Lesson Monday

# Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Tools for Handling Async Code

Text

So far, we've learned how to use an `XMLHttpRequest` object to make our API call. To handle the asynchrony of the request, or, waiting for the request to be sent out and a response returned to us, we used event listeners and callback functions. Why? Well, we only want our code to call the functions that process the API response once the API call is complete, and not before.

We should remember that JavaScript is non-blocking, so it won't wait for a line of asynchronous code to finish processing before proceeding with the next line of code. This is why we must use callback functions to instruct our code how to proceed once the asynchronous action is done. In this way, callback functions allow us to control the order in which asynchronous and synchronous functions run.

An important piece of history here is that **callback functions used to be the only tool available to handle asynchronous code**. This includes callbacks that we pass to event handlers that are called later, whenever the corresponding event happens.

Nowadays, callbacks are just one way for JavaScript developers to deal with async code. In this lesson (and following lessons), we'll learn about newer tools at our disposal to make API calls and handle asynchrony, all of which makes our lives easier as developers. We'll start this exploration by learning why these newer tools came about to begin with.

## The Pyramid of Doom, or Callback Hell

Quite often in larger applications we need to make multiple API calls, not just one like in our OpenWeather API project. It also happens often that the next API call we make depends on the response we get from the last API call. For an example, let's consider an application that tells you what something is based off of an image, and translates that word into multiple languages. In fact, this was a team week project from Epicodus students!

To make this application function, these students needed to make an API call to an AI processor to guess what the image is, then with that returned data, make another API call to translate the word into multiple languages.

Well, say we needed to make another API call based on the returned translation data, and then another based on that returned API data. A series of asynchronous calls that are each managed by a callback can get confusing fast, and code that is confusing is code that is difficult to debug and maintain.

Let's see what this might look like. To understand how a series of callback functions leads to code that's hard to understand, we'll look at a simple example that doesn't make API calls, but uses callback functions in the exact manner that we would if we were making API calls.

The following code takes in user input, and puts it through a series of functions; each function transforms the user input in a specific (and very arbitrary) way. Notably, the next function always relies on the result of the previous function.

```
// This function adds a string to the end of the user input
function stepOne(userInput, callback) {
  const result = userInput + " and manage asynchrony via ca
llbacks";
  callback(result)
}

// This function adds another string to the end of the inpu
t string,
// which is the result from the stepOne() function
function stepTwo(data, callback) {
  const result = data + "!!!!";
  callback(result)
}

// This function uppercases the input string,
// which is the result from the stepTwo() function
function stepThree(data, callback) {
  const result = data.toUpperCase();
  callback(result)
}

// This function lowercases the third word of the input str
ing,
// which is the result from the stepThree() function
function stepFour(data, callback) {
  dataArray = data.split(" ");
  dataArray[2] = dataArray[2].toLowerCase();
  const result = dataArray.join(" ");
  callback(result);
}

function getAPIData(userInput) {
  stepOne(userInput, function(firstChange) {
    stepTwo(firstChange, function(secondChange) {
      stepThree(secondChange, function(thirdChange) {
        stepFour(thirdChange, function(fourthChange) {
          console.log(fourthChange);
        });
      });
    });
  });
}

getAPIData("let's get API data");
```

Note that you can copy and paste the above code and add it to your DevTools. When you do, you will see `LET'S GET api DATA AND MANAGE ASYNCHRONY VIA CALLBACKS!!!!` printed to the console.

Again, while there's no API calls and no asynchrony (and not even error handling to make our code even more complex), this example shows us the trouble of using a series of nested callbacks to manage a series of async actions: it's hard to understand!

```
function getAPIData(userInput) {
  stepOne(userInput, function(firstChange) {
    stepTwo(firstChange, function(secondChange) {
      stepThree(secondChange, function(thirdChange) {
        stepFour(thirdChange, function(fourthChange) {
          console.log(fourthChange);
        });
      });
    });
  });
}
```

More than being hard to understand, there are quite a few problems with the code above. One of the biggest problems is that these four hypothetical functions are now reliant on each other. If we make a change to the `stepTwo` function, the entire house of cards might come tumbling down.

In this example, all the callbacks are in order and in the same file, making them somewhat easier to read, but imagine a large codebase where these callback functions might be located in separate files or used in multiple locations. If you are debugging or making an update, you'll have to work through a whole series of files.

Also, it's possible that some of the functions you use in this series of callbacks are used elsewhere in our code — making it impossible to write the code in the order that it would actually be called. The result is that developers end up with "spaghetti code." This means that they have to follow a strand of spaghetti through many other tangled strands of spaghetti to make updates or fixes to their code.

Finally, the example above is just four functions — now imagine a dozen functions, both async and sync. Some are used in just part of the codebase, and others used in many places. Again, this can get confusing fast.

Developers call this **callback hell**. Another popular name is **the pyramid of doom**, because the shape of the code looks like a pyramid on its side. No matter what you call it, developers try to avoid this style of programming, and have since come up with new tools to make the process of making a series of API calls that much easier.

However, large code bases tend to remain complex by the nature of being large. All we can do as developers is make sure that our code is descriptive, modular, and decoupled, our logic is separated, and we use tools for handling asynchrony and API calls that make our code easier to read and reason about. This will lead to codebases that are easier to debug and maintain, and that can scale as requirements change.

## JavaScript Tools for Managing Asynchrony

In this course section, we are going to learn about two other ways that JavaScript provides to handle async code:

- Promises

- Async functions with the `async` and `await` keywords

There are other tools that JavaScript uses to deal with asynchrony ranging from generators to observables but we can only cover so much in one section. For this section's independent project, you can use whichever method of handling asynchrony that you prefer.

To wrap up this lesson, we'll work through a brief overview of the three techniques we learn about in this section, including advantages and disadvantages of each.

## Callbacks

A callback is a function that's passed into another function as an argument, to be called later. They are a huge part of JavaScript and an essential tool for all developers.

**Async Advantages:** A callback is a simple way to handle basic asynchrony, especially if the code isn't too complex. Also, callbacks can handle just about anything so they are a great all-purpose tool.

**Async Disadvantages:** Callbacks quickly become difficult to work with when many are chained together. Code that uses many callbacks to handle asynchronous actions is difficult to read and reason about and bugs can be very difficult to deal with, leading developers into what is known as "callback hell", or the "pyramid of doom".

## Promises

Promises are a relatively new feature in JavaScript — they were originally added to ES6 (released in 2015). Promises wrap async code and are either resolved or rejected when the asynchronous action is complete. We can use a method called `Promise.prototype.then()` with a promise to handle a resolved or rejected promise. We'll learn more about them in the next lesson.

**Async Advantages:** Promises are easy to work with *once you understand them*. So be patient with yourself if they are confusing at first. We can write code in the order it runs and easily manage callbacks with promises. Promises are great with API calls!

**Async Disadvantages:** A promise can only be resolved or rejected once. If you want a piece of code that handles many async operations, a promise isn't the best way to go.

## Async Functions

Async functions are even newer than promises — they were added to JavaScript in ES7 (released in 2017). We can wrap code in an async function and force it to run in order as if it were synchronous code. This effectively makes JavaScript blocking, instead of non-blocking.

**Async advantages:** Because code in async functions runs in order, it's easier to read, write, and reason about. Async functions are also a concise and elegant way to write async code.

**Async disadvantages:** All of the code inside an async function will be blocking — and will run synchronously — so don't put too much code inside one because then you might start overriding JavaScript's non-blocking advantages.

## So Which Tool Do I Use?

Students often get confused throughout this section. At this point, it's tempting to want to know the best tool for the job and stick with that. But there are advantages and disadvantages to each tool depending on what you are doing. We can use the analogy for carpentry tools here. Want to attach a nail to a board? Use a hammer. How about many, many nails? Use a nail gun. How about a screw? Use a screwdriver. And many screws? Use a drill. We can't tell a carpenter just to use a hammer and forget about it. For the same reason, we can't just tell a developer to use promises and leave it at that.

However, promises are often the best way to handle API calls. That's because generally we'll make an API call just once in a function — and promises are rejected or resolved once. For that reason, most of the code you'll write during this section will probably use promises with callbacks — and this combo is a great way to go for this section's independent project as well. Just make sure you take some time to practice with each tool during the daily classwork.

Previous (/intermediate-javascript/asynchrony-and-apis/openweather-api-giphy-api)
Next (/intermediate-javascript/asynchrony-and-apis/introduction-to-promises)
Lesson 16 of 33
Last updated more than 3 months ago.
disable dark mode

Epicodus (http://www.epicodus.com)

© 2023 Epicodus (http://www.epicodus.com/), Inc.