

## 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](#). 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](#), which can help to reinforce what you've learned.

## 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

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.

**Submit Response**

**Stuck? Get a Hint**

**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?

## 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\(\)](#) 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\(\)](#) 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

How is `setInterval()` considered asynchronous code?



Submit Response

Stuck? Get a Hint

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 asynchronicity via callbacks, `setTimeout()`, and `setInterval()`. With this new knowledge, let's continue to implement asynchronicity into our programs!

## 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

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



[Submit Response](#)

[Stuck? Get a Hint](#)

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.

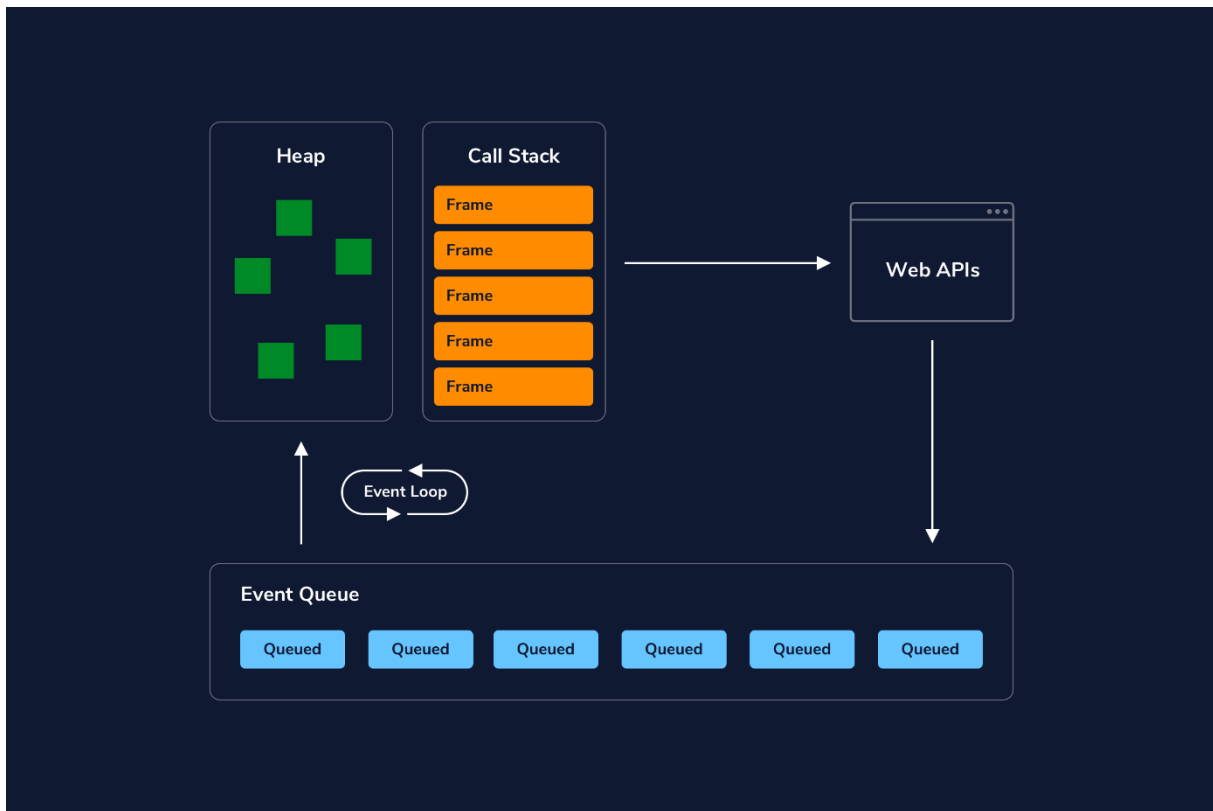
## Concurrency in JavaScript

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.



*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.

## 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. `usingsetTimeout` 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.

### Free response

How is concurrency in JavaScript different from other programming languages?

**Submit Response**

**Stuck? Get a Hint**

### Free response

Describe what role the heap and the stack play in the event loop.

**Submit Response**

**Stuck? Get a Hint**

**Free response**

**Describe in regard to the event loop what is happening when this code executes:**

```
const shopForBeans = () => {  
  return new Promise((resolve, reject) => {  
    const beanTypes = ['kidney', 'fava', 'pinto', 'black', 'garbanzo'];  
    setTimeout(() => {  
      let randomIndex = Math.floor(Math.random() * beanTypes.length);  
      let beanType = beanTypes[randomIndex];  
      console.log(`2. I bought ${beanType} beans because they were on sale.`);  
      resolve(beanType);  
    }, 1000);  
  });  
}
```

```
async function getBeans() {  
  console.log(`1. Heading to the store to buy beans...`);  
  let value = await shopForBeans();  
  console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);  
}
```

```
getBeans();  
console.log("Describe what happens with this `console.log()` statement as well.");
```

**Submit Response**

**Stuck? Get a Hint**

## **JAVASCRIPT PROMISES**

### **Introduction**

In web development, asynchronous programming is notorious for being a challenging topic.

An *asynchronous operation* is one that allows the computer to “move on” to other tasks while waiting for the asynchronous operation to complete. Asynchronous programming means that time-consuming operations don’t have to bring everything else in our programs to a halt.

There are countless examples of asynchronicity in our everyday lives. Cleaning our house, for example, involves asynchronous operations such as a dishwasher washing our dishes or a washing machine washing our clothes. While we wait on the completion of those operations, we're free to do other chores.

Similarly, web development makes use of asynchronous operations. Operations like making a network request or querying a database can be time-consuming, but JavaScript allows us to execute other tasks while awaiting their completion.

This lesson will teach you how modern JavaScript handles asynchronicity using the **Promise** object, introduced with ES6. Let's get started!

## JAVASCRIPT PROMISES

### What is a Promise?

Promises are objects that represent the eventual outcome of an asynchronous operation. A **Promise** object can be in one of three states:

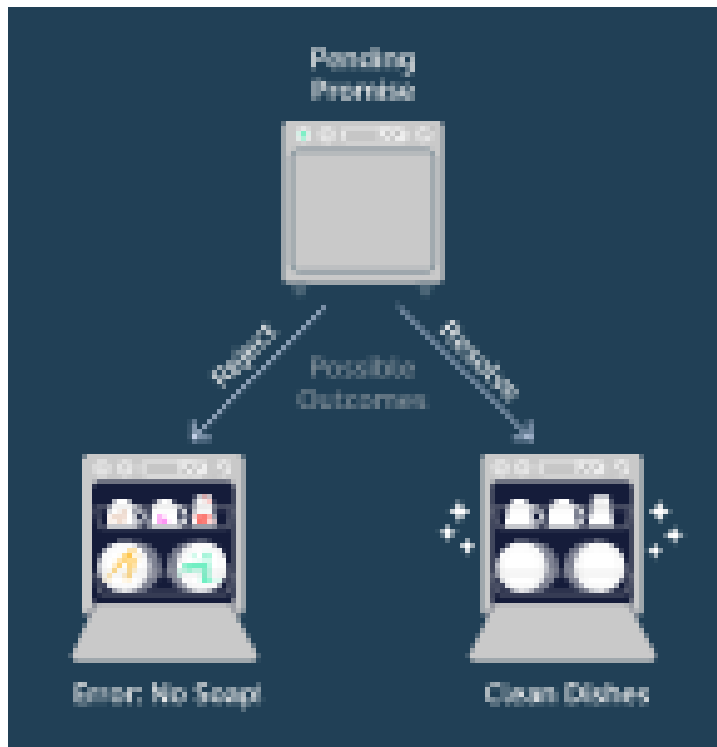
- **Pending:** The initial state— the operation has not completed yet.
- **Fulfilled:** The operation has completed successfully and the promise now has a *resolved value*. For example, a request's promise might resolve with a JSON object as its value.
- **Rejected:** The operation has failed and the promise has a reason for the failure. This reason is usually an **Error** of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected. Let's think of a dishwasher as having the states of a promise:

- **Pending:** The dishwasher is running but has not completed the washing cycle.
- **Fulfilled:** The dishwasher has completed the washing cycle and is full of clean dishes.
- **Rejected:** The dishwasher encountered a problem (it didn't receive soap!) and returns unclear dishes.

If our dishwashing promise is fulfilled, we'll be able to perform related tasks, such as unloading the clean dishes from the dishwasher. If it's rejected, we can take alternate steps, such as running it again with soap or washing the dishes by hand.

All promises eventually settle, enabling us to write logic for what to do if the promise fulfills or if it rejects.



## JAVASCRIPT PROMISES

### Constructing a Promise Object

Let's construct a promise! To create a new Promise object, we use the `new` keyword and the Promise constructor method:

```
const executorFunction = (resolve, reject) => { };  
const myFirstPromise = new Promise(executorFunction);
```

The Promise constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the `resolve()` and `reject()` functions. The `resolve()` and `reject()` functions aren't defined by the programmer. When the Promise constructor runs, JavaScript will pass its own `resolve()` and `reject()` functions into the executor function.

- `resolve` is a function with one argument. Under the hood, if invoked, `resolve()` will change the promise's status from pending to fulfilled, and the promise's resolved value will be set to the argument passed into `resolve()`.
- `reject` is a function that takes a reason or error as an argument. Under the hood, if invoked, `reject()` will change the promise's status from pending to rejected, and the promise's rejection reason will be set to the argument passed into `reject()`.

Let's look at an example executor function in a Promise constructor:

```
const executorFunction = (resolve, reject) => {
  if (someCondition) {
    resolve('I resolved!');
  } else {
    reject('I rejected!');
  }
}
const myFirstPromise = new Promise(executorFunction);
```

Let's break down what's happening above:

- We declare a variable `myFirstPromise`
- `myFirstPromise` is constructed using `new Promise()` which is the `Promise` constructor method.
- `executorFunction()` is passed to the constructor and has two functions as parameters: `resolve` and `reject`.
- If `someCondition` evaluates to `true`, we invoke `resolve()` with the string `'I resolved!'`
- If not, we invoke `reject()` with the string `'I rejected!'`

In our example, `myFirstPromise` resolves or rejects based on a simple condition, but, in practice, promises settle based on the results of asynchronous operations. For example, a database request may fulfill with the data from a query or reject with an error thrown. In this exercise, we'll construct promises which resolve synchronously to more easily understand how they work.

## JAVASCRIPT PROMISES

### The Node `setTimeout()` Function

Knowing how to construct a promise is useful, but most of the time, knowing how to *consume*, or use, promises will be key. Rather than constructing promises, you'll be handling `Promise` objects returned to you as the result of an asynchronous operation. These promises will start off pending but settle eventually.

Moving forward, we'll be simulating this by providing you with functions that return promises which settle after some time. To accomplish this, we'll be using `setTimeout()`. `setTimeout()` is a Node API (a comparable API is provided by web browsers) that uses callback functions to schedule tasks to be performed after a delay. `setTimeout()` has two parameters: a callback function and a delay in milliseconds.

```
const delayedHello = () => {
  console.log('Hi! This is an asynchronous greeting!');
};
```

```
setTimeout(delayedHello, 2000);
```

Here, we invoke `setTimeout()` with the callback function `delayedHello()` and `2000`. In at least two seconds `delayedHello()` will be invoked. But why is it "at least" two seconds and not exactly two seconds?

This delay is performed asynchronously—the rest of our program won't stop executing during the delay. Asynchronous JavaScript uses something called *the event-loop*. After two seconds, `delayedHello()` is added to a line of code waiting to be run. Before it can run, any synchronous code from the program will run. Next, any code in front of it in the line will run. This means it might be more than two seconds before `delayedHello()` is actually executed.

Let's look at how we'll be using `setTimeout()` to construct asynchronous promises:

```
const returnPromiseFunction = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {resolve('I resolved!')}, 1000);  
  });  
};
```

```
const prom = returnPromiseFunction();
```

In the example code, we invoked `returnPromiseFunction()` which returned a promise. We assigned that promise to the variable `prom`. Similar to the asynchronous promises you may encounter in production, `prom` will initially have a status of pending.

Let's explore `setTimeout()` a bit more.

## JAVASCRIPT PROMISES

### Consuming Promises

The initial state of an asynchronous promise is `pending`, but we have a guarantee that it will settle. How do we tell the computer what should happen then? Promise objects come with an aptly named `.then()` method. It allows us to say, "I have a promise, when it settles, then here's what I want to happen..."

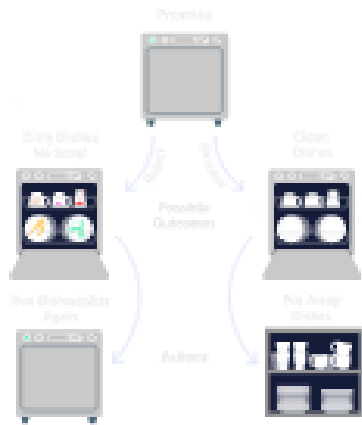
In the case of our dishwasher promise, the dishwasher will run then:

- If our promise rejects, this means we have dirty dishes, and we'll add soap and run the dishwasher again.
- If our promise fulfills, this means we have clean dishes, and we'll put the dishes away.

`.then()` is a higher-order function— it takes two callback functions as arguments. We refer to these callbacks as *handlers*. When the promise settles, the appropriate handler will be invoked with that settled value.

- The first handler, sometimes called `onFulfilled`, is a *success handler*, and it should contain the logic for the promise resolving.
- The second handler, sometimes called `onRejected`, is a *failure handler*, and it should contain the logic for the promise rejecting.

We can invoke `.then()` with one, both, or neither handler! This allows for flexibility, but it can also make for tricky debugging. If the appropriate handler is not provided, instead of throwing an error, `.then()` will just return a promise with the same settled value as the promise it was called on. One important feature of `.then()` is that it always returns a promise. We'll return to this in more detail in a later exercise and explore why it's so important.



## JAVASCRIPT PROMISES

### Success and Failure Callback Functions

To handle a “successful” promise, or a promise that resolved, we invoke `.then()` on the promise, passing in a success handler callback function:

```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});
```

```
const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};
```

```
prom.then(handleSuccess); // Prints: 'Yay!'
```

Let’s break down what’s happening in the example code:

- `prom` is a promise which will resolve to 'Yay!'.
- We define a function, `handleSuccess()`, which prints the argument passed to it.
- We invoke `prom`’s `.then()` function passing in our `handleSuccess()` function.
- Since `prom` resolves, `handleSuccess()` is invoked with `prom`’s resolved value, 'Yay', so 'Yay' is logged to the console.

With typical promise consumption, we won’t know whether a promise will resolve or reject, so we’ll need to provide the logic for either case. We can pass both a success callback and a failure callback to `.then()`.

```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve('Yay!');
  } else {
    reject('Ohhh nooooo!');
  }
});
```

```
const handleSuccess = (resolvedValue) => {
```

```
    console.log(resolvedValue);
  };

  const handleFailure = (rejectionReason) => {
    console.log(rejectionReason);
  };

  prom.then(handleSuccess, handleFailure);
```

Let's break down what's happening in the example code:

- `prom` is a promise which will randomly either resolve with 'Yay!' or reject with 'Ohhh noooo!'.
- We pass two handler functions to `.then()`. The first will be invoked with 'Yay!' if the promise resolves, and the second will be invoked with 'Ohhh noooo!' if the promise rejects.

*Note: The success callback is sometimes called the “success handler function” or the `onFulfilled` function. The failure callback is sometimes called the “failure handler function” or the `onRejected` function.*

Let's write some success and failure callbacks!

## JAVASCRIPT PROMISES

### Using `catch()` with Promises

One way to write cleaner code is to follow a principle called *separation of concerns*. Separation of concerns means organizing code into distinct sections each handling a specific task. It enables us to quickly navigate our code and know where to look if something isn't working.

Remember, `.then()` will return a promise with the same settled value as the promise it was called on if no appropriate handler was provided. This implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one `.then()`, we can chain a second `.then()` with a failure handler to a first `.then()` with a success handler and both cases will be handled.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .then(null, (rejectionReason) => {
    console.log(rejectionReason);
  });
```

Since JavaScript doesn't mind whitespace, we follow a common convention of putting each part of this chain on a new line to make it easier to read. To create even more readable code, we can use a different promise function: `.catch()`.

The `.catch()` function takes only one argument, `onRejected`. In the case of a rejected promise, this failure handler will be invoked with the reason for rejection.



Using `.catch()` accomplishes the same thing as using a `.then()` with only a failure handler.

Let's look at an example using `.catch()`:

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let's break down what's happening in the example code:

- `prom` is a promise which randomly either resolves with 'Yay!' or rejects with 'Ohhh noooo!'.
- We pass a success handler to `.then()` and a failure handler to `.catch()`.
- If the promise resolves, `.then()`'s success handler will be invoked with 'Yay!'.
- If the promise rejects, `.then()` will return a promise with the same rejection reason as the original promise and `.catch()`'s failure handler will be invoked with that rejection reason.

Let's practice writing `.catch()` functions.

## JAVASCRIPT PROMISES

### Chaining Multiple Promises

One common pattern we'll see with asynchronous programming is multiple operations which depend on each other to execute or that must be executed in a certain order. We might make one request to a database and use the data returned to us to make another request and so on! Let's illustrate this with another cleaning example, washing clothes:

We take our dirty clothes and put them in the washing machine. If the clothes are cleaned, then we'll want to put them in the dryer. After the dryer runs, if the clothes are dry, then we can fold them and put them away.

This process of chaining promises together is called *composition*. Promises are designed with composition in mind! Here's a simple promise chain in code:

```
firstPromiseFunction()
  .then((firstResolveVal) => {
    return secondPromiseFunction(firstResolveVal);
  })
  .then((secondResolveVal) => {
    console.log(secondResolveVal);
  });
```

Let's break down what's happening in the example:

- We invoke a function `firstPromiseFunction()` which returns a promise.
- We invoke `.then()` with an anonymous function as the success handler.

- Inside the success handler we return a new promise— the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- We invoke a second `.then()` to handle the logic for the second promise settling.
- Inside that `.then()`, we have a success handler which will log the second promise's resolved value to the console.

In order for our chain to work properly, we had to return the promise `secondPromiseFunction(firstResolveVal)`. This ensured that the return value of the first `.then()` was our second promise rather than the default return of a new promise with the same settled value as the initial.

Let's write some promise chains!

## JAVASCRIPT PROMISES

### Avoiding Common Mistakes

Promise composition allows for much more readable code than the nested callback syntax that preceded it. However, it can still be easy to make mistakes. In this exercise, we'll go over two common mistakes with promise composition.

**Mistake 1: Nesting promises instead of chaining them.**

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    return returnsSecondValue(firstResolveVal)
      .then((secondResolveVal) => {
        console.log(secondResolveVal);
      })
  })
```

Let's break down what's happening in the above code:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we invoke `returnsSecondValue()` with `firstResolveVal` which will return a new promise.
- We invoke a second `.then()` to handle the logic for the second promise settling all inside the first `then()`!
- Inside that second `.then()`, we have a success handler which will log the second promise's resolved value to the console.

Instead of having a clean chain of promises, we've nested the logic for one inside the logic of the other. Imagine if we were handling five or ten promises!

**Mistake 2: Forgetting to return a promise.**

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    returnsSecondValue(firstResolveVal)
```

```
})  
.then((someVal) => {  
  console.log(someVal);  
})
```

Let's break down what's happening in the example:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we create our second promise, but we forget to return it!
- We invoke a second `.then()`. It's supposed to handle the logic for the second promise, but since we didn't return, this `.then()` is invoked on a promise with the same settled value as the original promise!

Since forgetting to return our promise won't throw an error, this can be a really tricky thing to debug!

## JAVASCRIPT PROMISES

### Using `Promise.all()`

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we're dealing with multiple promises, but we don't care about the order? Let's think in terms of cleaning again.

For us to consider our house clean, we need our clothes to dry, our trash bins emptied, and the dishwasher to run. We need all of these tasks to complete but not in any particular order. Furthermore, since they're all getting done asynchronously, they should really all be happening at the same time!

To maximize efficiency we should use *concurrency*, multiple asynchronous operations happening together. With promises, we can do this with the function `Promise.all()`.

`Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

- If every promise in the argument array resolves, the single promise returned from `Promise.all()` will resolve with an array containing the resolve value from each promise in the argument array.
- If any promise from the argument array rejects, the single promise returned from `Promise.all()` will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Let's look at a code example:

```
let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(),  
returnsPromThree()]);
```

```
myPromises  
  .then((arrayOfValues) => {  
    console.log(arrayOfValues);
```

```
})  
.catch((rejectionReason) => {  
  console.log(rejectionReason);  
});
```

Let's break down what's happening:

- We declare `myPromises` assigned to invoking `Promise.all()`.
- We invoke `Promise.all()` with an array of three promises—the returned values from functions.
- We invoke `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
- We invoke `.catch()` with a failure handler which will print the first rejection message if any promise rejects.

## JAVASCRIPT PROMISES

### Review

Awesome job! Promises are a difficult concept even for experienced developers, so pat yourself on the back. You've learned a ton about asynchronous JavaScript and promises. Let's review:

- Promises are JavaScript objects that represent the eventual result of an asynchronous operation.
- Promises can be in one of three states: pending, resolved, or rejected.
- A promise is settled if it is either resolved or rejected.
- We construct a promise by using the `new` keyword and passing an executor function to the `Promise` constructor method.
- `setTimeout()` is a Node function which delays the execution of a callback function using the event-loop.
- We use `.then()` with a success handler callback containing the logic for what should happen if a promise resolves.
- We use `.catch()` with a failure handler callback containing the logic for what should happen if a promise rejects.
- Promise composition enables us to write complex, asynchronous code that's still readable. We do this by chaining multiple `.then()`'s and `.catch()`'s.
- To use promise composition correctly, we have to remember to return promises constructed within a `.then()`.
- We should chain multiple promises rather than nesting them.
- To take advantage of concurrency, we can use `Promise.all()`.

---

---

## ASYNC AWAIT

### Introduction

Often in web development, we need to handle asynchronous actions—actions we can wait on while moving on to other tasks. We make requests to networks, databases, or any number of similar operations. JavaScript is non-blocking: instead of stopping the execution of code while it waits, JavaScript uses an [event-loop](#) which allows it to

efficiently execute other tasks while it awaits the completion of these asynchronous actions.

Originally, JavaScript used callback functions to handle asynchronous actions. The problem with callbacks is that they encourage complexly nested code which quickly becomes difficult to read, debug, and scale. With ES6, JavaScript integrated native [promises](#) which allow us to write significantly more readable code. JavaScript is continually improving, and [ES8](#) provides a new syntax for handling our asynchronous action, `async...await`. The `async...await` syntax allows us to write asynchronous code that reads similarly to traditional synchronous, imperative programs.

The `async...await` syntax is [syntactic sugar](#)— it doesn't introduce new functionality into the language, but rather introduces a new syntax for using promises and [generators](#). Both of these were already built in to the language. Despite this, `async...await` powerfully improves the readability and scalability of our code. Let's learn how to use it!

## ASYNC AWAIT

### The async Keyword

The `async` keyword is used to write functions that handle asynchronous actions. We wrap our asynchronous logic inside a function prepended with the `async` keyword. Then, we invoke that function.

```
async function myFunc() {  
  // Function body here  
};
```

```
myFunc();
```

We'll be using `async` function declarations throughout this lesson, but we can also create `async` function expressions:

```
const myFunc = async () => {  
  // Function body here  
};
```

```
myFunc();
```

`async` functions always return a promise. This means we can use traditional promise syntax, like `.then()` and `.catch` with our `async` functions. An `async` function will return in one of three ways:

- If there's nothing returned from the function, it will return a promise with a resolved value of `undefined`.
- If there's a non-promise value returned from the function, it will return a promise resolved to that value.
- If a promise is returned from the function, it will simply return that promise

```
async function fivePromise() {  
  return 5;  
}
```

```
fivePromise()  
.then(resolvedValue => {  
  console.log(resolvedValue);  
}) // Prints 5
```

In the example above, even though we return 5 inside the function body, what's actually returned when we invoke `fivePromise()` is a promise with a resolved value of 5.

Let's write an async function

## ASYNC AWAIT

### The await Operator

In the last exercise, we covered the `async` keyword. By itself, it doesn't do much; `async` functions are almost always used with the additional keyword `await` inside the function body.

The `await` keyword can only be used inside an `async` function. `await` is an operator: it returns the resolved value of a promise. Since promises resolve in an indeterminate amount of time, `await` halts, or pauses, the execution of our `async` function until a given promise is resolved.

In most situations, we're dealing with promises that were returned from functions. Generally, these functions are through a library, and, in this lesson, we'll be providing them. We can `await` the resolution of the promise it returns inside an `async` function. In the example below, `myPromise()` is a function that returns a promise which will resolve to the string "I am resolved now!".

```
async function asyncFuncExample(){  
  let resolvedValue = await myPromise();  
  console.log(resolvedValue);  
}
```

```
asyncFuncExample(); // Prints: I am resolved now!
```

Within our `async` function, `asyncFuncExample()`, we use `await` to halt our execution until `myPromise()` is resolved and assign its resolved value to the variable `resolvedValue`. Then we log `resolvedValue` to the console. We're able to handle the logic for a promise in a way that reads like synchronous code.

## ASYNC AWAIT

### Writing async Functions

We've seen that the `await` keyword halts the execution of an `async` function until a promise is no longer pending. Don't forget the `await` keyword! It may seem obvious, but this can be a tricky mistake to catch because our function will still run—it just won't have the desired results.

We're going to explore this using the following function, which returns a promise that resolves to 'Yay, I resolved!' after a 1 second delay:

```
let myPromise = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {
```

```

    resolve('Yay, I resolved!')
  }, 1000);
});
}

```

Now we'll write two async functions which invoke myPromise():

```

async function noAwait() {
  let value = myPromise();
  console.log(value);
}

```

```

async function yesAwait() {
  let value = await myPromise();
  console.log(value);
}

```

```

noAwait(); // Prints: Promise { <pending> }
yesAwait(); // Prints: Yay, I resolved!

```

In the first async function, noAwait(), we left off the await keyword before myPromise(). In the second, yesAwait(), we included it. The noAwait() function logs Promise { <pending> } to the console. Without the await keyword, the function execution wasn't paused. The console.log() on the following line was executed before the promise had resolved.

Remember that the await operator returns the resolved value of a promise. When used properly in yesAwait(), the variable value was assigned the resolved value of the myPromise() promise, whereas in noAwait(), value was assigned the promise object itself.

## ASYNCAWAIT

### Handling Dependent Promises

The true beauty of async...await is when we have a series of asynchronous actions which depend on one another. For example, we may make a network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from the database. With native promise syntax, we use a chain of .then() functions making sure to return correctly each one:

```

function nativePromiseVersion() {
  returnsFirstPromise()
    .then((firstValue) => {
      console.log(firstValue);
      return returnsSecondPromise(firstValue);
    })
    .then((secondValue) => {
      console.log(secondValue);
    });
}

```

Let's break down what's happening in the nativePromiseVersion() function:

- Within our function we use two functions which return promises: `returnsFirstPromise()` and `returnsSecondPromise()`.
- We invoke `returnsFirstPromise()` and ensure that the first promise resolved by using `.then()`.
- In the callback of our first `.then()`, we log the resolved value of the first promise, `firstValue`, and then return `returnsSecondPromise(firstValue)`.
- We use another `.then()` to print the second promise's resolved value to the console.

Here's how we'd write an async function to accomplish the same thing:

```
async function asyncAwaitVersion() {
  let firstValue = await returnsFirstPromise();
  console.log(firstValue);
  let secondValue = await returnsSecondPromise(firstValue);
  console.log(secondValue);
}
```

Let's break down what's happening in our `asyncAwaitVersion()` function:

- We mark our function as `async`.
- Inside our function, we create a variable `firstValue` assigned `await returnsFirstPromise()`. This means `firstValue` is assigned the resolved value of the awaited promise.
- Next, we log `firstValue` to the console.
- Then, we create a variable `secondValue` assigned to `await returnsSecondPromise(firstValue)`. Therefore, `secondValue` is assigned this promise's resolved value.
- Finally, we log `secondValue` to the console.

Though using the `async...await` syntax can save us some typing, the length reduction isn't the main point. Given the two versions of the function, the `async...await` version more closely resembles synchronous code, which helps developers maintain and debug their code. The `async...await` syntax also makes it easy to store and refer to resolved values from promises further back in our chain which is a much more difficult task with native promise syntax. Let's create some async functions with multiple await statements!

## ASYNC AWAIT

### Handling Errors

When `.catch()` is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debugging challenging.

With `async...await`, we use `try...catch` statements for error handling. By using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also catch both synchronous and asynchronous errors. This makes for easier debugging!



```

async function usingTryCatch() {
  try {
    let resolveValue = await asyncFunction('thing that will fail');
    let secondValue = await secondAsyncFunction(resolveValue);
  } catch (err) {
    // Catches any errors in the try block
    console.log(err);
  }
}

```

usingTryCatch();

Remember, since async functions return promises we can still use native promise's .catch() with an async function

```

async function usingPromiseCatch() {
  let resolveValue = await asyncFunction('thing that will fail');
}

```

```

let rejectedPromise = usingPromiseCatch();
rejectedPromise.catch((rejectValue) => {
  console.log(rejectValue);
})

```

This is sometimes used in the global scope to catch final errors in complex code.

## ASYNCR AWAIT

### Handling Independent Promises

Remember that await halts the execution of our async function. This allows us to conveniently write synchronous-style code to handle dependent promises. But what if our async function contains multiple promises which are not dependent on the results of one another to execute?

```

async function waiting() {
  const firstValue = await firstAsyncThing();
  const secondValue = await secondAsyncThing();
  console.log(firstValue, secondValue);
}

```

```

async function concurrent() {
  const firstPromise = firstAsyncThing();
  const secondPromise = secondAsyncThing();
  console.log(await firstPromise, await secondPromise);
}

```

In the waiting() function, we pause our function until the first promise resolves, then we construct the second promise. Once that resolves, we print both resolved values to the console.

In our concurrent() function, both promises are constructed without using await. We then await each of their resolutions to print them to the console.

With our `concurrent()` function both promises' asynchronous operations can be run simultaneously. If possible, we want to get started on each asynchronous operation as soon as possible! Within our `async` functions we should still take advantage of *concurrency*, the ability to perform asynchronous actions at the same time.

Note: if we have multiple truly independent promises that we would like to execute fully in parallel, we must use individual `.then()` functions and avoid halting our execution with `await`.

## ASYNC AWAIT

### Await Promise.all()

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously is to await a `Promise.all()`.

We can pass an array of promises as the argument to `Promise.all()`, and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved. This promise's resolve value will be an array containing the resolved values of each promise from the argument array.

```
async function asyncPromAll() {
  const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(),
  asyncTask4()]);
  for (let i = 0; i < resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

In our above example, we await the resolution of a `Promise.all()`. This `Promise.all()` was invoked with an argument array containing four promises (returned from required-in functions). Next, we loop through our `resultArray`, and log each item to the console. The first element in `resultArray` is the resolved value of the `asyncTask1()` promise, the second is the value of the `asyncTask2()` promise, and so on.

`Promise.all()` allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. `Promise.all()` also has the benefit of *failing fast*, meaning it won't wait for the rest of the asynchronous actions to complete once any one has rejected. As soon as the first promise in the array rejects, the promise returned from `Promise.all()` will reject with that reason. As it was when working with native promises, `Promise.all()` is a good choice if multiple asynchronous tasks are all required, but none must wait for any other before executing.

## ASYNC AWAIT

### Review

Awesome work getting the hang of the `async...await` syntax! Let's review what you've learned:

- `async...await` is syntactic sugar built on native JavaScript promises and generators.
- We declare an `async` function with the keyword `async`.

- Inside an async function we use the await operator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending .
  - await returns the resolved value of the awaited promise.
  - We can write multiple await statements to produce code that reads like synchronous code.
  - We use try...catch statements within our async functions for error handling.
  - We should still take advantage of concurrency by writing async functions that allow asynchronous actions to happen in concurrently whenever possible.
- .....

## HTTP Requests

Understand the basics of how your web browser communicates with the internet.

### Background:

This page is generated by a web of HTML, CSS, and Javascript, sent to you by Codecademy via the internet. The internet is made up of a bunch of resources hosted on different servers. The term “resource” corresponds to any entity on the web, including HTML files, stylesheets, images, videos, and scripts. To access content on the internet, your browser must ask these servers for the resources it wants and then display these resources to you. This protocol of requests and responses enables you to view this page in your browser.

This article focuses on one fundamental part of how the internet functions: HTTP.

### What is HTTP?

HTTP stands for Hypertext Transfer Protocol and is used to structure requests and responses over the internet. HTTP requires data to be transferred from one point to another over the network.

The transfer of resources happens using TCP (Transmission Control Protocol). In viewing this webpage, TCP manages the channels between your browser and the server (in this case, codecademy.com). TCP is used to manage many types of internet connections in which one computer or device wants to send something to another. HTTP is the command language that the devices on both sides of the connection must follow in order to communicate.

### HTTP & TCP: How it Works

When you type an address such as [www.codecademy.com](http://www.codecademy.com) into your browser, you are commanding it to open a TCP channel to the server that responds to that URL (or

Uniform Resource Locator, which you can read more about on Wikipedia). A URL is like your home address or phone number because it describes how to reach you.

In this situation, your computer, which is making the request, is called the client. The URL you are requesting is the address that belongs to the server.

Once the TCP connection is established, the client sends a HTTP GET request to the server to retrieve the webpage it should display. After the server has sent the response, it closes the TCP connection. If you open the website in your browser again, or if your browser automatically requests something from the server, a new connection is opened which follows the same process described above. GET requests are one kind of HTTP method a client can call. You can learn more about the other common ones (POST, PUT and DELETE) in this article.

Let's explore an example of how GET requests (the most common type of request) are used to help your computer (the client) access resources on the web.

Suppose you want to check out the latest course offerings from <http://codecademy.com>. After you type the URL into your browser, your browser will extract the http part and recognize that it is the name of the network protocol to use. Then, it takes the domain name from the URL, in this case "codecademy.com", and asks the internet Domain Name Server to return an Internet Protocol (IP) address.

Now the client knows the destination's IP address. It then opens a connection to the server at that address, using the http protocol as specified. It will initiate a GET request to the server which contains the IP address of the host and optionally a data payload. The GET request contains the following text:

**GET / HTTP/1.1**

**Host: www.codecademy.com**

This identifies the type of request, the path on [www.codecademy.com](http://www.codecademy.com) (in this case, "/") and the protocol "HTTP/1.1." HTTP/1.1 is a revision of the first HTTP, which is now called HTTP/1.0. In HTTP/1.0, every resource request requires a separate connection to the server. HTTP/1.1 uses one connection more than once, so that additional content (like images or stylesheets) is retrieved even after the page has been retrieved. As a result, requests using HTTP/1.1 have less delay than those using HTTP/1.0.

The second line of the request contains the address of the server which is "www.codecademy.com". There may be additional lines as well depending on what data your browser chooses to send.

If the server is able to locate the path requested, the server might respond with the header:

**HTTP/1.1 200 OK**

**Content-Type: text/html**

This header is followed by the content requested, which in this case is the information needed to render [www.codecademy.com](http://www.codecademy.com).

The first line of the header, **HTTP/1.1 200 OK**, is confirmation that the server understands the protocol that the client wants to communicate with (HTTP/1.1), and an HTTP status code signifying that the resource was found on the server. The second line, **Content-Type: text/html**, shows the type of content that it will be sending to the client.

If the server is not able to locate the path requested by the client, it will respond with the header:

**HTTP/1.1 404 NOT FOUND**

In this case, the server identifies that it understands the HTTP protocol, but the **404 NOT FOUND** status code signifies that the specific piece of content requested was not found. This might happen if the content was moved or if you typed in the URL path incorrectly or if the page was removed. You can read more about the **404** status code, commonly called a **404 error**, [here](#).

### **An Analogy:**

It can be tricky to understand how HTTP functions because it's difficult to examine what your browser is actually doing. (And perhaps also because we explained it using acronyms that may be new to you.) Let's review what we learned by using an analogy that could be more familiar to you.

Imagine the internet is a town. You are a client and your address determines where you can be reached. Businesses in town, such as [Codecademy.com](http://Codecademy.com), serve requests that are sent to them. The other houses are filled with other clients like you that are making requests and expecting responses from these businesses in town. This town also has a crazy fast mail service, an army of mail delivery staff that can travel on trains that move at the speed of light.

Suppose you want to read the morning newspaper. In order to retrieve it, you write down what you need in a language called HTTP and ask your local mail delivery staff

agent to retrieve it from a specific business. The mail delivery person agrees and builds a railroad track (connection) between your house and the business nearly instantly, and rides the train car labeled “TCP” to the address of the business you provided.

Upon arriving at the business, she asks the first of several free employees ready to fulfill the request. The employee searches for the page of the newspaper that you requested but cannot find it and communicates that back to the mail delivery person.

The mail delivery person returns on the light speed train, ripping up the tracks on the way back, and tells you that there was a problem “404 Not Found.” After you check the spelling of what you had written, you realize that you misspelled the newspaper title. You correct it and provide the corrected title to the mail delivery person.

This time the mail delivery person is able to retrieve it from the business. You can now read your newspaper in peace until you decide you want to read the next page, at which point, you would make another request and give it to the mail delivery person.

## What is HTTPS?

Since your HTTP request can be read by anyone at certain network junctures, it might not be a good idea to deliver information such as your credit card or password using this protocol. Fortunately, many servers support HTTPS, short for HTTP Secure, which allows you to encrypt data that you send and receive. You can read more about HTTPS on Wikipedia.

HTTPS is important to use when passing sensitive or personal information to and from websites. However, it is up to the businesses maintaining the servers to set it up. In order to support HTTPS, the business must apply for a certificate from a Certificate Authority.

## Introduction to Web APIs

Learn what APIs are and how they’re useful tools for web development.

### What are APIs?

*An Application Programming Interface (API)* is a software tool that makes it easier for developers to interact with another application to use some of that application’s functionality. Like tools in the physical world, APIs are built to solve specific, repeated, and potentially complex problems.

Imagine we needed to twist a screw into a piece of wood. We could try to take the screw and twist it in with our fingers, but that would be difficult and inefficient. Twisting one screw would be hard enough — having to do it multiple times would be near impossible! Instead, we could make the task much easier by using a tool that someone else created, in this case, a screwdriver! When used properly, the

screwdriver can be used for one specific type of screw and can be reused for the same job over and over again!

Relating this analogy back to APIs, we probably don't want to have to write up the same code for the same problems over and over again. If the right API exists to solve our problem, we could make it easier on ourselves and use the API instead. But, before we fully commit to using an API, we should go over some important considerations!

## Types of APIs

There are two main categories of web APIs: browser APIs and 3rd party APIs.

*Browser APIs* are specific to writing code related to browsers and give developers access to information that the browser can also access. One example is the [geolocation API](#) which allows the program to know where a user is when accessing our app. This specific API requires that the user grant permissions to the browser to access their geolocation information. There are also browser APIs for audio, cryptography, VR, and much more. To see a comprehensive list of browser APIs and how to interact with them, check out [MDN's documentation of web APIs](#).

*Third-party APIs* are apps that provide some type of functionality or information from a third-party, usually a company. For example, we could use the [OpenWeather API](#) to get in-depth information about the weather in an area, forecasts, historical weather data, and more! On our own, we wouldn't even have access to this data and we would certainly not want to write up this code ourselves just for one app!

## Requesting Information from a Third-party API

Each API has a specific structure and protocol that we have to follow in order to gain access to its functionality.

## Rules and Requirements

Organizations that maintain third-party APIs often set rules and requirements for how developers can interact with their APIs. For OpenWeather, we need to sign up for an account and generate a special token called an *API key* that grants our account the ability to make API requests. These API keys are unique to individual accounts and should be kept secret. OpenWeather provides some free functionality and some paid functionality. So before committing to a third-party API, check out their specifications which can often be found in the API documentation. Here's [OpenWeather's documentation](#) to look over as an example.

## Making Requests

Some of an API's specifications relate to making a *request* for data. These specifications could include more parameters for specific information or the inclusion of an API key. For example, when using the OpenWeather API, we need to provide more information to search for weather information — such information could include: the name of a city, time of day, the type of forecast, etc. These specifications for making a request can also be found in the API documentation.

## Response Data

After we make a successful API request, the API sends back data. Many APIs format their data using [JavaScript Object Notation \(JSON\)](#) which looks like a JavaScript object. Here's a quick example of what JSON data might look like:

```
{
  "temperature" : {
    "celcius" : 25,
  },
  "city": "chicago",
}
```

It's then up to us how to determine how to *consume*, or use, this information in our apps. If we got back that sample JSON above, we could parse out the temperature information and display it on our app.

## Review

Congrats, we've now gone through the basics of web APIs! Here's a quick recap of what we covered:

- With web APIs, we have a tool that we can use to access the functionality and data of another application.
- There are two main types of APIs: browser and third-party.
  - Browser APIs require specific syntax and permissions.
  - Third-party APIs have their own rules and requirements set by the organizations that maintain them.
- When making a request to API, we might have to supply more details about what information we want.
- If we get a successful response, we still have to decide how to consume the response data.

That's a lot to take in, but knowing that these tools exist opens up more possibilities for what we can do with our apps!

.....

## What is REST?

Learn about the REST (Representational State Transfer) paradigm and how rest architecture streamlines communication between web components.

## REpresentational State Transfer

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called RESTful systems, are characterized by how they are stateless and separate the concerns of client and server. We will go into what these terms mean and why they are beneficial characteristics for services on the Web. Pay close attention: If you're looking for a career in tech, you may be asked to define rest during an interview.



## Separation of Client and Server

In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we improve the flexibility of the interface across platforms and improve scalability by simplifying the server components. Additionally, the separation allows each component the ability to evolve independently.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

## Statelessness

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of *resources*, rather than *commands*. Resources are the nouns of the Web - they describe any object, document, or *thing* that you may need to store or send to other services.

Because REST systems interact through standard operations on resources, they do not rely on the implementation of interfaces.

These constraints help RESTful applications achieve reliability, quick performance, and scalability, as components that can be managed, updated, and reused without affecting the system as a whole, even during operation of the system.

Now, we'll explore how the communication between the client and server actually happens when we are implementing a RESTful interface.

## Communication between Client and Server

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests. Let's take a look at the standard ways to make requests and send responses.

## Making Requests

REST requires that a client make a request to the server in order to retrieve or modify data on the server. A request generally consists of:

- an HTTP verb, which defines what kind of operation to perform
- a *header*, which allows the client to pass along information about the request
- a path to a resource
- an optional message body containing data

## HTTP Verbs

There are 4 basic HTTP verbs we use in requests to interact with resources in a REST system:

- GET — retrieve a specific resource (by id) or a collection of resources
- POST — create a new resource
- PUT — update a specific resource (by id)
- DELETE — remove a specific resource by id

You can learn more about these HTTP verbs in the following Codecademy article:

- [What is CRUD?](#)

### Headers and Accept parameters

In the header of the request, the client sends the type of content that it is able to receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client. The options for types of content are MIME Types (or Multipurpose Internet Mail Extensions, which you can read more about in the [MDN Web Docs](#)).

MIME Types, used to specify the content types in the Accept field, consist of a type and a subtype. They are separated by a slash (/).

For example, a text file containing HTML would be specified with the type text/html. If this text file contained CSS instead, it would be specified as text/css. A generic text file would be denoted as text/plain. This default value, text/plain, is not a catch-all, however. If a client is expecting text/css and receives text/plain, it will not be able to recognize the content.

Other types and commonly used subtypes:

- image — image/png, image/jpeg, image/gif
- audio — audio/wav, audio/mpeg
- video — video/mp4, video/ogg
- application — application/json, application/pdf, application/xml, application/octet-stream

For example, a client accessing a resource with id 23 in an articles resource on a server might send a GET request like this:

GET /articles/23

Accept: text/html, application/xhtml

The Accept header field in this case is saying that the client will accept the content in text/html or application/xhtml.

### Paths

Requests must contain a path to a resource that the operation should be performed on. In RESTful APIs, paths should be designed to help the client know what is going on.

Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.

A path like `fashionboutique.com/customers/223/orders/12` is clear in what it points to, even if you've never seen this specific path before, because it is hierarchical and descriptive. We can see that we are accessing the order with id 12 for the customer with id 223.

Paths should contain the information necessary to locate a resource with the degree of specificity needed. When referring to a list or collection of resources, it is not always necessary to add an id. For example, a POST request to the `fashionboutique.com/customers` path would not need an extra identifier, as the server will generate an id for the new object.

If we are trying to access a single resource, we would need to append an id to the path. For example: GET `fashionboutique.com/customers/:id` — retrieves the item in the customers resource with the id specified. DELETE `fashionboutique.com/customers/:id` — deletes the item in the customers resource with the id specified.

## Sending Responses

### Content Types

In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The content-type that the server sends back in the response should be one of the options that the client specified in the accept field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

```
GET /articles/23 HTTP/1.1
Accept: text/html, application/xhtml
```

The server might send back the content with the response header:

```
HTTP/1.1 200 (OK)
Content-Type: text/html
```

This would signify that the content requested is being returned in the response body with a content-type of `text/html`, which the client said it would be able to accept.

### Response Codes

Responses from the server contain status codes to alert the client to information about the success of the operation. As a developer, you do not need to know every status code (there are [many](#) of them), but you should know the most common ones and how they are used:

Status code	Meaning
200 (OK)	This is the standard response for successful HTTP requests.
201 (CREATED)	This is the standard response for an HTTP request that resulted in an item successfully created.

Status code	Meaning
204 (NO CONTENT)	This is the standard response for successful HTTP requests, where no content is returned in the response body.
400 (BAD REQUEST)	The request cannot be processed because of bad request syntax, except for 401 (Unauthorized) which is a client error.
403 (FORBIDDEN)	The client does not have permission to access this resource.
404 (NOT FOUND)	The resource could not be found at this time. It is possible it was deleted or never existed.
500 (INTERNAL SERVER ERROR)	The generic answer for an unexpected failure if there is no more specific error code available.

For each HTTP verb, there are expected status codes a server should return upon success:

- GET — return 200 (OK)
- POST — return 201 (CREATED)
- PUT — return 200 (OK)
- DELETE — return 204 (NO CONTENT) If the operation fails, return the most specific status code possible corresponding to the problem that was encountered.

### Examples of Requests and Responses

Let's say we have an application that allows you to view, create, edit, and delete customers and orders for a small clothing store hosted at [fashionboutique.com](http://fashionboutique.com). We could create an HTTP API that allows a client to perform these functions:

If we wanted to view all customers, the request would look like this:

GET <http://fashionboutique.com/customers>  
Accept: application/json

A possible response header would look like:

Status Code: 200 (OK)  
Content-type: application/json

followed by the customers data requested in application/json format.

Create a new customer by posting the data:

POST <http://fashionboutique.com/customers>  
Body:

```
{
  "customer": {
    "name" = "Scylla Buss",
    "email" = "scylla.buss@codecademy.org"
  }
}
```

The server then generates an id for that object and returns it back to the client, with a header like:

**201 (CREATED)**

**Content-type: application/json**

To view a single customer we *GET* it by specifying that customer's id:

**GET http://fashionboutique.com/customers/123**

**Accept: application/json**

A possible response header would look like:

**Status Code: 200 (OK)**

**Content-type: application/json**

followed by the data for the customer resource with id 23 in application/json format.

We can update that customer by *PUT* ting the new data:

**PUT http://fashionboutique.com/customers/123**

**Body:**

```
{
  "customer": {
    "name" = "Scylla Buss",
    "email" = "scyllabuss1@codecademy.com"
  }
}
```

A possible response header would have Status Code: 200 (OK), to notify the client that the item with id 123 has been modified.

We can also *DELETE* that customer by specifying its id:

**DELETE http://fashionboutique.com/customers/123**

The response would have a header containing Status Code: 204 (NO CONTENT), notifying the client that the item with id 123 has been deleted, and nothing in the body.

## Practice with REST

Let's imagine we are building a photo-collection site. We want to make an API to keep track of users, venues, and photos of those venues. This site has an index.html and a style.css. Each user has a username and a password. Each photo has a venue and an owner (i.e. the user who took the picture). Each venue has a name and street address. Can you design a REST system that would accommodate:

- storing users, photos, and venues
- accessing venues and accessing certain photos of a certain venue

Start by writing out:

- what kinds of requests we would want to make
- what responses the server should return
- what the content-type of each response should be

Possible Solution - Models

```

{
  "user": {
    "id": <Integer>,
    "username": <String>,
    "password": <String>
  }
}

{
  "photo": {
    "id": <Integer>,
    "venue_id": <Integer>,
    "author_id": <Integer>
  }
}

{
  "venue": {
    "id": <Integer>,
    "name": <String>,
    "address": <String>
  }
}

```

## Possible Solution - Requests/Responses

### GET Requests

Request- GET /index.html Accept: text/html Response- 200 (OK) Content-type: text/html

Request- GET /style.css Accept: text/css Response- 200 (OK) Content-type: text/css

Request- GET /venues Accept: application/json Response- 200 (OK) Content-type: application/json

Request- GET /venues/:id Accept: application/json Response- 200 (OK) Content-type: application/json

Request- GET /venues/:id/photos/:id Accept: application/json Response- 200 (OK) Content-type: image/png

### POST Requests

Request- POST /users Response- 201 (CREATED) Content-type: application/json

Request- POST /venues Response- 201 (CREATED) Content-type: application/json

Request- POST /venues/:id/photos Response- 201 (CREATED) Content-type: application/json

### PUT Requests

Request- PUT /users/:id Response- 200 (OK)

Request- PUT /venues/:id Response- 200 (OK)

Request- PUT /venues/:id/photos/:id Response- 200 (OK)

## DELETE Requests

Request- DELETE /venues/:id Response- 204 (NO CONTENT)

Request- DELETE /venues/:id/photos/:id Response- 204 (NO CONTENT)

## What Is JSON?

A brief guide to understanding JSON and its use cases.

### Introduction

In a world inundated with data, it is becoming more important to know how to work with a variety of data. As programmers, we need to be able to transfer our populated data structures from any language we choose to a format that is recognizable and readable by other languages and platforms. Fortunately for us, there exists such a data-exchange format.

### What is JSON?

JSON, or JavaScript Object Notation, is a popular, language-independent, standard format for storing and exchanging data. Adopted by [ECMA International](#), an industry association founded in 1961 to standardize information and communication systems, [JSON](#) has become the de facto standard that facilitates storing and sending data between all [programming languages](#).

### Common Uses of JSON

JSON is heavily used to facilitate data transfer in web applications between a client, such as a web browser, and a server. A typical example where such data transfer occurs is when you fill out a web form. The form data is converted from HTML to JavaScript objects to JSON objects and sent to a remote web server for processing. These transactions could be as simple as entering a search engine query to a multi-page job application.

When companies make their data public for other applications, like Spotify sharing its music library or Google sharing its map data, the information is formatted in JSON. This way, any application, regardless of language, can collect and parse the data.

Some of the popular web APIs that utilize JSON in data exchanges are:

- [Google Maps](#)
- [Google Auth 2.0 Authentication](#)
- [Facebook Social Graph API](#)
- [Spotify Music Web API](#)
- [LinkedIn Profile API](#)

### JSON Syntax

Since JSON is derived from the JavaScript programming language, its appearance is similar to that of JavaScript objects.

A sample JSON object is represented as follows:

```
{
  "student": {
    "name": "Rumaisa Mahoney",
    "age": 30,
    "fullTime": true,
    "languages": [ "JavaScript", "HTML", "CSS" ],
    "GPA": 3.9,
    "favoriteSubject": null
  }
}
```

Note the following syntax rules for JSON:

- The curly braces, {..}, hold objects.
- The square brackets, [..], hold arrays.
- Data is stored in name-value pairs separated by a colon, :.
- Every name-value pair is separated from another pair by a comma, ,. Similarly, every item in an array is delimited by a comma as well. [Trailing commas](#) are forbidden.
- JSON property names must be in double-quoted (" ") text even though JavaScript names do not hold by this stringency.

## JSON Data Types

A JSON data type must be one of the following:

- string (double-quoted)
- number (integer or floating point)
- object (name-value pair)
- array (comma-delimited)
- boolean (true or false)
- null

Try to find all the data types in this JSON example:

```
{
  "student": {
    "name": "Rumaisa Mahoney",
    "age": 30,
    "fullTime": true,
    "languages": [ "JavaScript", "HTML", "CSS" ],
    "GPA": 3.9,
    "favoriteSubject": null
  }
}
```

Notably, JSON doesn't cover every data type. Types that are not represented in JSON such as dates can be stored as a string and converted to a language-specific data structure. Here's an acceptable internationally-recognized date format in [ISO 8601](#):



"2014-01-01T23:28:56.782Z"

This above format contains parts which resemble a date and time. However, as a string, it is hard for a programming language to use as is. Conveniently, every programming language has built-in JSON facilities to convert this string into a more readable and usable format, such as:

Wed Jan 01 2014 13:28:56 GMT-1000 (Hawaiian Standard Time)

This pretty much covers the basic description of JSON, its popularity, and its syntax. Congratulations on reaching this milestone!

## Working with JSON in JavaScript

A user guide on how to work with JSON in Javascript.

### Introduction

JSON, short for JavaScript Object Notation, is a language-independent data format that has been accepted as an industry standard. Because it is based on the JavaScript programming language, JSON's syntax looks similar to a JavaScript object with [minor differences](#). We'll take a look at the subtle difference between them. Later on, we'll learn how to parse JSON and extract its content as JavaScript. Lastly, we'll learn how to write a JSON object with JavaScript. So, let's begin.

### JSON Object vs. JavaScript Object

Here is an example JSON object of a person named Kate, who is 30 years old, and whose hobbies include reading, writing, cooking, and playing tennis:

```
{
  "person": {
    "name": "Kate",
    "age": 30,
    "hobbies": [ "reading", "writing", "cooking", "tennis" ]
  }
}
```

Represented as a JavaScript object literal, the same information would appear as:

```
{
  person: {
    name: 'Kate',
    age: 30,
    hobbies: [ 'reading', 'writing', 'cooking', 'tennis' ]
  }
}
```

Notice a slight difference between the two formats.

- The name portion in each JSON name-value pair and all string values must be enclosed in double-quotes while this is optional in JavaScript.
- JavaScript accepts string values that are single or double-quoted, however, there exists JavaScript [coding guidelines](#) that prefer one style over another.

## Reading a JSON String

In a typical web application, the JSON data that we receive from a web request comes in the form of a string. At other times, JSON data is stored in a file that is used for authentication, configuration, or database storage. These files typically have a .json extension, and they have to be opened in order to retrieve the JSON string in it. In either case, we will need to convert the string to a format that we can use in order to access its parts. Each programming language has its own mechanism to handle this conversion. In JavaScript, for example, we have a built-in JSON class with a method called [.parse\(\)](#) that takes a JSON string as a parameter and returns a JavaScript object.

The following code converts a JSON string object, `jsonData`, into a JavaScript object, `jsObject`, and logs `jsObject` on the console.

```
const jsonData = '{ "book": { "name": "JSON Primer", "price": 29.99, "inStock": true, "rating": null } }';
```

```
const jsObject = JSON.parse(jsonData);
```

```
console.log(jsObject);
```

This will print out `jsObject` as follows:

```
{
  book: { name: 'JSON Primer', price: 29.99, inStock: true, rating: null }
}
```

Once we have converted a JSON object to a JavaScript object, we can access the individual properties inside the JavaScript object. To access a value inside a JavaScript object based on its property name, we can either use dot notation, (`.propertyName`), or bracket notation, (`['propertyName']`).

For instance, to retrieve the `book` property of `jsObject` we could do the following:

```
// Using the dot notation
const book = jsObject.book;
console.log(book);
console.log(book.name, book.price, book.inStock);
```

```
// Using the bracket notation
const book2 = jsObject["book"];
console.log(book2);
console.log(book2["name"], book2["price"], book2["inStock"]);
```

Both ways of accessing the `book` property return the same output:

```
{ name: 'JSON Primer', price: 29.99, inStock: true, rating: null }
JSON Primer 29.99 true
```

As you can see, after parsing `jsonData` into a JavaScript object that's stored in the variable, `book`, you can treat `book` like any other object! That means you can access property values, as shown above, edit existing values, iterate over the keys and values, etc...

## Exercise: Reading a JSON String

Now that we've shown you how to read a JSON string, let's practice with a code challenge by writing some code yourself.

#### Coding question

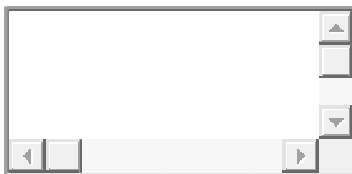
1. Create a variable called `jsObject` that is an object parsed from `jsonData`.
2. Print out the array of all the children property nested in `jsObject`. Be sure to use either bracket notation or dot notation to access the nested properties.

1

```
const jsonData = '{ "parent": { "name":  
"Sally", "age": 45, "children" : [ {  
"name": "Kim", "age": 3 }, { "name":  
"Lee", "age": 1 } ] } }';
```



Output:



Run

Check answer

Run your code to check your answer

#### Writing a JSON String

Before we can send off our data across the web, we need to convert them to a JSON string. In JavaScript, we would use the built-in JSON class method, [.stringify\(\)](#) to transform our JavaScript object to a JSON string.

The following code converts a JavaScript object, `jsObject`, into a JSON string, `jsonData`.

```
const jsObject = { book: 'JSON Primer', price: 29.99, inStock: true, rating: null };  
const jsonData = JSON.stringify(jsObject);  
console.log(jsonData);
```

This will display the following output:

```
{ "book": "JSON Primer", "price": 29.99, "inStock": true, "rating": null }
```

## Exercise: Writing a JSON String

Now that we've learned how to convert our JavaScript object to a JSON string, let's practice with another code challenge by you writing some code.

### Coding question

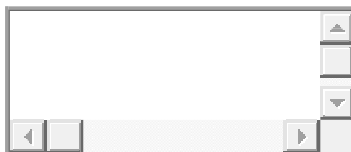
As a developer, you receive some data in the form of a JSON string in the variable, `jsonData`. However, the content of `jsonData` is not completely correct. The age value of the parent property should be 35 instead of 45. Without changing the content of `jsonData` directly, update the age value and then log a new JSON string with the correct value in the console.

Here is a step-by-step guide to solve this challenge:

1. Convert `jsonData` to a JavaScript object using `JSON.parse()` and save it as a const variable, for instance, `jsObject`.
2. Use either the dot, `.key`, or bracket, `['key']`, notation to access the parent property of `jsObject` followed by the age property and change its value from 45 to 35.
3. Convert `jsObject` back to a JSON string using `JSON.stringify()` and save it as another const variable, for instance, `jsObjectToJson`.
4. Log the `jsObjectToJson` string on the console.

1

```
const jsonData = '{"parent":  
{"name":"Sally","age":45,"children":[  
{"name":"Kim","age":3},{"name":"Lee",  
"age":1}]}}';
```



Output:



Run

Check answer

Run your code to check your answer

## Review

In this article, you have learned how to do the following:

- Compare JSON with JavaScript's Object literal syntax.
- Convert a JSON string into a JavaScript Object.
- Convert a JavaScript Object into a JSON string.

Congratulations on reaching this milestone! With a little more practice, you will soon handle JSON in Javascript with ease.

.....

## REQUESTS WITH FETCH API

### Introduction to Requests with ES6

Have you ever wondered what happens after you click a "Submit" button on a web page? For instance, if you are submitting information, where does the information go? How is the information processed? The answer to the previous questions revolves around [HTTP requests](#).

There are many types of HTTP requests. The four most commonly used types of HTTP requests are GET, POST, PUT, and DELETE. In this lesson, we'll cover GET and POST requests.

With a GET request, we're retrieving, or *getting*, information from some source (usually a website). For a POST request, we're *posting* information to a source that will process the information and send it back.

JavaScript uses an [event loop](#) to handle asynchronous function calls. When a program is run, function calls are made and added to a stack. The functions that make requests that need to wait for servers to respond then get sent to a separate queue. Once the stack has cleared, then the functions in the queue are executed.

Web developers use the event loop to create a smoother browsing experience by deciding when to call functions and how to handle asynchronous events. We will go into event loops in more detail in the [Concurrency Model and Event Loop in JavaScript](#) article.

To make asynchronous event handling easier, [promises](#) were introduced in ES6 JavaScript.

In this lesson, we will explain how to use `fetch()` and promises to handle requests. Then, we will simplify requests using `async` and `await`

## REQUESTS WITH FETCH API

### Intro to GET Requests using Fetch

The first type of requests we're going to tackle is GET requests using `fetch()`.

The `fetch()` function:

- Creates a request object that contains relevant information that an API needs.
- Sends that request object to the API endpoint provided.
- Returns a promise that ultimately resolves to a response object, which contains the status of the promise with information the API sent back.

Let's walk through the boilerplate code to the right for using `fetch()` to create a GET request step by step.

First, call the `fetch()` function and pass it a URL as a string for the first argument, determining the endpoint of the request.

```
fetch('https://api-to-call.com/endpoint')
```

The `.then()` method is chained at the end of the `fetch()` function and in its first argument, the response of the GET request is passed to the callback arrow function. The `.then()` method will fire only after the promise status of `fetch()` has been resolved.

Inside the callback function, the `ok` property of the response object returns a Boolean value. If there are no errors, `response.ok` will be true and the code will return `response.json()`.

If `response.ok` is a falsy value, our code will throw an error.

```
throw new Error('Request failed!');
```

A second argument passed to `.then()` will be another arrow function that will be triggered when the promise is rejected. It takes a single parameter, `networkError`. This object logs the `networkError` if we could not reach the endpoint at all (e.g., the server is down).

A second `.then()` method will run after the previous `.then()` method has finished running without error. It takes `jsonResponse`, which contains the returned `response.json()` object from the previous `.then()` method, as its parameter and can now be handled, however we may choose.

```
// fetch GET

fetch('http://api-to-call.com/endpoint').then(response => {
  if (response.ok) {
    return response.json();
  }
  throw new Error('Request failed!');
}, networkError => console.log(networkError.message))
.then(jsonResponse => {
  // Code to execute with jsonResponse
});
```



The diagram consists of four horizontal brackets on the right side of the code block, each pointing to a specific part of the code:

- The first bracket points to the `fetch()` call and is labeled "sends request".
- The second bracket points to the `if (response.ok) { return response.json(); }` block and is labeled "converts response object to JSON".
- The third bracket points to the error handling part `throw new Error('Request failed!');` and `networkError => console.log(networkError.message)` and is labeled "handles errors".
- The fourth bracket points to the final `.then(jsonResponse => { ... })` block and is labeled "handles success".

## REQUESTS WITH FETCH API

### Making a GET Request

In the previous exercise, we went over the boilerplate code for a GET request using `fetch()` and `.then()`. In this exercise, we're going to apply that code to access the [Datamuse API](#) and render the fetched information in the browser.

The Datamuse API is a word-finding query engine for developers. It can be used in apps to find words that match a given set of constraints that are likely in a given context.

If the request is successful, we'll get back an array of words that sound like the word we typed into the input field.

We may get some errors as we complete each step. This is because sometimes we've split a single step into one or more steps to make it easier to follow. By the end, we should be getting no errors.

## REQUESTS WITH FETCH API

### Handling a GET Request

Great job making it this far!

In the previous exercise, we called the `fetch()` function to make a GET request to the Datamuse API endpoint. Then, you chained a `.then()` method and passed two callback functions as arguments — one to handle the promise if it resolves, and one to handle network errors if the promise is rejected.

In this exercise, we will chain another `.then()` method, which will allow us to take the information that was returned with the promise and manipulate the webpage! Note that if there is an error returned in the first `.then()` method, the second `.then()` method will not execute.

## REQUESTS WITH FETCH API

### Intro to POST Requests using Fetch

In the previous exercise, we successfully wrote a GET request using the fetch API and handled Promises to get word suggestions from Datamuse. Give yourself a pat on the back (or two to treat yourself)!

Now, we're going to learn how to use `fetch()` to construct POST requests!

Take a look at the diagram to the right. It has the boilerplate code for a POST request using `fetch()`.

Notice that the `fetch()` call takes two arguments: an endpoint and an object that contains information needed for the POST request.

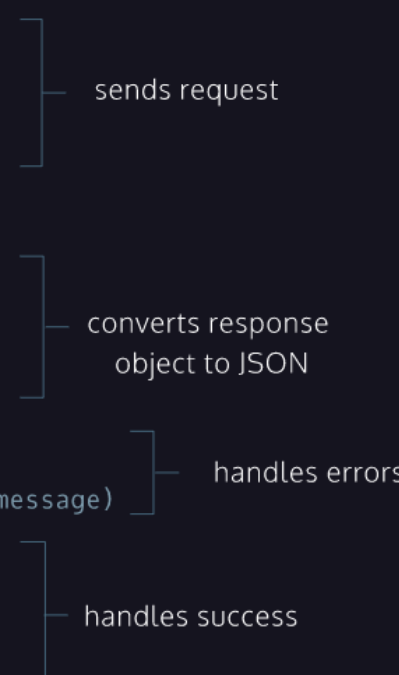
The object passed to the `fetch()` function as its second argument contains two properties: `method`, with a value of `'POST'`, and `body`, with a value of `JSON.stringify({id: '200'})`. This second argument determines that this request is a POST request and what information will be sent to the API.

A successful POST request will return a response body, which will vary depending on how the API is set up.

The rest of the request is identical to the GET request. A `.then()` method is chained to the `fetch()` function to check and return the response as well as throw an exception when a network error is encountered. A second `.then()` method is added on so that we can use the response however we may choose.

```
// fetch POST

fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'})
}).then(response => {
  if (response.ok) {
    return response.json();
  }
  throw new Error('Request failed!');
}, networkError => console.log(networkError.message))
  .then(jsonResponse => {
    // Code to execute with jsonResponse
  });
```



sends request

converts response object to JSON

handles errors

handles success

## REQUESTS WITH FETCH API

### Making a POST Request

In the previous exercise, we walked through the boilerplate code for making a POST request using `fetch()` and `.then()`. In this exercise, we're going to use that boilerplate code to shorten a URL using the [Rebrandly URL Shortener API](#).

We will need a Rebrandly API key. To do this, read through [the Rebrandly sign up guide](#) to set up your API.

Keep in mind, while it's ok to use your API key in these exercises, you should not share your key with anyone (not even to ask a question in the forums)! Also, if you reset the exercise at any point, you will have to paste in your API key again at the top.

## REQUESTS WITH FETCH API



## Handling a POST Request

In the previous exercise, we set up the POST request by providing the endpoint and the object containing all the necessary information. In this exercise, we'll handle the response.

The request returns a Promise which will either be resolved or rejected. If the promise resolves, we can check and return that response. We will chain another `.then()` method and handle the returned JSON object and display the information to our webpage.

Let's implement this knowledge into our code!

Remember that if you reset the exercise at any point, you will have to paste in your API key again at the top!

## REQUESTS WITH FETCH API

### Intro to async GET Requests

In the following exercises, we're going to take what we've learned about chaining Promises and make it simpler using functionality introduced in ES8: `async` and `await`. You read that right, you did the hard part already. Now it's time to make it easier.

The structure for this request will be slightly different. We will use the new keywords `async` and `await`, as well as the `try` and `catch` statements.

Take a look at the diagram on the right.

Here are some key points to keep in mind as we walk through the code:

- The `async` keyword is used to declare an `async` function that returns a promise.
- The `await` keyword can only be used within an `async` function. `await` suspends the program while waiting for a promise to resolve.
- In a `try...catch statement`, code in the `try` block will be run and in the event of an exception, the code in the `catch` statement will run.

Study the `async getData()` function to the right to see how the request can be written using `async` and `await`.

```
// async await GET

const getData = async () => {
  try {
    const response = await fetch('https://api-to-call.com/endpoint');
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request failed!');
  } catch (error) {
    console.log(error);
  }
}
```

## REQUESTS WITH FETCH API

### Making an async GET Request

In the previous exercise, we walked through the boilerplate code for making a GET request using `async` and `await`.

In this exercise, we're going to apply the code to get nouns that describe the inputted word using the [Datamuse API](#).

## REQUESTS WITH FETCH API

### Intro to async POST Requests

Now that you've made an `async` GET request, let's start getting familiar with the `async` POST request.

As we've seen before, a POST request requires more information. Take a look at the diagram to the right.

We still have the same structure of using `try` and `catch` as the `async` GET request we just learned about. But, in the `fetch()` call, we now have to include an additional argument that contains more information like `method` and `body`.

The `method` property value is set to `'POST'` to specify the type of request we are making. Then we have to include a `body` property with the value of `JSON.stringify({id: 200})`.

```
// async await POST

const getData = async () => {
  try {
    const response = await fetch('https://api-to-call.com/endpoint', {
      method: 'POST',
      body: JSON.stringify({id: 200})
    })
    if(response.ok){
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request failed!');
  } catch(error) {
    console.log(error);
  }
}
```

sends request

handles response if successful

handles response if unsuccessful

## REQUESTS WITH FETCH API

### Making an async POST Request

Since we've reviewed the boilerplate code for an `async` POST request, the next step is to incorporate that logic into making a real request.

In this exercise, we'll need to retrieve our Rebrandly API key to access the [Rebrandly API](#).

We will then pass in the endpoint and the request object into the `fetch()` method to make our POST request.

If you reset the exercise at any point, you will have to paste in your API key again at the top!

## REQUESTS WITH FETCH API

### Review

In this lesson, we learned how to make GET and POST requests using the Fetch API and `async/await` keywords. Let's recap on the concepts covered in the previous exercises:

- GET and POST requests can be created in a variety of ways.
- We can use `fetch()` and `async/await` to asynchronous request data from APIs.

- Promises are a type of JavaScript object that represents data that will eventually be returned from a request.
- The `fetch()` function can be used to create requests and will return promises.
- We can chain `.then()` methods to handle promises returned by the `fetch()` function.
- The `async` keyword is used to create asynchronous functions that will return promises.
- The `await` keyword can only be used with functions declared with the `async` keyword.
- The `await` keyword suspends the program while waiting for a promise to resolve.

Congratulations on learning all about asynchronous requests using `fetch()`, `async`, and `await`! These concepts are fundamental to helping you develop more robust web apps!