

HTTP Module + URL Module

Wie wir nun wissen, ermöglicht das Node.js HTTP Module das Bauen eines Webserver. Um das Routing und das Bearbeiten von Requests zu vereinfachen, verwendet man zusätzlich das URL Module. Damit lässt sich die Request URL in ihre einzelnen Bestandteile aufteilen und somit leichter auswerten.

My Chat Server

Ziel dieser Übung ist es, den Node.js Server mit einer Schnittstelle zu erweitern und eine einfache Kommunikation zwischen Client-App und Back-End zu demonstrieren.

Als Beispiel dient ein kleiner Chatroom. Darin kann jeder selbst Nachrichten posten und man sieht alle bisher geposteten Beiträge.

Anforderungen:

- 1:1 Klon von My First Webserver
- Zusätzliche Seite *chat.html*, die als Schnittstelle mit dem Back-End Service fungiert

Chat Front-End:

- Es werden alle bisher geposteten Beiträge angezeigt
- User kann in einem Formular seinen Namen und eine Nachricht eingeben und mit Button versenden
- Beim Buttonklick werden die Daten mit der GET Methode an die Route *./chat_upload* gesendet

Chat Back-End:

- Daten werden per GET Request über die Route *chat_upload* empfangen
- Daten (Namen + Nachricht) werden in einem Textfile *chat.txt* gespeichert
- Als Antwort leitet das Back-End auf eine Bestätigungsseite weiter:
 - *chat_confirm.html* falls das Speichern erfolgreich war
 - *chat_error.html* falls es bei den Daten oder beim Speichern einen Fehler gab

Verzeichnis:

Tutorial 1: Server erstellen	3
Tutorial 2: Routing mit dem URL-Module	3
Tutorial 3: Chatroom Front-End.....	4
Tutorial 4: Chatroom Back-End (Daten am Server auslesen)	6
Tutorial 5: Chatroom Back-End (Daten am Server speichern)	7
Tutorial 6: Chatroom Back-End (<i>chat.html</i> mit allen Nachrichten retournieren)	9

Screenshots der Musterlösung:

<http://localhost:3000/chat>

My Server

Home Status Chat Contact

Chatroom

Post a message:

Name

Submit

Message

Susi
Hallo!

Max
Hello World!

Susi
How are you?

<http://localhost:3000/chat> (senden einer Nachricht)

My Server

Home Status Chat Contact

Chatroom

Post a message:

Name

Submit

Message



http://localhost:3000/chat_confirm.html

My Server

Home Status Chat Contact

Chatroom

Thanks for your message!

back to Chatroom



My Server

Home Status Chat Contact

Chatroom

Post a message:

Name

Submit

Message

Susi
Hallo!

Max
Hello World!

Susi
How are you?

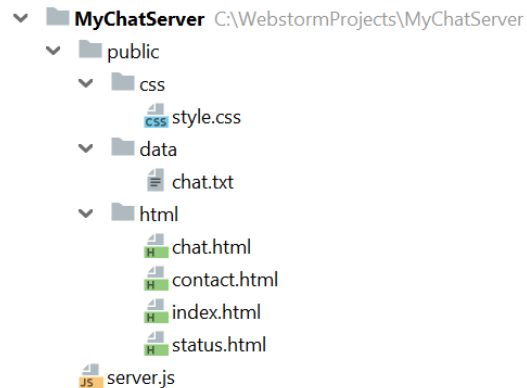
Max
Fine, thanks!

Tutorial 1: Server erstellen

Legen Sie ein neues WebStorm-Projekt *MyChatServer* an. Kopieren Sie die JS-Datei, die CSS-Datei und alle HTML-Dateien Ihres *MyFirstWebserver*.

Für eine übersichtliche Serverstruktur empfiehlt es sich alle öffentlich verfügbaren Dateien in einem *public* Order zu sammeln.

Erstellen Sie eine neue HTML-Datei *chat.html*. Aktualisieren Sie die Navigation (zusätzlicher Link zum Chatroom) in allen HTML-Files und passen Sie die Routen der JavaScript Datei an.



Tutorial 2: Routing mit dem URL-Module

Beim Routing haben wir bisher für jede Unterseite die gleiche Routine verwendet:

1. Abfrage des genauen Links
 2. Lesen des Files
 3. Den Inhalt des Files als Antwort senden
- ```
if (req.url === '/contact') {
 fs.readFile('html/contact.html', function (err, data) {
 res.writeHead(statusCode: 200, headers: { 'Content-Type': 'text/html' });
 res.write(data);
 res.end();
 });
}
```

Nachdem der Ablauf immer gleich ist, lässt sich das automatisieren. Wir verwenden dazu das URL-Module.

Zuerst müssen Sie wieder das URL-Module einbinden: `let url = require( id: 'url' );`

Nun können Sie mit der *parse* Methode die URL zerlegen:

```
let server = http.createServer(requestListener: function (req, res) {
 let q = url.parse(req.url, parseQueryString: true);
 /* routing code missing ... */
});
```

Das geparsete Objekt (gespeichert in der Variable *q*) besitzt nun mehrere Eigenschaften, die man Auslesen kann:

- *q.pathname* der Pfad (relativ zur Hostadresse)
- *q.host* der Hostname (Achtung: bei *localhost* ist *q.host = null*)
- *q.search* der GET-Request String

Die Route zum Stylesheet mit Verwendung des Pathnames sieht dann so aus:

```
/* routing to stylesheet */
if (q.pathname === '/style.css') {
 ...
}
```

Die Route zu den HTML können Sie nun automatisieren:

zuerst erstellt man den relativen Pfad, der zur gewünschten HTML-Datei führt.

```
let filename = './public/html' + q.pathname;
```

Anschließend liest man genau dieses File aus. Natürlich kann es jetzt aber passieren, dass ein File verlangt wird, das gar nicht existiert. Diesen Fall muss man dann mit einer Fehlermeldung behandeln. Im Webstandard steht dafür der HTTP-Statuscode 404 („file not found“). Die zweite Route ist somit (inkl. Fehlerbehandlung) wie folgt:

```
else {
 /* create intern file path */
 let filename = './public/html' + q.pathname;
 /* read and output file */
 fs.readFile(filename, 'utf8', function (err, data) {
 if (err) {
 res.writeHead(statusCode: 404, headers: { 'Content-Type': 'text/html' });
 return res.end(chunk: '404 Not Found');
 }
 res.writeHead(statusCode: 200, headers: { 'Content-Type': 'text/html' });
 res.write(data);
 return res.end();
 });
}
```

Was passiert nun, wenn nur `http://localhost:3000` im Browser aufgerufen wird?

Das Back-End baut den String `./public/html/` zusammen – dieses File existiert aber nicht! Stattdessen sollte eigentlich die Startseite `index.html` geöffnet werden (so wie wir es als User auch gewohnt sind!).

Diesen Fall muss man also wieder separat behandeln. Eine mögliche Lösung wäre zuerst immer auf die Startseite zu leiten und bei Anfrage einer Unterseite den String überschreiben:

```
let filename = './public/html/index.html'; /* default route to cover page */
if (q.pathname !== '/') {
 filename = './public/html' + q.pathname;
}
```



Denken Sie daran, den Code immer wieder zu testen!

## Tutorial 3: Chatroom Front-End

Die Chat-Seite soll zwei Funktionalitäten bieten:

1. Ein Formular für den Namen und die Nachricht mit Submit Button
2. Alle bisherigen geposteten Nachrichten anzeigen

### 1. Formular

Das Formular muss aus drei Feldern bestehen: Namen, Nachricht und Sendebutton.

Als Zieladresse (*action* Attribut) des Back-Ends ist `./chat_upload`.

Wichtig sind die *name* Attribute. Sie geben den Namen der Query-Strings Parameter an – mit den gleichen Namen kann man dann in JavaScript darauf zugreifen!

### 2. Nachrichten

Für die geposteten Nachrichten reservieren wir uns erstmal einen *div* Block. Der Inhalt folgt später.

Muster HTML-Code des Formulars und des reservierten *div* Blocks:

```
<main>
 <h2>Chatroom</h2>
 <form action="/chat upload">
 <h3>Post a message:</h3>
 <p>Name</p><input type="text" name="name" placeholder="Your name..">
 <input type="submit" value="Submit ">

 <p>Message</p><textarea name="message" rows="6" placeholder="Your message.."></textarea>
 </form>
 <div id="chat">
 <!-- chat messages will be displayed here -->
 </div>
</main>
```

Mit etwas CSS Code lässt sich das Formular ansprechend gestalten.

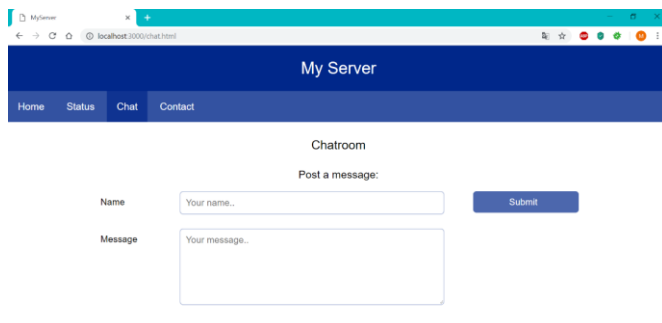
Zur Wiederholung siehe W3Schools: [https://www.w3schools.com/css/css\\_form.asp](https://www.w3schools.com/css/css_form.asp)

### Muster CSS-Code

```
}main form {
 padding-bottom: 5%;
 margin-bottom: 5%;
 border-bottom: 0.1vw solid rgba(0,39,138,0.7);
 border-radius: 0.1vw;
}
}
}main form h3 {
 margin: 2% 0;
 text-align: center;
 font-size: 1.3rem;
 font-weight: normal;
}
}
}main form p {
 display: inline-block;
 vertical-align: top;
 width: 15%;
 margin: 0 0 3% 5%;
 padding: 1% 0;
 font-size: 1.1rem;
}
}
}
```

```
}main form input, main form textarea {
 display: inline-block;
 vertical-align: top;
 box-sizing: border-box;
 width: 50%;
 padding: 1%;
 border-radius: 0.5vw;
 border: 0.12vw solid rgba(0,39,138,0.3);
 font-family: sans-serif;
 font-size: 1.1rem;
 outline: none;
}
}
}main form input:focus, main form textarea:focus {
 border: 0.15vw solid rgba(0,39,138,0.8);
 outline: none;
}
}
}main form input[type='submit'] {
 color: #fff;
 background-color: rgba(0,39,138,0.7);
 border: none;
 width: 20%;
 border-radius: 0.5vw;
 margin-left: 5%;
}
}
}main form input[type='submit']:hover {
 background-color: rgba(0,39,138,0.8);
 cursor: pointer;
}
}
```

### Oberfläche des Musters



## Tutorial 4: Chatroom Back-End (Daten am Server auslesen)

Beim Klick auf den Submit Button wird ein GET-Request erstellt. Eine URL könnte dann sein:

localhost:3000/chat\_upload?name=Max&message=HelloWorld%21

Zuerst benötigen wir wieder eine Abfrage der Route. Der Request wird auf der URL `/chat_upload` erwartet:

```
else if (q.pathname === '/chat_upload') {
```

Nun müssen wir Zugriff auf die Parameter `name` und `message` sowie deren Werte erhalten. Dazu verwenden wir wieder das URL-Module: mit `q.query` kann man nämlich direkt auf alle Parameter zugreifen! Sie müssen nur wissen, welche Namen sie den Formularfeldern zugewiesen haben.

Testen Sie zuerst einmal das Auslesen der Query mit einer einfachen Konsolenausgabe:

```
else if (q.pathname === '/chat_upload') {
 /* process chat query */
 console.log(q.query.name);
 console.log(q.query.message);

 res.write(chunk: '' +
 '<p>Data successfully submitted.</p>' +
 'back to Chatroom');
 return res.end();
}
```



Das war der HTML-Code des Formulars, wo die Namen vergeben wurden:

```
<input type="text" name="name" placeholder="Name" />
<textarea name="message" rows="6" placeholder="Post a message:" />
```

Starten Sie dazu den Server und übermitteln Sie eine Nachricht!



localhost:3000/chat\_upload?name=Max&message=Hello%21

Data successfully submitted.

[back to Chatroom](#)

Und in der Konsole:

Geschafft – Sie können nun mit der GET-Methode Daten an den Server senden und auslesen.

## Tutorial 5: Chatroom Back-End (Daten am Server speichern)

Das nächste Ziel ist nun die Daten nicht nur auszugeben, sondern in der Textdatei *data.txt* zu speichern. Zuerst bauen wir mit den zwei Textteilen die vollständige Zeile zusammen (hier wird die *Template String* Syntax verwendet, der Strichpunkt dient als Trennzeichen und NewLine als Ende der Zeile):

```
let line = `${q.query.name};${q.query.message}\n`;
```

Nun müssen Sie die Zeile in der Textdatei *data.txt* speichern. Verwenden Sie dazu die bekannte Methode *appendFile(...)* und übergeben Sie die fertige Zeile als Parameter. Nach dem Speichern soll dann wieder der kurze Infotext als Antwort gesendet werden:

```
else if (q.pathname === '/chat_upload') {
 /* process chat query */
 let line = `${q.query.name};${q.query.message}\n`;

 /* append data to file */
 fs.appendFile('./public/data/chat.txt', line, 'utf8', function (err) {
 if (err) throw err;
 console.log('@file chat.txt: new data added.');

Weiterleitung, nachdem Daten gesichert wurden


```
    /* response, answer */
    res.write( chunk: '' +
      '<p>Data successfully submitted.</p>' +
      '<a href= ".">back to Chatroom</a>');
    return res.end();
  });
}
```


```



Man achte auf die Verschachtelung:

- *Res.write(...)* steht im Callback von *appendFile(...)*. Das heißt es wird erst nach dem Speichern der Daten aufgerufen.



Testen Sie wieder, ob der Code funktioniert! Senden Sie eine Nachricht – diese muss dann in der Textdatei erscheinen! Probieren Sie es dann gleich noch einmal.

```
chat.txt x style.css x server.js x
1
2 Max; Hallo!
3
4
```

```
chat.txt x style.css x server.js x
1
2 Max; Hallo!
3 Susi;Hallo zusammen!
4
```

Zwei Kleinigkeiten fehlen noch.

1. Was, wenn falsche Daten gesendet werden? Oder gar keine Daten (was häufig passieren kann, wenn User einfach auf den Submit Button drücken, ohne etwas zu schreiben)? Man benötigt daher vor dem Speichern eine Überprüfung des Query-Objekts.

```
else if (q.pathname === '/chat_upload') {
 /* process chat query */
 if (q.query.name && q.query.message) {
 /* correct data */
 }
 else {
 /* wrong or no data */
 }
}
```

- Nach dem Speichern soll auf die HTML Seite *chat\_confirm.html* weitergeleitet werden. Falls es einen Fehler gab soll auf die HTML Seite *chat.error.html* weitergeleitet werden.

```
else if (q.pathname === '/chat_upload') {
 /* process chat query */
 if (q.query.name && q.query.message) {
 /* correct data --> save in text file --> answer with chat_confirm.html */
 }
 else {
 /* wrong or no data --> answer with chat.error.html */
 }
}
```

### Musterlösung

```
if (q.query.name && q.query.message) {
 let line = `${q.query.name};${q.query.message}\n`;
 /* append data to file */
 fs.appendFile('./public/data/chat.txt', line, 'utf8', function (err) {
 if (err) throw err;
 console.log('@file chat.txt: new data added.');
```

```
 fs.readFile('./public/html/chat_confirm.html', 'utf8', function (err, data) {
 if (err) {
 res.writeHead(statusCode: 404, headers: {'Content-Type': 'text/html'});
 return res.end(chunk: '404 Not Found');
```

```
 }
 res.writeHead(statusCode: 200, headers: {'Content-Type': 'text/html'});
 res.write(data);
 return res.end();
 });
 });
} else {
 /* error: wrong data submitted */
 fs.readFile('./public/html/chat_error.html', 'utf8', function (err, data) {
 if (err) {
 res.writeHead(statusCode: 404, headers: {'Content-Type': 'text/html'});
 return res.end(chunk: '404 Not Found');
```

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



Denken Sie daran, den Code zu **testen**!

- Senden sie mehrere richtig ausgefüllte Nachrichten.
- Senden Sie eine Nachricht ohne Namen und ohne Text (→ Error).
- Senden Sie eine Nachricht ohne Namen (→ Error!)
- Senden Sie eine Nachricht nur mit Namen und ohne Text (→ Error!).



Zum Abschluss können Sie noch das Design der weitergeleiteten HTML-Seiten anpassen.

### chat\_confirm.html

```
<main>
 <h2>Chatroom</h2>
 <div id="message">
 <p>Thanks for your message!</p>
 back to Chatroom
 </div>
</main>
```

### chat\_error.html

```
<main>
 <h2>Chatroom</h2>
 <div id="message">
 <p>Something went wrong. Please try again.</p>
 back to Chatroom
 </div>
</main>
```

### CSS Code Erweiterung

```
main #message p {
 text-align: center;
 font-size: 1.1rem;
 margin: 2% 0;
}

main #message a {
 display: block;
 margin: 0 auto;
 color: #fff;
 background-color: rgba(0,39,138,0.7);
 width: 20%;
 padding: 1%;
 border-radius: 0.5vw;
 text-align: center;
 text-decoration: none;
 font-size: 1.1rem;
}

main #message a:hover {
 background-color: rgba(0,39,138,0.8);
 cursor: pointer;
}
```



## Tutorial 6: Chatroom Back-End (*chat.html* mit allen Nachrichten retournieren)

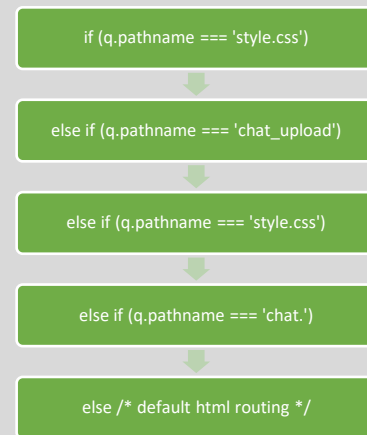
Das letzte Ziel ist alle Nachrichten aus der Textdatei in der HTML-Datei *chat.html* anzuzeigen.

Nachdem wir dafür einen eigenen Code brauchen, benötigen wir zuerst wieder eine neue Route:

```
/* chat page */
else if (q.pathname === '/chat.html') {
 /* load input formular */
 /* load all messages */
 /* send finished code */
}
```



Der Code ist  
inzwischen  
auf 5 Routen  
angewachsen:



Nun stoßen wir aber auf ein Problem: wir müssen zwei Files auslesen (die HTML-Datei und die Textdatei mit den Daten). Gleichzeitig dürfen wir aber nur eine Antwort mit *res.end(...)* senden.

Die einfachste – wenn auch nicht elegante und schon gar nicht performante – Lösung ist synchrones Lesen der Files. Damit haben Sie die Sicherheit, dass beide Files vollständig gelesen wurden!

```
/* chat page */
else if (q.pathname === '/chat.html') {
 /* load input formular */
 let chat_part1 = fs.readFileSync(path: './public/html/chat.html', options: 'utf8');

 /* load all messages */
 let chat_part2 = fs.readFileSync(path: './public/data/chat.txt', options: 'utf8');

 /* send finished code */
 res.writeHead(statusCode: 200, headers: { 'Content-Type': 'text/html' });
 res.write(chunk: `${chat_part1}${chat_part2}`);
 return res.end();
}
```

Testen Sie den Code!



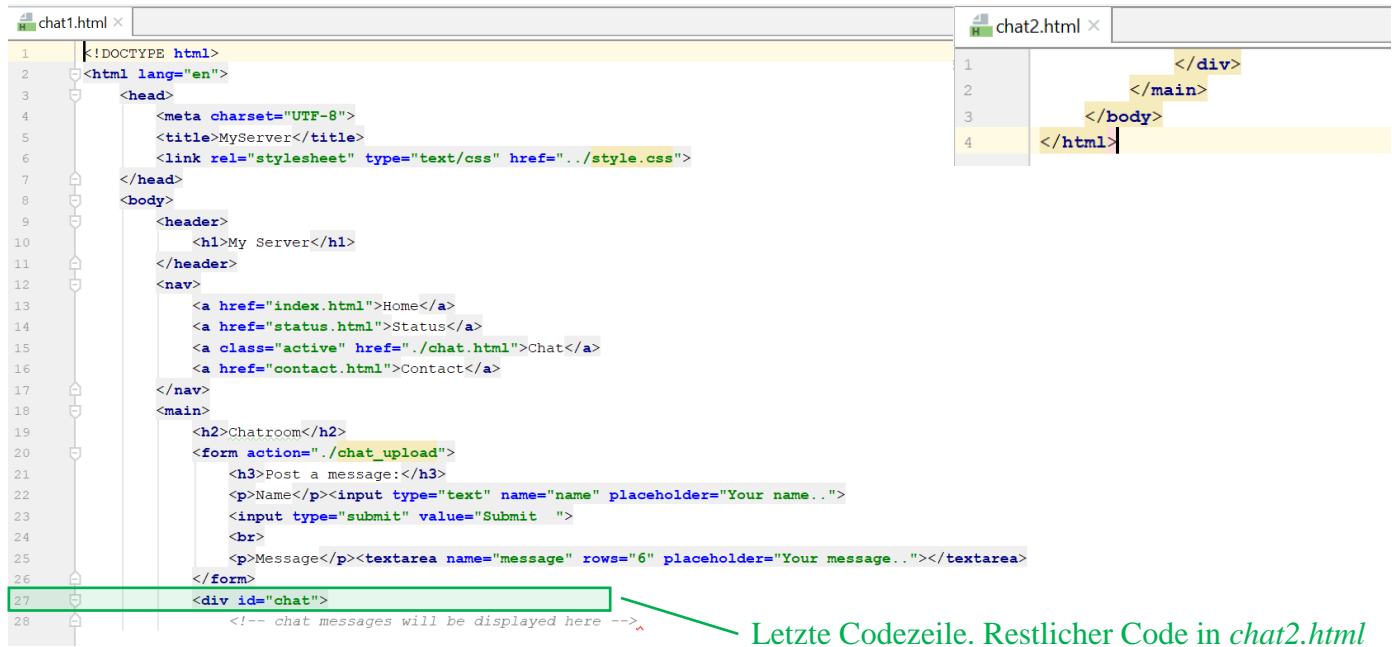
```
26 </form>
27 <div id="chat">
28 <!-- chat messages will be displayed here -->
29 </div>
30 </main>
31 </body>
32 </html>Susi;Hallo!.
33 Max;Hello World!
34
```



Es kommt kein valides HTML Dokument  
beim Client an: die Daten sind einfach ganz  
am Schluss angehängt!

Die Nachrichten sollen eigentlich im `<div id="chat">` Container eingefügt werden. Dazu müssen Sie das HTML-Dokument *chat.html* auf zwei Teile aufteilen:

- *chat1.html* der gesamte HTML-Code vor dem `<div>` Container
- *chat2.html* der restliche HTML-Code nach dem `<div>` Container



```
chat1.html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>MyServer</title>
6 <link rel="stylesheet" type="text/css" href="../style.css">
7 </head>
8 <body>
9 <header>
10 <h1>My Server</h1>
11 </header>
12 <nav>
13 Home
14 Status
15 Chat
16 Contact
17 </nav>
18 <main>
19 <h2>Chatroom</h2>
20 <form action="/chat_upload">
21 <h3>Post a message:</h3>
22 <p>Name</p><input type="text" name="name" placeholder="Your name..">
23 <input type="submit" value="Submit ">
24

25 <p>Message</p><textarea name="message" rows="6" placeholder="Your message.."></textarea>
26 </form>
27 <div id="chat">
28 <!-- chat messages will be displayed here -->

```

```
chat2.html
1 </div>
2 </main>
3 </body>
4 </html>
```

Letzte Codezeile. Restlicher Code in *chat2.html*


Nun müssen Sie nur mehr den Code der JavaScript Datei anpassen, sodass alle 3 Dateien ausgelesen und in der richtigen Reihenfolge zusammengefügt werden:

```
/* load first html code */
let chat_part1 = fs.readFileSync(path: './public/html/chat1.html', options: 'utf8');

/* load all messages */
let chat_part2 = fs.readFileSync(path: './public/data/chat.txt', options: 'utf8');

/* load second html code */
let chat_part3 = fs.readFileSync(path: './public/html/chat2.html', options: 'utf8');

/* send finished code */
res.writeHead(statusCode: 200, headers: { 'Content-Type': 'text/html' });
res.write(chunk: `${chat_part1}${chat_part2}${chat_part3}`);
return res.end();
```

Testen Sie wieder den Code! 

Fast geschafft – die Nachrichten sind zumindest schon an der richtigen Stelle!

```
27 <div id="chat">
28 <!-- chat messages will be displayed here -->Susi;Hallo!..
29 Max;Hello World!
30 </div>
31 </main>
32 </body>
33 </html>
```

Der letzte Part ist, dass die Nachrichten mit etwas HTML Code sauber gegliedert werden, damit man sie im Front-End mit CSS gestalten kann.

Teil 1: Einlesen modifizieren

- `chat_part2` als leeren String anlegen → darin werden dann die formatierten Daten gespeichert
- `data` speichert die ausgelesenen Daten aus der Textdatei

```
/* load files (with sync methods) */
let chat_part1 = fs.readFileSync(path: './public/html/chat1.html', options: 'utf8');
let chat_part2 = ''; /* will be built later */
let data = fs.readFileSync(path: './public/data/chat.txt', options: 'utf8');
let chat_part3 = fs.readFileSync(path: './public/html/chat2.html', options: 'utf8');
```

Teil 2: Daten mit HTML Tags formatieren

- Zuerst in einzelne Zeilen aufteilen mit der `split()` Methode (Trennzeichen: NewLine)
- Danach alle Zeile durchiterieren und jeweils:
  - Die Zeile aufteilen mit der `split()` Methode (Trennzeichen: Strichpunkt)  
→ `elements[0]` enthält den Namen  
→ `elements[1]` enthält die Nachricht
  - Den HTML-Code mit *Template String* zusammenbauen

```
/* build formatted data */
let lines = data.split(separator: '\n');
lines.forEach(callbackfn: function(line) {
 let elements = line.split(separator: ';');
 if (elements[1] !== undefined) chat_part2 += `<div><h3>${elements[0]}</h3><p>${elements[1]}</p></div>`;
});
```

in einzelne Zeilen aufteilen

alle Zeilen durchiterieren

Zur Sicherheit:  
Test, ob Nachricht vorhanden

Testen Sie wieder den Code!



Nun fehlt nur mehr etwas CSS, damit die Nachrichten im typischen Chatlook erschienen::

## CSS-Code für Chat-Nachrichten

```
main #chat div {
 background-color: rgba(0,39,138,0.3);
 padding: 1%;
 border-radius: 0.5vw;
 margin-bottom: 3%;
 width: 60%;
}
main #chat div:nth-child(even) {
 margin-left: 38%;
}
main #chat h3 {
 font-weight: bold;
 font-size: 1.0rem;
 margin: 0 0 2% 0;
}
main #chat p {
 font-size: 1.0rem;
 margin: 0 0 2% 0;
}
```

Zum letzten Mal: testen Sie wieder den Code!



Gratulation! Ihr Chat Server ist einsatzbereit!