Uni Webdev Backend Summary

Summary for the webdev backend course at HdM Stuttgart

Felicitas Pojtinger

2023-01-28

Meta

Contributing

These study materials are heavily based on professor Toenniessen's "Web Development Backend" lecture at HdM Stuttgart and prior work of fellow students.

Found an error or have a suggestion? Please open an issue on GitHub (github.com/pojntfx/uni-webdev-backend-notes):

Abbildung 1: QR code to source repository

License

Abbildung 2: AGPL-3.0 license badge

SPDX-License-Identifier: AGPL-3.0

Themen der Vorlesung

Themen der Vorlesung

1. **Einführung in Node.js** und einfache HTML-Fileserver
2. **RESTful Endpoints** mit Express.js
3. Die Template-Engine **EJS** und **Express-Sessions**
4. Datenbanken mit **MongoDB und Mongoose**

Einführung in Node.js

Warum Node.js

- Node.js kann **auf dem Server** verwendet werden, im Gegensatz zum Browser
- JavaScript ist die am **häufigsten verwendete Sprache** im Web und durch die Arbeit im Frontend bekannt
- JavaScript eignet sich durch seinen **Event-Loop** besonders gut für HTTP-Server
- Non-Blocking IO ermöglicht es, viele **parallele Anfragen** zu verarbeiten
- Node.js ist **sehr schnell**
- **Einfach zu erlernen**, da das populäre Backend-Framework Express.js auf Node.js aufbaut.
- Aktuelle Version von Node.js ist **16.17.1 LTS (Long Term Support)**
- Bedeutende Anwender sind unter anderem **Microsoft, Yahoo, SAP, PayPal** und viele andere große Unternehmen verwenden Node.js irgendwo.

Module

Node.js hat ein Modul-Konzept, das es ermöglicht, Funktionen und Variablen **in eigene Dateien auszulagern** und sie in anderen Dateien zu importieren.

Ein Beispiel dafür ist die Funktion add(x,y), die in eine **separate Datei** namens 01d_Export.js ausgelagert wird:

```
module.exports = function add(x, y) {
  return x + y;
};
```

In einer anderen Datei, z.B. 01d_AddiererFctModule.js, wird das Modul importiert und verwendet:

```
const add = require("./01d_Export");
const a = 5,
  b = 7;
```

Import & Export mit require/module

Export-Varianten:

a) Einzelne Methode oder Variable:

```
module.exports = function add(x, y) {
  return x + y;
};
```

b) Mehrere Methoden oder Variablen über ein Objekt:

```
module.exports = {
  add: (a, b) => a + b,
  subtract: (a, b) => a - b,
};
```

c) Mehrere einzelne Exports (mit der Convenience-Variable exports):

```
exports.add = (a, b) => a + b;
```

Import & Export mit ES6

Export:

```
export function add(x, y) {
  return x + y;
}

export function subtract(x, y) {
  return x - y;
}
```

Default-Export:

```
export default (x, y) {
  return x - y;
}
```

Import eines gesamten Moduls:

Callbacks vs. Promises vs. Async/Await

Callbacks:

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

fs.readFile("file.txt", function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

Promises:

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

fs.readFile("file.txt")
  .then((data) => console.log(data))
  .catch((err) => console.error(err));
```

Statischer Webserver

Mit Support für ein paar wenige MIME-Types:

```
const http = require("http");
const fs = require("fs");
const { extname } = require("path");

const app = http.createServer((request, response) => {
  fs.readFile(__dirname + request.url, (err, data) => {
    const status = err ? 400 : 200;

    if (extname(request.url) == ".html")
      response.writeHead(200, { status, "Content-Type": "tex
    if (extname(request.url) == ".js")
      response.writeHead(200, { status, "Content-Type": "tex
    if (extname(request.url) == ".css")
      response.writeHead(200, { status, "Content-Type": "tex
```

NPM: Pakete Installieren

- npm ist ein **Paketmanager** für Node.js (wie Maven bei Java oder pip bei Python)
- Mit npm können **Thirdy-Party-Libraries installiert** werden, die auf https://www.npmjs.com gesucht werden können.
- Installierte Pakete können **über require importiert** werden, ohne dass ein relativer Pfad angegeben werden muss.
- **Projektspezifische Installation**: npm install paket-name oder npm i paket-name
- **Globale Installation**: npm i -g paket-name
- **Installation von Entwicklungspaketen**: npm i -D paket-name
- package-lock.json enthält die **exakten Versionen** aller installierten Abhängigkeiten.
- Der node_modules Ordner enthält die **Dateien aller installierten Pakete.**
- Der node_modules Ordner sollte immer von **Git-Commits**

NPM: package.json

- Kann mit npm init erstellt werden
- Unter scripts in der package.json Datei können Command-Line Befehle gespeichert werden, die später ausgeführt werden können, indem man sie in der Kommandozeile aufruft, z.B. npm run start oder npm run test.

NPM: Paketauflösung

1. In einem **relativen Pfad** zur Datei, bis eine “package.json” Datei gefunden wird (npm Projekt Definition) und dort im “node_modules” Ordner
2. In den **global** installierten Paketen

Best Practice: Pakete sollten immer im Projekt installiert werden, damit dort alle Abhängigkeiten definiert sind. CLI-Tools können auch global, z.B. zur Projektinitialisierung, installiert werden.

RESTful Endpoints mit Express.js

Warum REST?

Jahr 2000: Überlastung der Web-Backends (Server)

- Client: Wenig JavaScript
- Backend:
  - HTML-Seiten (statisch)
  - **Rendering von HTML und JSON**
  - Zustand aller User-Dialoge
  - Datenbank-Zugriffe
  - **Komplette Dialogsteuerung** und Kontrolle auf dem Server

Heute: Zustandslose Web-Backends (Server)

- Client: Viel mehr JavaScript (Frameworks)
- Backend:
  - Datenbankzugriffe mit Rückgabe von **JSON-Objekten**
  - **Keine Zustandsverwaltung** einzelner User mehr

Was ist REST?

- REST steht für **“Representational State Transfer”**
  - **Repräsentation**: Darstellung einer Ressource (Daten + Metadaten)
  - **State**: Zustand der Anwendung, gegeben durch die Gesamtheit aller Repräsentationen der angezeigten Daten.
  - **Transfer**: Zustandsübergang durch Aufruf einer Ressource.
- **Roy Fielding** hat es in seiner Doktorarbeit im Jahr 2000 vorgestellt.
- REST **ist ein Architekturparadigma** zur Vereinfachung von verteilten Systemen.
- Es betont …
  - **Skalierbarkeit** der Komponenteninteraktionen
  - Generierung von **Interfaces**
  - **Unabhängige Bereitstellung** von Komponenten
  - Verwendung von **Zwischenkomponenten** um die Interaktionslatenz zu reduzieren, die Sicherheit durchzusetzen und Legacy-Systeme zu encapsulieren.
- REST hat **ursprünglich keine Beziehung zu HTTP** oder speziell

Merkmale einer REST-Architektur

- Client-Server-Modell. Zustandslos: Jeder Request enthält alle Informationen zur Ausführung
- Einheitliche Schnittstelle für die Erstellung, Abfrage, Aktualisierung und Löschung von Ressourcen:
  - **POST**: Erstellen
  - **GET**: Abfragen
  - **PUT**: Aktualisieren
  - **DELETE**: Löschen
- **Ressourcen** sind das zentrale Konzept in REST:
  - Datensätze aus einer Datenbank
  - Textdateien
  - Grafiken
  - Videos
  - Audio-Clips
  - PDF-Dokumente
  - HTML-, CSS- und JS-Dateien von einer Web-Anwendung
  - Services einer SOA

Idempotente Schnittstellen

Sichere und idempotente Schnittstellen:

- **GET**: Read auf eine Ressource
- **PATCH/PUT**: Update auf die Ressource
- **DELETE**: Delete einer Ressource
- **HEAD**: Austausch von Request- und Response-Headern als Zusatzinformation für die übermittelten Daten/Ressourcen (content-size, last-modified, content-type etc.)
- **OPTIONS**: Was kann mit einer Ressource gemacht werden? (Meta-information über mögliche HTTP-Verben.)

Unsichere und nicht-idempotente Schnittstelle: **POST** (Create auf eine Ressource). Im Gegensatz zu DELETE können hier nach einem erneuten Anruf ohne Checks weitere Objekte erstellt werden.

Einheitliche Schnittstellen

- Der **Pfad** einer URL kann statisch sein, z.B. /users, oder Plural.
- **URL-Parameter** sind variabel und werden in der Regel verwendet, um eine eindeutige Identifizierung der Ressource zu ermöglichen.
- **Query-Parameter** sind optionale Key-Value Paare, die in der Regel nur bei GET-Anfragen verwendet werden. Sie ermöglichen z.B. das sortieren von Ressourcen nach bestimmten Kriterien.
- Der **HTTP-Body** wird in der Regel verwendet, um JSON-Daten bei Anfragen wie PUT, POST, PATCH oder DELETE zu übertragen.

Einheitliche Schnittstellen mit REST

Pfad:

```
http://127.0.0.1:3000/fruits
```

```
const DATA = [
  { id: 1, name: "Apfel", color: "gelb,rot" },
  { id: 2, name: "Birne", color: "gelb,grün" },
  { id: 3, name: "Banane", color: "gelb" },
];

app.get("/fruits", (req, res) => {
  res.send(DATA);
});
```

URL-Parameter:

```
http://127.0.0.1:3000/fruits/2
```

Routenpfade in Express

```
app.get("/ab?cd", function (req, res) {
  res.send("ab?cd");
}); // acdabcd

app.get("/ab+cd", function (req, res) {
  res.send("ab+cd");
}); // abcdabbbbcd

app.get("/ab.*cd", function (req, res) {
  res.send("ab.*cd");
}); // abcdabxcd

app.get("/ab(cd)?e", function (req, res) {
  res.send("ab(cd)?e");
}); // /abe und /abcde
```

Middleware in Express

Wildcard-Route:

```
app.all(/.*/, (req, res, next) => {
  console.log(`wildcard-route: ${req.method} ${req.url}`);
  next();
});
```

Middleware (empfohlen):

```
app.use((req, res, next) => {
  console.log(`middleware: ${req.method} ${req.url}`);
  next();
});
```

Die next()-Methode führt immer den nächsten passenden Routen-Handler aus.

Mehrere Callback-Handler

```
let cb0 = function (req, res, next) {
  console.log("CB0");
  next();
};

let cb1 = function (req, res, next) {
  console.log("CB1");
  next();
};

let cb2 = function (req, res) {
  res.send("Hello from CB2!");
};

app.get("/example/c", [cb0, cb1, cb2]);
```

Chaining Routes

Mehrere HTTP-Verben für eine Route können mithilfe von Chaining Routes zusammengefasst werden.

```
app
  .route("/books")
  .get(function (req, res) {
    res.send("Get all books");
  })
  .post(function (req, res) {
    res.send("Add a book");
  });


app
  .route("/books/:id")
  .put(function (req, res) {
    res.send("Update the book");
```

Modularisierung

Modularisierung von Routen in Express kann mithilfe von express.Router erreicht werden.

Erstellung einer Router-Datei birds.js:

```
const express = require("express");
const router = express.Router();

// Middleware
router.use(function timeLog(req, res, next) {
  console.log("Time: ", Date.now());
  next();
});

// Routen
router.get("/", function (req, res) {
```

Weitere Methoden von Express

Result:

- res.status(code): Setzt den HTTP-Statuscode der Antwort (z.B. 200 für erfolgreiche Anfrage, 404 für nicht gefunden)
- res.redirect(url): Leitet den Request an eine andere URL um
- res.cookie(key, value, options): Setzt ein Cookie im Browser des Users, optionale Parameter können angegeben werden wie z.B. die Dauer des Cookies und ob es sicher übertragen werden soll
- res.attachment(path_to_file): Sendet eine Datei als Attachment (z.B. Download)
- res.download(path_to_file): Sendet eine Datei zum Download und zeigt eine entsprechende Benachrichtigung im Browser des Users

Request:

- req.headers(): Gibt ein Objekt mit allen HTTP-Request-Headern zurück

Fehlerhandling

404 als JSON zurückgeben:

```
app.use("/users", require("./routes/users"));
app.use("/products", require("./routes/products"));

// Middleware nach allen Routes
app.use((req, res) => {
  res.status(404);
  res.json({ message: "Not found" });
});
```

Exceptions:

- Wenn in einem Route-Handler eine Exception geworfen wird, sendet Express **standardmäßig eine HTML-Seite mit der Fehlermeldung und dem Stack-Trace** zurück.
- Das kann ein Sicherheitsproblem darstellen, da sensible

HTTP-Verben und HTML-Forms

- In Express kann man HTTP-Verben wie PATCH, PUT oder DELETE auf Endpoints mappen, die jedoch nur GET und POST verstehen
- Eine Lösung dafür ist die Verwendung einer speziellen Middleware wie method-override:

```
const express = require("express");
const methodOverride = require("method-override");
const app = express();

app.use(methodOverride("_method"));
```

Jetzt kann man eine PATCH-Route definieren, die dann auch über ein Formular angesprochen werden kann:

```
app.patch("/fruits", (req, res) => {
  // some code ...
});
```

Die Template-Engine EJS und Express-Sessions

Einführung in EJS

- EJS ist eine Template-Engine für JavaScript
- Ermöglicht die Generierung von HTML-Seiten oder Snippets im Web-Backend
- Express-Server verwendet vorhandene HTML-Templates, füllt diese mit Daten aus der Datenbank, und generiert damit fertiges HTML (ganze Seiten oder Snippets)

Verwendung von EJS

Der Code auf dem **Server**, der die EJS-Template-Engine verwendet, sieht wie folgt aus:

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

app.set("view engine", "ejs");

app.get("/user", (req, res) => {
  const user = {
    name: "John Doe",
    email: "johndoe@example.com",
    phone: "555-555-5555",
  };
  res.render("user-template", { user });
});
```

Schleifen in EJS

Server:

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

app.set("view engine", "ejs");

const DATA = [
  { id: 1, name: "Apfel", color: "gelb,rot" },
  { id: 2, name: "Birne", color: "gelb,grün" },
  { id: 3, name: "Banane", color: "gelb" },
];

app.get("/fruits", (req, res) => {
  res.render("all", { fruits: DATA }); // all.ejs Template
});
```

State mit Cookies durch cookie-parser

- npm-Package cookie-parser ermöglicht zustandsbehaftete Server
- Cookies sind name-value-Paare, gesendet von Server, gespeichert im Browser
- Ermöglichen Identifizierung des Aufrufers bei zukünftigen Requests
- Beispiel: Verwaltung von Warenkorb eines Users auf e-Commerce-Website

So können **Cookies gesetzt** werden:

```
const cookieParser = require("cookie-parser");

app.use(cookieParser());

response.cookie("userID", "xyz12345"); // Einzelner Cookie

response
```

State mit Cookies durch express-session

Mit dem npm-Package express-session kann man zustandsbehaftete Server bauen:

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

const app = express();

app.use(
  session({
    secret: "mykey", // Für Encoding und Decoding des Cookie
    resave: false, // Nur speichern nach Änderung
    saveUninitialized: true, // Anfangs immer speichern
    cookie: { maxAge: 5000 }, // Ablaufzeit in Millisekunden
  })
);
```

Datenbanken mit MongoDB und Mongoose

Datenbanken in Webanwendungen

- **Bisher**: Daten volatil in globalem Array im Backend gespeichert
- **Zukünftig**: Daten persistent in Datenbank im Backend gespeichert
- **Ziel**: Effiziente Verwaltung von Daten, insbesondere bei großen Mengen.

Grundlagen zu MongoDB

- **MongoDB**: Backend-Datenbanksystem für JS objects (hierarchische Dokumente)
- **Einfaches Datenmodell**: Datenbank enthält Collections, die Documents (JS objects) enthalten
- **Analog zu RDB**: Tabellen enthalten Datensätze
- **Vorteil von MongoDB**: Keine Format-Konvertierung von Node.js notwendig, da es sich um eine NoSQL-Datenbank handelt.
- **Achtung**: Kein fixes Datenbankschema in MongoDB, das heißt, in einer Collection können beliebige Datensätze gespeichert werden (dynamisches Schema).
- Das hat **sowohl Vorteile** (einfach und bequem) **als auch Nachteile** (hohe Disziplin der Entwickler erforderlich)
- **Empfehlung: Validierung** der Daten beim Lesen und Speichern **auf Applikationsebene** durchführen
- Beobachtung: **Ähnlichkeit zum OO-Datenbankmodell**, da

Verwendung von MongoDB in Express

Zuerst **mongodb installieren**: npm i –s mongodb

Dann **mit DB verbinden**:

```
let db = null;
const url = `mongodb://localhost:27017`;

MongoClient.connect(url, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then((connection) => {
  db = connection.db("food");
  console.log("connected to database food ...");
});
```

Erstellen einer Collection:

```
app.post("/example-create-collection-fruits", async (req, res
  await db.createCollection("fruits");

  res.send("Collection fruits created ...");
});
```

Löschen einer Datenbank:

```
app.post("/example-drop-db-food", async (req, res) => {
  await db.dropDatabase("food");
  res.send("Database food dropped!");
});
```

db.dropCollection für das ***Löschen einer Collection***

Importieren von Dokumenten:

Projektionen in MongoDB

- **Bisher: Formatierung der Abfrage-Ergebnisse in der Anwendung** durch den Aufruf von `result.map` auf JavaScript-Arrays
- **Ineffizient**, wenn der Endpoint nur einen kleinen Ausschnitt der Objekte liefern soll
- **Lösung:** Verwendung von Projektionen, um Abfrage-Ergebnisse **bereits in der Datenbank** zu **formatieren**, reduziert Traffic zwischen Festplatte und Hauptspeicher.

Ein/Auschluss von Attributen:

```
app.get("/example-fields/restaurants", async (req, res) => {
  const { borough, cuisine } = req.query;

  const restaurants = await db
    .collection("restaurants")
    .find(
      { borough, cuisine }
```

Indizes in MongoDB

Datenbank-Indizes **beschleunigen die Zugriffe für Queries und Updates**, wenn nicht konkret mit der _id gesucht wird.

Anlegen eines Index:

```
await db.collection("restaurants").createIndex({ cuisine: 1
```

Form: {name: 'cuisine_1'}

Anlegen eines kombinierten Index:

```
await db
  .collection("restaurants")
  .createIndex({ cuisine: 1, "address.zipcode": -1 });
```

Form: {name: 'cuisine_1_address.zipcode_-1'}

Abfragen aller Indizes:

```
const indexes = await db.collection("restaurants").getIndexe
```

Bewertung von MongoDB

Vorteile:

- Einfache Schnittstelle
- Mächtige Query-Möglichkeiten
- Gut skalierbar (Mongo-Instanzen, Replica Sets)
- Nahtlose Integration mit JavaScript (JS objects/BSON-Dokumente)

Nachteile:

- Umständliche Schnittstelle
- Fehlendes Datenbank-Schema → Chaos möglich
- Keine semantische Datenmodellierung
- Validierung muss von Anwendung gemacht werden

Grundlagen zu Mongoose

- Mongoose: Eine komfortable Bibliothek über npm-Package mongodb in Node.js
- API sehr ähnlich zum MongoDB-API mit **geringem Lernaufwand** und **ES6-Klassen**
- Ermöglicht Datenbank-Schemata, **semantische Datenmodellierung mit Validierung** der Daten
- Vereinfachte und **einheitliche** Query-Schnittstelle
- Sehr **mächtig**

Datenbankverbindung in Mongoose herstellen

Ist sehr ähnlich zu MongoDB:

```
const url = "mongodb://localhost:27017/food_mongoose";

mongoose
  .connect(url, { useNewUrlParser: true, useUnifiedTopology:
  .then(() => {
    console.log("connected to database food_mongoose ...");
  });
```

Schematas in Mongoose

Definition von Schemata:

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

const fruitSchema = new mongoose.Schema({
  name: { type: String, required: true },
  color: { type: String, required: true },
  img: { data: Buffer, contentType: String },
});

const Fruit = mongoose.model("Fruit", fruitSchema); // `Fruit
```

Schema Types aus ES6:

```
const schemaExample = new mongoose.Schema({
  bool: Boolean,
  updated: Date,
```

Queries in Mongoose

Auslesen aller Dokumente:

- Queries liefern nicht nur einfache JS-Objekte, sondern **intelligente mongoose-Dokumente**
- Diese Dokumente **haben zusätzliche Methoden und Attribute** im Vergleich zu JS-Objekten
- Fruit.find().lean() liefert nur einfache JS-Objekte für bessere Performance

```
app.get("/example-list/fruits", async (req, res) => {
  const fruits = await Fruit.find();

  res.send(fruits);
});
```

Hinzufügen eines Dokuments:


```
app.post("/example-create/fruits", async (req, res) => {
```

Validation in Mongoose

Eingebaute Validatoren:

```
const breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, "Too few eggs"],
    max: 12,
  },
  bacon: {
    type: Number,
    required: [true, "Why no bacon?"],
  },
  drink: {
    type: String,
    enum: ["Coffee", "Tea"],
    required: function () {
```