Async functions - making promises friendly

By Jake Archibald
(https://developers.google.com/web/resources/contributors#jakearchibald)
Human boy working on web standards at Google

Async functions are enabled by default in Chrome 55 and they're quite frankly marvelous. They allow you to write promise-based code as if it were synchronous, but without blocking the main thread. They make your asynchronous code less "clever" and more readable.

Async functions work like this:

```
async function myFirstAsyncFunction() {
  try {
    const fulfilledValue = await promise;
  }
  catch (rejectedValue) {
    // …
  }
}
```

If you use the `async` keyword before a function definition, you can then use `await` within the function. When you `await` a promise, the function is paused in a non-blocking way until the promise settles. If the promise fulfills, you get the value back. If the promise rejects, the rejected value is thrown.

Note: If you're unfamiliar with promises, check out our promises guide (https://developers.google.com/web/fundamentals/getting-started/primers/promises).

Example: Logging a fetch

Say we wanted to fetch a URL and log the response as text. Here's how it looks using promises:

```
function logFetch(url) {
  return fetch(url)
    .then(response => response.text())
    .then(text => {
      console.log(text);
    }).catch(err => {
      console.error('fetch failed', err);
    });
}
```

And here's the same thing using async functions:

```
async function logFetch(url) {
  try {
    const response = await fetch(url);
    console.log(await response.text());
  }
  catch (err) {
    console.log('fetch failed', err);
  }
}
```

It's the same number of lines, but all the callbacks are gone. This makes it way easier to read, especially for those less familiar with promises.

Note: Anything you `await` is passed through `Promise.resolve()`, so you can safely `await` non-native promises.

Async return values

Async functions *always* return a promise, whether you use `await` or not. That promise resolves with whatever the async function returns, or rejects with whatever the async function throws. So with:

```
// wait ms milliseconds
function wait(ms) {
  return new Promise(r => setTimeout(r, ms));
}

async function hello() {
  await wait(500);
  return 'world';
}
```

…calling `hello()` returns a promise that *fulfills* with `"world"`.

```
async function foo() {
  await wait(500);
  throw Error('bar');
}
```

…calling `foo()` returns a promise that *rejects* with `Error('bar')`.

Example: Streaming a response

The benefit of async functions increases in more complex examples. Say we wanted to stream a response while logging out the chunks, and return the final size.

Note: The phrase "logging out the chunks" made me sick in my mouth.

Here it is with promises:

```
function getResponseSize(url) {
  return fetch(url).then(response => {
    const reader = response.body.getReader();
    let total = 0;

    return reader.read().then(function processResult(result) {
      if (result.done) return total;

      const value = result.value;
      total += value.length;
      console.log('Received chunk', value);

      return reader.read().then(processResult);
    })
  });
}
```

Check me out, Jake "wielder of promises" Archibald. See how I'm calling `processResult` inside itself to set up an asynchronous loop? Writing that made me feel *very smart*. But like most "smart" code, you have to stare at it for ages to figure out what it's doing, like one of those magic-eye pictures from the 90's.

Let's try that again with async functions:

```
async function getResponseSize(url) {
  const response = await fetch(url);
  const reader = response.body.getReader();
  let result = await reader.read();
  let total = 0;

  while (!result.done) {
    const value = result.value;
    total += value.length;
    console.log('Received chunk', value);
    // get the next result
    result = await reader.read();
  }

  return total;
}
```

All the "smart" is gone. The asynchronous loop that made me feel so smug is replaced with a trusty, boring, while-loop. Much better. In future, we'll get async iterators (https://github.com/tc39/proposal-async-iteration), which would replace the `while` loop with a for-of loop (https://gist.github.com/jakearchibald/0b37865637daf884943cf88c2cba1376), making it even neater.

Note: I'm sort-of in love with streams. If you're unfamiliar with streaming, check out my guide (https://jakearchibald.com/2016/streams-ftw/#streams-the-fetch-api).

Other async function syntax

We've seen `async function() {}` already, but the `async` keyword can be used with other function syntax:

Arrow functions

```
// map some URLs to json-promises
const jsonPromises = urls.map(async url => {
  const response = await fetch(url);
  return response.json();
});
```

Note: `array.map(func)` doesn't care that I gave it an async function, it just sees it as a function that returns a promise. It won't wait for the first function to complete before calling the second.

Object methods

```
const storage = {
  async getAvatar(name) {
    const cache = await caches.open('avatars');
    return cache.match(`/avatars/${name}.jpg`);
  }
};

storage.getAvatar('jaffathecake').then(…);
```

Class methods

```
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }

  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}

const storage = new Storage();
storage.getAvatar('jaffathecake').then(…);
```

Note: Class constructors and getters/settings cannot be async.

Careful! Avoid going too sequential

Although you're writing code that looks synchronous, ensure you don't miss the opportunity to do things in parallel.

```
async function series() {
  await wait(500); // Wait 500ms…
  await wait(500); // …then wait another 500ms.
  return "done!";
}
```

The above takes 1000ms to complete, whereas:

```
async function parallel() {
  const wait1 = wait(500); // Start a 500ms timer asynchronously…
  const wait2 = wait(500); // …meaning this timer happens in parallel.
  await wait1; // Wait 500ms for the first timer…
  await wait2; // …by which time this timer has already finished.
  return "done!";
}
```

…the above takes 500ms to complete, because both waits happen at the same time. Let's look at a practical example…

Example: Outputting fetches in order

Say we wanted to fetch a series URLs and log them as soon as possible, in the correct order.

Deep breath - here's how that looks with promises:

```
function logInOrder(urls) {
  // fetch all the URLs
  const textPromises = urls.map(url => {
    return fetch(url).then(response => response.text());
  });

  // log them in order
  textPromises.reduce((chain, textPromise) => {
    return chain.then(() => textPromise)
      .then(text => console.log(text));
  }, Promise.resolve());
}
```

Yeah, that's right, I'm using `reduce` to chain a sequence of promises. I'm *so smart*. But this is a bit of *so smart* coding we're better off without.

However, when converting the above to an async function, it's tempting to go *too sequential*:

👎 **Not recommended** - too sequential

```
async function logInOrder(urls) {
  for (const url of urls) {
    const response = await fetch(url);
    console.log(await response.text());
  }
}
```

Looks much neater, but my second fetch doesn't begin until my first fetch has been fully read, and so on. This is much slower than the promises example that performs the fetches in parallel. Thankfully there's an ideal middle-ground:

👍 **Recommended** - nice and parallel

```
async function logInOrder(urls) {
  // fetch all the URLs in parallel
  const textPromises = urls.map(async url => {
    const response = await fetch(url);
    return response.text();
  });

  // log them in sequence
  for (const textPromise of textPromises) {
    console.log(await textPromise);
  }
}
```

In this example, the URLs are fetched and read in parallel, but the "smart" `reduce` bit is replaced with a standard, boring, readable for-loop.

Browser support & workarounds

At time of writing, async functions are enabled by default in Chrome, Edge, Firefox, and Safari.

Workaround - Generators

If you're targeting browsers that support generators (which includes the latest version of every major browser (http://kangax.github.io/compat-table/es6/#test-generators)) you can sort-of polyfill async functions.

Babel (https://babeljs.io/) will do this for you, here's an example via the Babel REPL (https://goo.gl/0Cg1Sq) - note how similar the transpiled code is. This transformation is part of Babel's es2017 preset (http://babeljs.io/docs/plugins/preset-es2017/).

Note: Babel REPL is fun to say. Try it.

I recommend the transpiling approach, because you can just turn it off once your target browsers support async functions, but if you *really* don't want to use a transpiler, you can take Babel's polyfill (https://gist.github.com/jakearchibald/edbc78f73f7df4f7f3182b3c7e522d25) and use it yourself. Instead of:

```
async function slowEcho(val) {
  await wait(1000);
  return val;
}
```

…you'd include the polyfill (https://gist.github.com/jakearchibald/edbc78f73f7df4f7f3182b3c7e522d25) and write:

```
const slowEcho = createAsyncFunction(function*(val) {
  yield wait(1000);
  return val;
});
```

Note that you have to pass a generator (**`function*`**) to **`createAsyncFunction`**, and use **`yield`** instead of **`await`**. Other than that it works the same.

Workaround - regenerator

If you're targeting older browsers, Babel can also transpile generators, allowing you to use async functions all the way down to IE8. To do this you need Babel's es2017 preset (http://babeljs.io/docs/plugins/preset-es2017/) *and* the es2015 preset (http://babeljs.io/docs/plugins/preset-es2015/).

The output is not as pretty (https://goo.gl/jlXboV), so watch out for code-bloat.

Async all the things!

Once async functions land across all browsers, use them on every promise-returning function! Not only do they make your code tider, but it makes sure that function will *always* return a promise.

I got really excited about async functions back in 2014 (https://jakearchibald.com/2014/es7-async-functions/), and it's great to see them land, for real, in browsers. Whoop!

Last updated September 26, 2017.

Chromium Blog
The latest news on the Chromium blog.

GitHub
Fork our code samples and other open-source projects.

Twitter
Connect with @ChromiumDev on Twitter.

Videos
Check out our videos.

Events
Attend a developer event and get hacking.