# Uni Webdev Backend Summary

Summary for the webdev backend course at HdM Stuttgart

Felicitas Pojtinger

2023-01-28

# Inhaltsverzeichnis

# 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:

```
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;
const s = add(a, b);
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:

```
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;
exports.subtract = (a, b) => a - b;
```

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

**Import-Varianten**:

a) Gesamtes Modul importieren:

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

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

b) Destrukturierende Zuweisung:

```
const { readFile } = require("fs");
readFile();

const { readFile, ...fs } = require("fs");
readFile();
fs.writeFile();
```

### 3.4 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**:

```
import * as math from "./math.js";

console.log(math.add(5, 2)); // Ausgabe: 7
console.log(math.subtract(5, 2)); // Ausgabe: 3
```

**Import mit destrukturierenden Zuweisung**:

```
import { add as addition, subtract } from "./math.js";

console.log(addition(5, 2)); // Ausgabe: 7
console.log(subtract(5, 2)); // Ausgabe: 3
```

### 3.5 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));
```

**async/await**:

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

async function readFileExample() {
  try {
    const data = await fs.readFile("file.txt");
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}

readFileExample();
```

### 3.6 Statischer Webserver

Mit Support für ein paar wenige MIME-Types:

```
const http = require("http");
const fs = require("fs");
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/
             javascript" });
13     if (extname(request.url) == ".css")
14       response.writeHead(200, { status, "Content-Type": "text/css" });
15
16     response.write(data);
17     response.end();
18   });
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**:

```
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
```

```
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/:id", (req, res) => {
  const id = parseInt(req.params.id);

  const item = DATA.find((o) => o.id === id);

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

**Query-Parameter**:

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

```
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) => {
  const id = parseInt(req.query.id);

  const item = DATA.find((o) => o.id === id);

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

**HTTP-Body (URL-Encoded)**

> Nutze `express.urlencoded`

```
app.use(express.urlencoded({ extended: true }));

const DATA = [{ id: 1, name: "Apfel", color: "gelb,rot" }];

app.post("/fruits", (req, res) => {
  const { name, color } = req.body;

  if (DATA.find((o) => o.name === name)) {
    res.send("Duplicate name");
  } else {
    const id = Math.max(...DATA.map((o) => o.id)) + 1;
    const fruit = { id, name, color };
    DATA.push(fruit);
    res.send(fruit);
  }
});
```

**HTTP-Body (JSON)**

> Nutze `express.json`

```
app.use(express.json());

const DATA = [{ id: 1, name: "Apfel", color: "gelb,rot" }];

app.post("/fruits", (req, res) => {
  const { name, color } = req.body;

  if (DATA.find((o) => o.name === name)) {
    res.send("Duplicate name");
  } else {
    const id = Math.max(...DATA.map((o) => o.id)) + 1;
    const fruit = { id, name, color };
    DATA.push(fruit);
    res.send(fruit);
  }
```

```
});
```

### 4.7 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

app.get(/a/, function (req, res) {
  res.send("/a/");
}); // alles mit 'a' drin

app.get(/.*fly$/, function (req, res) {
  res.send("/.*fly$/");
});
```

### 4.8 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.

### 4.9 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]);
```

Wichtig: `next` nicht vergessen!

### 4.10 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");
  })
  .delete(function (req, res) {
    res.send("Delete the book");
  });
```

**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`:

```
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) {
  res.send("Birds home page");
});

router.get("/about", function (req, res) {
  res.send("About birds");
});

module.exports = router;
```

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

```
const express = require("express");
const app = express();
const birds = require("./birds");

app.use("/birds", birds);

app.listen(3000);

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**:

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

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

- 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.

```
app.get("/", async (req, res, next) => {
  try {
    throw new Error("Something went wrong");
  } catch (err) {
    next(err);
  }
});

app.use((err, req, res, next) => {
  res.status(500);
  res.json({ message: "InternalServerError" });
  console.error(err);
});
```

### 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:

```
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 ...
});
```

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

```
<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:

```
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 });
});
```

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

```
<html>
  <body>
    <h1>User Information</h1>
    <table>
      <tr>
        <td>Name:</td>
        <td><%= user.name %></td>
      </tr>
      <tr>
        <td>Email:</td>
        <td><%= user.email %></td>
      </tr>
      <tr>
        <td>Phone:</td>
        <td><%= user.phone %></td>
      </tr>
    </table>
```

```
  </body>
</html>
```

### 5.3 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
});

app.get("/fruits/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const fruit = DATA.find((o) => o.id === id);
  res.render("fruit", fruit); // fruit.ejs Template
});
app.listen(3000);

console.log("EJS server running on localhost:3000");
```

**Template**:

```
<html>
  <body>
    <table>
      <tr>
        <th>Name</th>
        <th>Farbe</th>
      </tr>
      <% fruits.forEach( o => { %>
      <tr>
        <td><%= o.name %></td>
        <td><%= o.color %></td>
      </tr>
      <% }) %>
    </table>
  </body>
</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:

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

app.use(cookieParser());

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

response
  .cookie("userID", "xyz12345")
  .cookie("verein", "VfB Stuttgart", { maxAge: 90000 }); // Mehrere
     Cookies, der zweite mit 90000 milli secs Lebensdauer
```

So können **Cookies ausgelesen werden**:

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

app.use(cookieParser());

const cookies = request.cookies;

let userID = cookies.userID;
let verein = cookies.verein;
```

### 5.5 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 Cookies
    resave: false, // Nur speichern nach Änderung
    saveUninitialized: true, // Anfangs immer speichern
    cookie: { maxAge: 5000 }, // Ablaufzeit in Millisekunden
  })
```

```
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**:

```
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 ...");
});
```

## 6.4 Queries in MongoDB

**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**:

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

**Exportieren von Dokumenten**:

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

**Einfügen eines Dokument**:

```
app.post("/example-create/fruits", async (req, res) => {
  const { name, color } = req.body;
  const fruit = { name, color };

  await db.collection("fruits").insertOne(fruit);

  res.send(`${name} inserted ...`);
});
```

**Auslesen eines Dokuments**:

```
app.get("/example-find-one/fruits/:name", async (req, res) => {
  const { name } = req.params;

  const fruit = await db.collection("fruits").findOne({ name });

  if (fruit) {
    res.send(fruit);
  } else {
    res.status(400).send("not found ...");
  }
});
```

**Einfügen mehrerer Dokumente**:

```
app.post("/example-insert-many/fruits", async (req, res) => {
  const fruits = [
    { name: "Apfel", color: "gelb,rot" },
    { name: "Birne", color: "gelb,grün" },
    { name: "Kiwi", color: "grün" },
    { name: "Banane", color: "gelb" },
    { name: "Pfirsich", color: "gelb,rot" },
  ];

  await db.collection("fruits").insertMany(fruits);

  res.send("Fruits inserted");
});
```

**Auslesen aller Dokumente**:

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

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

**Löschen eines Dokuments**:

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

```
  const { name } = req.params;

  const result = await db.collection("fruits").deleteOne({ name });

  if (result.deletedCount > 0) {
    res.send(`${name} deleted ...`);
  } else {
    res.status(400).send("fruit not found, nothing to delete ...");
  }
});
```

**Löschen mehrerer Dokumente**:

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

**Aktualisierung von Dokumenten**:

```
app.patch("/example-update-cuisine/:name", async (req, res) => {
  const { name } = req.params;
  const { cuisine } = req.body;

  const result = await db.collection("restaurants").updateOne(
    { name },
    {
      $set: { cuisine },
      $currentDate: { lastModified: true }, // Änderungsdatum
    }
  );

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

**Hinzufügen von Arrayelementen in Dokumenten**:

```
app.post("/example-push-grade-score/restaurants/:name", async (req, res
   ) => {
  const { name } = req.params;
  const { grade, score } = req.body;
  const newGrade = { date: new Date(), grade, score };

  const result = await db.collection("restaurants").updateOne(
    { name },
    (update = {
      $push: { grades: newGrade },
      $currentDate: { lastModified: true },
    })
  );

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

**Löschen von Arrayelementen in Dokumenten**:

```
app.post("/example-pop-grade-score/restaurants/:name", async (req, res)
    => {
  const { name } = req.params;
  const query = { name };

  const result = await db.collection("restaurants").updateOne(query, {
    $pop: { grades: 1 },
    $currentDate: { lastModified: true },
  });

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

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

**Tiefe Queries**:

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

  const restaurants = await db
    .collection("restaurants")
    .find({ "address.zipcode": zip, cuisine })
    .toArray();

  res.send(restaurants.map((o) => ({ name: o.name, zip: o.address.
     zipcode })));
});
```

**BSON für Vergleichsoperatoren**:

```
app.get("/example-zip-range/restaurants", async (req, res) => {
  const { zipMin, zipMax, cuisine } = req.query;

  const restaurants = await db
    .collection("restaurants")
    .find({
      cuisine,
      "address.zipcode": { $gte: zipMin, $lt: zipMax }, // es gibt auch
          $eq, $in, $neq, $nin ...
    })
    .toArray();

  res.send(restaurants.map((o) => ({ name: o.name, zip: o.address.
     zipcode })));
});
```

**BSON für Oder-Verknüpfung**:

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

  const restaurants = await db
    .collection("restaurants")
    .find(
      { $or: [{ "address.zipcode": zip }, { cuisine }] } // es gibt
          auch noch $and, $not und $nor
    )
    .toArray();

  res.send(
    restaurants.map((o) => ({
      name: o.name,
      cuisine: o.cuisine,
      zip: o.address.zipcode,
    }))
  );
});
```

## 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/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 },
      { projection: { name: 1, address: 1, _id: 0 } } // 1 bedeuted
          Einschluss eines Attributs, 0 den Ausschluss
    )
    .toArray();

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

**Sortieren**:

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

  const restaurants = await db
    .collection("restaurants")
    .find({ borough, cuisine }, { projection: { name: 1, address: 1,
       _id: 0 } })
    .sort({ name: 1 }) // Mit 1 wird aufsteigend sortiert, mit 0
       absteigend.
    .toArray();
  res.send(restaurants);
});
```

**Aggregation**:

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

  const restaurants = await db
    .collection("restaurants")
    .aggregate([
      {
        // Wie `WHERE` in SQL
        $match: { borough, cuisine },
      },
      {
        // Wie `SELECT` in SQL
        $project: {
          name: "$name", // auch name: 1 möglich
          avg_score: { $avg: "$grades.score" },
        }, // auch $min, $max, $sum
      },
      { $sort: { name: 1 } },
    ])
    .toArray();

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

**Gruppierung**:

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

  const restaurants = await db
    .collection("restaurants")
    .aggregate([
      {
        $match: { borough, cuisine },
      },
```

```
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 `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**

## 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**:

```
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` maps to a collection `fruits`
```

**Schema Types aus ES6**:

```
const schemaExample = new mongoose.Schema({
  bool: Boolean,
  updated: Date,
  age: Number,
  array: [],
  arrayofString: [String],
  arrayofArrays: [[]],
  arrayofArrayOfNumbers: [[Number]],
  map: Map,
  mapOfString: { type: Map, of: String },
});
```

**Weitere Schema Types aus MongoDB**:

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

**Indizes in Schemas**:

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

**Options-Objekt**:

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

```
  },
  {
    // Das Options-Objekt
    timestamps: true, // Automatisch `createdAt` und `modifiedAt`
        verwalten
  }
);
```

**Verschachtelte Schemata mit Kopien**:

```
const ratingSchema = new mongoose.Schema({
  grade: {
    type: Number,
    min: 1,
    max: 6,
  },
  comment: String,
  date: Date,
});

const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: String, required: true },
  ratings: [ratingSchema],
});

const Rating = mongoose.model("Rating", ratingSchema);
const Product = mongoose.model("Product", productSchema);
```

**Verschachtelte Schemata mit Referenzen**:

```
const productSchema = new mongoose.Schema({
  name: String,
  category: {
    // 1:1-Beziehung
    type: mongoose.ObjectId,
    ref: "Category",
  },
});

const Product = mongoose.model("Product", productSchema);

const categorySchema = new mongoose.Schema({
  name: String,
  products: [
    // 1:n-Beziehung
    {
      type: mongoose.ObjectId,
      ref: "Product",
    },
  ],
```

```
});

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

```
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) => {
  const { name, color } = req.body;
  const doc = await Fruit.findOne({ name });
  if (doc) {
    res.status(400).send("fruit found, delete first ...");

    return;
  }

  const fruit = new Fruit({ name, color });

  const imgPath = path.join(IMAGE_PATH, `${name}.png`);
  try {
    fruit.img.data = await fs.readFile(imgPath);
    fruit.img.contentType = "image/png";
  } catch (err) {
    console.log(`No image for ${name} found.`);
  }

  await fruit.save();

  res.send(`${name} inserted ...`);
});
```

**Umwandlung von MongoDB-Queries in Mongoose-Queries**:

```
app.get("/example-group-by-color/fruits/:color", async (req, res) => {
  const { color } = req.params;
  const result = await Fruit.aggregate([
    {
      $match: { color: { $regex: color } },
    },
    {
      $group: {
        _id: "$color",
        count: { $sum: 1 },
      },
    },
    { $sort: { _id: 1 } },
  ]);
  res.send(result);
});
```

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

### 6.12 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 () {
      return this.bacon > 3;
    },
  },
});
```

**Eigene Validatoren**:

```
function myValidator(val) {
  return val === "something";
}

```

```
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 }
       });
```