2 days ago · 5 min read

Is async/await a step back to JavaScript?

Thoughts on the famous pair of magic keywords, promise chains and a bit of functional programming

```
async function asyncHandleRequest(req, res) {
  try {
    const { user } = req
    await isUserValidAsync(user)
    const [data, rate] = await Promise.all([getUserDataAsync(user), getRateAsync('service')])
    const savedData = await updateUserDataAsync(user, updateData(data, rate))
    res.send(savedData)
  } catch (err) {
    res.error('An error ocurred!')
  }
}
```

Using **async/await** in a function that handles a request.

From callbacks to promises

By the end of 2015, I started to hear about a new set of keywords coming into the JavaScript world that would save us from the promise chain hell, which, in turn, saved us from callback hell! I remember those were exciting times, while trying to understand how **async/await** worked under the hood and how that magic couple was going to simplify our developer lives once and for all.

Lets see some examples to understand how we get to **async/await** in the first place. Lets imagine we are working on our API and have to respond a request by a series of asynchronous operations:

- check the user is valid
- gather data from the database
- gather data from an external service
- manipulate and write data back to the database

Let's also suppose we do not have any knowledge of promises yet, because we traveled back in time a couple of years, so we use a

callbacks-based code to deal with the request. The solution would be something like the following:

```
function handleRequestCallbacks(req, res) {
  var user = req.user
  isUserValid(user, function (err) {
    if (err) {
      res.error('An error ocurred!')
      return
    }
    getUserData(user, function (err, data) {
      if (err) {
        res.error('An error ocurred!')
        return
      }
      getRate('service', function (err, rate) {
        if (err) {
          res.error('An error ocurred!')
          return
        }
        const newData = updateData(data, rate)
        updateUserData(user, newData, function (err, savedD
```

Typical callbacks-based solution

And this is the so-called **callback hell**! By now you should be familiar with it. Everybody learned to hate it as it is difficult to read, to debug, to change, it keeps getting deeper and deeper in the indentation as you nest more asynchronous operations, the error handling keeps repeating itself, and so on.

We could have used the famous **async** library to clean up the code a bit and it would be much shorter, as the error handling is at least in a single place but there would be too much overhead yet:

```
function handleRequestAsync(req, res) {
  var user = req.user
  async.waterfall([
    async.apply(isUserValid, user),
    async.apply(async.parallel, {
      data: async.apply(getUserData, user),
      rate: async.apply(getRate, 'service')
    }),
    function (results, callback) {
      const newData = updateData(results.data, results.rate
      updateUserData(user, newData, callback)
    }
  ], function (err, data) {
    if (err) {
```

Still callbacks but with the help of **async** to make our lives much better.

Feeling better after finishing the refactor, but time will teach us that better was not enough.

Later, one day, we learned how to use **promises** and thought the world was not evil anymore with us and we felt like refactoring our code one more time and more and more libraries were also moving into the **promises** world.

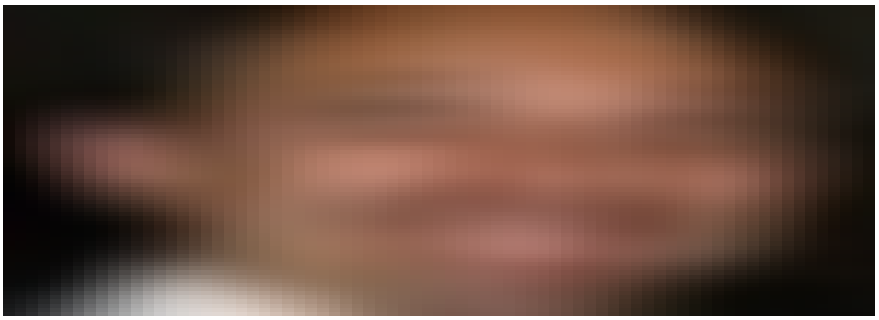
```
function handleRequestPromises(req, res) {
  var user = req.user
  isUserValidAsync(user).then(function () {
    return Promise.all([
      getUserDataAsync(user),
      getRateAsync('service')
    ])
  }).then(function (results) {
    const newData = updateData(results[0], results[1])
    return updateUserDataAsync(user, newData)
  }).then(function (data) {
```

Promises rule!

That was much better, much shorter and much cleaner! Still it had some issues, as there were still too much overhead in the form of

multiple `then()` calls, `function () {...}` blocks and the need of adding multiple `return` statements all over the place.

We finally start to hear about ES6, all the new stuff that is coming into JavaScript including the **arrow functions** (and a bit of destructuring just to have some more fun) and decide to give our beautiful code one more chance.



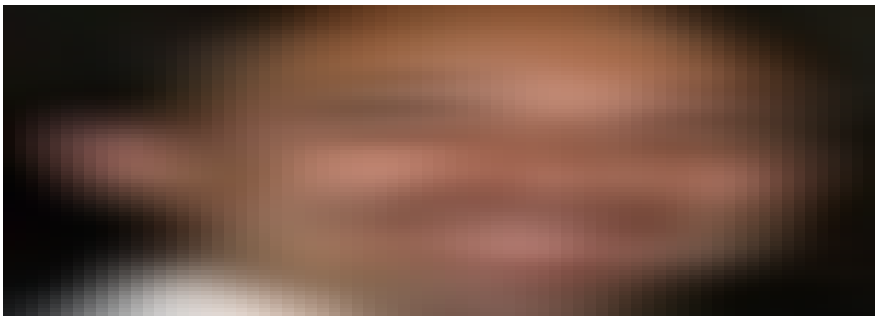
We thought promises were heaven and ES6 came into existence!

And this is it! That request handler function was clean, easy to read and reason about, easy to change in the case we needed to add, remove or replace something in the pipeline! We were chaining functions one after the other to mutate the data we gathered through different asynchronous operations, did not have to define intermediate variables to hold that state as was now handled in the chain and the error handling was clear and in a single place. Now we were sure we were definitely in the heaven of JavaScript developers! But was that true? Was that filing going to last?

And async/await arrives

A few months later, **async/await** comes into the scene. It was going to get into ES7, then postponed, but as Babel started to be adopted everywhere, we jumped into the train. We learned we can mark a function as **async** and that keyword will allow us to use **await** inside the function to "halt" the execution flow until the awaited promise settles to make our code look like synchronous again. We also learned the **async** function will always return a promise and we can use **try/catch** blocks again to handle our errors.

Not too convinced of the benefits, we give our code a new chance and go for a final refactor.



Synchronous-looking asynchronous code using **async/await**.

And now the code looked like plain old imperative synchronous code back again. Soon, we started to feel confident writing code like that as we implemented it in one place after the other and life continues as usual.

But something deep in your head, tells you something is not 100% OK…

The functional programming paradigm

Although **functional programming** has been around for more than 30/40 years now, it looks like only recently it is gaining momentum and more and more books, talks, videos, and all kind of software-development-related media is taking care of it. It looks like only recently, we all are starting to understand the benefits of the functional approach to think and develop a program, faster, easier and with higher quality.

And, of course, as we are curious as hell, we all start learning some of its principles. We acquire new words for our day-to-day vocabulary as **functors**, **monads**, **monoids**, **aplicatives**, and our dev-friends start thinking we are awesome again because we say those strange words quite often!

And as we continue diving the functional programming paradigm, we start realizing there is real value in there. Those functional programming advocates were not just crazy people. They were possibly right!

We understand the benefits of immutability, to not store or mutate state, to create complex logic by combining/composing simple functions, to avoid managing loops and let the magic be done by the

language interpreter itself so we can focus on what really matters, to even replace conditional branching and error handling by just combining more functions. And this is when functional programming gets you crazy, you have that “aha moment” and nothing is never as it was before!!!

But… wait a minute!!!

As we continue happily writing code with **async/await** we realize we have seen all those functional patterns in the past. Did we? Sure! We remember how we used promises and how we chained functional transformations one after the other, without the need to manage state or to branch our code or manage errors in an imperative way. We were already using the Promise **monad** in the past with all the associated benefits but we didn’t know that word by that time!

And we suddenly realize why the code based on **async/await** was looking weird! We were back writing plain imperative code like in the 80’s, handling errors with **try/catch** blocks like in the 90’s, managing internal state and variables, doing asynchronous operations and writing code that looks like synchronous that suddenly stops and then automagically continues executing as the asynchronous operations complete (cognitive dissonance, perhaps?)…

Final thoughts

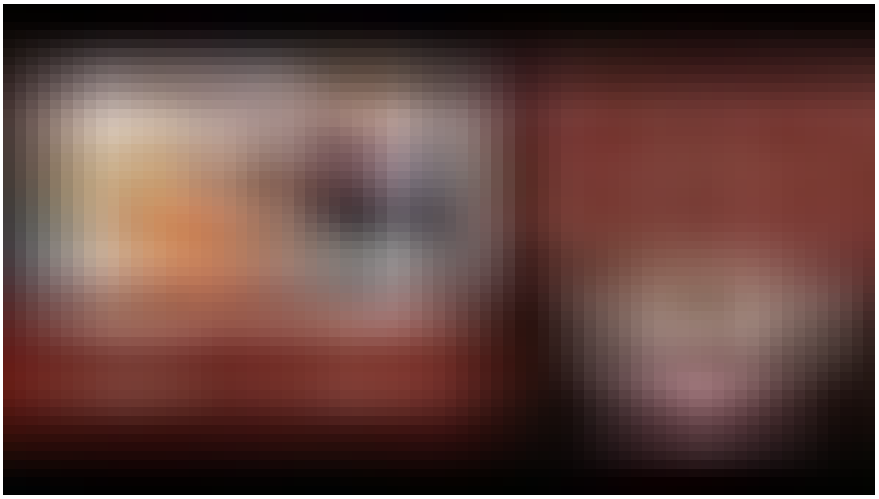
Don’t get me wrong, **async/await** is not the source of all evil in the world. I actually learned to like it after a few months of using it. So, if you feel confortable writing imperative code, learning how to use **async/await** to manage your asynchronous operations might be a good move.

But if you like **promises** and you like to learn and apply more and more functional programming principles to your code, you might want to just skip **async/await** code entirely, stop thinking imperative and move to this new-old paradigm.

Happy coding!

. . .

This is the video by Brian Lonsdorf that made me challenge all what I knew about handling asynchronous operations in JavaScript. Take a look at it!



Composition and functional programming by Brian Lonsdorf. https://www.youtube.com/watch?v=SfWR3dKnFIo

And, for those willing to read more qualified opinions about async/await being not-so-a-good thing, read Gorgi Kosev's great article on the subject: