HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Web Programming
JavaScript – node.js - react

Lecture 4-1:
JavaScript Asynchronous 2
16.11.

Stefan Noll
Manuel Fehrenbach
Winter Semester 22/23

AJAX and Interacting with Servers

To be honest, this chapter is perhaps the most significant one in the entire book. Let us understand this claim with a simple example. If you think about it for a moment, you will realize that every time you open a social media website, like Facebook, on your browser, the content is not always the same. You will see different content because every time you open Facebook, your browser sends several requests to the servers hosted by Facebook. The response to these requests renders the **user interface** (**UI**) that you see on your browser. In the mean real world out there, your work will significantly involve creating UI that is dependent on the data you receive from servers.

In this chapter, we will not only learn how to interact with servers (and all the hullaballoo associated with it), but also lay down the foundations for creating a live webpage using APIs in the next chapter! In this chapter, you will be learning about AJAX (a JavaScript process for connecting to the servers), JSON (the universal format for sending or receiving data from the servers), requests, responses, and understand the basics for creating a live application.

Structure

In this chapter, we will cover the following topics:

- JSON – The Web's favorite file format
- AJAX
- HTTP Headers

Objective

After studying this chapter, you should be able to interact with actual real-life servers (finally!). You can update UI efficiently using data received from the servers and send requests and receive responses safely across the internet

JSON – The Web's favorite file format

In the introduction, we read about browsers exchanging data with the servers. In the simplest of imaginations, we can visualize this as follows:

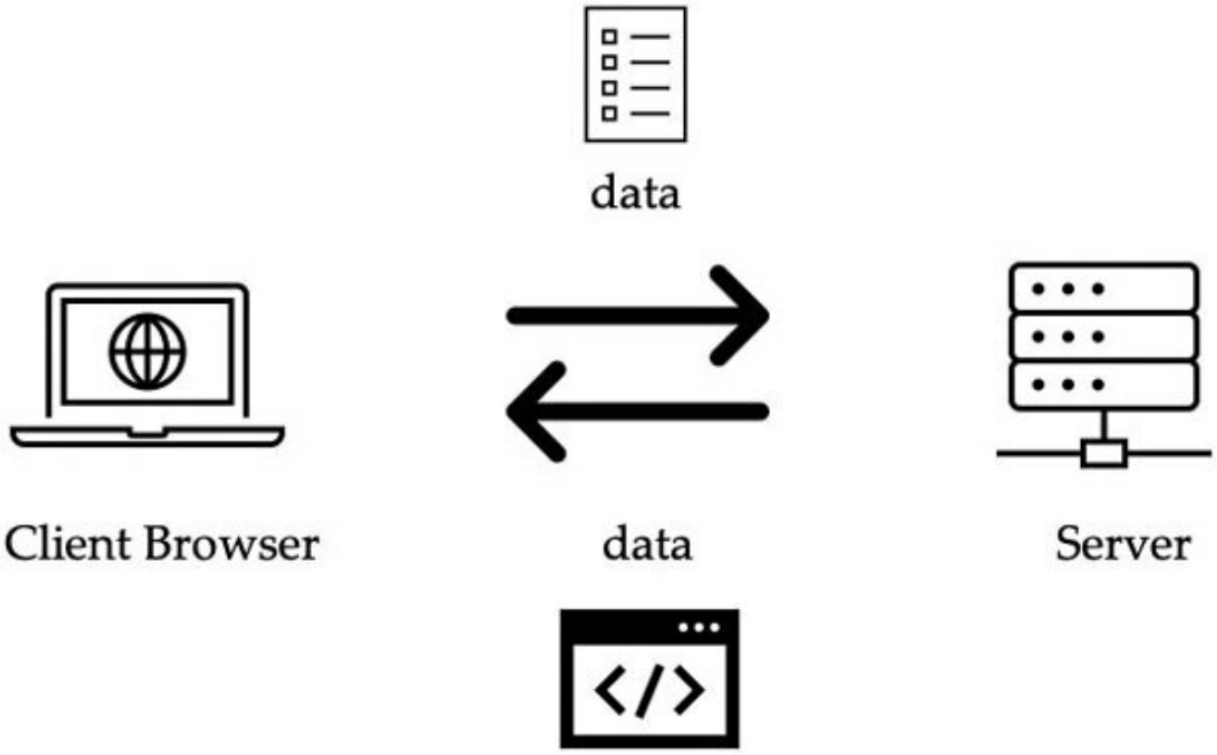

Figure 9.1: *Graphical visualization of data exchange between browser and server*

If you let your imaginations run wild, you can think of a hundred different formats for exchanging data between a browser sitting on a client machine and a server. However, imagine if the server has to send a 400MB video as a response over the internet. Even with the best internet speeds, it's fair to say the user would be bored to death waiting for the browser to receive this video as the server response. To add to that, imagine if your JavaScript is not asynchronous! It would mean your user will see nothing but a blank screen or a loading icon until the video ultimately reaches the browser. (It could take more than half-hour on slow networks!)

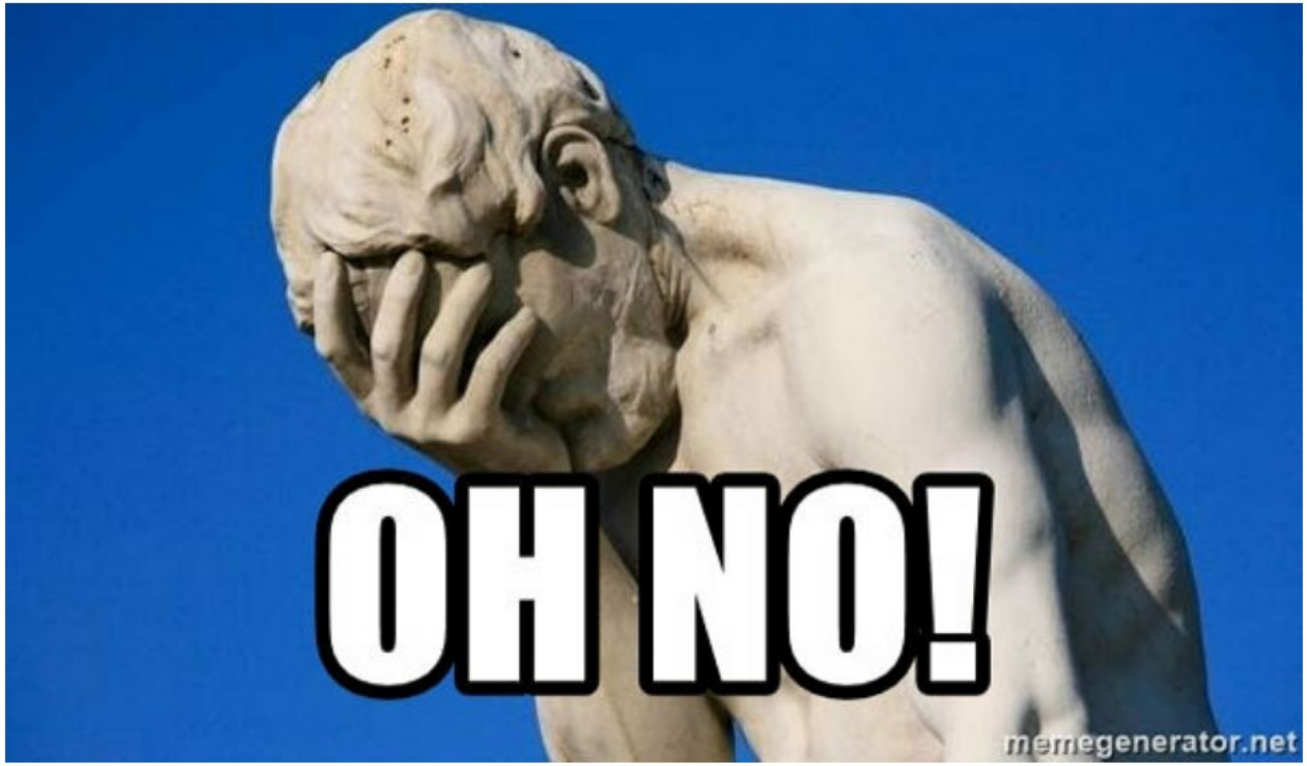

Figure 9.2: *When disappointment strikes.*

Therefore, for simplicity, browsers exchange data in text format. It is simple and fairly lightweight.

Tip: A potential next question could be: How will you show videos or images that reside on the server? While there are several answers to this question, one of the best answers is a relative path to the image/video stored in the server. In this scenario, we store the media files like the images and the videos inside a folder residing on the server. In the server response, the browser only receives the relative path to that image/video. Using that path in the `<img>` or the `<video>` tag, it will show the respective image/video without having the user waste several bytes of data to download it.

In *Chapter 5: Objects*, we learned that objects are a brilliant way of emulating real-life data. It means if you want to send multiple information about a specific object like a car, a person or quite literally anything, you can store all that information inside a simple object.

Combining the best of both the worlds of text and objects, JavaScript introduced a new format called **JavaScript Object Notation** (**JSON**).

Officially, according to the **Mozilla Developer Network** (**MDN**) docs, **JavaScript Object Notation** (**JSON**) is a standard text-based format for representing structured data based on JavaScript object syntax.

In its simplest form, JSON is a string that resembles the JavaScript object. It can also be used independently from JavaScript. In fact, several programming environments have the ability to read and generate JSON. You can also store JSON as a file. JSON files must be saved with the extension `.json`.

Structure of JSON

As mentioned earlier, the JSON object bores a marked resemblance to the JavaScript object. Without much further ado, let's quickly take a look at the structure of a JSON object.

```
"{
  "firstName":"John",
  "lastName":"Doe",
  "address": {
    "building":"River View Apartments",
    "apartment":12,
    "road":"York Road",
    "city":"London",
    "country":"England"
  },
  "cars": ["BMW","Audi","Jaguar"]
}"
```

***Code 9.1**: Syntax of JSON Object*

```
1. "{
2.    "firstName":"John",
3.    "lastName":"Doe",
4.    "address": {
5.       "building":"River View Apartments",
6.       "apartment":12,
7.       "road":"York Road",
8.       "city":"London",
9.       "country":"England"
10.   },
11.   "cars": ["BMW","Audi","Jaguar"]
12. }"
```

***Code 9.1**: Syntax of JSON Object*

As you can see in the *code example 9.1*, the JSON object is remarkably similar to the JavaScript object literal. Just like the JavaScript object literal, the data inside a JSON object is in key: value pairs and separated by commas. Similarly, objects (like address in the *code example 9.1*) and arrays (like cars in the *code example 9.1*) are denoted using the curly braces and square bracket notation, respectively. Also, as per the JavaScript standards, the JSON value having number as a data type should not be in quotes.

However, there are some significant differences between a JSON object and a JavaScript object literal:

- In the JSON objects, every key in the *key: value* pair needs to be in double-quotes.
- The JSON values cannot be a function, a date, or undefined.
- You *cannot* use single quotes for JSON keys or values.

JSON works in two formats:

- **String**: When data needs to be transmitted.
- **Object**: When you need to access the data.

JavaScript provides a global JSON object that helps in converting JSON from one form to another. We will understand both the formats in the next section.

Working with JSON

Now that we have a fair understanding of the structure of a JSON, and we know that JSON exists in two different forms, let's underline the process for working with JSON.

We know that the best format for data exchange between browsers and servers is the text format. However, once we receive the data on our browsers from the servers in the text format, we need to convert the text format to an object format for accessing the data. The global JSON object of JavaScript provides two methods for easy conversion between the two formats.

JSON.stringify()

To convert a JavaScript object into a text format to be sent across the internet, JavaScript provides a JSON method called **`JSON.stringify()`**. All we need to do is provide the object inside the parentheses of the **`JSON.stringify()`** method. Let's understand this with an example.

```
let obj = {
  firstName: "John",
  lastName: "Doe",
  address: {
    building: "River View Apartments",
    apartment: 12,
    road: "York Road",
    city: "London",
    country: "England"
  },
  cars: ["BMW", "Audi", "Jaguar"]
}


JSON.stringify(obj);
```

***Code 9.2:** JSON.stringify() method*

As you can see in the *code example 9.2*, we have a JavaScript object **person**. To convert this to a JSON string, we need to pass the object person inside the parentheses of the method **JSON.stringify()**. When you execute the *code example 9.2*, the results would look something like the following:

Figure 9.3: *Output of code example 9.2*

As you can see in the preceding screenshot, the output of line number 14 of the *code example 9.2* is a JSON string. If you observe *figure 9.3* closely, you can see that the object person of the *code example 9.2* has been converted to the JSON format in *figure 9.3*. The keys of the object person are now encased in double-quotes, as per the JSON format. In fact, the output itself is printed inside double quotes. If you use the JavaScript operator **typeof** in front of this output, you will receive the output as a string, as shown in the following screenshot:

Figure 9.4: *The data type of the output of the stringify method*

```
1. let obj = {
2.   firstName: "John",
3.   lastName: "Doe",
4.   address: {
5.     building: "River View Apartments",
6.     apartment: 12,
7.     road: "York Road",
8.     city: "London",
9.     country: "England"
10.  },
11.  cars: ["BMW", "Audi", "Jaguar"]
12. }

13.
14. JSON.stringify(obj);
```

Code 9.2: *JSON.stringify() method*

JSON.parse()

In the previous section, we learned how to convert a JavaScript object to a JSON string. Now, we also need to understand how to convert a JSON string received from the server to a JavaScript object that we can use in our web application. To convert a JSON string to a JavaScript object, JavaScript provides us with the method **`JSON.parse()`**. Similar to the **`JSON.stringify()`** method, we need to pass the JSON string inside the parentheses of the **`JSON.parse()`** method. Let's look at an example.

Figure 9.5: *An example of JSON.parse()*

As shown in the preceding screenshot, we have used a JSON text (the one that we have received as the output in *figure 9.3*) and supplied it to the **`JSON.parse()`** method. As you can see, the output of the **`JSON.parse()`** method is a JavaScript object, denoted by the absence of the double-quotes.

Exchanging data

In the previous sections, we saw the process of converting JSON from one form to another. Moreover, we also understood when to use JSON as a string, and when to use it as a JavaScript object. With these points in mind, let's look at the process of exchanging data between a browser sitting on a client machine and a server with the help of a diagram:

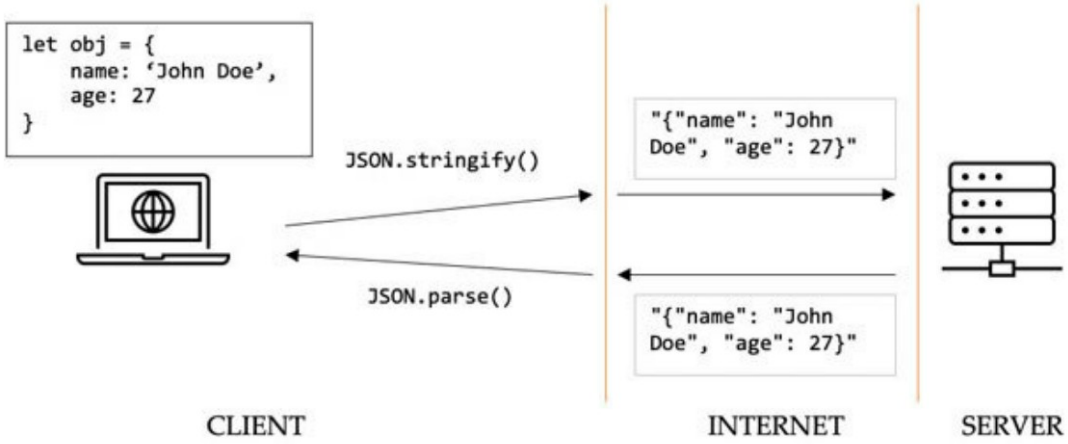

Figure 9.6: *An example of JSON.parse()*

As shown in the preceding screenshot, when the client browser is sending data across the internet, it uses **`JSON.stringify()`** to convert the JavaScript object into a JSON text. Similarly, when the client browser receives data from the server in a JSON text, it uses **`JSON.parse()`** to convert the text to a JavaScript object.

AJAX

Now that we have read about the format used for exchanging data between the servers, let's finally learn how to interact with servers.

AJAX is the JavaScript process of connecting to the servers. It stands for Asynchronous JavaScript and XML.

AJAX is a web developer's dream. It not only enables sending and receiving data from the servers, but it also updates a web page without reloading.

However, before we understand AJAX and learn the simple four-step process of AJAX to interact with a server, we need to take a quick pitstop, and first need to understand a protocol involved in the exchange of data between servers and clients.

HTTP

The protocol or the system of rules that we use in web development for the exchange of data is the **Hypertext Transfer Protocol** (**HTTP**). The HTTP works in a simple request-response fashion that we have seen earlier in this chapter. The logic is as follows:

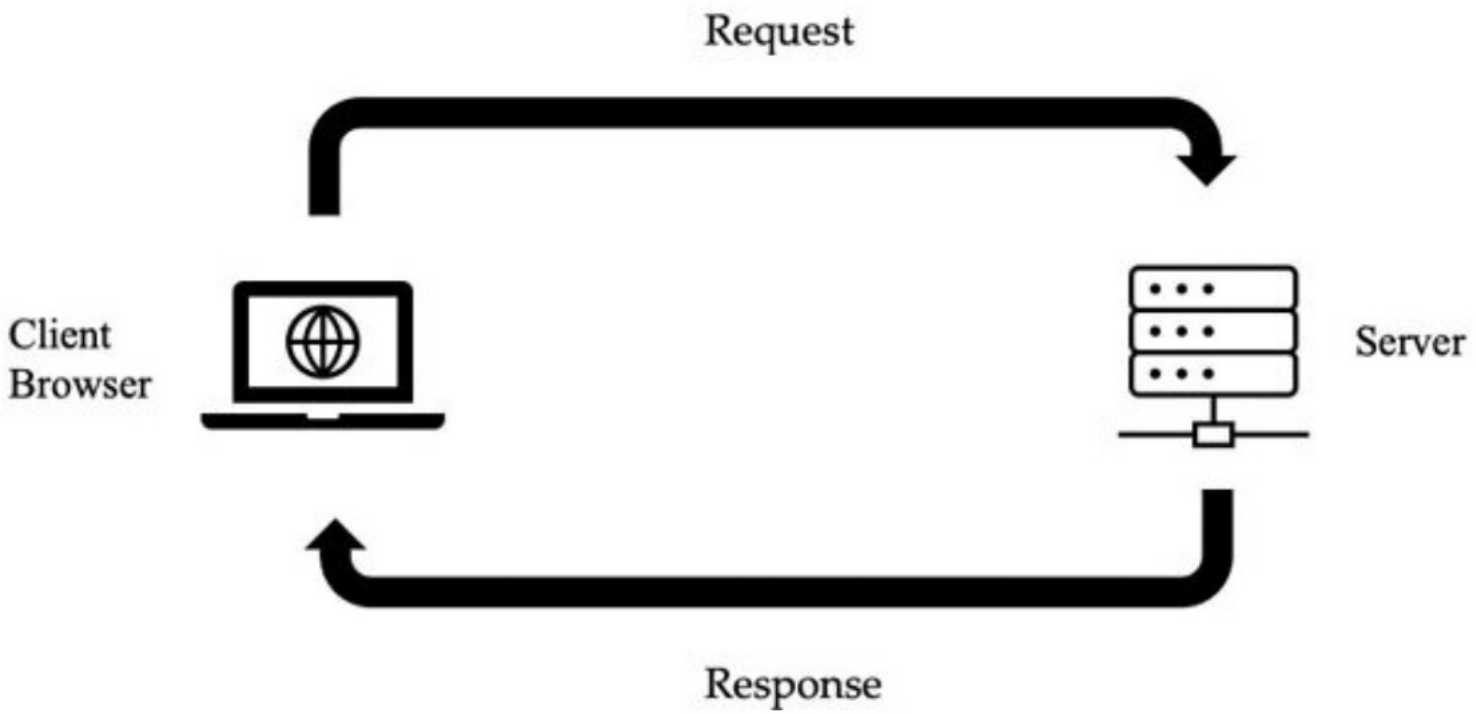

Figure 9.7: *The logic of HTTP*

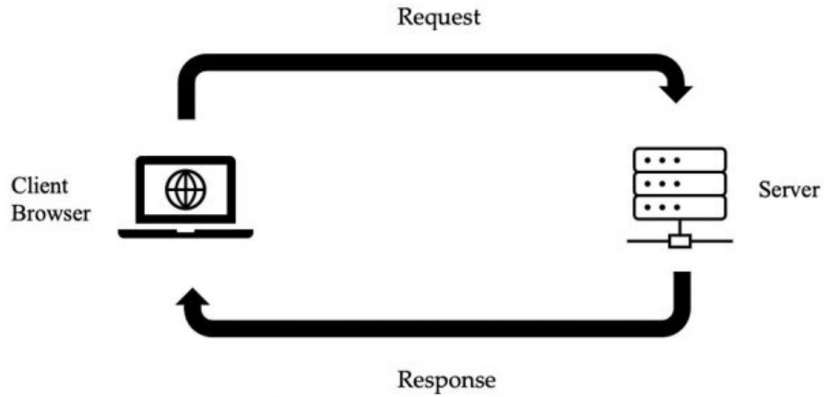

Figure 9.7: *The logic of HTTP*

As shown in the preceding screenshot, the client browser sends a request to the server. The client browser can request several things from the server. It includes, but is not limited to, HTML file for showing on the browser, media like images, and videos, fonts, and data like objects and arrays (as we have seen in JSON). In fact, at times, the client browser can even send a request for updating the data stored in the server. The server receives this request, parses it, and sends a response back to the client. In response, the server always sends information about the status of the request. In addition to that, the server also sends data in the response if the browser has requested it.

Let's first explore the various methods of requests.

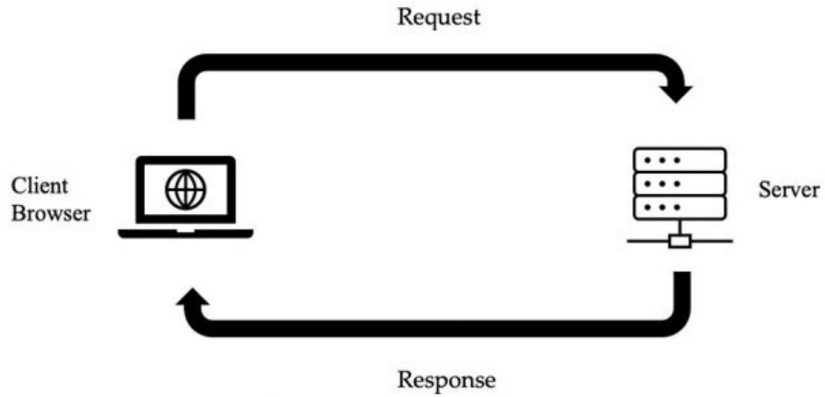

Figure 9.7: *The logic of HTTP*

HTTP Methods

There are several HTTP methods - **`GET`**, **`POST`**, **`PUT`**, **`PATCH`**, **`DELETE`**, **`HEAD`**, and **`OPTIONS`**. Each of these methods caters to a different need. However, the most commonly used methods are **`GET`** and **`POST`**.

Let's go through each of the HTTP methods one by one:

GET

The GET method is used for requesting data from a specified resource. An example of this would be requesting an HTML file or the list of employees from a server.

However, we need to be aware of several pointers when using the GET method:

- The GET method should be used only for requesting the data. It should *not* be used for modifying the data.
- If we need to send data as a part of the **`GET`** request, it will be sent in the URL of the request. An example of this is to send the bank account details of a user for fetching the transaction details of the user. Therefore, it is advisable to not use **`GET`** requests when you need to send sensitive data in the requests.
- The **`GET`** requests can be cached and bookmarked. They remain in the browser history.
- The **`GET`** requests have restrictions on the length of the data.

POST

The **POST** method is specifically used for sending data to a server. Its primary purpose is to create the data on the server. A few additional points on the **POST** request are as follows:

- The data sent in a **POST** request is stored in the request body of the **HTTP** request.
- The **POST** requests are neither cached nor book-marked. They do not remain in the browser history.
- The **POST** requests do not have any restriction on the length of the data.
- Sending a **POST** request will create a new resource on the server.

PUT

The **`PUT`** method is also used for sending data to a server. Like **`POST`**, its primary purpose is to create and also, update the data on the server. There are the following two major differences between **`POST`** and **`PUT`**:

- **Exact URI**: PUT requests need to know the exact URI of the resource while **`POST`** does not mandatorily require the entire URI.

  For example, let's imagine we have a URI for a list of employees on the server, and it is **`/employees`**. It means that if you send a **`POST`** request to the URI **`/employees`** with the data of the new employee, it can create a new **`employee`** on the server and assign it a unique ID. For the sake of understanding the example, let's imagine that this employee has been assigned the **`ID 423`**. Alternatively, we can also create the new user on the server by sending data as a **`POST`** request specifically to **`/employees/423`**. If no IDs are provided, the **`POST`** request will automatically generate a new ID for the new data. This shows that the **`POST`** request does not necessarily need a complete URI to work.

However, if you are sending a PUT request to **`/employees`** with the data of the new employee, it will fail. As mentioned before, the **`PUT`** request needs to know the exact URI to work. Therefore, if you send a PUT request to the **`/employees/423`**, it will either create a new employee if the ID 423 does not exist or it will replace the employee at the ID 423.

	POST	PUT
`/employees/`	✓	✗
`/employees/123/`	✓	✓

Table 9.1: *POST vs PUT*

- **Idempotence**: Probably, one of the most significant reasons for working with **PUT** over **`POST`** is the property of idempotence. It means that if you accidentally or deliberately send the **`POST`** request N times to the server with the same data, it will create N different resources on the server. However, if you accidentally or deliberately send the **`PUT`** request N times to the server with the same data, it will be equivalent to a single request modification.

PATCH

The **PATCH** request is useful for the scenarios where we wish to make a partial update to the resource stored in the server. For instance, let's say we have a resource on the server that looks like the following:

```
1. {
2.   firstName: 'John',
3.   lastName: 'Doe',
4.   age: 24,
5.   location: 'Detroit'
6. }
```

Code 9.3: *A resource on the server*

Let's say we wish to change the location of the resource shown in the *code example 9.3*. If we send a **PUT** request with the data **{location: 'New York'}**, it will replace the entire object with the location as New York. It's because, for the **PUT** request, we require the entire resource. If we want to send only one part of the resource to be modified, we need to use the **PATCH** request.

Therefore, when we send a **PATCH** request with the data **{location: 'New York'}**, it will update the resource as the following:

```
1. {
2.   firstName: 'John',
3.   lastName: 'Doe',
4.   age: 24,
5.   location: 'New York'
6. }
```

Code 9.4: *The resource after the PATCH request*

To simplify understanding **`POST`** vs **`PUT`** vs **`PATCH`**, refer to the following diagram:

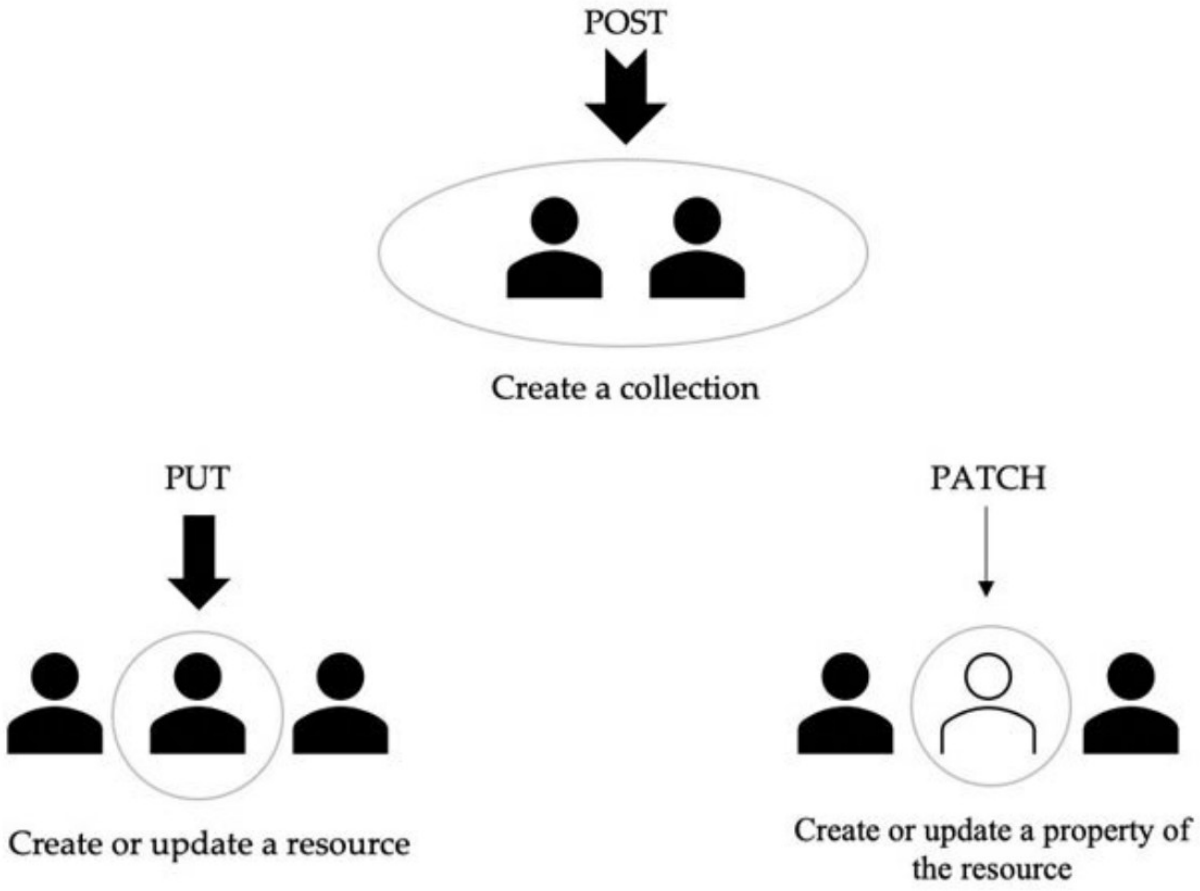

Figure 9.8: *POST vs PUT vs PATCH*

DELETE

The **`DELETE`** method deletes the specified resource.

HEAD

The **`HEAD`** method is similar to the **`GET`** method, but it does not return a body in the response. It means if you send a **`GET`** request to the URI **`/employees`**, you will get a list of employees in the response. However, if you send a **`HEAD`** request to the same URI, it will not return the list of employees. **`HEAD`** methods are extremely useful for checking what a **`GET`** request will return. It's run before making the actual **`GET`** request.

OPTIONS

The **`OPTIONS`** request is used for requesting information about the communication options for the target resource. For example, if we want to know what request methods are allowed for a particular URI, we can use **`OPTIONS`** to fetch that information.

HTTP status messages

In the introduction of the section – HTTP Methods, we learned that the server always returns the status of the request. As we will see in the next topic – The Process of AJAX, the status responses are useful to understand what happened to our requests at the server.

For instance, if you open **https://www.google.com** on your Google Chrome browser, and open the **`Network`** tab of your Chrome Developer Tools, you will see the first request titled **`www.google.com`**. As shown in the following screenshot, if you look at the **`Headers`** tab, you will see a key aptly called **Status Code**.

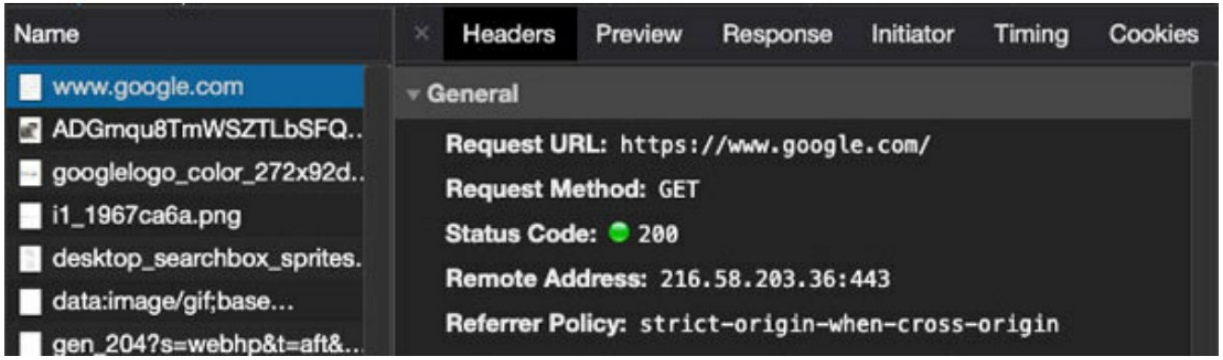

Figure 9.9: *The status code shown in Network tab of Chrome Developer Tools*

The status messages come in two parts – a status code and a status message. The first digit of the status code defines a category of the status message. You can see the entire list of status messages here – **https://developer.mozilla.org/en-US/docs/Web/HTTP/Status**.

For the sake of understanding, we will look at a few of the significant status codes in every category that are used frequently in web development:

1xx: Information

Status code beginning with 1 are informational in nature. For example, the status code 100 has the status message as Continue. It means the server has received the request headers, and it informs the client to proceed to send the body of the request.

2xx: Successful

Status codes beginning with 2 are successful in nature. In *figure 9.9*, we have seen the status code 200, which signifies that the request has been successful. A few of the commonly used status codes starting with 2 are listed as follows:

Status Code	Status Message	Description
200	OK	The standard response for successful requests.
201	Created	A new resource has been created.

Table 9.2: *Codes beginning with 2*

3xx: Redirection

Status codes beginning with 3 denote redirection. It means the URL you accessed has been moved, or you have been redirected to another location. A few of the commonly used status codes that begin with 3 are as follows:

Status Code	Status Message	Description
301	Moved Permanently	The requested page or resource has been moved to a new URL.
302	Found	The requested page or resource has been temporarily moved to a new URL.

Table 9.3: *Codes beginning with 3*

4xx: Client Error

One of the most commonly seen status codes are the ones that begin with 4. It denotes error from the client browser or the client machine. A few of the commonly seen 4xx status codes are as follows:

Status Code	Status Message	Description
400	Bad Request	Request cannot be fulfilled because of a bad syntax.
401	Unautho-rized	The request was legal, but the server refuses to respond. This happens com-monly when the authentication has not yet been provided.
403	Forbidden	The server does not allow this request.
404	Not Found	The requested page or resource could not be found.

Table 9.4: *Codes beginning with 4*

5xx: Client Error

Finally, the status code beginning with 5 refer to errors happening on the server. A few of the commonly seen 5xx status codes are as follows:

Status Code	Status Message	Description
500	Internal Server Error	A generic server error message.
501	Not Imple-mented	Server does not recognize the request or lacks the ability to complete the request.
502	Bad Gateway	The server was acting as a proxy/ gateway and it received an invalid response.
503	Service Unavailable	The server is unavailable.

Table 9.5: *Codes beginning with 5*

We will be using these status codes in the next topic – The Process of AJAX.\

The Process of AJAX

Finally, now that we have updated our knowledge with almost every pre-requisite information necessary for implementing AJAX, let's begin learning the process of AJAX.

There are four simple steps for working with AJAX, which are as follows:

Figure 9.10: *The process of AJAX*

As shown in the preceding screenshot, the process of AJAX involves the following four basic steps:

1. **Create Request**: As basic as it sounds, the very first step in the process of AJAX is to create a request that needs to be sent to the server. The foundation of the entire AJAX process is the **`XMLHttpRequest`** object.

> **Note:** To break it down, XML stands for Extensible Markup Language. It is a markup language like HTML that uses tags to define elements within a document.
>
> HTTP stands for Hypertext Transfer Protocol, the topic we learned in the previous section.

The **`XMLHttpRequest`** object is used for exchanging data with a web server behind the scenes. All the modern browsers at the time of writing (including Google Chrome, Mozilla Firefox, Microsoft Internet Explorer 7+, Microsoft Edge, Safari, Opera) support the **`XMLHttpRequest`** object.

The syntax for creating a **`XMLHttpRequest`** is as follows:

```
let request = new XMLHttpRequest();
```

Code 9.5: *Creating a request*

As shown in the *code example 9.5*, to create a **`XMLHttpRequest`** object, we need to use the JavaScript keyword new. Since we need to use this request often, it's wise to store the request in a variable.

2. **Open Request**: After creating the request, the next step is to open that request. Now, this is the most vital step in the process of AJAX. In this step, we need to provide the URL of the server that we intend to connect with our webpage/web application. Moreover, we also need to specify the method or type of request, whether this request will be asynchronous, and whether we need a username and a password.

   The general verbose syntax for opening a request is as follows:

```
open(method, url, async, username, password)
```

***Code 9.6**: Syntax of opening a request*

As shown in the preceding *code example 9.6*, the open method takes five parameters. However, the last three options are optional. Let's understand each parameter one by one.

Method: The very first parameter is the method parameter. The method parameter helps the server understand the type of request made by the client browser. In the section –HTTP Methods, we had a look at the several methods of an HTTP request. We can add any one of those methods as the first parameter to the open method.

URL: The second parameter to the open method is the URL of the server. This URL can be an API endpoint or a location on the server. We will be looking at the APIs in more detail in the next chapter. Alternatively, it can also be the location of a file on the server. Generally, these files can perform actions on the server before sending back the response.

Asynchronous: The third parameter is an optional parameter to set whether the request is asynchronous. It's a boolean value – so you can either send true or false as the third parameter to the open method. As we already know the meaning of asynchronous from the last chapter, setting the third parameter as true will not make JavaScript stop the execution, and wait for the response of this request. Instead, JavaScript will continue its execution and the user interaction until the server response arrives. However, if the third parameter is set to false, JavaScript will stop all its execution until the response of this request is received. Its default value is true.

Username and Password: The fourth and the fifth parameters are also optional. The fourth and the fifth parameters are for username and password, respectively. They can be used for authentication purposes. Their default values are null.

Continuing our *code example 9.6*, let's try the open method with an actual API request:

```
request.open('GET', 'https://www.the-
mealdb.com/api/json/v1/1/search.php?
s=Arrabiata', true);
```

***Code 9.7**: An example of opening a request*

In preceding *code example 9.7*, we have used the variable request that we have created in *code example 9.5*. We invoke the open method on the variable request and pass three parameters to it. The first parameter should be the type of HTTP request. Therefore, we have added the '**GET**' method as the first parameter. For the second parameter that should be a URL, we have added an API of a popular public API platform called The Meal DB. Finally, we have added the optional third parameter for asynchronous behavior as true. We do not need the optional fourth and fifth parameters.

3. **Send Request**: The third step in the process of AJAX is to send the request that we have created in the first two steps. The syntax for sending a request is **`request.send()`** in case of **`GET`** requests, and **`request.send(str)`** in case of **`POST`** requests. Continuing the example used in *code examples 9.5* and *9.7*, we will now send the following request:

```
request.send();
```

Code 9.8: *Sending a request*

As shown in *code example 9.8*, since the **`XMLHttpRequest`** that we have created earlier is a **`GET`** method, we will invoke the send method without any parameters.

4. **Wait for the response**: With the first three steps, we have completed creating, opening, and sending the request to a server. The server now parses the request, prepares a response, and returns the response to the client. To understand the response, and how to receive it, we first need to understand a concept called **`readyState`**.

The **`readyState`** property belongs to the **`XMLHttpRequest`** object. It holds the state of the **`XMLHttpRequest`** client. There are five values of the **`readyState`** property:

The **readyState** property belongs to the **XMLHttpRequest** object. It holds the state of the **XMLHttpRequest** client. There are five values of the **readyState** property:

Value	State	Description
0	UNSENT	The client has been created. When we complete the first step of the AJAX process, the value of the `readyState` becomes 0.
1	OPENED	The `open()` method has been invoked. When we complete the second step of the AJAX process, the value of the `readyState` becomes 1.
2	HEADERS_RECEIVED	The send() method has been invoked. When we complete the third step of the AJAX process, the value of the readyState becomes 2. At this point, the response headers have been received.
3	LOADING	The response's body is being received.
4	DONE	The operation is complete. It means either the data transfer has been successful or failed.

Table 9.6: *Values of readyState*

As we can see in *table 9.6*, the values of the property **readyState** changes with every action or every step of the AJAX process. Moreover, from *table 9.6*, we can see that when the value of **readyState** denotes 4, it signifies that the process of interaction with the server is complete.

The **XMLHttpRequest** object contains a property called **onreadystatechange**. Every time the value of the **readyState** property changes, the event handler inside the **onreadystatechange** property is called. Let's try understanding the **onreadystatechange** property with an example:

The **XMLHttpRequest** object contains a property called **onreadystatechange**. Every time the value of the **readyState** property changes, the event handler inside the **onreadystatechange** property is called. Let's try understanding the **onreadystatechange** property with an example:

```
1. request.onreadystatechange = function()
   {
2.   if(this.readyState === 4 && this.sta-
   tus === 200) {
3.     console.log(this.responseText);
4.   }
5. }
```

Code 9.9: *An example of the onreadystatechange property*

In *code example 9.9*, we have continued using the variable request that we have created and used in *code examples 9.5*, *9.7*, and *9.8*. As you can see in *code example 9.9*, we have attached a function for listening to the event when the **readyState** property of the variable request changes. Inside the function, we check for the condition whether the **readyState** property is 4 and the status of the response is 200. As we have seen in *tables 9.2* and *9.6*, the **readyState** property is 4 when the entire operation of request and response has been completed. Similarly, the status of the request becomes 200 when the request has been successful. To sum it up, it means that our request to the sever has completed and it has been successful. When the two conditions are met, we execute the statement to log the **responseText**. The property **responseText** returns the text received from the server.

Combining our entire code, it looks like the following:

```
1. let request = new XMLHttpRequest();
2. request.open('GET', 'https://www.the-
   mealdb.com/api/json/v1/1/search.php?
   s=Arrabiata', true);
3. request.send();
4. request.onreadystatechange = function()
   {
5.   if(this.readyState === 4 && this.sta-
   tus === 200) {
6.     console.log(this.responseText);
7.   }
8. }
```

***Code 9.10**: The complete process of AJAX*

As you can see from *code example 9.10*, it's fairly easy to create and send a request to the server. Also, we have limitless opportunities to deal with the server responses. Instead of the **`console.log`** statement, we can also include a callback function to deal with the successful response. Alternatively, if the request fails and we get a status starting with 4 or 5, we can include another condition to deal with the failed requests. In fact, we can also execute special code for situations where we want to execute something before the request is sent or during the loading of the request.

HTTP Headers

In the previous section, we successfully completed an interaction with a public API server. Let's have a quick look at the Network Panel of our Chrome Developer Tools to see the request in action.

To achieve this, let's create a small project. On a favorable location on your machine, create a folder and name it *'Nom Nom'*. This will be the name of our simple application. You are free to choose whatever name you'd like to give to this project. Inside this folder, create an HTML file called **`index.html`**. The following is what our simple HTML file would look like:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.   <meta charset="UTF-8">
5.   <meta name="viewport" content="width=de-
   vice-width, initial-scale=1.0">
6.   <title>Nom Nom</title>
7. </head>
8. <body>
9.   <div>
10.     <h1>Welcome to Nom Nom</h1>
11.     <h5>The best place to learn mouth-wa-
   tering recipes!</h5>
12.   </div>
13.   <script src="scripts/index.js"></script>
14. </body>
15. </html>
```

Code 9.11: *The index.html file*

Now, we will create a folder on the same level as the **`index.html`** file and name it as scripts. You can visualize this as follows:

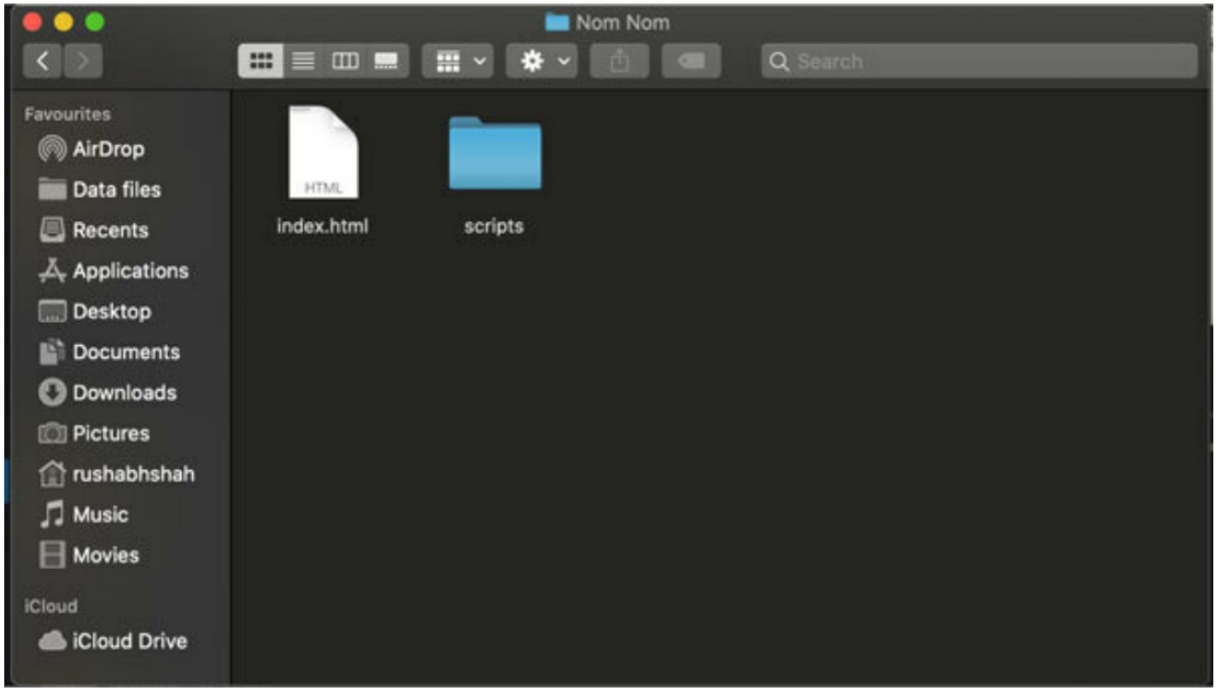

Figure 9.11: *The folder structure for Nom Nom*

Inside the scripts folder, we will create a JavaScript file and name it **`index.js`**. Inside **`index.js`**, we will paste the code that we have seen in *code example 9.10*.

When you click on the **`index.html`** file inside the folder **`Nom Nom`**, it will open up our web page on the default browser of your machine. Since I constantly use Google Chrome for development, the default browser on my machine is Google Chrome. After clicking on **`index.html`**, our web page will look like the following on the browser:

Figure 9.12: *The web page of Nom Nom*

As we had learned in *Chapter 2: The Developer's Tools*, we can open the Chrome Developer Tools by pressing right-click on any part of the web page inside the browser and clicking on **`Inspect`**. From there, we can navigate to the **`Network`** panel by clicking on the **`Network`** tab at the top of the Developer Tools. When you complete this, you will see something like the following:

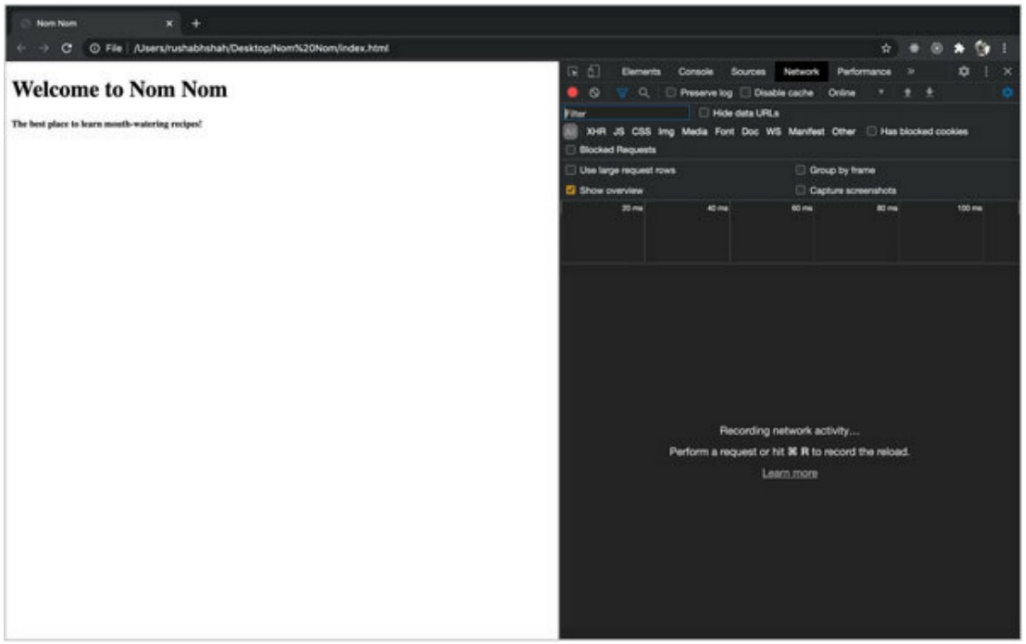

Figure 9.13: *View of the Network Panel of the Chrome Developer Tools*

As you can see in the preceding screenshot, when you open the **`Network`** panel for the first time, it's blank. In fact, it asks us to perform a request or hit refresh. So, let's go ahead and refresh the page by pressing the shortcut key *F5* on your machine, or by clicking on the button next to the address bar of your browser.

When you hit refresh, you will see three requests populated on your **`Network`** panel, as shown in the following screenshot. If you hover on each of the items in the **`Name`** column, you will notice that the three items are the requests for index.html, index.js, and finally, our server API request.

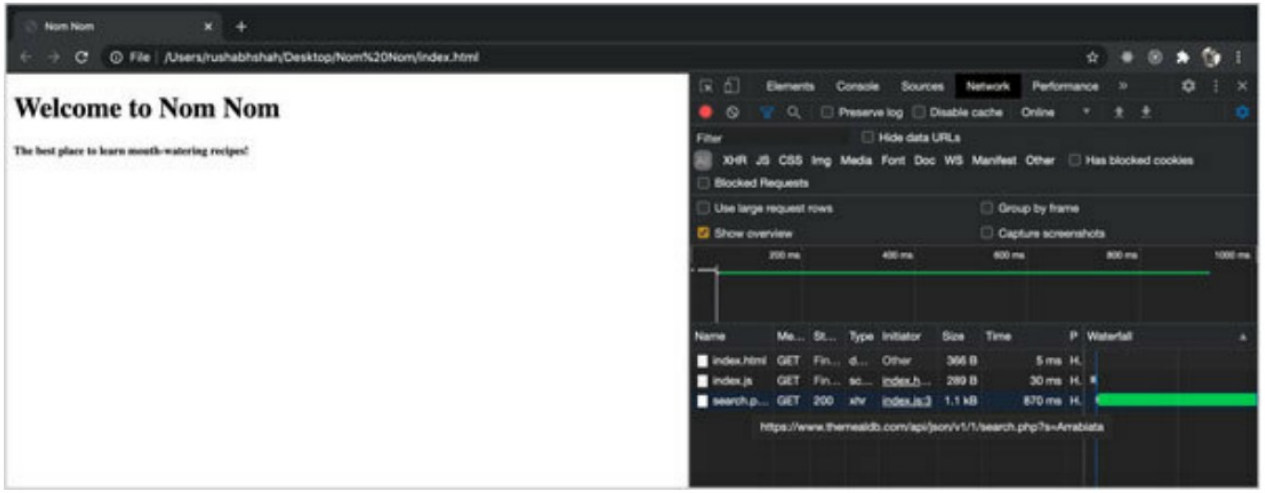

Figure 9.14: *Requests shown in Network Panel*

If we click on the last request shown in *figure 9.14*, it will reveal a sidebar next to the **`Name`** column of the requests. Based on your history of working with the **`Network`** panel, the sidebar might open up on any of the tabs shown in the following screenshot. However, it's very likely that if you are opening it for the first time, it might open up on the **`Preview`** tab:

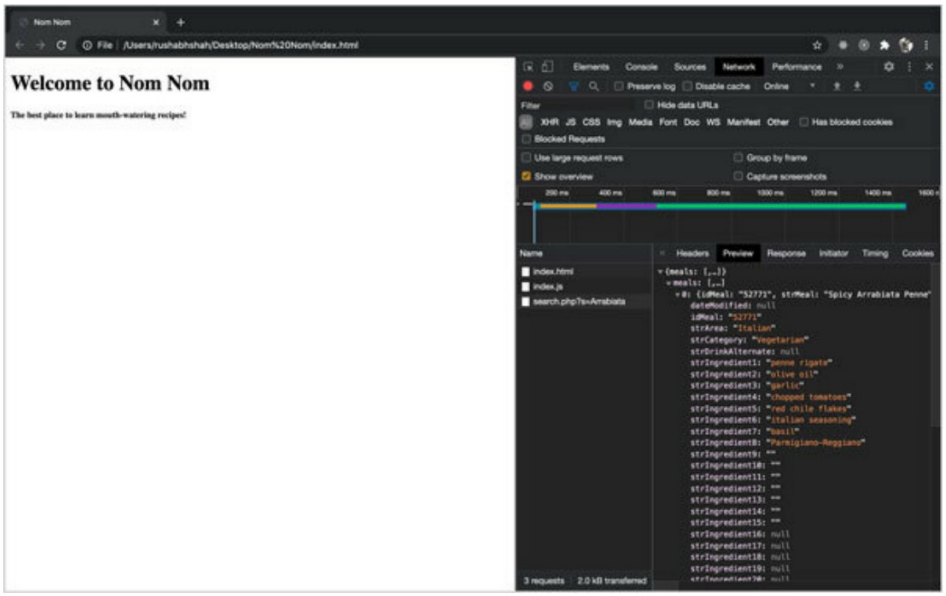

Figure 9.15: *The Preview tab of the sidebar on Network Panel*

As shown in the preceding screenshot, the **`Preview`** tab shows you a preview of the response sent by the server to our request. As you can see, for the API GET request **https://www.themealdb.com/api/json/v1/1/search.php?s=Arrabiata**, the server responds with an object pertaining to the dish 'Spicy Arrabiata Penne'. However, as we have seen earlier in this chapter, the server does not send an object as a response across the internet. In fact, it sends a JSON string as a response. If you move to the **`Response`** tab of the sidebar inside the **`Network`** panel, you will see the actual response sent by the server. The Chrome Developer Tools basically detect and convert the JSON string into an object for the Preview tab, so that it becomes easier for the developers to comprehend the response.

In this section, we will be concentrating on the **`Headers`** tab of the sidebar in the **`Network`** panel. So, if you move to the **`Headers`** tab on the sidebar inside the **`Networks`** panel, you will see something like the following:

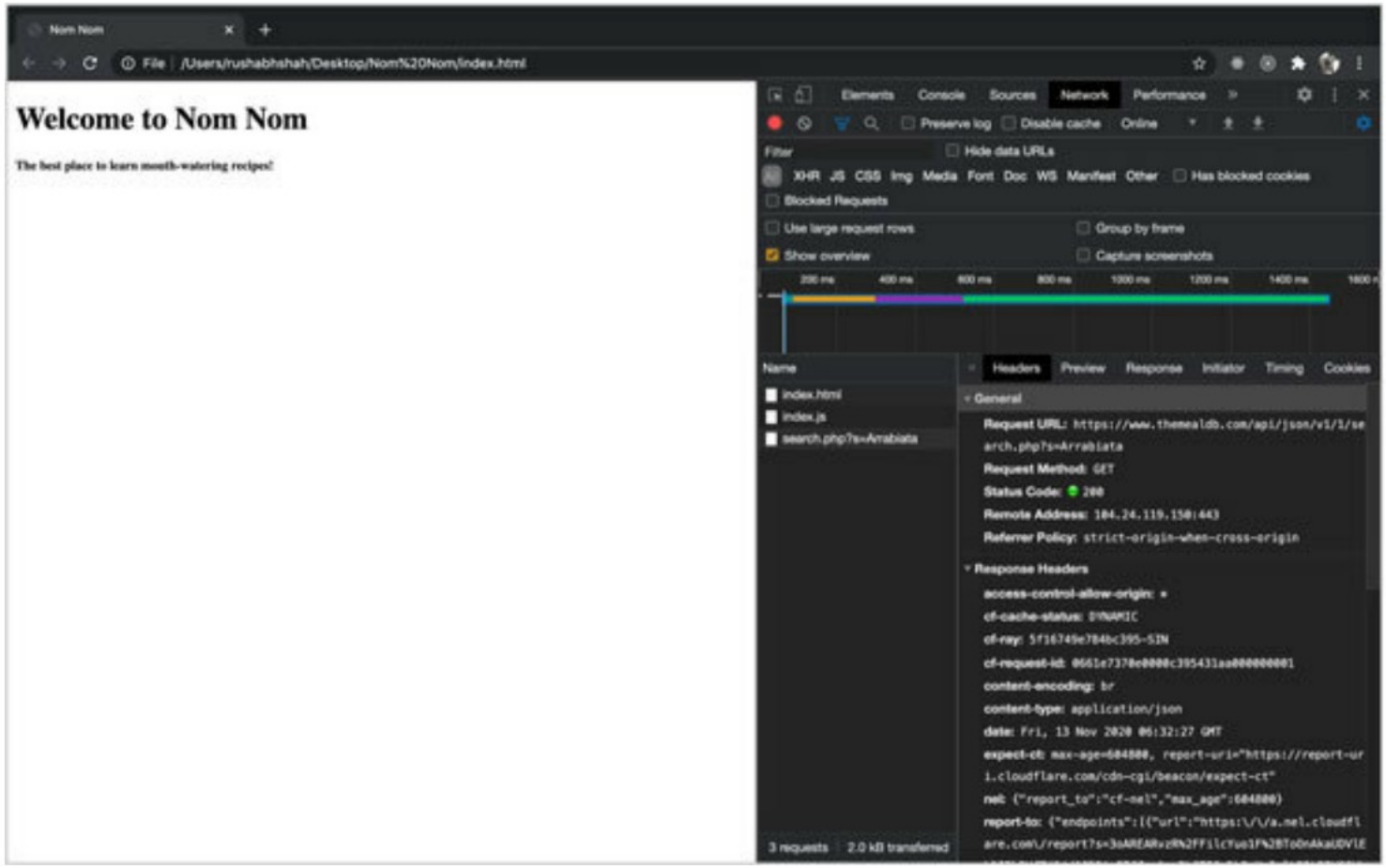

Figure 9.16: *The Headers tab of the sidebar on Network Panel*

If you scroll down the **`Headers`** tab, you will see four different sections. They are titled General, Response Headers, Request Headers, and Query String Parameters. So, what exactly are headers?

HTTP Headers let both the client and the server send additional information with the request or the response. The HTTP Headers consist of a case-insensitive name followed by its value. The HTTP headers can be grouped into four different categories:

- **General Headers**: These headers are applicable to both the request sent by the client and the response sent by the server. However, they are not related to the data inside the body or the content of the request or the response. The most common general headers are Date (it contains the date and time at which the message was originated), Cache-Control (it holds instructions for caching both requests and responses), or Connection (it specifies whether the connection should remain open after the current transaction finishes), and so on.

- **Request Headers**: These headers are applicable in the HTTP request, but they are not related to the content of the request. These requests send additional information about the request, so that it becomes easier for the server to send the accurate response. The request headers like Accept, Accept-* or If-* (where * denotes that it can be replaced by another keyword) are useful for adding conditions to the request. For example, Accept informs the server about the types of data that the server can send back. Similarly, Accept-Language informs the server about the languages that will be acceptable when the server sends the response. Apart from the conditional requests, you also have headers like User-Agent (which informs the server about operating system, software vendor, software version, and so on), Cache-Control (which informs the server about the cache policies of the response), Referer (which informs about the previous web page which called this web page), and so on.

- **Response Headers**: Response headers are used in HTTP response, and they are not related to the content of the message. They are useful for giving more details about the response.

  A few of the response headers include Age (which conveys the time since the client requested the server), Retry-After (which can be used to indicate the client how long the service is expected to be unavailable), Set-Cookie (it contains a name-value pair of information to retain for this particular URL), and so on.

- **Entity Headers**: Entity headers contain information about the body of the resource. Commonly used entity headers include Content-Length (the size of the resource in bytes), Content-Type (the media type of the resource, also known as MIME type), Content-Encoding (the compression algorithm for the resource), Content-Language (the human language intended for the audience), and so on. Entity Headers do not form a separate column on the **`Headers`** tab inside the **`Network`** panel. They are distributed amongst the **`Request`** and **`Response`** headers.

The Query String Parameters section that you see in the **`Headers`** tab on sending the request to the MealDB API server is the string attached to the end of the GET request. As we have read in the previous section, GET requests append additional information that needs to be sent to the server, to the URL of the request (hence, it is not safe!). Whenever a set of information is to be appended to a request, it can be achieved by adding the string after appending a question mark (?) to the end of the request URL.

Figure 9.17: *The Request URL of the API shows the Query String Parameters*

As we can see in the preceding screenshot, the Request URL that should end at **`search.php`**, contains an extra string ?s=Arrabiata. The string **`s=Arrabiata`** helps the server to filter and send all the recipes which have the name *Arrabiata* in the title. Whenever we need to add such a string to the URL, we first need to append a question mark symbol (?) at the end of the URL. This question mark symbol denotes that we are sending additional data to the server for a **`GET`** request. Then, the actual string(s) need to be added in the key=value format after the question mark. In case of multiple strings, we need to use the ampersand (&) symbol to separate the multiple strings. This addition to the URL is called as a *query string*.

For the sake of sanity, we have not printed the entire list of the header requests in this book. If you wish to have a look at the entire list of HTTP headers, you can check it out on the official MDN docs here – **https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers**.

Figure 9.17: *The Request URL of the API shows the Query String Parameters*

If you look at the brief list of HTTP headers shared before *figure 9.17*, or the complete list shown in the preceding link shared, you will figure that there might be several scenarios where you will need to send headers in the request. For instance, if you want to inform the server to send you the response only in particular formats, you will need to send the required extensions to the server. As we have seen in the preceding list shared, you will have to specify the required formats in the Accept header while sending the request to the server. For such instances, AJAX provides a method called **`setRequestHeader()`**.

setRequestHeader()

The **`XMLHttpRequest`** method **`setRequestHeader()`** sets the value of a HTTP Request Header. This method should be called only after you have opened a request, and before sending a request, as shown in the following screenshot:

Figure 9.18: *The exact moment when you should call setRequestHeader*

If this method is called several times with the same header, the values are merged into one single request header. In *figure 9.18*, we have specifically set the request header to accept only responses with the format '**`application/json`**'. If the **`Accept`** header is not set, the default value '*/*' is sent.

Conclusion

In this chapter, we explored and learned how to interact with a server using APIs. We started this chapter by understanding JSON – the format used for exchanging data across the internet. We applied this knowledge of JSON and its methods in the next section, AJAX, and the process of AJAX. In that section, we tried out a live example with a public API of the MealDB platform. However, before that, we familiarized ourselves with the different HTTP methods and the status codes. Finally, at the end of this chapter, we studied the various HTTP Headers. Everything that we have learned in this chapter, along with the knowledge of the past eight chapters, will help us in creating a live application in the next chapter.

In the next chapter, we will be using the APIs of the MealDB platform to create a web application for showing recipes. Here, we will not just write the code for creating the web application, but also look at the several non-technical aspects of creating an application, like how to zero in on the idea, looking at the target users, how to calculate the number of pages, and what needs to be done after the application has been created.

Points to remember

- For the sake of faster exchange and simplicity, data is sent across the internet between different entities in text format. **JavaScript Object Notation** (**JSON**) is the standard text-based format representing a JavaScript object syntax. It is used for exchanging data across the internet.
- The JSON and a JavaScript object literal are similar but not the same. Unlike the JavaScript object literal, you cannot use single quotes in JSON. Similarly, all the keys in JSON need to be wrapped in double quotes, and you cannot have functions, dates or undefined values for the JSON values.
- For web development, we use a protocol called **Hypertext Transfer Protocol** (**HTTP**) for exchanging data between clients (browsers) and servers. There are several HTTP methods for exchanging data with the servers, based on the requirements. The `GET` and the `POST` requests are more commonly used.
- The server always sends a status response to a request. Even if the request has failed, a status message is always returned to the client.
- Asynchronous JavaScript and XML (AJAX) is the standard process for interacting with servers. It uses a JavaScript object `XMLHttpRequest` for creating, opening, sending, and receiving a response.
- The benefit of using AJAX is that the rendering of the web page is not affected by the interaction with servers. In fact, even if a part of the UI needs to be changed, the browser does not need to refresh the page with AJAX.
- Finally, HTTP headers provide additional information to the servers and the clients about the requests, and the responses respectively. In the case of clients, they need to use headers to help the server curate accurate results. In case of servers, headers are sent to provide the client more information about the response.