Mostafa Gaafar Follow
Software Engineer by passion. Once a Petroleum Engineer. https://twitter.com/imGaafar
Mar 25 · 6 min read

6 Reasons Why JavaScript's Async/Await Blows Promises Away (Tutorial)

In case you missed it, Node now supports async/await out of the box since version 7.6. If you haven't tried it yet, here are a bunch of reasons with examples why you should adopt it immediately and never look back.

[UPDATE]: Node 8 LTS is out now with full Async/Await support.

[EDIT]: It seems that the embedded code on gist does not work on medium native app, but it works on mobile browsers. If you are reading this on the app, tap on the share icon and choose "open in browser" in order to see code snippets.

Async/await 101

For those who have never heard of this topic before, here's a quick intro

- Async/await is a new way to write asynchronous code. Previous options for asynchronous code are callbacks and promises.
- Async/await is actually built on top of promises. It cannot be used with plain callbacks or node callbacks.
- Async/await is, like promises, non blocking.
- Async/await makes asynchronous code look and behave a little more like synchronous code. This is where all its power lies.

Syntax

Assuming a function `getJSON` that returns a promise, and that promise resolves with some JSON object. We just want to call it and log that JSON, then return `"done"`.

This is how you would implement it using promises

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return "done"
    })
```

And this is how it looks with async/await

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

```

There are a few differences here

1. Our function has the keyword `async` before it. The `await` keyword can only be used inside functions defined with `async` . Any `async` function returns a promise implicitly, and the resolve value of the promise will be whatever you `return` from the function (which is the string `"done"` in our case).
2. The above point implies that we can't use await in the top level of our code since that is not inside an `async` function.

```
// this will not work in top level
// await makeRequest()

// this will work
makeRequest().then((result) => {
```

3. `await getJSON()` means that the `console.log` call will wait until `getJSON()` promise resolves and print it value.

Why Is It better?

1. Concise and clean

Look at how much code we didn't write! Even in the contrived example above, it's clear we saved a decent amount of code. We didn't have to write `.then` , create an anonymous function to handle the response, or give a name `data` to a variable that we don't need to use. We also avoided nesting our code. These small advantages add up quickly, which will become more obvious in the following code examples.

. . .

2. Error handling

Async/await makes it finally possible to handle both synchronous and asynchronous errors with the same construct, good old `try/catch`. In the example below with promises, the `try/catch` will not handle if `JSON.parse` fails because it's happening inside a promise. We need to call `.catch` on the promise and duplicate our error handling code, which will (hopefully) be more sophisticated than `console.log` in your production ready code.

```
const makeRequest = () => {
  try {
    getJSON()
      .then(result => {
        // this parse may fail
        const data = JSON.parse(result)
        console.log(data)
      })
      // uncomment this block to handle asynchronous errors
      // .catch((err) => {
      //   console.log(err)
```

Now look at the same code with async/await. The `catch` block now will handle parsing errors.

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
```

. . .

3. Conditionals

Imagine something like the code below which fetches some data and decides whether it should return that or get more details based on some value in the data.

```javascript
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
            console.log(moreData)
            return moreData
          })
      } else {
```

Just looking at this gives you a headache. It's easy to get lost in all that nesting (6 levels), braces, and return statements that are only needed to propagate the final result up to the main promise.

This example becomes way more readable when rewritten with async/await.

```javascript
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
```

. . .

4. Intermediate values

You have probably found yourself in a situation where you call a `promise1` and then use what it returns to call `promise2`, then use the results of both promises to call a `promise3`. Your code most likely looked like this

```javascript
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // do something
      return promise2(value1)
        .then(value2 => {
          // do something
          return promise3(value1, value2)
```

If `promise3` didn't require `value1` it would be easy to flatten the promise nesting a bit. If you are the kind of person who couldn't live with this, you could wrap both values 1 & 2 in a `Promise.all` and avoid deeper nesting, like this

```javascript
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // do something
      return Promise.all([value1, promise2(value1)])
    })
    .then(([value1, value2]) => {
      // do something
```

This approach sacrifices semantics for the sake of readability. There is no reason for `value1` & `value2` to belong in an array together, except to avoid nesting promises.

This same logic becomes ridiculously simple and intuitive with async/await. It makes you wonder about all the things you could have done in the time that you spent struggling to make promises look less hideous.

```javascript
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
```

. . .

5. Error stacks

Imagine a piece of code that calls multiple promises in a chain, and somewhere down the chain an error is thrown.

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => {
      throw new Error("oops");
    })
}

makeRequest()
```

The error stack returned from a promise chain gives no clue of where the error happened. Even worse, it's misleading; the only function name it contains is `callAPromise` which is totally innocent of this error (the file and line number are still useful though).

However, the error stack from async/await points to the function that contains the error

```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  throw new Error("oops");
}

makeRequest()
```

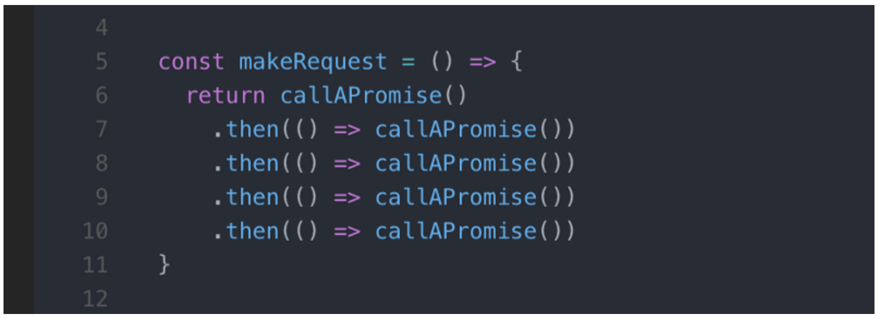
This is not a huge plus when you're developing on your local environment and have the file open in an editor, but it's quite useful when you're trying to make sense of error logs coming from your production server. In such cases, knowing the error happened in `makeRequest` is better than knowing that the error came from a `then` after a `then` after a `then` …

. . .

6. Debugging

Last but not least, a killer advantage when using async/await is that it's much easier to debug. Debugging promises has always been such a pain for 2 reasons

1. You can't set breakpoints in arrow functions that return expressions (no body).

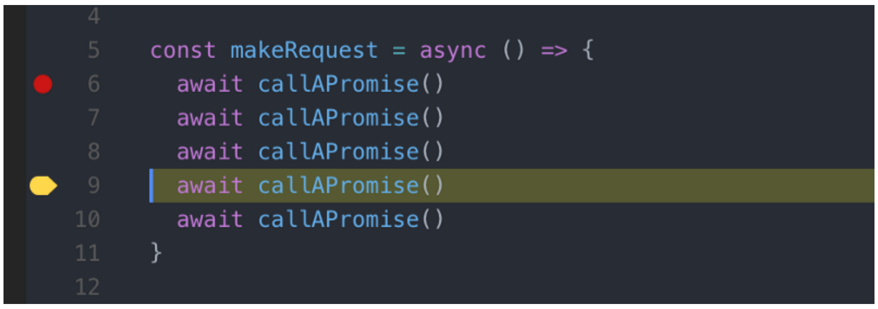


```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
}
```

Try setting a breakpoint anywhere here

2. If you set a breakpoint inside a `.then` block and use debug shortcuts like step-over, the debugger will not move to the the following `.then` because it only "steps" through synchronous code.

With async/await you don't need arrow functions as much, and you can step through await calls exactly as if they were normal synchronous calls.



```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
}
```

. . .

In Conclusion

Async/await is one of the most revolutionary features that have been added to JavaScript in the past few years. It makes you realize what a syntactical mess promises are, and provides an intuitive replacement.

Concerns

Some valid skepticism you might have about using this feature

- It makes asynchronous code less obvious: Our eyes learned to spot asynchronous code whenever we see a callback or a `.then` , it will take a few weeks for your eyes to adjust to the new signs, but C# had this feature for years and people who are familiar with it know it's worth this minor, temporary inconvenience.
- Node 7 is not an LTS release: Yes, but node 8 is coming next month, and migrating you codebase to the new version will most likely take little to no effort. [UPDATE]: Node 8 LTS is now out.

Follow me on twitter @imgaafar