Modern Promises With async And await

15 minutes

Modern `Promise`s With `async` And `await`

While `Promise`s helped revolutionize the way that JavaScript programmers could structure asynchronous code, the technical committee that governs JavaScript realized it could take this feature one step further. It could design a language feature that allowed programmers to write true synchronous code based on `Promise`s. Thus, the `async` and `await` keywords came into being in 2017.

When you finish this article, you should be able to:

- Properly implement a function declaration with the `async function` expression;
- Explain how the JavaScript runtime creates an implicit `Promise` for `async function` declarations; and,
- Execute functions marked with the `async` keyword using the `await` keyword.

Classic promise example

Let's review with an example of classic promise handling using two functions. `walkTheDog` will return a promise that resolves with a `'happy dog'` after 1 second. `doChores` will act as our main function and will handle that promise with a `then`:

```
function walkTheDog() {
```

```
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

function doChores() {
  console.log('before walking the dog');
  walkTheDog()
    .then(res => {
        console.log(res);
        console.log('after walking the dog');
    });
  return 'done';
}

console.log(doChores());

// prints:
//
// before walking the dog
// done
// happy dog
// after walking the dog
```

Notice that `'done'` will be returned by `doChores` before the promise resolves, because it is asynchronous.

If we want to take any actions after the promise resolves, we can do so by chaining `then`. This code works, but we may have one complaint regarding aesthetics: there is some added bulk because the `then` accepts a callback containing the code we want to execute after the promise resolves. This bulk compounds further if we want to chain multiple `then`s. Let's refactor this. Enter `async` and `await`.

`async` function declarations

Declaring a function with async will create the function so it returns an implicit promise containing it's result. Let's declare our `doChores` function as `async` and check it's return value. For now we'll leave out the explicit `walkTheDog` promise:

```
async function doChores() {
  // ...
  return 'done';
```

```
  return 'done';
}

console.log(doChores());
// prints:
// Promise { 'done' }
```

This function now returns a promise automatically! Notice that the promise returned contains the immediately resolved value of `'done'`. An `async` declaration isn't super useful by itself. However, it allows us to utilize the `await` keyword inside the function.

`await` ing a promise

The await operator can be used to wait for promise to be fulfilled. We are only allowed to use `await` in an `async` function. Using `await` outside of an `async` will result in a SyntaxError. When a promise is `await` ed, execution of the containing `async` function will pause until the promise is fulfilled.

Let's use `await` in our `doChores` function:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('before walking the dog');
  const res = await walkTheDog();
  console.log(res);
  console.log('after walking the dog');
}

doChores();
// prints:
// before walking the dog
// happy dog
// after walking the dog
```

Whoa! This code looks synchronous. Instead of using `then`, we can `await` the `walkTheDog()` promise, pausing execution until the promise is fulfilled. Once fulfilled, the `await` expression will evaluate to the `resolve` d value, `'happy dog'` in this case.

`happy dog` in this case.

Remember that the `async doChores` function will implicitly return a promise. Now that promise will fulfill once the entire function is finished executing. The function's return value will be the resolved value of the implicit promise. Let's handle it with `then`:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('before walking the dog');
  const res = await walkTheDog();
  console.log('after walking the dog');
  return res.toUpperCase();
}

doChores().then(result => console.log(result));
// prints:
// before walking the dog
// after walking the dog
// HAPPY DOG
```

You're probably wondering why we chain `then` and not simply use `await doChores()`, that's because we can only use `await` *inside* of an `async` function. Currently our call to `doChores` is not within any function.

For fun, let's use a surrounding `async` function to `await doChores()`. We'll also add some numbered print statements to show the order of execution:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('2');
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('1');
  const res = await walkTheDog();
  console.log('3');
```

```
  return res.toUpperCase();
}

async function wrapper() {
  console.log('0');
  const finalResult = await doChores();
  console.log('4');
  console.log(finalResult + '!!!');
}

wrapper();
// prints:
// 0
// 1
// 2
// 3
// 4
// HAPPY DOG!!!
```

Refactoring a promise chain

Refactoring a promise chain is straightforward with `async` / `await`. Let's say wanted to print the resolved values for 3 promises in order:

```
function wrapper() {
  promise1
    .then(res1 => {
      console.log(res1);
      return promise2;
    })
    .then(res2 => {
      console.log(res2);
      return promise3;
    })
    .then(res3 => {
      console.log(res3);
    });
}
```

We can refactor it into this:

```
async function wrapper() {
  console.log(await promise1);
  console.log(await promise2);
  console.log(await promise3);
  console.log(await promise4);
}
```

Error handling

Since `async` / `await` allows for seemingly synchronous execution, we can use a normal `try...catch` pattern to handle errors when the promise is rejected:

```
function action() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('uh-oh'); // rejected
    }, 3000);
  });
}

async function handlePromise() {
  try {
    const res = await action();
    console.log('resolved with', res);
  } catch (err) {
    console.log('rejected because of', err);
  }
}

handlePromise();
// prints:
// rejected because of: uh-oh
```

What you learned

In this article, you learned how to use the `async` and `await` keywords in modern JavaScript to truly turn asynchronous code into synchronous style code. You will want to do this to make the code more readable and maintainable. The steps to do this are to

- mark a function with the `async` keyword to make JavaScript have it return a `Promise` if your code doesn't, and
- use the `await` keyword to turn the invocation of a function marked `async` into a blocking call that returns the resolved value of the underlying `Promise` or throws an exception.

Did you find this lesson helpful?

No   Yes

✓ Mark As Complete

Finished with this task? Click **Mark as Complete** to continue to the next page!