**Introduction: Async JavaScript and HTTP Requests**

**In this Unit, you’ll learn about asynchronous JavaScript and how to use it in web development.**

The goal of this unit is to learn about working with asynchronous JavaScript. This will allow you to eventually make HTTP requests to APIs (Application Programming Interfaces). Working with APIs will enable you to work with data stored on remote servers, including data from third-party sites (such as Instagram and Reddit).

After this unit, you will be able to:

* Write asynchronous JavaScript with async-await and promises syntax
* Explain the different types of HTTP requests
* Describe REST protocol
* Work with JSON data
* Make HTTP requests to external web APIs

You will put all of this knowledge into practice with an upcoming Portfolio Project. You can complete the Portfolio Project either in parallel with or after taking the prerequisite content — it’s up to you!

Learning is social. Whatever you’re working on, be sure to connect with the Codecademy community in the [forums](https://discuss.codecademy.com/). Remember to check in with the community regularly, including for things like asking for code reviews on your project work and providing code reviews to others in the [projects category](https://discuss.codecademy.com/c/project/1833), which can help to reinforce what you’ve learned.

**Summary Three Articles:**

Article 1:

Synchronous code and asynchronous code both have roles to play in programming. Understanding the concept of how asynchronous code works gives us an extra tool to make our apps work faster and more efficiently. We can avoid blocking users and give them a more seamless browsing experience. However, we would need to consider the number of threads that our programming language can access, which also depends on what resources our computer has. With this in mind, consider what type of code you need, is it synchronous or asynchronous?

Article 2:

Asynchronous code can really benefit sites and apps that rely on actions that take time. Even though JavaScript is a single-threaded language, it can still execute asynchronous code using the event loop. We took a look at some of the main ways javascript accomplishes synchronicity via callbacks, **setTimeout()**, and **setInterval()**. With this new knowledge, let’s continue to implement asynchronicity into our programs!

Article 3:

Thanks to the event loop, JavaScript is a single-threaded, event-driven language that can run non-blocking code asynchronously. The Event Loop can be summarized as: when code is executed, it is handled by the heap and call stack, which interact with Node and Web APIs. Those APIs enable concurrency and pass asynchronous messages back to the stack via an event queue. The event queue’s interaction with the call stack is managed by an event loop. Altogether, those parts maintain the order of code execution when we run asynchronous functions.

**Article 1-General Asynchronous Programming Concepts**

**Explore asynchronous programming and how it allows applications/apps to run operations in a non-sequential order.**

**What is Synchronous Code?**

Before we define asynchronous code, let’s first start with synchronous code. We don’t even have to start with code, let’s use a real-life example.

Consider the building of a house. We would first need to first lay down the bricks that make our foundation. Then, we layer more bricks on top of each other, building the house from the ground up. We can’t skip a level and expect our house to be stable. Therefore, the laying of bricks happens *synchronously*, or in sequential order.

Likewise, *synchronous code* executes in sequential order — it starts with the code at the top of the file and executes line by line until it gets to the end of the file. This type of behavior is known as *blocking* (or blocking code) since each line of code cannot execute until the previous line finishes.

**What is Asynchronous Code?**

Let’s begin again with examining a real-life scenario, like baking a cake. We could start to preheat the oven and prepare our cake’s ingredients while we wait for our oven to heat up. The wait for the oven and the preparation of ingredients can happen at the same time, and thus these actions happen *asynchronously*.

Similarly, *asynchronous code* can be executed in parallel to other code that is already running. Without the need to wait for other code to finish before executing, our apps can save time and be more efficient. This type of behavior is considered *non-blocking*.

**Asynchronous Code Under the Hood**

For most programming languages, the ability to execute asynchronous code depends on the number of *threads* that an app has access to. We can think of a thread as a resource that a computer provides an app to do a task. Typically one thread allows for an app to complete one task. If we return to our house example, our computers thread tasks might look like this:

Thread 1: build house foundation -> build walls -> construct floor

A single thread could work for a synchronous task like building a house. However, in our cake baking example, our threads would have to look like this:

Thread 1: preheat oven  
Thread 2: prepare ingredients -> bake-cake

We won’t discuss in-depth how many threads an app can access but we should note that the more threads we have, the more tasks we can run concurrently. Also, in most modern-day computers, multithreading is achieved by having a CPU that has multiple cores or by some other technology.

**Asynchronous Code in Web Development**

Similar to how asynchronous behavior is useful in baking a cake, it can also be helpful for web programming. If we use synchronous (blocking) code in the browser, we might be stopping a user from being able to interact with a web app until the code is done running. This isn’t a great user experience. If our app takes a long time to load, our users might think that something’s wrong and might even opt to browse a different site!

However, if we opt for an asynchronous approach, we can cut down on the wait time. We’d load only the code that’s necessary for user interactions and then load up other bits of code in the background. With asynchronous code, we can create better user experiences and make apps that work more efficiently!

Free Response Question

In your own words, how are synchronous actions different from asynchronous actions? Try to also come up with your own examples of both types of actions.

Your response

Synchronous actions are different than asynchronous actions because they the tasks are highly dependant on eachother hence can not be run in parallel. An example of synchronous would be user visiting an online store and browsing items>adding them to cart > buying the items where all these tasks cannot be performed asynchronously. While an asynchronous example would be that of doing house chores such as washing clothes in clothes washer, dishes in dish washer while we can attend to other house hold activities.

Our answer

Synchronous actions happen sequentially, one after the other. Whereas, asynchronous actions can happen at the same without one action blocking the other.

  Edit Response

**Review**

Synchronous code and asynchronous code both have roles to play in programming. Understanding the concept of how asynchronous code works gives us an extra tool to make our apps work faster and more efficiently. We can avoid blocking users and give them a more seamless browsing experience. However, we would need to consider the number of threads that our programming language can access, which also depends on what resources our computer has. With this in mind, consider what type of code you need, is it synchronous or asynchronous?

**Article 2-Introduction to Asynchronous JavaScript**

**Learn how JavaScript enables asynchronous actions.**

**Asynchronous Code in Web Development**

JavaScript provides us with a seamless web browsing experience using asynchronous code. Sites often allow us to perform different interactions like scrolling through content, clicking to create a pop-up modal, typing out text, etc. When a site is set up to respond to different user actions at the same time, it’s likely that this site is using asynchronous JavaScript code. Such code takes into consideration how users might use the site without *blocking* them (forcing the user to wait for code from one interaction to finish before moving on to the next).

It is our job as developers to think about how much time it takes to complete a task and how to plan around that wait. Tasks like contacting the back-end to retrieve information, querying our database for user information, or making a request to an external server, like a 3rd party API, take varying amounts of time. Since we aren’t sure when we’ll get this information back, we can use asynchronous code to run these tasks in the background. Let’s see how JavaScript handles asynchronous code.

**JavaScript and Asynchronous Code**

JavaScript is a *single-threaded* language. This means it has a single thread that can carry out one task at a time. However, Javascript has what is known as the *event loop*, a specific design that allows it to perform asynchronous tasks even while only using a single thread (more on this later!). Let’s examine some examples of asynchronous code in Javascript!

**Asynchronous Callbacks**

One common example of asynchronicity in JavaScript is the use of *asynchronous callbacks*. This is a type of callback function that executes after a specific condition is met and runs concurrently to any other code currently running. Let’s look at an example:

easterEgg.addEventListener('click', () => {  
  console.log('Up, Up, Down, Down, Left, Right, Left, Right, B, A');  
});

In the code above, the function passed as the second argument of **.addEventListener()** is an asynchronous callback — this function doesn’t execute until the **easterEgg** is clicked.

**setTimeout**

In addition to asynchronous callbacks, JavaScript provides a handful of built-in functions that can perform tasks asynchronously. One function that is commonly used is the **[setTimeout()]**([https://developer.mozilla.org/en-US/docs/Web/API/setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout" \t "_blank)) function.

With **setTimeout()** we can write code that tells our JavaScript program to wait a minimum amount of time before executing its callback function. Take a look at this example:

setTimeout(() => {  
  console.log('Delay the printing of this string, please.');  
}, 1000);

Notice that **setTimeout()** takes 2 arguments, a callback function and a number specifying how long to wait before executing the function. In the example above, the function will print **'Delay the printing of this string, please.'** after **1000** milliseconds (or 1 second) have passed.

Since **setTimeout()** is non-blocking, we can be executing multiple lines of code at the same time! . Imagine if we had a program like this:

setTimeout(() => {  
  console.log('Delay the printing of this string, please.');  
}, 1000);  
console.log('Doing important stuff.');  
console.log('Still doing important stuff.');

Which outputs:

'Doing important stuff.'  
'Still doing important stuff.'   
'Delay the printing of this string, please.'

If we take a closer look at the output, we’ll see that our **setTimeout()**‘s callback function didn’t execute until after our other very important **console.log()** statements were executed.

In web development, this means we can write code to wait for an event to trigger all while a user goes on interacting with our app. One such example could be if a user goes to a shopping site and gets notified that an item is up for sale and only for a limited time. Our asynchronous code could allow the user to interact with our site and when the sale timer expires, our code will remove the sale item.

**setInterval()**

Another common built-in function is **[setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/setInterval" \t "_blank)** which also takes a callback function and a number specifying how often the callback function should execute. For example:

setInterval(() => {  
  alert('Are you paying attention???')  
}, 300000)

The **setInterval()** would call the **alert()** function and show a pop-up message of **'Are you paying attention???'** every **300000** milliseconds (or 5 minutes). Note: Please don’t actually do this in your apps, thank you.

While we wait for our alert to chime in every 5 minutes, our users could still use our app! Note: Again, please don’t do this.

With **setInterval()**, we can programmatically create an alarm, a countdown timer, set the frequency of an animation, and so much more!

Free Response Question

How is **setInterval()** considered asynchronous code?

Your response

setInterval() allows us to execute code after a said amount of time repeatedly hence allowing us to do other stuff i mean execute other code asynchronously

Our answer

**setInterval()** is considered asynchronous code because the supplied callback function is not blocked by other code — the callback function can be executed in parallel to other code being executed. The determining factor for when the callback function is executed is determined by the second argument provided.

  Edit Response

**Review**

Asynchronous code can really benefit sites and apps that rely on actions that take time. Even though JavaScript is a single-threaded language, it can still execute asynchronous code using the event loop. We took a look at some of the main ways javascript accomplishes synchronicity via callbacks, **setTimeout()**, and **setInterval()**. With this new knowledge, let’s continue to implement asynchronicity into our programs!

**Article 3-Concurrency Model and Event Loop in JavaScript**

**How JavaScript uses its event loop to emulate concurrency**

If you’ve learned about asynchronous programming, you may wonder how your code can actually be non-blocking and move on to other tasks while it waits for asynchronous operations to complete. This article will remove some of the abstractions about how JavaScript can emulate concurrency by looking at what’s going on with the event loop behind the scenes. But what exactly is the event loop? And why do we need it?

**Why Do We Need an Event Loop?**

JavaScript is a *single-threaded* language, which means that two statements can’t be executed simultaneously. For example, if you have a **for** loop that takes a while to process, it’ll have to finish executing before the rest of your code runs. That results in blocking code. But as we already learned, we can run non-blocking code in JavaScript, which is where the Event Loop comes in. Input/output (I/O) is handled with events and callbacks so code execution can continue. Let’s look at an example of blocking and non-blocking code. Run this block of code yourself locally.

console.log("I'm learning about");  
   
for (let idx=0; idx < 999999999; idx++) {}  
   
// The second console.log() statement is  
// delayed by the for loop's execution  
console.log("the Event Loop");

Free Response Question

What happened when you ran the code? What did you notice about the timing of the execution of your **console.log()** statements?

Your response

The first statement logged instantly while the second statement logged after an interval caused by the for loop as the for loop had to complete its loop before executing and logging the second statement.

Our answer

The code logged 2 lines to the console. The first line logged, then the **for** loop executed, and after some time, the last line logged to the console.

  Edit Response

The example above has synchronous code with a long **for** loop. Here’s what happens:

1. The code executes and “I’m learning about” is logged to the console.
2. Next, a **for** loop executes and runs 999999999 loops, which results in blocking code. If you run this locally, this is where the pause happens.
3. Finally, “the Event Loop” is logged.

Now let’s take a look at the non-blocking example. There are functions like **setTimeout()** that work differently thanks to the Event Loop. Run the code:

console.log("I’m learning about");  
setTimeout(() => { console.log("Event Loop");}, 2000);  
console.log("the");

In this case, the code snippet uses the **setTimeout()** function to demonstrate how JavaScript can be non-blocking with use of the event loop. Here’s what happens:

1. A statement is logged.
2. The **setTimeout()** function is executed.
3. A third line of code executes and logs text: “the”.
4. Finally, the **setTimeout()** function timer completes and additional text is logged: “Event Loop”.

In this case, JavaScript is still single-threaded, but the event loop is enabling something called concurrency.

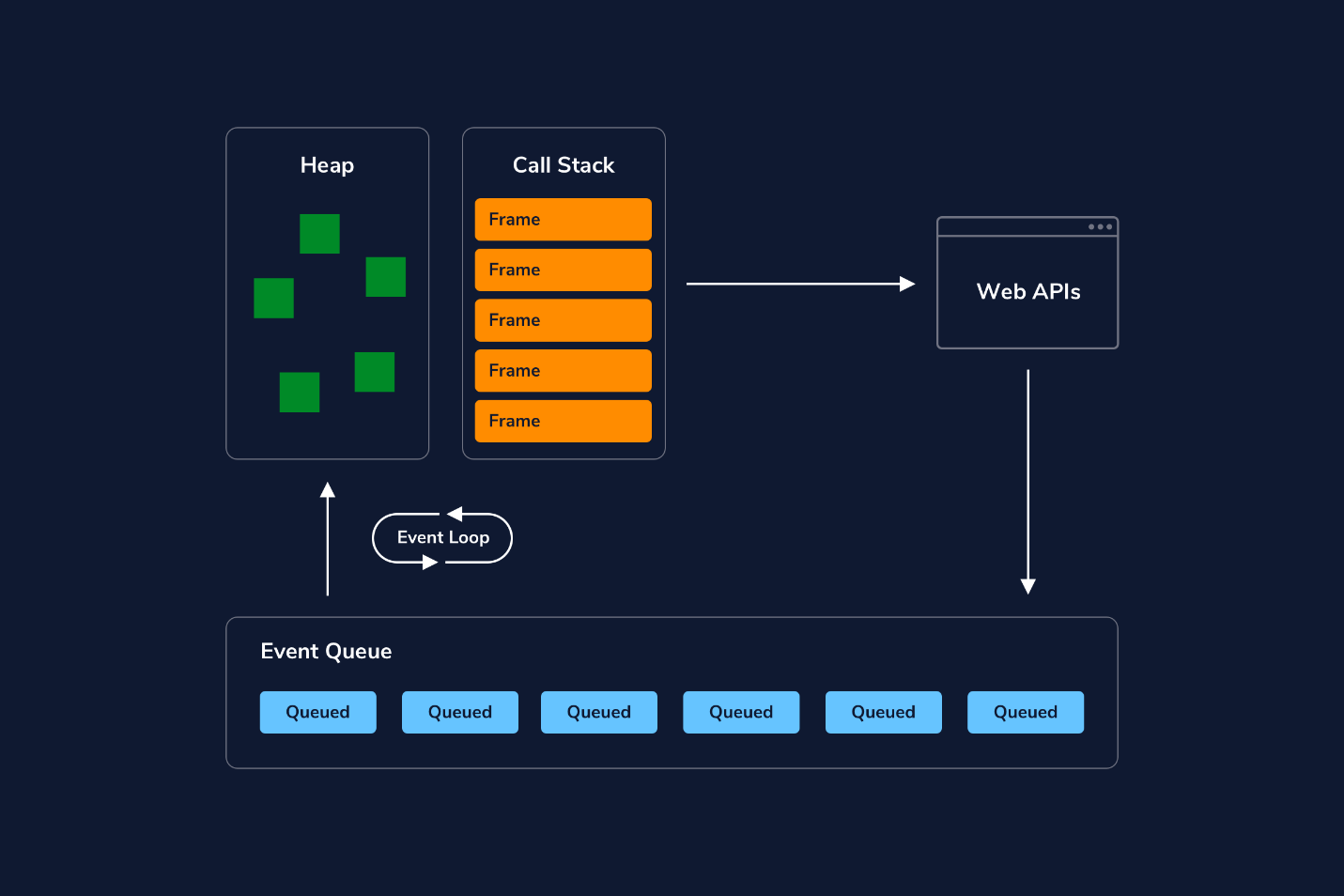
**Concurrency in JavaScript**

Usually when we think about *concurrency* in programming, it means that two or more procedures are executed at the same time on the same shared resources. Since JavaScript is single-threaded, as we saw in the **for** loop example, we’ll never have that flavor of “true” concurrency. However, we can emulate concurrency using the event loop.

**What Is the Event Loop?**

At a high level, the event loop is a system for managing code execution. In the diagram, you can see an overview of how the parts that make up the event loop fit together.

We have data structures that we call the heap and the call stack, which are part of the JavaScript engine. The heap and call stack interact with Node and Web APIs, which pass messages back to the stack via an event queue. The event queue’s interaction with the call stack is managed by an event loop. All together, those parts maintain the order of code execution when we run asynchronous functions. Don’t worry about understanding what those terms mean yet–we’ll dive into them shortly.

[](https://static-assets.codecademy.com/Courses/Learn-JavaScript/Event-Loop-and-Concurrency/JavaScript-Engine-Diagram.png)

*Note: You can click on the image to enlarge it.*

**Understand the Components of the Event Loop**

The *event loop* is made up of these parts:

* Memory Heap
* Call Stack
* Event Queue
* Event Loop
* Node or Web APIs

Let’s take a closer look at each part before we put it all together.

**The Heap**

The *heap* is a block of memory where we store objects in an unordered manner. JavaScript variables and objects that are currently in use are stored in the heap.

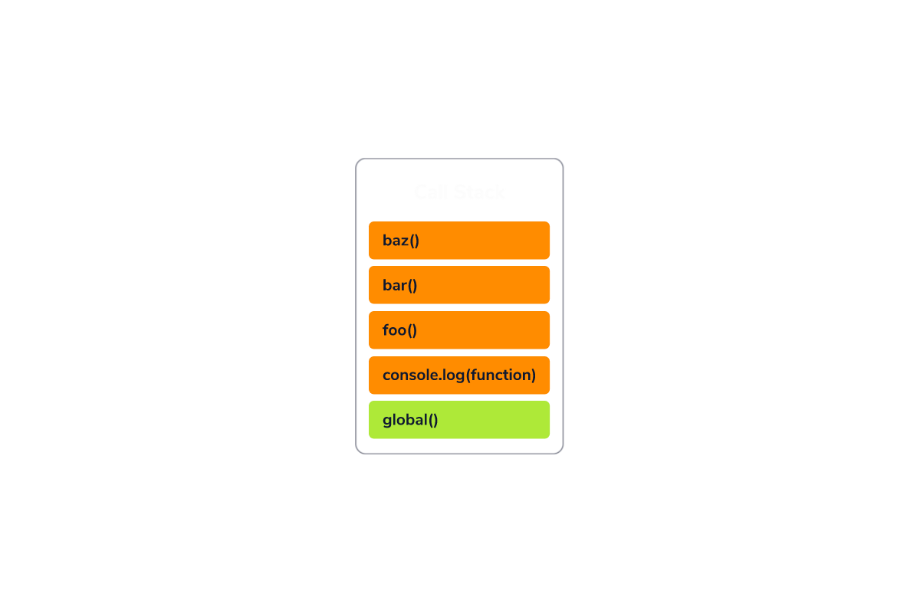
**The Call Stack**

The *stack*, or *call stack*, tracks what function is currently being run in your code.

When you invoke a function, a *frame* is added to the stack. Frames connect that function’s arguments and local variables from the heap. Frames enter the stack in a *last in, first out* (LIFO) order. In the code snippet below, a series of nested functions are declared, then **foo()** is called and logged.

function foo() {  
 return function bar() {  
   return function baz() {  
     return 'I love CodeCademy'  
   }  
 }  
}  
console.log(foo()()());

The function executing at any given point in time is at the top of the stack. In our example code, since we have nested functions, they will all be added to the stack until the innermost function has been executed. When the function finishes executing e.g. returns, its frame is removed from the stack. When we execute **console.log(foo()()())**, we’d see the stack build as follows:



You might have noticed that **global()** is at the bottom of the stack–when you first initiate a program, the *global execution context* is added to the call stack, which contains the global variable and lexical environment. Each subsequent frame for a called function has a function execution context that includes the function’s lexical and variable environment.

So when we say the call stack tracks what function is currently being run in our code, what we are tracking is the current execution context. When a function runs to completion, it is popped off of the call stack. The memory, or the frame, is cleared.

**The Event Queue**

The *event queue* is a list of messages corresponding to functions that are waiting to be processed. In the diagram, these messages are entering the event queue from sources such as various web APIs or async functions that were called and are returning additional events to be handled by the stack. Messages enter the queue in a first in, first out (FIFO) order. No code is executed in the event queue; instead, it holds functions that are waiting to be added back into the stack.

**The Event Loop**

This event loop is a specific part of our overall event loop concept. Messages that are waiting in the event queue to be added back into the stack are added back via the event loop. When the call stack is empty, if there is anything in the event queue, the event loop can add those one at a time to the stack for execution.

1. First the event loop will poll the stack to see if it is empty.
2. It will add the first waiting message.
3. It will repeat steps 1 and 2 until the stack has cleared.

\*\*\*\*\*\*\* MOST IMPORTANT \*\*\*\*\*\*\*

**The Event Loop in Action**

Now that we know all of the pieces of the event loop, let’s walk through some code to understand the event loop in action.

console.log("This is the first line of code in app.js.");  
   
function usingsetTimeout() {  
    console.log("I'm going to be queued in the Event Loop.");  
}  
setTimeout(usingsetTimeout, 3000);  
   
console.log("This is the last line of code in app.js.");

1. **console.log("This is the first line of code in app.js.");** is added to the stack, executes, then pops off of the stack.
2. **setTimeout()** is added to the stack.
3. **setTimeout()**’s callback is passed to be executed by a web API. The timer will run for 3 seconds. After 3 seconds elapse, the callback function, **usingsetTimeout()** is pushed to the Event Queue.
4. The Event Loop, meanwhile, will check periodically if the stack is cleared to handle any messages in the Event Queue.
5. **console.log("This is the last line of code in app.js.");** is added to the stack, executes, then pops off of the stack.
6. The stack is now empty, so the event loop pushes **usingsetTimeout** onto the stack.
7. **console.log("I'm going to be queued in the Event Loop.");** is added to the stack, executes, gets popped
8. **usesetTimeout** pops off of the stack.

**Summary**

Thanks to the event loop, JavaScript is a single-threaded, event-driven language that can run non-blocking code asynchronously. The Event Loop can be summarized as: when code is executed, it is handled by the heap and call stack, which interact with Node and Web APIs. Those APIs enable concurrency and pass asynchronous messages back to the stack via an event queue. The event queue’s interaction with the call stack is managed by an event loop. All together, those parts maintain the order of code execution when we run asynchronous functions.