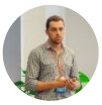
How JavaScript works: Event loop and the rise of Async programming + 5 ways to better coding with async/await



Alexander Zlatkov
Oct 11, 2017 · 17 min read

Welcome to post # 4 of the series dedicated to exploring JavaScript and its building components. In the process of identifying and describing the core elements, we also share some rules of thumb we use when building SessionStack, a JavaScript application that has to be robust and highly-performant in order to stay competitive.

Did you miss the first three chapters? You can find them here:

1. An overview of the engine, the runtime, and the call stack
2. Inside Google's V8 engine + 5 tips on how to write optimized code
3. Memory management + how to handle 4 common memory leaks

This time we'll expand on our first post by reviewing the drawbacks to programming in a single-threaded environment and how to overcome them to build stunning JavaScript UIs. As the tradition goes, at the end of the article we'll share 5 tips on how to write cleaner code with async/await.

Why having a single thread is a limitation?

In the first post we launched, we pondered over the question *what happens when you have function calls in the Call Stack that take a huge amount of time to be processed*.

Imagine, for example, a complex image transformation algorithm that's running in the browser.

While the Call Stack has functions to execute, the browser can't do anything else — it's being blocked. This means that the browser can't render, it can't run any other code, it's just stuck. And here comes the problem — your app UI is no longer efficient and pleasing.

Your app **is stuck**.

In some cases, this might not be such a critical issue. But hey — here's an even bigger problem. Once your browser starts processing too many tasks in the Call Stack, it may stop being responsive for a long time. At that point, a lot of browsers would take action by raising an error, asking whether they should terminate the page:

It's ugly, and it completely ruins your UX:

The building blocks of a JavaScript program

You may be writing your JavaScript application in a single .js file, but your program is almost certainly comprised of several blocks, only one of which is going to execute *now*, and the rest will execute *later*. The most common block unit is the function.

The problem most developers new to JavaScript seem to have is understanding that *later* doesn't necessarily happen strictly and immediately after *now*. In other words, tasks that cannot complete *now* are, by definition, going to complete asynchronously, which means you won't have the above-mentioned blocking behavior as you might have subconsciously expected or hoped for.

Let's take a look at the following example:

```
// ajax(..) is some arbitrary Ajax function given by a library
var response = ajax('https://example.com/api');

console.log(response);
// `response` won't have the response
```

sample1.js hosted with ❤ by GitHub view raw

You're probably aware that standard Ajax requests don't complete synchronously, which means that at the time of code execution the ajax(..) function does not yet have any value to return back to be assigned to a response variable.

A simple way of "waiting" for an asynchronous function to return its result is to use a function called **callback:**

```
ajax('https://example.com/api', function(response) {
    console.log(response); // `response` is now available
});
```

sample2.js hosted with ❤ by GitHub view raw

Just a note: you can actually make **synchronous** Ajax requests. Never, ever do that. If you make a synchronous Ajax request, the UI of your JavaScript app will be blocked — the user won't be able to click, enter data, navigate, or scroll. This would prevent any user interaction. It's a terrible practice.

This is how it looks like, but please, never do this — don't ruin the web:

```
// This is assuming that you're using jQuery
jQuery.ajax({
    url: 'https://api.example.com/endpoint',
    success: function(response) {
        // This is your callback.
    },
    async: false // And this is a terrible idea
});
```

sample3.js hosted with ❤ by GitHub view raw

We used an Ajax request just as an example. You can have any chunk of code execute asynchronously.

This can be done with the `setTimeout(callback, milliseconds)` function. What the `setTimeout` function does is to set up an event (a timeout) to happen later. Let's take a look:

```
function first() {
    console.log('first');
}
function second() {
    console.log('second');
}
function third() {
    console.log('third');
}
first();
setTimeout(second, 1000); // Invoke `second` after 1000ms
third();
```

sample4.js hosted with ❤️ by GitHub view raw

The output in the console will be the following:

```
first
third
second
```

Dissecting the Event Loop

We'll start with a somewhat of an odd claim — despite allowing async JavaScript code (like the `setTimeout` we just discussed), until ES6, JavaScript itself has actually never had any direct notion of asynchrony built into it. The JavaScript engine has never done anything more than executing a single chunk of your program at any given moment.

For more details on how JavaScript engines work (Google's V8 specifically), check one of our previous articles on the topic.

So, who tells the JS Engine to execute chunks of your program? In reality, the JS Engine doesn't run in isolation — it runs inside a *hosting* environment, which for most developers is the typical web browser or Node.js. Actually, nowadays, JavaScript gets embedded into all kinds of devices, from robots to light bulbs. Every single device represents a different type of hosting environment for the JS Engine.

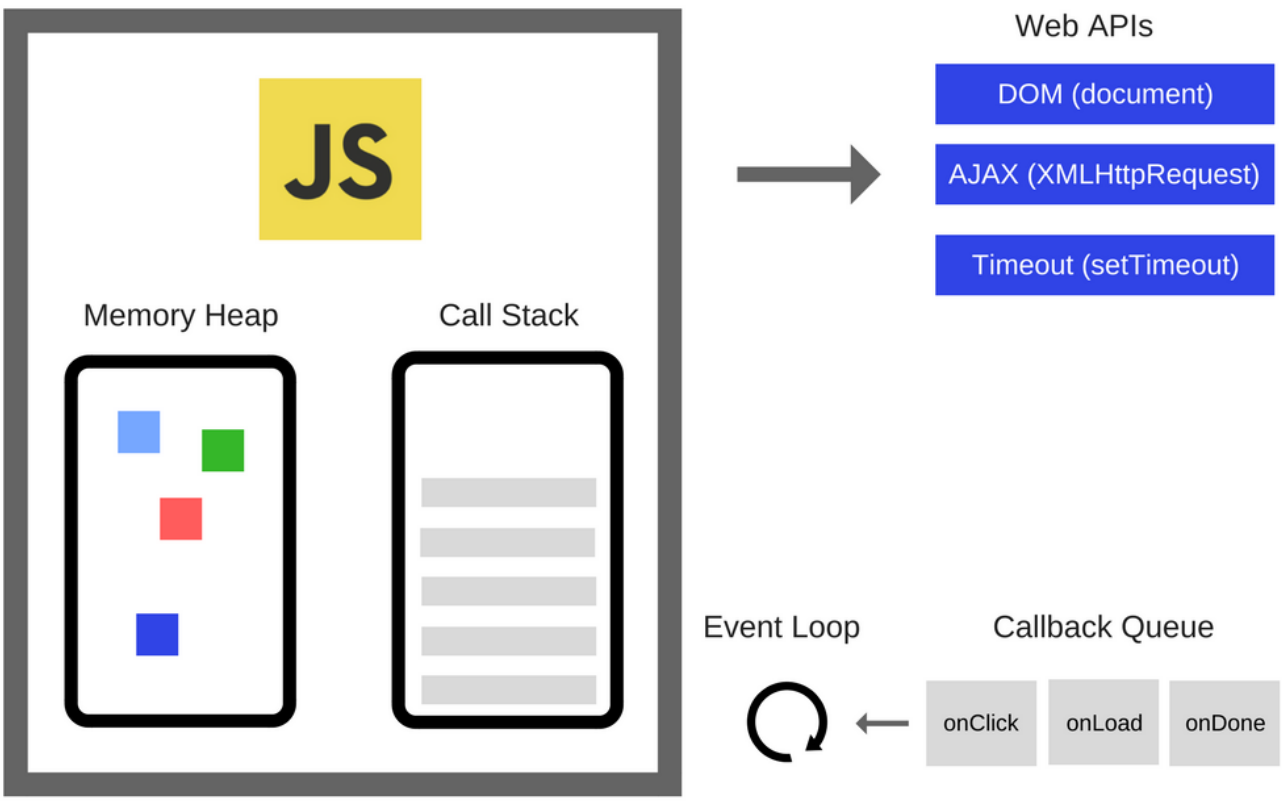
The common denominator in all environments is a built-in mechanism called the **event loop,** which handles the execution of multiple chunks of your program over time, each time invoking the JS Engine.

This means that the JS Engine is just an on-demand execution environment for any arbitrary JS code. It's the surrounding environment that schedules the events (the JS code executions).

So, for example, when your JavaScript program makes an Ajax request to fetch some data from the server, you set up the "response" code in a function (the "callback"), and the JS Engine tells the hosting environment:
"Hey, I'm going to suspend execution for now, but whenever you finish with that network request, and you have some data, please *call* this function *back*."

The browser is then set up to listen for the response from the network, and when it has something to return to you, it will schedule the callback function to be executed by inserting it into the *event loop*.

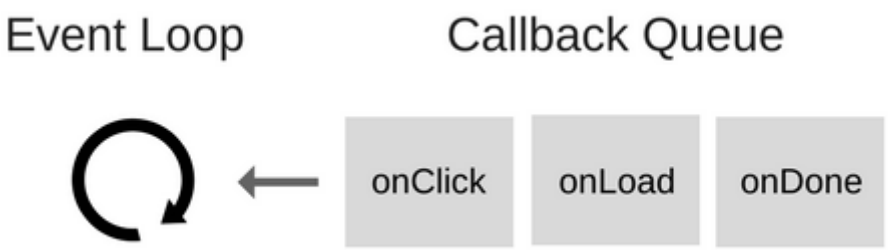
Let's look at the below diagram:



You can read more about the Memory Heap and the Call Stack in our previous article.

And what are these Web APIs? In essence, they are threads that you can't access, you can just make calls to them. They are the pieces of the browser in which concurrency kicks in. If you're a Node.js developer, these are the C++ APIs.

So what is the *event loop after all*?



The Event Loop has one simple job — to monitor the Call Stack and the Callback Queue. If the Call Stack is empty, it will take the first event from the queue and will push it to the Call Stack, which effectively runs it.

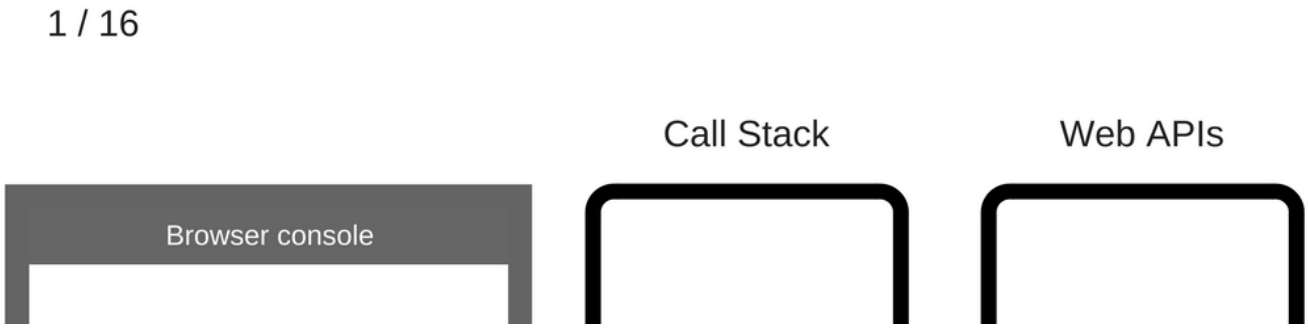
Such an iteration is called a **tick** in the Event Loop. Each event is just a function callback.

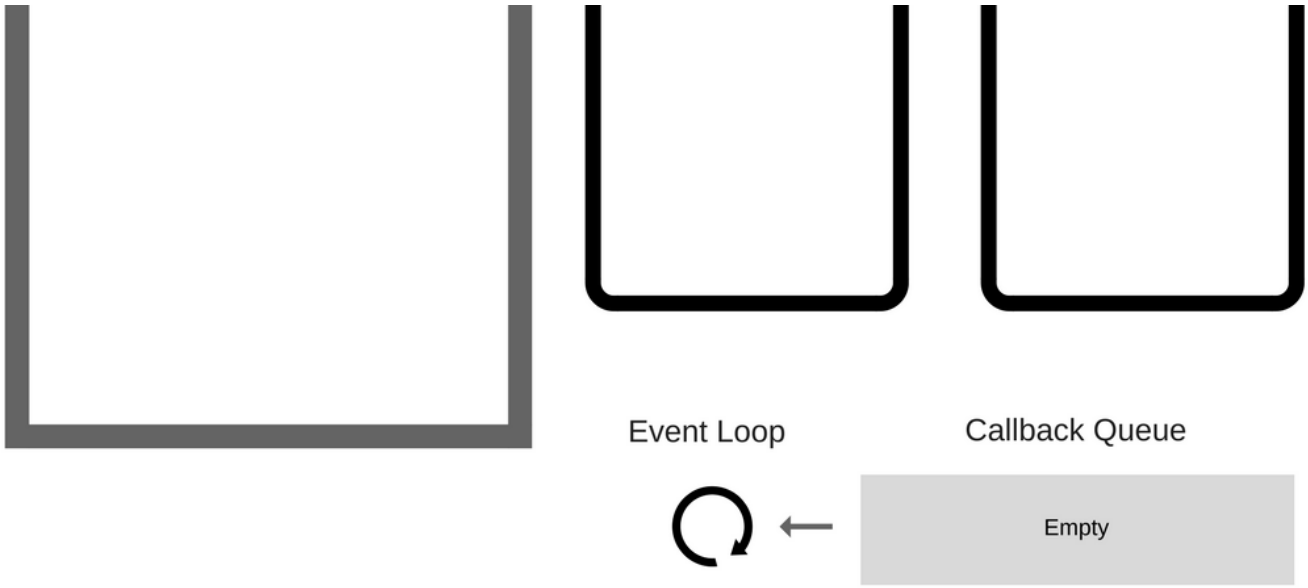
```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

sample5.js hosted with ❤ by GitHub view raw

Let's "execute" this code and see what happens:

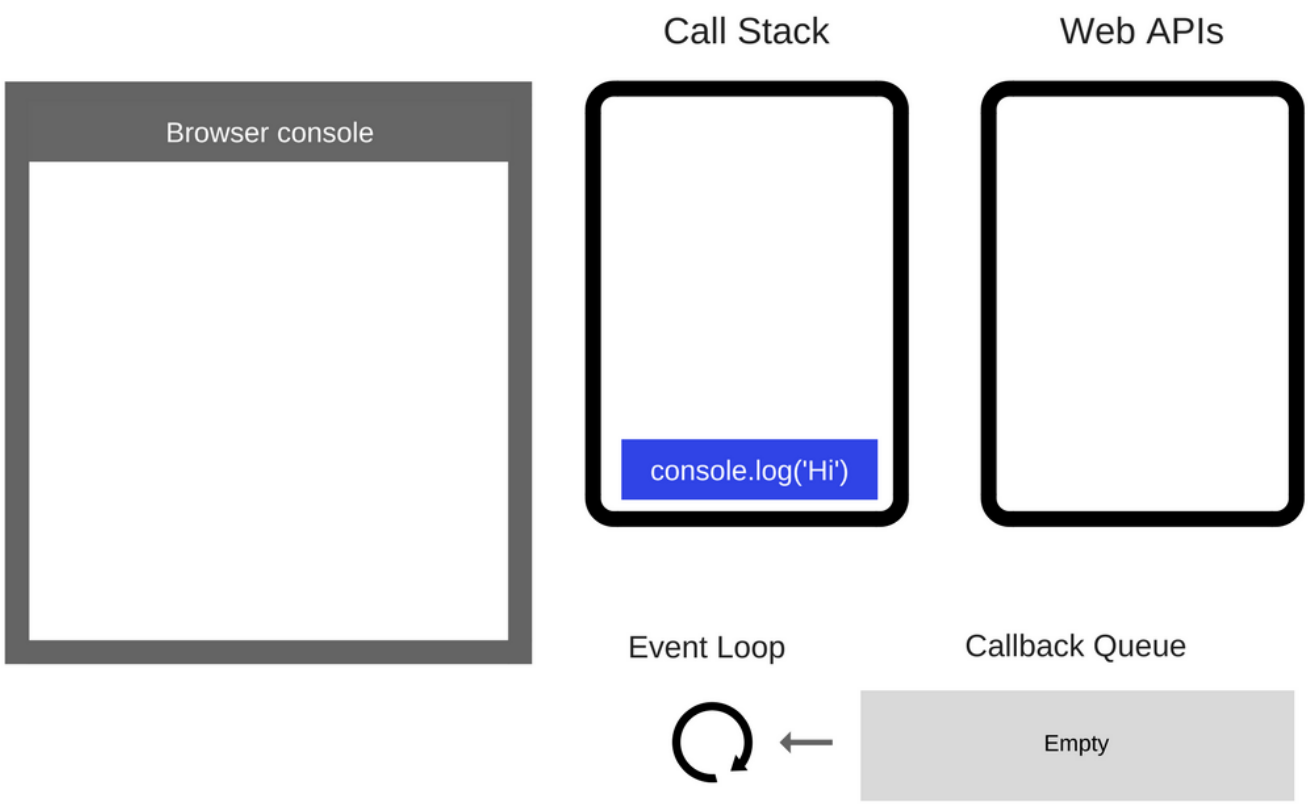
1. The state is clear. The browser console is clear, and the Call Stack is empty.





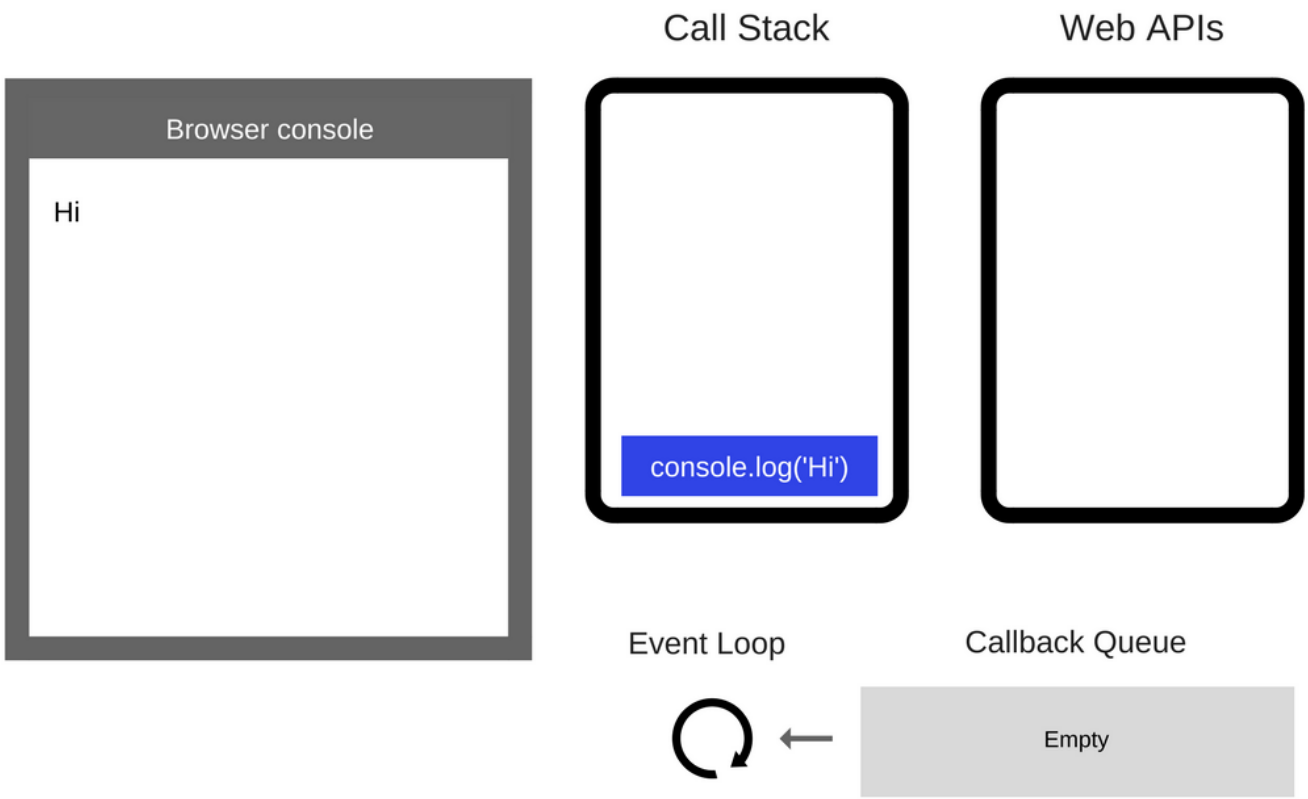
2. `console.log('Hi')` is added to the Call Stack.

2 / 16



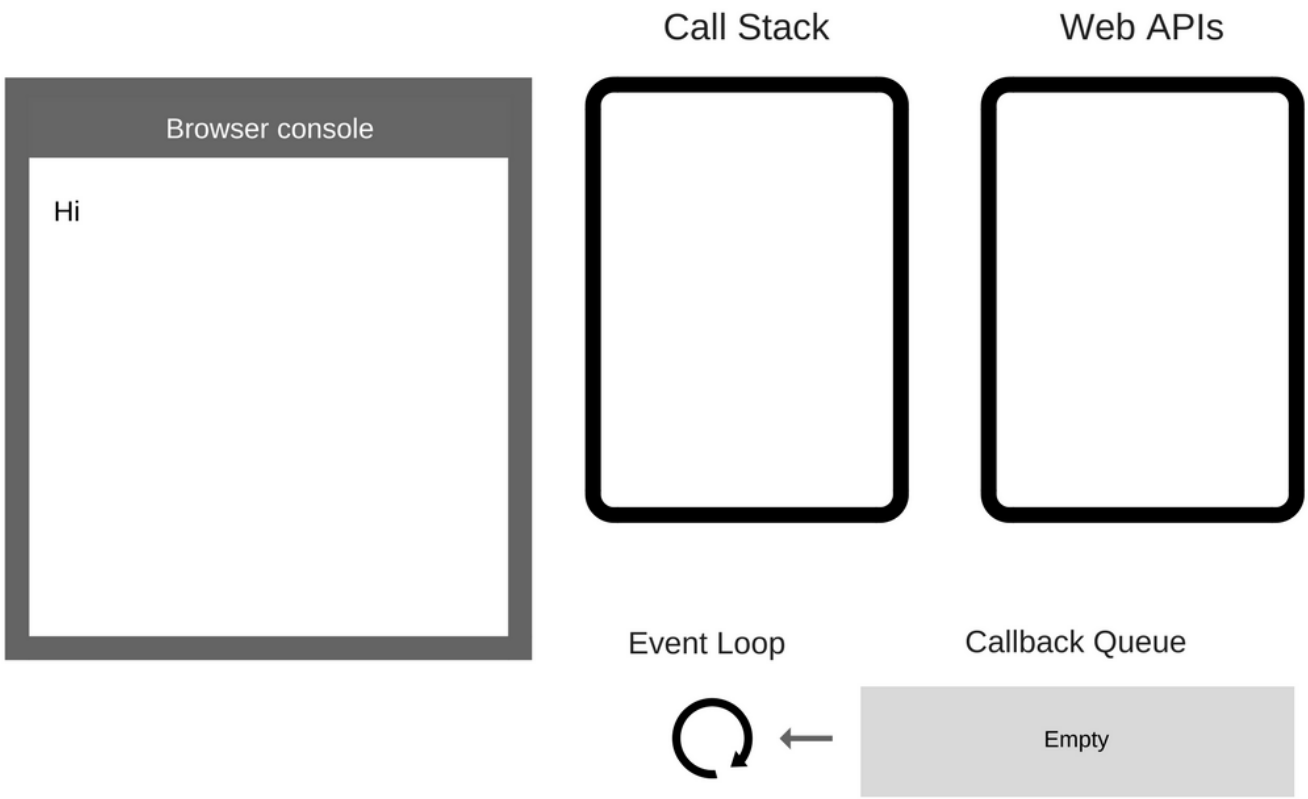
3. `console.log('Hi')` is executed.

3 / 16



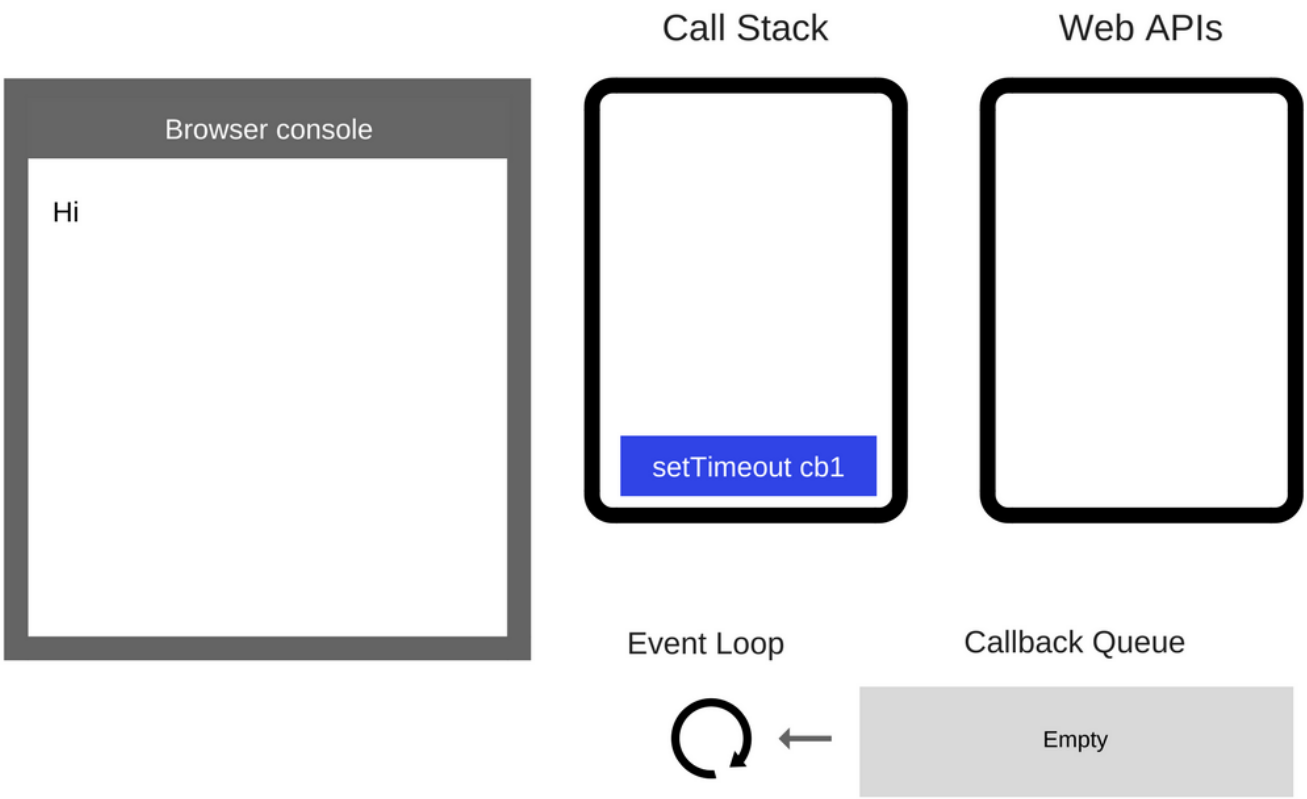
4. `console.log('Hi')` is removed from the Call Stack.

4 / 16



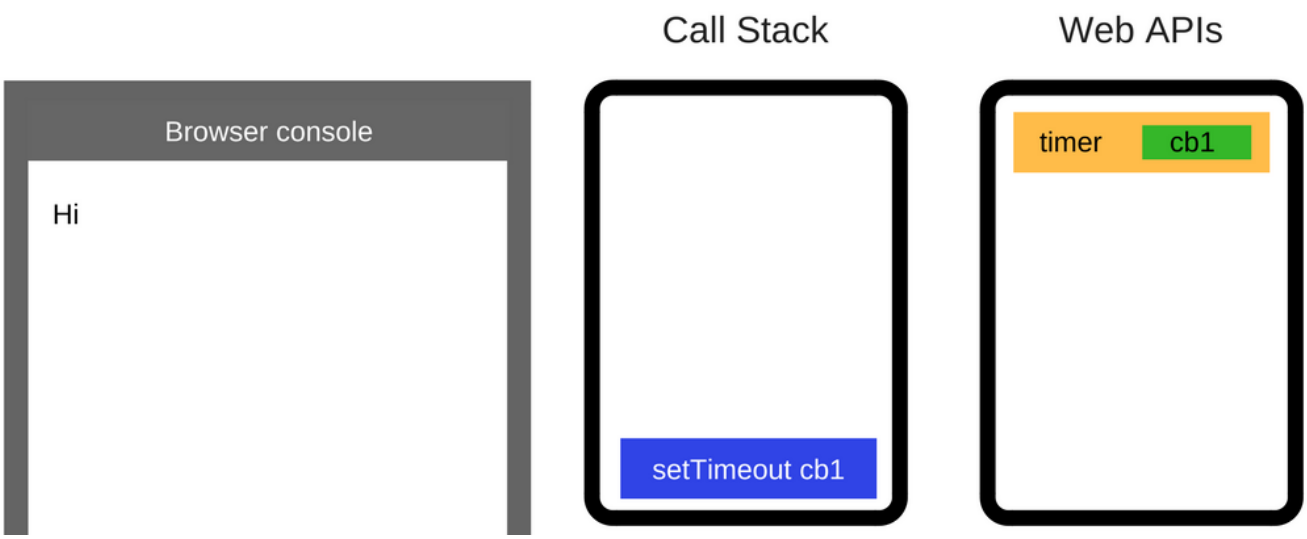
5. `setTimeout(function cb1() { ... })` is added to the Call Stack.

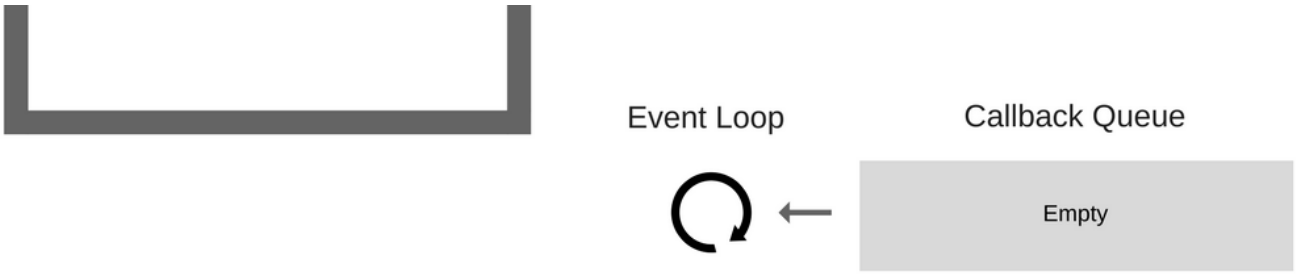
5 / 16



6. `setTimeout(function cb1() { ... })` is executed. The browser creates a timer as part of the Web APIs. It is going to handle the countdown for you.

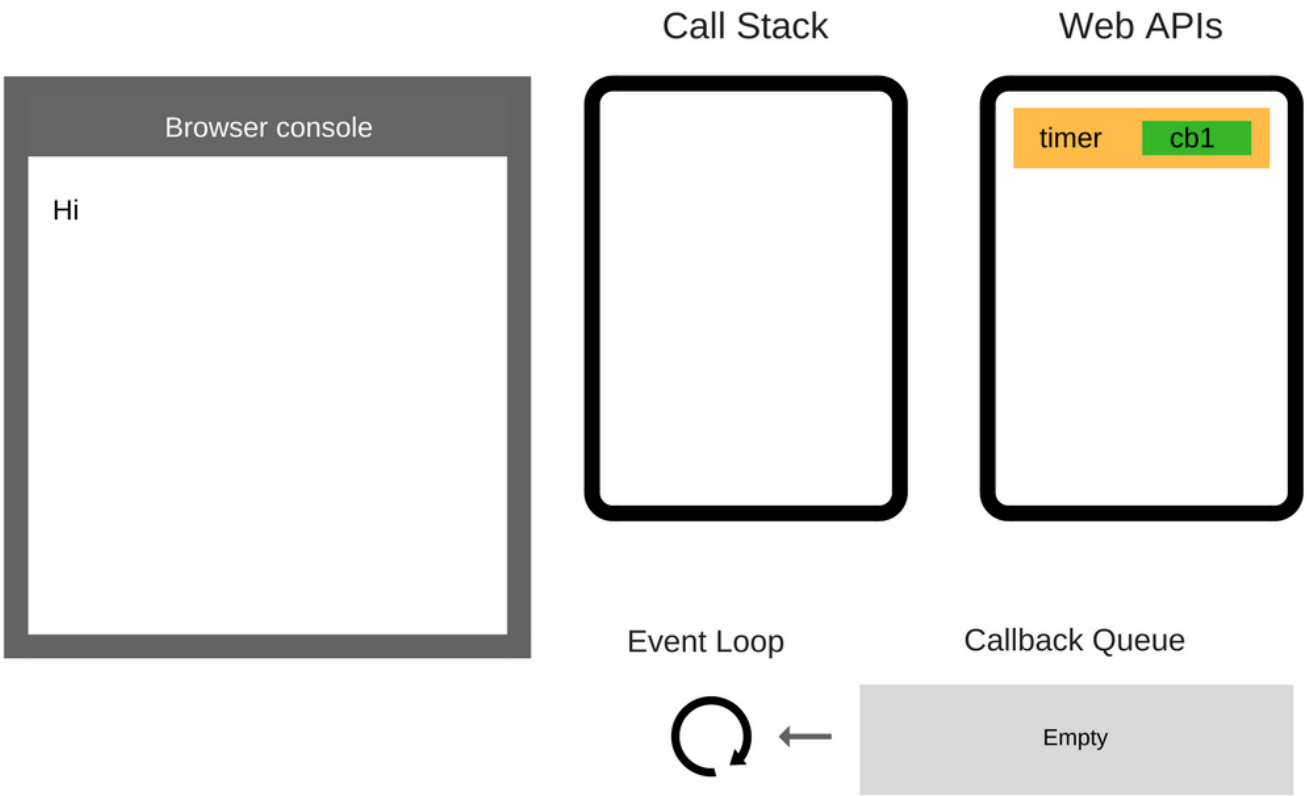
6 / 16





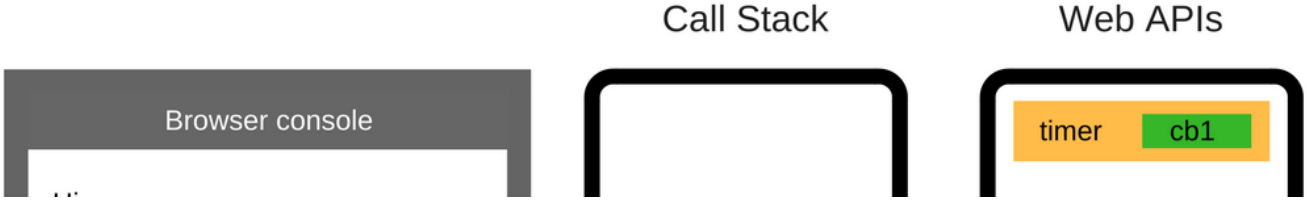
7. The `setTimeout(function cb1() { ... })` itself is complete and is removed from the Call Stack.

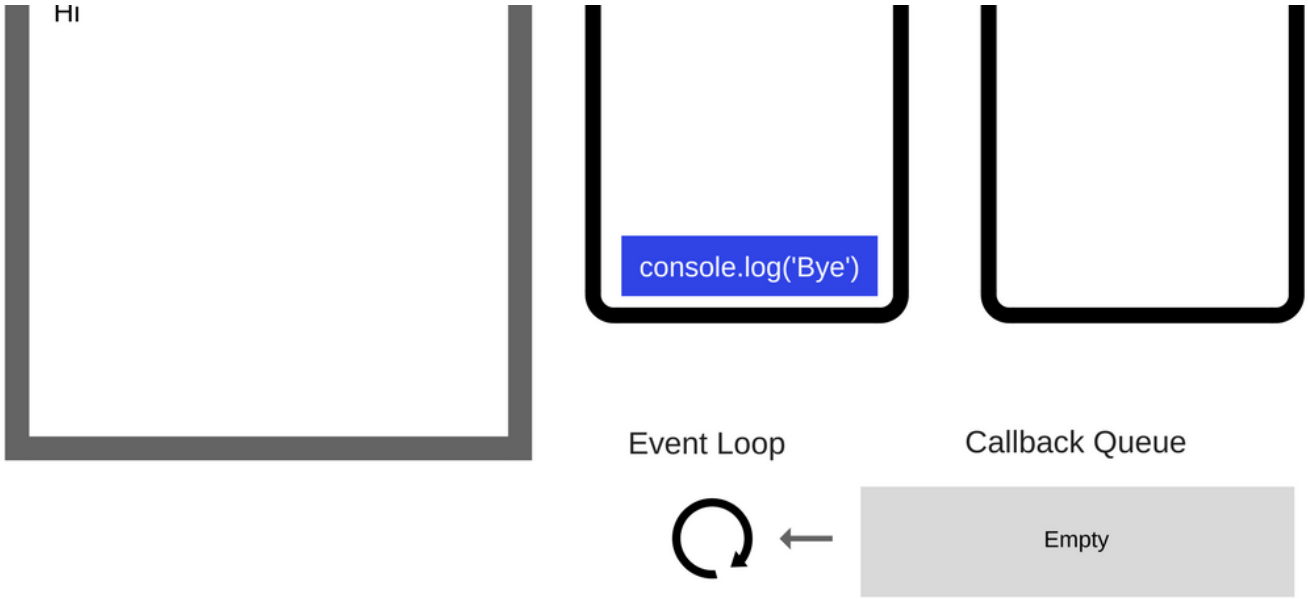
7 / 16



8. `console.log('Bye')` is added to the Call Stack.

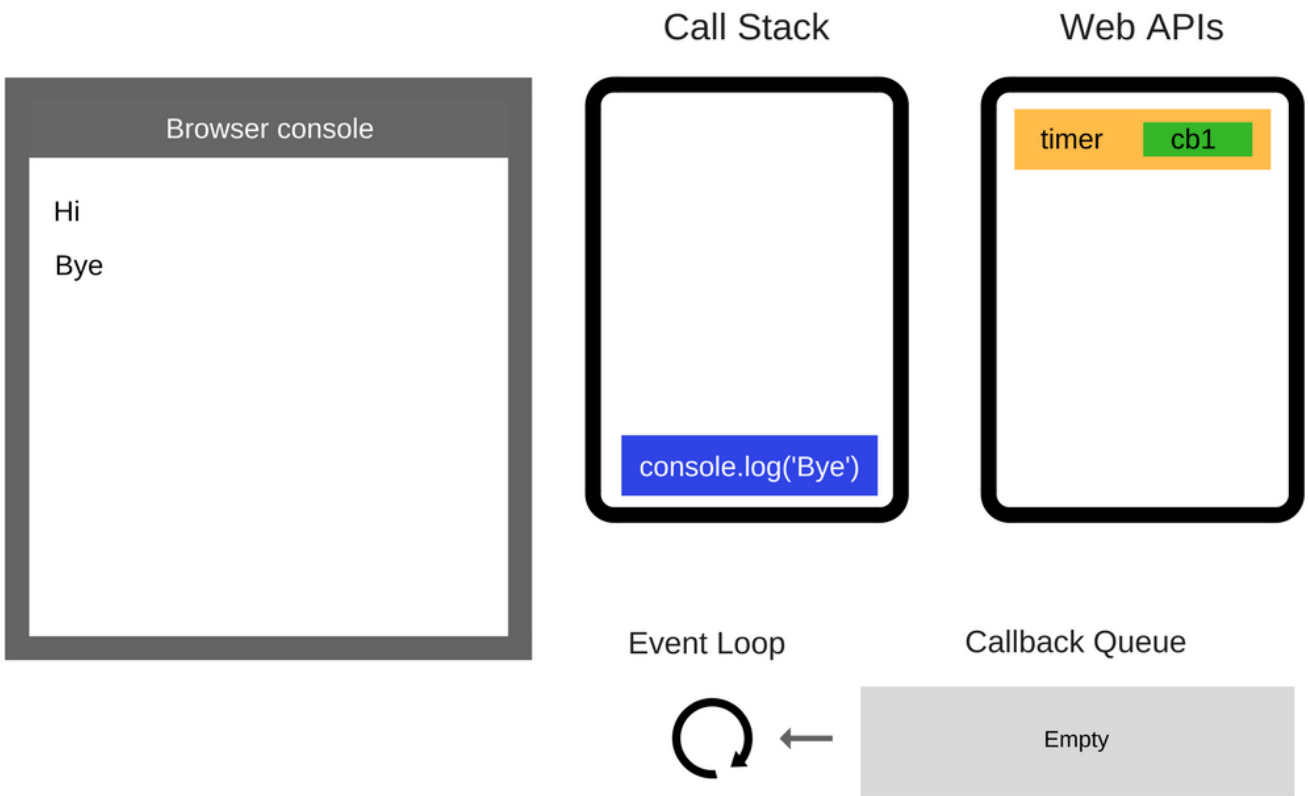
8 / 16





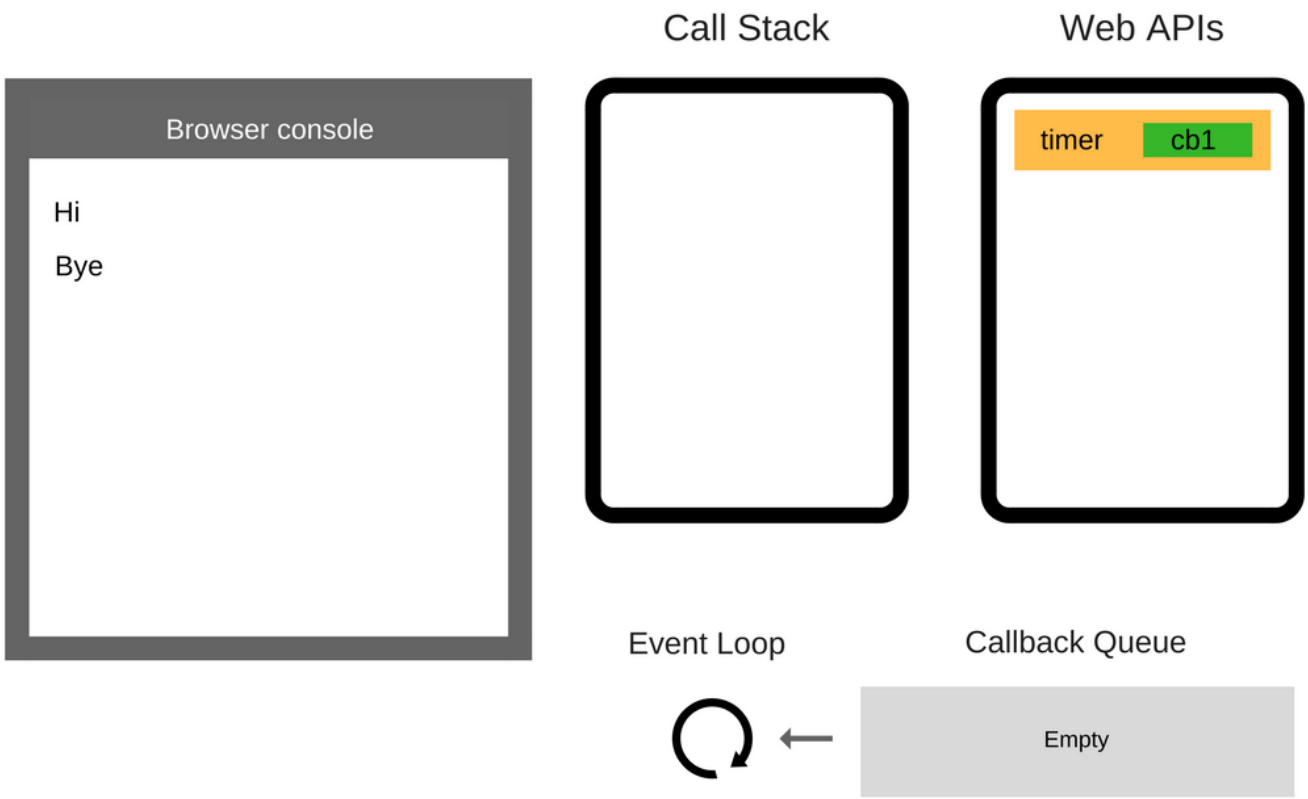
9. `console.log('Bye')` is executed.

9 / 16



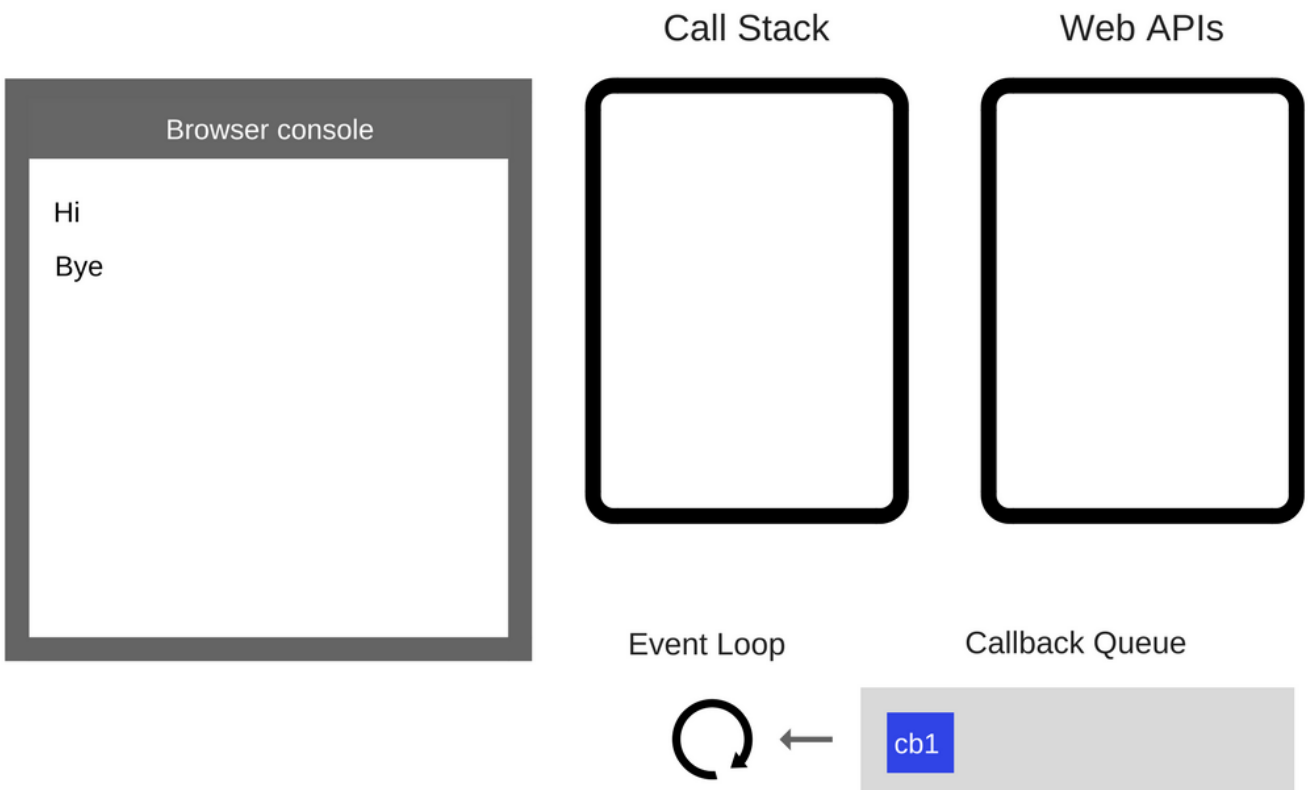
10. `console.log('Bye')` is removed from the Call Stack.

10 / 16



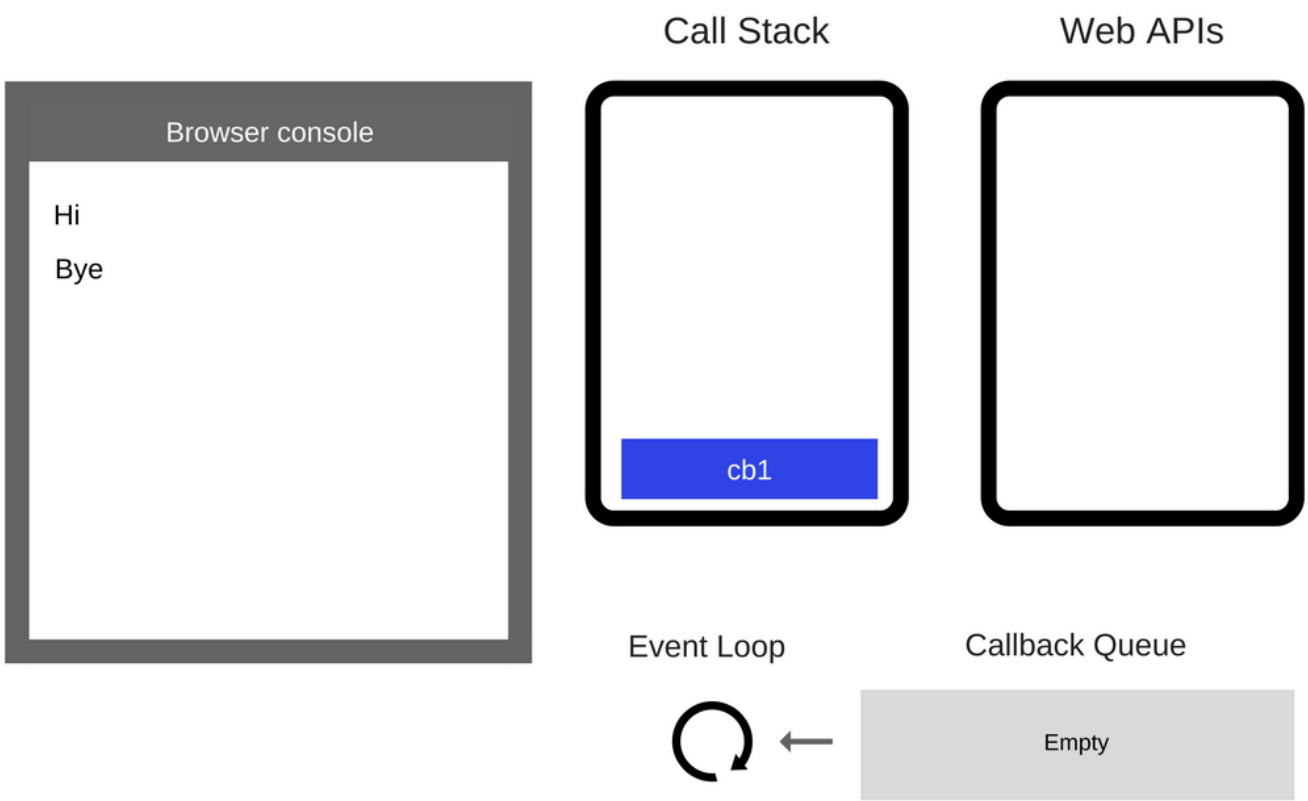
11. After at least 5000 ms, the timer completes and it pushes the `cb1` callback to the Callback Queue.

11 / 16



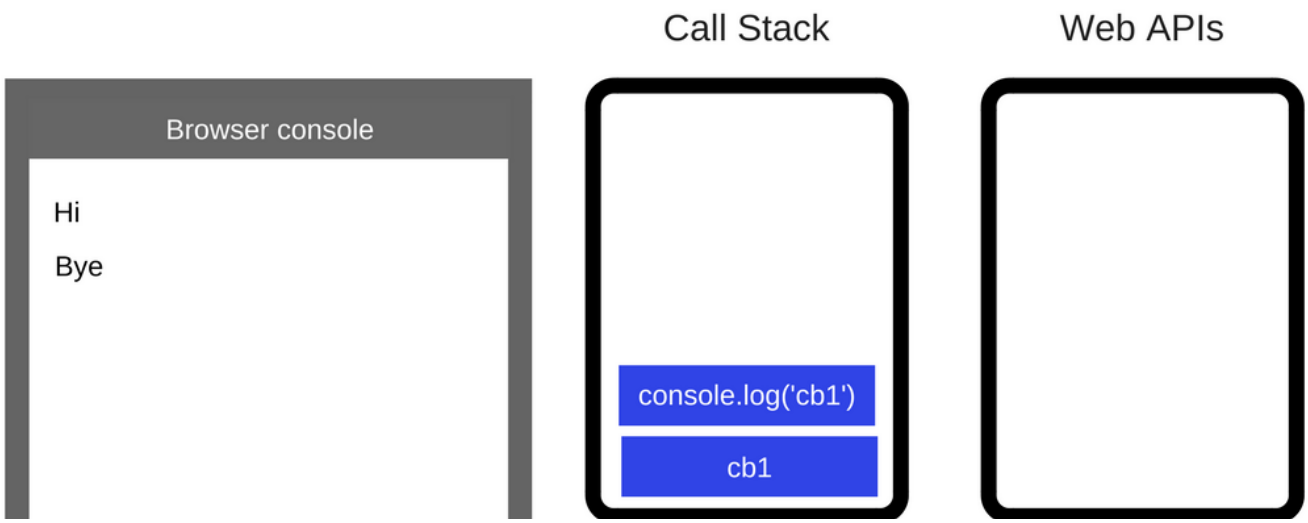
12. The Event Loop takes `cb1` from the Callback Queue and pushes it to the Call Stack.

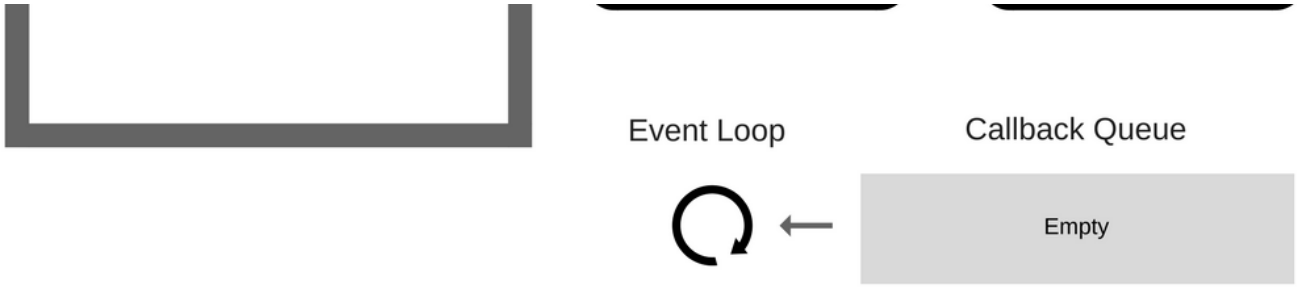
12 / 16



13. `cb1` is executed and adds `console.log('cb1')` to the Call Stack.

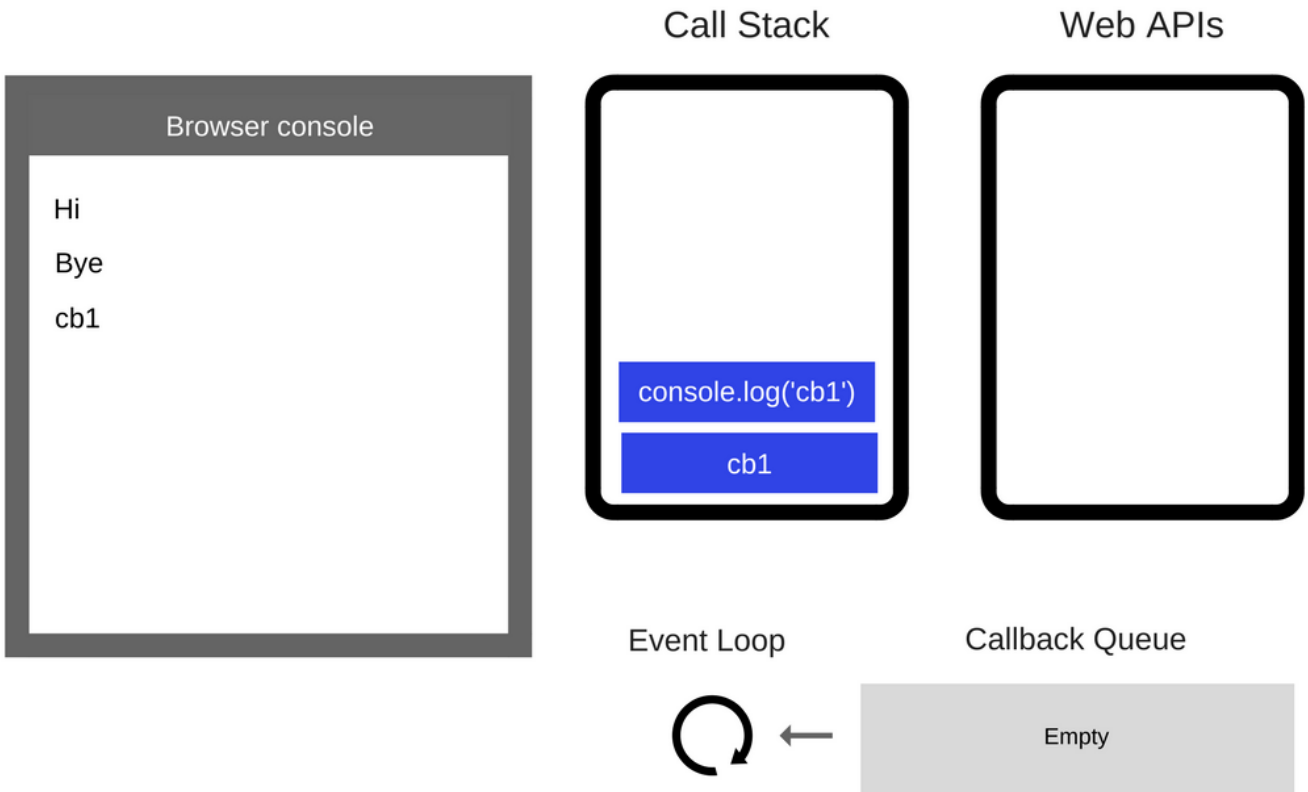
13 / 16





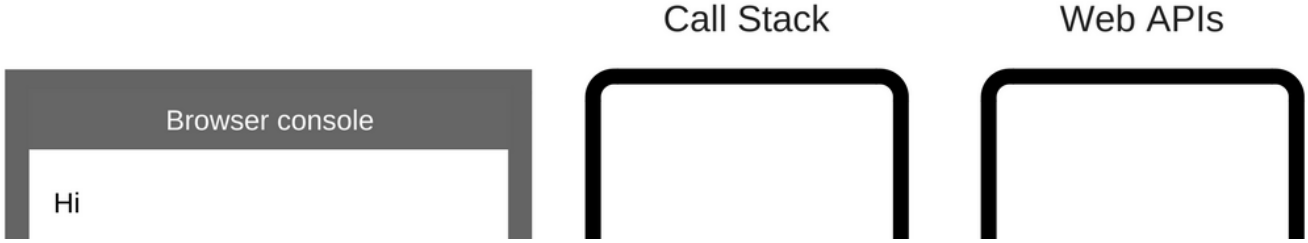
14. `console.log('cb1')` is executed.

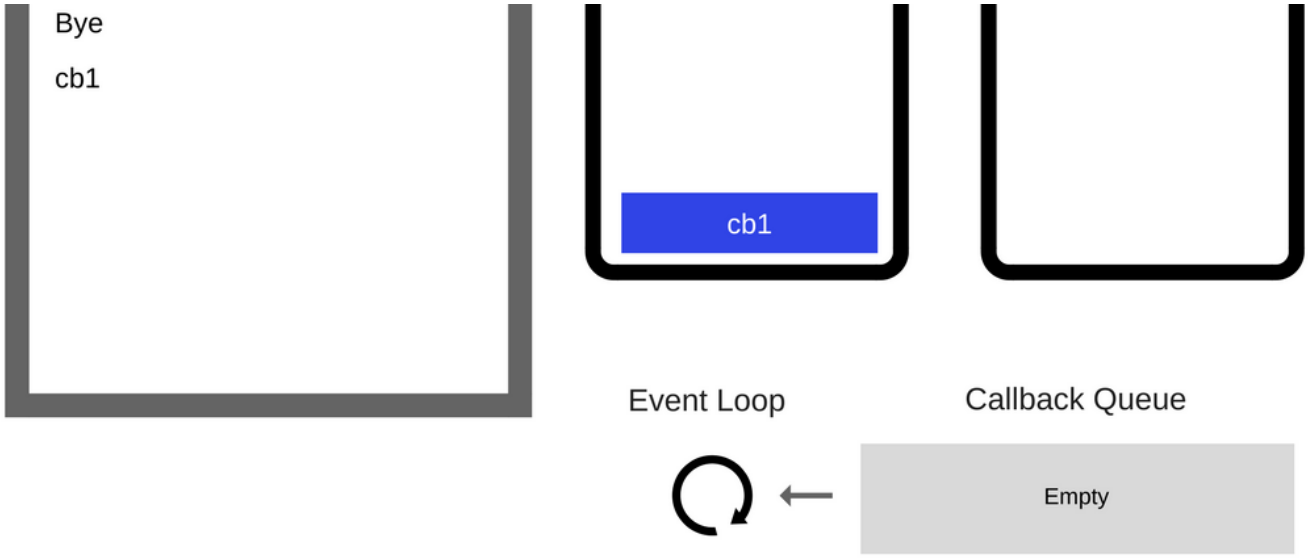
14 / 16



15. `console.log('cb1')` is removed from the Call Stack.

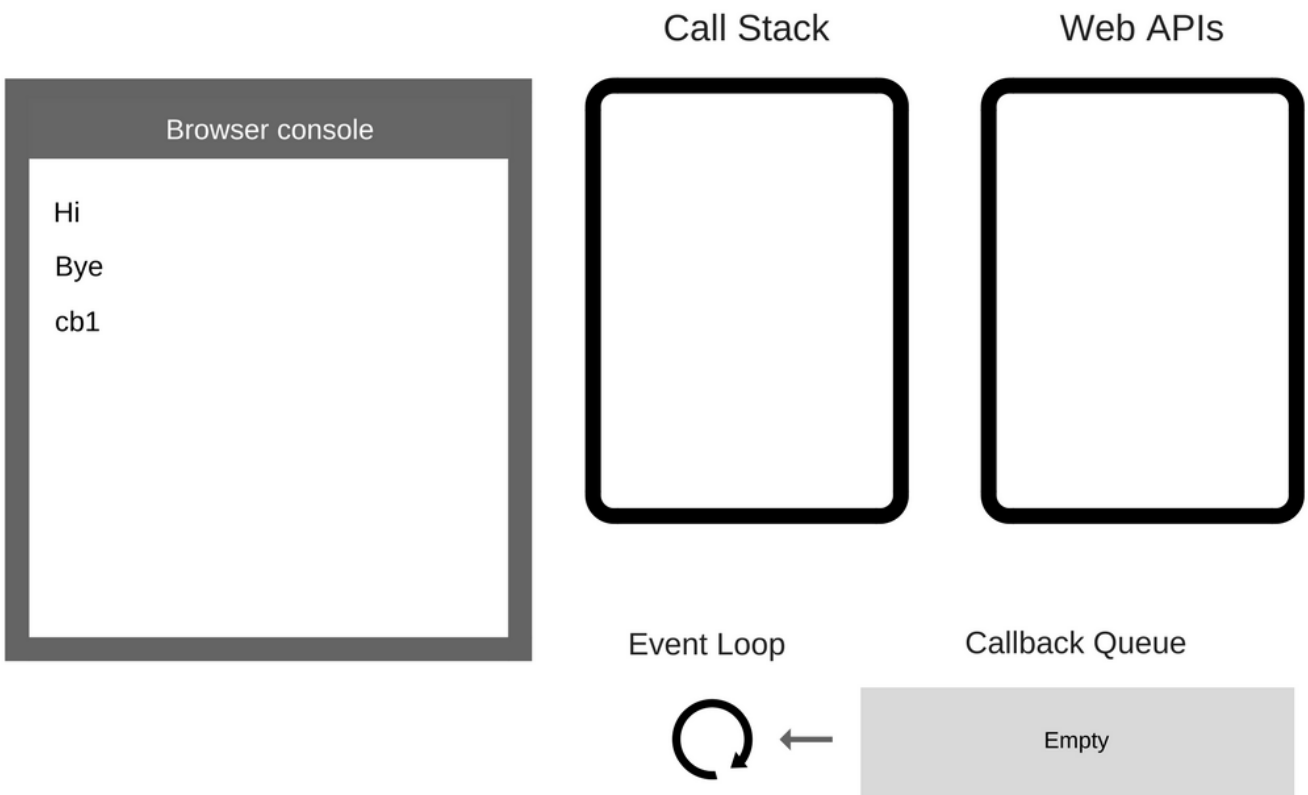
15 / 16





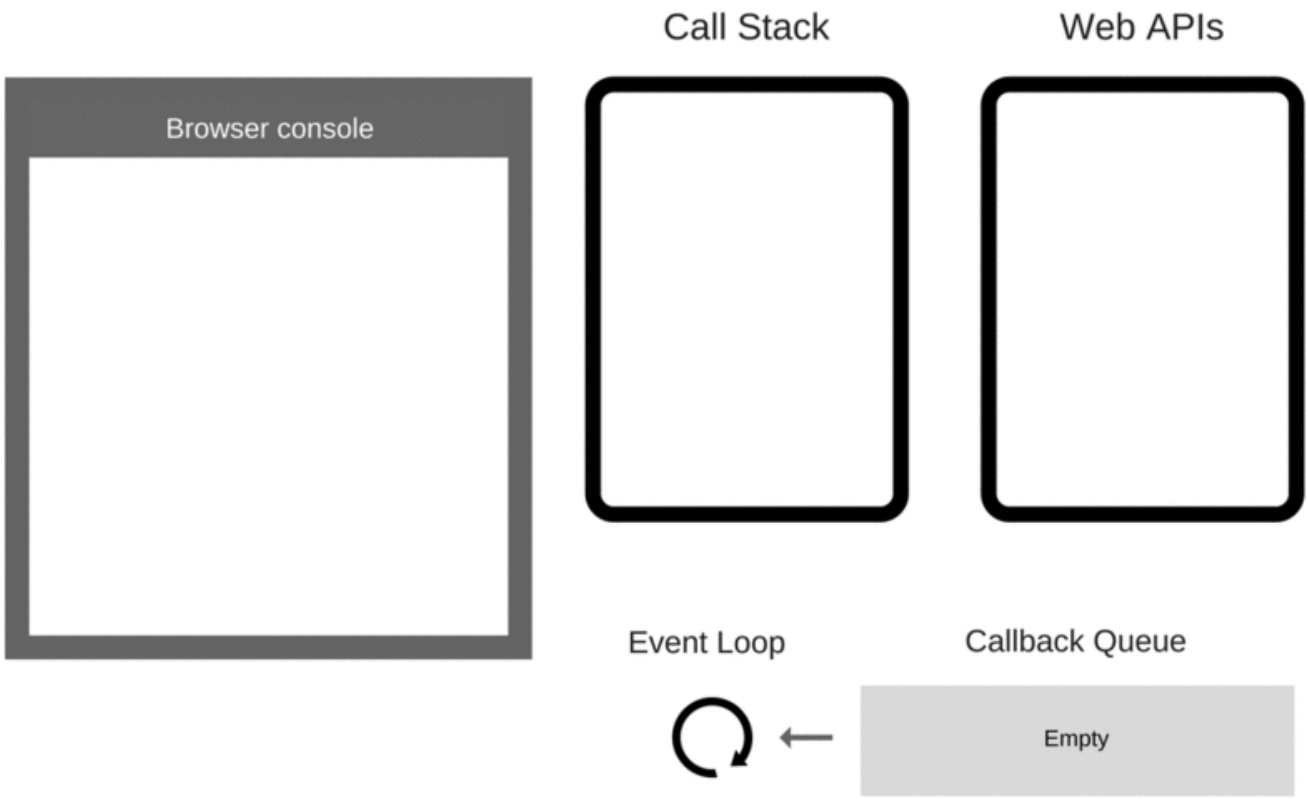
16. `cb1` is removed from the Call Stack.

16 / 16



A quick recap:

1 / 16



It's interesting to note that ES6 specifies how the event loop should work, meaning that technically it's within the scope of the JS engine's responsibilities, which is no longer playing just a hosting environment role. One main reason for this change is the introduction of Promises in ES6 because the latter require access to a direct, fine-grained control over scheduling operations on the event loop queue (we'll discuss them in a greater detail later).

How setTimeout(...) works

It's important to note that `setTimeout(…)` doesn't automatically put your callback on the event loop queue. It sets up a timer. When the timer expires, the environment places your callback into the event loop, so that some future tick will pick it up and execute it.Take a look at this code:

```
1  setTimeout(myCallback, 1000);
```

sample6.js hosted with ❤ by GitHub view raw

That doesn't mean that `myCallback` will be executed in 1,000 ms but rather that, in 1,000 ms, `myCallback` will be added to the queue. The queue, however, might have other events that have been added earlier — your callback will have to wait.

There are quite a few articles and tutorials on getting started with async code in JavaScript that suggest doing a setTimeout(callback, 0). Well, now you know what the Event Loop does and how setTimeout works: calling setTimeout with 0 as a second argument just defers the callback until the Call Stack is clear.

Take a look at the following code:

```
console.log('Hi');
setTimeout(function() {
    console.log('callback');
}, 0);
console.log('Bye');
```

sample7.js hosted with ❤ by GitHub view raw

Although the wait time is set to 0 ms, the result in the browser console will be the following:

```
Hi
Bye
callback
```

What are Jobs in ES6 ?

A new concept called the “Job Queue” was introduced in ES6. It’s a layer on top of the Event Loop queue. You are most likely to bump into it when dealing with the asynchronous behavior of Promises (we’ll talk about them too).

We’ll just touch on the concept now so that when we discuss async behavior with Promises, later on, you understand how those actions are being scheduled and processed.

Imagine it like this: the Job Queue is a queue that’s attached to the end of every tick in the Event Loop queue. Certain async actions that may occur during a tick of the event loop will not cause a whole new event to be added to the event loop queue, but will instead add an item (aka Job) to the end of the current tick’s Job queue.

This means that you can add another functionality to be executed later, and you can rest assured that it will be executed right after, before anything else.

A Job can also cause more Jobs to be added to the end of the same queue. In theory, it's possible for a Job "loop" (a Job that keeps adding other Jobs, etc.) to spin indefinitely, thus starving the program of the necessary resources needed to move on to the next event loop tick. Conceptually, this would be similar to just expressing a long-running or infinite loop (like `while (true)` ..) in your code.

Jobs are kind of like the `setTimeout(callback, 0)` "hack" but implemented in such a way that they introduce a much more well-defined and guaranteed ordering: later, but as soon as possible.

Callbacks

As you already know, callbacks are by far the most common way to express and manage asynchrony in JavaScript programs. Indeed, the callback is the most fundamental async pattern in the JavaScript language. Countless JS programs, even very sophisticated and complex ones, have been written on top of no other async foundation than the callback.

Except that callbacks don't come with no shortcomings. Many developers are trying to find better async patterns. It's impossible, however, to effectively use any abstraction if you don't understand what's actually under the hood.

In the following chapter, we'll explore couple of these abstractions in depth to show why more sophisticated async patterns (that will be discussed in subsequent posts) are necessary and even recommended.

Nested Callbacks

Look at the following code:

```
listen('click', function (e){
    setTimeout(function(){
        ajax('https://api.example.com/endpoint', function (text){
            if (text == "hello") {
                doSomething();
            }
            else if (text == "world") {
                doSomethingElse();
            }
        });
    }, 500);
});
```

sample7.js hosted with ❤ by GitHub view raw

We've got a chain of three functions nested together, each one representing a step in an asynchronous series.

This kind of code is often called a "callback hell". But the "callback hell" actually has almost nothing to do with the nesting/indentation. It's a much deeper problem than that.

First, we're waiting for the "click" event, then we're waiting for the timer to fire, then we're waiting for the Ajax response to come back, at which point it might get all repeated again.

At first glance, this code may seem to map its asynchrony naturally to sequential steps like:

```
listen('click', function (e) {
        // ..
});
```

sample8.js hosted with ❤ by GitHub view raw

Then we have:

```
setTimeout(function(){
    // ..
}, 500);
```

sample9.js hosted with ❤ by GitHub view raw

Then later we have:

```
ajax('https://api.example.com/endpoint', function (text){
    // ..
});
```

sample10.js hosted with ❤ by GitHub view raw

And finally:

```
if (text == "hello") {
    doSomething();
}
else if (text == "world") {
```

```
    doSomethingElse();
}
```

sample11.js hosted with ❤ by GitHub view raw

So, such a sequential way of expressing your async code seems a lot more natural, doesn't it? There must be such a way, right?

Promises

Take a look at the following code:

```
var x = 1;
var y = 2;
console.log(x + y);
```

sample12.js hosted with ❤ by GitHub view raw

It's all very straightforward: it sums the values of `x` and `y` and prints it to the console. What if, however, the value of `x` or `y` was missing and was still to be determined? Say, we need to retrieve the values of both `x` and `y` from the server, before they can be used in the expression. Let's imagine that we have a function `loadX` and `loadY` that respectively load the values of `x` and `y` from the server. Then, imagine that we have a function `sum` that sums the values of `x` and `y` once both of them are loaded.

It could look like this (quite ugly, isn't it):

```
function sum(getX, getY, callback) {
    var x, y;
    getX(function(result) {
        x = result;
        if (y !== undefined) {
            callback(x + y);
        }
    });
    getY(function(result) {
        y = result;
        if (x !== undefined) {
            callback(x + y);
        }
    });
}
// A sync or async function that retrieves the value of `x`
function fetchX() {
```

```
    // ..
}


// A sync or async function that retrieves the value of `y`
function fetchY() {
    // ..
}
sum(fetchX, fetchY, function(result) {
    console.log(result);
});
```

sample13.js hosted with ❤ by GitHub view raw

There is something very important here — in that snippet, we treated `x` and `y` as **future** values, and we expressed an operation `sum(…)` that (from the outside) did not care whether `x` or `y` or both were or weren't available right away.

Of course, this rough callbacks-based approach leaves much to be desired. It's just a first tiny step towards understanding the benefits of reasoning about *future values* without worrying about the time aspect of when they will be available.

Promise Value

Let's just briefly glimpse at how we can express the `x + y` example with Promises:

```
function sum(xPromise, yPromise) {
        // `Promise.all([ .. ])` takes an array of promises,
        // and returns a new promise that waits on them
        // all to finish
        return Promise.all([xPromise, yPromise])

        // when that promise is resolved, let's take the
        // received `X` and `Y` values and add them together.
        .then(function(values){
                // `values` is an array of the messages from the
                // previously resolved promises
                return values[0] + values[1];
        } );
}

// `fetchX()` and `fetchY()` return promises for
// their respective values, which may be ready
// *now* or *later*.
sum(fetchX(), fetchY())

// we get a promise back for the sum of those
```

```
// we get a promise back for the sum of those
// two numbers.
// now we chain-call `then(...)` to wait for the
// resolution of that returned promise.
.then(function(sum){
    console.log(sum);
});
```

sample14.js hosted with ❤️ by GitHub view raw

There are two layers of Promises in this snippet.

`fetchX()` and `fetchY()` are called directly, and the values they return (promises!) are passed to `sum(...)`. The underlying values these promises represent may be ready *now* or *later*, but each promise normalizes its behavior to be the same regardless. We reason about `x` and `y` values in a time-independent way. They are *future values*, period.

The second layer is the promise that `sum(...)` creates (via `Promise.all([ ... ])`) and returns, which we wait on by calling `then(...)`. When the `sum(...)` operation completes, our sum *future value* is ready and we can print it out. We hide the logic for waiting on the `x` and `y` *future values* inside of `sum(...)`.

Note: Inside `sum(…)`, the `Promise.all([ … ])` call creates a promise (which is waiting on `promiseX` and `promiseY` to resolve). The chained call to `.then(...)` creates another promise, which the return `values[0] + values[1]` line immediately resolves (with the result of the addition). Thus, the `then(...)` call we chain off the end of the `sum(...)` call — at the end of the snippet — is actually operating on that second promise returned, rather than the first one created by `Promise.all([ ... ])`. Also, although we are not chaining off the end of that second `then(...)`, it too has created another promise, had we chosen to observe/use it. This Promise chaining stuff will be explained in much greater detail later in this chapter.

With Promises, the `then(...)` call can actually take two functions, the first for fulfillment (as shown earlier), and the second for rejection:

```
sum(fetchX(), fetchY())
.then(
    // fullfillment handler
    function(sum) {
        console.log( sum );
    },
    // rejection handler
```

```
    // rejection handler
    function(err) {
        console.error( err ); // bummer!
    }
);
```

sample15.js hosted with ❤ by GitHub view raw

If something went wrong when getting `x` or `y`, or something somehow failed during the addition, the promise that `sum(...)` returns would be rejected, and the second callback error handler passed to `then(...)` would receive the rejection value from the promise.

Because Promises encapsulate the time-dependent state — waiting on the fulfillment or rejection of the underlying value — from the outside, the Promise itself is time-independent, and thus Promises can be composed (combined) in predictable ways regardless of the timing or outcome underneath.

Moreover, once a Promise is resolved, it stays that way forever — it becomes an *immutable value* at that point — and can then be *observed* as many times as necessary.

It's really useful that you can actually chain promises:

```
function delay(time) {
    return new Promise(function(resolve, reject){
        setTimeout(resolve, time);
    });
}

delay(1000)
.then(function(){
    console.log("after 1000ms");
    return delay(2000);
})
.then(function(){
    console.log("after another 2000ms");
})
.then(function(){
    console.log("step 4 (next Job)");
    return delay(5000);
})
// ...
```

sample16.js hosted with ❤ by GitHub view raw

Calling `delay(2000)` creates a promise that will fulfill in 2000ms, and then we return that from the first `then(...)` fulfillment callback, which causes the second `then(...)`'s promise to wait on that 2000ms promise.

Note: Because a Promise is externally immutable once resolved, it's now safe to pass that value around to any party, knowing that it cannot be modified accidentally or maliciously. This is especially true in relation to multiple parties observing the resolution of a Promise. It's not possible for one party to affect another party's ability to observe Promise resolution. Immutability may sound like an academic topic, but it's actually one of the most fundamental and important aspects of Promise design, and shouldn't be casually passed over.

To Promise or not to Promise?

An important detail about Promises is knowing for sure if some value is an actual Promise or not. In other words, is it a value that will behave like a Promise?

We know that Promises are constructed by the `new Promise(…)` syntax, and you might think that `p instanceof Promise` would be a sufficient check. Well, not quite.

Mainly because you can receive a Promise value from another browser window (e.g. iframe), which would have its own Promise, different from the one in the current window or frame, and that check would fail to identify the Promise instance.

Moreover, a library or framework may choose to vend its own Promises and not use the native ES6 Promise implementation to do so. In fact, you may very well be using Promises with libraries in older browsers that have no Promise at all.

Swallowing exceptions

If at any point in the creation of a Promise, or in the observation of its resolution, a JavaScript exception error occurs, such as a `TypeError` or `ReferenceError`, that exception will be caught, and it will force the Promise in question to become rejected.

For example:

```
var p = new Promise(function(resolve, reject){
    foo.bar();    // `foo` is not defined, so error!
    resolve(374); // never gets here :(
});

p.then(
```

```
    function fulfilled(){
        // never gets here :(
    },
    function rejected(err){
        // `err` will be a `TypeError` exception object
        // from the `foo.bar()` line.
    }
);
```

sample17.js hosted with ❤ by GitHub view raw

But what happens if a Promise is fulfilled yet there was a JS exception error during the observation (in a `then(…)` registered callback)? Even though it won't be lost, you may find the way they're handled a bit surprising. Until you dig a little deeper:

```
var p = new Promise( function(resolve,reject){
        resolve(374);
});

p.then(function fulfilled(message){
    foo.bar();
    console.log(message);   // never reached
},
    function rejected(err){
        // never reached
    }
);
```

sample18.js hosted with ❤ by GitHub view raw

It looks like the exception from `foo.bar()` really did get swallowed. It wasn't, though. There was something deeper that went wrong, however, which we failed to listen for. The `p.then(…)` call itself returns another promise, and it's that promise that will be rejected with the `TypeError` exception.

Handling uncaught exceptions

There are other approaches which many would say are *better*.

A common suggestion is that Promises should have a `done(…)` added to them, which essentially marks the Promise chain as "done." `done(…)` doesn't create and return a Promise, so the callbacks passed to `done(..)` are obviously not wired up to report problems to a chained Promise that doesn't exist.

It's treated as you might usually expect in uncaught error conditions: any exception inside a `done(..)` rejection handler would be thrown as a global uncaught error (in the developer console, basically):

```
var p = Promise.resolve(374);

p.then(function fulfilled(msg){
    // numbers don't have string functions,
    // so will throw an error
    console.log(msg.toLowerCase());
})
.done(null, function() {
    // If an exception is caused here, it will be thrown globally
});
```

sample19.js hosted with ❤ by GitHub view raw

What's happening in ES8? Async/await

JavaScript ES8 introduced `async/await` that makes the job of working with Promises easier. We'll briefly go through the possibilities `async/await` offers and how to leverage them to write async code.

So, let's see how async/await works.

You define an asynchronous function using the `async` function declaration. Such functions return an AsyncFunction object. The `AsyncFunction` object represents the asynchronous function which executes the code, contained within that function.

When an async function is called, it returns a `Promise` . When the async function returns a value, that's not a `Promise` , a `Promise` will be automatically created and it will be resolved with the returned value from the function. When the `async` function throws an exception, the `Promise` will be rejected with the thrown value.

An `async` function can contain an `await` expression, that pauses the execution of the function and waits for the passed Promise's resolution, and then resumes the async function's execution and returns the resolved value.

You can think of a `Promise` in JavaScript as the equivalent of Java's `Future` or `C#`'s Task.

> *The purpose of `async/await` is to simplify the behavior of using promises.*

Let's take a look at the following example:

```
// Just a standard JavaScript function
function getNumber1() {
    return Promise.resolve('374');
}
// This function does the same as getNumber1
async function getNumber2() {
    return 374;
}
```

sample20.js hosted with ❤ by GitHub view raw

Similarly, functions that are throwing exceptions are equivalent to functions which return promises that have been rejected:

```
function f1() {
    return Promise.reject('Some error');
}
async function f2() {
    throw 'Some error';
}
```

sample21.js hosted with ❤ by GitHub view raw

The `await` keyword can only be used in `async` functions and allows you to synchronously wait on a Promise. If we use promises outside of an `async` function, we'll still have to use `then` callbacks:

```
async function loadData() {
    // `rp` is a request-promise function.
    var promise1 = rp('https://api.example.com/endpoint1');
    var promise2 = rp('https://api.example.com/endpoint2');

    // Currently, both requests are fired, concurrently and
    // now we'll have to wait for them to finish
    var response1 = await promise1;
    var response2 = await promise2;
    return response1 + ' ' + response2;
}
// Since, we're not in an `async function` anymore
// we have to use `then`.
loadData().then(() => console.log('Done'));
```

sample22.js hosted with ❤ by GitHub view raw

You can also define async functions using an “async function expression”. An async function expression is very similar to and has almost the same syntax as, an async function statement. The main difference between an async function expression and an async function statement is the function name, which can be omitted in async function expressions to create anonymous functions. An async function expression can be used as an IIFE (Immediately Invoked Function Expression) which runs as soon as it is defined.

It looks like this:

```
var loadData = async function() {
    // `rp` is a request-promise function.
    var promise1 = rp('https://api.example.com/endpoint1');
    var promise2 = rp('https://api.example.com/endpoint2');

    // Currently, both requests are fired, concurrently and
    // now we'll have to wait for them to finish
    var response1 = await promise1;
    var response2 = await promise2;
    return response1 + ' ' + response2;
}
```

sample23.js hosted with ❤ by GitHub view raw

More importantly, async/await is supported in all major browsers:

If this compatibility is not what you are after, there are also several JS transpilers like Babel and TypeScript.

At the end of the day, the important thing is not to blindly choose the “latest” approach to writing async code. It’s essential to understand the internals of async JavaScript,

learn why it's so critical and comprehend in-depth the internals of the method you have chosen. Every approach has pros and cons as with everything else in programming.

5 Tips on writing highly maintainable, non-brittle async code

1. **Clean code:** Using async/await allows you to write a lot less code. Every time you use async/await you skip a few unnecessary steps: write .then, create an anonymous function to handle the response, name the response from that callback e.g.

```
// `rp` is a request-promise function.
rp('https://api.example.com/endpoint1').then(function(data) {
 // …
});
```

sample24.js hosted with ❤ by GitHub view raw

Versus:

```
// `rp` is a request-promise function.
var response = await rp('https://api.example.com/endpoint1');
```

sample25.js hosted with ❤ by GitHub view raw

2. **Error handling:** Async/await makes it possible to handle both sync and async errors with the same code construct — the well-known try/catch statements. Let's see how it looks with Promises:

```
function loadData() {
    try { // Catches synchronous errors.
        getJSON().then(function(response) {
            var parsed = JSON.parse(response);
            console.log(parsed);
        }).catch(function(e) { // Catches asynchronous errors
            console.log(e);
        });
    } catch(e) {
        console.log(e);
    }
}
```

sample26.js hosted with ❤ by GitHub view raw

Versus:

```
async function loadData() {
    try {
        var data = JSON.parse(await getJSON());
        console.log(data);
    } catch(e) {
        console.log(e);
    }
}
```

sample27.js hosted with ❤️ by GitHub view raw

3. **Conditionals:** Writing conditional code with `async/await` is a lot more straightforward:

```
function loadData() {
  return getJSON()
    .then(function(response) {
      if (response.needsAnotherRequest) {
        return makeAnotherRequest(response)
          .then(function(anotherResponse) {
            console.log(anotherResponse)
            return anotherResponse
          })
      } else {
        console.log(response)
        return response
      }
    })
}
```

sample28.js hosted with ❤️ by GitHub view raw

Versus:

```
async function loadData() {
  var response = await getJSON();
  if (response.needsAnotherRequest) {
    var anotherResponse = await makeAnotherRequest(response);
    console.log(anotherResponse)
    return anotherResponse
  } else {
    console.log(response);
    return response;
```

```
  }
}
```

sample29.js hosted with ❤️ by GitHub view raw

4. **Stack Frames:** Unlike with `async/await`, the error stack returned from a promise chain gives no clue of where the error happened. Look at the following:

```
function loadData() {
  return callAPromise()
    .then(callback1)
    .then(callback2)
    .then(callback3)
    .then(() => {
      throw new Error("boom");
    })
}
loadData()
  .catch(function(e) {
    console.log(err);
// Error: boom at callAPromise.then.then.then.then (index.js:8:13)
});
```

sample30.js hosted with ❤️ by GitHub view raw

Versus:

```
async function loadData() {
  await callAPromise1()
  await callAPromise2()
  await callAPromise3()
  await callAPromise4()
  await callAPromise5()
  throw new Error("boom");
}
loadData()
  .catch(function(e) {
    console.log(err);
    // output
    // Error: boom at loadData (index.js:7:9)
});
```

sample31.js hosted with ❤️ by GitHub view raw

5. **Debugging:** If you have used promises, you know that debugging them is a nightmare. For example, if you set a breakpoint inside a .then block and use debug shortcuts like “stop-over”, the debugger will not move to the following .then because it only “steps” through synchronous code.
With `async/await` you can step through await calls exactly as if they were normal synchronous functions.

Writing **async JavaScript code is important** not only for the apps themselves but **for libraries as well**.

For example, the SessionStack library records everything in your web app/website: all DOM changes, user interactions, JavaScript exceptions, stack traces, failed network requests, and debug messages.

And this all has to happen in your production environment without impacting any of the UX. We need to heavily optimize our code and make it asynchronous as much as possible so that we can increase the number of events that are being processed by the Event Loop.

And not just the library! When you replay a user session in SessionStack, we have to render everything that happened in your user’s browser at the time the problem occurred, and we have to reconstruct the whole state, allowing you to jump back and forth in the session timeline. In order to make this possible, we’re heavily employing the async opportunities that JavaScript provides.

There is a free plan that allows you to get started for free.

Resources:

- https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md
- https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch3.md
- http://nikgrozev.com/2017/10/01/async-await/

JavaScript Tutorial Web Development Programming Asynchronous

About Help Legal