Free email newsletter: “ES.next News”

2016-10-22

Tips for using async functions (ES2017)

Labels: dev, esnext, javascript

This blog post gives tips for using async functions. If you are unfamiliar with them, you can read chapter “Async functions” in “Exploring ES2016 and ES2017”.

1. Know your Promises

The foundation of async functions is Promises. That’s why understanding the latter is crucial for understanding the former. Especially when connecting old code that isn’t based on Promises with async functions, you often have no choice but to use Promises directly.

For example, this is a “promisified” version of `XMLHttpRequest`:

```
function httpGet(url, responseType="") {
    return new Promise(
        function (resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            };
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            xhr.responseType = responseType;
            request.send();
        });
}
```

The API of `XMLHttpRequest` is based on callbacks. Promisifying it via an async function would mean that you’d have to fulfill or reject the Promise returned by the function from within callbacks. That’s impossible, because you can only do so via `return` and `throw`. And you can’t `return` the result of a function from within a callback. `throw` has similar constraints.

Therefore, the common coding style for async functions will be:

- Use Promises directly to build asynchronous primitives.
- Use those primitives via async functions.

Further reading: chapter “Promises for asynchronous programming” in “Exploring ES6”.

2. Async functions are started synchronously, settled asynchronously

Most popular (last 30 days)

ECMAScript 2017: the final feature set
ECMAScript 6 modules: the final syntax
ES proposal: import() – dynamically importing ES modules
Making transpiled ES modules more spec-compliant
ES proposal: Shared memory and atomics
Classes in ECMAScript 6 (final semantics)
Communicating between Web Workers via MessageChannel

Most popular (all time)

ECMAScript 6 modules: the final syntax
Classes in ECMAScript 6 (final semantics)
Iterating over arrays and objects in JavaScript
ECMAScript 6’s new array methods
The final feature set of ECMAScript 2016 (ES7)
WebAssembly: a binary format for the web
Basic JavaScript for the impatient programmer
Google Dart to “ultimately ... replace JavaScript”
Six nifty ES6 tricks
Google’s Polymer and the future of web UI frameworks

Blog archive

► 2017 (4)
▼ 2016 (38)

Dr. Axel Rauschmayer

This is how async functions are executed:

1. The result of an async function is always a Promise `p`. That Promise is created when starting the execution of the async function.
2. The body is executed. Execution may finish permanently via `return` or `throw`. Or it may finish temporarily via `await`; in which case execution will usually continue later on.
3. The Promise `p` is returned.

While executing the body of the async function, `return x` resolves the Promise `p` with `x`, while `throw err` rejects `p` with `err`. The notification of a settlement happens asynchronously. In other words: the callbacks of `then()` and `catch()` are always executed after the current code is finished.

The following code demonstrates how that works:

```
async function asyncFunc() {
    console.log('asyncFunc()'); // (A)
    return 'abc';
}
asyncFunc().
then(x => console.log(`Resolved: ${x}`)); // (B)
console.log('main'); // (C)

// Output:
// asyncFunc()
// main
// Resolved: abc
```

You can rely on the following order:

1. Line (A): the async function is started synchronously. The async function's Promise is resolved via `return`.
2. Line (C): execution continues.
3. Line (B): Notification of Promise resolution happens asynchronously.

3. Returned Promises are not wrapped

Resolving a Promise is a standard operation. `return` uses it to resolve the Promise `p` of an async function. That means:

1. Returning a non-Promise value fulfills `p` with that value.
2. Returning a Promise means that `p` now mirrors the state of that Promise.

Therefore, you can return a Promise and that Promise won't be wrapped in a Promise:

```
async function asyncFunc() {
    return Promise.resolve(123);
}
asyncFunc()
.then(x => console.log(x)) // 123
```

Intriguingly, returning a rejected Promise leads to the result of the async function being rejected (normally, you'd use `throw` for that):

```
async function asyncFunc() {
    return Promise.reject(new Error('Problem!'));
}
asyncFunc()
.catch(err => console.error(err)); // Error: Problem!
```

That is in line with how Promise resolution works. It enables you to forward both fulfillments and rejections of another asynchronous computation, without an `await`:

```
async function asyncFunc() {
    return anotherAsyncFunc();
}
```

- ▶ December (1)
- ▶ November (4)
- ▼ October (4)
  - Three ways of understanding Promises
  - Tips for using async functions (ES2017)
  - ES proposal: asynchronous iteration
  - ES proposal: Rest/Spread Properties
- ▶ September (5)
- ▶ August (1)
- ▶ June (1)
- ▶ May (2)
- ▶ April (2)
- ▶ March (2)
- ▶ February (8)
- ▶ January (8)
- ▶ 2015 (65)
- ▶ 2014 (55)
- ▶ 2013 (98)
- ▶ 2012 (177)
- ▶ 2011 (380)
- ▶ 2010 (174)
- ▶ 2009 (68)
- ▶ 2008 (46)
- ▶ 2007 (12)
- ▶ 2006 (1)
- ▶ 2005 (2)

Labels

dev (592)
javascript (400)
computers (316)
life (194)
jslang (179)
esnext (156)
apple (107)
webdev (95)
mobile (83)
scitech (50)
hack (49)
mac (47)
google (39)
java (37)
ios (33)
business (32)
video (32)
clientjs (31)

Tweets by @rauschma

Axel Rauschm… @rauschma
Enjoying "Troll Hunters":
– "Let's call him 'Gnome Chomsky'"
– "Juliet dies in this? Nooo!"
9h

Axel Rauschmayer Retweeted
Jonathan Creamer @jcreamer898
A nice little shout out to @rauschma... infoworld.com/article/316483 … #es2017 #async
JavaScr… ECMASc… infoworld…
13h

Axel Rauschmayer @rauschma
Not a physics book! twitter.com/ManningBooks/s …
02 Feb

Axel Rauschmayer Retweeted
Jordan Harband @ljharb
Making our React components forbid extra props has caught SO many bugs. I highly recommend it.

npmjs.com/airbnb-prop-ty…
npm: air… Custom … npmjs.com
02 Feb

Axel Rauschmayer Retweeted
Seth Petry-Johnson @spetryjohnson
Open strong. Be bold. Tell a story. Do something to get me interested. Opening w/ the "obligatory 'about me' slide" puts me to sleep :)
02 Feb

Google

hci (27)
entertainment (26)
nodejs (26)
society (26)
browser (25)
firefox (25)
html5 (24)
ipad (24)
movie (23)
psychology (22)
2ality (18)
tv (18)
android (17)
social (17)
chrome (16)
fun (16)
jsmodules (16)
tablet (16)
humor (15)
politics (15)
web (15)
cloud (14)
hardware (14)
microsoft (14)
software engineering (13)
blogging (12)
gaming (12)
eclipse (11)
gwt (11)
numbers (11)
programming languages (11)
app store (10)
media (10)
nature (10)
security (10)
semantic web (10)
software (10)
twitter (10)
webos (10)
12quirks (9)
education (9)
jstools (9)
photo (9)
webcomponents (9)
windows 8 (9)
async (8)
idea (8)
iphone (8)
itunes (8)
scifi-fantasy (8)
app (7)
babel (7)
bookmarklet (7)

The previous code is roughly similar to – but more efficient than – the following code (which unwraps the Promise of `anotherAsyncFunc()` only to wrap it again):

```
async function asyncFunc() {
    return await anotherAsyncFunc();
}
```

4. Don't forget `await`

One easy mistake to make in async functions is to forget `await` when making an asynchronous function call:

```
async function asyncFunc() {
    const value = otherAsyncFunc(); // missing `await`!
    ...
}
```

In this example, `value` is set to a Promise, which is usually not what you want in async functions.

`await` can even make sense if an async function doesn't return anything. Then its Promise is simply used as a signal for telling the caller that it is finished. For example:

```
async function foo() {
    await step1(); // (A)
    ...
}
```

The `await` in line (A) guarantees that `step1()` is completely finished before the remainder of `foo()` is executed.

5. You don't need `await` if you "fire and forget"

Sometimes, you only want to trigger an asynchronous computation and are not interested in when it is finished. The following code is an example:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello'); // don't wait
    writer.write('world'); // don't wait
    await writer.close(); // wait for file to close
}
```

Here, we don't care when individual writes are finished, only that they are executed in the right order (which the API would have to guarantee, but that is encouraged by the execution model of async functions – as we have seen).

The `await` in the last line of `asyncFunc()` ensures that the function is only fulfilled after the file was successfully closed.

Given that returned Promises are not wrapped, you can also `return` instead of `await writer.close()`:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello');
    writer.write('world');
    return writer.close();
}
```

Both versions have pros and cons, the `await` version is probably slightly easier to understand.

6. Parallelism

The following code make two asynchronous function calls, `asyncFunc1()` and `asyncFunc2()`.

```
async function foo() {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2();
}
```

However, these two function calls are executed sequentially. Executing them in parallel tends to speed things up. You can use `Promise.all()` to do so:

```
async function foo() {
    const [result1, result2] = await Promise.all([
        asyncFunc1(),
        asyncFunc2(),
    ]);
}
```

Instead of awaiting two Promises, we are now awaiting a Promise for an Array with two elements.

7. No `await` in callbacks

Remember that `await` only affects the innermost async function that surrounds it and can only be used directly inside async functions. That is a problem if you want to use one of the Array utility functions `map()`, `forEach()`, etc., which rely on callbacks.

7.1. `Array.prototype.map()`

Let's start with the Array method `map()`. In the following code, we want to download the files pointed to by an Array of URLs and return them in an Array.

```
async function downloadContent(urls) {
    return urls.map(url => {
        // Wrong syntax!
        const content = await httpGet(url);
        return content;
    });
}
```

This does not work, because `await` is syntactically illegal inside normal arrow functions. How about using an async arrow function, then?

```
async function downloadContent(urls) {
    return urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
}
```

There are two issues with this code:

- The result is now an Array of Promises, not an Array of strings.
- The work performed by the callbacks isn't finished once `map()` is finished, because `await` only pauses the surrounding arrow function and `httpGet()` is resolved asynchronously. That means you can't use `await` to wait until `downloadContent()` is finished.

We can fix both issues via `Promise.all()`, which converts an Array of Promises to a Promise for an Array (with the values fulfilled by the Promises):

```
async function downloadContent(urls) {
    const promiseArray = urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
    return await Promise.all(promiseArray);
}
```

The callback for `map()` doesn't do much with the result of `httpGet()`, it only forwards it. Therefore, we don't need an async arrow function here, a normal arrow function will do:

chromeos (7)
english (7)
es proposal (7)
fringe (7)
html (7)
jsint (7)
jsshell (7)
thunderbolt (7)
webapp (7)
advancedjs (6)
blogger (6)
crowdsourcing (6)
latex (6)
lion (6)
promises (6)
ted (6)
book (5)
environment (5)
gadget (5)
googleio (5)
intel (5)
jsarrays (5)
jshistory (5)
layout (5)
light peak (5)
michael j. fox (5)
music (5)
pdf (5)
polymer (5)
shell (5)
tc39 (5)
template literals (5)
underscorejs (5)
vlc (5)
__proto__ (4)
coffeescript (4)
concurrency (4)
dart (4)
facebook (4)
gimp (4)
googleplus (4)
health (4)
howto (4)
hp (4)
javafx (4)
kindle (4)
leopard (4)
macbook (4)
motorola (4)
münchen (4)
occupy (4)
pl fundamentals (4)
presenting (4)
publishing (4)

series (4)
textbook (4)
web design (4)
amazon (3)
asmjs (3)
back to the future (3)
bitwise_ops (3)
css (3)
es2016 (3)
flattr (3)
fluentconf (3)
food (3)
foreign languages (3)
house (3)
icloud (3)
info mgmt (3)
jsfuture (3)
jsstyle (3)
linux (3)
mozilla (3)
python (3)
regexp (3)
samsung (3)
tizen (3)
traffic (3)
typedjs (3)
unix (3)
adobe (2)
angry birds (2)
angularjs (2)
astronomy (2)
audio (2)
comic (2)
design (2)
dom (2)
ecommerce (2)
eval (2)
exploring es6 (2)
facebook flow (2)
facets (2)
flash (2)
free (2)
futurama (2)
guide (2)
history (2)
hyena (2)
internet explorer (2)
iteration (2)
journalism (2)
jquery (2)
jsengine (2)
jslib (2)
law (2)
lightning (2)

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return await Promise.all(promiseArray);
}
```

There is one small improvement that we still can make: This async function is slightly inefficient – it first unwraps the result of `Promise.all()` via `await`, before wrapping it again via `return`. Given that `return` doesn't wrap Promises, we can return the result of `Promise.all()` directly:

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return Promise.all(promiseArray);
}
```

7.2. `Array.prototype.forEach()`

Let's use the Array method `forEach()` to log the contents of several files pointed to via URLs:

```
async function logContent(urls) {
    urls.forEach(url => {
        // Wrong syntax
        const content = await httpGet(url);
        console.log(content);
    });
}
```

Again, this code will produce a syntax error, because you can't use `await` inside normal arrow functions.

Let's use an async arrow function:

```
async function logContent(urls) {
    urls.forEach(async url => {
        const content = await httpGet(url);
        console.log(content);
    });
    // Not finished here
}
```

This does work, but there is one caveat: the Promise returned by `httpGet()` is resolved asynchronously, which means that the callbacks are not finished when `forEach()` returns. As a consequence, you can't await the end of `logContent()`.

If that's not what you want, you can convert `forEach()` into a `for-of` loop:

```
async function logContent(urls) {
    for (const url of urls) {
        const content = await httpGet(url);
        console.log(content);
    }
}
```

Now everything is finished after the `for-of` loop. However, the processing steps happen sequentially: `httpGet()` is only called a second time *after* the first call is finished. If you want the processing steps to happen in parallel, you must use `Promise.all()`:

```
async function logContent(urls) {
    await Promise.all(urls.map(
        async url => {
            const content = await httpGet(url);
            console.log(content);
        }));
}
```

markdown (2)
math (2)
meego (2)
month (2)
nike (2)
nokia (2)
npm (2)
programming (2)
raffle (2)
repl (2)
servo (2)
sponsor (2)
steve jobs (2)
travel (2)
typescript (2)
usb (2)
winphone (2)
wwdc (2)
airbender (1)
amdefine (1)
aol (1)
app urls (1)
architecture (1)
atscript (1)
basic income (1)
biology (1)
blink (1)
bluetooth (1)
canada (1)
clip (1)
coding (1)
community (1)
cross-platform (1)
deutsch (1)
diaspora (1)
distributed-social-network (1)
dsl (1)
dvd (1)
dzone (1)
emacs (1)
emberjs (1)
energy (1)
esnext news (1)
esprop (1)
example (1)
facetator (1)
feedback (1)
firefly (1)
firefoxos (1)
fritzbox (1)
german (1)
git (1)
guest (1)
guice (1)

`map()` is used to create an Array of Promises. We are not interested in the results they fulfill, we only `await` until all of them are fulfilled. That means that we are completely done at the end of this async function. We could just as well return `Promise.all()`, but then the result of the function would be an Array whose elements are all `undefined`.

8. Immediately Invoked Async Function Expressions

Sometimes, it'd be nice if you could use `await` at the top level of a module or script. Alas, it's only available inside async functions. You therefore have several options. You can either create an async function `main()` and call it immediately afterwards:

```
async function main() {
    console.log(await asyncFunction());
}
main();
```

Or you can use an Immediately Invoked Async Function Expression:

```
(async function () {
    console.log(await asyncFunction());
})();
```

Another option is an Immediately Invoked Async Arrow Function:

```
(async () => {
    console.log(await asyncFunction());
})();
```

9. Unit testing with async functions

The following code uses the test-framework mocha to unit-test the asynchronous functions `asyncFunc1()` and `asyncFunc2()`:

```
import assert from 'assert';

// Bug: the following test always succeeds
test('Testing async code', function () {
    asyncFunc1() // (A)
    .then(result1 => {
        assert.strictEqual(result1, 'a'); // (B)
        return asyncFunc2();
    })
    .then(result2 => {
        assert.strictEqual(result2, 'b'); // (C)
    });
});
```

However, this test always succeeds, because mocha doesn't wait until the assertions in line (B) and line (C) are executed.

You can fix this by returning the result of the Promise chain, because mocha recognizes if a test returns a Promise and then waits until that Promise is settled (unless there is a timeout).

```
return asyncFunc1() // (A)
```

Conveniently, async functions always return Promises, which makes them perfect for this kind of unit test:

```
import assert from 'assert';
test('Testing async code', async function () {
    const result1 = await asyncFunc1();
    assert.strictEqual(result1, 'a');
    const result2 = await asyncFunc2();
    assert.strictEqual(result2, 'b');
});
```

h.264 (1)
home entertainment (1)
hosting (1)
htc (1)
ical (1)
jsdom (1)
jsmyth (1)
library (1)
location (1)
marketing (1)
mars (1)
meta-data (1)
middle east (1)
mpaa (1)
msl (1)
mssurface (1)
netflix (1)
nsa (1)
obama (1)
openoffice (1)
opinion (1)
oracle (1)
organizing (1)
philosophy (1)
pixar (1)
pnacl (1)
prism (1)
privacy (1)
proxies (1)
puzzle (1)
raspberry pi (1)
read (1)
rodney (1)
rust (1)
safari (1)
sponsoring (1)
star trek (1)
static generation (1)
talk (1)
technique (1)
theora (1)
thunderbird (1)
typography (1)
unicode (1)
v8 (1)
voice control (1)
webassembly (1)
webkit (1)
webm (1)
webpack (1)
yahoo (1)

There are thus two advantages to using async functions for asynchronous unit tests in mocha: the code is more concise and returning Promises is taken care of, too.

10. Don't worry about unhandled rejections

JavaScript engines are becoming increasingly good at warning about rejections that are not handled. For example, the following code would often fail silently in the past, but most modern JavaScript engines now report an unhandled rejection:

```
async function foo() {
    throw new Error('Problem!');
}
foo();
```

16 Comments The 2ality blog 1 Login

Recommend 6 Share Sort by Best

Join the discussion…

Sean • 3 months ago

From everything I see, async / await is essentially the node implementation of co.js, so I assume underneath it uses generators. The only tiny benefit I see with co is that you don't need to define and call your asynchronous functions as co self executes.

https://github.com/tj/co

Are there any benefits to using async / await of co.js?

great article,

regards

Sean

1 • Reply • Share ›

Tracker1 ➔ Sean • 3 months ago

The async-await syntax is cleaner than overloading generator syntax imho, even though it's usually polyfilled with a generator, that doesn't have to be the case. Also, there is an extension for async iterables at stage-2 that will be used with an async generator, which may confuse things for some.

Async seems to be influenced by C#'s async/await as well as co in terms of polyfill/transform. I'm looking forward to seeing it native and not behind a flag in node 8, and in the major browsers by the end of 2017.

• Reply • Share ›

Sean ➔ Tracker1 • 3 months ago

Personally I am waiting for typescript 2.1. supposed to be transparent and fully supported...

• Reply • Share ›

avi tshuva • 3 months ago

You forgot something very important.
It is not just promises involved, but also yield (generator kind of yield). This is the case at least in the emulation of async in typescript.
It makes the whole business a little more complicated, at times...

1 • Reply • Share ›

Axel Rauschmayer Mod ➔ avi tshuva • 3 months ago `yield` is not allowed inside async functions. It is, however, allowed inside async generators: http://www.2ality.com/2016/10/...

1 ^ | ˅ • Reply • Share ›

avi tshuva ➦ Axel Rauschmayer • 3 months ago

Correct, but i didn't explain it correctly: YIELDING is done implicitly inside async, whenever you do await.
It is the same mechanism, ie, "stopping" execution at the yield/await point, getting back to the event loop, and when the time comes, returning to the next code instruction.

That fact, makes it necessary to think about function re-entracy much more often.

1 ^ | ˅ • Reply • Share ›

Alex Mills ➦ avi tshuva • 3 months ago

yupp, agree

1 ^ | ˅ • Reply • Share ›

Alex Mills ➦ Axel Rauschmayer • 3 months ago

await and yield are basically the same exact thing

^ | ˅ • Reply • Share ›

Clem • 10 days ago

for `array.forEach()` loop, can I `await` a void promise (A) to sync?

```
async function logContent(urls) {
  urls.forEach(async url => {
    const content = await httpGet(url);
    console.log(content);
  });
  await Promise.resolve(); // (A)
}
```

^ | ˅ • Reply • Share ›

BigM • 3 months ago

thanks for article.
For web design training please visit: http://www.bigmtraining.com/

^ | ˅ • Reply • Share ›

kungfoobar • 3 months ago

Great post Axel (as always!)

I recently did some work on an alternative approach, resulting in the "wait-for-stuff" module. It relies on "deasync" to do it's magic.

I'm wondering what's your take these kind of solutions as alternatives to "async\await" ?

deasync - https://www.npmjs.com/package/...
wait-for-stuff - https://www.npmjs.com/package/...

^ | ˅ • Reply • Share ›

hemanth.hm • 3 months ago

One of the use cases /me had tried -> https://h3manth.com/new/blog/2...

^ | ˅ • Reply • Share ›

nicosommi • 3 months ago

Good post Axel. It is important to understand that async functions are promises that have been incorporated into the language.

I would like to read your thoughts about current transpilations for async functions. Currently a one-line function on the babel repl turns out into a giant monster.

^ | ˅ • Reply • Share ›

Axel Rauschmayer Mod ➦ nicosommi • 3 months ago

Babel transpiles async functions to generators. Afterwards, it depends how you transpile generators. For example, if you only switch on ES2017 in the Babel REPL, the code looks quite nice: http://babeljs.io/repl/

1 ^ | ∨ • Reply • Share ›

coderitual • 3 months ago

Great article Axel! I'm happy you also explained this wrapping unwrapping process when value is returned. One thing I would add to your examples is another way of execution in parallel.

```
async function foo() {
const result1 = asyncFunc1();
const result2 = asyncFunc2();
await result1;
await result2;
}
```

Cheers!

^ | ∨ • Reply • Share ›

MaxArt • 3 months ago

Oh boy, I foresee quite some headaches to get async/await to work until we get used to it.

Excellent tips about `map` and async functions, though.

^ | ∨ • Reply • Share ›

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)