5. Asynchronous JavaScript, AJAX and APIs

Obsah

1 Introduction to Asynchronous JavaScript

Let's first start with code example of synchronous code (when the things happen one after another). Fairly straight forward right? **First** logs out "hello from first" and calls the **second**, second logs out "Hello from second" and calls the **third**, third logs out "Hello from third" and that's it, the code has finished. Notice that I'm using regular functions instead of arrow functions. It's like that, because I found great tool for visualization of code execution (link is in section **1.3 How does asynchronous code work?),** but that tool accepts only regular functions.

```
function third() {
   console.log("Hello");
}

function second() {
   console.log("Hello from second");
   third();
}

function first() {
   console.log("Hello from first ");
   second();
}

first();
```

What do you think will happen in the next code example, when we put our console.log in **second** inside **setTimeout()**? We've seen **setTimeout()** already, but as a reminder the **setTimeout()** method calls a function or evaluates an expression (first parameter) after a specified number of milliseconds (second parameter).

First logs out "Hello from first" and calls the **second**, second runs the **setTimeout** BUT it doesn't wait for its execution to finish, instead of it calls the **third** right away. The **third** logs out "Hello from third" and after two seconds of waiting, our code from the **setTimeout** function is finished, and logs out "Hello from second". Why so? Well, because **setTimeout()** works asynchronously. But what is an async code, why is it useful, and how does that actually work? Well, let's find out!

```
function third() {
   console.log("Hello");
}

function second() {
   setTimeout(function () {
      console.log("Hello from second");
   }, 2000);
   third();
}

function first() {
   console.log("Hello from first ");
   second();
}

first();
```

1.1 What is an asynchronous code?

As I said, synchronous code is when the things in app happen one after another.

An asynchronous code on the other hand, allows multiple things to happen at the same time. When you start an asynchronous action, your program doesn't wait for that action to finish, it just jumps to another action and continues with the code execution. When the asynchronous action finishes, the program is informed and gets access to the result (for example, get data from the server). (1)

1.2 Why do we need asynchronous code?

JavaScript is single threaded, so it can run only one task at a time. (2) It means, that if you run some intensive chunk of code, nothing else works until that code is finished, so the app may seem like it is frozen. This is called **blocking**. (3)

Let's see some code blocking in code. In the following example, we have a button, and then a for loop that runs 30 000 times. That's a lot of cycles, so it takes some time until it finishes. While the loop is executing, nothing else can happen, so if the user clicks on the button, nothing happens until the for loop finishes and then finally, the click call-back function runs. In other words, for loop **is blocking** the click call-back function from its execution.

```
const btn = document.querySelector("#btn");

btn.addEventListener("click", () => console.log("btn clicked"));

console.log("before loop");

for (let i = 0; i < 30000; i++) {
   console.log(i);
}

console.log("after loop");
```

I admit, that the example above is not really something from the real world, but it demonstrates a problem of synchronous code. So, what are the real-world situations, when asynchronous code is very useful? Let's list few of them: (2)

- Requesting data from the server
    - This is a very common task, but not an easy one. You first make a request, then that request is sent via the internet to the server, then the server needs to process your request and get the data you want and send it to you via the internet again. This simply takes a lot of time, right? It would've been a terrible user experience, if your app would just be frozen until you get the data from the server. In real case scenario, you want to show some loading screen to your user, and allow him to abort the request, if it takes too long.
- Accessing a video stream
- Processing uploaded file

1.3 How does asynchronous code work?

Great talk about it - https://www.youtube.com/watch?v=8aGhZQkoFbQ

Awesome visualization tool for visualising code execution (I took it from the video above) - http://latentflip.com/loupe/?code=ZnVuY3Rpb24gdGhpcmQoKSB7CiAgICAgICAgY29uc29sZS5sb2coIkhlbGxvIik7CiAgICAgIH0KCiAgICAgIGZ1bmN0aW9uIHNlY29uZCgpIHsKICAgICAgICAvLyBzZXRUaW1lb3V0KCgpID0%2BIGNvbnNvbGUubG9nKCJIZWxsbyBmcm9tIHNlY29uZCIpLCAyMDAwKTsKICAgICAgICBzZXRUaW1lb3V0KGZ1bmN0aW9uICgpIHsKICAgICAgICAgIGNvbnNvbGUubG9nKCJIZWxsbyBmcm9tIHNlY29uZCIpOwogICAgICAgIH0sIDUwMDApOwogICAgICAgIHRoaXJkKCk7CiAgICAgIH0KCiAgICAgIGZ1bmN0aW9uIGZpcnN0KCkgewogICAgICAgIGNvbnNvbGUubG9nKCJIZWxsbyBmcm9tIGZpcnN0ICIpOwogICAgICAgIHNlY29uZCgpOwogICAgICB9CgogICAgICBmaXJzdCgpOw%3D%3D!!!

2 Types of async code in JavaScript

In JavaScript, there are two types of async code, we've actually used one of them a lot in the class about the DOM, can you guess what it is?

- Async callbacks
- Promises

2.1 Async callbacks

We used this one, when we declared event listener in the class about the DOM. That callback function which is declared as a second parameter of **addEventListener()** is asynchronous callback. (2)

It makes sense, right? You declare what should happen after for example a button click, and that logic is executed after that button click. It doesn't run synchronously, it simply runs when needed. (The code execution does not get stuck when it reaches some event listener.)

2.2 Promises

Promises are another type of asynchronous code. They are used for example, when you want to make a request to a server, or when you want to read some file.

Promise has three states: (4)

- **pending**: initial state, neither fulfilled nor rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.

Most of the time, you don't create your own promises, as you mostly just consume them, but for the sake of learning, let's create a simple promise, so you know how it looks and how to work with it.

Let's create a promise, that returns a user data after 3 seconds. There are two methods in promise - **resolve** and **reject**. **Resolve** method is called, if the promise worked as expected (e.g. server sent us the data we wanted) (5)

```
const getUser = new Promise((resolve, reject) => {
   setTimeout(() => {
      resolve({ name: "Mark", age: 28, nationality: "UK" });
   }, 3000);
});
```

Promises sometimes don't end up successfully, for example, when we request a data from the server, but those data are simply not there, or we can't connect to the server, because of bad internet connection. That's why there is that second method, **reject**. **Reject** method usually returns some reason, why the promise wasn't successful. (5) Let's rewrite previous example a bit, to include reject if the user doesn't exist.

In the following example, we check whether the user exists, if it does, the Promise is successfully resolved, if it doesn't the promise is rejected, and new error is created with a message, that user is not found.

```
const getUser = new Promise((resolve, reject) => {
   const user = { name: "Mark", age: 28, nationality: "UK" };
   setTimeout(() => {
      if (user) {
         resolve(user);
      } else {
         reject(new Error("User not found"));
      }
   }, 2000);
});
```

2.2.1 .then() and .catch() methods

Okay, we have seen how to create a promise, but how to work with them? We need to work with that data we receive in the Promise response, right? Turns out, that there are two ways to deal with promises. First one is **.then()** and **.catch()**.

Let's use them to deal with our Promise from the previous code example. It's fairly simple. What we say here is: "Call a promise **getUser**, if that promise responses successfully, **then** log out received user data, if not, then **catch** the error and log that out"

```
getUser
   .then((res) => console.log(res))
   .catch((err) => console.log(err));
```

Sometimes, you need to deal with more promises at once. That's why, you can chain **then**. Let's have a look at that.

First, add a function, that returns new promise, so now we have the following. First is our promise that we already know, and second thing is a function, that accepts a user returned from our promise, and returns another promise, that processes our user and returns his name.

```
const getUser = new Promise((resolve, reject) => {
   const user = { name: "Mark", age: 28, nationality: "UK" };
   setTimeout(() => {
      if (user) {
         resolve(user);
      } else {
         reject(new Error("User not found"));
      }
   }, 2000);
});

const getUserName = (user) => {
   return new Promise((resolve, reject) => {
      setTimeout(() => {
         resolve(user.name);
      }, 2000);
   });
};
```

To deal with this, we need to chain then. In the following example, a response received from the promise is passed to a function, that returns another promise, which we need to wait for as well to log out the result of that promise (e.g. name of given user). You can chain catch as well, but it's just better to have only one catch at the very bottom, as that one catches all potential errors on the way. (6)

```
getUser
   .then((res) => getUserName(res))
   .then((name) => console.log(name))
   .catch((err) => console.log(err));
```

2.2.2 Async/Await

This is the second and more modern way to deal with promises, although it's not necessarily better. (7) (8).

An async function is a function declared with the **async** keyword. Async functions are instances of the **AsyncFunction** constructor, and the **await** keyword is permitted within them. The **async** and **await** keywords enable asynchronous, promise-based behaviour to be written in a cleaner style, avoiding the need to explicitly configure promise chains. (9)

Uf, okay, so what does the definition above means in plain English? It basically says, that it deals with promises, but in an easier, cleaner and easily readable style than **then/catch**.

How is it different from **then/catch**? Well in the following code example, you can see that first, the function is defined with a keyword **async**. Thanks to this word, you can use

await inside this function (in order to use **await** the function needs to be **async**) and also an **async** function returns a promise.

Let's ignore **try/catch** keywords for a second and focus on what is inside **try**. You can notice **await** keyword there, so what is it doing?

The **await** expression causes **async** function execution to pause until a **Promise** is settled (that is, fulfilled or rejected), and to resume execution of the **async** function after fulfilment. When resumed, the value of the **await** expression is that of the fulfilled **Promise**. (10)

In simple words, it allows you to write asynchronous code in a synchronous way, but never forget, even though it looks synchronous, it is NOT.

So, in the following example, first **getUser** promise is resolved, then **getUserName** is resolved, and after all that, name returned from **getUserName** is logged into the console.

Good video about it - https://www.youtube.com/watch?v=568g8hxJJp4

```
const fetchUser = async () => {
   try {
      const res = await getUser;
      const name = await getUserName(res);
      console.log(name);
   } catch (e) {
      console.log(e);
   }
};
```

2.2.3 Side note – try/catch

Even though in the example above it is used in combination with asynchronous code, it is not strictly tight to it.

The **try** statement allows you to define a block of code to be tested for errors while it is being executed. The **catch** statement allows you to define a block of code to be executed, if an error occurs in the **try** block. (11)

So, in the example above, the interpreter **tries** to run our asynchronous task (**try** statement), and if it is not successful, it **catches** the error in the **catch** statement.

But as I said, **try/catch** can be used anywhere we want. For example, in the following example, I'm trying to log out a variable that doesn't exist, so the error is caught in the **catch** statement.

```
try {
   console.log(someNonExistingVariable);
} catch (e) {
   console.log(e);
}
```

try/catch is simply useful, when you want to handle exceptions by yourself, as it allows you to have your own custom error handling.

2.2.4 .then()/.catch() vs Async/await

There is not much to compare between those two approaches BUT, one thing is very important. Every time you write an **await**, you're writing a blocking code. (7)

Let's come back to our previous **async/wait** code example. I said there, that **name** is logged into the console, after two previous promises are resolved, but console.log is not written asynchronously in any way, right? That's what I meant, when I said, that **await** is blocking. It simply blocks the code in **async** function until that **await** promise is resolved.

Then()/catch() is not blocking, so what is logged out from the **console.log**, in the following code? Well, **undefined** is returned. Why? Because only blocks of code in **then** and **catch** run asynchronously, the rest of the code runs synchronously. So, what actually happens is, that promise is executed asynchronously, but **console**.**log** doesn't wait until the promise gets resulted, it runs synchronously right away.

```
const fetchUserThen = () => {
   let name;
   getUser
      .then((res) => getUserName(res))
      .then((n) => (name = n))
      .catch((err) => console.log(err));
   console.log(name);
};

fetchUserThen();
```

If I want to log out the name, I have to write that **console.log** inside **then** block of code as shown in the following example.

```
const fetchUserThen = () => {
   let name;
   getUser
      .then((res) => getUserName(res))
      .then((n) => {
          name = n;
          console.log(name);
      })
      .catch((err) => console.log(err));
};

fetchUserThen();
```

Otherwise **async/await** is not that much different in comparison to **then/catch**. **Async/await** is just an alternate way of writing and dealing with promise-based (asynchronous code) (12). So even though it is generally being said, that async/await is easier to read, no one can stop you from using then/catch, as you can achieve the same behaviour with it. (13)

3 Dealing with servers

Now it gets a little bit more complicated as there are many new words we need to introduce.

With HTML, CSS and JavaScript alone, you can create a nice web application, that is able to interact with users and present them with some content. Usually, the content is not hard coded (I mean, that it is not written inside our files with HTML, CSS and JavaScript code). Why? Well, in many apps we need to allow users to create their own content – create reviews, post images, write articles, etc. But things that are hard coded inside our files with code, cannot be changed without actually opening those files writing directly into them. We don't force our users to do that, as that would be a terrible user experience, plus we don't want to expose our code base to everyone that easily. And another bad thing about hard coding is, that many times we don't need all data that our app contains. We need just certain pieces of that data, that are actually present in the current page of our web app.

Wouldn't it be great to have some storage, where all our application data is stored? Some place, from which we would be able to retrieve whatever piece of data we currently need? Well that's exactly why servers are used in web apps. In very simple words, A server is a computer that provides data to other computers. (14)

3.1 Client-Server communication in web apps

So, there are usually two sides in web apps. Client – that is the browser where our application runs. Server – some computer that acts as a storage for our application. These two sides are partners that work side by side to make our app running smoothly. But as all partners, they need some way to communicate to each other. This is a subject about JavaScript, so let's just say it in simple words, to get the general idea. Server and client (browser) communicate with each other via HTTP protocol (this protocol is basically a set of rules that are describing how the communication should happen).

This communication is done by requests and responses. (15)

1. A client (a browser) sends an **HTTP request** to the web (for example when the user wants to login, or when the user wants to open some article)
2. A web server receives the request
3. The server runs an application to process the request
4. The server returns an **HTTP response** (output) to the browser
5. The client (the browser) receives the response

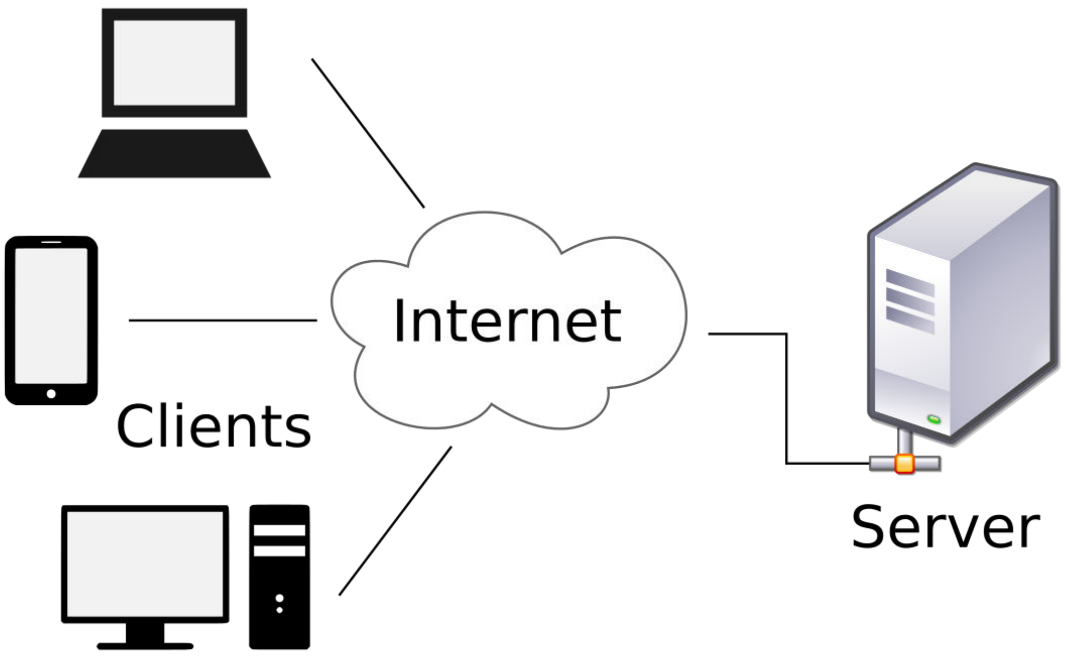


Figure 1 - client-server communication schema. Source: (16)

3.2 What is an API

In JavaScript, the communication with server is very often done via **Fetch API**. So, let's first explain, what is an API.

API stands for Application Programming Interface. As a programmer, there is no way that you will avoid working with APIs. They are simply everywhere. So, what are they?

Well, you can think of an API like a menu in a restaurant. The menu provides a list of dishes you can order, along with a description of each dish. When you specify what menu items you want, the restaurant's kitchen does the work and provides you with some finished dishes. You don't know exactly how the restaurant prepares that food, and you don't really need to. (17)

Okay, that was just a metaphor to get a rough idea, so let's be a bit more concrete. Imagine that you are developing a web app, and you want to access a device camera. You can definitely do that in modern browsers. Thanks to an API provided by browser, you can just call provided commands and that's it! You can have access to user's camera in the matter of seconds, without having to program your own code implementation of accessing the camera. You just ask the API to do something, and you don't care how it is actually done.

We've actually worked with one API already. Can you guess what it was? It was the DOM! Let's remember what the DOM is. The DOM is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. (18) It makes sense, right? We just used some provided commands, that allowed us to access DOM elements and manipulate with them.

Good videos about API explanation:

- https://www.youtube.com/watch?v=s7wmiS2mSXY
- https://www.youtube.com/watch?v=6STSHbdXQWI

3.3 Fetch API

Now that we've gone through the basic idea of how communication between the client and the server works and explained what an API is, let's discuss how it exactly works in modern web apps.

As I mentioned in the previous section, the communication is done via **Fetch API**. The Fetch API provides an interface for fetching resources (including across the network). (19) This is a great example of an API, as it allows you to make request to servers via simple commands. You don't need to create your own implementation of fetching data from servers. Thanks to Fetch API, you just need to say what you want to do via prepared commands, without the need to specify, how exactly it should happen. That's something what the browser is doing for you.

Let's have a look at the code example. Fetch needs two parameters, the URL to which it should send given request, and method of that request (GET, POST, PATCH, DELETE, …), if you don't provide a method to fetch, it uses GET method, but I put the method there, so it is more obvious that the method is important. Fetch sends request to the given URL across the network and then waits for the response. This step takes some time, that's why it is asynchronous. Once the server sends a response to the browser, then that response is returned to us, so we can work with it. The following code is the simplest form of a fetch call, there is no error handling, and the response is not really readable. This is just for you to understand the basic principle, we will go through proper calling in the next section, don't worry.

```
const fetchData = async () => {
   const response = await fetch(
      `http://jsonplaceholder.typicode.com/posts`,
      {
         method: "GET",
      }
   );
};

fetchData();
```

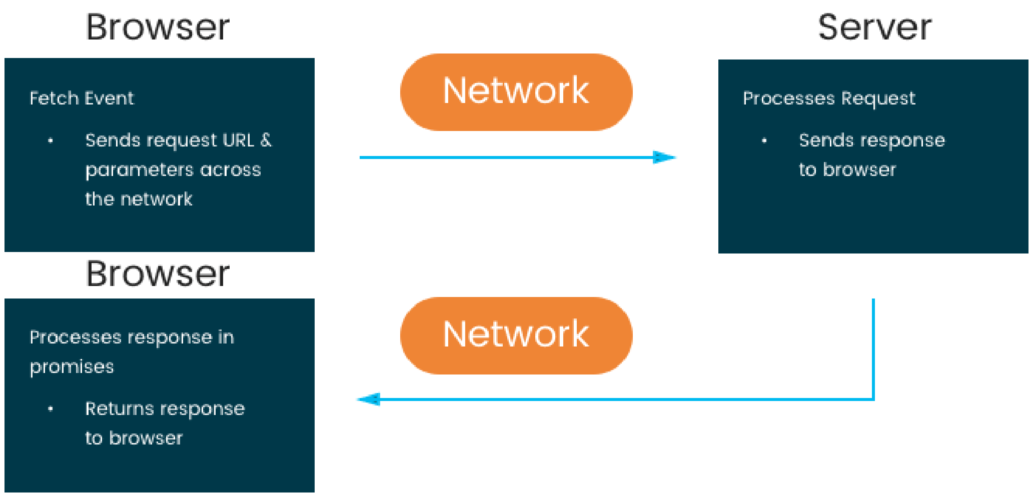

Browser
Fetch Event
Sends request URL & parameters across the network
Network
Server
Processes Request
Sends response to browser
Browser
Processes response in promises
Returns response to browser
Network

Figure 2 - Fetch API diagram. Source: (20)

4 REST and RESTful API

In many modern JavaScript applications, when you're using Fetch API, you are mostly dealing with RESTful APIs as well. In fact, that's what we've done in the previous section **3.3 Fetch API** when we made a request to **jsonplacehoder**, and we will interact with another RESTful API in the next section, where we will be creating very simple app to see some more real-world examples of async code and Fetch API.

Before I explain what the RESTful API is, let's first say, why it is needed. As we've learnt, majority of web apps has a client and a server. These two communicate with each other via the HTTP, BUT this alone is not enough for communication. HTTP takes care about the actual data transfer, but there is one thing missing. Some rules of how the HTTP should be used. Well this is where **REST** and **RESTful** comes into play.

This is going to be the last heavily theoretical part of this class, so bear with me.

4.1 RESTful API

What is RESTful API? Well, RESTful API is an API, that follows REST architectural style (set of constraints). (21)

Little warning – on the internet, you can notice that words REST and RESTful are used interchangeably. It is like that, because REST is an architectural style, and RESTful is the interpretation of it. (22) This sounds very confusing, but basically, when you create an API following REST architecture style, then that API is RESTful API. But it doesn't matter otherwise.

4.2 REST

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. (23)

It means when a RESTful API is called, the server will **transfer** to the client a **representation** of the **state** of the requested resource. Representation of that state is mostly in JSON format (21)

As I said, REST has set of constraints, but details of those constraints are not that important for us to know, as we want to just communicate with some server, not create it.

If you want to know more, you can visit one of the listed websites:

- https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f
- https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582
- https://www.geeksforgeeks.org/rest-api-architectural-constraints/
- https://www.mulesoft.com/resources/api/what-is-rest-api-design

4.3 How RESTful API works

As I said, we don't really need to know all those constraints of REST, BUT we need to know, how RESTful API works, so we can work with it properly.

So, RESTful API follows REST architectural style. I'm still repeating, that REST is just an architectural style, why? Well, because it means, that its basically just a set of theoretical guidelines, that you don't need to follow exactly, in order to make your API work. But if you don't follow those guidelines, you just make it much harder for other programmers to work with your API. We are not going to create our own RESTful API, but this is important to realize, because it also means, that not all APIs that you are going to work with, are going to be really RESTful, so you will have to adjust to it.

Let's finally explain, how RESTful API works.

In simple words. REST is a way for two computer systems to communicate over HTTP in a similar way to web browsers and servers. (24)

That's a very short explanation, so let's expand that a bit.

So, you have the client (your web app) and the server, where is a database with data that you want to work with.

In order to work with databases, you need to use some server side language like PHP, Python or JavaScript (NodeJS), and create a backend code, where you need to write SQL statements, and other functions to process that data selected from the database so they are in a form you need it. When you work with RESTful API, all that backend heavy lifting is already done for you. It's like a completely different world, that just works, and you don't need to care about how. But you need to communicate with it, in order to get from there the data, you want. In other words, you have your web app (one world) and the backend service (another world), and you need a middleman that allows those two to communicate with each other. Hmm, middleman, you heard that metaphor already, right? What serves as a middleman in computer world? That's right! An API!

So how do they communicate with each other? REST relies heavily on HTTP and allows us to do CRUD (create, receive, update, delete) operations. (22)

Each operation uses its own HTTP method: (22)

- GET - getting
- POST - creation
- PUT/PATCH - update (modification)
- DELETE – removal

The actual communication goes as follows: Client (web app) makes a request via the HTTP. It does that, by specifying the URL of an API endpoint (url), and HTTP method (GET/POST/PUT|/PATCH/DELETE). Those URL endpoints are usually specified in the RESTful API documentation. When the RESTful API receives a request, it then executes logic specified for that given endpoint and sends a response. This response usually includes data and http status code (for example 200 – OK – for successful request). For example, we make a request to url "API/users" and the API sends a list of users in the response.
All transferred data between the web app and the RESTful API are usually exchanged in JSON format, but XML or CSV can be used as well. (24) JSON is mostly used just because

it’s very lightweight, and it’s very easy for humans to read and write and it is easy for machines to parse and generate. (25)

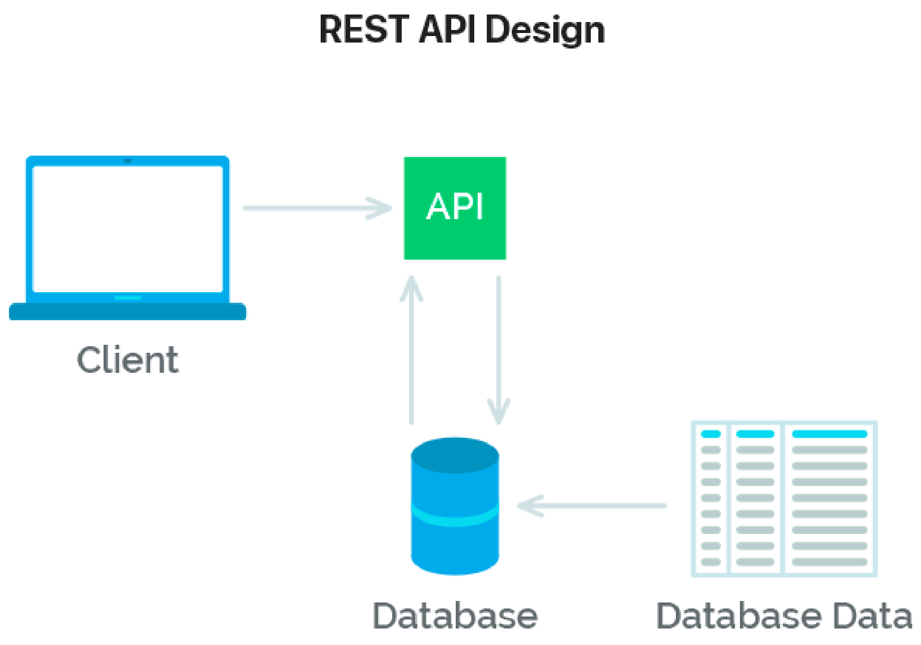


Figure 3 - RESTful API principle schema. Source: (22)

Interesting videos about RESTful API principle:

- https://www.youtube.com/watch?v=7YcW25PHnAA
- https://www.youtube.com/watch?v=SLwpqD8n3d0

5 App – What is your nationality?

5.1 Instructions

This app is very simple. It asks for the user’s first name. When the user enters his/her first name, then the app makes a request to the third party RESTful API https://nationalize.io/ where that first name is analysed, and the user’s nationality is guessed based on that first name. This guess, including the probability, is then sent as a response to our app. Our app then processes that response and presents that response’s data to our user in a modal.

5.2 How final app is going to look like

5.2.1 Initial app load

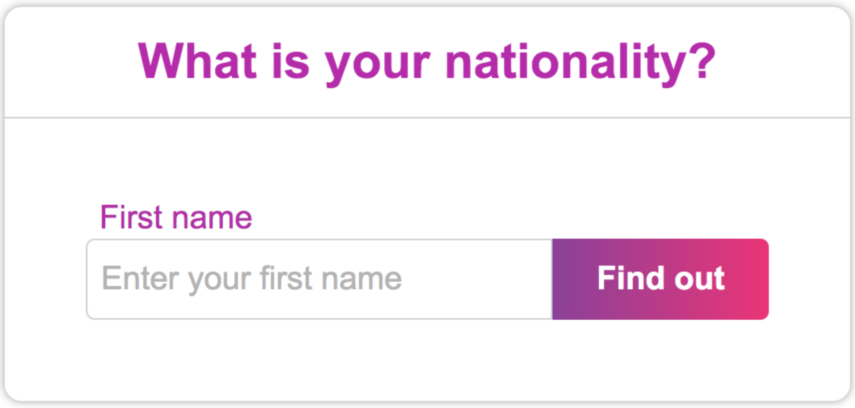


Figure 4 - Picture of an application's initial load. Source: my own image

5.2.2 When the user enters his/her first name

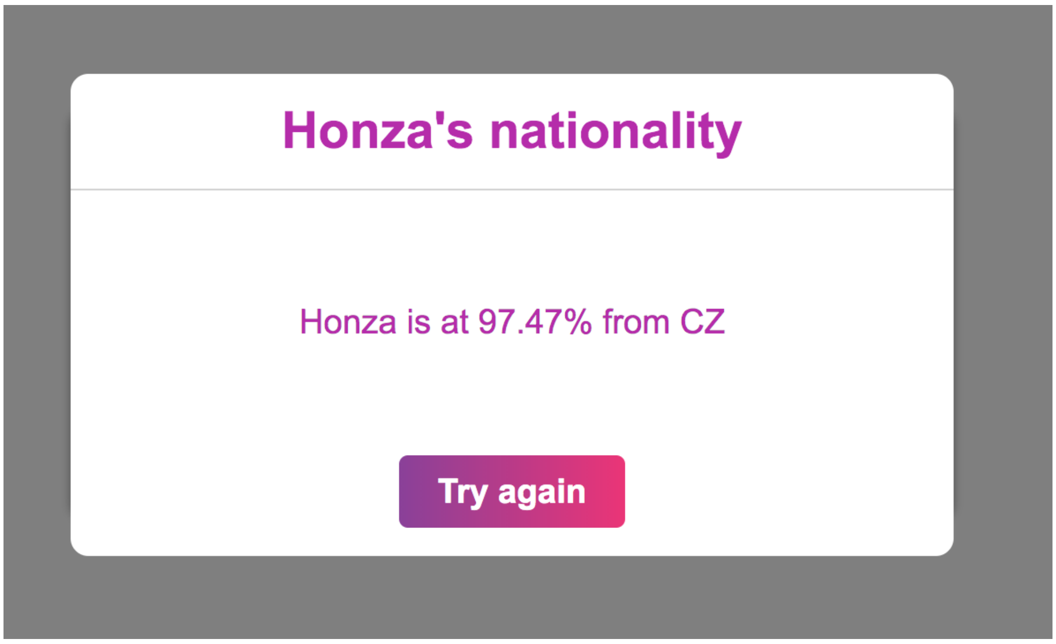


Figure 5 - Picture of an app modal, where data from the nationalize.io API are presented to the user. Source: my own image

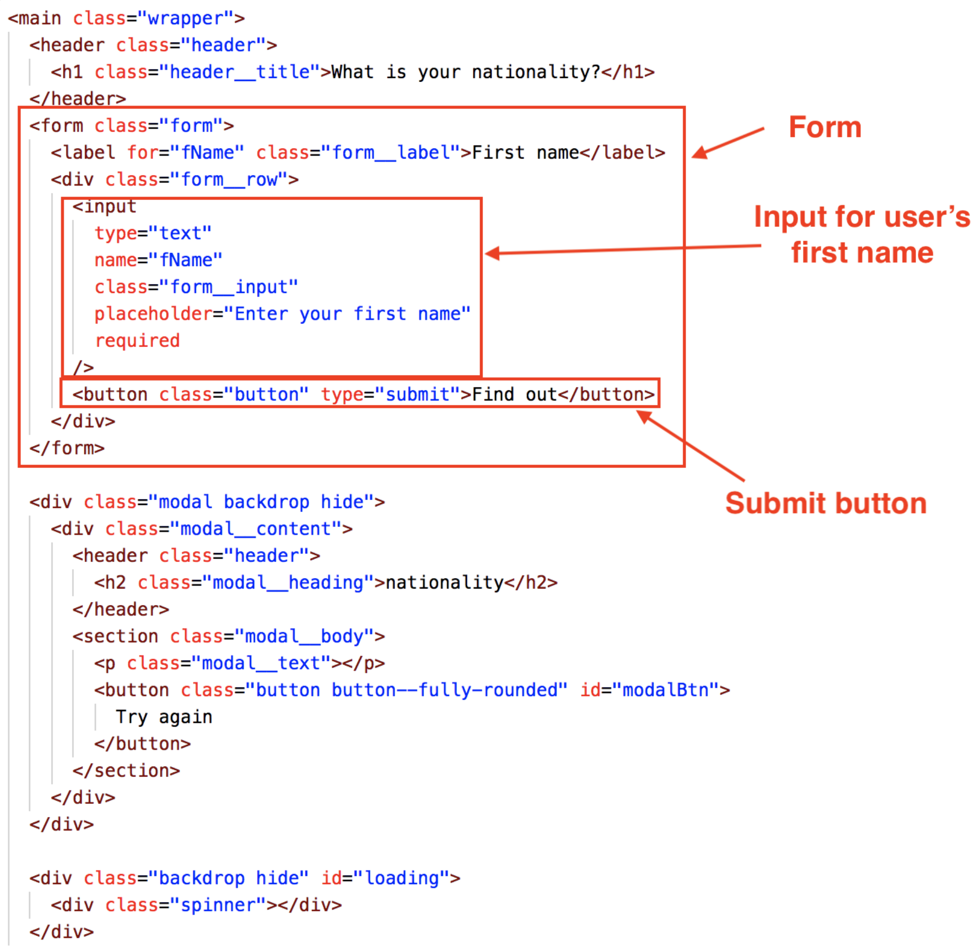
5.3 First steps

To save us some time, we are already given an HTML code with some styling. So, what do we need to do first? Well, we have very simple form with just one input (for the user's first name) and a button for submitting the form. Let's inspect our HTML code to see, what elements we need to select and work with. This app is a little bit more complicated that the one from the previous class, so let's inspect only part of the HTML code, we need to work with right now, and ignore the modal code for a moment.

As you can see in the following image, where is the HTML code of our app, three elements are probably important for us. Form, input and submit button. I said probably, because now you have two options to achieve our desired functionality. You can either select the input and the submit button OR you can just select the form itself, and extract all you need from the form. I chose the second solution. But why can I do it? Well, the form itself, contains all its inputs, so all I need to do, is assign name attributes to those inputs. And another thing, if you have a button inside a form, it is considered as a submit button of that form. Thanks to that, I don't need to listen to the click event, but I can actually listen to the form's submit event.

Just so you know, I also added **type="submit"** to that button just so it's clearer to you, but I don't even need to do that. All buttons inside forms are considered as submit buttons. If you don't want those buttons to behave as submit buttons, you need to assign them **type="button".**

Anyway, what is important for me right now is, that I don't need to listen to the click event,



```html
<main class="wrapper">
  <header class="header">
    <h1 class="header__title">What is your nationality?</h1>
  </header>
  <form class="form">
    <label for="fName" class="form__label">First name</label>
    <div class="form__row">
      <input
        type="text"
        name="fName"
        class="form__input"
        placeholder="Enter your first name"
        required
      />
      <button class="button" type="submit">Find out</button>
    </div>
  </form>

  <div class="modal backdrop hide">
    <div class="modal__content">
      <header class="header">
        <h2 class="modal__heading">nationality</h2>
      </header>
      <section class="modal__body">
        <p class="modal__text"></p>
        <button class="button button--fully-rounded" id="modalBtn">
          Try again
        </button>
      </section>
    </div>
  </div>

  <div class="backdrop hide" id="loading">
    <div class="spinner"></div>
  </div>
```

Figure 6 - HTML code of the app with first part of highlighted code. Source: my own image

5.4 Working with the form

To start, let's select the form.

```javascript
const form = document.querySelector(".form");
```

Now I can attach an event listener to it, where the first thing we need to do is to **preventDefault**, because we don't want our page to reload after form submit. Then we can make a call to nationalize.io API. In order to do that, we need to use Fetch API. Wait a minute, we use an API, so we can work with another API? Well, it can sound a bit weird and confusing, but yes. In many cases, you need to work with multiple APIs to achieve your goal. In this case, we want to get data from nationalize.io, so we need to use its API, and to do that, we can to make an asynchronous HTTP request, for which we need to use Fetch API.

Let's come back to our code. In order to retrieve the data about some name, nationalize API expects to get a name parameter from us (that thing after the question mark). We need to insert the user's name into that param. As I said, we can access that via the form in combination with the input's name – **e.target.fName.value**.

Then the method of the request is defined, which is GET in this case. Then the request is made, and nationalize.io sends us a response. This is still not enough, because the body (place, where the data are) of the response is in ReadableStream. We need to convert that into something readable. That is done via method .json(). We can execute this method on our response. This method returns another promise, where we can finally find our desired data in readable format (as objects or arrays).

But now you can notice, that not only one country guess is returned. It actually returned three countries, with the highest probabilities. What to do know? This is a great place to use higher order function **.reduce()**! We haven't talked about this one yet.

.reduce() method reduces an array to a single value. How? Method executes a provided function for each value of the array (from left-to-right). The return value of the function is stored in an accumulator (result/total). (26)

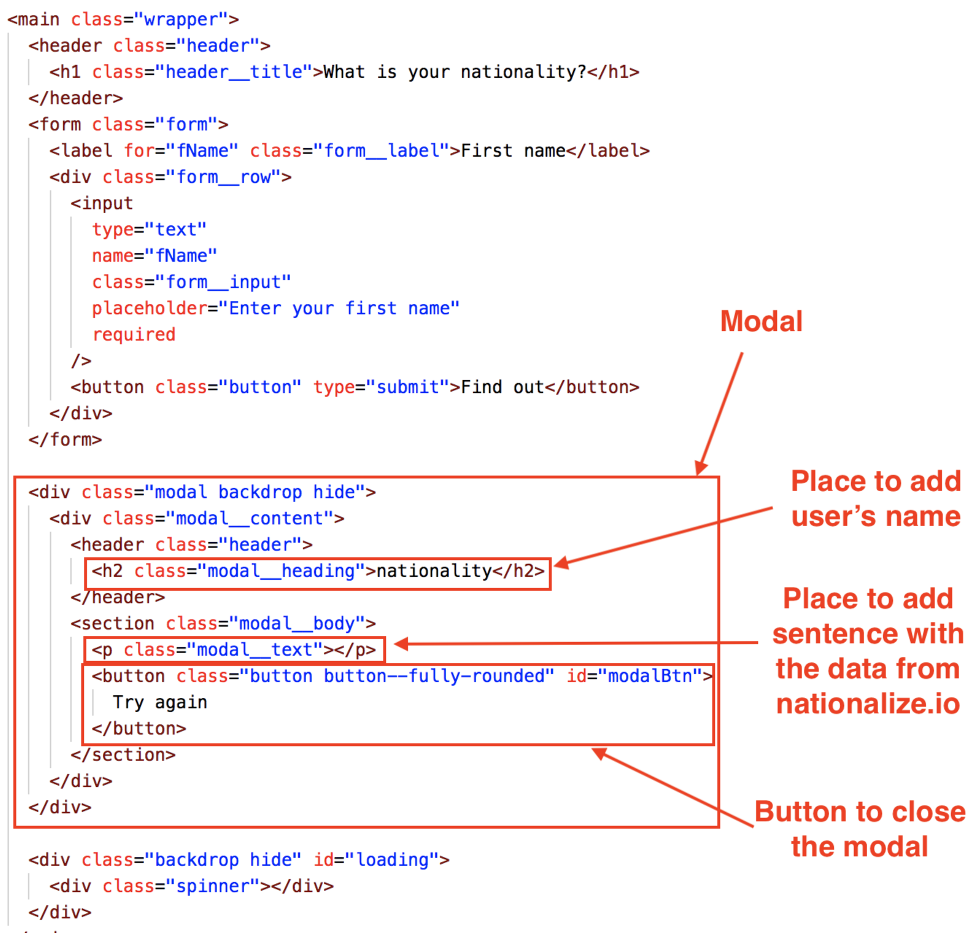
In our provided function, we said, that each element should be compared to our current maximum, and if the element is higher, then it should become the new maximum.

Now that we have our processed response, where we have the inserted name and nationality, where is the country code of guessed nationality as well as the probability of that guess, we need to send this data to our modal. This is done by calling **renderNationalityModal** function, that renders modal with the response.

```
form.addEventListener("submit", async (e) => {
   e.preventDefault();
   //https://api.nationalize.io/?name=moris
   const res = await fetch(
      `https://api.nationalize.io/?name=${e.target.fName.value}`,
      {
         method: "GET",
      }
   );

   const jsoned = await res.json();

   const highestProbability = jsoned.country.reduce((max, country) =>
          max.probability > country.probability ? max : country);

   const user = { name: jsoned.name, nationality: highestProbability };

   renderNationalityModal(user);
}
```

5.5 Rendering the modal

We have the data we need, now it's just the matter of inserting them into the places we want. Let's inspect our HTML again.



```html
<main class="wrapper">
  <header class="header">
    <h1 class="header__title">What is your nationality?</h1>
  </header>
  <form class="form">
    <label for="fName" class="form__label">First name</label>
    <div class="form__row">
      <input
        type="text"
        name="fName"
        class="form__input"
        placeholder="Enter your first name"
        required
      />
      <button class="button" type="submit">Find out</button>
    </div>
  </form>

  <div class="modal backdrop hide">
    <div class="modal__content">
      <header class="header">
        <h2 class="modal__heading">nationality</h2>
      </header>
      <section class="modal__body">
        <p class="modal__text"></p>
        <button class="button button--fully-rounded" id="modalBtn">
          Try again
        </button>
      </section>
    </div>
  </div>

  <div class="backdrop hide" id="loading">
    <div class="spinner"></div>
  </div>
```

Modal
Place to add user's name
Place to add sentence with the data from nationalize.io
Button to close the modal

Figure 7 - Figure 6 - HTML code of the app with second part of highlighted code. Source: my own image

We have to select multiple elements. Input – to clear it and blur it out. Modal, so we can open/close it. Modal heading, to add user's name, so it's clear what the response is for (it's always good to be as clear as possible, as it improves the UX). Modal text, so we can show the response data to the user in clear and human readable way and modal button, so the user can close the modal.

```javascript
const input = document.querySelector(".form__input");
const modal = document.querySelector(".modal");
const modalHeading = document.querySelector(".modal__heading");
const modalText = document.querySelector(".modal__text");
const modalBtn = document.querySelector("#modalBtn");
const loading = document.querySelector("#loading");
```

Now that we selected all the needed elements, let's finally create the logic for modal rendering. The first thing you can notice is the way that the function params are accepted. Do you remember what this is? It's the destructuring! Whenever an object is accepted as a parameter to some function it's always a good idea to use destructuring, as it just makes the code more readable and easier to work with. Following lines are fairly straight forward. First the input value is cleared out, so the user can enter new name right away after he/she closes the modal. Input is blurred as well (blurred = loses focus). Then the **handleModalChange** is called, I will show that function in a minute, but it's fairly simple function that just shows or hides modal. Then the data we retrieved from nationalize.io are inserted into the DOM element, where we want our text to be.

```
const renderNationalityModal = ({ name, nationality }) => {
   input.value = "";
   input.blur();
   handleModalChange(true);
   modalHeading.innerHTML = `${name}'s nationality`;
   modalText.innerHTML = `${name} is at ${formatProbability(
   nationality.probability
   )} from ${nationality.country_id}`;
};
```

There was one extra thing in the previous code example - **formatProbability** function. It is there, because probability from nationalize.io is in range from 0 to 1. For people in general, it's better to have range from 0 to 100 with percentage sign, that's exactly what this function is doing.

```
const formatProbability = (num) => `${Math.round(num * 10000) / 100}%`;
```

Just to be clear, let's show the **handleModalchange** method. It just adds/removes hide class, which either sets **display: block** (initial display value for div), or **display: none**. I did it this way, as there is no need to be fancy as in the previous app. It would just add more complexity.

```
const handleModalChange = (show) =>
     show ? modal.classList.remove("hide") : modal.classList.add("hide");
```

The only thing missing now, is the event listener for our close modal button, so let's code that. It just calls our **handleModalChange** method and closes the modal.

```
modalBtn.addEventListener("click", () => handleModalChange(false));
```

5.6 Important improvement – error catching

Our app works, BUT there is one major flaw. Our Fetch call can go wrong, but we have no way of telling what went wrong, as we don't catch errors at all! In this small app, you can probably guess that the problem is probably in the fetch call, but in normal apps, you usually dozens of fetch calls and other logic that can go wrong, so it's always good to catch errors in some way, so the app itself tells you what went wrong as that can save you from anxious console logging of everything you can think of.

Let's improve our fetch call so it catches errors. The first thing to do is definitely wrap our fetch call in **try/catch**. BUT this is not enough. Fetch API by default, throws you an error ONLY when a network error is encountered or CORS is misconfigured on the server-side. (27) Basically, it throws an error, if you are not connected to the internet or if the URL you gave it is wrong. But it doesn't throw an error, if you send wrong params in the url to the server, or if the server didn't find the data you wanted, or if you request data that you don't have permissions for. That may sound weird at first, but it makes sense when you think about it. I mean, the Fetch was executed correctly right? You sent a request, and the server responded, so there is no reason for the Fetch API to throw an error.

So how to catch errors that are sent in the response? Well, first thing you can definitely do, is to check whether the response property **"ok"** is true or false. (27) It is ok (true) if the response status code is in the range from 200 to 299. It is like that, because this range is used for successful responses. (28) If "**ok"** property is false, then you can throw an error, that includes property **statusText** where you can find out a bit more about why the response is not successful, it can be for example wrong parameter.

There are also many other potential errors. For example, in our code, we expect to get property country, which contains an array of guessed nationalities with its probabilities. What can happen is, that the server doesn't have any data for given input (for example for Czech names with diacritics). So, the response is successful, but our country array is empty. We should check that as well and this is what we do in the following code of course.

```
try {
   const res = await fetch(
      `https://api.nationalize.io/?name=${e.target.fName.value}`,
      {
         method: "GET",
      }
   );

   if (!res.ok) throw new Error(res.statusText);

   const jsoned = await res.json();

   if (jsoned.country.length === 0)
        throw new Error("Empty array returned from the promise");

   const highestProbability = jsoned.country.reduce((max, country) =>
           max.probability > country.probability ? max : country
   );

   const user = { name: jsoned.name, nationality: highestProbability };

   renderNationalityModal(user);
   } catch (err) {
      console.log(err);
   }
});
```

5.7 UI improvement – loading

Now we are finally catching errors, so nothing can surprise us. I mean, we are only logging those errors into the console, in real apps, you should show these errors to the user as well, so he knows what happened. But for this demo app, this is enough, and we are happy. But some users are not happy, why? Because they have slower internet connection, so it takes a bit longer before they get the response from the server. But as the app is right now, whenever they insert a name, nothing happens for a while as the app is waiting for the response from the server so it can present the response to the user. The user has no way of telling if the request was actually sent and now the app is waiting for the response, or if it is just stuck or ignoring user's input. What to do with this? We should add a loading screen, so the user knows what is happening.

Loading screen is already prepared for us in the HTML code, so we just need to grab it in our code.

```
const loading = document.querySelector("#loading");
```

Let's create a logic for that loading screen. It's just a simple if else that adds/removes class with **display:none**. The same type of logic as for **handleModalChange**.

```
const handleLoadingChange = (isLoading) =>
   isLoading
      ? loading.classList.remove("hide")
      : loading.classList.add("hide");
```

Now, when do we need to show our loading screen? Well, we need to show loading when before the fetch call is executed, and hide it, after we get the response or when some error is thrown.

```
try {
   handleLoadingChange(true); //SHOW LOADING BEFORE FETCH CALL

   const res = await fetch(
      `https://api.nationalize.io/?name=${e.target.fName.value}`,
       {
          method: "GET",
       }
     );

     if (!res.ok) throw new Error(res.statusText);

     const jsoned = await res.json();

     handleLoadingChange(false); //HIDE LOADING AFTER YOU GET A RESPONSE

     if (jsoned.country.length === 0)
       throw new Error("Empty array returned from the promise");

     const highestProbability = jsoned.country.reduce((max, country) =>
        max.probability > country.probability ? max : country
     );

     const user = { name: jsoned.name, nationality: highestProbability };

     renderNationalityModal(user);
    } catch (err) {
     console.log(err);
     handleLoadingChange(false); //OR HIDE IT WHEN SOME ERROR IS THROWN
    }
});
```

6 Citovaná literatura

1. **Haverberke, Marijn.** Asynchronous Programming. *eloquentjavascript.net.* [Online] https://eloquentjavascript.net/11_async.html.
2. **© 2005-2020 Mozilla and individual contributors.** Introducing asynchronous JavaScript. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing.
3. —. General asynchronous programming concepts. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts#JavaScript_is_single_threaded.
4. —. Promise. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.
5. **Eygi, Cem.** JavaScript Promise Tutorial: Resolve, Reject, and Chaining in JS and ES6. *freeCodeCamp.* [Online] 8. June 2020. https://www.freecodecamp.org/news/javascript-es6-promises-for-beginners-resolve-reject-and-chaining-explained/.
6. **© 2005-2020 Mozilla and individual contributors.** Using Promises . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises.
7. **K, Gokul N.** Should I use Promises or Async-Await . *Hackernoon.* [Online] 17. August 2018. https://hackernoon.com/should-i-use-promises-or-async-await-126ab5c98789.
8. **Wangperawong, Pan.** Async-Await vs. Then to Avoid Callback Hell □□. *DEV.* [Online] 19. April 2019. https://dev.to/pan/when-to-use-async-await-vs-then-with-promises-1gb7.
9. **© 2005-2020 Mozilla and individual contributors.** async function. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.
10. —. await. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await.
11. **Copyright 1999-2020 by Refsnes Data.** JavaScript Errors - Throw and Try to Catch ❮ PreviousNext ❯ . *w3schools.com.* [Online] https://www.w3schools.com/js/js_errors.asp.
12. **Salman, Arfat.** Deeply Understanding JavaScript Async and Await with Examples. *Bits and Pieces.* [Online] 1. May 2019. https://blog.bitsrc.io/understanding-javascript-async-and-await-with-examples-a010b03926ea.
13. **Diallo, Hamidou.** Async Await vs then/catch . *Medium.* [Online] 25. November 2019. https://medium.com/@dio.hamidou/async-await-vs-then-catch-4f64d42e6392.
14. **Server . *TechTerms.* [Online] 16. April 2014. https://techterms.com/definition/server.**
15. Copyright 1999-2020 by Refsnes Data. What is HTTP? . *w3schools.com.* [Online] https://www.w3schools.com/whatis/whatis_http.asp.
16. Dixit, Shubhang. Beginners Guide to Client Server Communication. *Medium.* [Online] 30. July 2019. https://medium.com/@subhangdxt/beginners-guide-to-client-server-communication-8099cf0ac3af.
17. Hoffman, Chriss. What Is an API? *How-To-Geek.* [Online] 21. March 2018. https://www.howtogeek.com/343877/what-is-an-api/.
18. Robie, Jonathan a Research, Texcel . What is the Document Object Model? *w3.org.* [Online] https://www.w3.org/TR/REC-DOM-Level-1/introduction.html.
19. © 2005-2020 Mozilla and individual contributors. Fetch API . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
20. Fletes, Derek. How To Make A Chart Using Fetch & REST API's . *ZinChart.* [Online] 14. December 2017. https://blog.zingchart.com/how-to-make-a-chart-using-fetch-rest-apis/.

21. Avraham, Shif Ben. What is REST — A Simple Explanation for Beginners, Part 1: Introduction. *Medium.* [Online] 5. September 2017. https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f.
22. Redka, Vasyl. A Beginner's Tutorial for Understanding RESTful API. *MLSDev.* [Online] 2. August 2016. https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api.
23. © 2020 Codecademy. What is REST? *codeacademy.* [Online] https://www.codecademy.com/articles/what-is-rest.
24. Buckler, Craig. What Is a REST API? . *sitepoint.* [Online] 5. February 2020. https://www.sitepoint.com/developers-rest-api/.
25. Farcic, Viktor. REST API with JSON. *Technology Conversations.* [Online] 12. August 2014. https://technologyconversations.com/2014/08/12/rest-api-with-json/.
26. Copyright 1999-2020 by Refsnes Data. JavaScript Array reduce() Method . *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_reduce.asp.
27. © 2005-2020 Mozilla and individual contributors. Using Fetch . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.
28. —. Response.ok . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Response/ok.
29. —. Promise.reject(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject.
30. ThehungryBrain. REST API Architectural Constraints . *GeeksForGeeks.* [Online] https://www.geeksforgeeks.org/rest-api-architectural-constraints/.