While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the “report an issue“ button at the bottom of the tutorial.

Introduction

Promises give us an easier way to deal with asynchrony in our code in a sequential manner. Considering that our brains are not designed to deal with asynchronicity efficiently, this is a much welcome addition. **Async/await functions**, a new addition with ES2017 (`ES8`), help us even more in allowing us to write completely synchronous-looking code while performing asynchronous tasks behind the scenes.

The functionality achieved using async functions can be recreated by combining promises with generators, but async functions give us what we need without any extra boilerplate code.

Simple Example

In the following example, we first declare a function that returns a promise that resolves to a value of 🤡 after 2 seconds. We then declare an `async` function and `await` for the promise to resolve before logging the message to the console:

```
function scaryClown() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('🤡');
    }, 2000);
```

Copy

```
  });
}

async function msg() {
  const msg = await scaryClown();
  console.log('Message:', msg);
}

msg(); // Message: 🤡 <-- after 2 seconds
```

`await` is a new operator used to wait for a promise to resolve or reject. It can only be used inside an async function.

The power of async functions becomes more evident when there are multiple steps involved:

Copy

```
function who() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('🤡');
    }, 200);
  });
}

function what() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('lurks');
    }, 300);
  });
}

function where() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('in the shadows');
    }, 500);
  });
}

async function msg() {
  const a = await who();
  const b = await what();
  const c = await where();
```

```
  console.log(`${ a } ${ b } ${ c }`);
}

msg(); // 🤡 lurks in the shadows <-- after 1 second
```

A word of caution however, in the above example each step is done sequentially, with each additional step waiting for the step before to resolve or reject before continuing. If you instead want the steps to happen in parallel, you can simply use `Promise.all` to wait for all the promises to have fulfilled:

```
// ...

async function msg() {
  const [a, b, c] = await Promise.all([who(), what(), where()]);

  console.log(`${ a } ${ b } ${ c }`);
}

msg(); // 🤡 lurks in the shadows <-- after 500ms
```

`Promise.all` returns an array with the resolved values once all the passed-in promises have resolved.

In the above we also make use of some nice array destructuring to make our code succinct.

Promise-Returning

Async functions always return a promise, so the following may not produce the result you're after:

```
async function hello() {
  return 'Hello Alligator!';
}

const b = hello();

console.log(b); // [object Promise] { ... }
```

Since what's returned is a promise, you could do something like this instead:

```
async function hello() {
  return 'Hello Alligator!';
}

const b = hello();

b.then(x => console.log(x)); // Hello Alligator!
```

Copy

...or just this:

```
async function hello() {
  return 'Hello Alligator!';
}

hello().then(x => console.log(x)); // Hello Alligator!
```

Copy

Different Forms

So far with our examples we saw the async function as a function declaration, but we can also define async function expressions and async arrow functions:

Async Function Expression

Here's the async function from our first example, but defined as a function expression:

```
const msg = async function() {
  const msg = await scaryClown();
  console.log('Message:', msg);
}
```

Copy

Async Arrow Function

Here's that same example once again, but this time defined as an arrow function:

```
const msg = async () => {
  const msg = await scaryClown();
  console.log('Message:', msg);
}
```

Copy

Error Handling

Something else that's very nice about async functions is that error handling is also done completely synchronously, using good old **try...catch** statements. Let's demonstrate by using a promise that will reject half the time:

```
function yayOrNay() {
  return new Promise((resolve, reject) => {
    const val = Math.round(Math.random() * 1); // 0 or 1, at random

    val ? resolve('Lucky!!') : reject('Nope 😠');
  });
}

async function msg() {
  try {
    const msg = await yayOrNay();
    console.log(msg);
  } catch(err) {
    console.log(err);
  }
}

msg(); // Lucky!!
msg(); // Lucky!!
msg(); // Lucky!!
msg(); // Nope 😠
```

Copy

Given that async functions always return a promise, you can also deal with unhandled errors as you would normally using a catch statement:

```
async function msg() {
  const msg = await yayOrNay();
  console.log(msg);
}

msg().catch(x => console.log(x));
```

Copy

This synchronous error handling doesn't just work when a promise is rejected, but also when there's an actual runtime or syntax error happening. In the following example, the second time with call our `msg` function we pass in a `number` value that doesn't have a `toUpperCase` method in its prototype chain. Our **try...catch** block catches that error just as well:

```
function caserUpper(val) {
  return new Promise((resolve, reject) => {
    resolve(val.toUpperCase());
  });
}

async function msg(x) {
  try {
    const msg = await caserUpper(x);
    console.log(msg);
  } catch(err) {
    console.log('Ohh no:', err.message);
  }
}

msg('Hello'); // HELLO
msg(34); // Ohh no: val.toUpperCase is not a function
```

Copy

Async Functions With Promise-Based APIS

As we showed in our primer to the Fetch API, web APIs that are promise-based are a perfect candidate for async functions:

```
async function fetchUsers(endpoint) {
  const res = await fetch(endpoint);
  let data = await res.json();

  data = data.map(user => user.username);

  console.log(data);
}

fetchUsers('https://jsonplaceholder.typicode.com/users');
// ret", "Antonette", "Samantha", "Karianne", "Kamren", "Leopoldo_Corkery"
```

Copy

Browser Support: As of 2020, 94% of browsers worldwide can handle async/await in javascript Notable exceptions are IE11 and Opera Mini.

Conclusion

Before **Async/await functions,** JavaScript code that relied on lots of asynchronous events *(for example: code that made lots of calls to APIs)* would end up in what some called “callback hell” - A chain of functions and callbacks that was very difficult to read and understand.

Async and await allow us to write asynchronous JavaScript code that reads much more clearly.

About the authors

Alligator.io Author