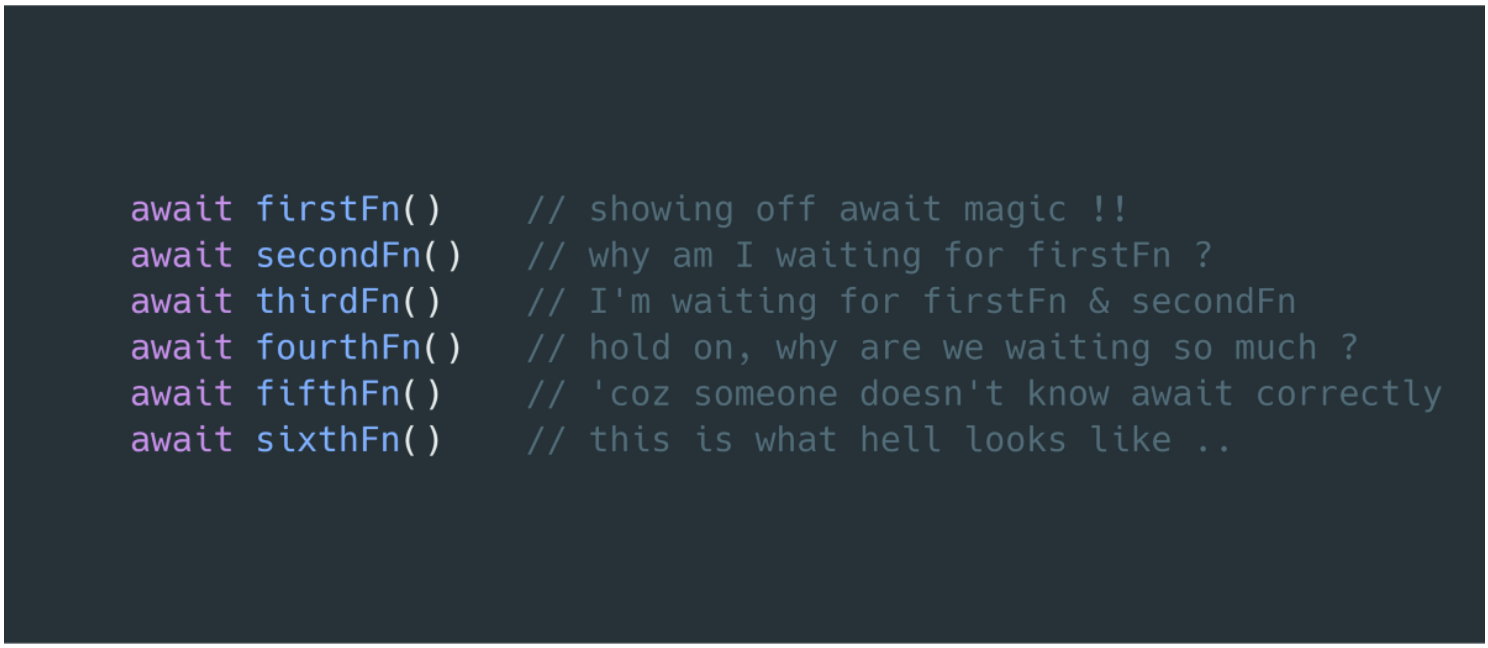
How to escape async/await hell

Aditya Agarwal Follow
Apr 7, 2018 · 5 min read



async/await freed us from callback hell, but people have started abusing it—leading to the birth of async/await hell.

In this article, I will try to explain what async/await hell is, and I'll also share some tips to escape it.

What is async/await hell

While working with Asynchronous JavaScript, people often write multiple statements one after the other and slap an **await** before a function call. This causes performance issues, as many times one statement doesn't depend on the previous one—but you still have to wait for the previous one to complete.

An example of async/await hell

Consider if you wrote a script to order a pizza and a drink. The script might look like this:

```
(async () => {
  const pizzaData = await getPizzaData()    // async call
  const drinkData = await getDrinkData()    // async call
  const chosenPizza = choosePizza()    // sync call
  const chosenDrink = chooseDrink()    // sync call
  await addPizzaToCart(chosenPizza)    // async call
  await addDrinkToCart(chosenDrink)    // async call
```

On the surface it looks correct, and it does work. But this is not a good implementation, because it leaves concurrency out of the picture. Let's understand what its doing so that we can nail down the issue.

Explanation

We have wrapped our code in an async IIFE. The following occurs in this exact order:

1. Get the list of pizzas.
2. Get the list of drinks.
3. Choose one pizza from the list.
4. Choose one drink from the list.
5. Add the chosen pizza to the cart.
6. Add the chosen drink to the cart.
7. Order the items in the cart.

So what's wrong ?

As I stressed earlier, all these statements execute one by one. There is no concurrency here. Think carefully: why are we waiting to get the list of pizzas before trying to get the list of drinks? We should just try to get both the lists together. However when we need to choose a pizza, we do need to have the list of pizzas beforehand. The same goes for the drinks.

So we can conclude that the pizza related work and drink related work can happen in parallel, but the individual steps involved in pizza related work need to happen sequentially (one by one).

Another example of bad implementation

This JavaScript snippet will get the items in the cart and place a request to order them.

```
async function orderItems() {
  const items = await getCartItems()    // async call
  const noOfItems = items.length
  for(var i = 0; i < noOfItems; i++) {
    await sendRequest(items[i])    // async call
  }
```

In this case, the for loop has to wait for the `sendRequest()` function to complete before continuing the next iteration. However, we don't actually need to wait. We want to send all the requests as quickly as possible and then we can wait for all of them to complete.

I hope that now you are getting closer to understanding what is async/await hell and how severely it affects the performance of your program. Now I want to ask you a question.

What if we forget the await keyword ?

If you forget to use **await** while calling an async function, the function starts executing. This means that await is not required for executing the function. The async function will return a promise, which you can use later.

```
(async () => {
  const value = doSomeAsyncTask()
  console.log(value) // an unresolved promise
})()
```

Another consequence is that the compiler won't know that you want to wait for the function to execute completely. Thus the compiler will exit the program without finishing the async task. So we do need the **await** keyword.

One interesting property of promises is that you can get a promise in one line and wait for it to resolve in another. This is the key to escaping async/await hell.

```
(async () => {
  const promise = doSomeAsyncTask()
  const value = await promise
  console.log(value) // the actual value
```

As you can see, `doSomeAsyncTask()` is returning a promise. At this point `doSomeAsyncTask()` has started its execution. To get the resolved value of the promise, we use the await keyword and that will tell JavaScript to not execute the next line immediately, but instead wait for the promise to resolve and then execute the next line.

How to get out of async/await hell ?

You should follow these steps to escape async/await hell.

Find statements which depend on the execution of other statements

In our first example, we were selecting a pizza and a drink. We concluded that, before choosing a pizza, we need to have the list of pizzas. And before adding the pizza to the cart, we'd need to choose a pizza. So we can say that these three steps depend on each other. We cannot do one thing until we have finished the previous thing.

But if we look at it more broadly, we find that selecting a pizza doesn't depend on selecting a drink, so we can select them in parallel. That is one thing that machines can do better than we can.

Thus we have discovered some statements which depend on the execution of other statements and some which do not.

Group-dependent statements in async functions

As we saw, selecting pizza involves dependent statements like getting the list of pizzas, choosing one, and then adding the chosen pizza to the cart. We should group these statements in an async function. This way we get two async functions, `selectPizza()` and `selectDrink()` .

Execute these async functions concurrently

We then take advantage of the event loop to run these async non blocking functions concurrently. Two common patterns of doing this is **returning promises early** and the **Promise.all method**.

Let's fix the examples

Following the three steps, let's apply them on our examples.

```
async function selectPizza() {
  const pizzaData = await getPizzaData()    // async call
  const chosenPizza = choosePizza()    // sync call
  await addPizzaToCart(chosenPizza)    // async call
}

async function selectDrink() {
  const drinkData = await getDrinkData()    // async call
  const chosenDrink = chooseDrink()    // sync call
  await addDrinkToCart(chosenDrink)    // async call
}

(async () => {
  const pizzaPromise = selectPizza()
  const drinkPromise = selectDrink()
```

Now we have grouped the statements into two functions. Inside the function, each statement depends on the execution of the previous one. Then we concurrently execute both the functions `selectPizza()` and `selectDrink()` .

In the second example, we need to deal with an unknown number of promises. Dealing with this situation is super easy: we just create an array and push the promises in it. Then using `Promise.all()` we concurrently wait for all the promises to resolve.

```
async function orderItems() {
  const items = await getCartItems()    // async call
  const noOfItems = items.length
  const promises = []
  for(var i = 0; i < noOfItems; i++) {
    const orderPromise = sendRequest(items[i])    // async
    promises.push(orderPromise)    // sync call
  }
  await Promise.all(promises)    // async call
}

// Although I prefer it this way
```

I hope this article helped you see beyond the basics of async/await, and also helped you improve the performance of your application.

If you liked the article, please clap your heart out. Tip—You can clap 50 times!

Please also share on Fb and Twitter. If you'd like to get updates, follow me on Twitter and Medium or subscribe to my newsletter! If anything is not clear or you want to point out something, please comment down below.