

# **Web Programming JavaScript – node.js - react**

Lecture 4-2:  
**Node.js Intro**  
16.11.

Stefan Noll  
Manuel Fehrenbach  
Winter Semester 22/23

REST

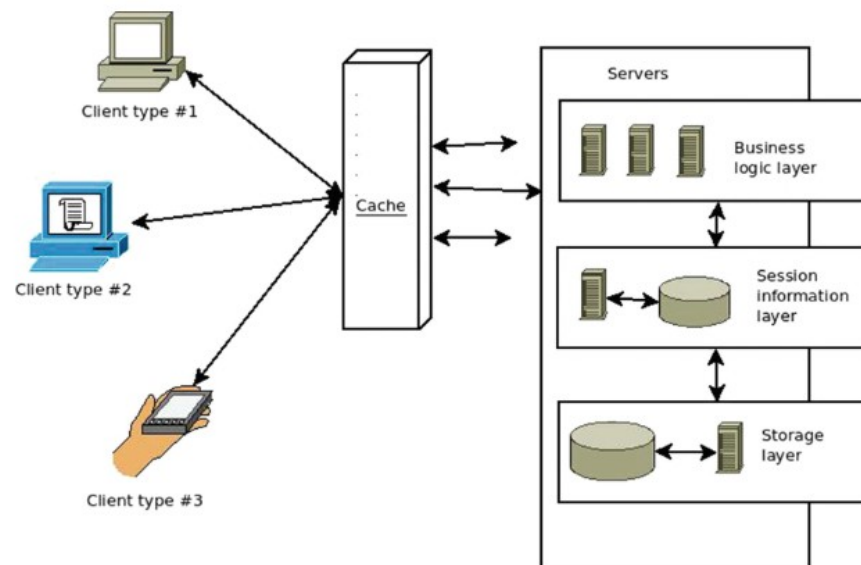
## - Where did it start?

- Roy Fielding, a main author of the HTTP protocol, mentions REST first time in his doctoral dissertation as an architectural style for distributed hypermedia systems
- Simple: **REST (Representational State Transfer)** is an architectural style defined to help create and organize distributed systems.
- Important aspect of REST is that it is an **architectural style**, not a guideline, not a standard, nor anything that would imply that there are a set of hard rules to follow to end up having RESTful architecture
- Because it is a style there is room for misinterpretations
- The main idea behind REST is that a distributed system, organized RESTfully, will improve in the following areas:
  - **Performance:** communication is meant to be efficient and simple
  - **Scalability:** any distributed system should be able to handle this aspect well enough, and simple interaction proposed by REST allows for this.
  - **Simplicity of interface:** A simple interface allows for simpler interactions between systems
  - **Modifiability of components:** Components can be modified independently of each other at the minimum cost and risk
  - **Portability:** It can be implemented and consumed by any type of technology
  - **Reliability:** stateless constraint proposed by REST allows for faster/easier recovery of a system after failure
  - **Visibility:** improved visibility, because monitoring system doesn't need to look further than a single request messages to determine the full state of a said request

- **REST Constraints (1):**
  - **Client-Server:**
    - A server is in charge of handling a set of services, and listens for requests regarding said service
  - **Stateless:**
    - Communication between client and server must be stateless, meaning that each request done from the client must have all the information required for the server to understand it, without taking advantage of any stored data
  - **Cacheable:**
    - It proposes that every response to a request must be explicitly or implicitly set as cacheable (when applicable)
  - **Uniform Interface:**
    - By defining a standard and uniform interface for all of your services, you effectively simplified the implementation of independent clients by giving them a clear set of rules to follow

- **REST Constraints (2):**
  - **Layered System:**
    - REST was designed with the internet in mind, and be able to handle massive amount of traffic.
    - Concept of layering was introduced, to achieve this.
      - Each layer can only use the layer under it, and output is communicated to the layer above

### Example of multilayered architecture



- **REST Constraints (3):**
  - **Resources:**
    - Anything that can be named can be a resource
    - Resources define what the services are going to be about, the type of information that is going to be transferred, and their related actions
    - Resources are structured most common as JSON or XML  
(A single resource can have more than one representation)
  - **Content Negotiation:**
    - Client sends header with the information of the different content types supported, with an optional indicator of how supported/preferred that format is
  - **Resource Identifier:**
    - Should provide a unique way of identification at any given moment and it should provide the full path to the resource.
    - The Identifier of each resource must be able to reference it unequivocally at any given moment in time.
      - Not ok: GET api/v1/books/last (invalid source ID, not unique)
      - Ok: GET apiv1/books/j-k-rowling/harry-potter-and-the-deathly-hollows  
(URI unique, because books with the same name can't exist from the same author)

- **REST Constraints (4):**
  - **Resources:**
    - **Actions**
      - HTTP provides methods which can be used to reference what should be done to the resource
        - **GET:** Access a resource in a read only mode
        - **POST:** Normally used to send a new resource into the server
        - **PUT:** Normally used to update a given resource (update action)
        - **DELETE:** Used to delete a resource
        - **HEAD:** Not part of CRUD actions, but the method is used to ask if a given resource exists without returning any of its representation
        - **OPTIONS:** Not part of CRUD actions, but used to retrieve a list of available methods on a given resource (i.e. What can the client do with a specific resource?)
      - To filter, search or working with sub-resources use „?“
        - PUT /api/v1/blogposts/12342?**action=like**
        - GET /api/v1/books?**q=[SEARCH-TERM]**
        - GET /api/v1/authors?**filters=[COMMA SEPARATED LIST OF FILTERS]**
      - If „?“ is not enough, there are other methods that can be used and still follow the REST style, but will not be discussed further

- **REST Constraints (4):**
  - **Status Code:**
    - **200 => OK**
    - **400 => Bad request**
    - **401 => Unauthorized**
    - **404 => Not Found**
    - **500 => Internal server error**

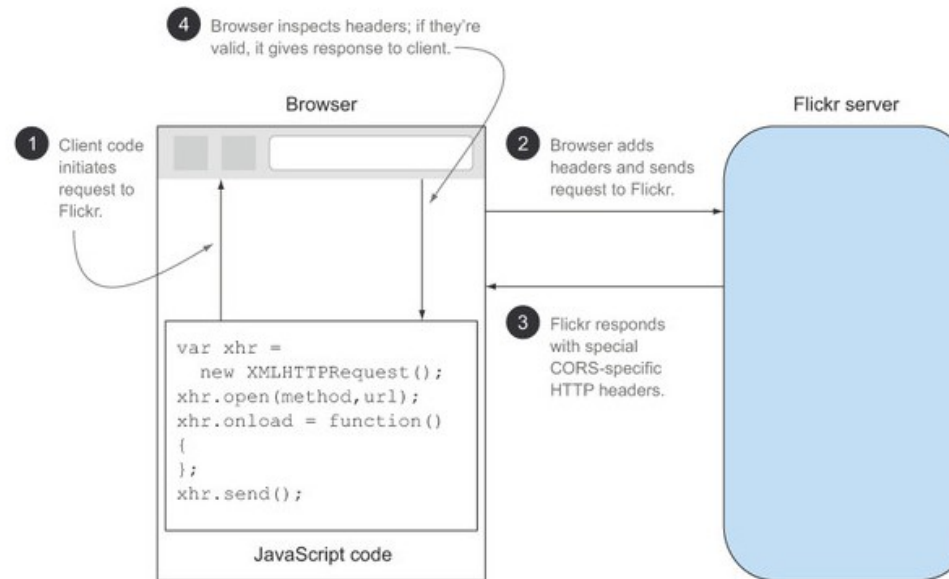


CORS

- Cross-Origin Resource Sharing
  - If CORS is not defined on the web server, resource request from a different source then the web server self will be denied.
  - With CORS the web server can specify which request sources can access the web server resources
  - CORS is a unique web technology in that it has both a server-side and a client-side component
    - Server-side component configures which types of cross-origin requests are allowed
    - Client-side component controls how corss-oriigin requests are made

- What issues does CORS solve and what is CORS?
  - For many years same origin policy prevented client-side JavaScript request to web-server because browser do not permit such requests
  - CORS allows cross-origin request in that, that it puts the web server in charge of who can make requests, and what kind of requests are allowed.
  - A web server can chose how many can access the the API: **One IP, several IPs or prevent access to all clients.**
  - Through **request headers** and **response headers** cross-origin request are allowed. Through these server and browser communicate with each other how cross-origin request behave

Figure 1.1. End-to-end CORS request flow



1. The CORS request is initiated by the JavaScript client code.
2. The browser includes additional HTTP headers on the request before sending the request to the server.
3. The server includes HTTP headers in the response that indicate whether the request is allowed.
4. If the request is allowed, the browser sends the response to the client code.

# Security

- XSS
  - Cross-site scripting vulnerabilities exist since 1996 during the early days of the World Wide Web. A time when e-commerce began to take off (Netscape, Yahoo,...)
  - After HTML was popular to create website, JavaScript was introduced and with it an unknown harbinger of cross-site scripting
  - JavaScript enabled web developers to create interactive web pages (image rollovers, floating menus, and the despised pop-up window)
  - Hackers found out, that they could load any website forcefully into a HTML frame within the same browser window. Through JavaScript they could read the data from the other website (different frame). Whrough this they could read username and password, steal cookies, or compromise any confidential information on the screen.
  - Because of this problem Netscape Communications (the dominat browser at that time) introduced „same-origin policy“. Restricting JavaScript on one website from accessing data in an other website

- XSS
  - In 1999 David Ross was working on security responses for Internet Explorer. He demonstrated, that through „script injection“ it is still possible to bypass all security mechanism to that day.
  - Through „script injection“ JavaScript code is added to a website via a HTML-form which will later be executed.
  - Later this kind of attack was named „cross site scripting“ (XSS)
- XSS in React
  - To be able to use XSS, one must be able to execute JavaScript Code through input-forms.
  - In React all inputs are converted to string and through this it is nearly impossible to use XSS attack in a React website
  - If the JSX-Tag ***dangerouslySetInnerHTML*** it is possible to use XSS attacks in React website

- CSRF
  - Cross-Site Request Forgery (CSRF) attacks take advantage of the way browsers operate and the trust relationship between a website and the browser.
  - Because some API endpoints trust too much the browser it is possible to use this trust to use CSRF attacks. We can craft links and forms that with little effort can cause a user to make requests on his or her own behalf, unknown to the user generating request.
  - CSRF attacks will oftentimes go unnoticed by the user that is being attacked, because requests in the browser occur in the background (behind the scenes). Through this it is possible to take advantage of privileged users to perform operations against a server without them knowing.
  - CSRF caused havoc throughout the web since its inception in the early 2000s
  - One way to combat CSRF attacks is to use cookie pairs (CSRF tokens), which will be set for forms. After every request these cookie pairs are changed.  
The server checks both cookies, if they are the same and only if they are the same the request is valid.
- CSRF in React
  - CSRF attacks can only be stopped by the backend.
  - The developer needs to search in the API-Doc, if there is something mentioned about CSRF and how the developer needs to implement it to use the API.



JWT

- JWT (JSON Web Token)
  - Is an open standard (RFC 7519)
  - With JWT it is possible to exchange data and information in a secure way.
  - To exchange data and information a JSON Object is used
  - On the JSON Object one can either use the HMAC algorithm or private/public key method to sign the JSON object.

- How is a JWT designed?
  - header
    - type: JWT => what kind of type it is (which standard is used)
    - alg: HS256 => which algorithm was used to sign the token
  - payload:
    - contains the data also named claims
    - claims
      - registered claims:
        - iss: issuer
        - exp: expiration time
        - sub: subject
        - ...
      - public claims
        - Can be defined at will by those using JWTs
      - private claims:
        - Defined by the parties who agreed to use specific claims
  - signature
    - To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM5OTk5fQ.4457QkKUD-McXzmmz81mEP1QaT70TGiVamr2zICh5C8
```

## Decoded

EDIT THE PAYLOAD AND SECRET

### HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

### PAYLOAD: DATA

```
{
  "sub": "123",
  "name": "Hans Müller",
  "iat": 1516239999
}
```

### VERIFY SIGNATURE

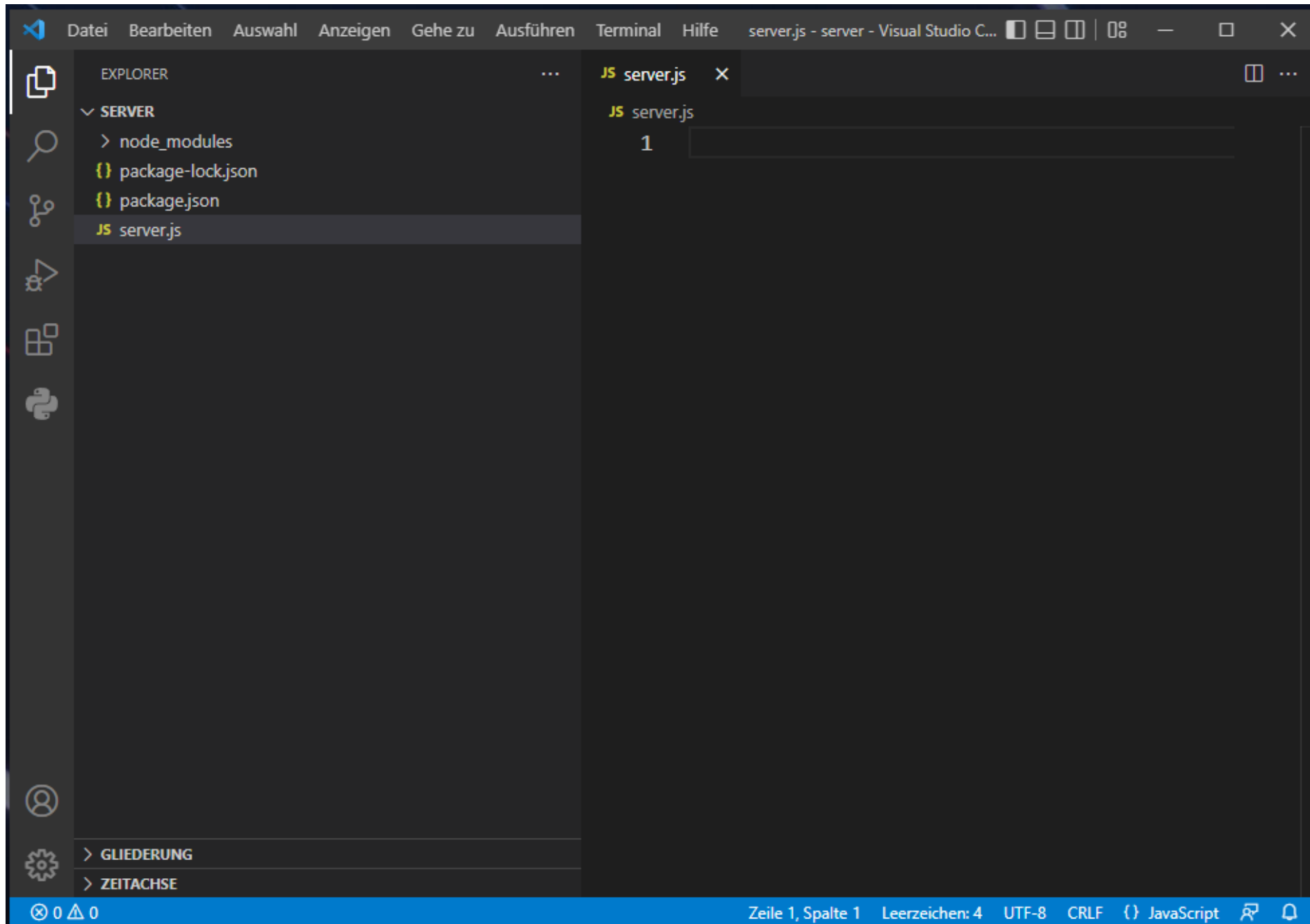
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

- How to use JWT / save JWTs in the browser or website
  - JWTs which you receive from a server or api can be saved in different ways in the website/browser each with their pros and cons
  - The token can be saved in the **windows.sessionStorage**. After the tab is closed the JWT token is deleted. The token can also be stored in the **windows.localStorage**, even after closing the tabs the token is still there.
    - Pro:
      - Easy access in JavaScript
    - Con:
      - Vulnerable to XSS attacks
  - The token can be saved in a HTTPOnly cookie. The cookie is set on the api/server backend and there is not a way to access the token through JavaScript. To save data in a HTTPOnly cookie the server need to set the cookie flag: httpOnly while creating the cookie
    - Pro:
      - Easy to set, send with every request to the server without the developer doing anything
      - Secure against XSS attacks (only if HTTPOnly flag is set to true)
    - Con:
      - No access through JavaScript

# Simple backend in Node.js and express

- Preparing:
  - Create a new Node.js project „server“ (1. create directory „server“ 2. open your command line tool and navigate to your project directory 3. execute **npm init** in the command line tool)
  - In this project we install the following packages via your command line tool:
    - cors (npm i cors)
    - body-parser (npm i body-parser)
    - cookie-parser (npm i cookie-parser)
    - express (npm i express)
    - jsonwebtoken (npm i jsonwebtoken)
    - crypto-js (npm i crypto-js)
  - We create a file „server.js“ in this directory
  - We download and install the software „postman“
    - With this software you are able to test your API without the need for a GUI/website
    - <https://www.postman.com/downloads/>

The project in Visual Studio Code should look like this



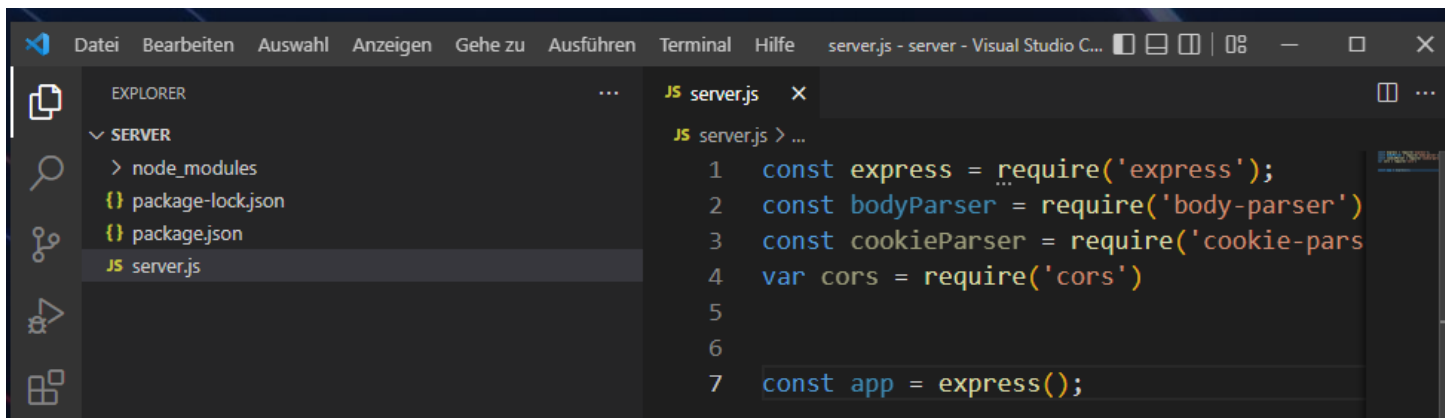


In the next steps we will import the packages installed before. For this open the server.js file and import express, body-parser, cookie-parser, cors

After importing the required packages, we create an express object with the line: **const app = express();**

### Important:

For a simple express web server there is actually no need for any packages as Node.js provides this out of the box. Most used package for web servers is still **express** as it's provides a simple interface and useful functions, it's also the only needed package for simple web server.



```
1  const express = require('express');
2  const bodyParser = require('body-parser')
3  const cookieParser = require('cookie-pars
4  var cors = require('cors')
5
6
7  const app = express();
```

- What is express and what does the **express()** function do?
  - It creates an express object and with this object we get access to all the functionality express.js provides us. An Express application is essentially a series of **middleware function calls**
  - A middleware is a **chain of function calls** before the response is sent back to the client.
  - A middleware has access to **request/response object** and the next **middleware function** in the application request-response cycle.
  - We are able to use several middlewares that help us make routes secure (authentication is required for routes). Express also allows us to perform some checks before a response is sent back to the client.
  - Some other tasks a middleware can perform are:
    - **Execute any code**
    - **Make changes to the request and the response object**
    - **End the request-response cycle**
    - **Call the next middleware function in the stack**
  - If the current middleware function **does not end the request-response cycle**, it must call **next()** to pass control to the next middleware function. Otherwise the request will be left hanging

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cookieParser = require('cookie-parser');
4  var cors = require('cors')
5
6
7  const app = express();
8
9
10 const port = 3005;
11 app.listen(port, ()=>{
12     console.log(`Example server listening at
13     |         |         |         |         |
14     |         |         |         |         |
15     |         |         |         |         |
16     |         |         |         |         |
17     |         |         |         |         |
18     |         |         |         |         |
19     |         |         |         |         |
20     |         |         |         |         |
21     |         |         |         |         |
22     |         |         |         |         |
23     |         |         |         |         |
24     |         |         |         |         |
25     |         |         |         |         |
26     |         |         |         |         |
27     |         |         |         |         |
28     |         |         |         |         |
29     |         |         |         |         |
30     |         |         |         |         |
31     |         |         |         |         |
32     |         |         |         |         |
33     |         |         |         |         |
34     |         |         |         |         |
35     |         |         |         |         |
36     |         |         |         |         |
37     |         |         |         |         |
38     |         |         |         |         |
39     |         |         |         |         |
40     |         |         |         |         |
41     |         |         |         |         |
42     |         |         |         |         |
43     |         |         |         |         |
44     |         |         |         |         |
45     |         |         |         |         |
46     |         |         |         |         |
47     |         |         |         |         |
48     |         |         |         |         |
49     |         |         |         |         |
50     |         |         |         |         |
51     |         |         |         |         |
52     |         |         |         |         |
53     |         |         |         |         |
54     |         |         |         |         |
55     |         |         |         |         |
56     |         |         |         |         |
57     |         |         |         |         |
58     |         |         |         |         |
59     |         |         |         |         |
60     |         |         |         |         |
61     |         |         |         |         |
62     |         |         |         |         |
63     |         |         |         |         |
64     |         |         |         |         |
65     |         |         |         |         |
66     |         |         |         |         |
67     |         |         |         |         |
68     |         |         |         |         |
69     |         |         |         |         |
70     |         |         |         |         |
71     |         |         |         |         |
72     |         |         |         |         |
73     |         |         |         |         |
74     |         |         |         |         |
75     |         |         |         |         |
76     |         |         |         |         |
77     |         |         |         |         |
78     |         |         |         |         |
79     |         |         |         |         |
80     |         |         |         |         |
81     |         |         |         |         |
82     |         |         |         |         |
83     |         |         |         |         |
84     |         |         |         |         |
85     |         |         |         |         |
86     |         |         |         |         |
87     |         |         |         |         |
88     |         |         |         |         |
89     |         |         |         |         |
90     |         |         |         |         |
91     |         |         |         |         |
92     |         |         |         |         |
93     |         |         |         |         |
94     |         |         |         |         |
95     |         |         |         |         |
96     |         |         |         |         |
97     |         |         |         |         |
98     |         |         |         |         |
99     |         |         |         |         |
100    |         |         |         |         |
101    |         |         |         |         |
102    |         |         |         |         |
103    |         |         |         |         |
104    |         |         |         |         |
105    |         |         |         |         |
106    |         |         |         |         |
107    |         |         |         |         |
108    |         |         |         |         |
109    |         |         |         |         |
110    |         |         |         |         |
111    |         |         |         |         |
112    |         |         |         |         |
113    |         |         |         |         |
114    |         |         |         |         |
115    |         |         |         |         |
116    |         |         |         |         |
117    |         |         |         |         |
118    |         |         |         |         |
119    |         |         |         |         |
120    |         |         |         |         |
121    |         |         |         |         |
122    |         |         |         |         |
123    |         |         |         |         |
124    |         |         |         |         |
125    |         |         |         |         |
126    |         |         |         |         |
127    |         |         |         |         |
128    |         |         |         |         |
129    |         |         |         |         |
130    |         |         |         |         |
131    |         |         |         |         |
132    |         |         |         |         |
133    |         |         |         |         |
134    |         |         |         |         |
135    |         |         |         |         |
136    |         |         |         |         |
137    |         |         |         |         |
138    |         |         |         |         |
139    |         |         |         |         |
140    |         |         |         |         |
141    |         |         |         |         |
142    |         |         |         |         |
143    |         |         |         |         |
144    |         |         |         |         |
145    |         |         |         |         |
146    |         |         |         |         |
147    |         |         |         |         |
148    |         |         |         |         |
149    |         |         |         |         |
150    |         |         |         |         |
151    |         |         |         |         |
152    |         |         |         |         |
153    |         |         |         |         |
154    |         |         |         |         |
155    |         |         |         |         |
156    |         |         |         |         |
157    |         |         |         |         |
158    |         |         |         |         |
159    |         |         |         |         |
160    |         |         |         |         |
161    |         |         |         |         |
162    |         |         |         |         |
163    |         |         |         |         |
164    |         |         |         |         |
165    |         |         |         |         |
166    |         |         |         |         |
167    |         |         |         |         |
168    |         |         |         |         |
169    |         |         |         |         |
170    |         |         |         |         |
171    |         |         |         |         |
172    |         |         |         |         |
173    |         |         |         |         |
174    |         |         |         |         |
175    |         |         |         |         |
176    |         |         |         |         |
177    |         |         |         |         |
178    |         |         |         |         |
179    |         |         |         |         |
180    |         |         |         |         |
181    |         |         |         |         |
182    |         |         |         |         |
183    |         |         |         |         |
184    |         |         |         |         |
185    |         |         |         |         |
186    |         |         |         |         |
187    |         |         |         |         |
188    |         |         |         |         |
189    |         |         |         |         |
190    |         |         |         |         |
191    |         |         |         |         |
192    |         |         |         |         |
193    |         |         |         |         |
194    |         |         |         |         |
195    |         |         |         |         |
196    |         |         |         |         |
197    |         |         |         |         |
198    |         |         |         |         |
199    |         |         |         |         |
200    |         |         |         |         |
201    |         |         |         |         |
202    |         |         |         |         |
203    |         |         |         |         |
204    |         |         |         |         |
205    |         |         |         |         |
206    |         |         |         |         |
207    |         |         |         |         |
208    |         |         |         |         |
209    |         |         |         |         |
210    |         |         |         |         |
211    |         |         |         |         |
212    |         |         |         |         |
213    |         |         |         |         |
214    |         |         |         |         |
215    |         |         |         |         |
216    |         |         |         |         |
217    |         |         |         |         |
218    |         |         |         |         |
219    |         |         |         |         |
220    |         |         |         |         |
221    |         |         |         |         |
222    |         |         |         |         |
223    |         |         |         |         |
224    |         |         |         |         |
225    |         |         |         |         |
226    |         |         |         |         |
227    |         |         |         |         |
228    |         |         |         |         |
229    |         |         |         |         |
230    |         |         |         |         |
231    |         |         |         |         |
232    |         |         |         |         |
233    |         |         |         |         |
234    |         |         |         |         |
235    |         |         |         |         |
236    |         |         |         |         |
237    |         |         |         |         |
238    |         |         |         |         |
239    |         |         |         |         |
240    |         |         |         |         |
241    |         |         |         |         |
242    |         |         |         |         |
243    |         |         |         |         |
244    |         |         |         |         |
245    |         |         |         |         |
246    |         |         |         |         |
247    |         |         |         |         |
248    |         |         |         |         |
249    |         |         |         |         |
250    |         |         |         |         |
251    |         |         |         |         |
252    |         |         |         |         |
253    |         |         |         |         |
254    |         |         |         |         |
255    |         |         |         |         |
256    |         |         |         |         |
257    |         |         |         |         |
258    |         |         |         |         |
259    |         |         |         |         |
260    |         |         |         |         |
261    |         |         |         |         |
262    |         |         |         |         |
263    |         |         |         |         |
264    |         |         |         |         |
265    |         |         |         |         |
266    |         |         |         |         |
267    |         |         |         |         |
268    |         |         |         |         |
269    |         |         |         |         |
270    |         |         |         |         |
271    |         |         |         |         |
272    |         |         |         |         |
273    |         |         |         |         |
274    |         |         |         |         |
275    |         |         |         |         |
276    |         |         |         |         |
277    |         |         |         |         |
278    |         |         |         |         |
279    |         |         |         |         |
280    |         |         |         |         |
281    |         |         |         |         |
282    |         |         |         |         |
283    |         |         |         |         |
284    |         |         |         |         |
285    |         |         |         |         |
286    |         |         |         |         |
287    |         |         |         |         |
288    |         |         |         |         |
289    |         |         |         |         |
290    |         |         |         |         |
291    |         |         |         |         |
292    |         |         |         |         |
293    |         |         |         |         |
29
```

```
JS server.js > ...
3 const cookieParser = require('cookie-parser');
4 var cors = require('cors');
5 const app = express();
6
7 app.get('/', (request, response) => {
8     response.status(200).send("Hello World!");
9 });
10
11 app.post('/add', (req, res) => {
12     res.status(200).send("New item added!");
13 });
14
15 app.put('/change', (req, res) => {
16     res.status(200).send("Item updated!");
17 });
18
19 app.delete('/delete', (req, res) => {
20     res.status(200).send("Item deleted!");
21 });
22
23 const port = 3005;
24 app.listen(port, () => {
25     console.log(`Example server listening at
26     |         |         |         |
27     |         |         |         | http://localhost:${port}`)
28 });
```

Zeile 6, Spalte 1   Leerzeichen: 4   UTF-8   CRLF   {} Java

As we can see, we use the **app** object and the http-request methods (GET,POST,PUT,DELETE) are function in the **app** object.

The first parameter of these functions is the **path**. If you make a http-request to the server and the path is correct the corresponding route on the server is executed.

Second parameter as shown here is a **callback** function, which will be executed **after** the correct route was found. The first parameter in the callback function is the **request** object and the second parameter is the **response** object.

To complete the **request-response cycle**, we send a response back to the client. As we can see here, we set first call function to set a **status** for our response and after that we call the **send** function where we can pass our response data: (string,json,etc.)

With these routes we created our first API server although it can't really do much.

```

PROBLEME  AUSGABE  DEBUGGING-KONSOLE  TERMINAL  JUPYTER  node + - [ ] [ ] ^ x

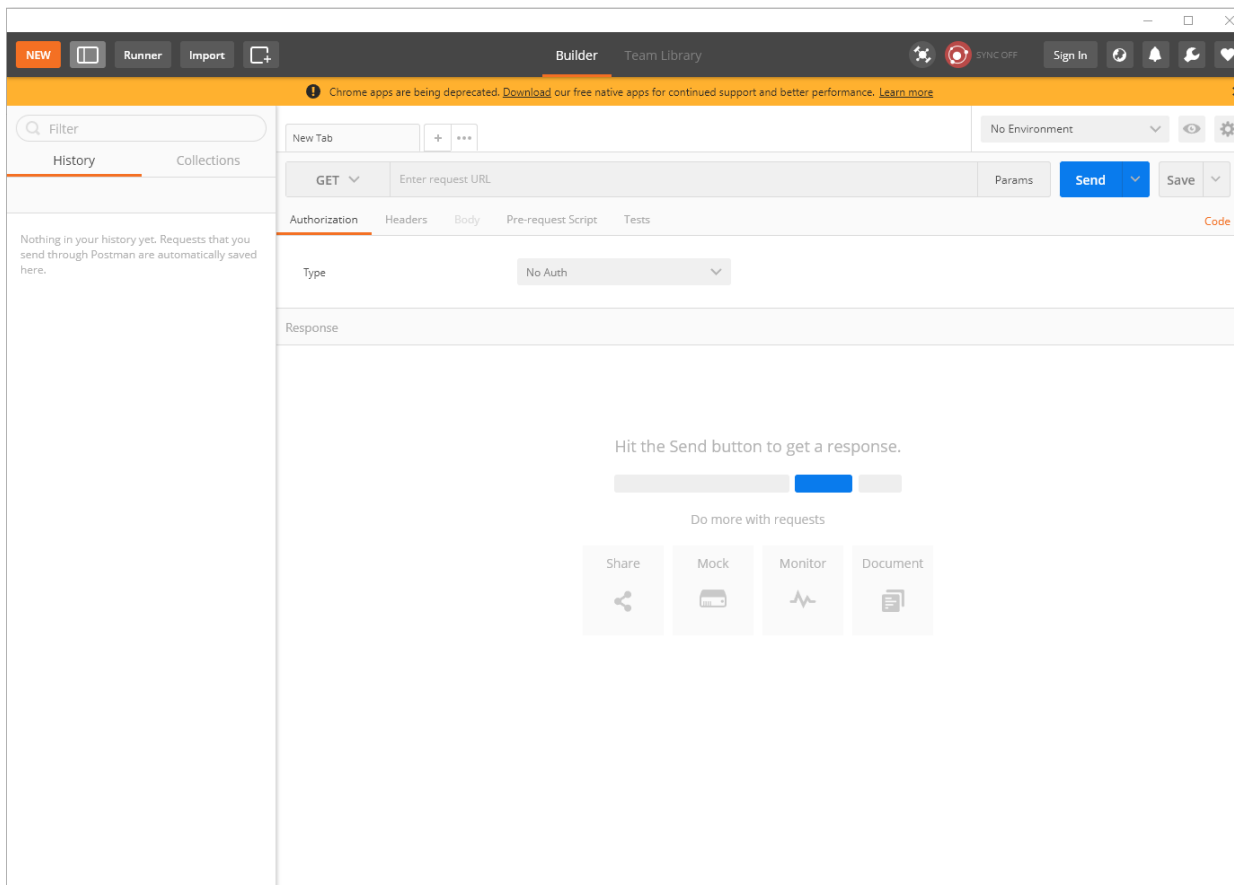
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Lernen Sie das neue plattformübergreifende PowerShell kennen - https://aka.ms/pscore6

PS F:\GitHub\ibs_lecture\code\server> node server.js
Example server listening at http://localhost:3005

```

To start our server we simply run **node server.js** in the terminal in our „server“ directory. After that we should see in terminal the output, that the server is listening to port 3005. Our server is now ready to receive requests.



If we open postman and close all the pop-ups, we should see something like in the picture.

1. Select HTTP-Request Methode
2. URL of our server (API endpoint)

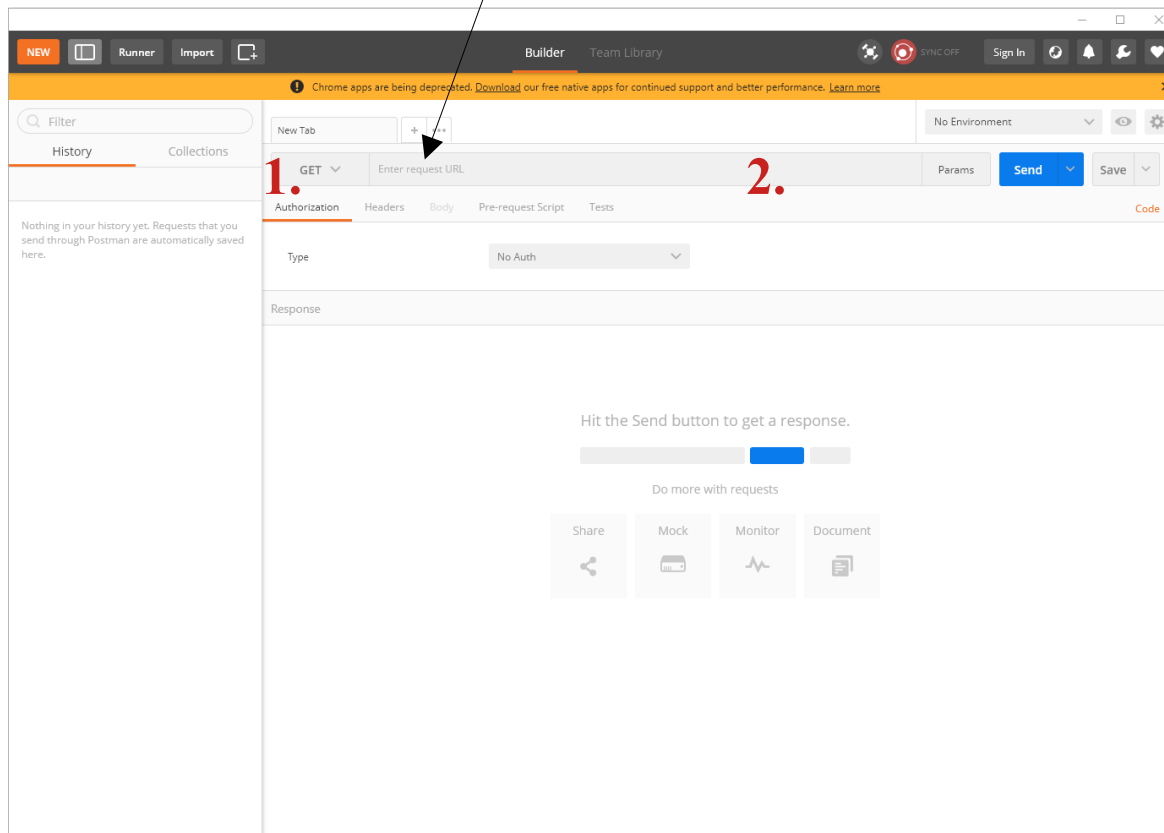
GET ▾

Enter request URL

Params

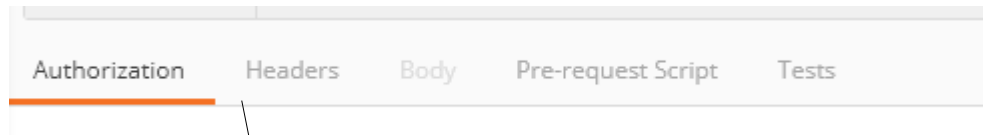
Send ▾

Save ▾

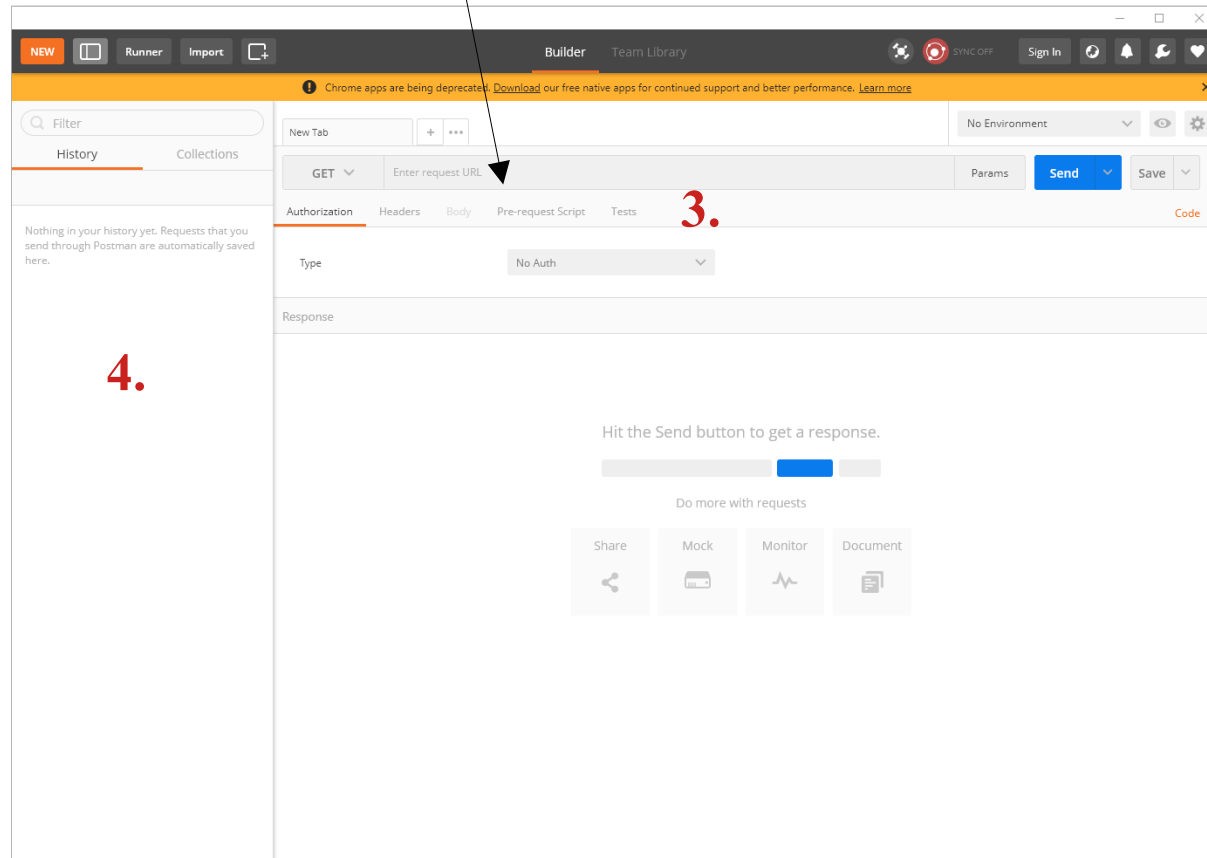


The screenshot shows the Postman application interface. A black arrow points from the '1.' annotation to the 'GET' dropdown menu. Another black arrow points from the '2.' annotation to the 'Enter request URL' input field. The interface includes a top bar with 'NEW', 'Runner', 'Import', and 'Builder' tabs. Below the top bar is a search filter and a 'History' section. The main area is divided into tabs: 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Authorization' tab is selected, showing a 'Type' dropdown set to 'No Auth'. Below the tabs is a 'Response' section with a message 'Hit the Send button to get a response.' and a progress bar. At the bottom, there are buttons for 'Share', 'Mock', 'Monitor', and 'Document'.

### 3. Different options to send data with our request

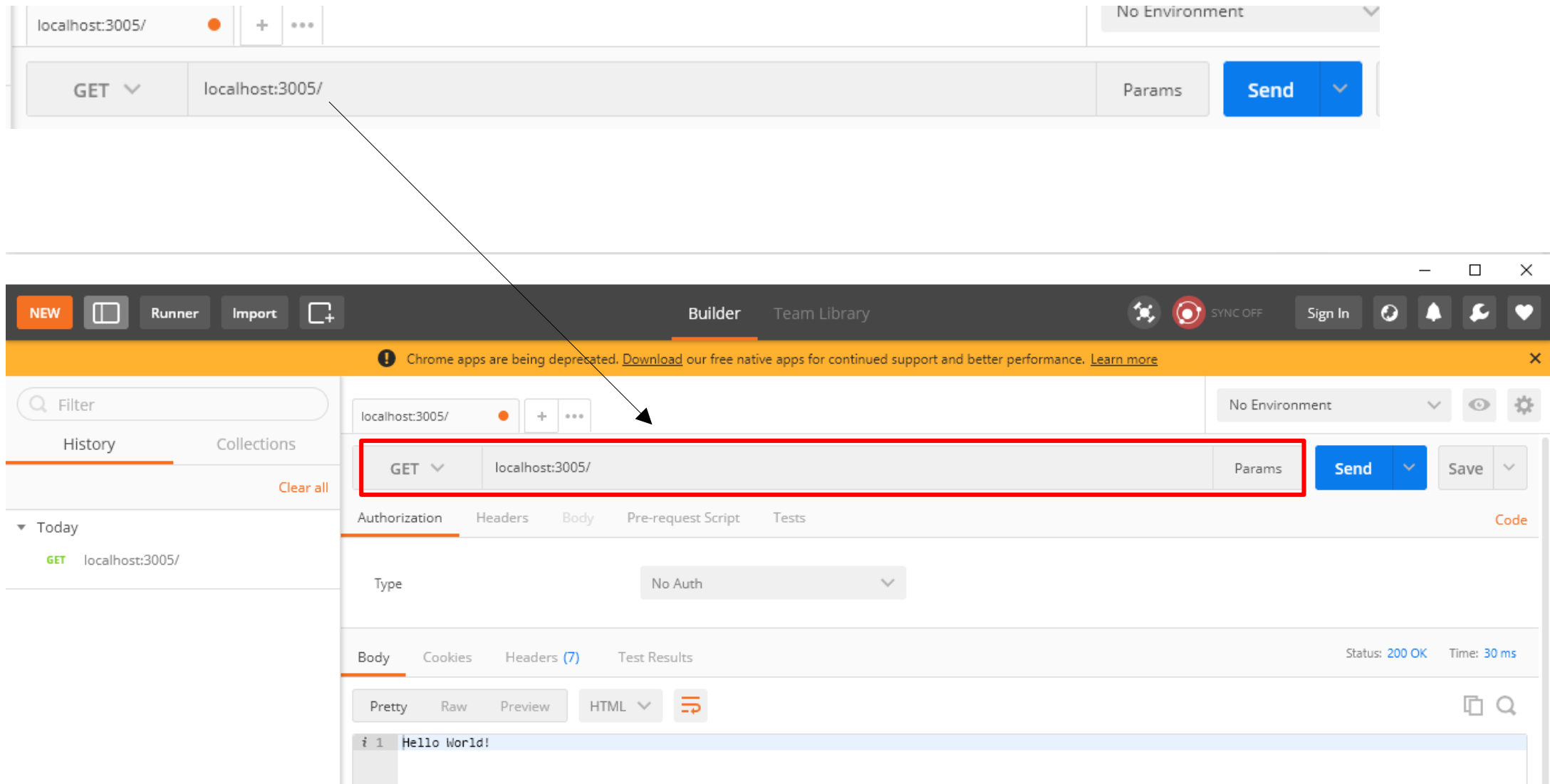


### 4. History of made request





- To test our server we make a simple GET request with postman to the server.
- As we can see in the Body field, the server response is the expected string „**Hello World!**“
- To test the other routes we simply have to change the method and URL and press the send button.



The image shows a screenshot of the Postman application interface. At the top, there is a tab for 'localhost:3005/'. Below this, a request is configured with the method 'GET' and the URL 'localhost:3005/'. The 'Send' button is visible. A red box highlights the request configuration area. Below the request configuration, the 'Body' tab is selected, showing the response 'Hello World!'. The status bar at the bottom right indicates 'Status: 200 OK' and 'Time: 30 ms'.

Next steps:

- 1) We have now a simple API server running and tested the endpoints, let's extend the server with more functionality
- 2) For the extension of our server, we create a simple JavaScript object, which will be subject to change if a specific request was made
- 3) We will specify which URL is allowed to access our API and what kind of data our API expects to receive
- 4) We will also extend the routes, some routes will expect query parameter and some expect data send within the body of a request
- 5) Last but not least we will see how our server creates a cookie as well JWT token and how we can secure some routes (although we do not use it)

## Overview Server.js

```
JS server.js X
JS server.js > ...
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cookieParser = require('cookie-parser');
4  var cors = require('cors');
5  const app = express();
6
7  //JavaScript Object which we will change with routes
8  const data = {};
9
10 //We specify which URL is allowed to make request to our API Server
11 //Origin specifies which url is allowed to make request
12 //credentials specifies that header not omitted and is passed on
13 app.use(cors({
14     origin: "http://localhost:3000",
15     credentials: true
16 })))
17
18 // parse application/x-www-form-urlencoded
19 app.use(bodyParser.urlencoded({ extended: false }))
20 // parse application/json
21 app.use(bodyParser.json())
22 app.use(cookieParser());
23
```

1. Is our JavaScript object which will be changed through our routes
2. Here we use the **cors** package and the **cors()** function with which we specify which origin-url is allowed to make request to our api-server and we specify that the header is passed on

```
JS server.js  X
JS server.js > ...
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cookieParser = require('cookie-parser');
4  var cors = require('cors');
5  const app = express();
6
7  //JavaScript Object which we will change with routes
8  const data = {}; 1.
9
10 //We specify which URL is allowed to make request to our API Server
11 //Origin specifies which url is allowed to make request
12 //credentials specifies that header not omitted and is passed on
13 app.use(cors({
14   origin: "http://localhost:3000", 2.
15   credentials: true
16 }));
17
```

3. We use the **body-parser** package/function and specify in which way the body of request are formatted and how the request body should be encoded (**bodyparser.urlencoded([options])**)

1. The extended option allows to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true). The "extended" syntax allows for rich objects and arrays to be encoded into the URL-encoded format, allowing for a JSON-like experience with URL-encoded. For more information, please see the qs library.
2. Defaults to true, but using the default has been deprecated. Please research into the difference between qs and querystring and choose the appropriate setting.

4. Here we make use of the **cookieParser** library so, that cookies which are sent with the request are automatically parsed to an JavaScript object and are available within the receive object of the request.

```
17 |  
18 // parse application/x-www-form-urlencoded  
19 app.use(bodyParser.urlencoded({ extended: false })) 3.  
20 // parse application/json  
21 app.use(bodyParser.json())  
22 app.use(cookieParser()); 4.  
23
```

As we could see from 2. - 4., to add functionality to our express server, we use **app.use()**. With this function we can add functions to the **middleware life-cycle**.

If we add functions to the middleware via **app.use()**, the middleware functions are executed for every request made to our api-server.

Later we will see how we can add middleware functions which target only specific routes.

```
JS server.js  X
JS server.js > ...
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cookieParser = require('cookie-parser');
4  var cors = require('cors');
5  const app = express();
6
7  //JavaScript Object which we will change with routes
8  const data = {}; 1
9
10 //We specify which URL is allowed to make request to our API Server
11 //Origin specifies which url is allowed to make request
12 //credentials specifies that header not omitted and is passed on
13 app.use(cors({
14     origin: "http://localhost:3000", 2
15     credentials: true
16 }));
17
18 // parse application/x-www-form-urlencoded
19 app.use(bodyParser.urlencoded({ extended: false })); 3
20 // parse application/json
21 app.use(bodyParser.json());
22 app.use(cookieParser()); 4
23
```

- What is the **cors** package and what other functions does it have? (npmjs-cors)
  - enables express middleware to use different **CORS** options
  - **app.use(cors())** //enables all CORS Requests
  - Options that can be used in the **cors** function are:
    - **origin**: Possible value (bool, string, regex, array, function)
      - Bool: **true**=to reflect request origin, **false**=to disable cors
      - String: to set origin to **specific origin** (e.g. example.com)
      - Array: set origin to **an array of valid origin** (String/RegExp) (e.g. ["http://example1.com", /\.example2\.com\$/])
    - **methods**: Configure **Access-Control-Allow-Methods** Cors header (e.g. „GET,POST“ or [„GET“, „POST“, ...])
    - **allowedHeaders**: Configures the **Access-Control-Allow-Headers** CORS header. Expects a comma-delimited string (ex: 'Content-Type,Authorization') or an array (ex: ['Content-Type', 'Authorization']).
    - **exposedHeaders**: Configures the **Access-Control-Expose-Headers** CORS header. Expects a comma-delimited string (ex: 'Content-Range,X-Content-Range') or an array (ex: ['Content-Range', 'X-Content-Range']). If not specified, no custom headers are exposed.
    - **credentials**: Configures the **Access-Control-Allow-Credentials** CORS header. Set to true to pass the header, otherwise it is omitted.
    - **maxAge**: Configures the **Access-Control-Max-Age** CORS header. Set to an integer to pass the header, otherwise it is omitted.
    - **preflightContinue**: Pass the CORS preflight response to the next handler.
    - **optionsSuccessStatus**: Provides a status code to use for successful OPTIONS requests, since some legacy browsers (IE11, various SmartTVs) choke on 204.

Default configuration is:

```
{  
  "origin": "*",  
  "methods": "GET,HEAD,PUT,PATCH,POST,DELETE",  
  "preflightContinue": false,  
  "optionsSuccessStatus": 204  
}
```



- What is the **body-parser** package and what other functions does it have?(npmjs-body-parser)
  - Parse incoming request bodies in a middleware before your handlers, available under the req.body property.
  - To use this package we need the imported **body-parser** object (in our case we named the object **bodyparser**) and execute the parser functions we want to use.  
(e.g.**app.use(bodyParser.json())**)
  - This packages can parse the following request bodies:
    - **JSON Body**
    - **Raw Body**
    - **Text Body**
    - **URL-encoded form body**
  - **bodyParser.json([options]):**
    - Returns middleware that only parses **json** and only looks at requests where the Content-Type header matches the type option.

## Options:

- **inflate:** true=deflated (compressed) bodies will be inflated, false=deflated bodies are rejected, (**default=true**)
- **limit:** controls the maximum request body size. If number=bytes (passed to bytes library for parsing), default =‘100kb’
- **reviver:** The reviver option is passed directly to JSON.parse as the second argument
- **strict:** When set to true, will only accept arrays and objects; when false will accept anything JSON.parse accepts. Defaults to true
- **type:** The type option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function
- **verify:** The verify option, if supplied, is called as verify(req, res, buf, encoding), where buf is a Buffer of the raw request body and encoding is the encoding of the request. The parsing can be aborted by throwing an error.

- As as our api-server should be able to do more than return simple string to the client, we first have to slightly modify our JavaScript object link in the picture. With this we can save books in this object

```
//JavaScript Object which we will change with routes
/*{
  title:"test",
  isbn:"1234567891",
  author:"Jonny Trip"
}*/
const data ={books:[]};
```

- The next change is done to our GET route as we want to **return all books** or **all books from a specific author**. To return all books from a specific author we use **URL queries** (e.g. localhost:3005/books?author=Jonny Trip)
- Express let's us access the **query parameter** fairly easy, as we simple use the **request** object and the **query** property. Within the query property are all url query parameter available.
- After we have the author url query parameter we simply iterate over the books array and fill a new array with all books form the author and return this array to the client.
- Or if no author url query parameter was used, we simply return the whole array of books to the client

```
app.get('/books',(request, response)=>{
  if(request.query.author){
    let booksAuthor = [];
    for(let book of data.books){
      if(book.author == request.query.author){
        booksAuthor.push(book);
      }
    }
    response.status(200).send(booksAuthor)
  }else{
    response.status(200).send(data.books)
  }
});
```

- There is also another way to filter data with a GET request and this is to use **parameters directly** in the URL as shown in the picture. (e.g. localhost:3005/books/**1234567891**)
- Here the **:isbn** is a parameter in the URL and we get access to it through the **request** object and the **params** property.
- After we have our **isbn** parameter we simply look in the books array after the book with the correct isbn. We use here a JavaScript function map, that will return us an index  $\geq 0$  or if nothing is found an index of -1;

```
app.get('/books/:isbn', (request, response) => {  
  const isbn = request.params.isbn;  
  if (isbn.length == 10 || isbn.length == 13) {  
    let searchedBookIndex = data.books.findIndex((v) => v.isbn == isbn)  
    if (searchedBookIndex != -1) {  
      let searchedBook = {...data.books[searchedBookIndex]};  
      response.status(200).send(searchedBook)  
    } else {  
      response.status(400).send(`No book found with the isbn ${isbn}`)  
    }  
  } else {  
    response.status(400).send("error")  
  }  
});
```

- In our POST route the **data** is in the **request body** and is already formatted in **JSON** through our middleware **bodyParser.json()**
- We save the received JSON object in a new JavaScript object and make some checks if everything is alright with the data we received from the client. After that we add the new book.

```
app.post('/book', (req,res)=>{
  let newBook = req.body;
  if(newBook.author != ''
    && (newBook.isbn.length==10 || newBook.isbn.length ==13)
    && newBook.title != ''){
    data.books.push({
      title:newBook.title,
      isbn:newBook.isbn,
      author:newBook.author
    })
    res.status(200).send("New item added!");
  }else{
    res.status(400).send("data have the wrong format"+
      " or are not complete");
  }
})
```

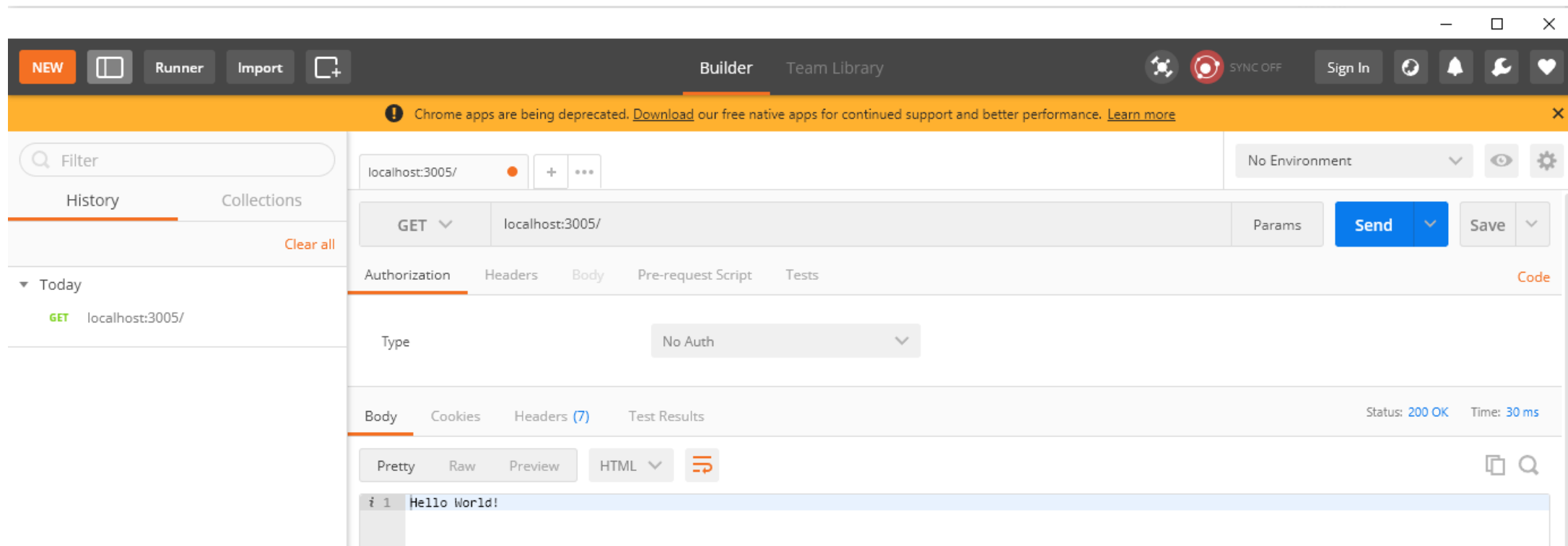
- In our PUT route we do the same as in our POST route with the difference, that after we do the checks, we first search for the book we want to change. We use again the JavaScript **map** function that returns us the **index** of the book in the array. After we have the **index** we **change** the book.
- As we want to be able to change **everything** in the book we have to have a **new property** the client needs to send to us. This property is **isbn\_search** as we also want to be able to change the old isbn.

```
app.put('/book',(req,res)=>{
  let bookToChange = req.body;
  if(bookToChange.author != ''
    && (bookToChange.isbn.length==10 || bookToChange.isbn.length ==13)
    && (bookToChange.isbn_search.length==10 || bookToChange.isbn_search.length ==13)
    && bookToChange.title != ''){
    let searchedBookIndex = data.books.findIndex(
      (v)=>v.isbn == bookToChange.isbn_search)
    if(searchedBookIndex != -1){
      data.books[searchedBookIndex].title = bookToChange.title;
      data.books[searchedBookIndex].isbn = bookToChange.isbn;
      data.books[searchedBookIndex].author = bookToChange.author;
      res.status(200).send("Book was updated");
    }else{
      res.status(400).send("Book not found!");
    }
  }else{
    res.status(400).send("data have the wrong format"+
      " or are not complete");
  }
})
```

- In our DELETE route nothing new is happening. We get the **isbn** in the **body** of the **request object** and do some checks. We use again the **map** function to get the **index** of the book and afterwards we use the **splice** function to **delete** the book from the array.

```
app.delete('/book',(req,res)=>{
  const isbn = req.body.isbn
  if(isbn.length==10 || isbn.length ==13){
    let searchedBookIndex = data.books.findIndex((v)=>v.isbn == isbn)
    if(searchedBookIndex != -1){
      data.books.splice(searchedBookIndex,1)
      res.status(200).send("Book was deleted");
    }else{
      res.status(400).send("Book not found!");
    }
  }else{
    res.status(400).send("data have the wrong format"+
      " or are not complete");
  }
})
```

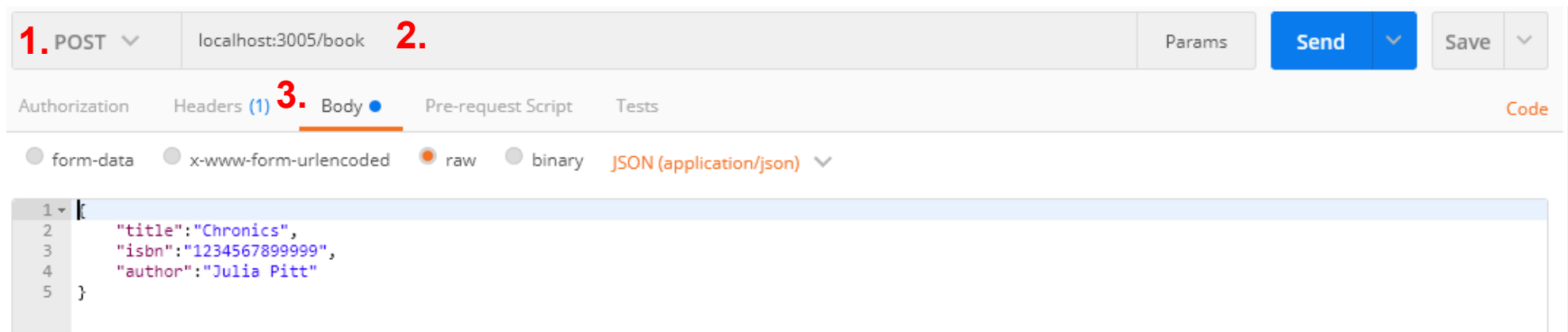


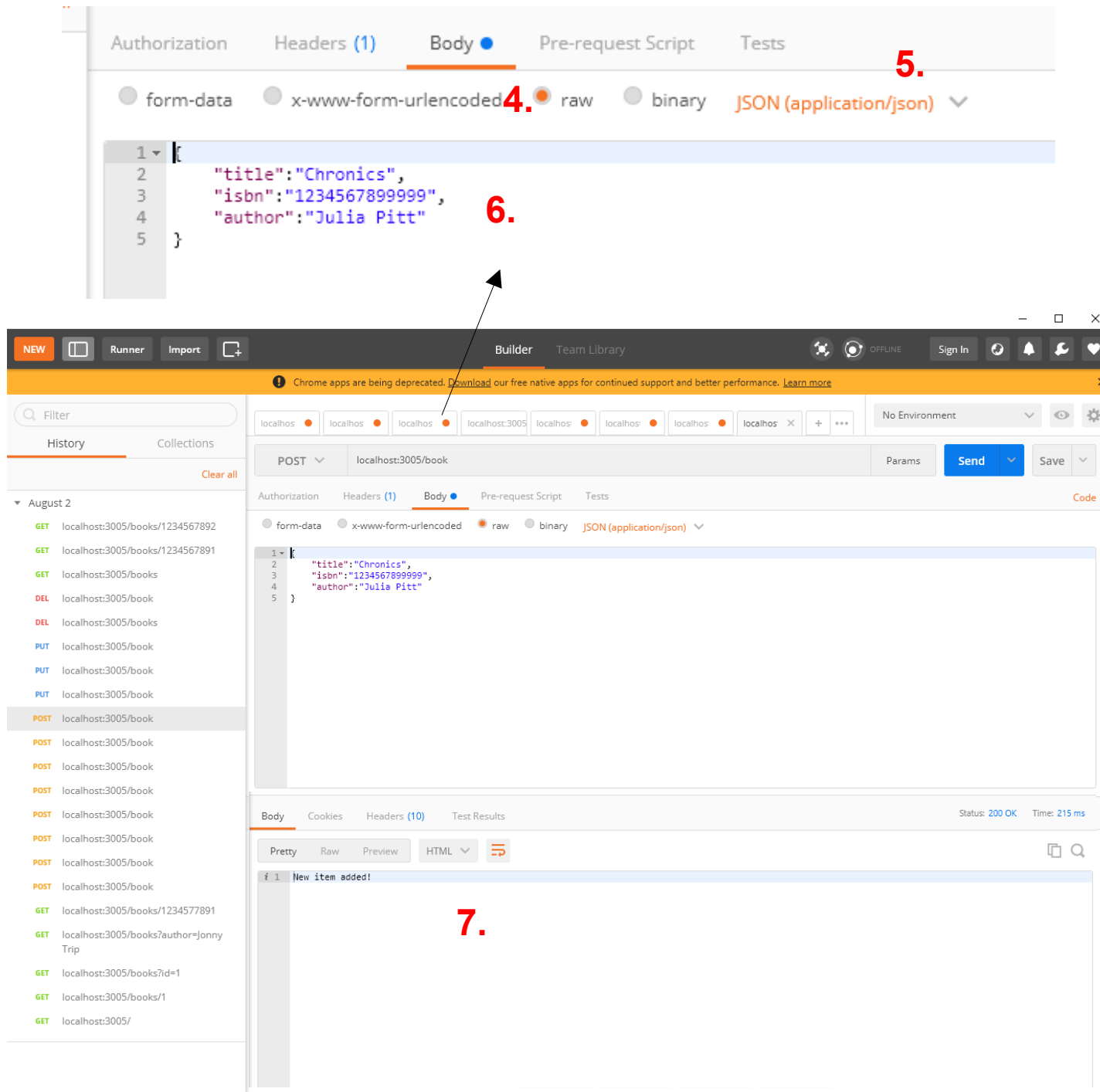


- To test our server with the new routes, we are using Postman again.
- This time we send JSON data with our requests in the body of the request.
- How we are doing this, will be shown in the following slides

## POST-Request in Postman:

- 1) Chose the **POST** Methode
- 2) Insert the right **URL**
- 3) Select the **body tab**





POST-Request in Postman:

4) Chose the **raw** option

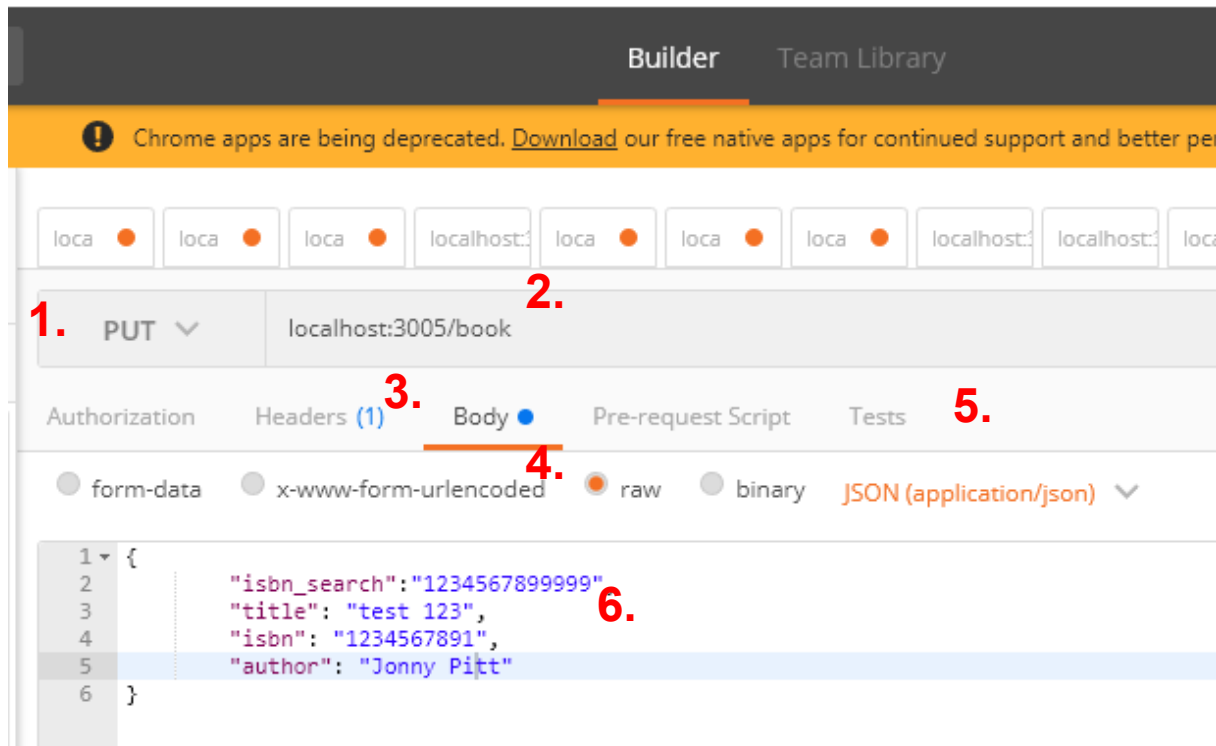
5) In the drop-down menu select **JSON (application/json)**

6) In the editor window write a **JSON object** like the ones the api-routes expects (e.g like in the picture)

7) If you press send a second windows should appear which should tell if either a new book was added or a error happend.

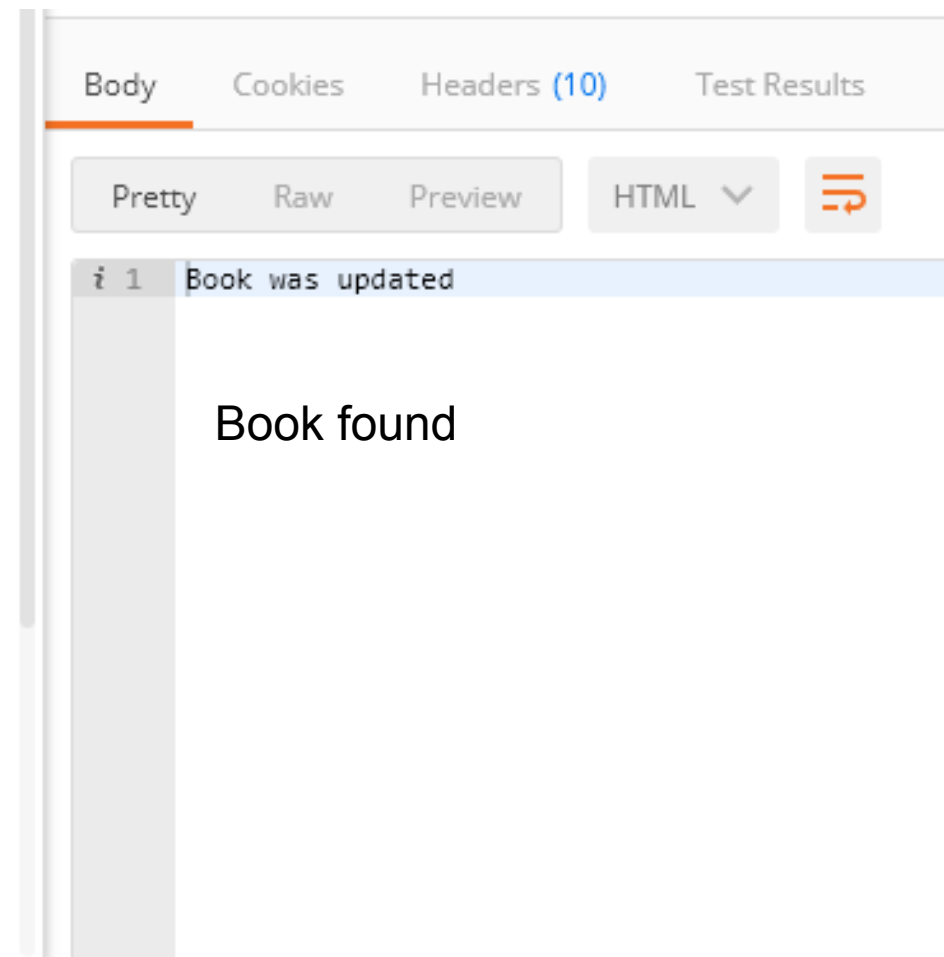
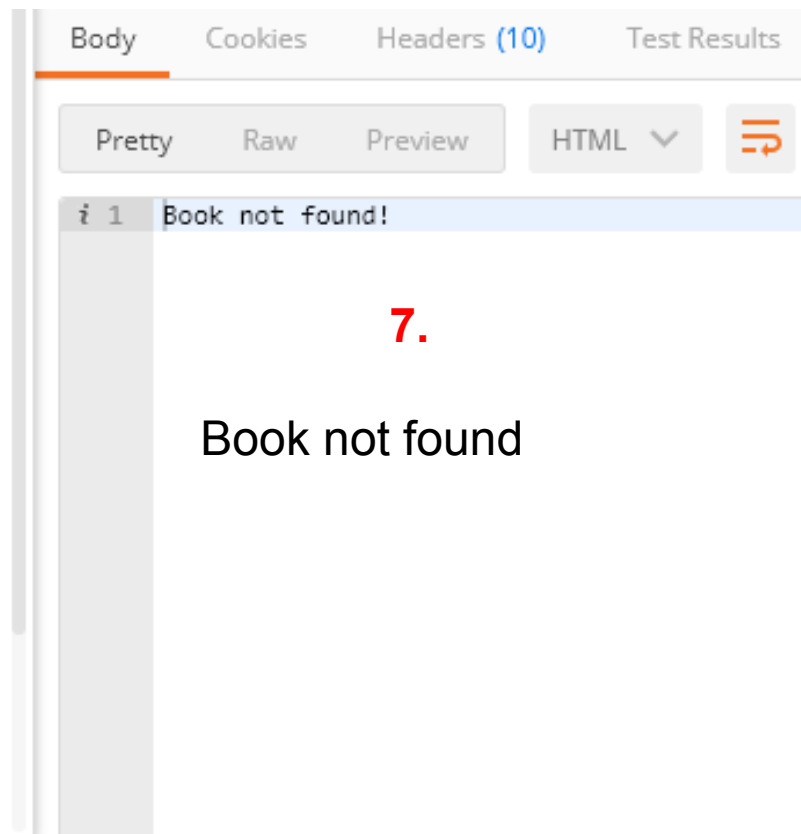
## PUT-Request in Postman:

- 1) Chose the **PUT** Methode
- 2) Insert the right **URL**
- 3) Select the **body** tab
- 4) Chose the **raw** option
- 5) In the drop-down menu select **JSON (application/json)**
- 6) In the editor window write a **JSON object** like the ones the api-routes expects (e.g like in the picture)



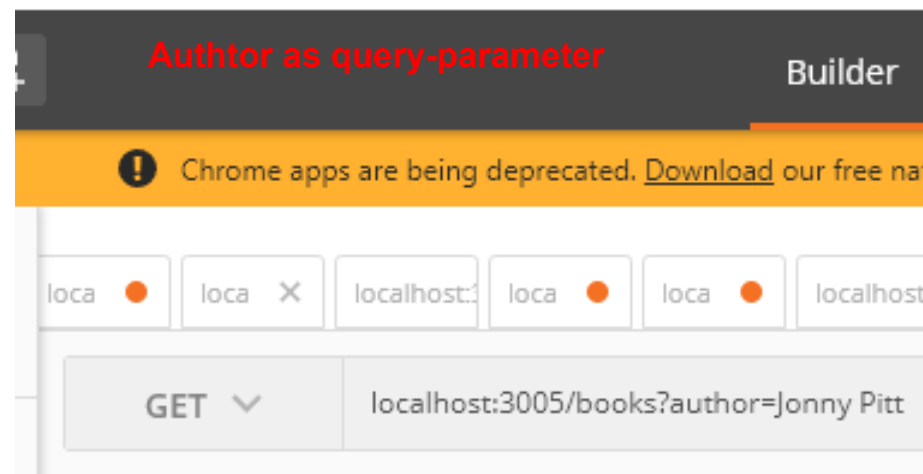
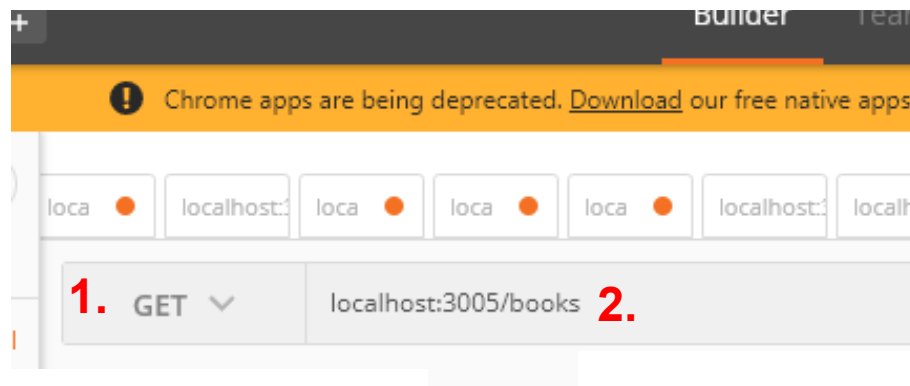
## PUT-Request in Postman:

7) If you press **send**, a second window should appear which should tell if either a book was updated or an error happened. (The book which should be updated needs to exist)



## GET-Request in Postman:

- 1) Chose the **GET** Methode
- 2) Insert the right **URL**
  - 1) The URL could either have **query-parameters** or **not** or expects **data within the URL** (like in the three different pictures)
- 3) If you press **send** a second windows should appear which should show the returned books



```

Pretty Raw Preview JSON ↕
1 [
2   {
3     "title": "test 123",
4     "isbn": "1234567891",
5     "author": "Jonny Pitt"
6   },
7   {
8     "title": "test 123",
9     "isbn": "1234567891",
10    "author": "Jonny aasdadasd"
11  }
12 ]

```

### Response object from URL with ISBN

Body Cookies Headers (10) Test Results

```

Pretty Raw Preview JSON ↕
1 {
2   "title": "test 123",
3   "isbn": "1234567891",
4   "author": "Jonny Pitt"
5 }

```

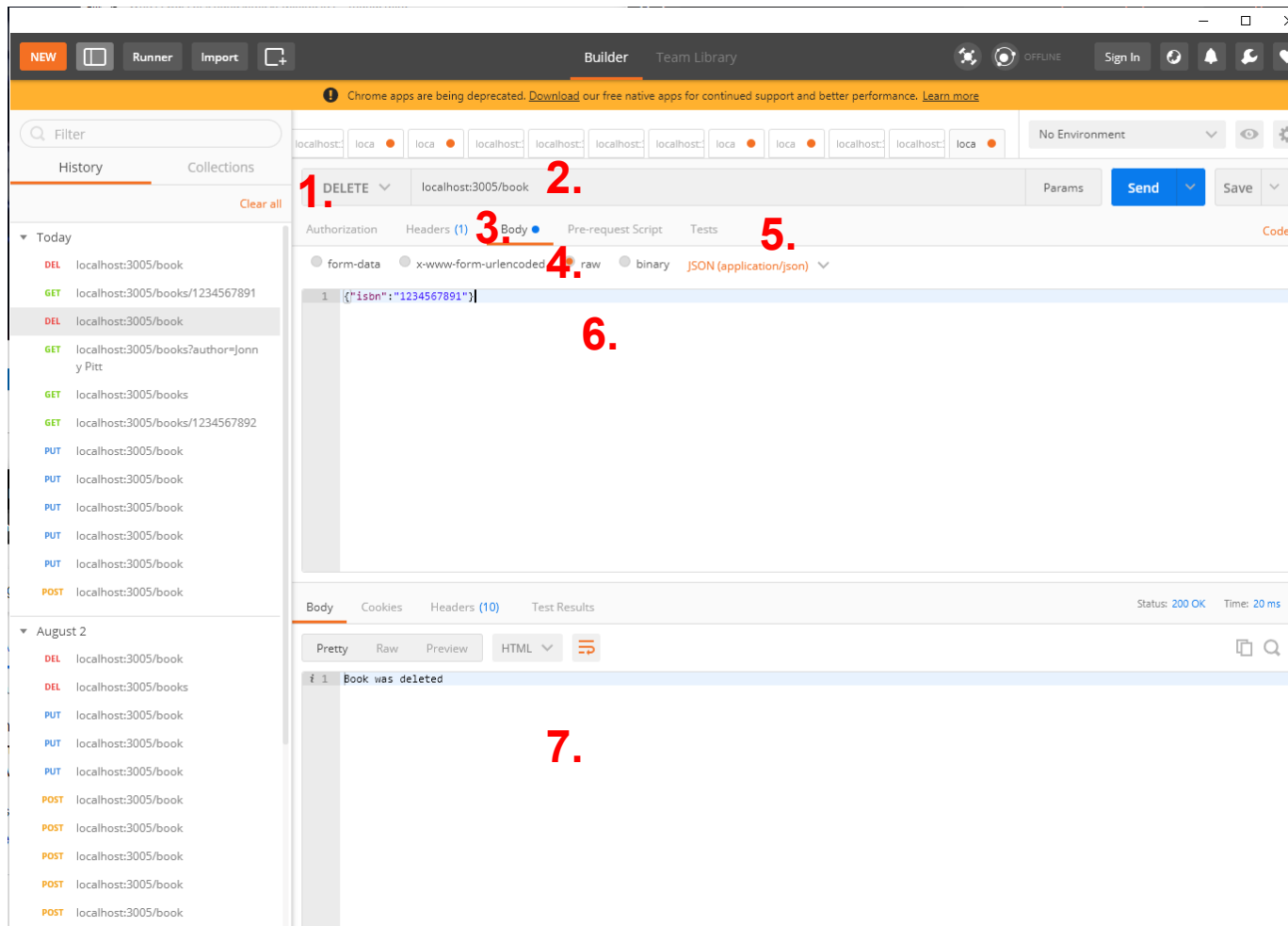
### Response object from URL with Author as query-parameter

Body Cookies Headers (10) Test Results

```

Pretty Raw Preview JSON ↕
1 [
2   {
3     "title": "test 123",
4     "isbn": "1234567891",
5     "author": "Jonny Pitt"
6   }
7 ]

```



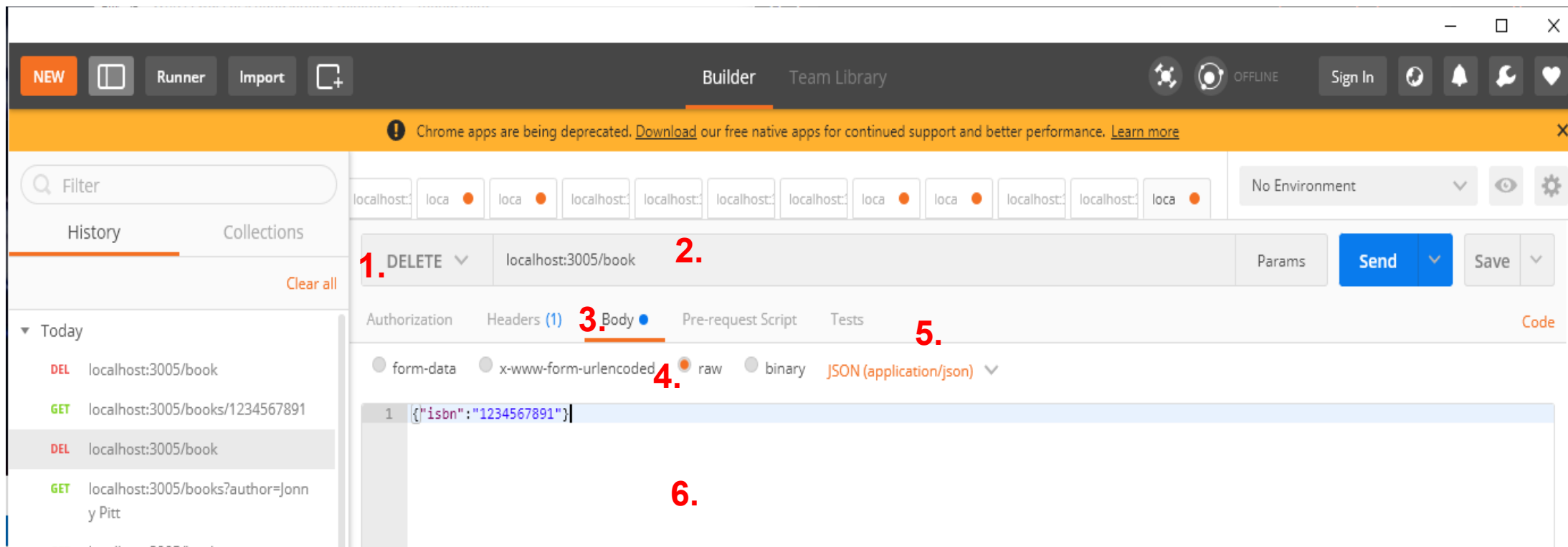
## DELETE-Request in Postman:

- 1) Chose the **DELETE** Methode
- 2) Insert the right **URL**
- 3) Select the **body** tab
- 4) Chose the **raw** option
- 5) In the drop-down menu select **JSON (application/json)**
- 6) In the editor window write a **JSON object** like the ones the api-routes expects (e.g like in the picture)
- 7) If you press send a second windows should appear which should tell if either a new book was added or a error happend.



## DELETE-Request in Postman:

- 1) Chose the **DELETE** Methode
- 2) Insert the right **URL**
- 3) Select the **body tab**
- 4) Chose the **raw** option
- 5) In the drop-down menu select **JSON (application/json)**
- 6) In the editor window write a **JSON object** like the ones the api-routes expects (e.g like in the picture)
- 7) If you press send a second windows should appear which should tell if either a new book was added or a error happend.



# Authentication/JWT/Cookies

- To use authentication in the backend we use in this example a cookie and jwt approach.
- The first thing we need to do is to write a function, which will generate a jwt token and cookie, when the client sign-up/in.
- The Node.js packages we need for this are:
  - **const jwt = require('jsonwebtoken');**
- In the function you can see below we create a **jwt token** save it in a **cookie** in the **response** object

```
function generateToken(res, email, id){  
  const expiration = 604800000;  
  const token = jwt.sign({email, id}, "986caae7d6a0a011e2dc2d0c2d5a8f2", {  
    expiresIn: '7d',  
  });  
  return res.cookie('token', token, {  
    expires: new Date(Date.now() + expiration),  
    secure: false, // set to true if your using https  
    httpOnly: true  
  });  
}
```

- We have a variable expiration which tells the cookie when it will expire in millisec
- Next we create the jwt token and sign it.
  - **jwt.sign(json-object,secret)**
  - **Important: the secret needs to be the same when we later verify the token**
- After we created the jwt token we take the response object (**res**) and create a cookie and return the response object
  - **res.cookie(cookie-name,data,options)**
    - Options:
      - **expires:** The current date + 7days in millisecs
      - **secure:** if http or https is used
      - **httpOnly:** true=JavaScript has no access to the cookie data, false=JavaScript can access the cookie

```
function generateToken(res, email,id){
  const expiration = 604800000;
  const token = jwt.sign({email, id}, "986caae7d6a0a011e2dc2d0c2d5a8f2", {
    expiresIn: '7d',
  });
  return res.cookie('token', token, {
    expires: new Date(Date.now() + expiration),
    secure: false, // set to true if your using https
    httpOnly: true
  });
}
```

- With the verifyToken function we have middleware we can use on specific routes (we see an example later)
- If we get a request we check if there is cookie within this request and if there is cookie with right name (**token =>as defined in the function before**)
- After the check we decrypt the token and extract the data within the token
  - **jwt.verify(token,secret) //Secret which was used while signing the jwt-token**
- The extracted data from the jwt token is saved in the request-body object in a new object-property
- These data can be then be used to make client-specific database requests
- As this is a **middleware** we need to call the function **next()** to go the next function in the **request-response cycle**

```
const verifyToken = async (req, res, next) => {
  const token = req.cookies.token || '';
  try {
    if (!token) {
      return res.status(401).json('you need to login')
    }
    const decrypt = await jwt.verify(token, "986caae7d6a0a011e2dc2d0c2d5a8f2");
    req.user = {
      email: decrypt.email,
      id:decrypt.id
    };
    next();
  } catch (err) {}
  return res.status(500).json(err.toString());
};
```

- In the picture below we can see an example of a login
- We get an email and password from the client, we make a database request to check if there is user and if all the data are correct. Before we can check if the passwords are correct, we need to **encrypt** the password as **plain passwords** should **never** be saved in the database.
- After the checks are successful we call the **generateToken()** function which we saw earlier
- After the function call we send a response to the client.

```
app.post('/login', async function(req,res){
  let userData = req.body;
  let user = await User.findOne({ email: userData.email });
  if(user){
    let pw = SHA256(userData.password);

    if(user.password === pw.toString()){
      generateToken(res,userData.email, user._id);
      res.status(201).send("successfully signed in!");
    }else{
      res.status(401).send("user or password wrong!");
    }
  }else{
    res.status(401).send("user does not exists");
  }
});
```

- In the picture below we can see the usage of the middleware function **verifyToken**
- The **middleware** is the **second parameter** in our route
- If everything went well in the middleware the **rest** of the route is **executed**.  
If not the code in the route is **never executed**, as the client got a **response** from **within** the **middleware** and the **request-response cycle was completed** in the middleware

```
app.get('/books', verifyToken, (request, response) => {  
  if (request.query.author) {  
    let booksAuthor = [];  
    for (let book of data.books) {  
      if (book.author == request.query.author) {  
        booksAuthor.push(book);  
      }  
    }  
    response.status(200).send(booksAuthor)  
  } else {  
    response.status(200).send(data.books)  
  }  
});
```

- Exercise 2

- 1) Create a new Node.js project and initialize it
- 2) Write a simple server which should manage to-do tasks. The server should have 4 routes.

The sever should only accept JSON request body and should send JSON object back to the client. **No Authentication is needed**

- 1) GET: get all the tasks
  - 2) POST: add a new task
  - 3) PUT: update a task
  - 4) DELETE: delete a task
- 3) The to-do's are saved in a JavaScript JSON object and the routes change this JSON object.  
The server has internal id-counter which counts +1 for every new task added to the object

How could the JSON object look like:

```
let to_do = {  
  tasks:[  
    {  
      id:1,  
      title:"Test",  
      completed:false  
    }  
  ]  
}
```