
Uni Webdev Backend Summary

Summary for the webdev backend course at HdM
Stuttgart

Felicitas Pojtinger

2023-01-28

Inhaltsverzeichnis

1	Meta	3
1.1	Contributing	3
1.2	License	3
2	Themen der Vorlesung	4
3	Einführung in Node.js	4
3.1	Warum Node.js	4
3.2	Module	4
3.3	Import & Export mit <code>require/module</code>	5
3.4	Import & Export mit ES6	6
3.5	Callbacks vs. Promises vs. Async/Await	7
3.6	Statischer Webserver	7
3.7	NPM: Pakete Installieren	9
3.8	NPM: <code>package.json</code>	9
3.9	NPM: Paketauflösung	9
4	RESTful Endpoints mit Express.js	9
4.1	Warum REST?	9
4.2	Was ist REST?	10
4.3	Merkmale einer REST-Architektur	10
4.4	Idempotente Schnittstellen	11
4.5	Einheitliche Schnittstellen	11
4.6	Einheitliche Schnittstellen mit REST	12
4.7	Routenpfade in Express	14
4.8	Middleware in Express	14
4.9	Mehrere Callback-Handler	15
4.10	Chaining Routes	15
4.11	Modularisierung	16
4.12	Weitere Methoden von Express	16
4.13	Fehlerhandling	17
4.14	HTTP-Verben und HTML-Forms	18
5	Die Template-Engine EJS und Express-Sessions	19
5.1	Einführung in EJS	19
5.2	Verwendung von EJS	19
5.3	Schleifen in EJS	20

5.4	State mit Cookies durch cookie-parser	21
5.5	State mit Cookies durch express-session	21
6	Datenbanken mit MongoDB und Mongoose	22
6.1	Datenbanken in Webanwendungen	22
6.2	Grundlagen zu MongoDB	22
6.3	Verwendung von MongoDB in Express	23
6.4	Queries in MongoDB	23
6.5	Projektionen in MongoDB	27
6.6	Indizes in MongoDB	29
6.7	Bewertung von MongoDB	30
6.8	Grundlagen zu Mongoose	30
6.9	Datenbankverbindung in Mongoose herstellen	30
6.10	Schematas in Mongoose	30
6.11	Queries in Mongoose	33
6.12	Validation in Mongoose	34

1 Meta

1.1 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

If you like the study materials, a GitHub star is always appreciated :)

1.2 License



Abbildung 2: AGPL-3.0 license badge

Uni Webdev Backend Notes (c) 2023 Felicitas Pojtinger and contributors

SPDX-License-Identifier: AGPL-3.0

2 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**

3 Einführung in Node.js

3.1 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 Front-end 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.
- Weitere Pakete können über den **Node Package Manager (npm)** hinzugefügt werden

Vorteile:

Sehr einfache APIs, **schnell zu lernen**

Nachteile:

- Etwas **kompliziertes Programmiermodell** (Event-basiert), typisch für JavaScript
- **Multithreading** in Node.js nur über Worker-Threads möglich

3.2 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:

```
1 module.exports = function add(x, y) {  
2   return x + y;  
3 };
```

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

```
1 const add = require("./01d_Export");  
2 const a = 5,  
3   b = 7;  
4 const s = add(a, b);  
5 console.log(`${a} + ${b} = ${s}`);
```

- Mit Modulen kann der Code bei größeren Programmen übersichtlicher gestaltet werden und es ist **weniger fehleranfällig bei Änderungen** oder in der Wartung.
- Durch die Trennung des Codes in Module, **erhöht sich die Übersichtlichkeit** innerhalb des jeweiligen Moduls
- Einzelne Module können für sich finalisiert oder refaktoriert werden, **ohne dass es Auswirkungen auf den Rest des Codes** hat.
- Durch die Aufteilung des Codes in Module, wird es einfacher, die Arbeit unter Teammitgliedern aufzuteilen, was wiederum zu **weniger Merge-Konflikten** führt.

3.3 Import & Export mit require/module

Export-Varianten:

a) Einzelne Methode oder Variable:

```
1 module.exports = function add(x, y) {  
2   return x + y;  
3 };
```

b) Mehrere Methoden oder Variablen über ein Objekt:

```
1 module.exports = {  
2   add: (a, b) => a + b,  
3   subtract: (a, b) => a - b,  
4 };
```

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

```
1 exports.add = (a, b) => a + b;  
2 exports.subtract = (a, b) => a - b;
```

Wenn Sie jedoch **module.exports** direkt zuweisen, werden alle vorherigen Exporte überschrieben.

Import-Varianten:

a) Gesamtes Modul importieren:

```
1 const fs = require("fs");
2 fs.readFile();
3
4 const readFile = require("fs").readFile;
5 readFile();
```

b) Destrukturierende Zuweisung:

```
1 const { readFile } = require("fs");
2 readFile();
3
4 const { readFile, ...fs } = require("fs");
5 readFile();
6 fs.writeFile();
```

3.4 Import & Export mit ES6

Export:

```
1 export function add(x, y) {
2   return x + y;
3 }
4
5 export function subtract(x, y) {
6   return x - y;
7 }
```

Default-Export:

```
1 export default (x, y) {
2   return x - y;
3 }
```

Import eines gesamten Moduls:

```
1 import * as math from "./math.js";
2
3 console.log(math.add(5, 2)); // Ausgabe: 7
4 console.log(math.subtract(5, 2)); // Ausgabe: 3
```

Import mit destrukturierenden Zuweisung:

```
1 import { add as addition, subtract } from "./math.js";
2
3 console.log(addition(5, 2)); // Ausgabe: 7
4 console.log(subtract(5, 2)); // Ausgabe: 3
```

3.5 Callbacks vs. Promises vs. Async/Await

Callbacks:

```
1 const fs = require("fs");
2
3 fs.readFile("file.txt", function (err, data) {
4   if (err) throw err;
5   console.log(data);
6 });
```

Promises:

```
1 const fs = require("fs").promises;
2
3 fs.readFile("file.txt")
4   .then((data) => console.log(data))
5   .catch((err) => console.error(err));
```

async/await:

```
1 const fs = require("fs").promises;
2
3 async function readFileExample() {
4   try {
5     const data = await fs.readFile("file.txt");
6     console.log(data);
7   } catch (err) {
8     console.error(err);
9   }
10 }
11
12 readFileExample();
```

3.6 Statischer Webserver

Mit Support für ein paar wenige MIME-Types:

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



```
4
5 const app = http.createServer((request, response) => {
6   fs.readFile(__dirname + request.url, (err, data) => {
7     const status = err ? 400 : 200;
8
9     if (extname(request.url) == ".html")
10      response.writeHead(200, { status, "Content-Type": "text/html" });
11    if (extname(request.url) == ".js")
12      response.writeHead(200, { status, "Content-Type": "text/
13        javascript" });
14    if (extname(request.url) == ".css")
15      response.writeHead(200, { status, "Content-Type": "text/css" });
16
17    response.write(data);
18    response.end();
19  });
20 });
21 app.listen(3000);
22
23 console.log("Listening on :3000");
```

Mit Support für ein alle MIME-Types:

```
1 $ npm install node-static
```

```
1 const http = require("http");
2 const fileserver = new (require("node-static").Server)();
3
4 const app = http.createServer((request, response) => {
5   fileserver.serve(request, response);
6 });
7
8 app.listen(3000);
9
10 console.log("Listening on :3000");
```

Mit Support für ein alle MIME-Types & Express:

```
1 $ npm install express
```

```
1 const express = require("express");
2 const app = express();
3
4 app.use("/WDBackend", express.static(__dirname + "/public"));
5 app.listen(3000);
6
7 console.log("Listening on :3000");
```

3.7 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 ausgeschlossen werden**
- Die `package-lock.json` sollte hingegen **immer committed** werden.

3.8 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`.

3.9 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.

4 RESTful Endpoints mit Express.js

4.1 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

4.2 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 gestalteten URLs.

4.3 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
- Jede Ressource hat eine **eindeutige URI/URL**
- Jede Ressource trägt **Caching-Informationen**

4.4 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.

4.5 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.

4.6 Einheitliche Schnittstellen mit REST

Pfad:

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

```
1 const DATA = [  
2   { id: 1, name: "Apfel", color: "gelb,rot" },  
3   { id: 2, name: "Birne", color: "gelb,grün" },  
4   { id: 3, name: "Banane", color: "gelb" },  
5 ];  
6  
7 app.get("/fruits", (req, res) => {  
8   res.send(DATA);  
9 });
```

URL-Parameter:

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

```
1 const DATA = [  
2   { id: 1, name: "Apfel", color: "gelb,rot" },  
3   { id: 2, name: "Birne", color: "gelb,grün" },  
4   { id: 3, name: "Banane", color: "gelb" },  
5 ];  
6  
7 app.get("/fruits/:id", (req, res) => {  
8   const id = parseInt(req.params.id);  
9  
10  const item = DATA.find((o) => o.id === id);  
11  
12  res.send(item);  
13 });
```

Query-Parameter:

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

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

```
5 ];
6
7 app.get("/fruits", (req, res) => {
8   const id = parseInt(req.query.id);
9
10  const item = DATA.find((o) => o.id === id);
11
12  res.send(item);
13 });
```

HTTP-Body (URL-Encoded)

Nutze `express.urlencoded`

```
1 app.use(express.urlencoded({ extended: true }));
2
3 const DATA = [{ id: 1, name: "Apfel", color: "gelb,rot" }];
4
5 app.post("/fruits", (req, res) => {
6   const { name, color } = req.body;
7
8   if (DATA.find((o) => o.name === name)) {
9     res.send("Duplicate name");
10  } else {
11    const id = Math.max(...DATA.map((o) => o.id)) + 1;
12    const fruit = { id, name, color };
13    DATA.push(fruit);
14    res.send(fruit);
15  }
16 });
```

HTTP-Body (JSON)

Nutze `express.json`

```
1 app.use(express.json());
2
3 const DATA = [{ id: 1, name: "Apfel", color: "gelb,rot" }];
4
5 app.post("/fruits", (req, res) => {
6   const { name, color } = req.body;
7
8   if (DATA.find((o) => o.name === name)) {
9     res.send("Duplicate name");
10  } else {
11    const id = Math.max(...DATA.map((o) => o.id)) + 1;
12    const fruit = { id, name, color };
13    DATA.push(fruit);
14    res.send(fruit);
15  }
```

```
16 });
```

4.7 Routenpfade in Express

```
1 app.get("/ab?cd", function (req, res) {
2   res.send("ab?cd");
3 }); // acdabcd
4
5 app.get("/ab+cd", function (req, res) {
6   res.send("ab+cd");
7 }); // abcdabbbbcd
8
9 app.get("/ab.*cd", function (req, res) {
10  res.send("ab.*cd");
11 }); // abcdabxcd
12
13 app.get("/ab(cd)?e", function (req, res) {
14  res.send("ab(cd)?e");
15 }); // /abe und /abcde
16
17 app.get(/a/, function (req, res) {
18  res.send("/a/");
19 }); // alles mit 'a' drin
20
21 app.get(/.*fly$/, function (req, res) {
22  res.send(/.*fly$/);
23 });
```

4.8 Middleware in Express

Wildcard-Route:

```
1 app.all(/.*/, (req, res, next) => {
2   console.log(`wildcard-route: ${req.method} ${req.url}`);
3   next();
4 });
```

Middleware (empfohlen):

```
1 app.use((req, res, next) => {
2   console.log(`middleware: ${req.method} ${req.url}`);
3   next();
4 });
```

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

4.9 Mehrere Callback-Handler

```
1 let cb0 = function (req, res, next) {
2   console.log("CB0");
3   next();
4 };
5
6 let cb1 = function (req, res, next) {
7   console.log("CB1");
8   next();
9 };
10
11 let cb2 = function (req, res) {
12   res.send("Hello from CB2!");
13 };
14
15 app.get("/example/c", [cb0, cb1, cb2]);
```

Wichtig: `next` nicht vergessen!

4.10 Chaining Routes

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

```
1 app
2   .route("/books")
3   .get(function (req, res) {
4     res.send("Get all books");
5   })
6   .post(function (req, res) {
7     res.send("Add a book");
8   });
9
10 app
11   .route("/books/:id")
12   .put(function (req, res) {
13     res.send("Update the book");
14   })
15   .delete(function (req, res) {
16     res.send("Delete the book");
17   });
```

Vorteile: weniger fehleranfällig, leichter zu pflegen (da man die Route nur einmal schreibt)

4.11 Modularisierung

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

Erstellung einer Router-Datei `birds.js`:

```
1 const express = require("express");
2 const router = express.Router();
3
4 // Middleware
5 router.use(function timeLog(req, res, next) {
6   console.log("Time: ", Date.now());
7   next();
8 });
9
10 // Routen
11 router.get("/", function (req, res) {
12   res.send("Birds home page");
13 });
14
15 router.get("/about", function (req, res) {
16   res.send("About birds");
17 });
18
19 module.exports = router;
```

Einbindung des Routers in die Anwendung `app.js`:

```
1 const express = require("express");
2 const app = express();
3 const birds = require("./birds");
4
5 app.use("/birds", birds);
6
7 app.listen(3000);
8
9 console.log("Listening on :3000");
```

4.12 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 übertra-

gen 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
- `req.cookies()`: Gibt ein Objekt mit allen Cookies zurück, die im Request enthalten sind (benötigt die Middleware `cookie-parser`)

4.13 Fehlerhandling

404 als JSON zurückgeben:

```
1 app.use("/users", require("./routes/users"));
2 app.use("/products", require("./routes/products"));
3
4 // Middleware nach allen Routes
5 app.use((req, res) => {
6   res.status(404);
7   res.json({ message: "Not found" });
8 });
```

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 Informationen preisgegeben werden können. Eine Lösung wäre, stattdessen eine vernünftige JSON-response zu senden:

```
1 try {
2   throw new Error("Something went wrong");
3 } catch (err) {
4   res.status(500).json({ message: "InternalServerError" });
5 }
```

- Ein größeres Problem entsteht, wenn **eine Exception in einem Promise auftritt**, da es zu einem globalen Fehler `UnhandledPromiseRejection` im Node-Prozess kommt und keine Response gesendet wird.
- In zukünftigen Node-Versionen wird dieser Fehler nicht mehr global abgefangen und stattdessen der Prozess mit einem Fehlercode beendet, was zu einem **Absturz der gesamten Server-Anwendung** führen kann.

- **Lösung:** Route-Handler in try/catch packen, Exceptions der next-Funktion übergeben und eine eigene Error-Middleware einbauen.

```
1 app.get("/", async (req, res, next) => {
2   try {
3     throw new Error("Something went wrong");
4   } catch (err) {
5     next(err);
6   }
7 });
8
9 app.use((err, req, res, next) => {
10   res.status(500);
11   res.json({ message: "InternalServerError" });
12   console.error(err);
13 });
```

4.14 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:

```
1 const express = require("express");
2 const methodOverride = require("method-override");
3 const app = express();
4
5 app.use(methodOverride("_method"));
```

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

```
1 app.patch("/fruits", (req, res) => {
2   // some code ...
3 });
```

In dem Formular muss dann der URL-Parameter `_method=patch` hinzugefügt werden:

```
1 <form action="/fruits?_method=patch" method="post">...</form>
```

Jetzt wird die PATCH-Route aufgerufen, wenn das Formular abgeschickt wird.

5 Die Template-Engine EJS und Express-Sessions

5.1 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)

5.2 Verwendung von EJS

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

```
1  const express = require("express");
2  const app = express();
3
4  app.set("view engine", "ejs");
5
6  app.get("/user", (req, res) => {
7    const user = {
8      name: "John Doe",
9      email: "johndoe@example.com",
10     phone: "555-555-5555",
11   };
12   res.render("user-template", { user });
13 });
```

Das **Template** `template.ejs` im Unterverzeichnis `views`, welcher ein JavaScript-Objekt `{vorname, adresse, telefon}` übergeben wird:

```
1  <html>
2    <body>
3      <h1>User Information</h1>
4      <table>
5        <tr>
6          <td>Name:</td>
7          <td><%= user.name %></td>
8        </tr>
9        <tr>
10         <td>Email:</td>
11         <td><%= user.email %></td>
12       </tr>
13       <tr>
14         <td>Phone:</td>
15         <td><%= user.phone %></td>
16       </tr>
17     </table>
```

```
18 </body>
19 </html>
```

5.3 Schleifen in EJS

Server:

```
1  const express = require("express");
2  const app = express();
3
4  app.set("view engine", "ejs");
5
6  const DATA = [
7    { id: 1, name: "Apfel", color: "gelb,rot" },
8    { id: 2, name: "Birne", color: "gelb,grün" },
9    { id: 3, name: "Banane", color: "gelb" },
10 ];
11
12 app.get("/fruits", (req, res) => {
13   res.render("all", { fruits: DATA }); // all.ejs Template
14 });
15
16 app.get("/fruits/:id", (req, res) => {
17   const id = parseInt(req.params.id);
18   const fruit = DATA.find((o) => o.id === id);
19   res.render("fruit", fruit); // fruit.ejs Template
20 });
21 app.listen(3000);
22
23 console.log("EJS server running on localhost:3000");
```

Template:

```
1 <html>
2   <body>
3     <table>
4       <tr>
5         <th>Name</th>
6         <th>Farbe</th>
7       </tr>
8       <% fruits.forEach( o => { %>
9         <tr>
10          <td><%= o.name %></td>
11          <td><%= o.color %></td>
12        </tr>
13      <% }) %>
14    </table>
15  </body>
16 </html>
```

5.4 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:

```
1 const cookieParser = require("cookie-parser");
2
3 app.use(cookieParser());
4
5 response.cookie("userID", "xyz12345"); // Einzelner Cookie
6
7 response
8   .cookie("userID", "xyz12345")
9   .cookie("verein", "VfB Stuttgart", { maxAge: 90000 }); // Mehrere
    Cookies, der zweite mit 90000 milli secs Lebensdauer
```

So können **Cookies ausgelesen** werden:

```
1 const cookieParser = require("cookie-parser");
2
3 app.use(cookieParser());
4
5 const cookies = request.cookies;
6
7 let userID = cookies.userID;
8 let verein = cookies.verein;
```

5.5 State mit Cookies durch express-session

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

```
1 const express = require("express");
2 const session = require("express-session");
3
4 const app = express();
5
6 app.use(
7   session({
8     secret: "mykey", // Für Encoding und Decoding des Cookies
9     resave: false, // Nur speichern nach Änderung
10    saveUninitialized: true, // Anfangs immer speichern
11    cookie: { maxAge: 5000 }, // Ablaufzeit in Millisekunden
12  })
13 )
```

```
13 );
14
15 app.get("/", function (req, res) {
16   if (req.session.count) {
17     // Eine Session kann beliebige Attribute bekommen
18     req.session.count++;
19     res.setHeader("Content-Type", "text/html");
20     res.write("<p>count: " + req.session.count + "</p>");
21     res.end();
22   } else {
23     req.session.count = 1;
24     res.end("Willkommen zu der Sitzung. Refresh!");
25   }
26 });
```

6 Datenbanken mit MongoDB und Mongoose

6.1 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.

6.2 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 Objekt-Beziehungen direkt gespeichert werden, anstatt über mehrere Tabellen mit Fremdschlüsseln.

6.3 Verwendung von MongoDB in Express

Zuerst **mongodb** installieren: `npm i -s mongodb`

Dann **mit DB verbinden**:

```
1 let db = null;
2 const url = `mongodb://localhost:27017`;
3
4 MongoClient.connect(url, {
5   useNewUrlParser: true,
6   useUnifiedTopology: true,
7 }).then((connection) => {
8   db = connection.db("food");
9   console.log("connected to database food ...");
10 });
```

6.4 Queries in MongoDB

Erstellen einer Collection:

```
1 app.post("/example-create-collection-fruits", async (req, res) => {
2   await db.createCollection("fruits");
3
4   res.send("Collection fruits created ...");
5 });
```

Löschen einer Datenbank:

```
1 app.post("/example-drop-db-food", async (req, res) => {
2   await db.dropDatabase("food");
3   res.send("Database food dropped!");
4 });
```

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

Importieren von Dokumenten:

```
1 $ mongoimport --db tools-test --collection restaurants --file
   restaurants.json
```

Exportieren von Dokumenten:

```
1 $ mongoexport --db tools-test --collection restaurants --out new-
   restaurants.json
```

Einfügen eines Dokument:


```
1 app.post("/example-create/fruits", async (req, res) => {
2   const { name, color } = req.body;
3   const fruit = { name, color };
4
5   await db.collection("fruits").insertOne(fruit);
6
7   res.send(`${name} inserted ...`);
8 });
```

Auslesen eines Dokuments:

```
1 app.get("/example-find-one/fruits/:name", async (req, res) => {
2   const { name } = req.params;
3
4   const fruit = await db.collection("fruits").findOne({ name });
5
6   if (fruit) {
7     res.send(fruit);
8   } else {
9     res.status(400).send("not found ...");
10  }
11 });
```

Einfügen mehrerer Dokumente:

```
1 app.post("/example-insert-many/fruits", async (req, res) => {
2   const fruits = [
3     { name: "Apfel", color: "gelb,rot" },
4     { name: "Birne", color: "gelb,grün" },
5     { name: "Kiwi", color: "grün" },
6     { name: "Banane", color: "gelb" },
7     { name: "Pfirsich", color: "gelb,rot" },
8   ];
9
10  await db.collection("fruits").insertMany(fruits);
11
12  res.send("Fruits inserted");
13 });
```

Auslesen aller Dokumente:

```
1 app.get("/example-list/fruits", async (req, res) => {
2   const fruits = await db.collection("fruits").find().toArray();
3
4   res.send(fruits);
5 });
```

Löschen eines Dokuments:

```
1 app.delete("/example-delete/fruits/:name", async (req, res) => {
```

```
2   const { name } = req.params;
3
4   const result = await db.collection("fruits").deleteOne({ name });
5
6   if (result.deletedCount > 0) {
7     res.send(` ${name} deleted ...`);
8   } else {
9     res.status(400).send("fruit not found, nothing to delete ...");
10  }
11  });
```

Löschen mehrerer Dokumente:

```
1  db.collection("fruits").deleteMany({ name: { $regex: name } });
```

Aktualisierung von Dokumenten:

```
1  app.patch("/example-update-cuisine/:name", async (req, res) => {
2    const { name } = req.params;
3    const { cuisine } = req.body;
4
5    const result = await db.collection("restaurants").updateOne(
6      { name },
7      {
8        $set: { cuisine },
9        $currentDate: { lastModified: true }, // Änderungsdatum
10     }
11   );
12
13   res.send(result);
14 });
```

Hinzufügen von Arrayelementen in Dokumenten:

```
1  app.post("/example-push-grade-score/restaurants/:name", async (req, res) => {
2    const { name } = req.params;
3    const { grade, score } = req.body;
4    const newGrade = { date: new Date(), grade, score };
5
6    const result = await db.collection("restaurants").updateOne(
7      { name },
8      (update = {
9        $push: { grades: newGrade },
10        $currentDate: { lastModified: true },
11      })
12    );
13
14   res.send(result);
15 });
```

Löschen von Arrayelementen in Dokumenten:

```
1 app.post("/example-pop-grade-score/restaurants/:name", async (req, res) => {
2   const { name } = req.params;
3   const query = { name };
4
5   const result = await db.collection("restaurants").updateOne(query, {
6     $pop: { grades: 1 },
7     $currentDate: { lastModified: true },
8   });
9
10  res.send(result);
11 });
```

- `$pop {<array>: 1}` entfernt das letzte Element,
- `$pop {<array>: -1}` entfernt das erste Element des Arrays.

Tiefe Queries:

```
1 app.get("/example-zip/restaurants", async (req, res) => {
2   const { cuisine, zip } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .find({ "address.zipcode": zip, cuisine })
7     .toArray();
8
9   res.send(restaurants.map((o) => ({ name: o.name, zip: o.address.
10     zipcode })));
11 });
```

BSON für Vergleichsoperatoren:

```
1 app.get("/example-zip-range/restaurants", async (req, res) => {
2   const { zipMin, zipMax, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .find({
7       cuisine,
8       "address.zipcode": { $gte: zipMin, $lt: zipMax }, // es gibt auch
9       $eq, $in, $neq, $nin ...
10     })
11     .toArray();
12
13   res.send(restaurants.map((o) => ({ name: o.name, zip: o.address.
14     zipcode })));
15 });
```

BSON für Oder-Verknüpfung:

```
1 app.get("/example-zip-or-cuisine/restaurants", async (req, res) => {
2   const { zip, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .find(
7       { $or: [{ "address.zipcode": zip }, { cuisine } ] } // es gibt
8         auch noch $and, $not und $nor
9     )
10    .toArray();
11  res.send(
12    restaurants.map((o) => ({
13      name: o.name,
14      cuisine: o.cuisine,
15      zip: o.address.zipcode,
16    }))
17  );
18 });
```

6.5 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/Ausschluss von Attributen:

```
1 app.get("/example-fields/restaurants", async (req, res) => {
2   const { borough, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .find(
7       { borough, cuisine },
8       { projection: { name: 1, address: 1, _id: 0 } } // 1 bedeutet
9         Einschluss eines Attributs, 0 den Ausschluss
10    )
11    .toArray();
12  res.send(restaurants);
13 });
```

Sortieren:

```
1 app.get("/example-sort/restaurants", async (req, res) => {
2   const { borough, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .find({ borough, cuisine }, { projection: { name: 1, address: 1,
7       _id: 0 } })
8     .sort({ name: 1 }) // Mit 1 wird aufsteigend sortiert, mit 0
7     absteigend.
8     .toArray();
9   res.send(restaurants);
10 });
```

Aggregation:

```
1 app.get("/example-avg-score/restaurants", async (req, res) => {
2   const { borough, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .aggregate([
7     {
8       // Wie `WHERE` in SQL
9       $match: { borough, cuisine },
10    },
11    {
12      // Wie `SELECT` in SQL
13      $project: {
14        name: "$name", // auch name: 1 möglich
15        avg_score: { $avg: "$grades.score" },
16      }, // auch $min, $max, $sum
17    },
18    { $sort: { name: 1 } },
19  ])
20   .toArray();
21
22   res.send(restaurants);
23 });
```

Gruppierung:

```
1 app.get("/example-group/restaurants", async (req, res) => {
2   const { borough, cuisine } = req.query;
3
4   const restaurants = await db
5     .collection("restaurants")
6     .aggregate([
7     {
8       $match: { borough, cuisine },
9     },
```

```
10     {
11         // Wie `GROUP BY` in SQL
12         $group: {
13             _id: "$address.zipcode",
14             count: { $sum: 1 }, // auch $min, $max, $avg
15         },
16     },
17     { $sort: { _id: 1 } },
18 ]
19 .toArray();
20
21 res.send(restaurants);
22 });
```

6.6 Indizes in MongoDB

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

Anlegen eines Index:

```
1 await db.collection("restaurants").createIndex({ cuisine: 1 }); // 1:
    Aufsteigend, -1: Absteigend
```

Form: {name: 'cuisine_1'}

Anlegen eines kombinierten Index:

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

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

Abfragen aller Indizes:

```
1 const indexes = await db.collection("restaurants").getIndexes();
```

Löschen eines Index:

```
1 const result = await db.collection("restaurants").dropIndex("cuisine_1"
    ); // 0: not ok, 1: success
```

Löschen aller Indizes:

```
1 const result = await db.collection("restaurants").dropIndexes();
```

6.7 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

6.8 Grundlagen zu Mongoose

- Mongoose: Eine komfortable Bibliothek über npm-Package `mongoose` 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**

6.9 Datenbankverbindung in Mongoose herstellen

Ist sehr ähnlich zu MongoDB:

```
1 const url = "mongodb://localhost:27017/food_mongoose";
2
3 mongoose
4   .connect(url, { useNewUrlParser: true, useUnifiedTopology: true })
5   .then(() => {
6     console.log("connected to database food_mongoose ...");
7   });
```

6.10 Schematas in Mongoose

Definition von Schemata:

```
1 const mongoose = require("mongoose");
2
3 const fruitSchema = new mongoose.Schema({
4   name: { type: String, required: true },
5   color: { type: String, required: true },
6   img: { data: Buffer, contentType: String },
7 });
8
9 const Fruit = mongoose.model("Fruit", fruitSchema); // `Fruit` maps to
   a collection `fruits`
```

Schema Types aus ES6:

```
1 const schemaExample = new mongoose.Schema({
2   bool: Boolean,
3   updated: Date,
4   age: Number,
5   array: [],
6   arrayOfString: [String],
7   arrayOfArrays: [[]],
8   arrayOfArrayOfNumbers: [[Number]],
9   map: Map,
10  mapOfString: { type: Map, of: String },
11 });
```

Weitere Schema Types aus MongoDB:

```
1 const schemaExample = new mongoose.Schema({
2   mixed: mongoose.Mixed, // Kann alles sein (`any`)
3   _someId: mongoose.ObjectId, // Explizite MongoDB-Id
4   decimal: mongoose.Decimal128, // 128-bit Floating-Point; `mongoose.
   Types.Decimal128.fromString('3.1415')`
5 });
```

Indizes in Schemas:

```
1 const schemaExample = new mongoose.Schema({
2   name: {
3     type: String,
4     required: true, // Feld zwingend notwendig
5     index: true, // Index wird automatisch angelegt
6   },
7 });
```

Options-Objekt:

```
1 const schemaExample = new mongoose.Schema(
2   {
3     name: String,
4     age: Number,
```



```
5   },
6   {
7     // Das Options-Objekt
8     timestamps: true, // Automatisch `createdAt` und `modifiedAt`
                        // verwalten
9   }
10 );
```

Verschachtelte Schemata mit Kopien:

```
1  const ratingSchema = new mongoose.Schema({
2    grade: {
3      type: Number,
4      min: 1,
5      max: 6,
6    },
7    comment: String,
8    date: Date,
9  });
10
11 const productSchema = new mongoose.Schema({
12   name: { type: String, required: true },
13   price: { type: String, required: true },
14   ratings: [ratingSchema],
15 });
16
17 const Rating = mongoose.model("Rating", ratingSchema);
18 const Product = mongoose.model("Product", productSchema);
```

Verschachtelte Schemata mit Referenzen:

```
1  const productSchema = new mongoose.Schema({
2    name: String,
3    category: {
4      // 1:1-Beziehung
5      type: mongoose.ObjectId,
6      ref: "Category",
7    },
8  });
9
10 const Product = mongoose.model("Product", productSchema);
11
12 const categorySchema = new mongoose.Schema({
13   name: String,
14   products: [
15     // 1:n-Beziehung
16     {
17       type: mongoose.ObjectId,
18       ref: "Product",
19     },
20   ],
21 });
```

```
21 });  
22  
23 const Category = mongoose.model("Category", categorySchema);
```

Auslesen von Referenzen mit `populate` (wie SQL JOIN): `await Product.find().populate("category")`

6.11 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

```
1 app.get("/example-list/fruits", async (req, res) => {  
2   const fruits = await Fruit.find();  
3  
4   res.send(fruits);  
5 });
```

Hinzufügen eines Dokuments:

```
1 app.post("/example-create/fruits", async (req, res) => {  
2   const { name, color } = req.body;  
3   const doc = await Fruit.findOne({ name });  
4   if (doc) {  
5     res.status(400).send("fruit found, delete first ...");  
6  
7     return;  
8   }  
9  
10  const fruit = new Fruit({ name, color });  
11  
12  const imgPath = path.join(IMAGE_PATH, `${name}.png`);  
13  try {  
14    fruit.img.data = await fs.readFile(imgPath);  
15    fruit.img.contentType = "image/png";  
16  } catch (err) {  
17    console.log(`No image for ${name} found.`);  
18  }  
19  
20  await fruit.save();  
21  
22  res.send(`${name} inserted ...`);  
23 });
```

Umwandlung von MongoDB-Queries in Mongoose-Queries:

```
1 app.get("/example-group-by-color/fruits/:color", async (req, res) => {
2   const { color } = req.params;
3   const result = await Fruit.aggregate([
4     {
5       $match: { color: { $regex: color } },
6     },
7     {
8       $group: {
9         _id: "$color",
10        count: { $sum: 1 },
11      },
12    },
13    { $sort: { _id: 1 } },
14  ]);
15  res.send(result);
16 });
```

Ersetze `db.collection("fruits")` durch `Fruit`

6.12 Validation in Mongoose

Eingebaute Validatoren:

```
1 const breakfastSchema = new Schema({
2   eggs: {
3     type: Number,
4     min: [6, "Too few eggs"],
5     max: 12,
6   },
7   bacon: {
8     type: Number,
9     required: [true, "Why no bacon?"],
10  },
11  drink: {
12    type: String,
13    enum: ["Coffee", "Tea"],
14    required: function () {
15      return this.bacon > 3;
16    },
17  },
18 });
```

Eigene Validatoren:

```
1 function myValidator(val) {
2   return val === "something";
3 }
4
```

```
5 new mongoose.Schema({ name: { type: String, validate: myValidator } });
6
7 // Hinzufügen einer Error-Message, {PATH} ist der fehlerhafte Pfad im
  Schema:
8 const customValidator = [
9   myValidator,
10  'Ups, {PATH} does not equal "something".',
11 ];
12
13 new mongoose.Schema({ name: { type: String, validate: customValidator }
    });
```