3rd February 2020

Async/await

There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.

Async functions

Let’s start with the `async` keyword. It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

For instance, this function returns a resolved promise with the result of `1` ; let’s test it:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

…We could explicitly return a promise, which would be the same:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

So, `async` ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, `await` , that works only inside `async` functions, and it’s pretty cool.

Await

The syntax:

```
// works only inside async functions
let value = await promise;
```

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

Here's an example with a promise that resolves in 1 second:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

The function execution "pauses" at the line `(*)` and resumes when the promise settles, with `result` becoming its result. So the code above shows "done!" in one second.

Let's emphasize: `await` literally makes JavaScript wait until the promise settles, and then go on with the result. That doesn't cost any CPU resources, because the engine can do other jobs in the meantime: execute other scripts, handle events, etc.

It's just a more elegant syntax of getting the promise result than `promise.then`, easier to read and write.

⚠ Can't use `await` in regular functions

If we try to use `await` in non-async function, there would be a syntax error:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

We will get this error if we do not put `async` before a function. As said, `await` only works inside an `async function`.

Let's take the `showAvatar()` example from the chapter Promises chaining and rewrite it using `async/await`:

1. We'll need to replace `.then` calls with `await`.
2. Also we should make the function `async` for them to work.

```
async function showAvatar() {

  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Pretty clean and easy to read, right? Much better than before.

`await` won't work in the top-level code

People who are just starting to use `await` tend to forget the fact that we can't use `await` in top-level code. For example, this will not work:

```
// syntax error in top-level code
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

But we can wrap it into an anonymous async function, like this:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

`await` accepts "thenables"

Like `promise.then`, `await` allows us to use thenable objects (those with a callable `then` method). The idea is that a third-party object may not be a promise, but promise-compatible: if it supports `.then`, that's enough to use it with `await`.

Here's a demo `Thenable` class; the `await` below accepts its instances:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // resolve with this.num*2 after 1000ms
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
};

async function f() {
  // waits for 1 second, then result becomes 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

If `await` gets a non-promise object with `.then`, it calls that method providing the built-in functions `resolve` and `reject` as arguments (just as it does for a regular `Promise` executor). Then `await` waits until one of them is called (in the example above it happens in the line `(*)`) and then proceeds with the result.

Async class methods

To declare an async class method, just prepend it with `async`:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
```

The meaning is the same: it ensures that the returned value is a promise and enables `await`.

Error handling

If a promise resolves normally, then `await promise` returns the result. But in the case of a rejection, it throws the error, just as if there were a `throw` statement at that line.

This code:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

…is the same as this:

```
async function f() {
  throw new Error("Whoops!");
}
```

In real situations, the promise may take some time before it rejects. In that case there will be a delay before `await` throws an error.

We can catch that error using `try..catch`, the same way as a regular `throw`:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

In the case of an error, the control jumps to the `catch` block. We can also wrap multiple lines:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

f();
```

If we don't have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add `.catch` there, then we get an unhandled promise error (viewable in the console). We can catch such errors using a global `unhandledrejection` event handler as described in the chapter Error handling with promises.

`async/await` and `promise.then/catch`

When we use `async/await`, we rarely need `.then`, because `await` handles the waiting for us. And we can use a regular `try..catch` instead of `.catch`. That's usually (but not always) more convenient.

But at the top level of the code, when we're outside any `async` function, we're syntactically unable to use `await`, so it's a normal practice to add `.then/catch` to handle the final result or falling-through error, like in the line `(*)` of the example above.

`async/await` works well with `Promise.all`

When we need to wait for multiple promises, we can wrap them in `Promise.all` and then `await`:

```
// wait for the array of results
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

In the case of an error, it propagates as usual, from the failed promise to `Promise.all`, and then becomes an exception that we can catch using `try..catch` around the call.

Summary

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows `await` to be used in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it's an error, the exception is generated — same as if `throw error` were called at that very place.
2. Otherwise, it returns the result.

Together they provide a great framework to write asynchronous code that is easy to both read and write.

With `async/await` we rarely need to write `promise.then/catch`, but we still shouldn't forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also

`Promise.all` is nice when we are waiting for many tasks simultaneously.

Tasks

Rewrite using async/await

Rewrite this example code from the chapter Promises chaining using `async/await` instead of `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    })
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404
```

solution

Rewrite "rethrow" with async/await

Below you can find the “rethrow” example from the chapter Promises chaining. Rewrite it using `async/await` instead of `.then/catch`.

And get rid of the recursion in favour of a loop in `demoGithubUser`: with `async/await` that becomes easy to do.

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}
```

```javascript

// Ask for a user name until github returns a valid user
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

solution

Call async from non-async

We have a “regular” function. How to call `async` from it and use its result?

```javascript
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // ...what to write here?
  // we need to call async wait() and wait to get 10
  // remember, we can't use "await"
}
```

P.S. The task is technically very simple, but the question is quite common for developers new to async/await.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)