Async/Await

Learn the modern way to write asynchronous JavaScript. Async/await is quickly becoming the industry standard and it's an important tool to learn.

`async`/`await`

We’ve covered callbacks and promises. Now we get to the latest async feature that will change how we write asynchronous functions in JS.

This is a feature that truly makes asynchronous code much easier to write. Let’s dive right in.

Introduction

An asynchronous function is created using by writing the `async` keyword before a function. Here’s what it looks like for a standard function.

```
async function fn() {...}
const fn = async function() {...}
```

Here’s what it looks like for an arrow function.

```
const fn = async () => {...}
```

Inside an asynchronous function, we can use the `await` keyword to make the function pause and wait for some operation to complete.

Here, I’ve written a function `wait` that takes in a number and returns a string after that number of seconds. The function `fn` calls `wait`. When `wait` completes and resolves, `fn` continues executing.

- Note that these asynchronous functions won’t work very well here in Educative’s code blocks. I recommend you use another JS environment such as https://repl.it/site/languages/javascript.

Example

```
function wait(secondsToWait) {
    return new Promise(resolve => {
      setTimeout(
        () => resolve(`Resolved after ${secondsToWait} seconds`),
        secondsToWait * 1000
      );
    });
}

async function fn() {
    console.log('Beginning fn');

    const result = await wait(2);
    console.log(result);

    console.log('Ending fn');
}

fn();
```

What's Going On

Let's walk through the series of steps here.

We start by invoking our async function `fn` on line 19. Immediately, `"Beginning fn"` is logged to the console.

The next line to run is line 13. This is the crux of `async`/`await`. On the right side of the statement, we have `await wait(2)`. This means that the function `wait` will be immediately invoked and until it's done, `fn` just sits there.

Once the appropriate amount of time has passed, in this case 2 seconds, the string `"Resolved after 2 seconds"` is resolved from the promise. This string is received by the variable `result`, which we proceed to log to the console.

The string `"Ending fn"` is then logged to the console.

Easier to Read

The point of this exercise is to show what asynchronous code looks like using `async`/`await`. The function `fn` looks like a normal, synchronous function. There are no callbacks, no nested function calls, no chained function calls, and no pyramid of doom.

This code is easier to reason about than anything we've seen so far, at least once we're used to it. All we have to do to understand the function is read it top to bottom. It's clear exactly where the function pauses and waits for something to happen.

Types of Asynchronous Calls

Series

We can make more than one asynchronous call in an async function. In the following code, the first logged statement will print one second after execution starts. The second will print *two seconds after the first one prints*. The total time the function takes will be 3 seconds.

```
function wait(secondsToWait) {
    return new Promise(resolve => {
      setTimeout(
        () => resolve(`Resolved after ${secondsToWait} seconds`),
        secondsToWait * 1000
      );
    });
}

async function fn() {
    const result1 = await wait(1);
    console.log(result1); // -> 1s after execution begins: "Resolved after 1 seconds"

    const result2 = await wait(2);
    console.log(result2); // -> 3s after execution begins: "Resolved after 2 seconds"
}

fn();
```

The function takes a total of 3 seconds. On line 11, `fn` will pause for one second and then print the result. It will wait for two more seconds on line 14 and then print the result.

Because the two async calls inside `fn` proceed in a linear fashion, we say that the calls are made **in series**.

Parallel

Instead of making calls sequentially, in series, we can also make calls *in parallel*. This means that two or more asynchronous function calls will be happening simultaneously.

Take a look at the code below.

```
function wait(secondsToWait) {
    return new Promise(resolve => {
      setTimeout(
        () => resolve(`Resolved after ${secondsToWait} seconds`),
        secondsToWait * 1000
      );
    });
}

async function fn() {
    const result1 = wait(1);
    const result2 = wait(2);

    console.log(result1); // -> Promise { <pending> }
    const r1Resolved = await result1;
    console.log(r1Resolved);
    // -> 1s after execution begins: "Resolved after 1 seconds"


    console.log(result2); // -> Promise { <pending> }
    const r2Resolved = await result2;
    console.log(r2Resolved);
    // -> 2s after execution begins: "Resolved after 2 seconds"
}

fn();
```

How it Works

Note the differences here. Mainly, note that lines 11 and 12 do not use `await`. Since we're not using `await`, the items that get inserted into `result1` and `result2` are Promises.

Since we didn't use `await`, the function will continue to run. The function call on line 11 will start and while our function waits for the Promise to resolve, it'll continue and start the function call on line 12.

When we log the first promise on line 14, it hasn't been fulfilled yet. It's still pending and that's exactly what we see printed out pending and that's exactly what we see printed out.

After this, on line 15, we then wait for the promise to resolve. This is precisely what `await` does. It instructs the JavaScript engine to block execution until the Promise completes. We assign this Promise's resolution to a new variable.

Once the first Promise resolves and prints, the function continues to line 21 and waits for the second promise.

True Parallelism

While the function above works as we want, there are use cases that require more tools than we've covered so far. For example, let's imagine that we don't know when our `wait` function will resolve. Here's a new version of that function.

```
function randomWait() {
    const seconds = Math.random() * 5;

    return new Promise(resolve => {
      setTimeout(
        () => resolve(`Resolved after ${seconds.toFixed(2)} seconds`),
        seconds * 1000
      );
    });
}

async function fn() {
    const result1 = randomWait();
    const result2 = randomWait();

    console.log(result1); // -> Promise { <pending> }
    const r1Resolved = await result1;
    console.log('1st call:', r1Resolved);

    console.log(result2); // -> Promise { <pending> }
    const r2Resolved = await result2;
    console.log('2nd call:', r2Resolved);
}

fn();
```

We've changed the name of the function from `wait` to `randomWait`. In addition, it no longer takes in a parameter; instead, it generates a random number between 0 and 5 and waits that number of seconds.

We now have a bit of a problem. There is a 50% chance that `result1` will resolve *after* `result2`. This means that half the time, `result2` will finish while the function is executing line 17. Instead of seeing the printout as soon as it's available, we now have to wait.

In other words, line 18 will *always* print before line 24, even if line 24's work is completed earlier. Half the time, line 24 will be forced to wait longer than it has to.

This is undesirable for a number of reasons. We're often waiting for our result even though it's ready and its Promise has resolved. Almost any time we have unnecessary wait times in an application, we should refactor it and improve its speed.

Imagine we're writing an application that will load 5 articles for a user to read when they log in to our site. We want to send these articles to the browser as fast as possible to make sure the reader sees them and doesn't leave the site.

If we have code that looks like the function above, we may have to wait on the article that takes the longest to load before sending *any* of them to the browser. This means that quite often, our application will be unnecessarily slow, and we'll see users leave because the content loads too slowly.

Here's how we can fix the above function to be truly parallel and print out results as soon as the Promises resolve.

```
function randomWait() {
    const seconds = Math.random() * 5;

    return new Promise(resolve => {
      setTimeout(
        () => resolve(`Resolved after ${seconds.toFixed(2)} seconds`),
        seconds * 1000
      );
    });
}

function fn() {
    const result1 = randomWait()
    .then(result => console.log('1st call:', result));
    const result2 = randomWait()
    .then(result => console.log('2nd call:', result));
}
```

```
fn();
```

In this example, the Promise that resolves first will print first. We're as likely to see

```
2nd call: Resolved after 2.21 seconds
1st call: Resolved after 4.10 seconds
```

as we are to see

```
1st call: Resolved after 2.21 seconds
2nd call: Resolved after 4.10 seconds
```

This function no longer uses `async` / `await` and I only include this example here to show you the limits of `async` / `await`.

Using `await` is fantastic when the tasks we're trying to be accomplished are supposed to be sequential and the asynchronous function calls need to happen in series. If we need true parallelism as in the function in this example, we need to go back to Promises.

Error Handling

There are no special mechanisms for error handling with `async` / `await`. We can use a standard `try` / `catch` block as we did with ES5 code. We'll have an example of this in the next lesson.

To find out where `async` / `await` can be the most useful, check out the next lesson. It's short and ties this mechanism back to the example we've used in earlier lessons.

That's it for `async` / `await`.