

Express – Node.js Web-Framework

Express ist ein flexibles Node.js Framework, das viele Features zum Bauen von Web- und Mobile-Applikationen bietet.

My Express Chat Server

Ziel dieser Übung ist es, den Node.js Chatserver als eine Node.js Express App zu implementieren.

Anforderungen:

- 1:1 Klon von MyChatServer
- Der Server ist mit Express realisiert
 - Alle statischen Files aus dem Ordner *html* und *css* sind public und können gelesen werden
 - Das Chat Frontend wird über den HTTP GET Request */chat* angefordert
 - Die Nachrichten werden über den HTTP POST Request */chat* gespeichert
- REST API
 - Über den HTTP GET Request */chat/<id>* kann eine einzelne Nachricht mit der jeweiligen ID abgefragt werden. Die Nachricht mit Name und Message wird im JSON-Format retourniert.

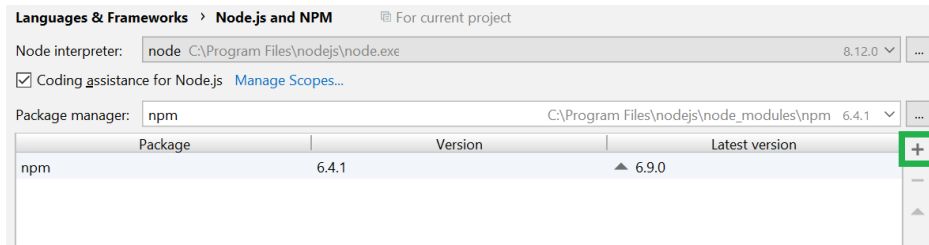
Verzeichnis:

Tutorial 1: Express installieren	2
Tutorial 2: Express Server hosten	2
Tutorial 3: Routing Static Files mit Express	3
Tutorial 4: POST Routing mit Express	4
Tutorial 5: RESTful API mit Express	7
Exkurs: Ladezeiten und Browser Caching.....	9

Tutorial 1: Express installieren

Bevor Sie Express nutzen können, müssen Sie das Modul installieren.

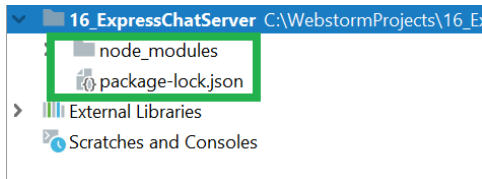
- Variante 1: mit NPM in der Konsole (siehe Skript)
- Variante 2: über das WebStorm Menü
 - *File -> Settings*
 - *Languages & Frameworks*
 - *Node.js and NPM*



- Express suchen und installieren:



Danach können Sie die Einstellungen verlassen. Bei erfolgreicher Installation müssen nun im Projektverzeichnis der Ordner *node_modules* und die Datei *package-lock.json* angelegt worden sein.



Tutorial 2: Express Server hosten

Legen Sie im Projektverzeichnis eine neue, leere Datei *app.js* an (sie wird die *server.js* Datei ersetzen, bei Express ist es Quasistandard die Datei mit *app* zu benennen).

Wie bei allen Modulen müssen Sie zunächst das Express Modul importieren. Anschließend kann Express gestartet werden:

```
// init express app
const express = require('express');
const app = express();
```

Die Express App ist in der Konstanten *const app* gespeichert. Dadurch haben wir Zugriff auf den Server und können ihn konfigurieren. Als erstes definieren wir den Port, über den der Server erreichbar ist:

```
// config server
app.listen(3000, () => {
  console.log('Access via localhost or IP 192.168.1.205');
  console.log('Listening to port: ' + 3000);
});
```



Kommt Ihnen der Code bekannt vor?
Suchen Sie in der `server.js` Datei der letzten Übung nach `server.listen(...)`.

Ein wesentlicher Vorteil von Express ist, dass man die einzelnen Routen übersichtlicher definieren kann. Für einfache Routen per GET Request verwendet man die Funktion `app.get(...)`:

```
app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

Testen Sie den Code! Starten Sie den Server und rufen Sie das Rootverzeichnis <http://localhost:3000> auf. Der Browser sollte Sie mit „Hello World“ begrüßen. 🔍



Mit Express kann man die Daten mit der Funktion `res.send(...)` schicken:

Vorher:

```
res.writeHead( statusCode: 200, headers: { 'Content-Type': 'text/html' } );
res.write(data);
return res.end();
```

Mit Express

```
res.send(data);
```

Hier wird der Status Code sowie der Content-Type automatisch mitgeliefert (mehr dazu: siehe Exkurs „Caching“)

Tutorial 3: Routing Static Files mit Express

Bei Ihrem Chatserver waren zuletzt fünf Routen definiert

- Externe Stylesheet /style.css
- Chat Frontend /chat.html
- Chat Upload /chat_upload
- Startseite | Root, /
- andere HTML Dateien /dateiname.html

Ein Großteil der Routen beschäftigt sich eigentlich nur damit statischen Inhalt bereitzustellen – es wird eine Datei mit dem FS Modul gelesen und als Antwort zurückgesendet.

Das geht in Express einfacher mit der Middlewarefunktion `express.static`! Als Parameter übergibt den internen Pfad zu dem `public` Ordner – dadurch sind alle Dateien darin öffentlich abrufbar.

In unserem Fall wollen wir alle CSS und HTML Dateien freigeben:

```
// init middleware
app.use(express.static('public/css'));
app.use(express.static('public/html'));
```

Testen Sie den Code! Was passiert bei Aufruf des Root Verzeichnisses <http://localhost:3000>? 

Jetzt fehlen nur mehr die speziellen Routen, bei denen mehr passiert als einfach nur eine Datei lesen und senden: `/chat` und `/chat_upload`. Dazu müssen Sie auf die Funktion `app.get(...)` zurückgreifen (mit der Sie zuvor „Hello World“ ausgegeben haben):

```
app.get('/chat_upload', (req, res) => {
  /* process chat query */

});
```

```
app.get('/chat', (req, res) => {
  /* load files (with sync methods) */
  /* build formatted data */
  /* send html page */

});
```

Der Code bzw. Algorithmus innerhalb der Routen bleibt größtenteils unverändert (Copy and Paste), außer:

- Sie müssen noch das **fs** Modul importieren
- Auf die Query-Objekte des GET Requests können Sie direkt mit **req.query.name** und **req.query.message** zugreifen. Sie benötigen dazu nicht mehr das URL Module!
- Für das erfolgreiche retournieren von Daten:
verwenden Sie **res.send(data)** anstelle von `res.writeHead(200, ...)`, `res.write(data)` und `res.end()`.
- Für 404 Fehlermeldungen:
verwenden sie **res.status(404).send('Ihre Fehlermeldung')** anstelle von `res.writeHead(404, ...)` und `res.end(...)`.

Tutorial 4: POST Routing mit Express

GET Requests werden eigentlich nur für Anfragen an den Server verwendet. Zum Beispiel


- Man fordert von YouTube ein Video mit einer ID:
<https://www.youtube.com/watch?v=hQ6JSJsNWZ0&list=PLR56THkyjMv7cP1oKLe-uSt1joQXJpo2m>
- Man fordert von Google ein Suchergebnis an:
<https://www.google.at/search?q=Htl+leonding>
- Man fordert vom Localhost alle Nachrichten von Max an:
<http://localhost:3000/chat.html?author=max> -> das wär spannend!

Um Daten an einen Server zu senden und dort zu speichern, verwendet man einen HTTP Request mit der POST Methode. Die Daten werden dabei nicht in der URL übertragen, sondern im Body des Requests übertragen.

Einer der Vorteile: POST Daten haben keine Einschränkung bei der Größe (bei GET können in der URL maximal 2048 Zeichen gesendet werden).

In Express ist die GET Route schnell auf eine POST Route umgeschrieben:

```
app.post('/chat_upload', (req, res) => {  
  /* code unchanged */  
});
```

Testen Sie den Code! 



Cannot GET /chat_upload

Offensichtlich läuft etwas schief – der Server antwortet nicht. Sie müssen im Formular angeben, dass die Daten mit der POST Methode gesendet werden sollen (per Default werden sie nämlich mit GET versandt):

```
<form action="/chat_upload" method="post">
```

Am Server ist auch eine kleine Anpassung notwendig: Sie müssen nicht mehr die Query auslesen, sondern den Inhalt des Bodys. Ersetzen Sie dazu `req.query.name` mit `req.body.name` sowie `req.query.message` mit `req.body.message`.

Testen Sie den Code erneut! 



Der Code funktioniert immer noch nicht, es werden keine Nachrichten gespeichert.

Jetzt müssen Sie debuggen: überprüfen Sie, welche Daten überhaupt ausgelesen werden mit einem einfachen `console.log(...)`:

```
app.post('/chat_upload', (req, res) => {  
  console.log(req.body);  
  console.log(req.body.name);  
  console.log(req.body.message);  
  res.send('debugging...');  
});
```

Starten Sie erneut den Server und senden Sie Daten über das Formular. 

Post a message:



Sie bekommen 2 Rückmeldungen:

`undefined` -> Das Body Objekt ist offensichtlich leer

`TypeError: Cannot read property 'name' of undefined`

-> von einem leeren Objekt kann keine Eigenschaft gelesen werden

Sie benötigen noch einen Parser, der den Inhalt des Bodys aufbereitet, sodass er über die Punktnotation lesbar ist. Dafür greifen wir auf das **Plugin body-parser** zu: <https://www.npmjs.com/package/body-parser>.

- Installieren Sie das Plugin über die Kommandozeile oder über die WebStorm Settings.

```
$ npm install body-parser
```

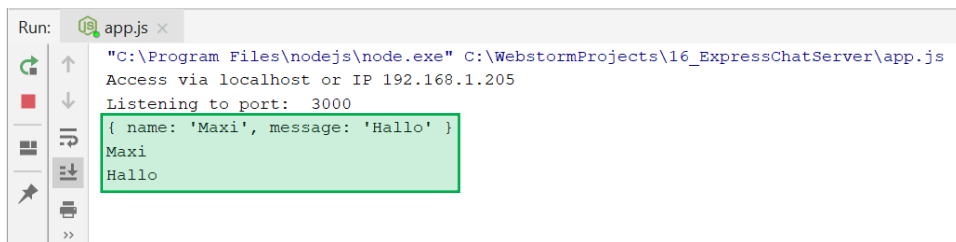
- Importieren Sie das Modul zu Beginn ihrer App.js Datei

```
const bodyParser = require('body-parser');
```

- Binden Sie den Parser als Middleware in die Express App ein:

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

Starten Sie erneut den Server und senden Sie Daten über das Formular. 🔍



```
Run: app.js x
"C:\Program Files\nodejs\node.exe" C:\WebstormProjects\16_ExpressChatServer\app.js
Access via localhost or IP 192.168.1.205
Listening to port: 3000
{ name: 'Maxi', message: 'Hallo' }
Maxi
Hallo
```

Bei jedem Request wird nun der Inhalt des Bodys von der Middleware *body-parser* in ein lesbares Objekt konvertiert!

Fügen Sie nun wieder den Originalcode ein (der die Daten auslest und in einem File speichert).

Starten Sie erneut den Server und senden Sie wieder Daten über das Formular. 🔍

Der Express Server erfüllt nun die gleichen Funktionen wie Ihr selbst gecodeter Server der letzten Übung!

Erweiterung: HTTP GET und POST Request über gleiche Route

Die Unterscheidung von Requests mit der GET Methode und mit der POST Methode bringt einen weiteren Vorteil mit sich: man kann beide Methoden über die gleiche Route bedienen!

Sie benötigen daher keine extra Route */chat_upload*. Sie können den Upload der Nachrichten auch über die Route */chat* abwickeln:

```
app.get('/chat', (req, res) => {
  /* code for displaying the chat room frontend */
});

app.post('/chat', (req, res) => {
  /* code for saving the message */
});
```



Denken Sie daran auch den Verweis im HTML Formular anzupassen:

```
<form action="/chat"
      method="post">
```

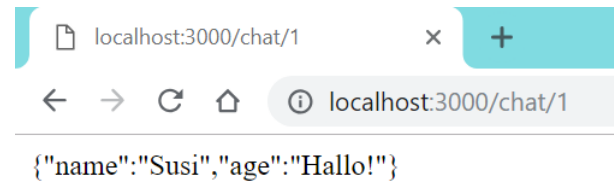
Testen Sie den Code erneut, indem Sie Nachrichten versenden! 🔍

Tutorial 5: RESTful API mit Express

Ziel dieses Abschnitts ist, eine Schnittstelle (API, Programmierschnittstelle) zu konfigurieren, über die einzelne Nachrichten über ihre ID abgefragt werden können.

Beispiele:

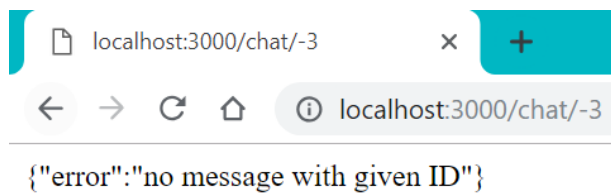
der Aufruf der URL <http://localhost:3000/chat/1> soll die erste gepostete Nachricht retournieren.



der Aufruf der URL <http://localhost:3000/chat/4> soll die vierte gepostete Nachricht retournieren.



der Aufruf der URL <http://localhost:3000/chat/-3> soll eine Fehlermeldung retournieren.



Let's start! Als Erstes müssen Sie die Route in Ihrer App definieren. Das funktioniert mithilfe eines Parameters – eingeleitet durch einen Doppelpunkt, gefolgt vom gewünschten Namen.

```
// route, get method with parameter
app.get('/chat/:id', (req, res) => {
  console.log(req.params.id);
});
```



Der Parameter kann über das Objekt `req.params` ausgelesen werden.

Testen Sie den Code!

Beim Aufruf von <http://localhost:3000/chat/14> muss die Zahl 14 ausgegeben werden! 🔍

Nun müssen Sie wie beim Chat Frontend die Textdatei `chat.txt` auslesen und in Ihre einzelnen Zeilen *lines* zerlegen.

```
app.get('/chat/:id' , (req, res) => {
  let data = fs.readFile('./public/data/chat.txt', 'utf8', function (err, data) {
    if (err) {
      res.status(404).send('Sorry, no messages stored.');
```

Alle Zeilen sind im Array *lines* gespeichert. Nun müssen Sie überprüfen, ob der angefragte Post überhaupt existiert:

```
if (req.params.id > 0 && req.params.id <= lines.length) {
  // to be continued ...
}
else {
  res.send('error');
```

Bei erfolgreichem Test des Parameters können Sie nun die gewünschte Zeile aus dem Array extrahieren:

```
// get the requested line
let line = lines[req.params.id - 1];
let elements = line.split('separator: ');
```



Warum wird ID – 1 gerechnet?

Damit werden die Anfragen intuitiver, weil Arrays mit Index 0 beginnen.

Nun müssen Sie nur mehr die Antwort zusammenbauen. Eine ordentliche API retourniert die Daten im standardisierten JSON-Format (im Grunde ein ganz normaler String, aber mit eindeutiger Syntax aus geschwungenen Klammern usw.).

```
if (elements[1] !== undefined) {
  let obj = { name: elements[0], age: elements[1] };
  res.send(JSON.stringify(obj));
}
else {
  res.send('error');
```



An alle sauberen Webentwickler:

Auch die Fehlermeldung soll bei einer API als JSON rückgemeldet werden, also besser so:

```
let obj = { error: "empty message" };
res.send(JSON.stringify(obj));
```

Testen Sie den Code mit einigen möglichen Fällen:



<http://localhost:3000/chat/1>

<http://localhost:3000/chat/4>

<http://localhost:3000/chat/-7>

<http://localhost:3000/chat/0>

<http://localhost:3000/chat/201>

Exkurs: Ladezeiten und Browser Caching

Das Um und Auf einer Webseite ist ihre Ladezeit. Dabei spielen unzählige Punkte wie Serverhardware, Serverkonfiguration, Netzwerkverbindung uvm. eine Rolle.

Viele Files werden beim Besuch einer Webseite mehr als 1x aufgerufen – und verändern sich dabei nicht! Das ruft nach Browser Caching. Der Browser soll die Datei nicht unzählige Mal neu vom Server laden, sondern stattdessen eine lokale im Cache gespeicherte Version laden.

Untersuchung des Netzwerktransfers unseres Chatserver

(using Chrome Development Tool: F12 -> Network)

„Übung 15: MyChatServer“ unter der Lupe:

Aufruf von <http://localhost:3000>

- Ladezeit gesamt ca. 400 ms
- Status Code immer 200 -> die Dateien werden jedes Mal neu vom Server geladen

Das Favicon wird nur einmal pro Session geladen (d.h. solange der Browser nicht geschlossen wird)

Aufruf von <http://localhost:3000/index.html>

- Ladezeit gesamt ca. 20 ms
- Status Code immer 200 -> die Dateien werden jedes Mal neu vom Server geladen

2 requests | 3.4 KB transferred | 3.2 KB resources | Finish: 16 ms | DOMContentLoaded: 12 ms | Load: 17 ms

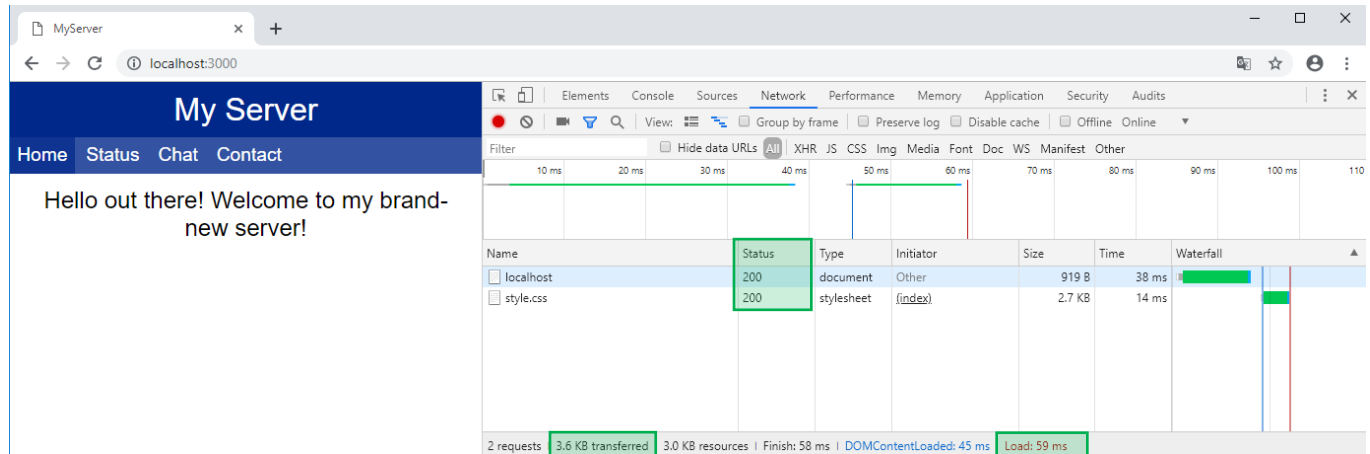


Wieso ist der Aufruf von <http://localhost:3000/index.html> um so viel schneller als <http://localhost:3000>?
Stichwort: Weiterleitung.

„Übung 16: MyExpressChatServer“ unter der Lupe:

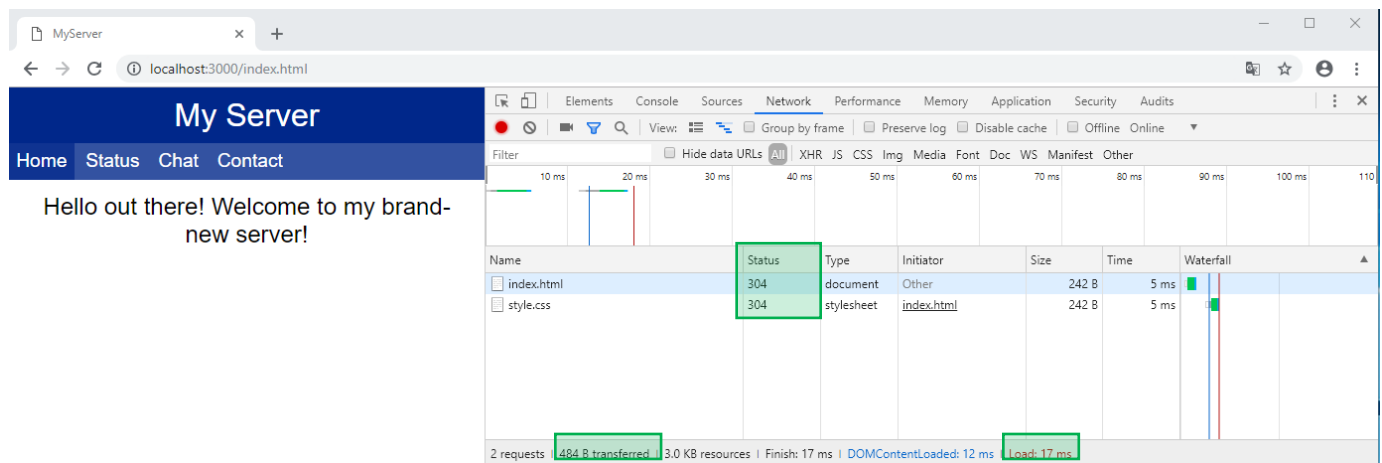
Aufruf von <http://localhost:3000>

- Ladezeit (Erstaufwurf) gesamt ca. 60 ms -> schneller wegen Express
- Status Code (Erstaufwurf) 200 -> Dateien werden neu vom Server geladen
- Bytes transferred 3.6 KB = 3600 Byte



Aufruf von <http://localhost:3000/index.html>

- Ladezeit (Zweitaufwurf) gesamt ca. 17 ms
- Status Code (Zweitaufwurf) 304 -> Datei seit letztem Mal unverändert
- Bytes transferred 484 Byte -> weniger, weil Dateien nicht neu geladen



Untersuchen Sie die Route `/chat.html` von Ihrem Chat Server (Übung 15) und Ihrem Express Server (Übung 16) mit dem Google Chrome Development Tool und dokumentieren Sie ihre Auswertung in einem Protokoll!

