wecodetheweb

Async ... await in Javascript

ES6 Javascript Node

Niels Gerritsen · Oct 27, 2017

We've all been through callback hell, maybe we use Promises and Observables to get some relief. Will async await liberate us once and for all?

Callback heaven 😇

Callbacks are the single most important language feature that enables asynchronous programming in Javascript. Take a look at the following example:

```
function onClick() {
  message.textContent ='Clicked!';
}

button.addEventListener('click', onClick);

message.textContent = 'Welcome!';
```

What happens in this piece of code, is that we pass a *callback* (the `onClick` function) to the `addEventListener` function. This tells the browser "When the button is clicked, call this function". After passing the callback, the Javascript engine will just continue

executing the code below it. When the user clicks the button, the browser will tell the Javascript engine to execute the `onClick` function as soon as possible.

> The mechanism that manages the callbacks between Javascript and the browser (or Node.js) is called the *Event loop*. This non-blocking way of doing user interaction is what makes Javascript so suitable for dynamic user interfaces in the browser. Using the same concept for disk I/O and networking is what also makes it really popular for highly dynamic web and IoT services using Node.js.

Promises to the rescue

Some of you might be familiar with callback hell. It's basically when you have a lot of callbacks, and those callbacks register other callbacks, and other callbacks, and everything becomes one big spaghetti of callbacks.

A solution for making that situation better is Promises. You can think of Promises as convenient callback 'containers'. They provide a cleaner interface for using callbacks:

```
fetch('./some-data')
  .then(data => data.json())
  .then(json => console.log(json));
```

As you can see, whe don't directly pass a callback to the `fetch` function, instead it returns an object with a `.then()` method, to which you can pass the callback. The advantage is that the then method returns another promise, resulting in the ability to chain then's. This is especially useful when doing multiple asynchronous operations. But be aware, these are still callbacks! Just more nice and tidy.

> Promises are just a 'nicer', standardized way to deal with callbacks.

Callback hell 🔥

So even though callback hell is less likely with Promises, you can still get some nasty callback constructions, even with Promises:

```
fetchUsers()
  .then((users) => {
    return fetchScores(users)
      .then((scores) => {
        return users.map(user => ({
          ...user,
          score: scores[user.id]
        }));
      });
  });
```

Why do we need to nest the second Promise? Well, in this example we kinda need to, because we need the users after fetching the scores. We could probably solve this case differently, but you're gonna get in these more complex situations at some point.

Callbacks will always be there in Javascript, and that's not a bad thing, it's what enables Javascripts asynchronousity. It's just hard when dealing with a lot of them.

Async await

Async await is a new syntax that is released with ES2017. It uses two keywords: **async** and **await** to make asynchronous logic easier to write.

The *async* keyword can be used to mark a function as *asynchronous*:

```
async function fetchUsersWithScores() {
  // Now an async function
}
```

Asynchronous functions **always** return a Promise. This `fetchUsersWithScores` function will now return a Promise, even if it's only doing synchronous logic.

The *await* keyword is then used to handle Promises inside the function:

```
async function fetchUsersWithScores() {
  const users = await fetchUsers();

  return users;
```

```
}
```

We fetch the users using the same function as in the Promise example. But do you notice how we are not chaining `.then()` to *fetchUsers*, although it returns a Promise? This is because *await* handles that Promise for us. It 'pauses' the function until *fetchUsers* is done, and returns the result.

> Async marks a function as asynchronous, the function will always return a Promise. Await handles Promises inside the async function, making the function's inner logic synchronous.

We can now fetch the scores and tie them together easily:

```
async function fetchUsersWithScores() {
  const users = await fetchUsers();
  const scores = await fetchScores(users);

  return users.map(user => ({
    ...user,
    score: scores[user.id]
  }));
}
```

So long, callback hell!

Doesn't this kill the asynchronous nature of Javascript?

Short answer, no! The async keyword marks a specific function as asynchronous, await *only* blocks the execution of that function, not all other functions in the application. You can still fully leverage have concurrency when using **async await**.

Note that the `fetchUsersWithScores` function itself still returns a Promise:

```
fetchUsersWithScores()
  .then(users => console.log(users));
```

We could actually run that function in parallel with something else if we wanted to:

```
fetchUsersWithScores()
  .then(users => console.log(users));

fetchTotalScore()
  .then(score => console.log(score));
```

And we could even create another async function that waits for both to be finished using `Promise.all([...promises])`:

```
async function fetchAllTheThings() {
  const [users, totalScore] = await Promise.all([
    fetchUsersWithScores(),
    fetchTotalScore()
  ]);

  return { users, totalScore };
}
```

> `Promise.all()` waits for all Promises in an array to succeed and returns their results as an array.

You get the drill? Make a function async, await Promises inside, and return the result. But what about errors?

Error handling

Normally you can chain a `.catch()` to a Promise to handle possible errors. However, as you've just seen, with await you get a single value as output. When an error occurs, it will throw the error and you can simply handle that with a regular **try - catch**:

```
async function fetchUsersWithScores() {
  try {
    const users = await fetchUsers();
    const scores = await fetchScores(users);

    return users.map(user => ({
```

```
      ...user,
      score: scores[user.id]
    }));
  } catch (error) {
    console.error(error.stack);
  }
}
```

Another thing you could do, is not catching the error inside the async function (because you might not be able to do anything useful with the error there), but chain a catch to the output of the async function:

```
fetchUsersWithScores()
  .then((usersWithScores) => {
    showUsers(usersWithScores)
  })
  .catch((error) => {
    showErrorPopup(error.message)
  });
```

Loops

How do we go about looping? You might be tempted to use the `.forEach()`:

```
async function saveUsers(users) {
  users.forEach((user) => {
    await saveUser(user)
  });
}
```

Nope, won't work! If you look closely, the await is *inside* a callback, which is another function, this will crash because that callback is not async. How about making it async?

```
async function saveUsers(users) {
  users.forEach(async (user) => {
    await saveUser(user)
  });
}
```

This does work, but `saveUsers` will **not** wait for the result of the `saveUser` calls to be finished. We could try to use a map:

```
async function saveUsers(users) {
  const promises = users.map(async (user) => {
    await saveUser(user)
  });

  await Promise.all(promises);
}
```

Hmmm, functionally this would work, but there is no point for that async callback to exist now. We could just map the `saveUser` calls as promises and be done with it:

```
function saveUsers(users) {
  return Promise.all(users.map(saveUser));
}
```

As you can see, no need for async await here.

There is also another point of attention, the `saveUser` calls are now parallel. But what if you don't want that? What if your database can only handle one at a time? This is a case where async await can come in handy again! A nice and clean wait to do this is by just using a simple **for...of** loop:

```
async function saveUsers(users) {
  for (user of users) {
    await saveUser(user)
  }
}
```

> When doing repetitive asynchronous operations in parralel, map the promises to an array and await the `Promise.all()`. When a sequential flow is required, use a `for...of` with awaits.

This is a great advantage of async await, you can use normal control structures like `for`, `do/while`, `switch` and `if/else` with asynchronous operations! Another example:

```
async function logIn(username, password) {
  const sessionToken = await doLoginRequest(username, password);

  if (!sessionToken) {
    throw new Error('Login failed');
  }

  return await getUserData(username, sessionToken);
}
```

Callback hell solved?

I must say I'm pretty exited for the future. Async await solves the issues with combining multiple Promises and makes complex asynchronous control flows more easy to code.

I have a point of attention though. Async await 'hides' some of the handling of Promises and callbacks, but it's still vital for newcomers to understand how these concepts work, or they will not surive Javascript. You still need Promises to handle the async functions and understand that await takes a Promise. You will also still need callbacks for tons of other situations.

When can I use it?

Now! Async await is released with the ES2017 spec. For Node.js users, if you upgrade to version 8+, you're good to go!

On the front-end you will need a transpiler like Babel to make async await shine. For Babel I would recommend using the 'env' preset with the 'regenerator runtime' to make it work. Note that there is a cost in the form of extra kilobytes when using the regenerator runtime.

Reference

- Philip Roberts: What the heck is the event loop anyway? | JSConf EU 2014 youtube.com/watch?v=8aGhZQkoFbQ
- async function Reference developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Promise Reference developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

---

Related posts

Meta programming with ES6 Proxies

ES6 Destructuring

Flux, what and why?