Async/await

There's a special syntax to work with promises in a more comfort 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 function, like this:

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

The word "async" before a function means one simple thing: a function always returns a promise. If the code has `return <non-promise>` in it, then JavaScript automatically wraps it into a resolved promise with that value.

For instance, the code above 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, that would be the same:

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

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

So, `async` ensures that the function returns a promise, 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:

```js
// 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 example with a promise that resolves in 1 second:

```js
async function f() {

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

  let result = await promise; // wait till 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 meanwhile: 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, that would be a syntax error:

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

We can get such error in case if we forget to put `async` before a function. As said, `await` only works inside `async function`.

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

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

```js
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 that, but we can't write `await` in the top-level code. That wouldn't work:

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

So we need to have a wrapping async function for the code that awaits. Just as in the example above.

`await` accepts thenables

Like `promise.then`, `await` allows to use thenable objects (those with a callable `then` method). Again, the idea is that a 3rd-party object may be not a promise, but promise-compatible: if it supports `.then`, that's enough to use with `await`.

For instance, here `await` accepts `new Thenable(1)`:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // 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 native functions `resolve`, `reject` as arguments. 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 methods

A class method can also be async, just put `async` before it.

Like here:

```
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 case of a rejection it throws the error, just 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. So `await` will wait, and then throw 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 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 (and can see it in the console). We can catch such errors using a global event handler as described in the chapter Promises chaining.

`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 (not always) more convenient.

But at the top level of the code, when we're outside of 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 errors.

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 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 to use `await` 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, so we can assign it to a value.

Together they provide a great framework to write asynchronous code that is easy both to 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 a nice thing to wait for many tasks simultaneously.

Tasks

Rewrite using async/await

Rewrite the one of examples 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') // (3)
  .catch(alert); // Error: 404
```

solution

Rewrite "rethrow" 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);
      }
    })
}

```

```
// 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

Previous lesson

Share Tutorial map

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- 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…)
- If you can't understand something in the article – please elaborate.

Aditya Agarwal • 5 months ago

have you started using async/await? If so be careful as you may have unknowingly crippled the performance of your application. See if this is the case here
https://medium.freecodecamp...

5 ^ | ⌄ • Reply • Share ›

solstice333 ➦ Aditya Agarwal • 3 months ago

good article. It's easy to get trapped coding synchronous logic when using `await`. IMO the easiest way to solve that is to group concurrent async calls with `Promise.all`

1 ^ | ⌄ • Reply • Share ›

vermoid • 3 months ago

This is the best async await getting started overview. Thanks for the effort.

3 ^ | ⌄ • Reply • Share ›

Varun verma • 3 months ago

you made it so simple and easy to understand

1 ^ | ⌄ • Reply • Share ›

jugal joshi • 7 months ago

Why to wrap promise.all with await... promise.all will wait itself till all tasks has been resolved.

1 ^ | ⌄ • Reply • Share ›

Mark McCoid ➦ jugal joshi • 6 months ago

Because Promise.all still returns a promise, which would need to be handled with a .then().
So, yes, you still need either the await keyword or a .then() statement.

6 ^ | ⌄ • Reply • Share ›

jagan • 20 days ago

Good Article

^ | ⌄ • Reply • Share ›

Omar Chajia • 21 days ago

Very straightforward, Thank you!

^ | ⌄ • Reply • Share ›

Prakash Sellathurai • 24 days ago

hhhh

^ | ⌄ • Reply • Share ›

Prakash Sellathurai • 24 days ago

hhh

^ | ⌄ • Reply • Share ›

Prakash Sellathurai • 24 days ago

hhhh

• Reply • Share ›

Aditya Mittal • a month ago

My two-cents:
1. If I promise my dad something, I must resolve/reject my promise and my dad must then/catch.
2. Async function always returns a promises so by point 1, the receiver must then/catch.

I say this because when I should .resolve and when I should .then was a big source of confusion for me when I was trying to learn from various tutorials.

• Reply • Share ›

Dave Thomas • a month ago

Impeccably well written!

• Reply • Share ›

Cyril • a month ago

When it says the operation waits at (*) but does not use computing resources, what is meant? what does the javascript engine do as its next line of execution?

• Reply • Share ›

Tomáš Poremba ➦ Cyril • 23 days ago

JS engine can do anything else, like updating your UI, handling other asynchronous taskt etc.

• Reply • Share ›

Narayan Choudhary • 2 months ago

nice

• Reply • Share ›

elgselgs • 3 months ago

Thank you so much!

• Reply • Share ›

Mikk Laos • 3 months ago

Thanks

• Reply • Share ›

曾琦瑋 • 6 months ago

It's a useful and clear tutorial of async and await for me. Thanks for your sharing!

• Reply • Share ›

Lom Se • 7 months ago

So easy to understand. Thanks for sharing this.

• Reply • Share ›

Anirudh Mathad • 8 months ago

1. What is the (*) references above ?

2. My solution for the last problem: http://plnkr.co/edit/GxGzUP...

• Reply • Share ›

ekanna • 9 months ago

Excellent!

• Reply • Share ›

Shaun Barry Botsis • a year ago

nice

• Reply • Share ›

ALSO ON **JAVASCRIPT.INFO**

Currying and partials

10 comments • a year ago

Avatar **Abhishek Mehandiratta** — The above function curried doesn't work for object methods.. The error hints that 'this' is not passed properly. …

Resource loading: onload and onerror

1 comment • a year ago

Avatar **pinto xavier** — if there are multiple scripts to be loaded like this statement preparation shown above how do i identify each script load …

JavaScript courses

4 comments • 6 months ago

Avatar **Parth** — I am professional JavaScript developer and I really like this tutorial and I highly

Capturing groups

4 comments • a year ago

Avatar **Shopein Tolumide** — I guess my answer would have its limitations compared to the one

contact us

about the project

RU / EN

powered by node.js & open source