WISE GUIDES
CAREER CHANGER

Rev. 1Aug2019

Exercise 52

WORKING WITH APIS

When a user types something like **http://facebook.com** into their browser, the following things happen:

An IP (Internet Protocol) address is a four-part series of numbers, separated by dots. For example, the IP address for facebook.com is 31.13.70.36

1. The browser asks a special server called a *domain name* server for the IP address associated with facebook.com.

2. The browser then sends an HTTP (HyperText Transfer Protocol) *request* to the associated IP address.

3. The Facebook server sends back an HTML *response*.

4. The browser translates the HTML into the browser *Document Object Model* (DOM) and renders the page.

You've been learning about *front-end* programming, working with HTML, CSS, and JavaScript. There is another realm of development, called *back-end* programming, that is responsible for serving up the HTML response (step 3 above).

As a front-end programmer, you don't need to know about how the server does its work. That was not always the case. In the early days of web development, a programmer needed to know how to work in both front-end and back-end environments. Today, they are two entirely separate fields.

Something, though, needs to connect the two. We don't try to load up all of Facebook or Amazon in the browser. We need a mechanism for communicating between front end and back end. That mechanism is the *Application Programming Interface* (API).

Building complete SPAs using the extremely popular technologies, React and Redux, is covered in my course at codingcareer.com.

What is an API? Think of it as a URL that can be called to do some specific things. What kind of things? Rather than try to explain what an API is and how to use it, let's get some actual experience. In this exercise, we'll use an API provided by the Online Movie Database.

data- attributes

Begin by looking at `index.html`. The only interesting thing to note is the use of a `data-title` attribute for the movie titles.

Usually when we want to get information from an HTML element, we would read either the `id` or the `class` attributes. As we've seen in previous exercises, that works fine for most things. Sometimes, though, we want information to be available that doesn't fit nicely into the id/class options. In this exercise, I want to attach the movie title to the `<div>` element. Could that be an `id`? Yes, but it's clumsy. It feels like we're using `id` for something other than its intended use.

HTML has a more elegant solution: we can create any custom attribute we wish, so long as it has a `data-` prefix. And we can use as many of these as we want. In addition to `data-title`, we could have `data-genre`, `data-yearReleased` — or anything else we wanted. Once we examine `index.js`, we'll see how these are used.

index.js

Let's start at the bottom of the page, where we set an event listener on each movie. The event listener is this:

```
let movieInfo = async event => {
  let data = await makeRequest(event.target.dataset.title)
  document.querySelector('#title').innerHTML = data.Title
  document.querySelector('#actors').innerHTML = data.Actors
  document.querySelector('#director').innerHTML = data.Director
  document.querySelector('#year').innerHTML = data.Year
  document.querySelector('#rating').innerHTML = data.imdbRating
  document.querySelector('#plot').innerHTML = data.Plot
  document.querySelector('img').src = data.Poster
}
```

In the second line of code, we see the snippet `event.target.dataset.title`. We just looked at the custom `data-` attributes. Here is how they become accessible to us. All custom `data-` attributes are grouped into the `dataset` variable. From there, individual `data-` components are available: `data-title` as `dataset.title`, `data-genre` as `dataset.genre`, and so forth.

asynchonicity

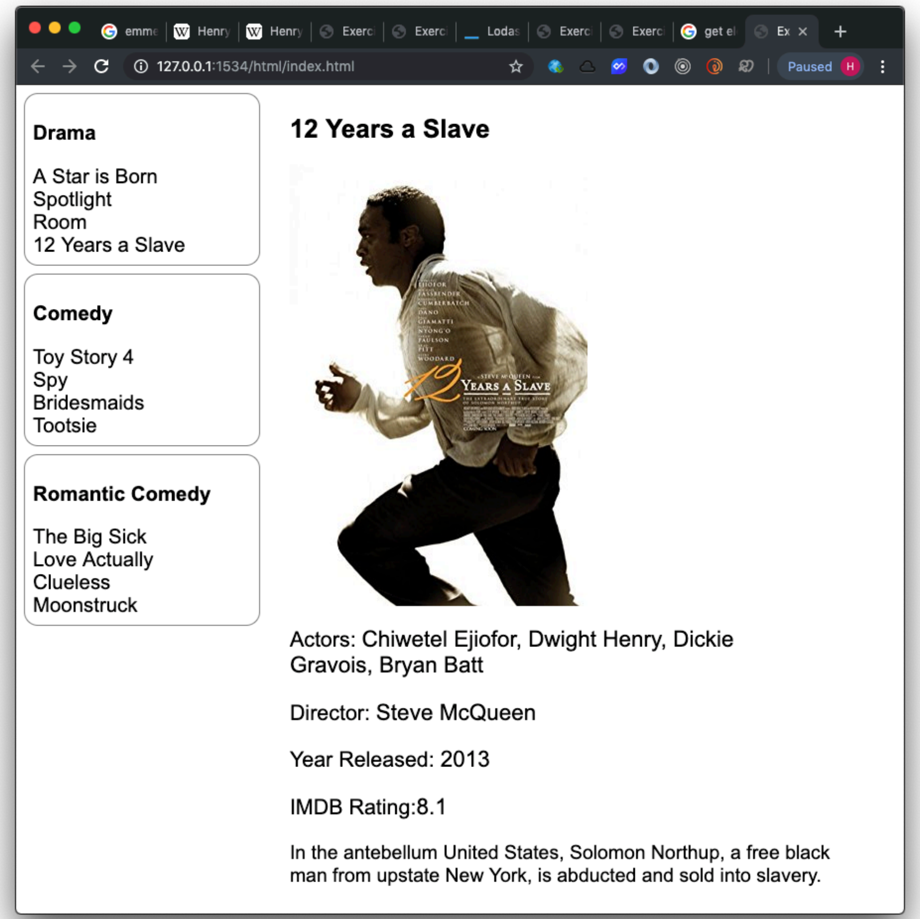
I neglected mentioning something about that second line. It has a keyword that we haven't seen before: `await`. And, looking at the function signature, there's another new keywords: `async`. What is going on?

Code normally executes from the top of a JavaScript page downwards. The second line of code doesn't execute until the first line is finished; the third line of code waits for the second line of code to complete, and so on.

This is called *synchronous* code. It works great most of the time. The times it might cause problems is if code at the top of the page takes a long time to complete. Then all the other code is prevented from executing while we wait. Such code is known as *blocking* code.

Why would code potentially take a while to complete? One very common reason is that we're not in control of the code that runs — and that is exactly what happens when we call a third-party API. Here's the API URL we'll use: **http://www.omdbapi.com/?apikey=f7d1d384&t=${movie}**. We'll get back a JSON object containing keys for `Title`, `Actors`, `Director`, `Year`, `imdbRating`, `Plot`, and `Poster`. We then output that data like this:

JSON stands for "JavaScript Object Notation". JSON is a format for data that works particularly well when sending data between browsers (aka "clients") and servers.



127.0.0.1:1534/html/index.html

Drama

A Star is Born
Spotlight
Room
12 Years a Slave

Comedy

Toy Story 4
Spy
Bridesmaids
Tootsie

Romantic Comedy

The Big Sick
Love Actually
Clueless
Moonstruck

12 Years a Slave

Actors: Chiwetel Ejiofor, Dwight Henry, Dickie Gravois, Bryan Batt

Director: Steve McQueen

Year Released: 2013

IMDB Rating:8.1

In the antebellum United States, Solomon Northup, a free black man from upstate New York, is abducted and sold into slavery.

Ideally, the time between making a call to the API and receiving the data back from the API is far less than one second. But there are times when something goes wrong: perhaps the API server is down, or it's very busy and therefore slow. Maybe it can't find the movie requested.

Whatever reason that caused getting a *response* back from our *request* to be delayed, blocking other code from executing during that wait is a pretty bad user experience. Some new features were added to JavaScript to alleviate the problem.

In this exercise, we'll be "consuming" a promise provided by the OMDB server. You can also create promises on your own, to be used by others, but that's beyond the scope of what we'll be using promises for (and also far more rarely used).

Instead of the API returning data immediately (which response might be delayed), the API immediately returns a *promise*. A promise ackowledges that a request has been made and promises to fulfill that request as soon as it can.

But our event listener function, `movieInfo`, doesn't have any promises in it, right? Right — but it makes a call to another function, `makeRequest`. Let's look at that:

"fetch" is a built-in JavaScript function that let us make an HTTP request in the background. This is similar to what happens when we type a URL into the browser search bar, only without the page refresh.

```
let makeRequest = async (movie) => {
  try {
    movie = movie.replace(' ', '+')
    let result = await fetch(`http://www.omdbapi.com/?apikey=f7d1d384&t=${movie}`)
    return result.json()
  } catch(e) {
    console.log(e)
  }
}
```

I've bolded the line of code that makes a call to an external API. You can read it like this:

1. make a call to the OMDB API for info on a particular movie

2. Once the info is returned, set it to the variable, `result`

One of the reasons that JSON has been so universally adopted is the ease of translation between JavaScript and JSON. In the bad old days, we used XML, which required extensive processing to translate.

We use `await` because we're waiting for the promise returned initially by the API to be fulfilled. Once we receive result, we call `json` on it. This function converts the JSON data structure into a JavaScript data structure.

Because the `makeRequest` function has is `await`ing a promise, we must alert anyone using the function that a response may be delayed. That's why functions that have `await` clauses in them must be marked `async`.

The last new thing we encounter is a try/catch block. This is used when we need to deal with the fact that we might encounter an error. Here, we're not trying to prevent one of *our* errors, but rather the possibility of an error being returned from the API server.

Suspect code is placed in the `try` block. If an error occurs, the `catch` block code will be run.