Community

TUTORIAL

Erstellen eines Webservers in Node.js mit dem HTTP-Modul

Node.js Development JavaScript Programming Project

By Stack Abuse
Posted May 28, 2020 1.5k

Deutsch

Der Autor hat den COVID-19 Relief Fund dazu ausgewählt, eine Spende im Rahmen des Programms Write for DOnations zu erhalten.

Einführung

Wenn Sie eine Webseite in Ihrem Browser anzeigen, stellen Sie eine Anfrage über das Internet an einen anderen Computer, der Ihnen dann die Webseite als Antwort bereitstellt. Dieser Computer, mit dem Sie über das Internet kommunizieren, ist ein *Webserver*. Ein Webserver empfängt HTTP-Anfragen von einem Client (wie Ihrem Browser) und stellt eine HTTP-Antwort bereit, z. B. eine HTML-Seite oder JSON aus einer API.

Am Zurückgeben einer Webseite durch einen Server sind verschiedene Softwareanwendungen beteiligt. Diese Software fällt im Allgemeinen in zwei Kategorien: Frontend und Backend. *Frontend-Code* befasst sich damit, wie Inhalte dargestellt werden, z. B. die Farbe einer Navigationsleiste und der Textstil. *Backend-Code* befasst sich damit, wie Daten ausgetauscht, verarbeitet und gespeichert w SCROLL TO TOP Netzwerkanfragen von Ihrem

Browser behandelt oder mit der Datenbank kommuniziert, wird in erster Linie mit Backend-Code verwaltet.

Mit Node.js können Entwickler JavaScript zum Schreiben von Backend-Code nutzen, auch wenn die Umgebung im Browser traditionell verwendet wurde, um Frontend-Code zu schreiben. Wenn Frontend und Backend so nah beieinander sind, verringert sich der Aufwand bei der Einrichtung eines Webservers. Das ist ein Grund dafür, warum Node.js eine beliebte Wahl zum Schreiben von Backend-Code ist.

In diesem Tutorial lernen Sie, wie Sie Webserver mit dem in Node.js enthaltenen `http`-Modul einrichten können. Sie werden Webserver erstellen, die JSON-Daten, CSV-Dateien und HTML-Webseiten zurückgeben können.

Voraussetzungen

- Stellen Sie sicher, dass Node.js auf Ihrem Entwicklungscomputer installiert ist. Dieses Tutorial verwendet die Node.js-Version 10.19.0. Um dies unter MacOS oder Ubuntu 18.04 zu installieren, folgen Sie den Schritten in Installation von Node.js und Erstellen einer lokalen Entwicklungsumgebung auf MacOS oder dem Abschnitt **Installation unter Verwendung eines PPA** von Installation von Node.js auf Ubuntu 18.04.
- Die Node.js-Plattform unterstützt eine standardmäßige Erstellung von Webservern. Bevor Sie beginnen, sollten Sie sich mit den Grundlagen von Node.js vertraut machen. Sie können damit starten, indem Sie unseren Leitfaden zu Schreiben und Ausführen Ihres ersten Programms in Node.js lesen.
- Außerdem verwenden wir asynchrone Programmierung für einen unserer Abschnitte. Wenn Sie mit der asynchronen Programmierung in Node.js oder dem `fs`-Modul für die Interaktion mit dem Dateisystem noch nicht vertraut sind,

können Sie in unserem Artikel mehr erfahren über das Schreiben von asynchronem Code in Node.js.

Schritt 1 — Einrichten eines grundlegenden HTTP-Servers

Beginnen wir zunächst mit der Einrichtung eines Servers, der dem Benutzer Klartext zurückgibt. Damit werden die zur Einrichtung eines Servers erforderlichen Schlüsselkonzepte behandelt, was als Grundlage für die Rückgabe komplexer Datenformate wie JSON dient.

Zuerst müssen wir eine zugängliche Codierungsumgebung einrichten, um unsere Übungen sowie andere Aufgaben in dem Artikel ausführen zu können. Erstellen Sie im Terminal einen Ordner namens `first-servers`:

```
$ mkdir first-servers
```

Gehen Sie dann in diesen Ordner hinein:

```
$ cd first-servers
```

Erstellen Sie nun die Datei, die den Code beinhalten wird:

```
$ touch hello.js
```

Öffnen Sie die Datei in einem Texteditor. Wir verwenden `nano`, da es im Terminal verfügbar ist:

```
$ nano hello.js
```

Wir starten mit dem Laden des `http`-Moduls, das bei allen Node.js-Installationen Standard ist. Fügen Sie die folgende Zeile zu `hello.js` hinzu:

first-servers/hello.js

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

Das Modul `http` enthält die Funktion zur Erstellung des Servers, wie wir später sehen werden. Wenn Sie mehr über Module in Node.js erfahren möchten, lesen Sie unseren Artikel Erstellen eines Node.js-Moduls.

Unser nächster Schritt besteht darin, zwei Konstanten zu definieren, also den Host und den Port, an die unser Server gebunden ist:

first-servers/hello.js

```
...
const host = 'localhost';
const port = 8000;
```

Wie bereits erwähnt, akzeptieren Webserver Anfragen von Browsern und anderen Clients. Wir können mit einem Webserver interagieren, indem wir einen Domänennamen eingeben, der über einen DNS-Server in eine IP-Adresse übersetzt wird. Eine IP-Adresse ist eine eindeutige Sequenz von Zahlen, die einen Rechner in einem Netzwerk wie dem Internet identifiziert. Weitere Informationen zu Domänennamenkonzepten finden Sie in unserem Artikel Eine Einführung in DNS-Begriffe, -Komponenten und -Konzepte.

Der Wert `localhost` ist eine spezielle private Adresse, die Computer zum Verweisen auf sich selbst verwenden. Es handelt sich dabei typischerweise um das Äquivalent der internen IP-Adresse `127.0.0.1` und ist nur für den lokalen Computer verfügbar, nicht für die lokalen Netzwerke, mit denen wir verbunden sind, oder mit dem

Der Port ist eine Zahl, die Server als Endpunkt oder „Tür“ zu unserer IP-Adresse verwenden. In unserem Beispiel verwenden wir Port `8000` für unseren Webserver. Ports `8080` und `8000` dienen typischerweise als Standardports für die Entwicklung, und meist nutzen Entwickler sie anstelle anderer Ports für HTTP-Server.

Wenn wir unseren Server an diesen Host und Port binden, können wir unseren Server erreichen, indem wir in einem lokalen Browser `http://localhost:8000` aufrufen.

Lassen Sie uns eine spezielle Funktion hinzufügen, die wir in Node.js einen *Request Listener* nennen. Diese Funktion dient dazu, eine eingehende HTTP-Anfrage zu bearbeiten und eine HTTP-Antwort zurückzugeben. Diese Funktion muss zwei Argumente aufweisen: ein Anfrageobjekt und ein Antwortobjekt. Das Anfrageobjekt erfasst alle eingehenden Daten der HTTP-Anforderung. Das Antwortobjekt dient der Rückgabe von HTTP-Antworten an den Server.

Wir möchten, dass unser erster Server folgende Nachricht zurückgibt, wenn jemand darauf zugreift: „`My first server!`".

Fügen wir als Nächstes diese Funktion hinzu:

first-servers/hello.js

```
...

const requestListener = function (req, res) {
    res.writeHead(200);
    res.end("My first server!");
};
```

Die Funktion würde normalerweise benannt, was sie tut. Wenn wir zum Beispiel eine Request Listener-Funktion erstellen, um eine Liste von

Büchern zurückzugeben, würden wir sie wahrscheinlich `listBooks()` nennen. Da dies ein Beispielfall ist, verwenden wir den allgemeinen Namen `requestListener`.

Alle Request Listener-Funktionen in Node.js akzeptieren zwei Argumente: `req` und `res` (wir können sie anders nennen, wenn wir möchten). Die HTTP-Anfrage, die der Benutzer sendet, wird in einem Anfrageobjekt erfasst. Das entspricht dem ersten Argument `req`. Die HTTP-Antwort, die wir an den Benutzer zurückgeben, wird gebildet, indem wir mit dem Antwortobjekt im zweiten Argument `res` interagieren.

Die erste Zeile `res.writeHead(200);` legt den HTTP-Statuscode der Antwort fest. HTTP-Statuscodes geben an, wie gut eine HTTP-Anfrage vom Server bearbeitet wurde. In diesem Fall entspricht der Statuscode `200` „OK“. Wenn Sie mehr über die verschiedenen HTTP-Codes, die Ihre Webserver zurückgeben können, und ihre jeweilige Bedeutung erfahren möchten, ist unser Leitfaden Fehlerbehebung für gängige HTTP-Fehlercodes ein perfekter Ausgangspunkt.

Die nächste Zeile der Funktion `res.end("My fist server!") ;` schreibt die HTTP-Antwort zurück an den Client, der sie angefordert hat. Diese Funktion gibt alle Daten zurück, die der Server zurückgeben muss. In diesem Fall werden Textdaten zurückgegeben.

Abschließend können wir unseren Server erstellen und unseren Request Listener verwenden:

first-servers/hello.js

```
...

const server = http.createServer(requestListener);
server.listen(port, host, ()
```

```
    console.log(`Server is running on http://${host}:${port}`);
});
```

Speichern und schließen Sie `nano`, indem Sie `Strg+X` drücken.

In der ersten Zeile erstellen wir ein neues `Server`-Objekt über die Funktion `createServer()` des `http`-Moduls. Dieser Server akzeptiert HTTP-Anfragen und übergibt sie an unsere Funktion `requestListener()`.

Nachdem wir unseren Server erstellt haben, müssen wir ihn nun an eine Netzwerkadresse binden. Das tun wir mit der Methode `server.listen()`. Sie akzeptiert drei Argumente: `Port`, `Host` und eine Rückruffunktion, die ausgelöst wird, sobald der Server mit dem Lauschen beginnt.

Alle diese Argumente sind optional; es ist aber eine gute Idee, explizit zu bestätigen, welchen Port und Host ein Webserver verwenden soll. Wenn Sie Webserver in verschiedenen Umgebungen bereitstellen, müssen Sie den Port und den Host kennen, bei denen sie ausgeführt werden, um den Lastausgleich oder einen DNS-Alias einzurichten.

Die Rückruffunktion protokolliert eine Nachricht an unsere Konsole, damit wir wissen, wann der Server mit dem Lauschen nach Verbindungen begonnen hat.

> **Anmerkung:** Obwohl `requestListener()` das Objekt `req` nicht nutzt, muss es dennoch das erste Argument der Funktion sein.

Mit weniger als fünfzehn Codezeilen verfügen wir nun über einen Webserver. Lassen Sie uns das in der Praxis ansehen und durchgängig testen, indem wir folgendes Programm ausführen:


```
$ node hello.js
```

In der Konsole sehen wir diese Ausgabe:

```
Output
Server is running on http://localhost:8000
```

Beachten Sie, dass die Eingabeaufforderung verschwindet. Das liegt daran, dass ein Node.js-Server ein lang laufender Vorgang ist. Der Server wird nur dann beendet, wenn ein Fehler auftritt, der zum Absturz des Servers führt, oder wenn wir den Node.js-Prozess anhalten, der den Server ausführt.

In einem separaten Terminalfenster kommunizieren wir mit dem Server über cURL, ein CLI-Tool zur Übertragung von Daten an und aus einem Netzwerk. Geben Sie den Befehl ein, um eine HTTP-`GET`-Anfrage an unseren laufenden Server zu stellen:

```
$ curl http://localhost:8000
```

Wenn wir `ENTER` drücken, zeigt unser Terminal die folgende Ausgabe:

```
Output
My first server!
```

Wir haben nun einen Server eingerichtet und unsere erste Serverantwort erhalten.

Lassen Sie uns genau ansehen, was passiert ist, als wir unseren Server getestet haben. Mit cURL haben wir eine `GET`-Anfrage an den Server bei `http://localhost:8000` gesendet. Unser Node.js-Server hat nach Verbindungen von dieser Adr SCROLL TO TOP er Server hat diese Anfrage an die Funktion `requestListener()` übergeben. Die Funktion hat Textdaten

mit dem Statuscode `200` zurückgegeben. Der Server hat diese Antwort dann an cURL zurückgesendet, wodurch die Nachricht in unserem Terminal angezeigt wurde.

Bevor wir fortfahren, beenden wir unseren laufenden Server, indem wir `Strg+C` drücken. Dadurch wird die Ausführung unseres Servers unterbrochen und wir kehren zur Befehlszeilenaufforderung zurück.

Bei den wenigstens Websites, die wir besuchen, oder APIs, die wir verwenden, werden Serverantworten in Klartext dargestellt. HTML-Seiten und JSON-Daten sind gängige Antwortformate. Im nächsten Schritt erfahren wir, wie sich HTTP-Antworten in gängigen Datenformaten zurückgeben lassen, die im Web genutzt werden.

Schritt 2 — Zurückgeben verschiedener Arten von Inhalten

Die Antwort, die wir von einem Webserver zurückgeben, kann eine Vielzahl von Formaten aufweisen. JSON und HTML wurden bereits erwähnt; wir können aber auch andere Textformate wie XML und CSV zurückgeben. Schließlich können Webserver Nicht-Text-Daten wie PDFs, gezippte Dateien, Audio und Video zurückgeben.

In diesem Artikel lernen Sie, wie Sie neben dem gerade zurückgegebenen Klartext die folgenden Arten von Daten zurückgeben:

- JSON
- CSV
- HTML

Diese drei Datentypen sind al[illegible] beliebte Formate für die Bereitstellung von Inhalten im [illegible] serverseitige

Entwicklungssprachen und Tools bieten Unterstützung für die Rückgabe dieser verschiedenen Datentypen. Im Kontext von Node.js müssen wir zwei Dinge tun:

1. Den Header `Content-Type` in unseren HTTP-Antworten mit dem entsprechenden Wert festlegen.
2. Sicherstellen, dass `res.end()` die Daten im richtigen Format erhält.

Lassen Sie uns dies mit einigen Beispielen in der Praxis ansehen. Der Code, den wir in diesem Abschnitt schreiben werden, sowie spätere Codes haben viele Ähnlichkeiten mit dem zuvor geschriebenen Code. Die meisten Änderungen gibt es innerhalb der Funktion `requestListener()`. Lassen Sie uns Dateien mit diesem „Vorlagencode“ erstellen, um zukünftige Abschnitte leichter folgbar zu machen.

Erstellen Sie eine neue Datei namens `html.js`: Diese Datei wird später verwendet, um in einer HTTP-Antwort HTML-Text zurückzugeben. Wir platzieren den Vorlagencode hier und kopieren ihn auf die anderen Server, die verschiedene Arten zurückgeben.

Geben Sie im Terminal Folgendes ein:

```
$ touch html.js
```

Öffnen Sie diese Datei nun in einem Texteditor:

```
$ nano html.js
```

Kopieren wir den „Vorlagencode“. Geben Sie Folgendes in `nano` ein:

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

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

Speichern und schließen Sie `html.js` mit `Strg+X` und kehren Sie dann zum Terminal zurück.

Lassen Sie uns diese Datei nun in zwei neue Dateien kopieren. Die erste Datei wird dazu dienen, CSV-Daten in der HTTP-Antwort zurückzugeben:

```
$ cp html.js csv.js
```

Die zweite Datei wird eine JSON-Antwort im Server zurückgegeben:

```
$ cp html.js json.js
```

Die verbleibenden Dateien werden für spätere Übungen benötigt:

```
$ cp html.js htmlFile.js
$ cp html.js routes.js
```

Wir sind nun bereit, um unsere Übungen fortzusetzen. Lassen Sie uns mit der Rückgabe von JSON beginnen.

Bereitstellen von JSON SCROLL TO TOP

JavaScript Object Notation, allgemein als JSON bezeichnet, ist ein textbasiertes Format für den Datenaustausch. Wie der Name bereits andeutet, wird es von JavaScript-Objekten abgeleitet, ist aber sprachunabhängig und lässt sich von jeder Programmiersprache verwenden, die seine Syntax parsen kann.

JSON wird häufig von APIs verwendet, um Daten zu akzeptieren und zurückzugeben. Die hohe Popularität beruht auf einer geringeren Datenübertragungsgröße als bei älteren Datenaustauschstandards wie XML sowie auf dem vorhandenen Tooling, mit dem Programme JSON ohne übermäßigen Aufwand parsen können. Wenn Sie mehr über JSON erfahren möchten, können Sie unseren Leitfaden Arbeiten mit JSON in JavaScript lesen.

Öffnen Sie die Datei `json.js` mit `nano`:

```
$ nano json.js
```

Wir möchten eine JSON-Antwort zurückgeben. Lassen Sie uns die Funktion `requestListener()` so anpassen, dass der entsprechende Header aller JSON-Antworten zurückgegeben wird, indem wir die hervorgehobenen Zeilen folgendermaßen ändern:

first-servers/json.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
};
...
```

Die Methode `res.setHeader()` fügt einen HTTP-Header hinzu. HTTP-Header sind zusätzliche Informationen, die an eine Anfrage oder

Antwort angehängt werden können. Die Methode `res.setHeader()` benötigt zwei Argumente: den Namen des Headers und seinen Wert.

Der Header `Content-Type` dient zum Anzeigen des Formats der Daten (auch als Medientyp bekannt), die mit der Anfrage oder Antwort gesendet werden. In diesem Fall ist unser `Content-Type application/json`.

Lassen Sie uns nun JSON-Inhalt an den Benutzer zurückgeben. Ändern Sie `json.js`, damit die Datei folgendermaßen aussieht:

first-servers/json.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    res.writeHead(200);
    res.end(`{"message": "This is a JSON response"}`);
};
...
```

Wie zuvor sagen wir dem Benutzer, dass seine Anfrage erfolgreich war, indem wir einen Statuscode von `200` zurückgeben. Dieses Mal enthält unser Zeichfolgenargument im Aufruf `response.end()` gültiges JSON.

Speichern und schließen Sie `json.js`, indem Sie `Strg+X` drücken. Lassen Sie uns nun den Server mit dem Befehl `node` ausführen:

```
$ node json.js
```

In einem anderen Terminal stellen wir eine Verbindung zum Server her, indem wir cURL verwenden:

```
$ curl http://localhost:8000
```

Wenn wir `ENTER` drücken, sehen wir folgendes Ergebnis:

```
Output
{"message": "This is a JSON response"}
```

Wir haben nun erfolgreich eine JSON-Antwort zurückgegeben, genauso wie viele der beliebten APIs, mit denen wir Apps einrichten. Beenden Sie den laufenden Server unbedingt mit `Strg+C`, damit wir zur standardmäßigen Terminalaufforderung zurückkehren können. Lassen Sie uns als Nächstes ein weiteres beliebtes Format zum Zurückgeben von Daten ansehen: CSV.

Bereitstellen von CSV

Das Dateiformat *Comma Separated Values* (CSV) ist ein Textstandard, der häufig für die Bereitstellung von Tabellendaten verwendet wird. In den meisten Fällen wird jede Zeile durch ein Neue-Zeile-Symbol und jedes Element in der Zeile durch ein Komma getrennt.

Öffnen Sie in unserem Arbeitsbereich mit einem Texteditor die Datei `csv.js`:

```
$ nano csv.js
```

Lassen Sie uns die folgenden Zeilen in unsere Funktion `requestListener()` einfügen:

first-servers/csv.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
    res.setHeader("Content-D                    chment;filename=oceanpals.csv"
```

```
};
...
```

Dieses Mal zeigt unser `Content-Type` an, dass eine CSV-Datei zurückgegeben wird, da der Wert `text/csv` lautet. Der zweite Header, den wir hinzufügen, ist `Content-Disposition`. Dieser Header teilt dem Browser mit, wie die Daten angezeigt werden sollen, insbesondere im Browser oder als separate Datei.

Wenn wir CSV-Antworten zurückgeben, laden die meisten modernen Browser die Datei automatisch herunter, selbst wenn der Header `Content-Disposition` nicht gesetzt ist. Wenn wir aber eine CSV-Datei zurückgeben, sollten wir diesen Header weiterhin hinzufügen, da er uns ermöglicht, den Namen der CSV-Datei festzulegen. In diesem Fall signalisieren wir dem Browser, dass diese CSV-Datei ein Anhang ist und heruntergeladen werden soll. Dann teilen wir dem Browser mit, dass der Name der Datei `oceanpals.csv` lautet.

Lassen Sie uns die CSV-Daten in die HTTP-Antwort schreiben:

first-servers/csv.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
    res.setHeader("Content-Disposition", "attachment;filename=oceanpals.csv"
    res.writeHead(200);
    res.end(`id,name,email\n1,Sammy Shark,shark@ocean.com`);
};
...
```

Wie zuvor geben wir mit unserer Antwort einen `200` / `OK` -Status zurück. Dieses Mal verfügt unser Aufr SCROLL TO TOP ɔer eine Zeichenfolge, die

eine gültige CSV ist. Das Komma trennt den Wert in den einzelnen Spalten, während das Neue-Zeile-Zeichen (`\n`) die Zeilen trennt. Wir verfügen über zwei Zeilen, eine für den Tabellen-Header und eine für die Daten.

Wir testen diesen Server im Browser. Speichern Sie `csv.js` und schließen Sie den Editor mit `Strg+X`.

Führen Sie den Server mit dem Befehl Node.js aus:

```
$ node csv.js
```

In einem anderen Terminal stellen wir eine Verbindung zum Server her, indem wir cURL verwenden:

```
$ curl http://localhost:8000
```

Das wird in der Konsole angezeigt:

```
Output
id,name,email
1,Sammy Shark,shark@ocean.com
```

Wenn wir in unserem Browser `http://localhost:8000` aufrufen, wird eine CSV-Datei heruntergeladen. Ihr Dateiname wird `oceanpals.csv` lauten.

Beenden Sie den laufenden Server mit `Strg+C`, um zur standardmäßigen Terminalaufforderung zurückzukehren.

Mit der Rückgabe von JSON und CSV haben wir zwei Fälle behandelt, die bei APIs beliebt sind. Lassen Sie uns mit der Ansicht fortfahren, die Benutzer von Websites in einem Brows

Bereitstellen von HTML

HTML (HyperText Markup Language) ist das gängigste Format, das verwendet wird, wenn wir wollen, dass Benutzer über einen Webbrowser mit unserem Server interagieren. Es dient dazu, Webinhalte zu strukturieren. Webbrowser dienen zum Anzeigen von HTML-Inhalten sowie beliebiger Stile, die wir mit CSS hinzufügen, einer anderen Frontend-Webtechnologie, mit der wir das Aussehen unserer Websites ändern können.

Lassen Sie uns die Datei `html.js` mit unserem Texteditor erneut öffnen:

```
$ nano html.js
```

Ändern Sie die Funktion `requestListener()`, um den entsprechenden `Content-Type`-Header für eine HTML-Antwort zurückzugeben:

first-servers/html.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
};
...
```

Lassen Sie uns nun HTML-Inhalt an den Benutzer zurückgeben. Fügen Sie die hervorgehobenen Zeilen in `html.js` hinzu, damit die Datei folgendermaßen aussieht:

first-servers/html.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(`<html><body><h1>This is HTML</h1></body></html>`);
```

```
};
...
```

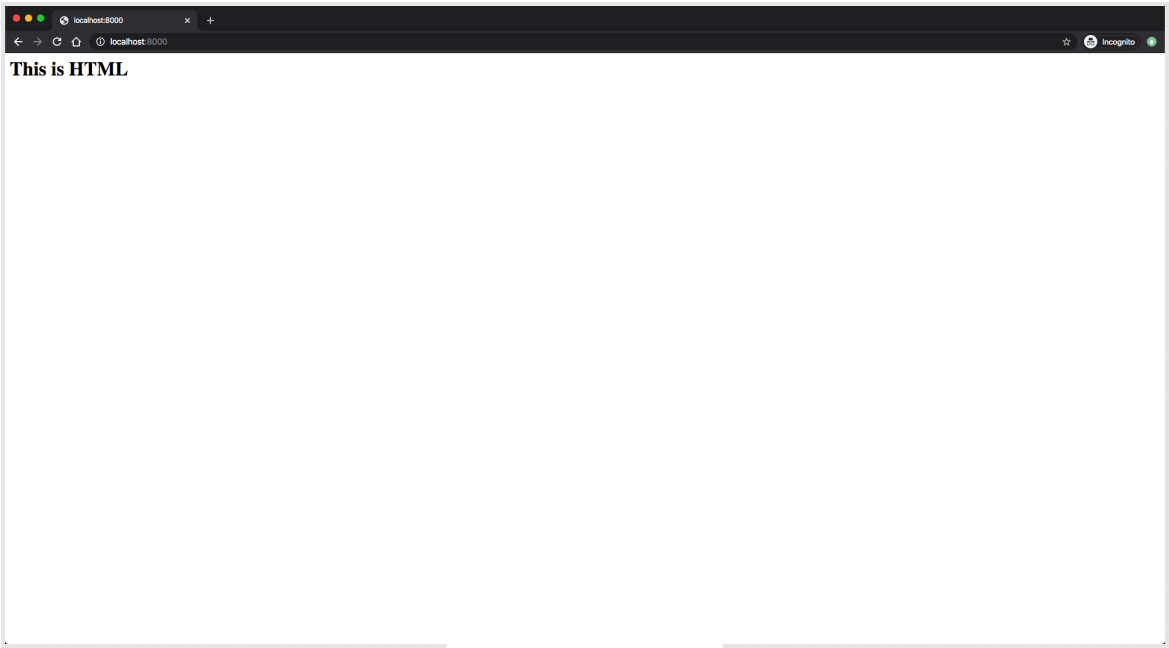
Wir fügen zuerst den HTTP-Statuscode ein. Dann rufen wir `response.end()` mit einem Zeichenfolgenargument auf, das gültiges HTML enthält. Wenn wir im Browser auf unseren Server zugreifen, sehen wir eine HTML-Seite mit einem Header-Tag, das den Inhalt `This is HTML` aufweist.

Speichern und schließen Sie Datei, indem Sie `Strg+X` drücken. Lassen Sie uns nun den Server mit dem Befehl `node` ausführen:

```
$ node html.js
```

Wir sehen `Server is running on http://localhost:8000`, sobald unser Programm gestartet wurde.

Rufen Sie nun im Browser `http://localhost:8000` auf. Unsere Seite wird so aussehen:



Beenden Sie den laufenden Server mit `Strg+C`, um zur standardmäßigen Terminalaufforderung zurückzukehren.

Es ist üblich, dass HTML in einer Datei geschrieben wird, getrennt vom serverseitigen Code (wie unsere Node.js-Programme). Lassen Sie uns als Nächstes sehen, wie wir HTML-Antworten aus Dateien zurückgeben können.

Schritt 3 — Bereitstellen einer HTML-Seite aus einer Datei

Wir können dem Benutzer HTML als Zeichenfolgen in Node.js bereitstellen, es ist jedoch empfehlenswert, HTML-Dateien zu laden und ihren Inhalt bereitzustellen. Wenn die HTML-Datei wächst, müssen wir so keine langen Zeichenfolgen in unserem Node.js-Code pflegen. Dadurch bleibt der Code kurz und können wir getrennt an den einzelnen Aspekten unserer Website arbeiten. Diese „Trennung von Aufgaben“ ist in vielen Webentwicklungs-Setups üblich; daher ist es gut zu wissen, wie HTML-Dateien zur Unterstützung in Node.js geladen werden.

Um HTML-Dateien bereitzustellen, laden wir die HTML-Datei mit dem `fs`-Modul und verwenden ihre Daten beim Schreiben unserer HTTP-Antwort.

Zuerst erstellen wir eine HTML-Datei, die der Webserver zurückgeben wird. Erstellen Sie eine neue HTML-Datei:

```
$ touch index.html
```

Öffnen Sie nun `index.html` in einem Texteditor:

```
$ nano index.html
```

Unsere Webseite wird minimal sein. Sie wird über einen orangen Hintergrund verfügen und in der Mitte einen Grußtext anzeigen. Fügen Sie diesen Code in die Datei ein:

first-servers/index.html

```
<!DOCTYPE html>

<head>
    <title>My Website</title>
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
        }

        html {
            width: 100%;
            height: 100%;
        }

        body {
            width: 100%;
            height: 100%;
            position: relative;
            background-color: rgb(236, 152, 42);
        }

        .center {
            width: 100%;
            height: 50%;
            margin: 0;
            position: absolute;
            top: 50%;
            left: 50%;
            transform: translate(-50%, -50%);
            color: white;
            font-family: "Trebuchet MS", Helvetica, sans-serif;
            text-align: cente
        }
```

```
        h1 {
            font-size: 144px;
        }

        p {
            font-size: 64px;
        }
    </style>
</head>

<body>
    <div class="center">
        <h1>Hello Again!</h1>
        <p>This is served from a file</p>
    </div>
</body>

</html>
```

Diese einzelne Webseite zeigt zwei Textzeilen an: `Hello Again!` und `This is served from a file`. Die Zeilen befinden sich in der Mitte der Seite übereinander. Die erste Zeile des Texts wird als Überschrift angezeigt. Das bedeutet, dass sie größer ist. Die zweite Zeile des Texts wird etwas kleiner erscheinen. Der gesamte Text wird weiß dargestellt, während die Webseite einen orangen Hintergrund hat.

Es ist zwar nicht Inhalt dieses Artikel oder dieser Reihe, Sie können bei Bedarf aber mehr über HTML, CSS und andere Frontend-Webtechnologien erfahren, indem Sie einen Blick in den Leitfaden Erste Schritte mit dem Web von Mozilla werfen.

Das ist alles, was wir für HTML benötigen. Speichern und schließen Sie die Datei also mit `Strg+X`. Wir können nun mit dem Servercode fortfahren.

Für diese Übung werden wir mit `htmlFile.js` arbeiten. Öffnen Sie die Datei mit dem Texteditor:

```
$ nano htmlFile.js
```

Da wir eine Datei lesen müssen, beginnen wir, indem wir das `fs`-Modul importieren:

first-servers/htmlFile.js

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

Dieses Modul enthält eine `readFile()`-Funktion, die wir zum Laden der HTML-Datei verwenden werden. Wir importieren die Zusagevariante anhand aktueller Best Practices für JavaScript. Wir verwenden Zusagen, da sie syntaktisch prägnanter sind als Rückrufe. Diese müssten wir verwenden, wenn wir `fs` lediglich `require('fs')` zuordnen. Weitere Informationen über Best Practices für asynchrone Programmierung finden Sie in unserem Leitfaden Schreiben von asynchronem Code in Node.js.

Wir möchten, dass unsere HTML-Datei gelesen wird, wenn ein Benutzer eine Anfrage an unser System stellt. Lassen Sie uns mit dem Ändern von `requestListener()` zum Lesen der Datei beginnen:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
};
...
```

Wir verwenden die Methode `fs.readFile()`, um die Datei zu laden. Ihr Argument lautet `__dirname + "/index.html"`. Der spezielle Variable `__dirname` weist den absoluten Pfad zu unserem Node.js-Code

ausgeführt wird. Dann hängen wir `/index.html` an, damit wir die zuvor erstellte HTML-Datei laden können.

Lassen Sie uns nun die HTML-Seite zurückgeben, sobald sie geladen ist:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
};
...
```

Wenn das Versprechen `fs.readFile()` erfolgreich aufgelöst wird, werden deren Daten zurückgegeben. Wir verwenden die Methode `then()`, um diesen Fall zu handhaben. Der Parameter `contents` enthält die Daten der HTML-Datei.

Wir setzen zunächst den Header `Content-Type` auf `text/html`, um dem Client zu sagen, dass wir HTML-Daten zurückgeben. Wir schreiben dann den Statuscode, um anzuzeigen, dass die Anfrage erfolgreich war. Abschließend senden wir dem Client die geladene HTML-Seite mit den Daten in der Variablen `contents`.

Die Methode `fs.readFile()` kann manchmal fehlschlagen. Daher sollten wir den Fall handhaben, wenn wir einen Fehler erhalten. Fügen Sie Folgendes in die Funktion `requestListener()` ein:

fi SCROLL TO TOP s

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
        .catch(err => {
            res.writeHead(500);
            res.end(err);
            return;
        });
};
...
```

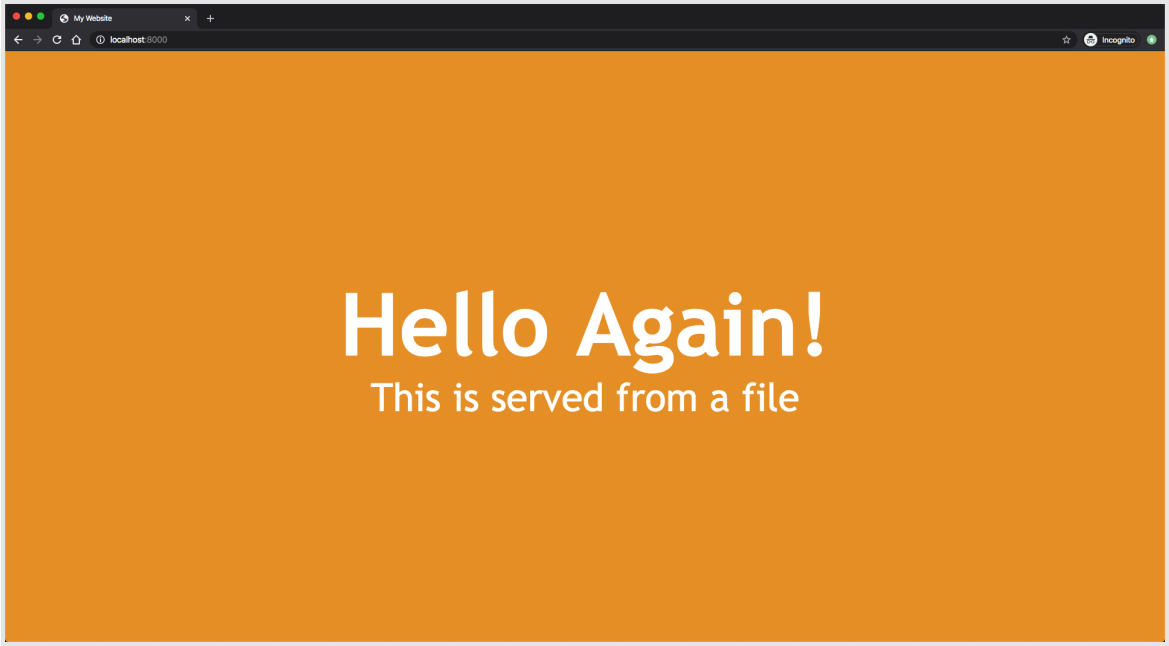
Speichern Sie die Datei und schließen Sie `nano` mit `Strg+X`.

Wenn ein Versprechen auf einen Fehler trifft, wird es verworfen. Wir behandeln diesen Fall mit der Methode `catch()`. Sie akzeptiert den Fehler, den `fs.readFile()` zurückgibt, legt den Statuscode auf `500` fest, was bedeutet, dass ein interner Fehler aufgetreten ist, und gibt den Fehler an den Benutzer zurück.

Führen Sie unseren Server mit dem Befehl `node` aus:

```
$ node htmlFile.js
```

Rufen Sie im Webbrowser `http://localhost:8000` auf. Sie sehen diese Seite:



Sie haben nun eine HTML-Seite vom Server an den Benutzer zurückgegeben. Sie können den laufenden Server mit `Strg+C` beenden. Wenn Sie dies tun, wird Ihnen die Eingabeaufforderung des Terminals angezeigt.

Wenn Sie in der Produktion Code wie hier verfassen, wollen Sie vielleicht nicht jedesmal eine HTML-Seite laden, wenn Sie eine HTTP-Anfrage erhalten. Zwar ist diese HTML-Seite nur etwa 800 Byte groß, doch können komplexere Websites Größen im Megabytebereich aufweisen. Große Dateien können eine Weile zum Laden benötigen. Wenn Ihre Website eine Menge Datenverkehr erwartet, kann es am besten sein, HTML-Dateien beim Starten zu laden und ihren Inhalt zu speichern. Nachdem sie geladen wurden, können Sie den Server einrichten und ihn an einer Adresse auf Anfragen lauschen lassen.

Um diese Methode zu demonstrieren, sehen wir uns an, wie wir unseren Server so anpassen können, dass er effizienter und besser skalierbar wird.

Effizientes Bereitstellen \ SCROLL TO TOP

Anstatt die HTML für jede Anfrage zu laden, laden wir sie in diesem Schritt nur einmal am Anfang. Die Anfrage gibt die beim Starten geladenen Daten zurück.

Öffnen Sie im Terminal mit einem Texteditor erneut das Skript Node.js:

```
$ nano htmlFile.js
```

Lassen Sie uns zunächst eine neue Variable hinzufügen, bevor wir die Funktion `requestListener()` erstellen:

first-servers/htmlFile.js

```
...
let indexFile;

const requestListener = function (req, res) {
...
```

Wenn wir dieses Programm ausführen, wird diese Variable den Inhalt der HTML-Datei enthalten.

Lassen Sie uns nun die Funktion `requestListener()` anpassen. Anstatt die Datei zu laden, gibt sie nun den Inhalt von `indexFile` zurück:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(indexFile);
};
...
```

Als Nächstes verschieben wir die Logik zum Dateilesen von der Funktion `requestListener()` in den Start unseres Servers. Nehmen Sie beim Erstellen des Servers die folgenden Änderungen vor:

first-servers/htmlFile.js

```
...

const server = http.createServer(requestListener);

fs.readFile(__dirname + "/index.html")
    .then(contents => {
        indexFile = contents;
        server.listen(port, host, () => {
            console.log(`Server is running on http://${host}:${port}`);
        });
    })
    .catch(err => {
        console.error(`Could not read index.html file: ${err}`);
        process.exit(1);
    });
```

Speichern Sie die Datei und schließen Sie `nano` mit `Strg+X`.

Der Code, der die Datei liest, ähnelt dem, was wir bei unserem ersten Versuch geschrieben haben. Nachdem wir die Datei erfolgreich gelesen haben, speichern wir den Inhalt jedoch in unserer globalen Variable `indexFile`. Dann starten wir den Server mit der Methode `listen()`. Das Entscheidende ist, dass die Datei geladen wird, bevor der Server ausgeführt wird. Auf diese Weise wird die Funktion `requestListener()` eine HTML-Seite zurückgeben, da `indexFile` keine leere Variable mehr ist.

Unsere Fehlerhandlung hat sich auch geändert. Wenn die Datei nicht geladen werden kann, erfassen wir den Fehler und geben ihn in unserer Konsole aus. Dann beenden w SCROLL TO TOP gramm mit der Funktion `exit()`, ohne den Server zu starten. So können wir sehen, warum das Lesen

der Datei fehlgeschlagen ist, das Problem adressieren und dann den Server neu starten.

Wir haben nun verschiedene Webserver erstellt, die verschiedene Arten von Daten an einen Benutzer zurückgeben. Bisher haben wir keine Anfragedaten verwendet, um zu bestimmen, was zurückgegeben werden soll. Wir müssen Anfragedaten verwenden, wenn wir verschiedene Routen oder Wege in einem Node.js-Server einrichten. Lassen Sie uns also als Nächstes betrachten, wie diese zusammenarbeiten.

Schritt 4 — Verwalten von Routen mit einem HTTP-Anfrageobjekt

Die meisten Websites, die wir besuchen, oder APIs, die wir verwenden, weisen normalerweise mehr als einen Endpunkt auf, sodass wir auf verschiedene Ressourcen zugreifen können. Ein gutes Beispiel dafür wäre ein Buchverwaltungssystem, das in einer Bücherei zum Einsatz kommt. Es müsste nicht nur Buchdaten verwalten, sondern auch Autorendaten, um das Angebot zu katalogisieren und Suchen möglichst einfach zu gestalten.

Zwar sind die Daten für Bücher und Autoren verwandt, doch sind sie zwei verschiedene Objekte. In diesem Fall codieren Softwareentwickler meist jedes Objekt auf verschiedenen Endpunkten, um den API-Benutzern zu zeigen, mit welchen Arten von Daten sie interagieren.

Lassen Sie uns einen neuen Server für eine kleine Bücherei erstellen, der zwei verschiedene Arten von Daten zurückgibt. Wenn Benutzer die Adresse unseres Servers unter `/books` aufrufen, erhalten Sie eine Liste von Büchern in JSON. Wenn sie zu `/authors` gehen, erhalten sie eine Liste von Autorendaten in JSON.

Bisher haben wir auf jede Anfrage, die wir erhalten haben, die gleiche Antwort zurückgegeben. Lassen Sie uns das kurz illustrieren.

Führen Sie unser JSON-Antwortbeispiel erneut aus:

```
$ node json.js
```

Lassen Sie uns wie zuvor in einem anderen Terminal eine cURL-Anfrage ausführen:

```
$ curl http://localhost:8000
```

Sie sehen:

```
Output
{"message": "This is a JSON response"}
```

Lassen Sie uns nun einen weiteren cURL-Befehl ausprobieren:

```
$ curl http://localhost:8000/todos
```

Nach dem Drücken von `Enter` sehen Sie das gleiche Ergebnis:

```
Output
{"message": "This is a JSON response"}
```

Wir haben keine spezielle Logik in unserer Funktion `requestListener()` zur Bearbeitung einer Anfrage erstellt, deren URL `/todos` enthält, sodass Node.js standardmäßig die gleiche JS SCROLL TO TOP ückgibt.

Da wir einen Miniaturserver zur Buchverwaltung erstellen möchten, trennen wir nun die Art von Daten, die zurückgegeben werden sollen, je nach Endpunkt der Benutzerzugriffe.

Beenden Sie zuerst den laufenden Server mit `Strg+C`.

Öffnen Sie nun `routes.js` in Ihrem Texteditor:

```
$ nano routes.js
```

Lassen Sie uns zunächst unsere JSON-Daten in Variablen vor der Funktion `requestListener()` speichern:

first-servers/routes.js

```
...
const books = JSON.stringify([
    { title: "The Alchemist", author: "Paulo Coelho", year: 1988 },
    { title: "The Prophet", author: "Kahlil Gibran", year: 1923 }
]);

const authors = JSON.stringify([
    { name: "Paulo Coelho", countryOfBirth: "Brazil", yearOfBirth: 1947 },
    { name: "Kahlil Gibran", countryOfBirth: "Lebanon", yearOfBirth: 1883 }
]);
...
```

Die Variable `books` ist eine Zeichenfolge, die JSON für eine Gruppe von Buchobjekten enthält. Jedes Buch hat einen Titel oder Namen, einen Autor und ein Veröffentlichungsjahr.

Die Variable `authors` ist eine Zeichenfolge, die JSON für eine Reihe von Autorenobjekten enthält. Jed SCROLL TO TOP Namen, ein Geburtsland und ein Geburtsjahr.

Nachdem wir nun über die Daten verfügen, die unsere Antworten zurückgeben werden, beginnen wir, die Funktion `requestListener()` zu ändern, um sie auf die richtigen Routen zurückzuleiten.

Zuerst stellen wir sicher, dass jede Antwort von unserem Server den richtigen `Content-Type`-Header aufweist:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
}
...
```

Jetzt möchten wir je nach URL-Pfad, den der Benutzer besucht, die richtige JSON zurückgeben. Lassen Sie uns eine `switch`-Anweisung zur URL der Anfrage erstellen:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {}
}
...
```

Um den URL-Pfad eines Anfrageobjekts zu erhalten, müssen wir auf seine `url`-Eigenschaft zugreifen. Wir können nun Fälle in die `switch`-Anweisung einfügen, um die entsprechende JSON zurückzugeben.

Die `switch`-Anweisung von JavaScript bietet eine Möglichkeit zu steuern, welcher Code je nach Wert eines Objekts oder JavaScript-Ausdrucks (z. B. Ergebnis mathematischer Ope[SCROLL TO TOP]uhrt wird. Wenn Sie eine

Lektion oder Erinnerung zu ihrer Verwendung benötigen, lesen Sie unseren Leitfaden Verwenden der switch-Anweisung in JavaScript.

Lassen Sie uns einen `case` (Fall) für die Situation einfügen, dass der Benutzer unsere Liste von Büchern erhalten möchte:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
    }
}
...
```

Wir setzen unseren Statuscode auf `200`, um anzuzeigen, dass die Anfrage in Ordnung ist, und geben die JSON zurück, die die Liste unserer Bücher enthält. Lassen Sie uns nun einen weiteren `case` (Fall) für unsere Autoren einfügen:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
            res.writeHead(20
            res.end(authors),
```

```
            break
    }
}
...
```

Wie zuvor wird der Statuscode `200` betragen, da die Anfrage in Ordnung ist. Dieses Mal geben wir die JSON zurück, die die Liste unserer Autoren enthält.

Wir wollen einen Fehler zurückgeben, wenn der Benutzer einen beliebigen anderen Pfad aufrufen möchte. Lassen Sie uns dazu den Standardfall einfügen:

routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
            res.writeHead(200);
            res.end(authors);
            break
        default:
            res.writeHead(404);
            res.end(JSON.stringify({error:"Resource not found"}));
    }
}
...
```

Wir verwenden das `default`-Stichwort in einer `switch`-Anweisung, um alle anderen Szenarien zu erfassen, die von unseren vorherigen Fällen nicht abgedeckt werden. Wir setzen den Statuscode auf `404`, um anzuzeigen, dass die gewünschte URL nicht ge[SCROLL TO TOP]nn legen wir ein JSON-Objekt fest, das eine Fehlermeldung enthält.

Lassen Sie uns unseren Server testen, um zu sehen, ob er sich wie erwartet verhält. Führen wir in einem anderen Terminal zuerst einen Befehl aus, um zu sehen, ob wir unsere Liste mit Büchern erhalten:

```
$ curl http://localhost:8000/books
```

Drücken Sie `Enter`, um die folgende Ausgabe zu sehen:

```
Output
[{"title":"The Alchemist","author":"Paulo Coelho","year":1988},{"title":"The Pr
```

Das funktioniert schon einmal gut. Lassen Sie uns das gleiche für `/authors` ausprobieren. Geben Sie den folgenden Befehl im Terminal ein:

```
$ curl http://localhost:8000/authors
```

Sie sehen die folgende Ausgabe, sobald der Befehl abgeschlossen wurde:

```
Output
[{"name":"Paulo Coelho","countryOfBirth":"Brazil","yearOfBirth":1947},{"name":'
```

Lassen Sie uns zuletzt eine fehlerhafte URL ausprobieren, um sicherzustellen, dass `requestListener()` die Fehlerantwort zurückgibt:

```
$ curl http://localhost:8000/notreal
```

Bei Eingabe dieses Befehls wird folgende Nachricht angezeigt:

```
Output
{"error":"Resource not found"}
```

Sie können den laufenden Server mit `Strg+C` beenden.

Wir haben nun verschiedene Wege für Benutzer erstellt, um verschiedene Daten zu erhalten. Wir haben auch eine Standardantwort hinzugefügt, die einen HTTP-Fehler zurückgibt, wenn der Benutzer eine URL eingibt, die wir nicht unterstützen.

Zusammenfassung

In diesem Tutorial haben Sie eine Reihe von Node.js-HTTP-Servern erstellt. Zuerst haben Sie eine einfache Textantwort zurückgegeben. Dann haben Sie von unserem Server verschiedene Arten von Daten zurückgegeben: JSON, CSV und HTML. Ab da konnten Sie das Laden von Dateien mit HTTP-Antworten kombinieren, um eine HTML-Seite vom Server an den Benutzer zurückzugeben, und eine API erstellen, die Informationen über die Anfrage des Benutzers verwendet, um zu bestimmen, welche Daten in der Antwort gesendet werden sollen.

Sie sind nun dazu in der Lage, Webserver zu erstellen, die eine Vielzahl von Anfragen und Antworten bearbeiten können. Mit diesem Wissen können Sie einen Server einrichten, der an verschiedenen Endpunkten viele HTML-Seiten an den Benutzer zurückgibt. Außerdem können Sie Ihre eigene API erstellen.

Um mehr über weitere HTTP-Webserver in Node.js zu erfahren, können Sie die Node.js-Dokumentation im `http`-Modul lesen. Wenn Sie noch mehr über Node.js erfahren möchten, können Sie zu der Serienseite Codieren in Node.js zurückkehren.