MDN web docs
moz://a

Search MDN

Technologies ▾

References & Guides ▾

Feedback ▾

Introducing asynchronous JavaScript

English ▾

Previous | Overview: Asynchronous | Next

In this article we briefly recap the problems associated with synchronous JavaScript, and take a first look at some of the different asynchronous techniques you'll encounter, showing how they can help us solve such problems.

Prerequisites:	Basic computer literacy, a reasonable understanding of JavaScript fundamentals.
Objective:	To gain familiarity with what asynchronous JavaScript is, how it differs from synchronous JavaScript, and what use cases it has.

Synchronous JavaScript

To allow us to understand what **asynchronous** JavaScript is, we ought to start off by making sure we understand what **synchronous** JavaScript is. This section recaps some of the information we saw in the previous article.

A lot of the functionality we have looked at in previous learning area modules is synchronous — you run some code, and the result is returned as soon as the browser can do so. Let's look at a simple example (see it live here, and see the source):

```
const btn = document.querySelect
btn.addEventListener('click', ()
  alert('You clicked me!');

  let pElem = document.createEle
  pElem.textContent = 'This is
  document.body.appendChild(pElem);
});
```

Would you answer 4 questions for us? Open the survey in a new tab and fill it out when you are done on the site. Thanks!

In this block, the lines are executed one after the other:

1. We grab a reference to a `<button>` element that is already available in the DOM.
2. We add a `click` event listener to it so that when the button is clicked:
    1. An `alert()` message appears.
    2. Once the alert is dismissed, we create a `<p>` element.
    3. We then give it some text content.
    4. Finally, we append the paragraph to the document body.

While each operation is being processed, nothing else can happen — rendering is paused. This is because as we said in the previous article, JavaScript is single threaded. Only one thing can happen at a time, on a single main thread, and everything else is blocked until an operation completes.

So in the example above, after you've clicked the button the paragraph won't appear until after the OK button is pressed in the alert box. You can try it for yourself:

> **Note**: It is important to remember that `alert()`, while being very useful for demonstrating a synchronous blocking operation, is terrible for use in real world applications.

Asynchronous JavaScript

For reasons illustrated earlier (e.g. related to blockin asynchronous code to run, especially those that acc external device, such as fetching a file from the netw data from it, accessing a video stream from a web cam, or broadcasting the display to a VR headset.

Why is this difficult to get to work using synchronous code? Let's look at a quick example. When you fetch an image from a server, you can't return the result immediately. That means that the following (pseudocode) wouldn't work:

```
let response = fetch('myImage.png');
let blob = response.blob();
// display your image blob in the UI somehow
```

That's because you don't know how long the image will take to download, so when you come to run the second line it will throw an error (possibly intermittently, possibly every time) because the `response` is not yet available. Instead, you need your code to wait until the `response` is returned before it tries to do anything else to it.

There are two main types of asynchronous code style you'll come across in JavaScript code, old-style callbacks and newer promise-style code. In the below sections we'll review each of these in turn.

Async callbacks

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened. Using callbacks is slightly old-fashioned now, but you'll still see them in use in a number of older-but-still-commonly-used APIs.

An example of an async callback is the second parameter of the `addEventListener()` method (as we saw in action above):

```
btn.addEventListener('click', ()
  alert('You clicked me!');

  let pElem = document.createEle
  pElem.textContent = 'This is 
  document.body.appendChild(pEle
});
```

The first parameter is the type of event to be listened for, and the second parameter is a callback function that is invoked when the event is fired.

When we pass a callback function as an argument to another function, we are only passing the function's reference as an argument, i.e, the callback function is **not** executed immediately. It is “called back” (hence the name) asynchronously somewhere inside the containing function’s body. The containing function is responsible for executing the callback function when the time comes.

You can write your own function containing a callback easily enough. Let's look at another example that loads a resource via the `XMLHttpRequest` API (run it live, and see the source):

```
function loadAsset(url, type, callback) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.responseType = type;

  xhr.onload = function() {
    callback(xhr.response);
  };

  xhr.send();
}

function displayImage(blob) {
  let objectURL = URL.createObjectURL(blob);

  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

loadAsset('coffee.jpg', 'blob', displayImage);
```

Here we create a `displayImage()` function that s
object URL, then creates an image to display the UR
`<body>`. However, we then create a `loadAsset()`
parameter, along with a URL to fetch and a content t
abbreviated to "XHR") to fetch the resource at the gi
response to the callback to do something with. In this case the callback is waiting on the XHR call to finish downloading the resource (using the `onload` event handler) before it passes it to the callback.

Callbacks are versatile — not only do they allow you to control the order in which functions are run and what data is passed between them, they also allow you to pass data to different functions depending on circumstance. So you could have different actions to run on the response downloaded, such as `processJSON()`, `displayText()`, etc.

Note that not all callbacks are async — some run synchronously. An example is when we use `Array.prototype.forEach()` to loop through the items in an array (see it live, and the source):

```
const gods = ['Apollo', 'Artemis', 'Ares', 'Zeus'];

gods.forEach(function (eachName, index){
  console.log(index + '. ' + eachName);
});
```

In this example we loop through an array of Greek gods and print the index numbers and values to the console. The expected parameter of `forEach()` is a callback function, which itself takes two parameters, a reference to the array name and index values. However, it doesn't wait for anything — it runs immediately.

Promises

Promises are the new style of async code that you'll see used in modern Web APIs. A good example is the `fetch()` API, which is basically like a modern, more efficient version of `XMLHttpRequest`. Let's look at a quick example, from our Fetching data from the server article:

```
fetch('products.json').then(func
  return response.json();
}).then(function(json) {
  products = json;
  initialize();
}).catch(function(err) {
  console.log('Fetch problem: ' + err.message);
});
```

> **Note**: You can find the finished version on GitHub (see the source here, and also see it running live).

Here we see `fetch()` taking a single parameter — the URL of a resource you want to fetch from the network — and returning a promise. The promise is an object representing the completion or failure of the async operation. It represents an intermediate state, as it were. In essence, it's the browser's way of saying "I promise to get back to you with the answer as soon as I can," hence the name "promise."

This concept can take practice to get used to; it feels a little like Schrödinger's cat in action. Neither of the possible outcomes have happened yet, so the fetch operation is currently waiting on the result of the browser trying to complete the operation at some point in the future. We've then got three further code blocks chained onto the end of the `fetch()`:

- Two `then()` blocks. Both contain a callback function that will run if the previous operation is successful, and each callback receives as input the result of the previous successful operation, so you can go forward and do something else to it. Each `.then()` block returns another promise, meaning that you can chain multiple `.then()` blocks onto each other, so multiple asynchronous operations can be made to run in order, one after another.
- The `catch()` block at the end runs if any of the `.then()` blocks fail — in a similar way to synchronous `try...catch` blocks, an error object is made available inside the `catch()`, which can be used to report the kind of error that has occurred. Note however that synchronous `try...catch` won't work with promises, although it will work with async/await, as you'll learn later on.

> **Note**: You'll learn a lot more about promises later on in the module, so don't worry if you don't understand them fully yet.

The event queue

Async operations like promises are put into an **even** has finished processing so that they *do not block* su The queued operations will complete as soon as pos JavaScript environment.

Promises versus callbacks

Promises have some similarities to old-style callbacks. They are essentially a returned object to which you attach callback functions, rather than having to pass callbacks into a function.

However, promises are specifically made for handling async operations, and have many advantages over old-style callbacks:

- You can chain multiple async operations together using multiple `.then()` operations, passing the result of one into the next one as an input. This is much harder to do with callbacks, which often ends up with a messy "pyramid of doom" (also known as callback hell).
- Promise callbacks are always called in the strict order they are placed in the event queue.
- Error handling is much better — all errors are handled by a single `.catch()` block at the end of the block, rather than being individually handled in each level of the "pyramid".
- Promises avoid inversion of control, unlike old-style callbacks, which lose full control of how the function will be executed when passing a callback to a third-party library.

The nature of asynchronous code

Let's explore an example that further illustrates the nature of async code, showing what can happen when we are not fully aware of code execution order and the problems of trying to treat asynchronous code like synchronous code. The following example is fairly similar to what we've seen before (see it live, and the source). One difference is that we've included a number of `console.log()` statements to illustrate an order that you might think the code would execute in.

```
console.log ('Starting');
let image;
```

```

fetch('coffee.jpg').then((respor
  console.log('It worked :)')
  return response.blob();
}).then((myBlob) => {
  let objectURL = URL.createObje
  image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}).catch((error) => {
  console.log('There has been a problem with your fetch operation:
});

console.log ('All done!');
```

The browser will begin executing the code, see the first `console.log()` statement (`Starting`) and execute it, and then create the `image` variable.

It will then move to the next line and begin executing the `fetch()` block but, because `fetch()` executes asynchronously without blocking, code execution continues after the promise-related code, thereby reaching the final `console.log()` statement (`All done!`) and outputting it to the console.

Only once the `fetch()` block has completely finished running and delivering its result through the `.then()` blocks will we finally see the second `console.log()` message (`It worked :)`) appear. So the messages have appeared in a different order to what you might expect:

- Starting
- All done!
- It worked :)

If this confuses you, then consider the following smaller example:

```
console.log("registering click handler");

button.addEventListener('click', () => {
  console.log("get click");
});

console.log("all done");
```

This is very similar in behavior — the first and third c
immediately, but the second one is blocked from run
button. The previous example works in the same wa
message is blocked on the promise chain fetching a
rather than a click.

In a less trivial code example, this kind of setup could cause a problem — you can't include an async code block that returns a result, which you then rely on later in a sync code block. You just can't guarantee that the async function will return before the browser has processed the sync block.

To see this in action, try taking a local copy of our example, and changing the third `console.log()` call to the following:

```
console.log ('All done! ' + image.src + 'displayed.');
```

You should now get an error in your console instead of the third message:

```
TypeError: image is undefined; can't access its "src" property
```

This is because at the time the browser tries to run the third `console.log()` statement, the `fetch()` block has not finished running so the `image` variable has not been given a value.

> **Note**: For security reasons, you can't `fetch()` files from your local filesystem (or run other such operations locally); to run the above example locally you'll have to run the example through a local webserver.

Active learning: make it all async!

To fix the problematic `fetch()` example and make the three `console.log()` statements appear in the desired order, you could make the third `console.log()` statement run async as well. This can be done by moving it inside another `.then()` block chained onto the end of the second one, or by simply moving it inside the second `then()` block. Try fixing this now.

> **Note**: If you get stuck, you can find an answer he
> find a lot more information on promises in our Gr
> Promises guide, later on in the module.

Would you answer 4 questions for us? Open the survey in a new tab and fill it out when you are done on the site. Thanks!

✖

Conclusion

In its most basic form, JavaScript is a synchronous, blocking, single-threaded language, in which only one operation can be in progress at a time. But web browsers define functions and APIs that allow us to register functions that should not be executed synchronously, and should instead be invoked asynchronously when some kind of event occurs (the passage of time, the user's interaction with the mouse, or the arrival of data over the network, for example). This means that you can let your code do several things at the same time without stopping or blocking your main thread.

Whether we want to run code synchronously or asynchronously will depend on what we're trying to do.

There are times when we want things to load and happen right away. For example when applying some user-defined styles to a webpage you'll want the styles to be applied as soon as possible.

If we're running an operation that takes time however, like querying a database and using the results to populate templates, it is better to push this off the main thread and complete the task asynchronously. Over time, you'll learn when it makes more sense to choose an asynchronous technique over a synchronous one.

Previous   Overview: Asynchronous   Next

In this module

- General asynchronous programming concepts
- Introducing asynchronous JavaScript
- Cooperative asynchronous JavaScript: Timeouts and intervals

- Graceful asynchronous programming with Promises
- Making asynchronous programming easier wit
- Choosing the right approach

Last modified: Nov 22, 2019, by MDN contributors

Related Topics

Complete beginners start here!

▶ Getting started with the Web

HTML — Structuring the Web

▶ Introduction to HTML

▶ Multimedia and embedding

▶ HTML tables

▶ HTML forms

CSS — Styling the Web

▶ CSS first steps

▶ CSS building blocks

▶ Styling text

▶ CSS layout

JavaScript — Dynamic client-side scripting

▶ JavaScript first steps

▶ JavaScript building blocks

▶ Introducing JavaScript objects

▼ Asynchronous JavaScript

Asynchronous JavaScript overview

General asynchronous programming concepts

Introducing asynchronous JavaScript

Cooperative asynchronous JavaScript: Timeouts and intervals

Graceful asynchronous programming with Promises

Making asynchronous programming easier with async and await

Choosing the right approach

▶ Client-side web APIs

Accessibility — Make the web usable by everyone

▶ Accessibility guides

▶ Accessibility assessment

Tools and testing

▶ Cross browser testing

Server-side website programming

▶ First steps

▶ Django web framework (Python)

▶ Express Web Framework (node.js/JavaScript)

Further resources

▶ Common questions

How to contribute

Would you answer 4 questions for us? Open the survey in a new tab and fill it out when you are done on the site. Thanks!

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

you@example.com

Sign up now