Express Grundlagen

Lernziele

- Grundlegende Vertrautheit mit dem Schreiben einer Serveranwendung
- Node vs. Express
- Warum Express verwenden?
- Wie man einen einfachen Server mit Express erstellt

Hintergrund

- In den kommenden Tagen werden wir Next.js lernen, das es uns ermöglicht, serverseitige Logik auszuführen, aber nicht im traditionellen reinen Server-Paradigma. Es ist wichtig, ein wenig Vertrautheit damit zu entwickeln, wie man seinen eigenen Server erstellt.

Rückblick: Wichtige Begriffe

Was ist ein Server?

- Ein Server ist eine Anwendung, die auf eingehende Anfragen hört.
- Er untersucht jede Anfrage und reagiert entsprechend den Regeln, mit denen er programmiert wurde.
- Wir verwenden ständig Server, aber diese werden von anderen Personen erstellt.
- Heute bauen wir unseren eigenen.

Was ist Node JS?

- Denke daran, dass Node JS eine JavaScript-Engine ist, die auf deinem Computer läuft.
- Es ist ziemlich ähnlich dem JavaScript, das wir für den Browser verwenden.
- Es enthält jedoch viele Module und Logik für die Handhabung von Dingen, die für Server relevant sind.
- Dazu gehören das Ausführen von Prozessen, das Abhören von eingehenden Anfragen, das Senden von Antworten, das Lesen und Schreiben von Dateien und vieles mehr.
- Diese Fähigkeiten sind im JavaScript, das wir im Browser verwenden, nicht vorhanden.
- Heute verwenden wir die berühmte und beliebte Express-Bibliothek.
- Sie gibt uns einige bequeme und hilfreiche Möglichkeiten, eine Server-Webanwendung zu erstellen.
- Aber es ist wichtig zu wissen, dass all diese Funktionalität auch mit reinem, "vanilla" Node JS gemacht werden kann, obwohl es mehr Codeaufwand erfordert.

Erstellen einer Express-App

- Du kannst eine Express-App erstellen, indem du das `express`-Paket in eine Node-App installierst.
- Erstelle eine App von Grund auf mit `npm init`.
- **NB**: Stelle sicher, dass du `"type": "module"` hinzufügst, da sonst `import` und `export` nicht funktionieren!
- Füge Express als Abhängigkeit hinzu, indem du `npm install express` ausführst.
- Die `package.json` sollte jetzt widerspiegeln, dass Express installiert wurde.

- Wahrscheinlich möchtest du `eslint` und `prettier` hinzufügen, um dir beim Linting und Formatieren deines Codes zu helfen.
- Füge die folgenden Dateien hinzu:

.prettierrc.json

```
{
  "printWidth": 80,
  "tabWidth": 2,
  "useTabs": false,
  "semi": true,
  "singleQuote": false,
  "quoteProps": "as-needed",
  "jsxSingleQuote": false,
  "trailingComma": "es5",
  "bracketSpacing": true,
  "bracketSameLine": false,
  "jsxBracketSameLine": false,
  "arrowParens": "always",
  "requirePragma": false,
  "insertPragma": false,
  "proseWrap": "preserve",
  "htmlWhitespaceSensitivity": "css",
  "vueIndentScriptAndStyle": false,
  "endOfLine": "lf",
  "embeddedLanguageFormatting": "auto",
  "singleAttributePerLine": false
}
```

.eslintrc.json

```
{
  "extends": ["eslint:recommended"],
  "root": true
}
```

A. Grundstruktur eines einfachen Express-Servers

```
// index.js
import express from "express";

// Liest den PORT-Wert aus der Umgebungsvariable `PORT`.
// Falls nicht gefunden, wird der Standardwert 3000 verwendet.
const PORT = process.env.PORT || 3000;
const app = express();

app.listen(PORT, () => {
```

```
  console.log(`Listening on port ${PORT}`);
});
```

Was macht das?

- Wir erstellen eine neue Express-App/Instanz mit `express()`.
- Wir weisen sie an, auf Anfragen am Port 3000 des Computers zu hören.
- Jede Ausgabe wird im Terminal protokolliert, wo wir die App gestartet haben!

Die App ausführen

```
node index.js # oder einfach `node .`
```

Verbesserung: Verwende nodemon

- Es wird **mühsam**, unsere App nach jeder Codeänderung neu zu starten.
- Stattdessen können wir `nodemon` verwenden, eine App, die unseren Code auf Änderungen **überwacht**.
- Wenn sie eine Änderung erkennt, wird die Anwendung neu gestartet.
- Um es in deiner App (als Entwicklungsabhängigkeit) zu installieren: `npm install --save-dev nodemon`.
- Füge ein **Script** in deine `package.json` ein, das `dev` genannt wird: `nodemon .`.
- Jetzt kannst du deine App mit `npm run dev` starten.

B. Beispielroute: `GET /` - Root-Route, die Text zurückgibt

- Dies weist den Server an, auf eine bestimmte Anfrage an die **Root**-Route zu reagieren. Es sendet eine reine Textantwort -- das ist **kein** HTML!

```
app.get("/", (request, response) => {
  // Wird im Terminal auf der SERVER-Seite protokolliert
  console.log("Jemand will die Root-Route!");
  // Wird an den Client gesendet (sichtbar im Browserfenster)
  response.send("Willkommen auf unserer Seite! 😎 ");
});
```

C. Beispielroute: `GET /contact` - HTML zurückgeben

- Baue eine HTML-Seite für das Kontaktformular.
- Denke an die wichtigen Regeln, die wir früher über Formulare in HTML besprochen haben!

```
<!-- contact-form.html -->
<!-- ... -->
<h1>Kontaktieren Sie uns</h1>
<form method="post">
```

```
  <div>
    <label for="message">Nachricht: </label>
    <input type="text" name="message" id="message" />
  </div>
  <div>
    <button type="submit">Senden</button>
  </div>
</form>
<!-- ... -->
```

```
// index.js

// ...
app.get("/contact", function (req, res) {
  console.log("contact.html wird geladen...");
  const dirName = path.dirname(new URL(import.meta.url).pathname);
  res.sendFile(path.join(dirName, "/contact-form.html"));
});
```

D. Beispielroute: `GET /staff` - JSON zurückgeben

- Wir können auch mit Daten in anderen Formaten antworten, z.B. JSON.
- Zur besseren Verständlichkeit können wir `.json` im Pfad der Route verwenden. Aber das ist eigentlich nicht nötig! Es hilft nur unseren Benutzern, besser zu verstehen, wie unser Server verwendet werden kann.

```
// index.js

// ...
const staff = ["Sally", "Bob", "Mike", "Rachel", "Andy", "Greg"];

app.get("/staff", (req, res) => {
  res.json({
    people: staff,
  });
});
```

- Teste es im Browser.
- Du solltest die JSON-Daten zurückbekommen. Beachte, dass sie auch mit dem richtigen Datentyp-Header gesendet werden!

E. Beispielroute: `GET /staff/:name` - mit einer Variable/Parameter

- Anstatt viele Routen für genaue Pfade zu erstellen, könnten wir eine Route mit einem **variablen Teil** erstellen: ein **Platzhalter** für jeden Wert!
- Dies erspart uns die Mühe, viele spezifische Routen zu pflegen.

```
app.get("/staff/:person", (req, res) => {
  // Hole den tatsächlichen Wert, der für den Parameter gesendet wurde,
aus `req.params`.
  const name = req.params.person;
  res.json({
    name: name,
    description: `${name} ist ein geschätzter Mitarbeiter in unserem
Unternehmen!`,
  });
});
```

- Wir sollten eine **Datenvalidierung** durchführen, um zu überprüfen, ob die erhaltenen Eingaben tatsächlich gültig sind.
- Dies ist ein größeres Thema, aber ein Anfang wäre, zuerst im "Datenbank" (Array der Namen) nach dem Namen zu **suchen**. Wenn keiner gefunden wird, könnten wir immer mit einem `404` antworten:

```
// index.js

// ...
// Antwortet mit einem Statuscode von 404 und einem gegebenen JSON-Body.
res.status(404).json({
  error: "Mitarbeiter nicht gefunden",
});
```

F. Beispielroute: `POST /contact` - Kontaktformular empfangen, Weiterleitung zu `/`

- Um das erwartete JSON zu **parsen**, das wir als Body dieser Anfrage erhalten, verwenden wir einige **Express Middleware**.
- In Express ist Middleware ein Stück Logik, das wir in die Express-Anfrageverarbeitungskette "einstecken" können.
- Typischerweise werden Middleware-Funktionen die Anfrage modifizieren oder überprüfen, und sie könnten auch die Anfrageverarbeitung frühzeitig abbrechen und eine Antwort senden.
- (Hier arbeiten wir mit einem JSON-Body. Es gibt andere Möglichkeiten, mit einem reinen HTML-Formular-Body zu arbeiten, aber das behandeln wir hier nicht.)

```
// index.js

app.use(express.json());

// ...

app.post("/contact", function (req, res) {
  console.log(
    "Wir haben eine Nachricht erhalten! Jemand schrieb:",
    req.body.message
```

```
  );
  // Wir möchten hier keine spezifische Seite anzeigen;
  // Stattdessen leiten wir auf die Startseite um.
  res.redirect("/");
});
```

- Teste dies **mit einem REST-Client**. Stelle sicher, dass du JSON als Anfrage-Body sendest!
- Normalerweise würden wir dies mit etwas JavaScript auf der Client-Seite erreichen:
  - Das Submit-Event im Formular behandeln
  - Das Standardverhalten des Events verhindern
  - Ein JSON-Objekt mit den Daten vorbereiten, die der Benutzer im Formular eingegeben hat
  - Diese JSON-Daten in einer neuen POST-Anfrage an den Server senden.
- Du solltest die Nachricht im Terminal sehen.

Zusammenfassung

- Wir haben uns Express als Bibliothek angesehen, um Server-Apps in NodeJS zu erstellen.
- Wir haben ein paar Routen erstellt.
- Wir haben experimentiert, wie man Antworten an den Client sendet.
- Wir haben Möglichkeiten gefunden, Text, Dateien und JSON-Daten zu senden.
- Wir haben Formular-Daten behandelt, die vom Client übermittelt wurden.

Resources

- nodemon
- Express home-page
- Learn Express and Node at Mozilla Developers Network (MDN)
- Routing in Express
- Using Middleware in Express