Async/await is a relatively new way to write asynchronous code in Javascript. Before we used callbacks (http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/) and promises (https://scotch.io/tutorials/javascript-promises-for-dummies). Async/await actually builds on top of promises. If you have not used promises before, this is a good point to go brush up on them, find link to a useful article here (http://techinpink.com/2017/02/24/introduction-to-javascript-promises/).

Why `async/await` **?** Well, simply put, async/await allows us to write asynchronous code in a synchronous manner.

Async functions

Table of Contents

To create an async function all we need to do is add the `async` keyword before the function definition, like this:

```js
async function asyncFunc() {
  return "Hey!";
}
```

The one thing you need to know about async functions is that; **they always returns a promise**.

In the case where we explicitly return something that is not a promise, like above, the return value is automatically wrapped into a resolved promise with the resolved value being the non-promise. For the code above, `asyncFunc().then(result => console.log(result))` will return the string `Hey!` .

Await

The `await` keyword **can only be used within an** `async` **block**, otherwise it'll throw a syntax error. This means you cannot use `await` in the top level of our code, basically, don't use it by itself.

When do we use it? If we have an asynchronous function inside of an async block. So let's say we need to fetch some data from our server and then use that data within our async block. We will use `await` to pause the function execution and resume after the data comes in. For example;

JS

```
async function asyncFunc() {
  // fetch data from a url endpoint
  const data = await axios.get("/some_url_endpoint");
  return data;
}
```

Why use await? Instead of using await we could have just used a promise right?

JS

```
async function asyncFunc() {
  let data;
  // fetch data from a url endpoint
  axios.get("/some_url_endpoint")
    .then((result) => {
      data = result
    });
  return data;
}
```

Await is simply a more elegant way to write a promise **within an async function**. It improves readability immensely and hence the reason we use it.

Let's assume we have a couple of asynchronous functions within our `async` block. Instead of chaining promises we could do this, which is much cleaner:

```
JS
async function asyncFunc() {
  // fetch data from a url endpoint
  const response = await axios.get("/some_url_endpoint");
  const data = await response.json();

  return data;
}
```

Error handling

How do we handle errors? We have a few options, let's explore them:

Try..catch

This is the most common way to handle errors when using `async-await`, good old `try-catch`. All you need to do is encapsulate your code in a `try` block and handle any errors that occur in a `catch`.

```js
async function asyncFunc() {
  try {
    // fetch data from a url endpoint
    const data = await axios.get("/some_url_endpoint");
    return data;
  } catch(error) {
    console.log("error", error);
    // appropriately handle the error
  }
}
```

If an error occurs when fetching data from our endpoint, execution is transferred to the `catch` block and we can handle whatever error is thrown. If we have multiple `await` lines the catch block catches the errors that occur in all the lines.

```js
async function asyncFunc() {
  try {
    // fetch data from a url endpoint
    const response = await axios.get("/some_url_endpoint"
    const data = await response.json();

    return data;
  } catch (error) {
    alert(error); // catches both errors
  }
}
```

If not try..catch

We can alternatively append `.catch()` on the promise generated by the `async` function. Let's recap: Remember that the `async` function returns a promise. If an error occurs then it returns a rejected promise.So on the function call we could do this:

```js
asyncFunc().catch((error) => {
  // handle error appropriately
});
```

FOLLOW @JOYKARE_(HTTPS://TWITTER.C

Software engineer, stationed in Nairobi, Kenya.

Main stack => Javascript. Front-end dev => ReactJS.

YOLO!

VIEW MY 9 POSTS

Fun facts

Async on class methods

Class methods can be `async`.


```js
class Example{
  async asyncMethod() {
    const data = await axios.get("/some_url_endpoint");
    return data
  }
}
```

To call the method we'd do:

```
JS

const exampleClass = new Example();
exampleClass.asyncMethod().then(//do whatever you want
```

Await is thenable

We can append a `.then()` on an await.

```
JS

async function asyncFunc() {
  // fetch data from a url endpoint
  const data = await axios.get("/some_url_endpoint")
    .then((result) => return result.names)

  return data;
}
```

Await <> Promise.all

If we have multiple promises we could use `Promise.all` with `await`
.

```js
async function asyncFunc() {
  const response = await Promise.all([
    axios.get("/some_url_endpoint"),
    axios.get("/some_url_endpoint")
  ]);
  ...
}
```

Conclusion

In summary, `async/await` is a cleaner syntax to write asynchronous Javascript code. It enhances readability and flow of your code.

Things to keep in mind while using `async/await` :

- » Async functions return a promise.
- » Await can only be used inside an async block.
- » Await waits until the function("promise") resolves or rejects.

Its quite easy to learn and use. Enjoy experimenting!!