".", (error, files) => {
   if (error) {
      console.error(error);
      return;
   }
   for (let file of files) {
      console.log(file);
   }
});
```

Weitere Funktionen

- `fs.stat()`
- `fs.access()`
- `fs.appendFile()` bzw. `fs.appendFileSync()`
- `fs.existsSync()`
- `fs.rename()` bzw. `fs.renameSync()`
- …

Zugriff auf Datenbanken

Datenbanken

- npm-Module zum Zugriff auf alle gängigen Datenbanksysteme verfügbar
  - Relationale DBMS
    - MySQL (mysql)
    - SQLite (sqlite3, sqlite)
    - Oracle Database, PostgreSQL, ...
  - NoSQL-DBMS
    - MongoDB (mongodb)
    - CouchDB, ...
    - Redis, ...
- Installation via `npm install`
- Hier nur Verbindungsaufbau und CRUD-Funktionalität
  - Modul "Datenbanken"
  - Fachliteratur

SQLite

```
// npm install sqlite3
// npm install sqlite (wrapper around sqlite3 that uses ES6 promises
// instead of callbacks)
const sqlite3 = require("sqlite3");

let db = new sqlite3.Database(":memory:");
try {
   db.serialize(() => {
      db.run("CREATE TABLE department (id CHARACTER(3) PRIMARY KEY,
              name TEXT)");

      let statement = db.prepare("INSERT INTO department VALUES (?, ?)");
      statement.run("INF", "Informatik");
      statement.run("WIR", "Wirtschaft");
      statement.finalize();

      db.run("CREATE TABLE course (id INTEGER PRIMARY KEY ASC,
            name TEXT, department CHARACTER(3))");

      statement = db.prepare("INSERT INTO course(name,
            department) VALUES (?, ?)");
      statement.run("Informatik (B.Sc.)", "INF");
      statement.run("Informatik - Digitale Medien und
            Spiele (B.Sc.)", "INF");
      statement.run("Informatik - Sichere und mobile
            Systeme (B.Sc.)", "INF");
      statement.run("Medizininformatik (B.Sc.)", "INF");
      statement.run("Betriebswirtschaft (B.Sc.)", "WIR");
      statement.finalize();
      …
   });
}
finally {
   db.close();
}
```

SQLite

```
const sqlite3 = require("sqlite3");

let db = new sqlite3.Database(":memory:");
try {
   db.serialize(() => {
      …
      db.each("SELECT course.name AS courseName, department.name as departmentName
            FROM (course LEFT JOIN department ON (course.department = department.id))", function (error, row) {
         if (error) {
            console.error(error);
            return;
         }
         console.log(`${row.departmentName}: ${row.courseName}`);
      });
   });
}
finally {
   db.close();
}
```

MySQL

```
// npm install mysql
const mysql = require("mysql");

var connection = mysql.createConnection({ host: "localhost", user: "cb", password: "abcd", database: "we" });
connection.connect();
try {
   connection.query("SELECT course.name AS courseName, department.name as departmentName
         FROM (course LEFT JOIN department ON (course.department = department.id))",
         function (error, results, fields) {
      if (error) {
         console.log(error);
         return;
      }

      for (let row of results) {
         console.log("%s: %s", row.departmentName, row.courseName);
      }
   });
}
finally {
   connection.end();
}
```

MongoDB

- Populärstes dokumentenorientiertes NoSQL-DBMS
- Verwaltung von Datensätzen in Dokumenten
  - JSON-ähnliche Objekte
- Aggregation von
  - Dokumenten in Collections
    - Collections entsprechen Tabellen in RDBMS
    - Unterschied: Dokumente in einer Collection müssen nicht zwingend die gleiche Struktur aufweisen ("Schema-frei")
  - Collections in Datenbanken
- Serverseitige DB-Programmierung mit JavaScript

MongoDB (Verbindungsaufbau)

```
// Install MongoDB, start service: mongod --dbpath "/home/user/mongodbdata"
// Create database 'Library': mongo && use Library
// npm install mongodb

const dbClient = require("mongodb").MongoClient;

dbClient.connect("mongodb://localhost:27017/Library", (error, db) => {
   if (error) {
      console.error(error);
      process.exit(-1);
   }
   console.log("Connected to MongoDB.");
   try {
      …
   }
   finally {
      db.close();
   }
});
```

MongoDB (Dokumente erzeugen)

```
try {
   let books = db.collection("books");

   // create documents
   books.insertOne({ author: "Stevenson, Louis", title: "Die Schatzinsel" });

   books.insertMany([
      { author: "Verne, Jules", title: "Die geheimnisvolle Insel" },
      { author: "Verne, Jules", title: "20.000 Meilen unter dem Meer" },
      { author: "Verne, Jules", title: "Matthias Sandorf" },
   ]);
}
finally {
   db.close();
}
```

MongoDB (Dokumente lesen)

```
try {
   let books = db.collection("books");

   // read documents
   let cursor = books.find();
   cursor.each((error, book) => {
      if (error) { console.error(error); return; }
      if (book) { console.log("%s: %s", book.author, book.title); }
   });

   // filter documents
   cursor = books.find({ author: "Verne, Jules" });
   cursor.each((error, book) => { … });
}
finally {
   db.close();
}
```

MongoDB (Dokumente aktualisieren)

```
try {
   let books = db.collection("books");

   // update document(s)
   books.updateOne({ title: "20.000 Meilen unter dem Meer" }, { $set: { title: "20.000 Meilen unter den Meeren" } });
}
finally {
   db.close();
}
```

MongoDB (Dokumente löschen)

```
try {
   let books = db.collection("books");

   // delete document(s)
   books.deleteOne({ title: "Matthias Sandorf" });
   books.deleteMany({});
}
finally {
   db.close();
}
```

Erstellen eines HTTP-Servers

Standardmodul http

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

const PORT = 8080;

http.createServer((request, response) => {
    let data = "Hello, World!";
    response.writeHead(200, { "Content-Type": "text/plain", "Content-Length": data.length });
    response.end(data);
}).listen(PORT, () => { console.log("HTTP server listening on port %d.", PORT); });
```

Routing

```
http.createServer((request, response) => {
    if (request.url === "/") {
        let data = "Hello, World!";
        response.writeHead(200, { "Content-Type": "text/plain", "Content-Length": data.length });
        response.end(data);
    }
    else if (request.url === "/json") {
        let data = JSON.stringify({
            message: "Hello, World!",
            success: true
        });
        response.writeHead(200, { "Content-Type": "application/json", "Content-Length": data.length });
        response.end(data);
    }
}).listen(8080);
```

Routing

```
http.createServer((request, response) => {
    ...
    else if (request.url === "/data" && request.method === "POST") {
        // parse and process POST data...
        response.writeHead(204);
        response.end();
    }
    else {
        response.writeHead("404");  // do not wait for timeout
        response.end();
    }
}).listen(8080);
```

Express

express

- http://expressjs.com/
- Leichtgewichtiger Wrapper um das Standardmodul `http`
  - De-facto-Standard
- Vereinfacht den Umgang mit
  - Routen
  - Auswerten von HTTP-Anfragen
  - Erzeugen der HTTP-Antworten
- Erweiterbar über sog. Middleware
- Installation:

```
npm install express
```

Express

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

let server = express();

server.get("/", (request, response) => {
    response.send("Hello, World!");
});

server.get("/json", (request, response) => {
    response.json({
        message: "Hello, World!",
        success: true
    });
});

server.get("/XXX", (request, response) => {
    response.sendStatus(404);
});

server.listen(8080);
```

Routing (HTTP-Methoden)

Express supports the following routing methods corresponding to the HTTP methods of the same names:

- `checkout`
- `copy`
- `delete`
- `get`
- `head`
- `lock`
- `merge`
- `mkactivity`
- `mkcol`
- `move`
- `m-search`
- `notify`
- `options`
- `patch`
- `post`
- `purge`
- `put`
- `report`
- `search`
- `subscribe`
- `trace`
- `unlock`
- `unsubscribe`

The API documentation has explicit entries only for the most popular HTTP methods `app.get()`, `app.post()`, `app.put()`, and `app.delete()`. However, the other methods listed above work in exactly the same way.

Quelle: http://expressjs.com/de/4x/api.html#routing-methods

Erzeugen der HTTP-Antwort

Methode	Beschreibung
res.download()	Gibt eine Eingabeaufforderung zum Herunterladen einer Datei aus.
res.end()	Beendet den Prozess “Antwort”.
res.json()	Sendet eine JSON-Antwort.
res.jsonp()	Sendet eine JSON-Antwort mit JSONP-Unterstützung.
res.redirect()	Leitet eine Anforderung um.
res.render()	Gibt eine Anzeigevorlage aus.
res.send()	Sendet eine Antwort mit unterschiedlichen Typen.
res.sendFile	Sendet eine Datei als Oktett-Stream.
res.sendStatus()	Legt den Antwortstatuscode fest und sendet dessen Zeichenfolgedarstellung als Antworthauptteil.

Quelle: http://expressjs.com/de/guide/routing.html

Middleware

- Middleware-Funktionen sind Funktionen, die
  - an beliebiger Stelle in eine Pipeline zwischen Eingang der Anforderung und Erzeugung der Antwort durch den Routing-Handler eingefügt werden können
  - Zugriff auf das Anforderungs- und Antwortobjekt (`request`, `response`) haben
  - einen Verweis (`next`) auf die nächste Middleware-Funktionen übergeben bekommen, um den Kontrollfluss an diese übergeben zu können (oder auch nicht)
- Typische Aufgaben:
  - Logging
  - Zugriffskontrolle
  - Parsen und Aufbereiten der Daten im HTTP-Body
  - …
- Einbindung von Middleware-Funktionen durch Übergabe an die Methode `use()`
  - Optionales erstes Argument: `path`
  - Reihenfolge der Aufrufe von `use()` entspricht Reihenfolge der Middleware-Funktionen in der Pipeline

Middleware

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

let server = express();

function log(request, response, next) {
    console.log(new Date().toUTCString(), request.ip, request.originalUrl);
    next();
}
server.use(log);
//server.use("/json", log);

server.get("/", (request, response) => { ... });
server.get("/json", (request, response) => { ... });

server.listen(8080);
```

Middleware static

- Integrierte Middleware-Funktion `express.static()`
- Bereitstellen statischer Inhalte
  - z.B. zur Auslieferung der Startseite und davon verwendeter Ressourcen (Grafiken, CSS-Dateien, JS-Dateien) einer (Single Page) Web-Anwendung

```
const express = require("express");
const path = require("path");

let server = express();

server.use(express.static(path.join(__dirname,
      "webapp")));
//server.use(express.static("/public",
//      path.join(__dirname, "webapp")));

server.get("/json", (request, response) => { ... });

server.listen(8080);
```

Middleware body-parser

- npm-Modul `body-parser`
- Installation:
  `npm install body-parser`
- Streamt und parst den Body der HTTP-Anfragen, erzeugt daraus ein entsprechendes Objekt und legt es in der Eigenschaft `request.body` ab
- Vereinfacht das Auswerten der Anfrage in POST/PUT-Anfragen

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

let server = express();

server.use(bodyParser.json());

server.post("/data", (request, response) => {
    console.log("POST", request.body.message,
            request.body.user);
    response.send("OK");
});

server.put("/data", (request, response) => {
    console.log("PUT", request.body.message,
            request.body.user);
    response.send("OK");
});

server.listen(8080);
```

Erstellen eines WebSocket-Servers

WebSocket

- Schicht 5-Protokoll, aufbauend auf TCP (wie HTTP)
- Etabliert eine ständige bidirektionale Verbindung
- Handshake ist HTTP-abwärtskompatibel und benötigt keinen zusätzlichen Port
  - Setzt hier auf Node.js-/Express-HTTP-Server auf

WebSocket-Echo-Server

```
// npm install websocket

const WebSocketServer = require("websocket").server;
const http = require("http");

const PORT = 8080;

let httpServer = http.createServer((request, response) => {
    let data = "Hello, World!";
    response.writeHead(200, { "Content-Type": "text/plain", "Content-Length": data.length });
    response.end(data);
}).listen(PORT, () => { console.log("HTTP/WebSocket server listening on port %d.", PORT); });

let webSocketserver = new WebSocketServer({ httpServer: httpServer });
```

WebSocket-Echo-Server

```
let webSocketserver = new WebSocketServer({ httpServer: httpServer });

webSocketserver.on("request", request => {
    let connection = request.accept();

    connection.on("message", function (message) {
        if (message.type === "utf8") {
            console.log("Server received: %s", message.utf8Data);
            connection.sendUTF(message.utf8Data);
        }
    });

    connection.on("close", () => {
        console.log("Client %s closed connection.", connection.remoteAddress);
    });
});
```

Packaging

Packaging von Node.js-Anwendungen

- Das Deployment von serverseitigen Node.js-Anwendungen in Form von Quellcode-Dateien ist (insbesondere bei kommerziellen Anwendungen) häufig nicht gewünscht.
- Lösung: Packaging mithilfe von Pkg
  - https://github.com/zeit/pkg
  - Erzeugt plattformspezifische ausführbare Binärdateien
    - Dekompilierung i.d.R. zu aufwendig, aber nicht ausgeschlossen
    - Im Hex-Editor sind vereinzelt noch Code-Fragment zu erkennen, daher Vorsicht bei hartkodierten Passwörtern etc.
  - Bettet eine Node.js-Laufzeitumgebung und die eigene Anwendung darin ein
    - Binärdatei ist relativ groß, i.d.R. zu groß für eine einfache CLI-Anwendung

Pkg

Pkg

- npm-Modul
  - Kommandozeile
  - Integration in beliebige Build-Tools
- Installation:

  ```
  npm install –g pkg
  ```
- Verwendung:

  ```
  pkg [--targets=node10-linux-x64] [--out-path=pkg] dist/Main.js
  ```
- Target(s), durch Kommata getrennt:
  - Node.js-Version: `node${n}` oder `latest`
  - Plattform: `freebsd`, `linux`, `alpine`, `macos`, `win`
  - Architektur: `x64`, `x86`, `armv6`, `armv7`

Pkg

- JavaScript-Dateien, deren Pfade als Literal an `require()` übergeben werden, werden automatisch eingebettet, z.B.

```
const http = require("http");
const Logger = require("./Logger");
```

- Manuelle Angaben in package.json nötig bei
  - dynamisch erzeugten Pfadangaben, z.B.

```
const module = require("./modules/" + moduleName)
```

  - Nicht-JavaScript-Dateien, die von der Anwendung vorausgesetzt werden, z.B.

```
app.use("webapp", express.static(path.join(path.dirname(
      require.main.filename), "webapp")));
```

Pkg

```
{
   "name": "myserver",
   [...],
   "devDependencies": {
      "pkg": "^4.4.0"
   },
   "scripts": {
      "pkg": "pkg --targets=node10-linux-x64 --out-path=pkg/content ."
   }
   "pkg": {
      "scripts": [
         "dist/modules/**/*.js"
      ]
      "assets": [
         "dist/webapp/**/*"
      ]
   }
}
```

Pkg

- Ausführung:

  ```
  myserver --a --b=10 c
  ```

  entspricht

  ```
  node dist/Main.js --a --b=10 c
  ```

Fragen?