

## 3. Understanding the Basics

### \* Chapter 23: Module Introduction





#### What's In This Module?

How Does The Web Work (Refresher)?

Creating a Node.js Server

Using Node Core Modules

Working with Requests & Responses  
(Basics)

Asynchronous Code & The Event Loop

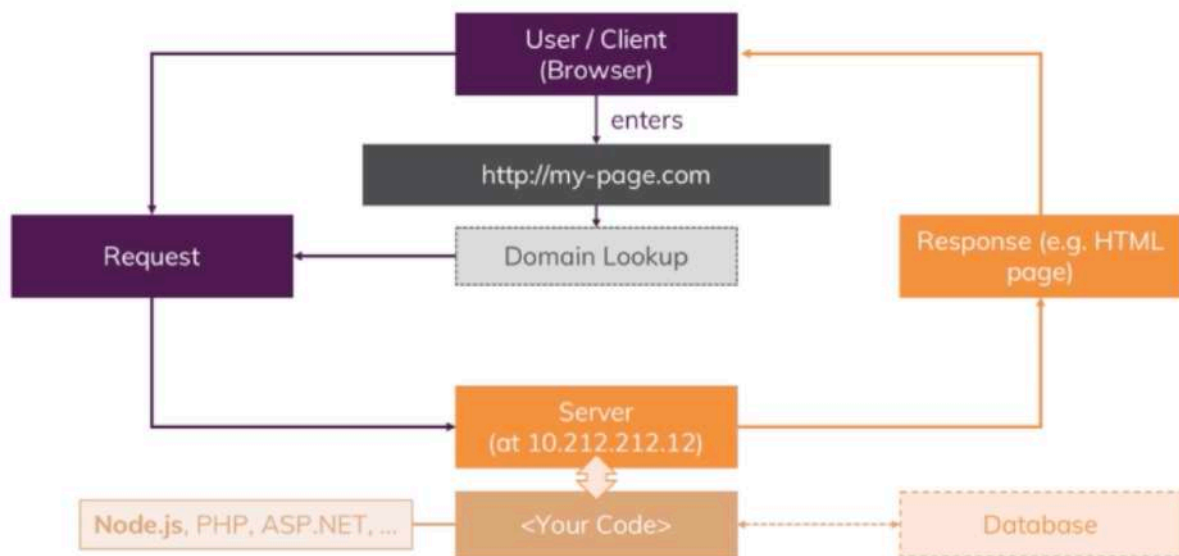
By Simon

### \* Chapter 24: How The Web Works





## How the Web Works



## HTTP, HTTPS

### Hyper Text Transfer Protocol

A Protocol for Transferring Data which is understood by Browser and Server

### Hyper Text Transfer Protocol Secure

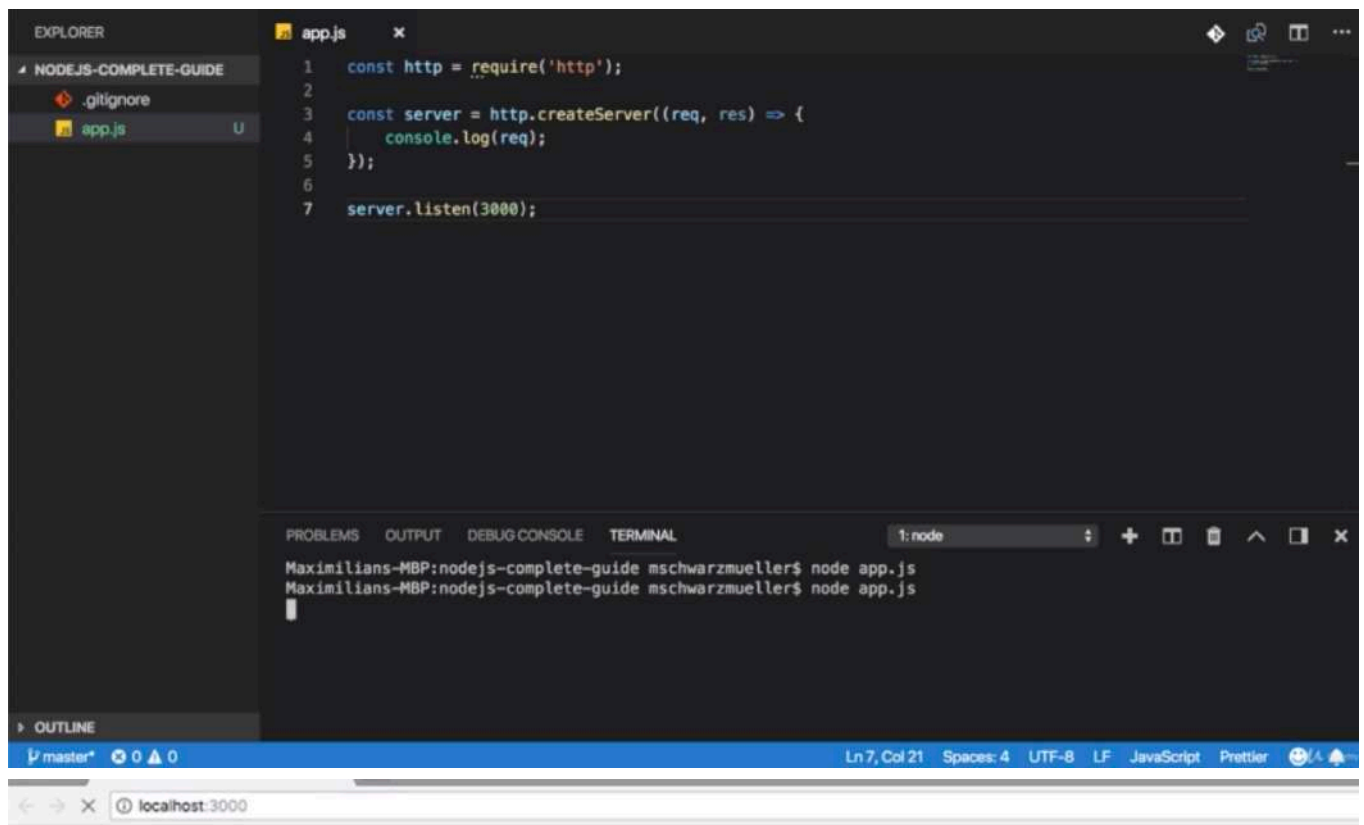
HTTP + Data Encryption (during Transmission)

## \* Chapter 25: Creating a Node Server

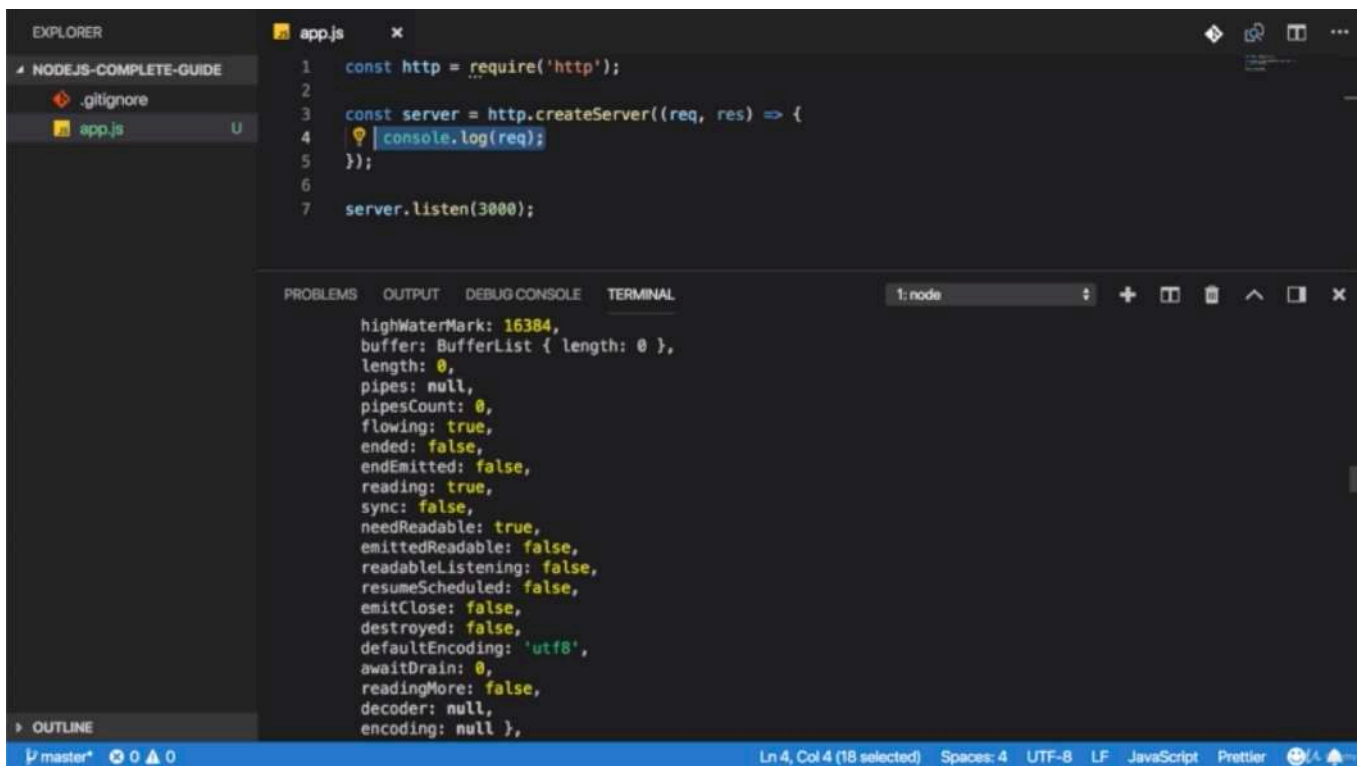








Search Google or type URL



The screenshot shows a VS Code editor with a file named `app.js` open. The code in `app.js` is as follows:

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   console.log(req);
5 });
6
7 server.listen(3000);
```

The terminal window at the bottom shows the output of the server, which is a detailed object representing the incoming request:

```
highWaterMark: 16384,
buffer: BufferList { length: 0 },
length: 0,
pipes: null,
pipesCount: 0,
flowing: true,
ended: false,
endEmitted: false,
reading: true,
sync: false,
needReadable: true,
emittedReadable: false,
readableListening: false,
resumeScheduled: false,
emitClose: false,
destroyed: false,
defaultEncoding: 'utf8',
awaitDrain: 0,
readingMore: false,
decoder: null,
encoding: null },
```

- in browser, nothing happens but if you go back to terminal, you will see a lot of output there and that is your request being logged to the console.

```
1 //app.js
2
3 //how to creating a node server 1.
4
5 const http = require('http')
6
7 function rqListener(req, res){
8
9 }
10
11 /**this is like saying that "Look for 'rqListener' function with this name
12 * and execute it for every incoming request" */
13 http.createServer(rqListener)
14
15 //how to creating a node server 2 (using anonymous function)
16
17 const http = require('http')
18 http.createServer(function(req, res){})
19
20 //how to creating a node server 3 (using arrow function)
21
22 const http = require('http')
23 http.createServer((req, res) => {})
```

```
1 //app.js
2
3 /**this is the way you import file in node.js
4 * you can import your own javascript files
5 * or if you don't have a path to one of your files,
6 * you can also import a core module like http
7 *
8 * a path to one of your files always has to start with ./(relative path) or /(absolute
path)
```

```

9  * and it's up to you to use '.js' or not
10 */
11 const http = require('http')
12
13 const server = http.createServer((req, res) => {
14     console.log(req)
15 })
16
17 /**'listen()' start a process where node.js will not immediately exit our script
18 * but where node.js will instead keep this running to listen for incoming request.
19 *
20 * default hostname will be the name of the machine this is running on,
21 * so for our local machine, this is localhost by default
22 */
23 server.listen(3000)

```

## \* Chapter 26: The Node Lifecycle & Event Loop









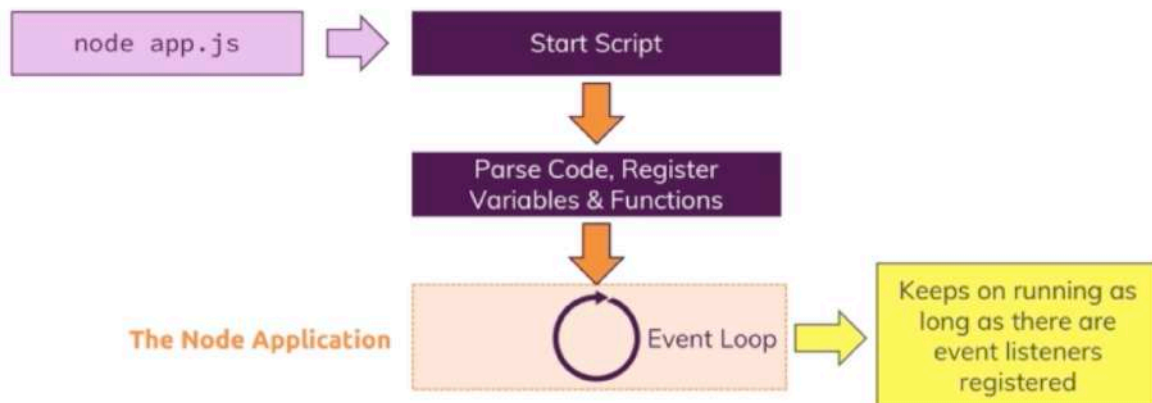








### Node.js Program Lifecycle



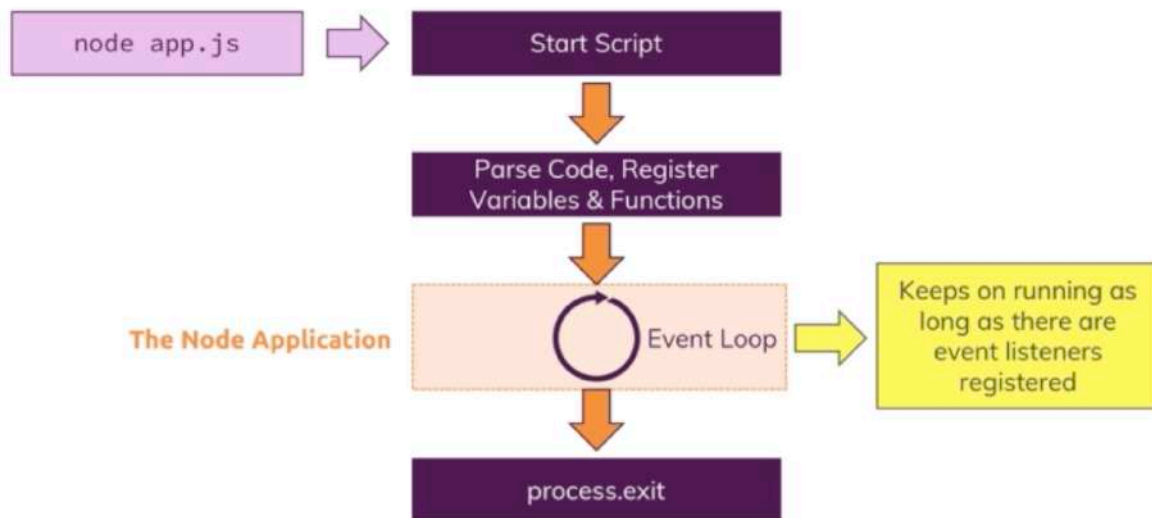
The screenshot shows a Visual Studio Code editor with a file explorer on the left containing 'NODEJS-COMPLETE-GUIDE', '.gitignore', and 'app.js'. The main editor displays 'app.js' with the following code:

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   console.log(req);
5 });
6
7 server.listen(3000);
```

The terminal at the bottom shows the command 'node app.js' being executed, with the output 'Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$'.



## Node.js Program Lifecycle



EXPLORER

app.js x

NODEJS-COMPLETE-GUIDE

.gitignore

app.js U

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   console.log(req);
5   process.exit();
6 });
7
8 server.listen(3000);
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

1: node

Maximilians-MBP:nodejs-complete-guide mschwarzmuellers\$ node app.js

OUTLINE

master 0 0

Ln 6, Col 2 (59 selected) Spaces: 4 UTF-8 LF JavaScript Prettier





```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   console.log(req);
5   process.exit();
6 });
7
8 server.listen(3000);
```

```
{
  _repeat: null,
  _destroyed: false,
  [Symbol(unref)]: true,
  [Symbol(asyncId)]: 8,
  [Symbol(triggerId)]: 7,
  [Symbol(kBytesRead)]: 0,
  [Symbol(kBytesWritten)]: 0,
  _consuming: false,
  _dumped: false
}
```

```
1 //app.js
2
3 const http = require('http');
4
5 const server = http.createServer((req, res) => {
6   console.log(req);
7   //process.exit()
8 })
9
10 server.listen(3000)
```

## \* Chapter 27: Controlling The Node.js Process



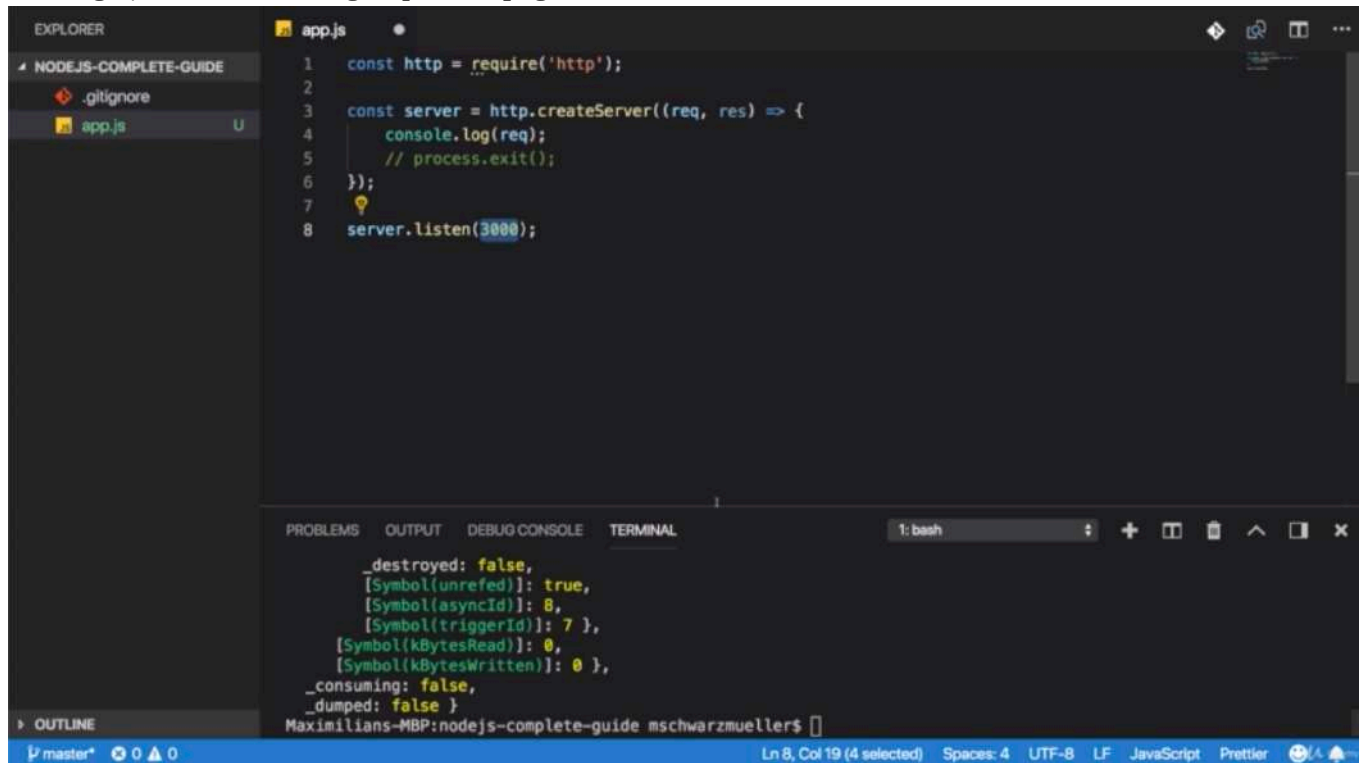
- Do you wanna quit your running Node.js Server
- You can always do that by pressing CTRL + C in the terminal / command prompty window where you started your server (i.e. where you ran node app.js)

## \* Chapter 28: Understanding Requests







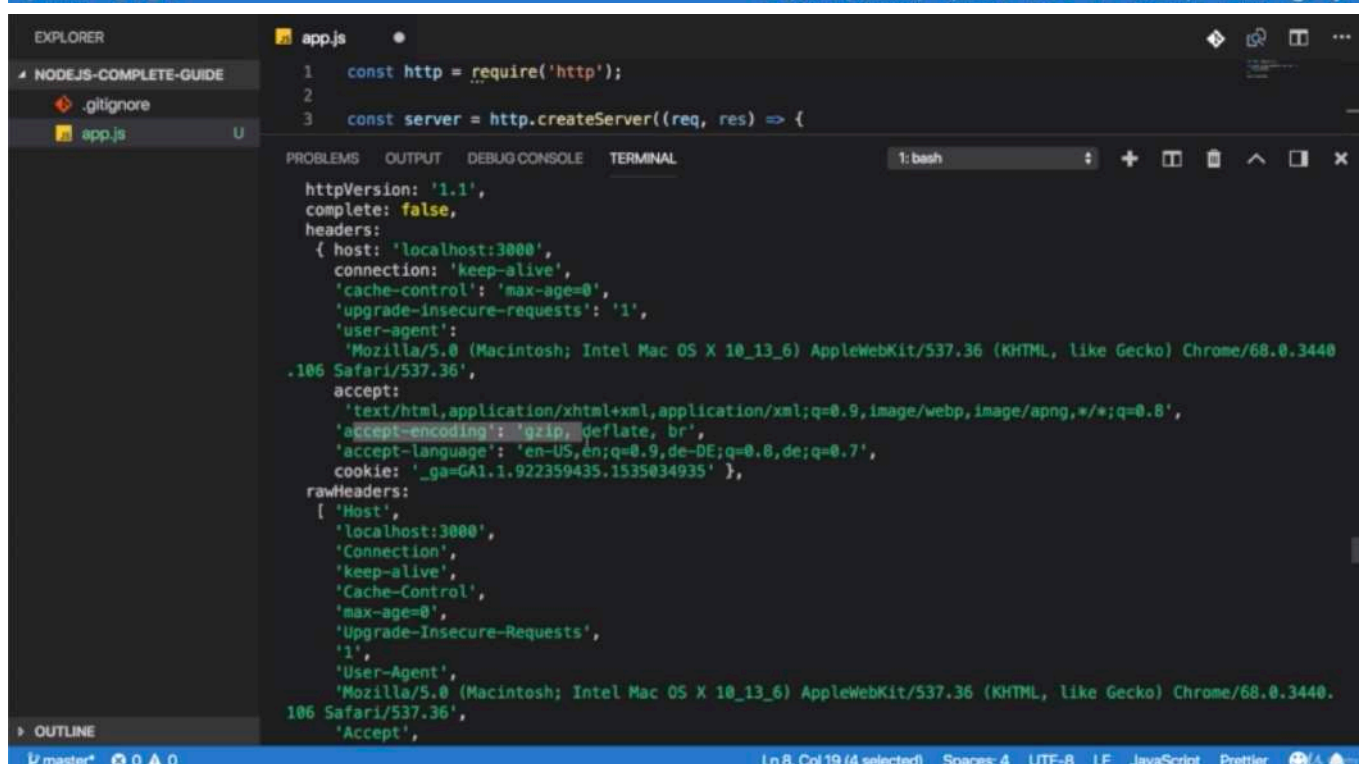


The screenshot shows the VS Code editor with the Explorer sidebar on the left displaying the file structure: NODEJS-COMPLETE-GUIDE, .gitignore, and app.js. The main editor area shows the app.js file with the following code:

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   console.log(req);
5   // process.exit();
6 });
7
8 server.listen(3000);
```

The bottom panel shows the TERMINAL tab with the following output:

```
_destroyed: false,
[Symbol(unref)]: true,
[Symbol(asyncId)]: 8,
[Symbol(triggerId)]: 7 },
[Symbol(kBytesRead)]: 0,
[Symbol(kBytesWritten)]: 0 },
_consuming: false,
_dumped: false }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



The screenshot shows the VS Code editor with the Explorer sidebar on the left displaying the file structure: NODEJS-COMPLETE-GUIDE, .gitignore, and app.js. The main editor area shows the app.js file with the following code:

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
```

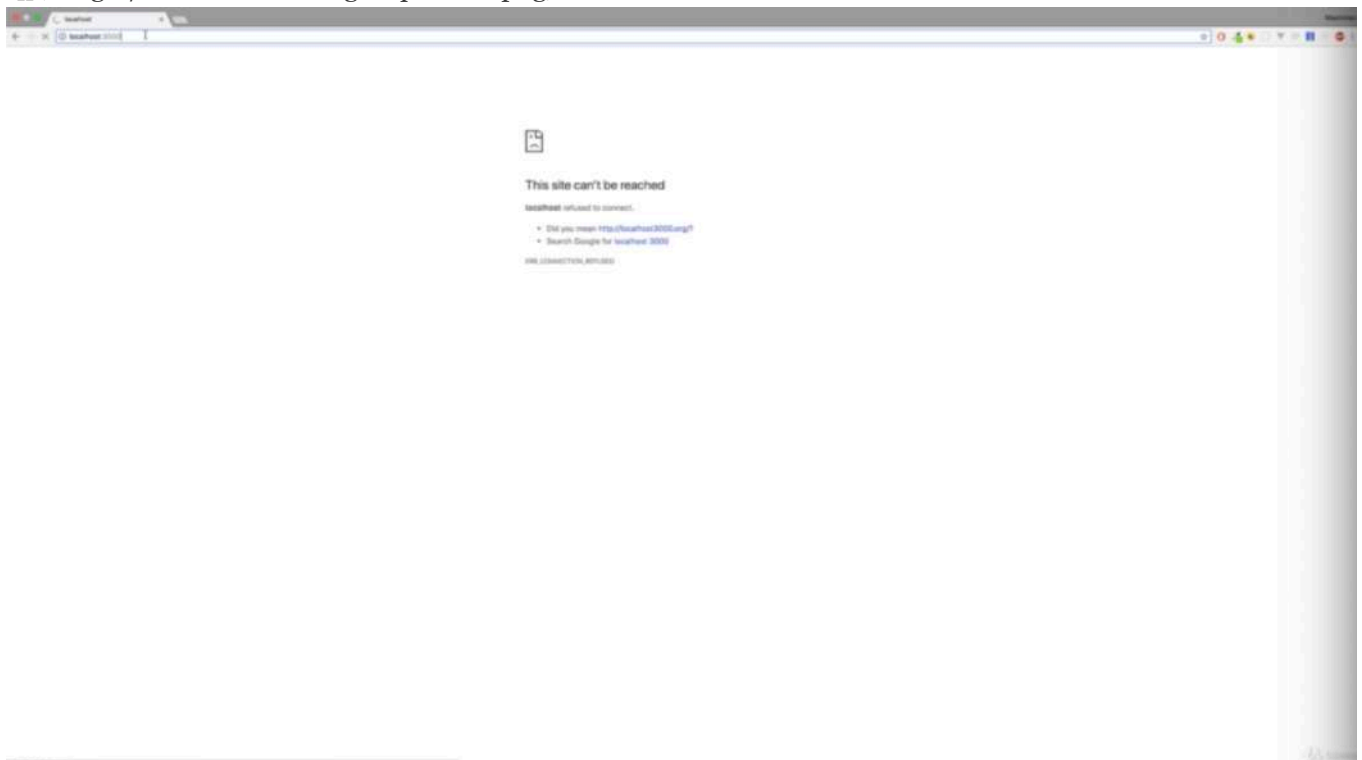
The bottom panel shows the TERMINAL tab with the following output:

```
httpVersion: '1.1',
complete: false,
headers:
  { host: 'localhost:3000',
    connection: 'keep-alive',
    'cache-control': 'max-age=0',
    'upgrade-insecure-requests': '1',
    'user-agent':
      'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440
.106 Safari/537.36',
    accept:
      'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
    'accept-encoding': 'gzip, deflate, br',
    'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
    cookie: '_ga=GA1.1.922359435.1535034935' },
rawHeaders:
  [ 'Host',
    'localhost:3000',
    'Connection',
    'keep-alive',
    'Cache-Control',
    'max-age=0',
    'Upgrade-Insecure-Requests',
    '1',
    'User-Agent',
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440
.106 Safari/537.36',
    'Accept',
```

```
2 const server = http.createServer((req, res) => {
3   console.log(req.url, req.method, req.headers);
4   // process.exit();
5 });
6
7
8 server.listen(3000);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmuellers$ node app.js
/ GET { host: 'localhost:3000',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'upgrade-insecure-requests': '1',
  'user-agent':
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36',
  accept:
    'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
  cookie: '_ga=GA1.1.922359435.1535034935' }
/ GET { host: 'localhost:3000',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'upgrade-insecure-requests': '1',
  'user-agent':
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36',
  accept:
    'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
  'accept-encoding': 'gzip, deflate, br',
```

- now the URL has changed. we still have all the header stuff, because we are outputting request headers but prior to that, we output the method which you see here
  - it's GET and you see the URL and the URL is just the slash / because URL is basically everything after our host and we just have localhost and that basically translate to localhost/
- 
- 
- 





```
2
3 const server = http.createServer((req, res) => {
4   console.log(req.url, req.method, req.headers);
5   // process.exit();
6 });
7
8 server.listen(3000);
```

```
'upgrade-insecure-requests': '1',
'user-agent':
'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.10
6 Safari/537.36',
accept:
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
'accept-encoding': 'gzip, deflate, br',
'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
cookie: '_ga=GA1.1.922359435.1535034935' }
^[[A^C
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node app.js
/test GET { host: 'localhost:3000',
connection: 'keep-alive',
'upgrade-insecure-requests': '1',
'user-agent':
'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.10
6 Safari/537.36',
accept:
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
'accept-encoding': 'gzip, deflate, br',
'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
cookie: '_ga=GA1.1.922359435.1535034935' }
```

- if i had /test, we see another output and there we see another output and there we see /test being logged here and then also get for the method and our headers.
- this is basically how we can access some information about our request.

## \* Chapter 29: Sending Responses







The image shows a VS Code editor window with a file named `app.js` open. The code is a simple HTTP server using `http.createServer()`. It logs the request details, sets the `Content-Type` header to `text/html`, and writes an HTML response with a title and body. The server listens on port 3000. The terminal output shows the server running and receiving a request from a Chrome browser. The request headers include `user-agent`, `accept`, `accept-encoding`, `accept-language`, and `cookie`.

```
2
3 const server = http.createServer((req, res) => {
4   console.log(req.url, req.method, req.headers);
5   // process.exit();
6   res.setHeader('Content-Type', 'text/html');
7   res.write('<html>');
8   res.write('<head><title>My First Page</title><head>');
9   res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
10  res.write('</html>');
11  res.end();
12 });
13
14 server.listen(3000);
15
```

```
'user-agent':
'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.10
6 Safari/537.36',
accept:
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
'accept-encoding': 'gzip, deflate, br',
'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
cookie: '_ga=GA1.1.922359435.1535034935' }
```

Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

Hello from my Node.js Server!



What about the other headers you see there? These are basically default headers which are set by the server.

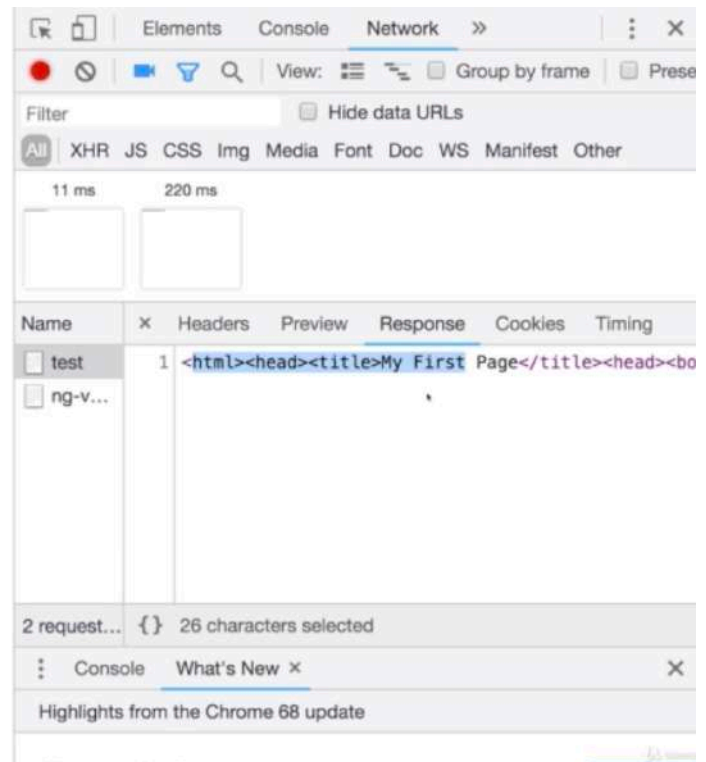
The image shows the Chrome DevTools Network tab. A request is selected, and the 'Headers' sub-tab is active. The 'Response Headers' section shows the following headers: `Connection: keep-alive`, `Content-Type: text/html`, `Date: Mon, 27 Aug 2018 11:54:17 GMT`, and `Transfer-Encoding: chunked`. The 'Request Headers' section shows the `Accept` header: `text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8`.

Name	Value
test	
ng-v...	
2 request...	

Name	Value
Connection	keep-alive
Content-Type	text/html
Date	Mon, 27 Aug 2018 11:54:17 GMT
Transfer-Encoding	chunked

Name	Value
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8

Hello from my Node.js Server!



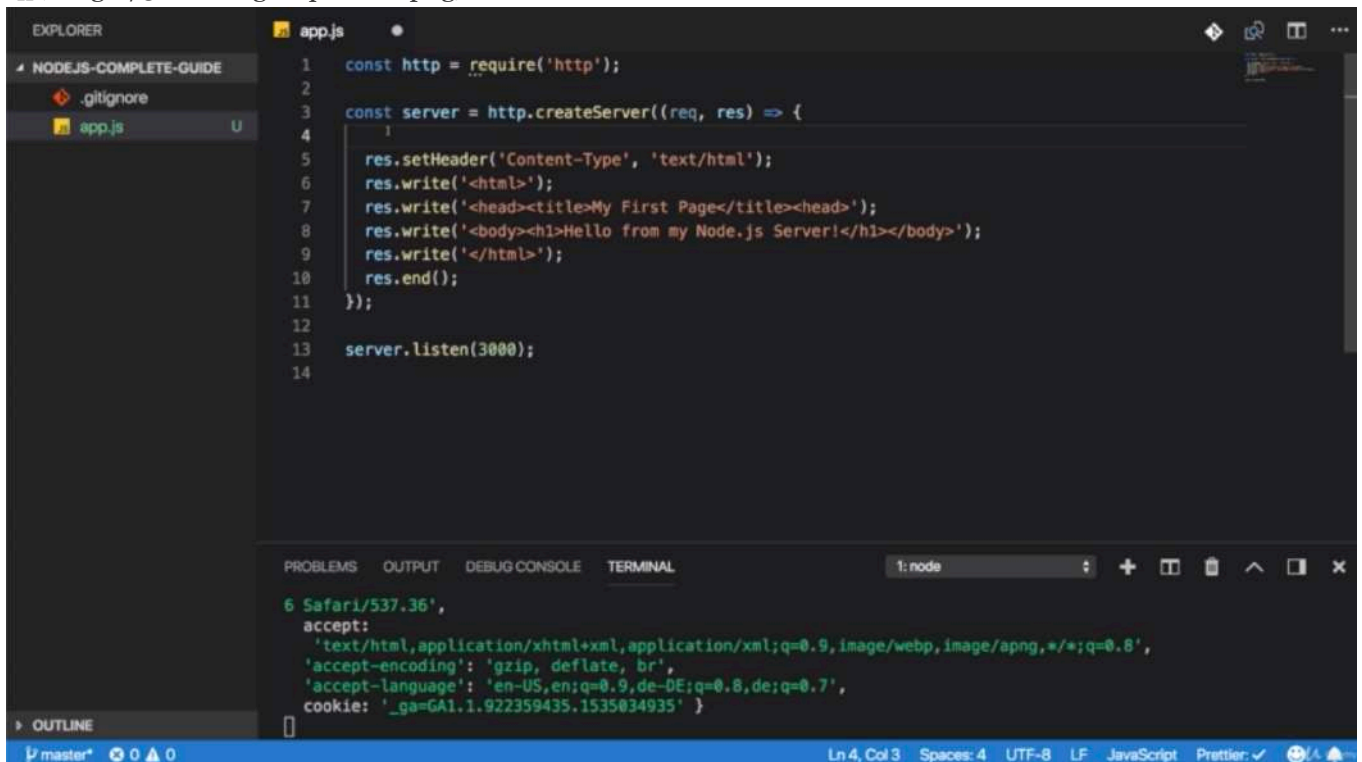
```
1 //app.js
2
3 const http = require('http');
4
5 const server = http.createServer((req, res) => {
6   console.log(req.url, req.method, req.headers);
7   //process.exit()
8   /**'Content-Type' is default header which the browser knows
9    * and understand and accepts
10   * and in setHeader, we set the value for this header key in 2nd argument
11   * now 'text/html' will attach a header to our response where we pass some meta
information
12   * saying that the type of the content which will also be part of the response is html
13   */
14   res.setHeader('Content-Type', 'text/html')
15   /**'write' allows us to write some data to the response
16   * and works in chunks
17   */
18   res.write('<html>')
19   res.write('<head><title>My First Page</title></head>')
20   res.write('<body><h1>Hello from my Node.js SErver!</h1></body>')
21   res.write('</html>')
22   /**once we are done with creating that response
23   * and we do this by calling end()
24   * after end(), we can write nothing
25   * because end() is the part where we will send it back to the client
26   */
27   res.end()
28 })
29
30 server.listen(3000)
```

## \* Chapter 31: Routing Requests



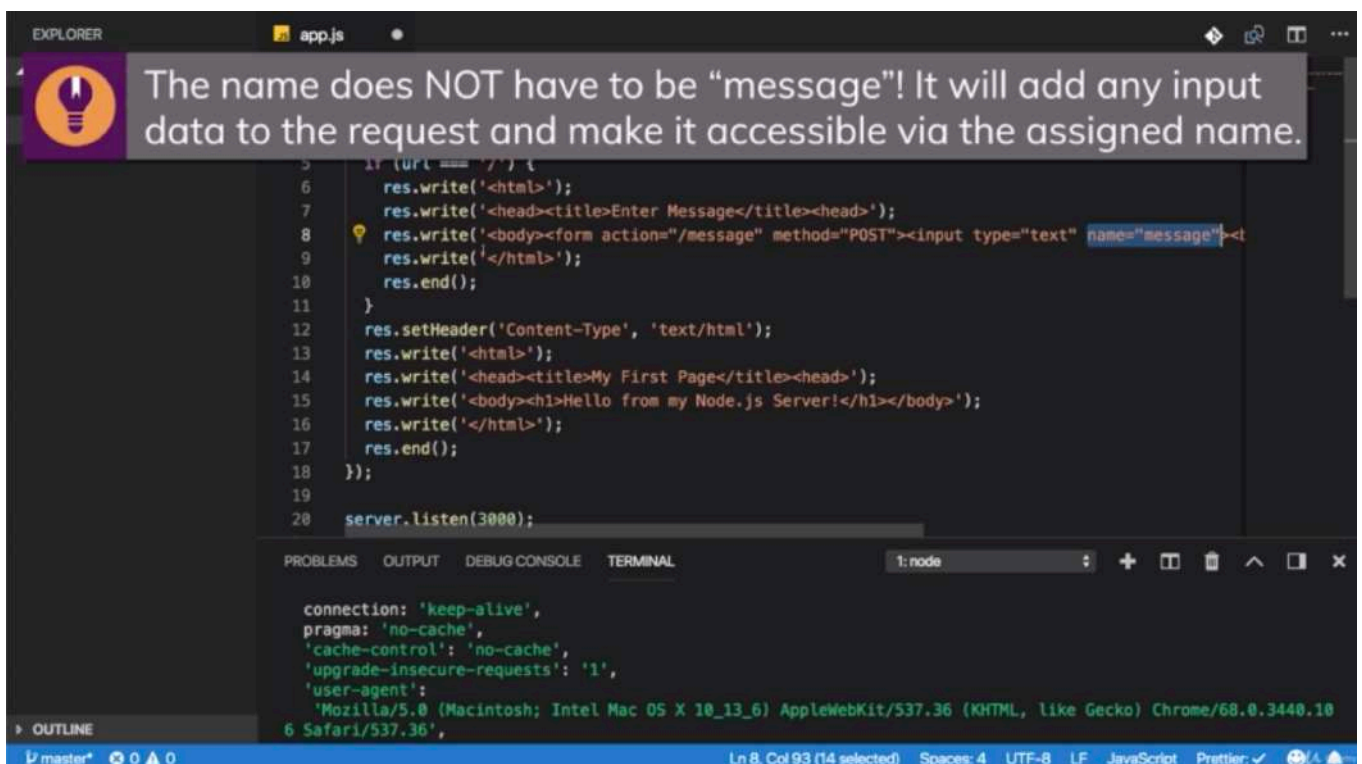






```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   1
5   res.setHeader('Content-Type', 'text/html');
6   res.write('<html>');
7   res.write('<head><title>My First Page</title></head>');
8   res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
9   res.write('</html>');
10  res.end();
11 });
12
13 server.listen(3000);
14
```

```
6 Safari/537.36',
  accept:
    'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
  cookie: '_ga=GA1.1.922359435.1535034935' }
```



The name does NOT have to be "message"! It will add any input data to the request and make it accessible via the assigned name.

```
5 if (url === '/') {
6   res.write('<html>');
7   res.write('<head><title>Enter Message</title></head>');
8   res.write('<body><form action="/message" method="POST"><input type="text" name="message"></form>');
9   res.write('</html>');
10  res.end();
11 }
12 res.setHeader('Content-Type', 'text/html');
13 res.write('<html>');
14 res.write('<head><title>My First Page</title></head>');
15 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
16 res.write('</html>');
17 res.end();
18 });
19
20 server.listen(3000);
```

```
connection: 'keep-alive',
pragma: 'no-cache',
'cache-control': 'no-cache',
'upgrade-insecure-requests': '1',
'user-agent':
  'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.10
6 Safari/537.36',
```

- let's say localhost:3000/, then we wanna load a page where the user can enter some data which we store in a file on the server once it is sent.

- we can do this by first of all parsing the URL. i'm storing it in a new constant and i do this by accessing request URL which was something like / or /test which what we entered.













EXPLORER

app.js

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   const url = req.url;
5   if (url === '/') {
6     res.write('<html>');
7     res.write('<head><title>Enter Message</title><head>');
8     res.write('<body><form action="/message" method="POST"><input type="text" name="message"><');
9     res.write('</html>');
10    return res.end();
11  }
12  res.setHeader('Content-Type', 'text/html');
13  res.write('<html>');
14  res.write('<head><title>My First Page</title><head>');
15  res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
16  res.write('</html>');
17  res.end();
18 });
19
20 server.listen(3000);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

1: node

```
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
'accept-encoding': 'gzip, deflate, br',
'accept-language': 'en-US,en;q=0.9,de-DE;q=0.8,de;q=0.7',
cookie: '_ga=GA1.1.922359435.1535034935' }
^C
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node app.js
```

master\* 0 0 0

Ln 10, Col 22 Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

Chrome DevTools Network tab showing a request to 'My First Page'.

Filter: Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other

11 ms 220 ms

Name	X	Headers	Preview	Response	Cookies	Timing
test	1			<html><head><title>My First Page</title><head><bo		
ng-v...						

2 request... {} 8 characters selected

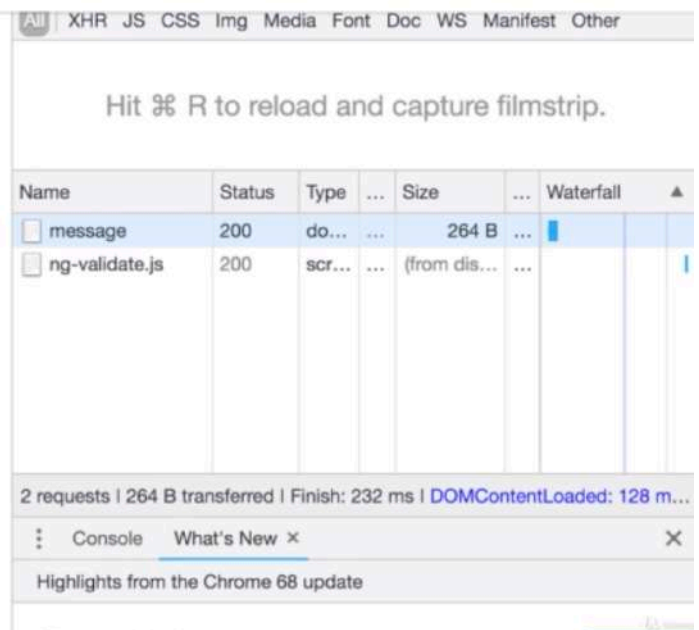
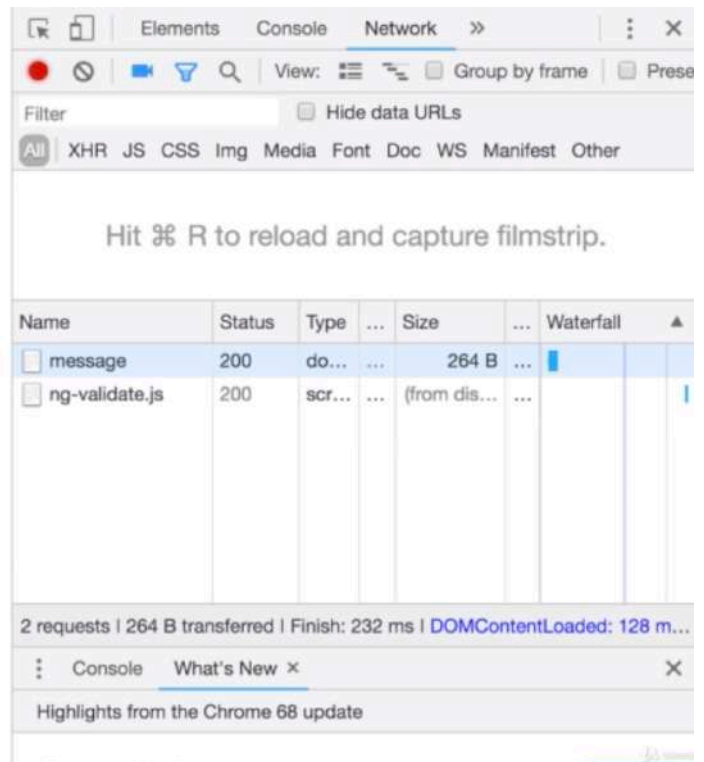
Console What's New x

Highlights from the Chrome 68 update





Hello from my Node.js Server!



```
1 //app.js
2
3 const http = require('http');
4
5 const server = http.createServer((req, res) => {
6   const url = req.url
7   if(url === '/'){
8     res.write('<html>')
9     res.write('<headd><title>My First Page</title></headd>')
10    /**action is the url this request which will be generated automatically should be
    sent to
11    * and i will use '/message' here
12    * and this will automatically target the host it's running on. so localhost:3000
13    * and http 'method' should be used and there we previously saw
```

```

14      * if we expand this, we get a GET request which is the default if we enter the URL
15      *
16      * 'POST' request has to be set up by you by creating such a form
17      * 'GET' request is automatically sent when you click a link or enter the URL
18      *
19      * 'message' will automatically put that message into the request
20      * so when we visit localhost:3000/ , we will return a response where we render this

```

#### HTML code

```

21      */
22      res.write(
23        '<body><form action="/message" method="POST"><input type="text" name="message">
<button type="submit">Send</button></form></body>
24      '
25      )
26
27      res.write('</html>')
28      /**this is not required to return the response
29      * but to return from this anonymous function and to not continue this code below
30      */
31      return res.end()
32    }
33    res.setHeader('Content-Type', 'text/html')
34    res.write('<html>')
35    res.write('<headd><title>My First Page</title></headd>')
36    res.write('<body><h1>Hello from my Node.js SServer!</h1></body>')
37    res.write('</html>')
38    res.end()
39  })
40
41  server.listen(3000)

```

```

1  //clean version
2
3  //app.js
4
5  const http = require('http');
6  const fs = require('fs');
7
8  const server = http.createServer((req, res) => {
9    const url = req.url;
10    const method = req.method;
11    if (url === '/') {
12      res.write('<html>');
13      res.write('<head><title>Enter Message</title><head>');
14      res.write('<body><form action="/message" method="POST"><input type="text"
name="message"><button type="submit">Send</button></form></body>');
15      res.write('</html>');
16      return res.end();
17    }
18    if (url === '/message' && method === 'POST') {
19      fs.writeFileSync('message.txt', 'DUMMY');
20      res.statusCode = 302;
21      res.setHeader('Location', '/');
22      return res.end();
23    }
24    res.setHeader('Content-Type', 'text/html');
25    res.write('<html>');

```

```

26 res.write('<head><title>My First Page</title></head>');
27 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
28 res.write('</html>');
29 res.end();
30 });
31
32 server.listen(3000);

```

## \* Chapter 32: Redirecting Requests







The screenshot shows a Visual Studio Code editor with a file explorer on the left containing 'app.js' and 'message.txt'. The main editor displays the code for 'app.js', which includes a GET endpoint for '/' and a POST endpoint for '/message'. The POST endpoint writes 'DUMMY' to 'message.txt' and redirects the user to the root path. The terminal at the bottom shows the command 'node app.js' being executed.

```

app.js
1  const url = req.url;
2  const method = req.method;
3
4  if (url === '/') {
5
6    res.write('<html>');
7    res.write('<head><title>Enter Message</title></head>');
8    res.write('<body><form action="/message" method="POST"><input type="text" name="message"></body>');
9    res.write('</html>');
10   return res.end();
11 }
12
13 if (url === '/message' && method === 'POST') {
14   fs.writeFileSync('message.txt', 'DUMMY');
15   res.statusCode = 302;
16   res.setHeader('Location', '/');
17   return res.end();
18 }
19
20 res.setHeader('Content-Type', 'text/html');
21 res.write('<html>');
22 res.write('<head><title>My First Page</title></head>');
23 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
24 res.write('</html>');
25 res.end();

```

Terminal Output:

```

Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node app.js

```

dashits

The top screenshot shows the Network tab with 4 requests. The bottom screenshot shows the Network tab with 3 requests and a message 'Hit ⌘ R to reload and capture filmstrip.'

Name	Status	Type	Size	Waterfall
localhost	200	do...	315 B	
ng-validate.js	200	scr...	(from dis...)	
message	302	x-...	115 B	
localhost	200	do...	108 B	

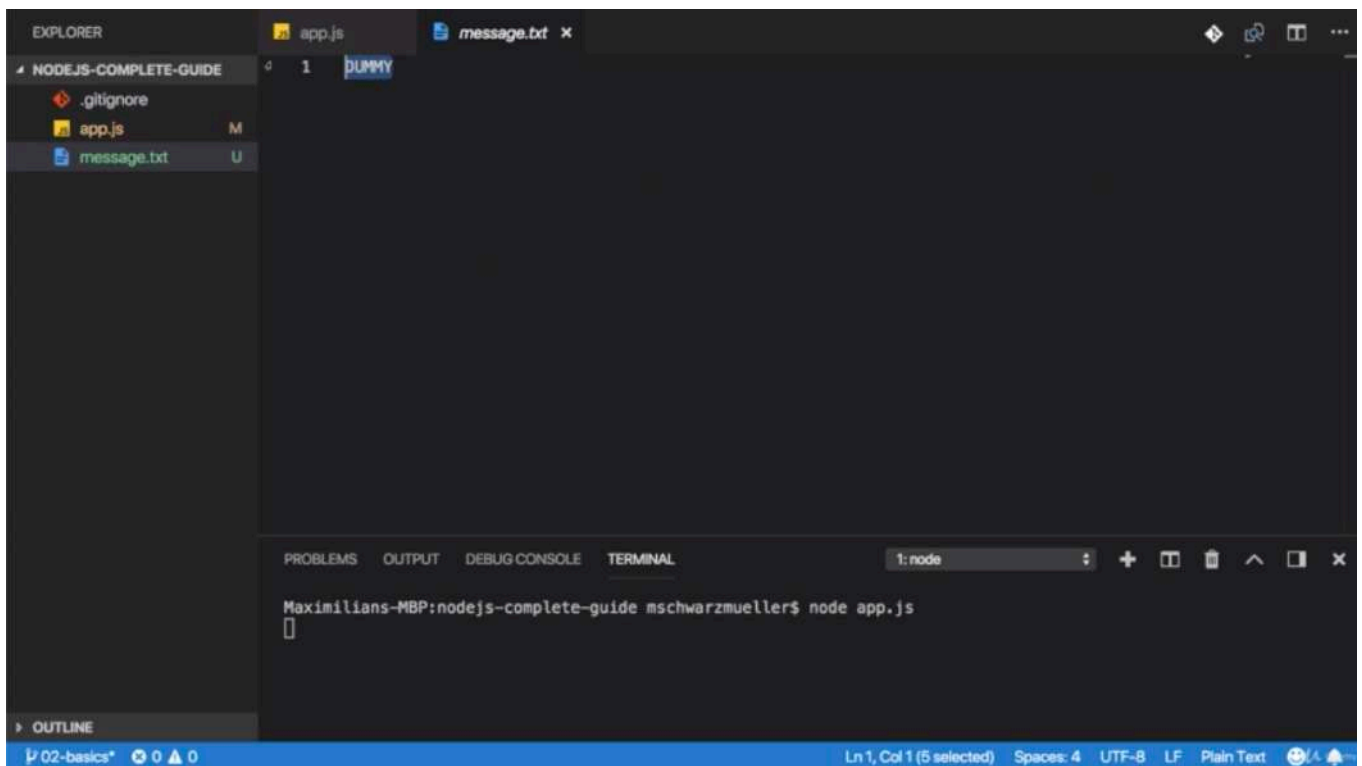
4 requests | 538 B transferred | Finish: -0 ms | Load: -3494 ms

Name	Status	Type	Size	Waterfall
message	302	x-...	115 B	
localhost	200	do...	315 B	
ng-validate.js	200	scr...	(from dis...)	

3 requests | 430 B transferred | Finish: 193 ms | DOMContentLoaded: 99 ms ...

- after filling input field and send any value, then you should simply reload that in the end because you get redirected but you can see that redirect in the network tab of the devtool in chrome, but status 302 indicates we send a request to message

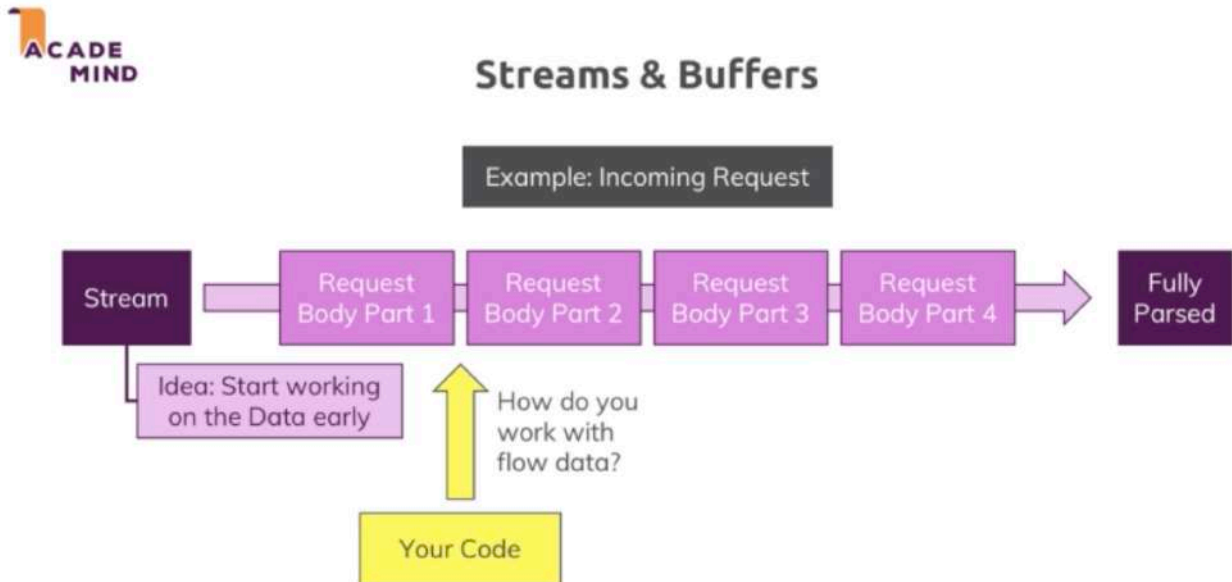




```
1 //app.js
2
3 const http = require('http');
4 const fs = require('fs');
5
6 const server = http.createServer((req, res) => {
7   const url = req.url;
8   const method = req.method;
9   if (url === '/') {
10     res.write('<html>');
11     res.write('<head><title>Enter Message</title><head>');
12     res.write('<body><form action="/message" method="POST"><input type="text"
name="message"><button type="submit">Send</button></form></body>');
13     res.write('</html>');
14     return res.end();
15   }
16   if (url === '/message' && method === 'POST') {
17     fs.writeFileSync('message.txt', 'DUMMY');
18     res.statusCode = 302;
19     res.setHeader('Location', '/');
20     return res.end();
21   }
22   res.setHeader('Content-Type', 'text/html');
23   res.write('<html>');
24   res.write('<head><title>My First Page</title><head>');
25   res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
26   res.write('</html>');
27   res.end();
28 });
29
30 server.listen(3000);
```

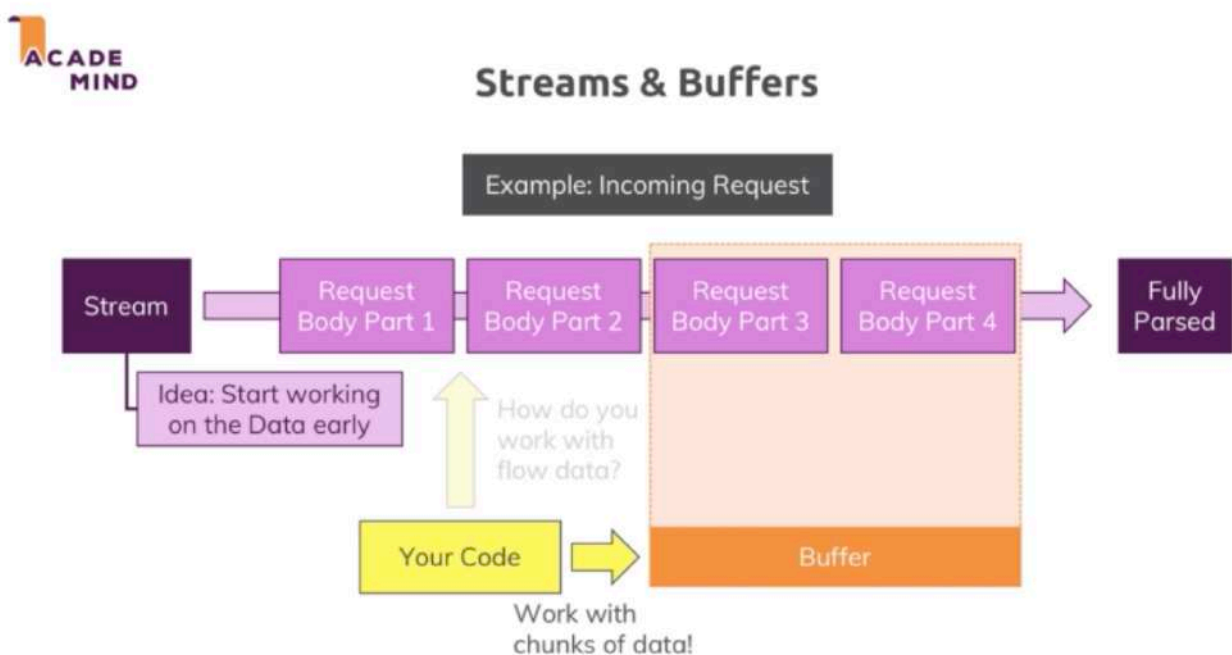
## \* Chapter 33: Parsing Request Bodies





- Our stream is an ongoing process. the request is simply read by node in chunks. in multiple parts and in the end at some point of time it's done. this is done so that we theretically can start working on this, on the individual chunks without having to wait for the full request being read. now for a simple request like the one we are working with, this is not really required. we only get one input field data. it doesn't take so long to parse that.
- but consider a file being uploaded. this will considerably longer and therefore streaming that data could make sense because it could allow you to start writing this to your disk. so to your hard drive where your app runs, your node app runs on your server while that data is coming in, so that you don't have to parse the entire file which is taking sometime and you have to wait for it being fully uploaded before you can do anything with it.
- this is how node handles all requests because it doesn't know it advance how complex and big they are.





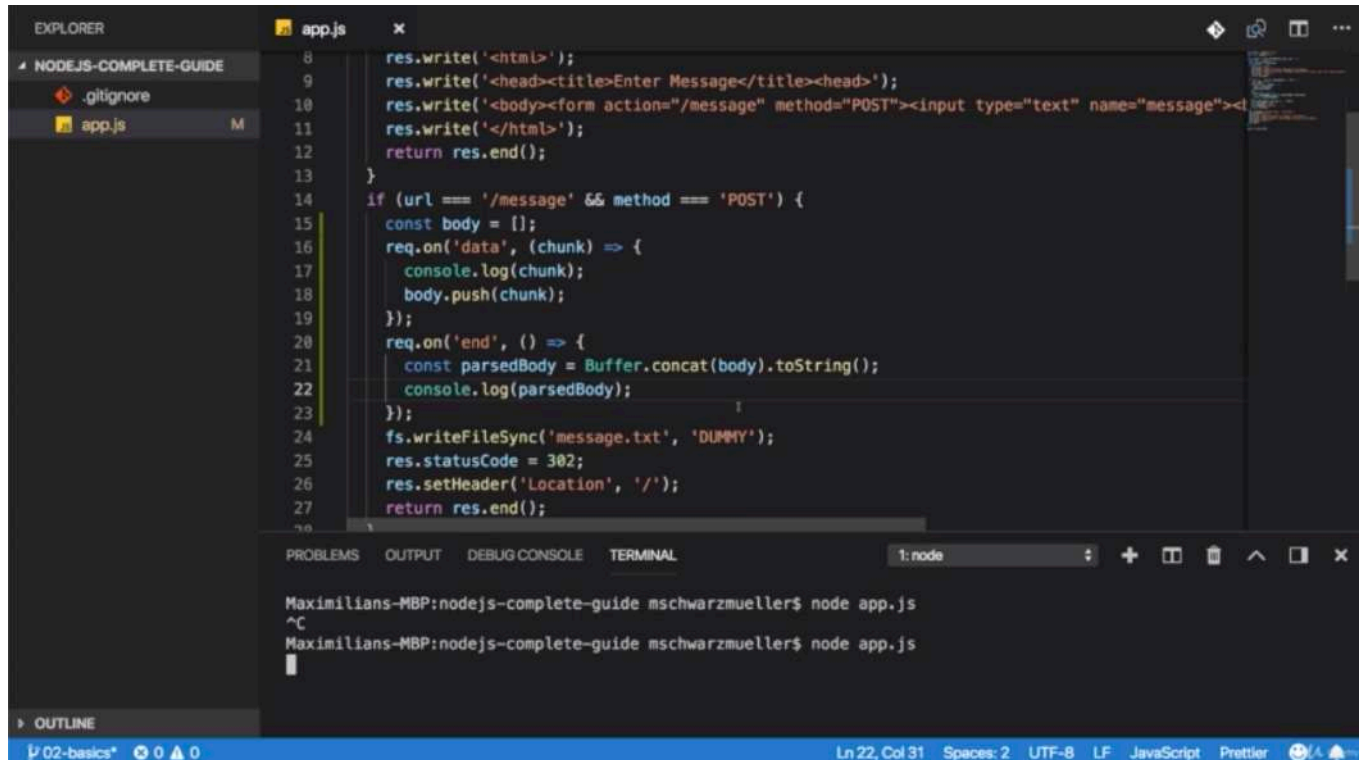
- the problem is with your code, you can't arbitrarily try to work with these chunks. instead to organize these incoming chunks, you use a so-called buffer. buffer is like a bus stop. if you consider buses, they're always driving but for users or customers being able to work with them, to climb on the bus and leave the bus, you need bus stops where you can track the bus and this is what a buffer is.

- buffer is a construct which allows you to hold multiple chunks and work with them before they are released once you are done and you work with the buffer.







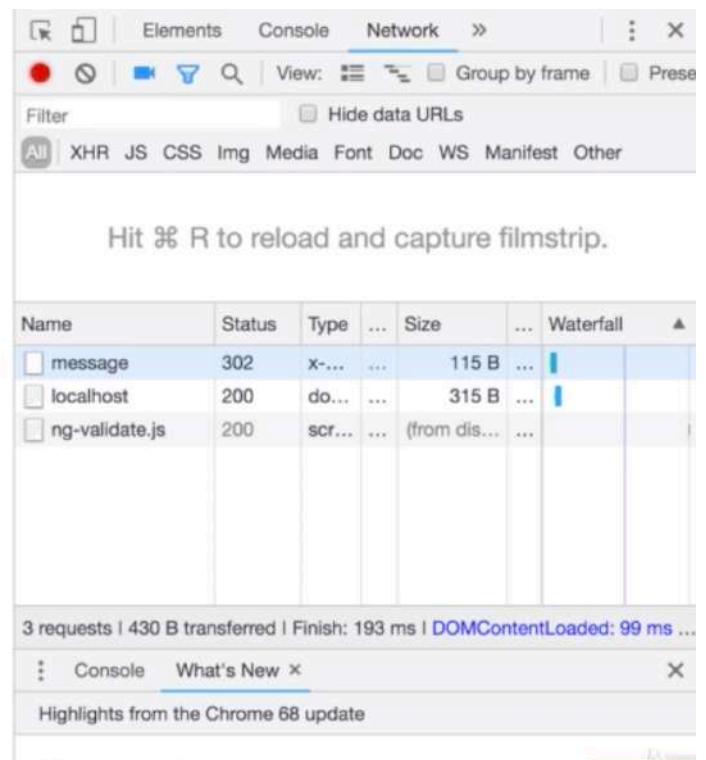


The screenshot shows a VS Code editor with a file named `app.js` open. The code is a Node.js HTTP server that listens for POST requests to `/message`. It uses `req.on('data', ...)` to receive chunks of the request body and `req.on('end', ...)` to process the complete body using `Buffer.concat`. The terminal shows the command `node app.js` being executed twice.

```
8 res.write('<html>');
9 res.write('<head><title>Enter Message</title><head>');
10 res.write('<body><form action="/message" method="POST"><input type="text" name="message"></form></body></html>');
11 res.write('</html>');
12 return res.end();
13 }
14 if (url === '/message' && method === 'POST') {
15   const body = [];
16   req.on('data', (chunk) => {
17     console.log(chunk);
18     body.push(chunk);
19   });
20   req.on('end', () => {
21     const parsedBody = Buffer.concat(body).toString();
22     console.log(parsedBody);
23   });
24   fs.writeFileSync('message.txt', 'DUMMY');
25   res.statusCode = 302;
26   res.setHeader('Location', '/');
27   return res.end();
28 }
```

Terminal output:

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node app.js
^C
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node app.js
```



The screenshot shows the Chrome DevTools Network tab. It displays a list of requests, including a POST request to `message` with a status of 302. The table below summarizes the requests shown.

Name	Status	Type	Size	Waterfall
message	302	x-...	115 B	
localhost	200	do...	315 B	
ng-validate.js	200	scr...	(from dis...)	

Summary: 3 requests | 430 B transferred | Finish: 193 ms | DOMContentLoaded: 99 ms ...



```
8 res.write('<html>');
9 res.write('<head><title>Enter Message</title><head>');
10 res.write('<body><form action="/message" method="POST"><input type="text" name="message"><');
11 res.write('</html>');
12 return res.end();
13 }
14 if (url === '/message' && method === 'POST') {
15   const body = [];
16   req.on('data', (chunk) => {
17     console.log(chunk);
18     body.push(chunk);
19   });
20   req.on('end', () => {
21     const parsedBody = Buffer.concat(body).toString();
22     console.log(parsedBody);
23   });
24   fs.writeFileSync('message.txt', 'DUMMY');
25   res.statusCode = 302;
26   res.setHeader('Location', '/');
27   return res.end();
28 }
```

Maximilians-MBP:nodejs-complete-guide mschwarzmuellers\$ node app.js  
^C  
Maximilians-MBP:nodejs-complete-guide mschwarzmuellers\$ node app.js  
<Buffer 6d 65 73 73 61 67 65 3d 66 61 73 64 66 61 73 64 66>  
message=fasdfasdf

- ‘<Buffer 6d 65 73 73 61 67 65 3d 66 61 73 64 66 61 73 64 66>’ is a chunk we can’t work with.
  - but the ‘message=fasdfasdf’ which is the parsedBody receive or yields this line and that is something we can work with
- and it’s message equals something because we named our input here message and as i said, that form will automatically send that request where it takes all the input data and puts it into the request body as key value pairs where the names assigned to the inputs are the keys and the values are what the user entered and that’s what we have here. ‘message=fasdfasdf’
- an with that, we can now work with that and finally store the input in our file and we can do that ‘parsedBody’
-   
  
  
  




EXPLORER

NODEJS-COMPLETE-GUIDE

.gitignore

app.jsM

message.txtU

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

req.method;

{

nl>');

ad<<title>Enter Message</title><head>');

dy<<form action="/message" method="POST"><input type="text" name="message"><button type="submit

tml>');

();

ssage' && method === 'POST') {

};

(chunk) => {

chunk;

unk;

};

() => {

Body = Buffer.concat(body).toString();

parsedBody);

};

nc('message.txt', 'DUMMY');

= 302;

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

1: node

Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

^C

Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

<Buffer 6d 65 73 73 61 67 65 3d 66 61 73 64 66 61 73 64 66>

message=fasdfasdf

OUTLINE

02-basics\* 0 0 0

Ln 10, Col 86 (1 selected) Spaces: 2 UTF-8 LF JavaScript Prettier

EXPLORER

NODEJS-COMPLETE-GUIDE

.gitignore

app.jsM

message.txtU

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

res.write('<body><form action="/message" method="POST"><input type="text" name="message"><

res.write('</html>');

return res.end();

}

if (url === '/message' && method === 'POST') {

const body = [];

req.on('data', (chunk) => {

console.log(chunk);

body.push(chunk);

});

req.on('end', () => {

const parsedBody = Buffer.concat(body).toString();

const message = parsedBody.split('=')[1];

fs.writeFileSync('message.txt', message);

});

res.statusCode = 302;

res.setHeader('Location', '/');

return res.end();

}

res.setHeader('Content-Type', 'text/html');

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

1: node

^C

Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

<Buffer 6d 65 73 73 61 67 65 3d 66 61 73 64 66 61 73 64 66>

message=fasdfasdf

^C

Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

OUTLINE

02-basics\* 0 0 0

Ln 21, Col 34 Spaces: 2 UTF-8 LF JavaScript Prettier

hello

Send

Elements Console Network >>

View: [Icons] Group by frame [X] Preserve

Filter [X] Hide data URLs

[All] XHR JS CSS Img Media Font Doc WS Manifest Other

Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	...	Size	...	Waterfall	▲
message	302	x-...	...	115 B	...		
localhost	200	do...	...	315 B	...		
ng-validate.js	200	scr...	...	(from dis...	...		

3 requests | 430 B transferred | Finish: 246 ms | DOMContentLoaded: 138 m...

Console What's New X

Highlights from the Chrome 68 update

Send

Elements Console Network >>

View: [Icons] Group by frame [X] Preserve

Filter [X] Hide data URLs

[All] XHR JS CSS Img Media Font Doc WS Manifest Other

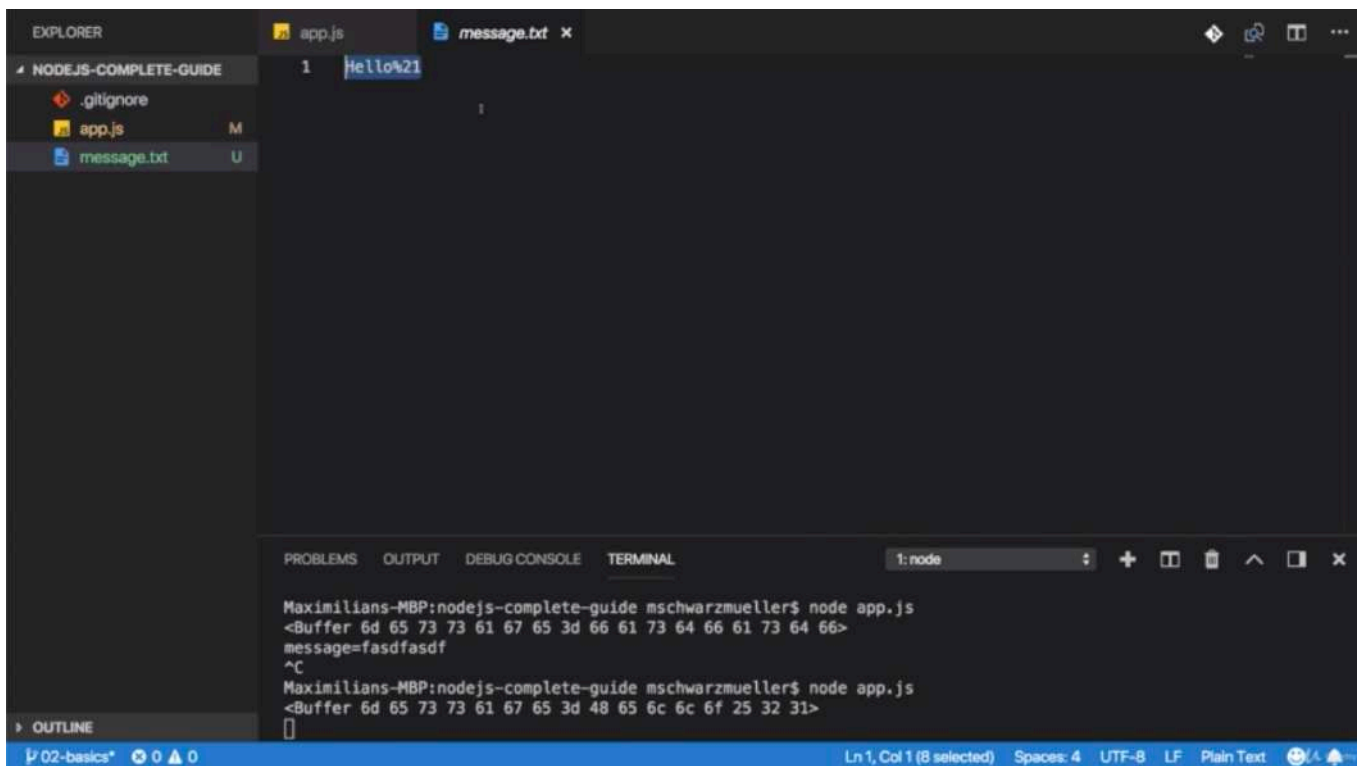
Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	...	Size	...	Waterfall	▲
message	302	x-...	...	115 B	...		
localhost	200	do...	...	315 B	...		
ng-validate.js	200	scr...	...	(from dis...	...		

3 requests | 430 B transferred | Finish: 215 ms | DOMContentLoaded: 113 m...

Console What's New X

Highlights from the Chrome 68 update



```
1 //app.js
2
3 const http = require('http');
4 const fs = require('fs');
5
6 const server = http.createServer((req, res) => {
7   const url = req.url;
8   const method = req.method;
9   if (url === '/') {
10     res.write('<html>');
11     res.write('<head><title>Enter Message</title><head>');
12     res.write('<body><form action="/message" method="POST"><input type="text"
name="message"><button type="submit">Send</button></form></body>');
13     res.write('</html>');
14     return res.end();
15   }
16   if (url === '/message' && method === 'POST') {
17     /**i will create a new constant 'body' here
18     * because i will try to read the request body
19     *
20     * const means 'we can never re-assign new value.'
21     * but push() we are changing the object behind that body element we are editing
22     * that data in that object not the value itself,
23     *
24     */
25     const body = [];
26     /**'on()' allows us to listen to certain events and the event i wanna listen to is the
data event
27     * data event will be fire whenever a new chunk is ready to be read
28     * that buffer thing is helping us with that.
29     *
30     * we have to add a second argument which is that function
31     * which should be executed for every data event.
32     *
33     * this listener receives a chunk of data.
```

```

34     * so here we receive a chunk which is something we can work with here
35     * and we have to something with this chunk to be able to interact with it.
36     */
37     req.on('data', (chunk) => {
38         console.log(chunk);
39         body.push(chunk);
40     });
41     /*'end' listener will be fired when once it's done parsing the incoming requests data
or incoming requests in general.
42     *
43     * i will use Buffer object which is available globally.
44     * why we convert to String is because the incoming data will be text
45     * because the body of that request will be text.
46     * if it were a file, we would have to do something different.
47     */
48     req.on('end', () => {
49         const parsedBody = Buffer.concat(body).toString();
50         /*'[1]' is index of value in key-value pair */
51         const message = parsedBody.split('=')[1];
52         fs.writeFileSync('message.txt', message);
53     });
54     res.statusCode = 302;
55     res.setHeader('Location', '/');
56     return res.end();
57 }
58 res.setHeader('Content-Type', 'text/html');
59 res.write('<html>');
60 res.write('<head><title>My First Page</title><head>');
61 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
62 res.write('</html>');
63 res.end();
64 });
65
66 server.listen(3000);

```

```

1 //message.txt
2
3 '%21' is exclamation mark(!)
4
5 Hello%21

```

```

1 //clean version
2
3 //app.js
4
5 const http = require('http');
6 const fs = require('fs');
7
8 const server = http.createServer((req, res) => {
9     const url = req.url;
10    const method = req.method;
11    if (url === '/') {
12        res.write('<html>');
13        res.write('<head><title>Enter Message</title><head>');
14        res.write('<body><form action="/message" method="POST"><input type="text"
name="message"><button type="submit">Send</button></form></body>');
15        res.write('</html>');

```

```

16     return res.end();
17 }
18 if (url === '/message' && method === 'POST') {
19     const body = [];
20     req.on('data', (chunk) => {
21         console.log(chunk);
22         body.push(chunk);
23     });
24     req.on('end', () => {
25         const parsedBody = Buffer.concat(body).toString();
26         const message = parsedBody.split('=')[1];
27         fs.writeFileSync('message.txt', message);
28     });
29     res.statusCode = 302;
30     res.setHeader('Location', '/');
31     return res.end();
32 }
33 res.setHeader('Content-Type', 'text/html');
34 res.write('<html>');
35 res.write('<head><title>My First Page</title><head>');
36 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
37 res.write('</html>');
38 res.end();
39 });
40
41 server.listen(3000);
42

```

## \* Chapter 34: Understanding Event Driven Code Execution

- the order of execution of your code is not necessarily the order in which you write it. for example,

```

1 req.on('end', () => {
2     const parsedBody = Buffer.concat(body).toString();
3     const message = parsedBody.split('=')[1]
4     fs.writeFileSync('message.txt', message)
5 })

```

will execute after this code.

```

1     res.statusCode = 302;
2     res.setHeader('Location', '/');
3     return res.end();

```

so it will even execute after we already sent the response.

- sending the response doesn't mean that our event listeners are dead. they will still execute even if the response is already gone.

- but it also means that if we do something in the event listener that should influence the response this is a wrong way of setting it up. we should then also move the response code into the event listener. if we had such a dependency like below

```

1 req.on('end', () => {

```

```

2   const parsedBody = Buffer.concat(body).toString();
3   const message = parsedBody.split('=')[1];
4   fs.writeFileSync('message.txt', message)
5   res.statusCode = 302
6   res.setHeader('Location', '/')
7   return res.end()
8 })

```

- and 'req.on()' and 'http.createServer()' are examples where node.js uses a pattern where you pass a function to a function and node.js will execute these passed in functions at a later point of time which is called asynchronously. now it's not always the case that a passed-in function is necessarily executed at a later point of time. but node.js has used this pattern heavily and let you know when this is the case and when node executes something asynchronously.

- in such case, node.js won't immediately run that function. instead, it will add a new event listener internally, it manages all these listeners initially. in this case for the 'end' event. and then it will then call that function for you once it is done

- so in the end, you can think of this like node.js having some internal registry of events and listeners to these events and a function like this is such a listener and when served something happens.

- when node.js is done parsing your request, it will go through the registry and see 'i'm done with the request so i should now send the end event.' so let's see which listener i have for that, and will then find this function

```

1 req.on('end', () => {
2   const parsedBody = Buffer.concat(body).toString();
3   const message = parsedBody.split('=')[1];
4   fs.writeFileSync('message.txt', message)
5   res.statusCode = 302
6   res.setHeader('Location', '/')
7   return res.end()
8 })

```

- and any other functions will now call them but it will not POST code execution.

- flow is like this

- it will reach if statement and if these conditions are met, it will go inside of it.

- it will then register these 2 handlers(req.on()) and not immediately execute inner function. instead the functions are just registered internally in. it's event emitter registry and then it will jump to the next line(res.setHeader()~res.end())







EXPLORER

app.js

NODEJS-COMPLETE-GUIDE

.gitignore

app.js

message.txt

```
16 req.on('data', (chunk) => {
17   console.log(chunk);
18   body.push(chunk);
19 });
20 req.on('end', () => {
21   const parsedBody = Buffer.concat(body).toString();
22   const message = parsedBody.split('=')[1];
23   fs.writeFileSync('message.txt', message);
24   res.statusCode = 302;
25   res.setHeader('Location', '/');
26   return res.end();
27 });
28
29 res.setHeader('Content-Type', 'text/html');
30 res.write('<html>');
31 res.write('<head><title>My First Page</title><head>');
32 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
33 res.write('</html>');
34 res.end();
35 });
36
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

1: node

message=fasdfasdf  
^C  
Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js  
<Buffer 6d 65 73 73 61 67 65 3d 48 65 6c 6c 6f 25 32 31>  
^C  
Maximilians-MBP:nodejs-complete-guide mschwarzmueller\$ node app.js

OUTLINE

02-basics\* 0 0 0

Ln 29, Col 46 (45 selected) Spaces: 2 UTF-8 LF JavaScript Prettier

href

String

Elements Console Network

View: Group by frame Preserve

Filter Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other

Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	Size	Waterfall
message	302	x-...	115 B	
localhost	200	do...	315 B	
ng-validate.js	200	scr...	(from dis...	

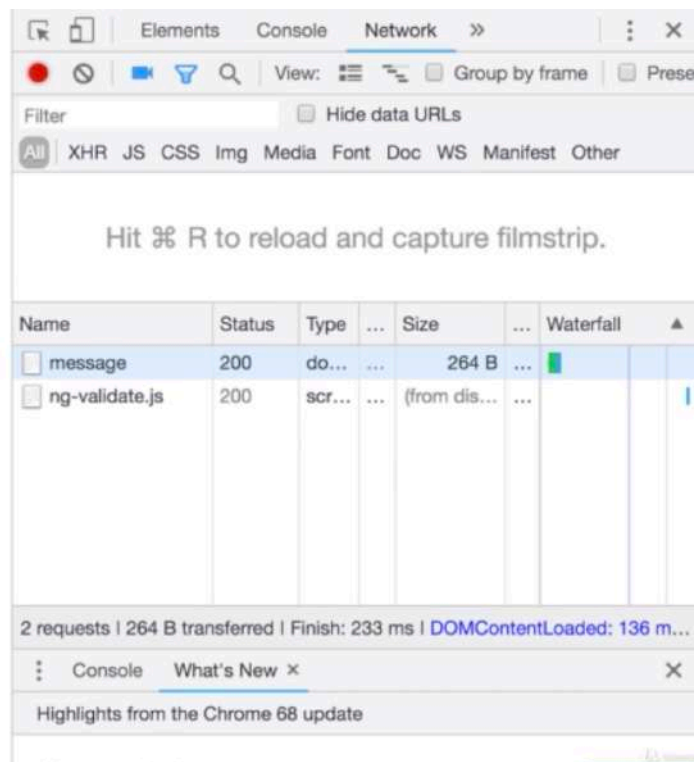
3 requests | 430 B transferred | Finish: 215 ms | DOMContentLoaded: 113 m...

Console What's New

Highlights from the Chrome 68 update

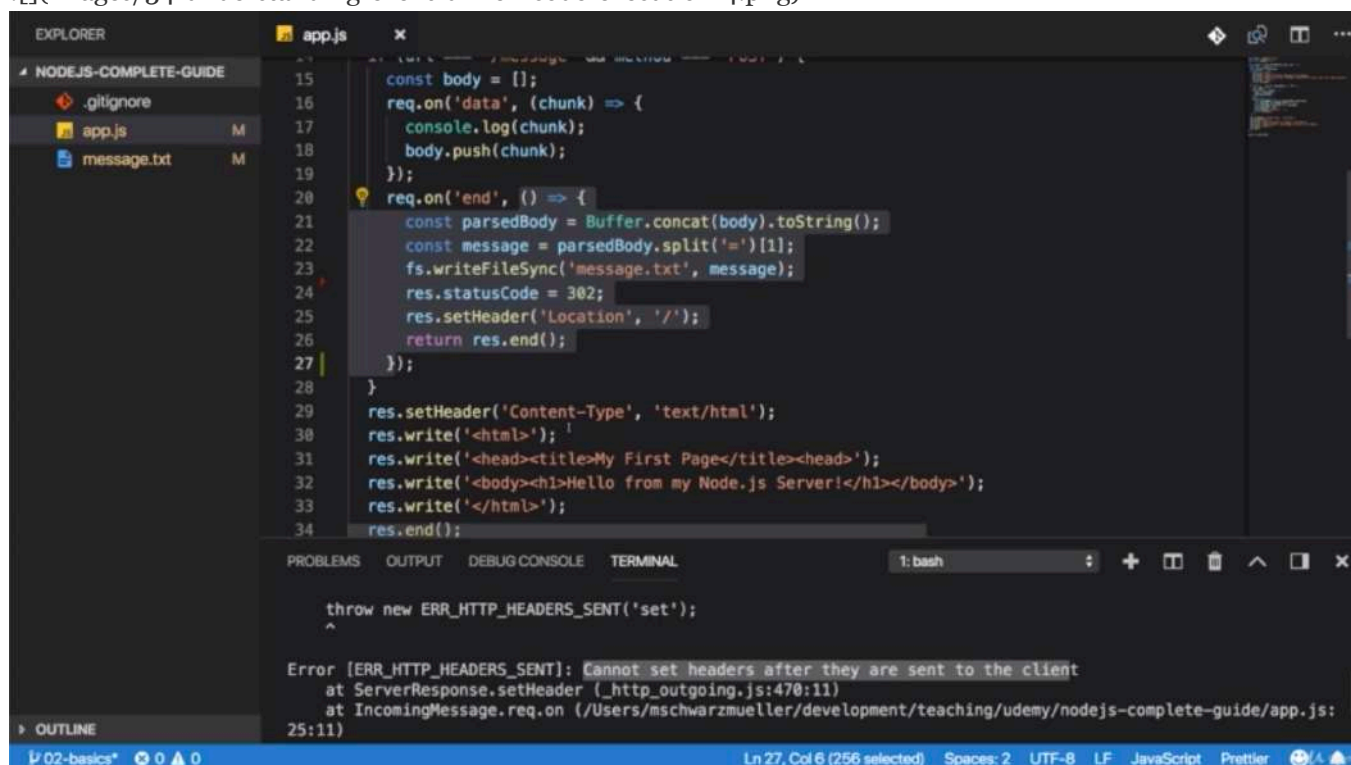


Hello from my Node.js Server!



- if i enter something here and i get redirected to this page or not even redirect it as you can see. there is no 300 status code





- instead of just this line(res.setHeader()~res.end()). because it executes these lines because it will not execute inner function of below right away.

```
1 req.on('end', () => {  
2   const parsedBody = Buffer.concat(body).toString();  
3   const message = parsedBody.split('=')[1];  
4   fs.writeFileSync('message.txt', message)  
5   res.statusCode = 302  
6   res.setHeader('Location', '/')  
7   return res.end()  
8 })
```



- so this return statement will therefore not quit this overarching function. instead it just registers this callback and immediately move onto the next line. this above codes eventually will be executed but that is already too late. that's why we get 'Cannot set headers after they are sent to the client' because it already moved the long and executed this code.

- when all of sudden the parsing of the request finished, and it executed callback innerfunction and tried to again send a response which obviously it too late because it already did here.

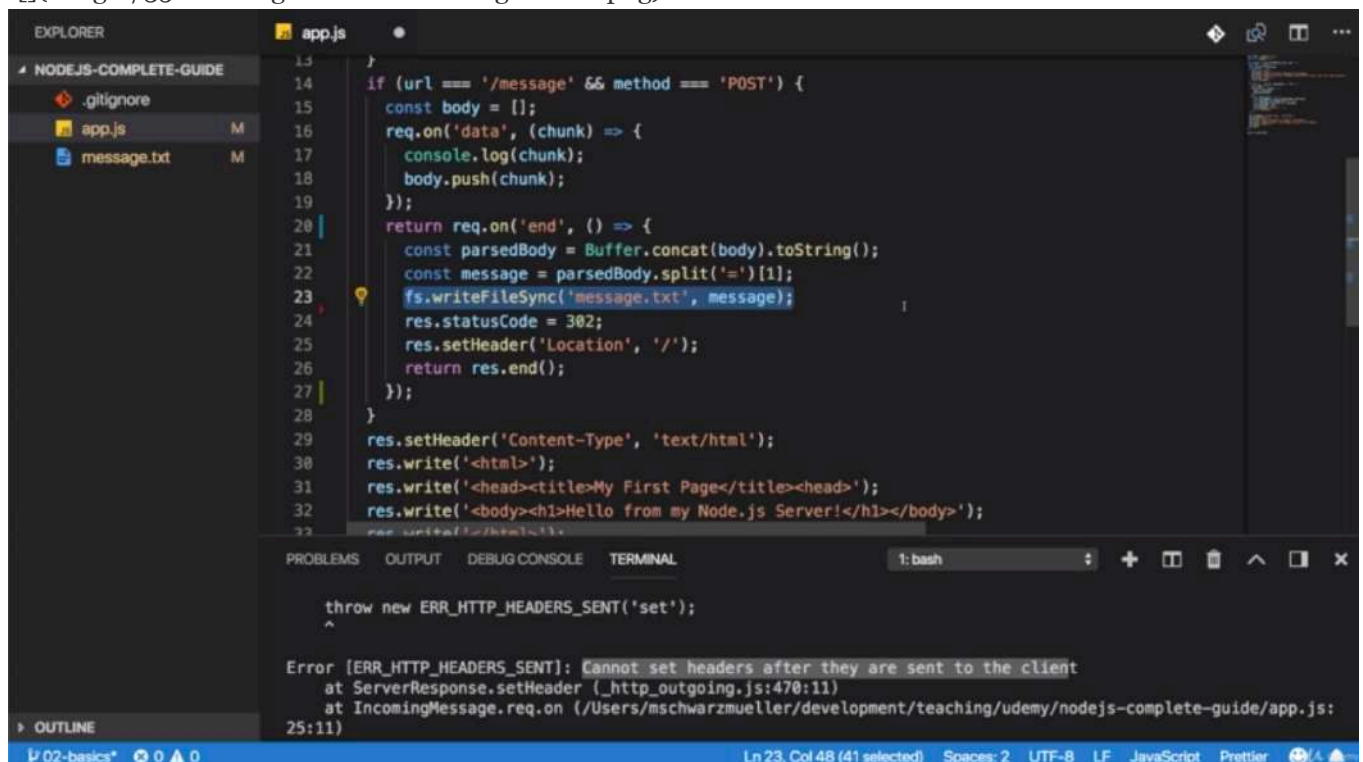
- you can register callback function which run sometime in the future but not necessarily right now. so therefore the next line of code(res.setHeader()~res.end()) will run before callback function

- this setup is important because otherwise node would have to pause until it's done pause until it wrote the file and if it does that, it will simply slow our server down and it's not able to handle our incoming requests or do anything of that kind until it's done

## \* Chapter 35: Blocking and Non-Blocking Code

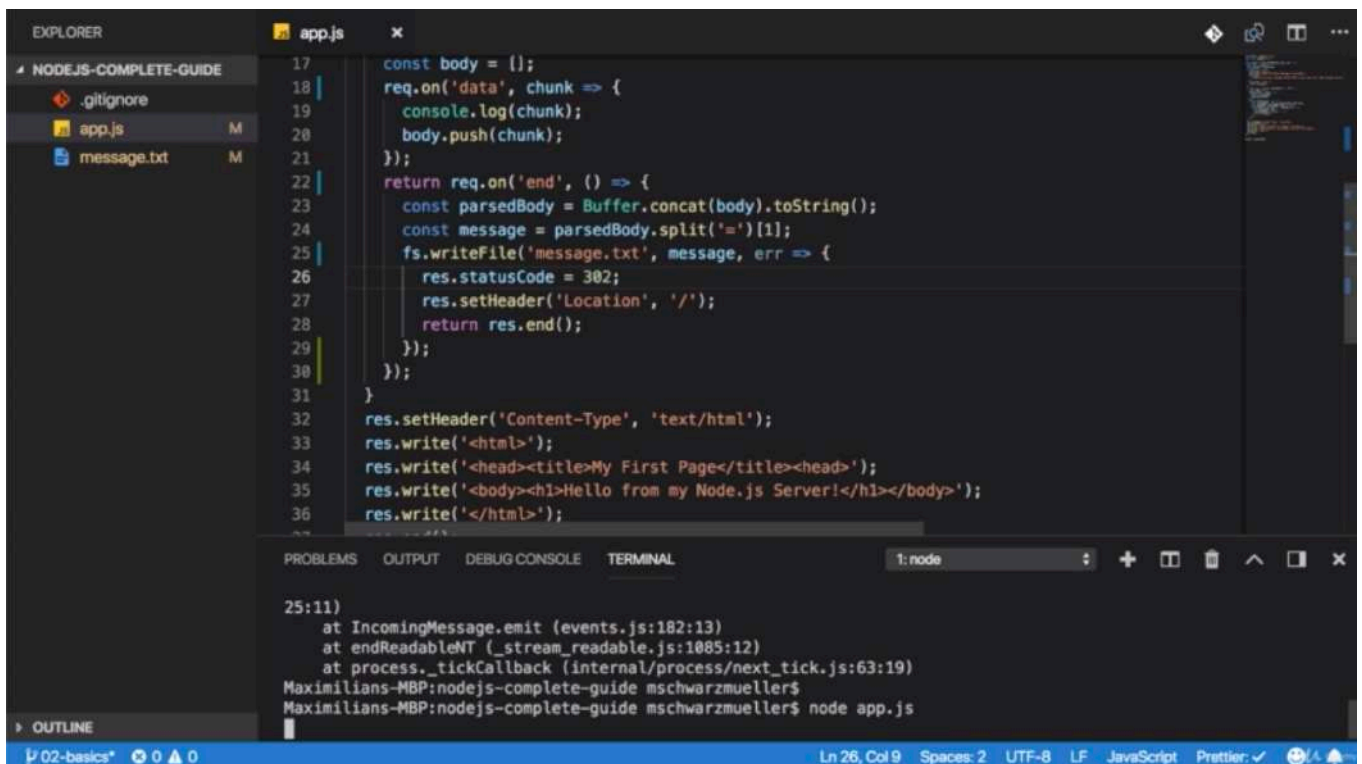






```
13 }
14 if (url === '/message' && method === 'POST') {
15   const body = [];
16   req.on('data', (chunk) => {
17     console.log(chunk);
18     body.push(chunk);
19   });
20   return req.on('end', () => {
21     const parsedBody = Buffer.concat(body).toString();
22     const message = parsedBody.split('=')[1];
23     fs.writeFileSync('message.txt', message);
24     res.statusCode = 302;
25     res.setHeader('Location', '/');
26     return res.end();
27   });
28 }
29 res.setHeader('Content-Type', 'text/html');
30 res.write('<html>');
31 res.write('<head><title>My First Page</title><head>');
32 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
33 res.write('</html>');
34 res.end();
```

```
throw new ERR_HTTP_HEADERS_SENT('set');
^
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
    at ServerResponse.setHeader (_http_outgoing.js:470:11)
    at IncomingMessage.req.on (/Users/mschwarzmueller/development/teaching/udemy/nodejs-complete-guide/app.js:
25:11)
```

The image shows a VS Code editor window with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'NODEJS-COMPLETE-GUIDE' with files '.gitignore', 'app.js', and 'message.txt'. The 'app.js' file is open in the editor, showing JavaScript code that handles an HTTP request, logs the body, and writes it to 'message.txt'. The terminal shows the command 'node app.js' being executed, and the output indicates that the file was written successfully. The status bar at the bottom shows 'Ln 26, Col 9' and 'Spaces: 2'.

- 'Sync' in 'writeFileSync' stands for synchronous and this is special method which will block code execution until this file is created.

- if you do something with the huge file, you block the code execution until huge file is done. then the next line and all the other code will not continue to run until that file operation is done.

- so it's better to use 'writeFile()' method than 'writeFileSync()' method. and 3rd argument is callbackfunction which will be executed when it's done. so there i pass another function and this callback function receive an error object which will be null if no error occurs.













chrome://devtools/

View: [Icons] Group by frame [Icons] Preserve

Filter [x] Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other

Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	...	Size	...	Waterfall	▲
localhost	200	do...	...	315 B	...		
ng-validate.js	200	scr...	...	(from dis...	...		

2 requests | 315 B transferred | Finish: 230 ms | DOMContentLoaded: 132 m...

⋮ Console What's New ✕

Highlights from the Chrome 68 update

hello agant

Send

chrome://devtools/

View: [Icons] Group by frame [Icons] Preserve

Filter [x] Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other

Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	...	Size	...	Waterfall	▲
localhost	200	do...	...	315 B	...		
ng-validate.js	200	scr...	...	(from dis...	...		

2 requests | 315 B transferred | Finish: 230 ms | DOMContentLoaded: 132 m...

⋮ Console What's New ✕

Highlights from the Chrome 68 update

Send

Hit ⌘ R to reload and capture filmstrip.

Name	Status	Type	Size	Waterfall
message	302	x-...	115 B	
localhost	200	do...	315 B	
ng-validate.js	200	scr...	(from dis...)	

3 requests | 430 B transferred | Finish: 209 ms | DOMContentLoaded: 113 m...

Console What's New

Highlights from the Chrome 68 update

EXPLORER

app.js message.txt

1 | Hello+again

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

1: node

```
at IncomingMessage.emit (events.js:182:13)
at endReadableNT (_stream_readable.js:1085:12)
at process._tickCallback (internal/process/next_tick.js:63:19)
Maximilians-MBP:nodejs-complete-guide mschwarzmuellers$
Maximilians-MBP:nodejs-complete-guide mschwarzmuellers$ node app.js
<Buffer 6d 65 73 73 61 67 65 3d 48 65 6c 6c 6f 2b 61 67 61 69 6e>
```

OUTLINE

02-basics\* 0 0

Ln 1, Col 7 Spaces: 4 UTF-8 LF Plain Text

```
1 const http = require('http');
2 const fs = require('fs');
3
4 const server = http.createServer((req, res) => {
5   const url = req.url;
6   const method = req.method;
7   if (url === '/') {
8     res.write('<html>');
9     res.write('<head><title>Enter Message</title><head>');
10    res.write(
11      '<body><form action="/message" method="POST"><input type="text" name="message"><button
type="submit">Send</button></form></body>'
12    );
13    res.write('</html>');
```

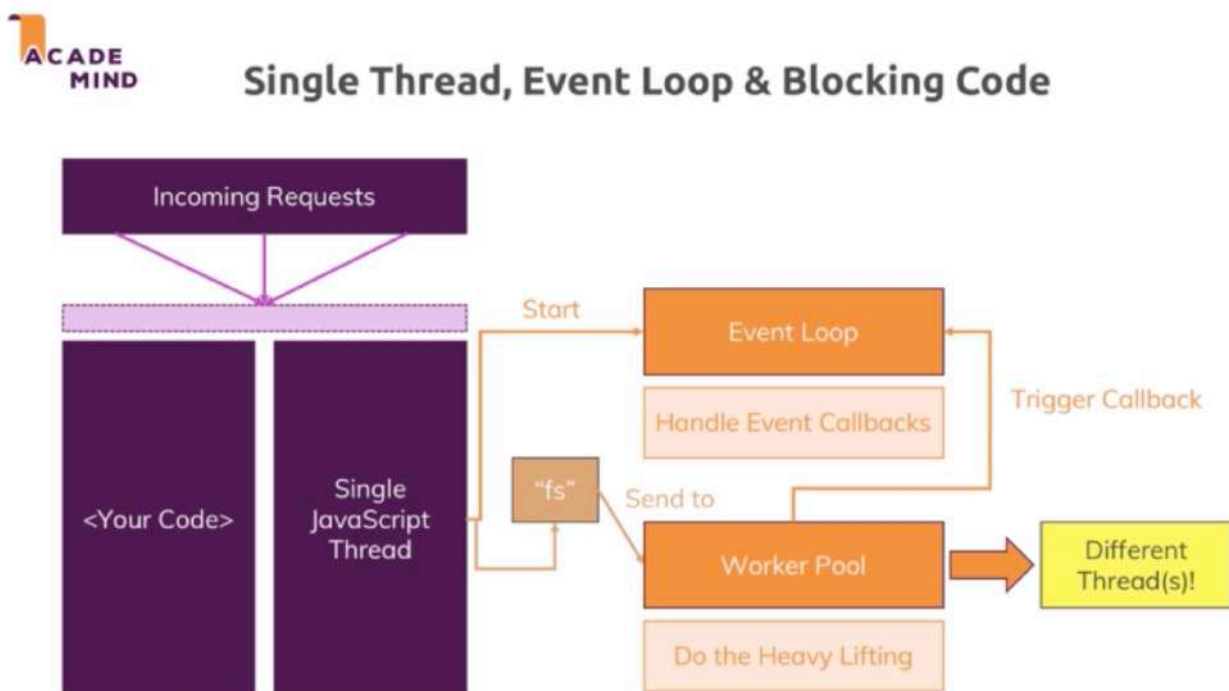
```

14     return res.end();
15 }
16 if (url === '/message' && method === 'POST') {
17     const body = [];
18     req.on('data', chunk => {
19         console.log(chunk);
20         body.push(chunk);
21     });
22     return req.on('end', () => {
23         const parsedBody = Buffer.concat(body).toString();
24         const message = parsedBody.split('=')[1];
25         fs.writeFile('message.txt', message, err => {
26             res.statusCode = 302;
27             res.setHeader('Location', '/');
28             return res.end();
29         });
30     });
31 }
32 res.setHeader('Content-Type', 'text/html');
33 res.write('<html>');
34 res.write('<head><title>My First Page</title><head>');
35 res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
36 res.write('</html>');
37 res.end();
38 });
39
40 server.listen(3000);
41

```

## \* Chapter 36: Node.js - Looking Behind The Scenes

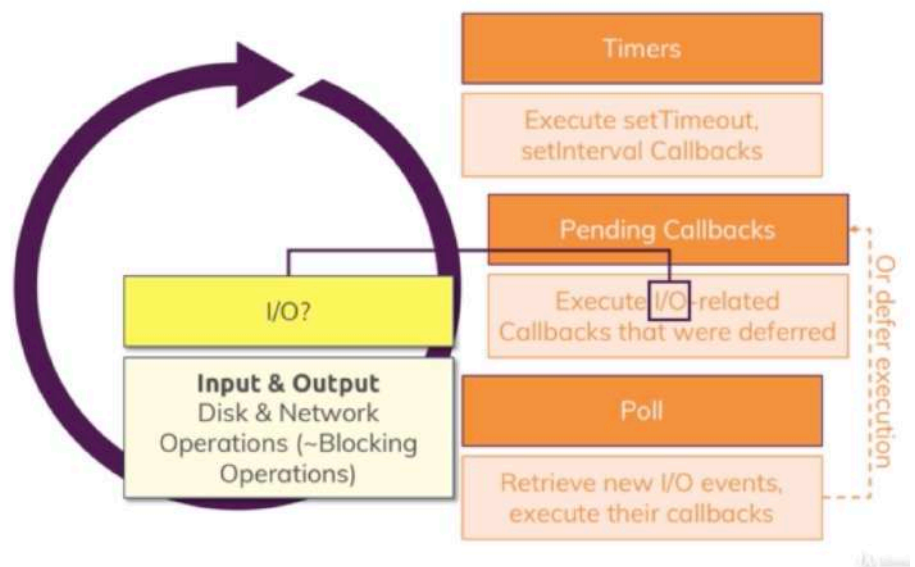




- 'thread' is basically like a process in your operating system.
  - Event Loop is automatically started by node.js when your program starts.
  - long taking file operation is not handled by event loop, just the callback that we might have defined on writeFile once it's done. the event loop will only handle callbacks that contain fast finishing code.
  - Worker Pool is spun up and managed by node.js automatically and automatically. this worker pool is responsible for all the heavy lifting. Worker Pool is really detached from your code and this worker pool is therefore doing all the heavy lifting. if you're doing something with a file, well a worker from that pool will take care and will do its job totally detached from your code and from the request and from the event loop.
  - The one connection to the event loop we will have though is that once the worker is done, so for example once we read a file, it will trigger the callback for that read file operation and since the event loop is responsible for the event and the callbacks, this will in the end end up in the event loop.
- 
- 

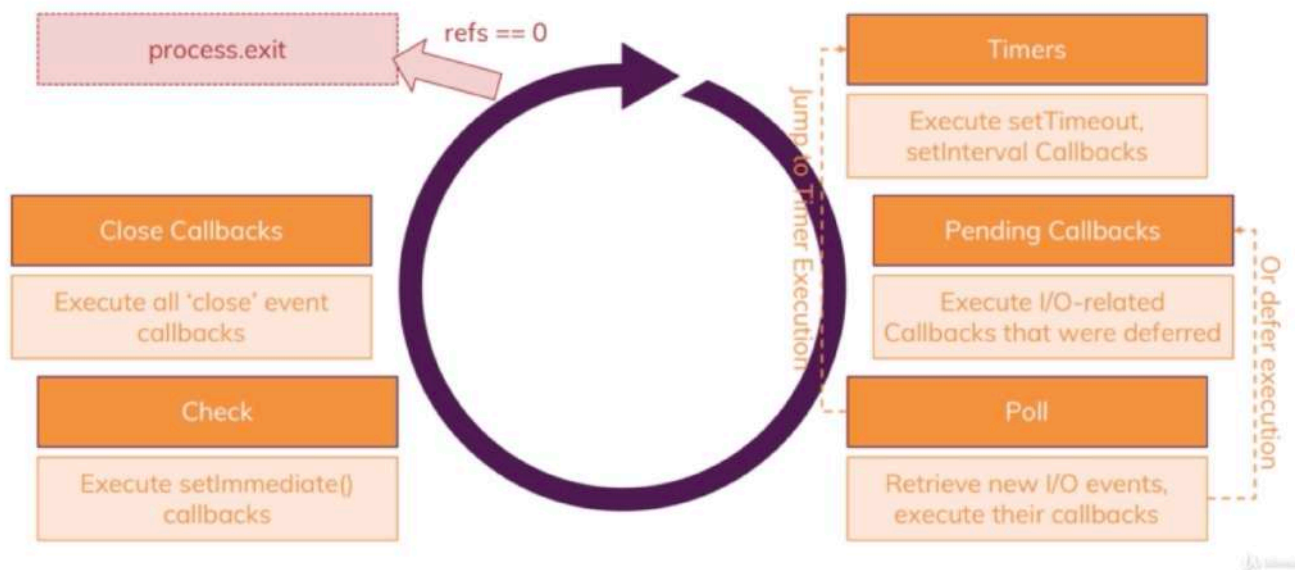


## The Event Loop





## The Event Loop



- at the beginning of each new iteration, it checks if there are any timer callbacks it should execute.  
 - in node.js, you set a timer and always pass a method, a function that should be executed once that timer completes and node.js is aware of this and at that beginning of each new loop iteration, it executes any due timer callback. so any callbacks that have to be executed because timer completes.

- then as a next step, it checks other callback for example, if we had write or read file, we might have a callback because that operation finished and it will then execute these callbacks.  
 - in here, I/O means blocking operation  
 - it's important to understand that node.js will leave that phase at a certain point of time and that can also mean that if there are too many outstanding callbacks, it will continue its loop iteration and postpone these callbacks to the next iteration to execute them. After working on these open callbacks and hopefully finishing them all, it will enter a poll phase

- The Poll phase is a phase where node.js will look for new I/O event and do its best to execute their callbacks immediately if possible. if that's not possible, it will defer the execution and register this as a pending callback.

- and it also will check if there are any timer callbacks due to be executed and if that is the case, it will jump to that timer phase and execute them right away. so it can actually jump back there and not finish the iteration otherwise it continue.

- and next setImmediate() callbacks will be executed in a so-called check phase. setImmediate() is like setTimeout or setInterval, just that it will execute immediately but always after any open callbacks have been executed. so typically faster than setTimeout with 1 millisecond of open duration, but after the current cycle well finished or at least finished open callbacks that were due to be handled in that current iteration.

- now we are entering a highly theoretical terrain. now we are nearing the end of each iteration cycle and now node.js will execute all close event callbacks. so if you registered any close events and in our code we haven't but

if you had any close event, this would be the point of time where node.js executes their appropriate callbacks.

- close events are basically handled separately or their callbacks are handled separately.

-----  
--  
- and then we might exit the whole node.js program but only if there are no remaining event handler which are registered and that is what i mean with this `refs==null` here, internally node.js keeps track of its open listener and it basically has a counter, references or refs which increments by 1 for every new callback that is registered, every new event listener that is registered. so every new future work that it has to do. and it reduces that counter by 1 for every event listener that it doesn't need anymore every callback it finished and since in a server environment, we create a server with `createServer` and then listen to incoming request with `listen`, this is an event which never is finished by default and therefore, we always have at least one reference and therefore we don't exit in a normal node web server program. we can call the exit function.

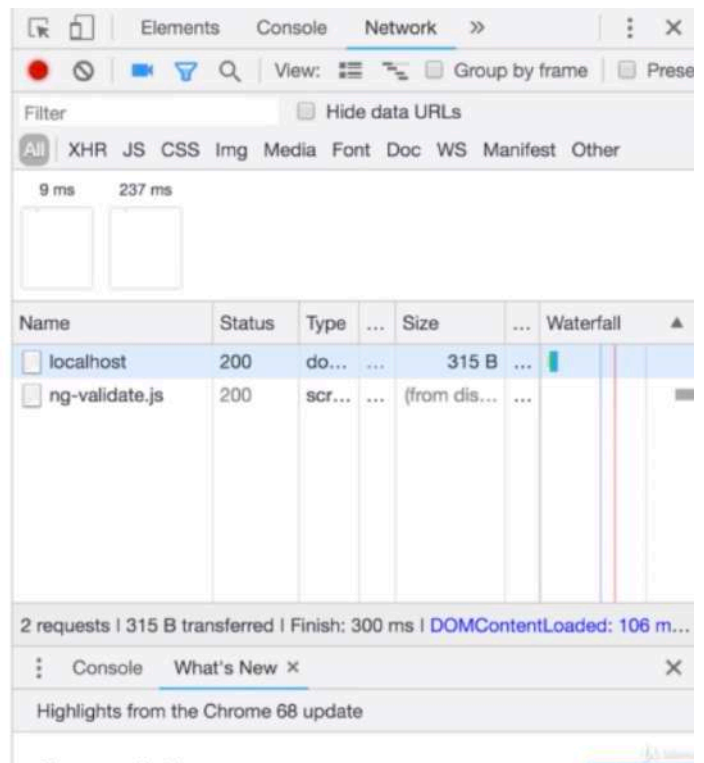
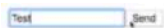
- when we used node to execute a file that didn't listen to a web server or on a web server, then it also finishes eventually once its done with its works.

## \* Chapter 37: Using The Node Modules System



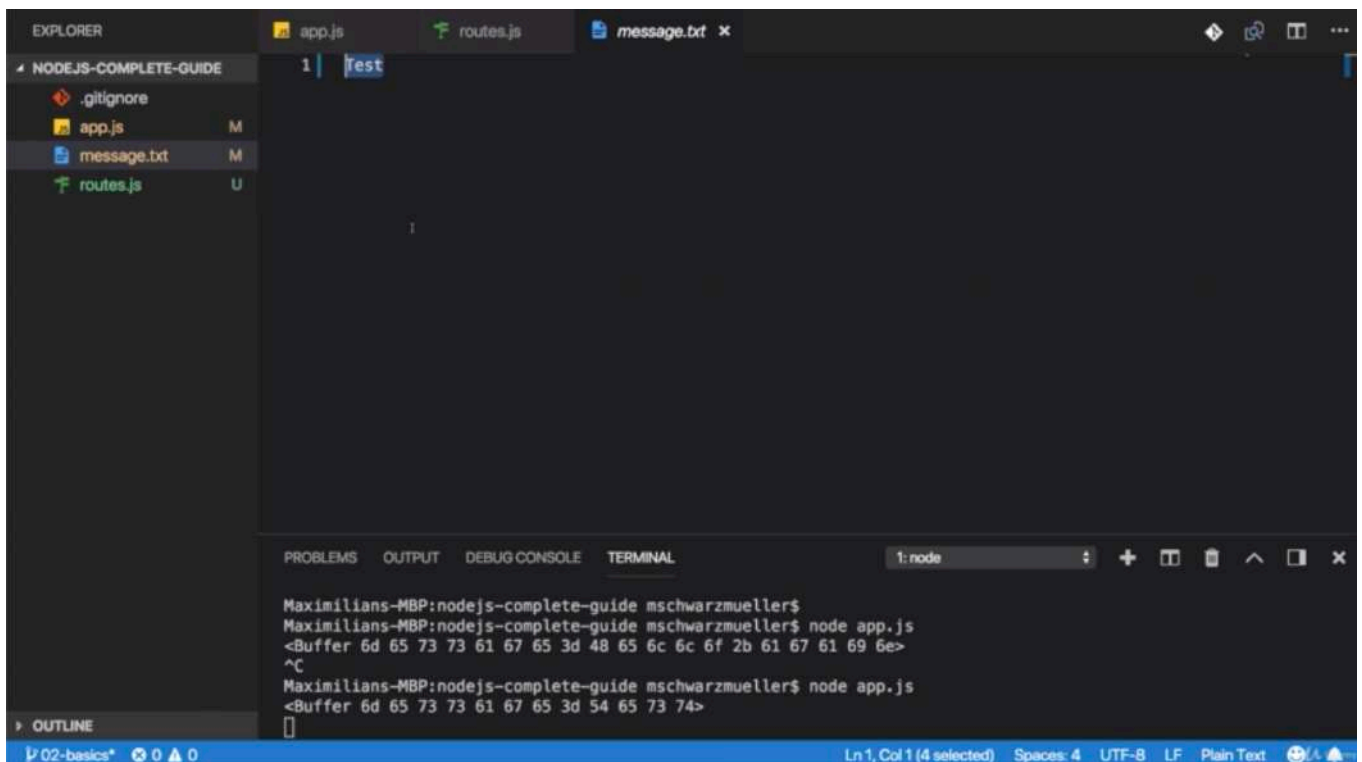
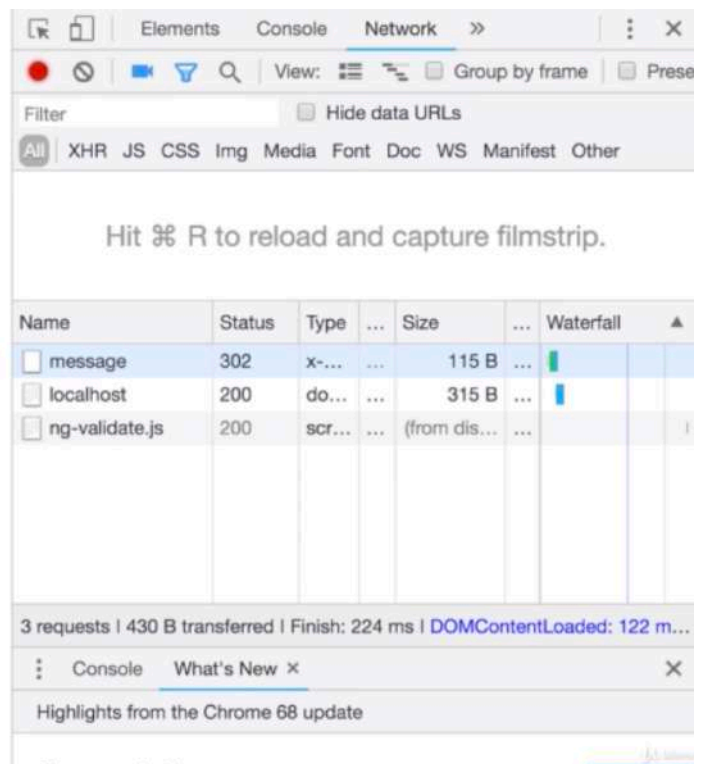




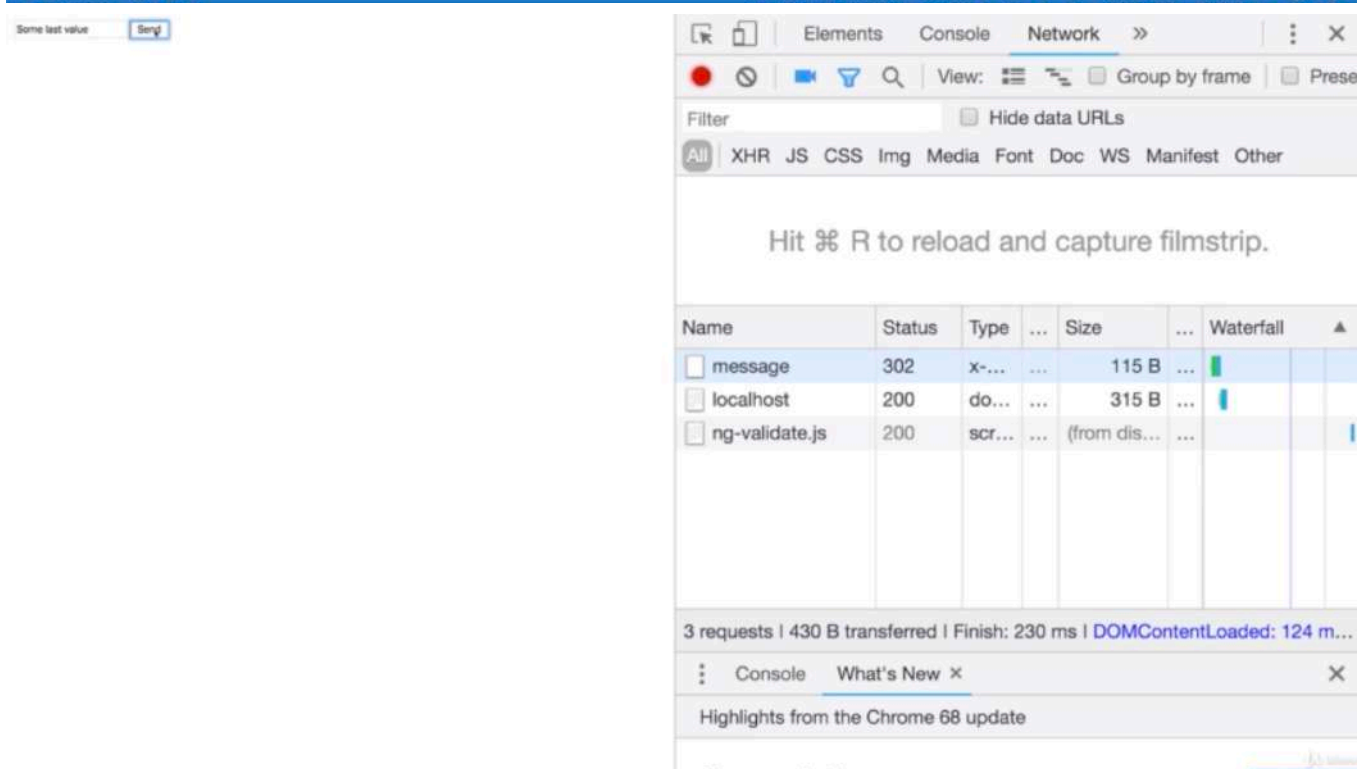


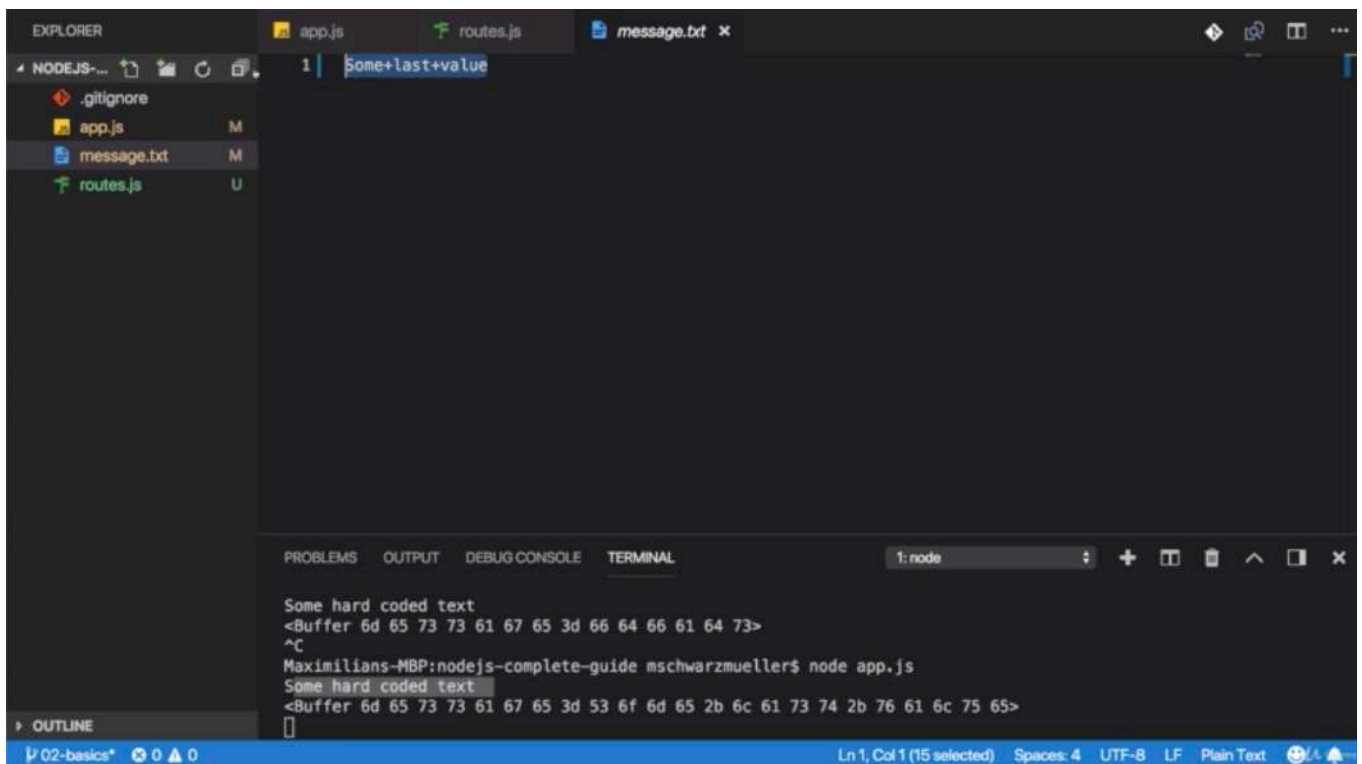


Send



- we split our code over 2 files, having 1 file which is very lean that just spins up the server.
  - that's important that also create a connection to another file through that import and through that export where we export our requestHandler function.
  - now One important node about node module system, the file content is actually cached by node and we can't edit it externally.
- 
- 
- 





- if we somehow would define routes as an object and we tried to add a new property on the fly like above, this would not manipulate the original file. so this is basically locked, not accessible from outside. we can only export stuff that we can now read from outside.

- though you could have functions which you export that start changing stuff inside of that file

- sometimes you export many things then do like below

```
1 module.exports = {  
2   handler: requestHandler,  
3   someText: 'Some ahrd coded text'  
4 }
```

or

```
1 module.exports.handler = requestHandler  
2 module.exports.someText = 'Some hard coded text'
```

or

```
1 //shortcut provided by node.js  
2  
3 exports.handler = requestHandler  
4 exports.someText = 'Some hard coded text'
```

to

```
1 const routes = require('./routes')  
2  
3 console.log(routes.someText)  
4 const server = http.createServer(routes.handler);  
5  
6 server.listen(3000);
```

```
1 //app.js  
2  
3 const http = require('http');
```

```

4
5 const routes = require('./routes');
6
7 console.log(routes.someText);
8
9 const server = http.createServer(routes.handler);
10
11 server.listen(3000);
12
13 //routes.js
14
15 const fs = require('fs');
16
17 const requestHandler = (req, res) => {
18   const url = req.url;
19   const method = req.method;
20   if (url === '/') {
21     res.write('<html>');
22     res.write('<head><title>Enter Message</title></head>');
23     res.write(
24       '<body><form action="/message" method="POST"><input type="text" name="message"><button
25       type="submit">Send</button></form></body>'
26     );
27     res.write('</html>');
28     return res.end();
29   }
30
31   if (url === '/message' && method === 'POST') {
32     const body = [];
33     req.on('data', chunk => {
34       console.log(chunk);
35       body.push(chunk);
36     });
37     return req.on('end', () => {
38       const parsedBody = Buffer.concat(body).toString();
39       const message = parsedBody.split('=')[1];
40       fs.writeFile('message.txt', message, err => {
41         res.statusCode = 302;
42         res.setHeader('Location', '/');
43         return res.end();
44       });
45     });
46   }
47
48   res.setHeader('Content-Type', 'text/html');
49   res.write('<html>');
50   res.write('<head><title>My First Page</title></head>');
51   res.write('<body><h1>Hello from my Node.js Server!</h1></body>');
52   res.write('</html>');
53   res.end();
54 };
55
56 // module.exports = requestHandler;
57
58 // module.exports = {
59 //   handler: requestHandler,
60 //   someText: 'Some hard coded text'
61 // }

```

```

47 // };
48
49 // module.exports.handler = requestHandler;
50 // module.exports.someText = 'Some text';
51
52 exports.handler = requestHandler;
53 exports.someText = 'Some hard coded text';

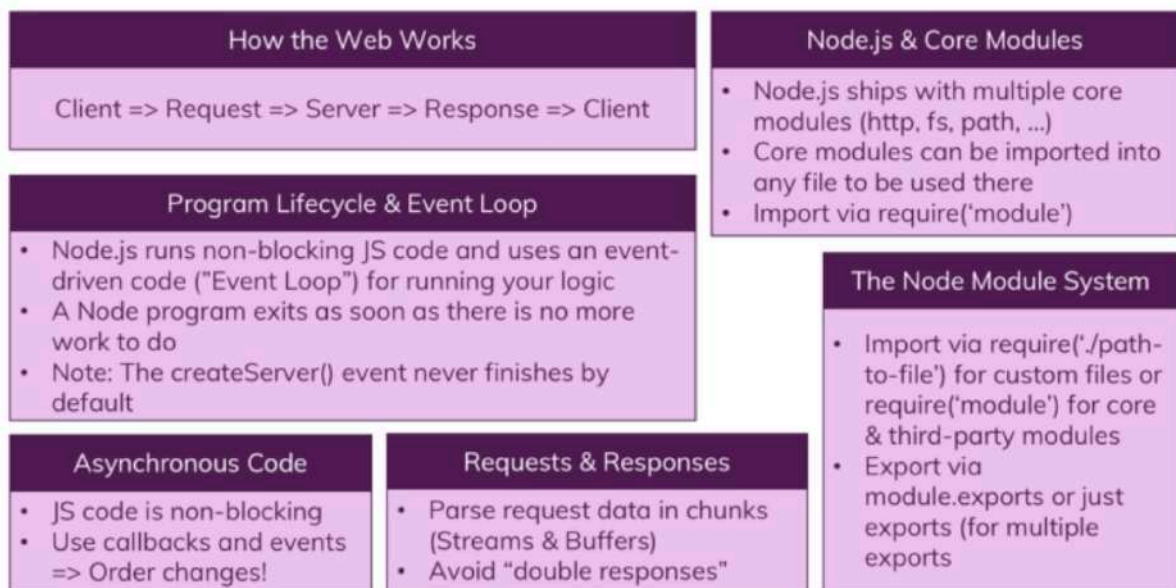
```

## \* Chapter 38: Wrap Up





### Module Summary



## \* Assignment

```

1 //app.js
2
3 const http = require('http');
4
5 const server = http.createServer((req, res) => {
6   const url = req.url;
7   if (url === '/') {
8     res.setHeader('Content-Type', 'text/html');
9     res.write('<html>');
10    res.write('<head><title>Assignment 1</title></head>');
11    res.write(
12      '<body><form action="/create-user" method="POST"><input type="text" name="username">
13      <button type="submit">Send</button></form></body>'
14    );
15    res.write('</html>');
16    return res.end();
17   }
18   if (url === '/users') {

```

```
18     res.setHeader('Content-Type', 'text/html');
19     res.write('<html>');
20     res.write('<head><title>Assignment 1</title></head>');
21     res.write('<body><ul><li>User 1</li><li>User 2</li></ul></body>');
22     res.write('</html>');
23     return res.end();
24 }
25 // Send a HTML response with some "Page not found text
26 if (url === '/create-user') {
27     const body = [];
28     req.on('data', chunk => {
29         body.push(chunk);
30     });
31     req.on('end', () => {
32         const parsedBody = Buffer.concat(body).toString();
33         console.log(parsedBody.split('=')[1]); // username=whatever-the-user-entered
34     });
35     res.statusCode = 302;
36     res.setHeader('Location', '/');
37     res.end();
38 }
39 });
40
41 server.listen(3000);
```