TASK

Promises

Visit our website

Introduction

WELCOME TO THE PROMISES TASK!

One thing you may have noticed in programming thus far is that many of the concepts could be applied to a form of logic in real life. Conditional statements are a perfect example of this.

In today's task, we're going to be going over one of these concepts, "Promises". These are a significant portion of working in the programming field as they are found across multiple languages.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** to start a chat with a code reviewer, as well as share your thoughts and problems with peers. You can also schedule a call or get support via email.

Our code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

BEFORE WE CONTINUE - WE NEED TO INSTALL NODE FOR THE UPCOMING TASKS

FOR LINUX/WSL USERS

The easiest way to install software in Linux (specifically the Debian variations like Ubuntu) is using the **apt-get** package manager. To install NodeJS on Linux/WSL, you simply type in:

```
> sudo apt install npm nodejs
```

Let's break down what this command means:

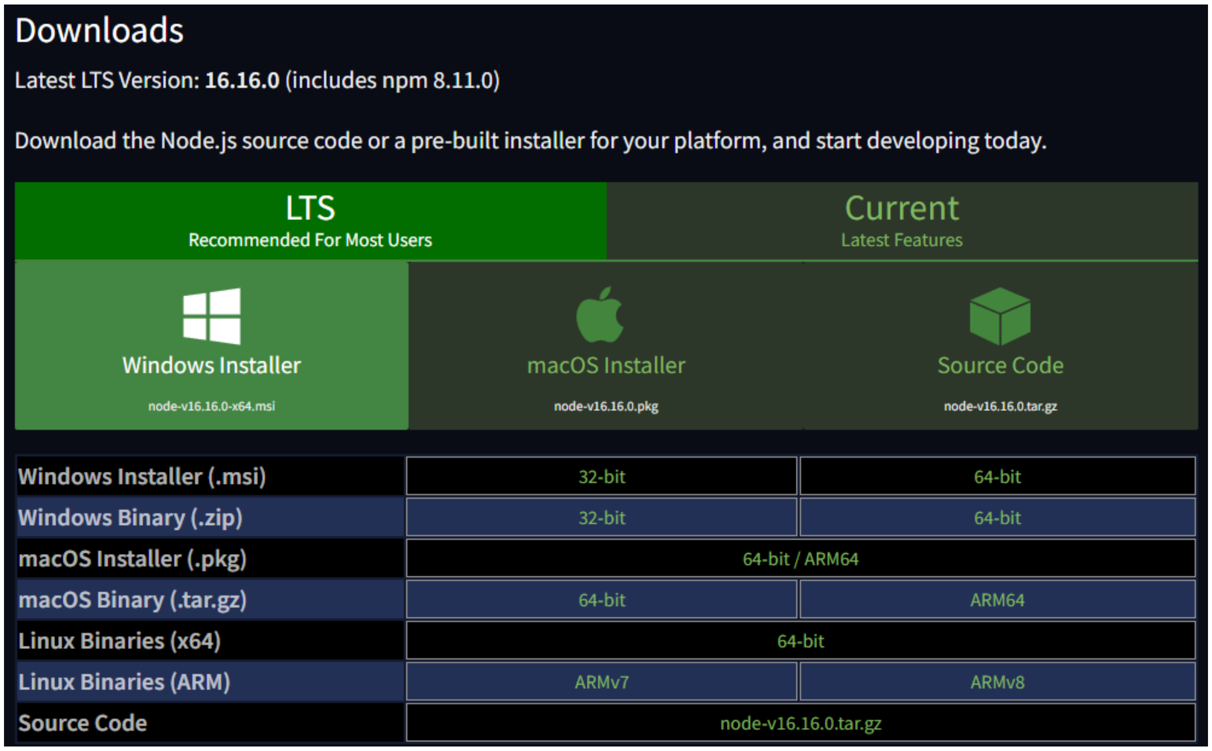
1. sudo: this is placed at the beginning of any command that needs super-user privileges, such as this one. This means "super-user do". If you aren't logged in as a super-user, this command will prompt you for your password
2. apt: run the apt-get application (apt is just short for apt-get).
3. install: use the install option within apt.
4. npm: this is the node package manager. This will be useful later on.
5. nodejs: the package to install. The list of installable packages can be seen with apt-cache search.

You can also follow these steps to install Node:
https://heynode.com/tutorial/install-nodejs-locally-nvm/

FOR WINDOWS USERS

Go to the NodeJS download page **here**.

You will be presented with a set of download options that looks something like this:



Downloads

Latest LTS Version: **16.16.0** (includes npm 8.11.0)

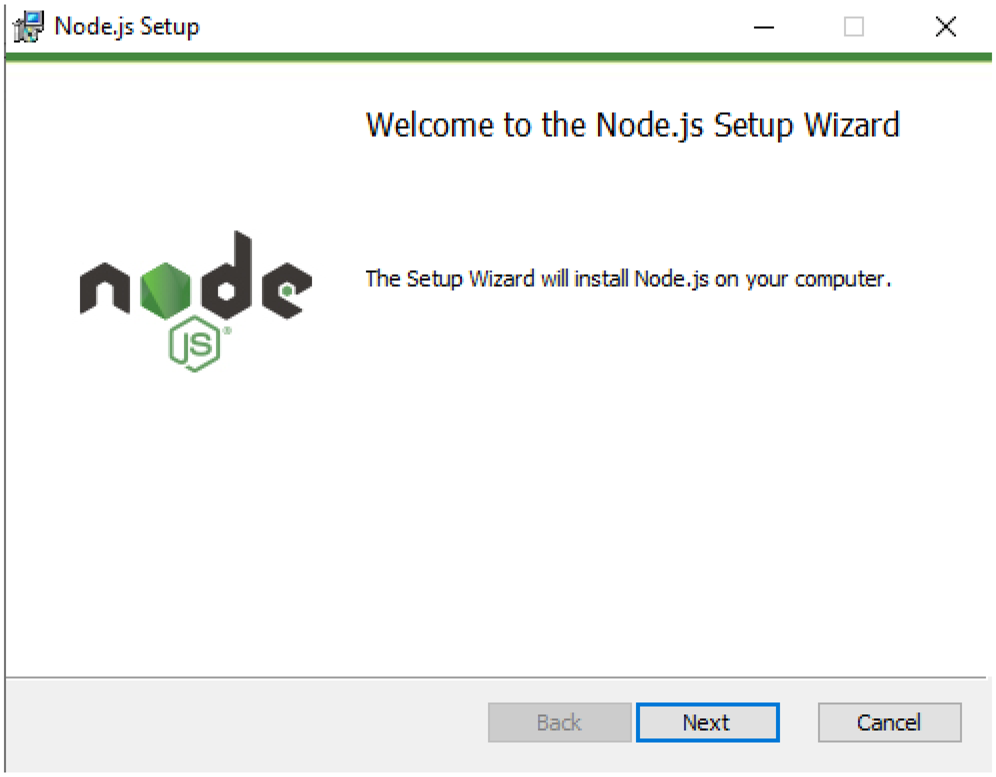
Download the Node.js source code or a pre-built installer for your platform, and start developing today.

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v16.16.0.tar.gz	

Because you are running Windows, you should choose the Windows Installer (.msi). Most of you will need to choose the 64-bit version. If you are running a 32-bit system, then you will need to use the 32-bit installer.

If you're unsure which system you're using currently, just click the **Start** button and then select **Settings**. Then click **System** and choose **About**. This should reveal the bit-version for your particular System. If you're using a Mac, then you'll find your processor type in **About This Mac**.

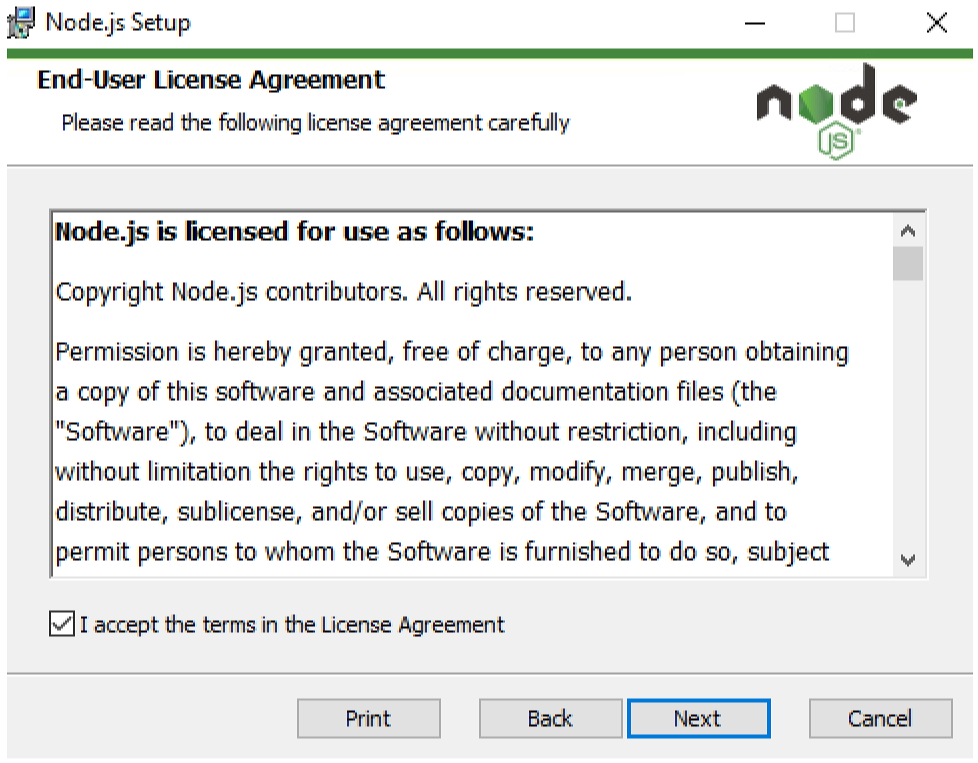
After downloading it, double-click the file to open it. You will be greeted with something that looks like this:



Welcome to the Node.js Setup Wizard

The Setup Wizard will install Node.js on your computer.

It will ask you to accept the EULA. Click **accept**, then click **Next**.



End-User License Agreement

Please read the following license agreement carefully

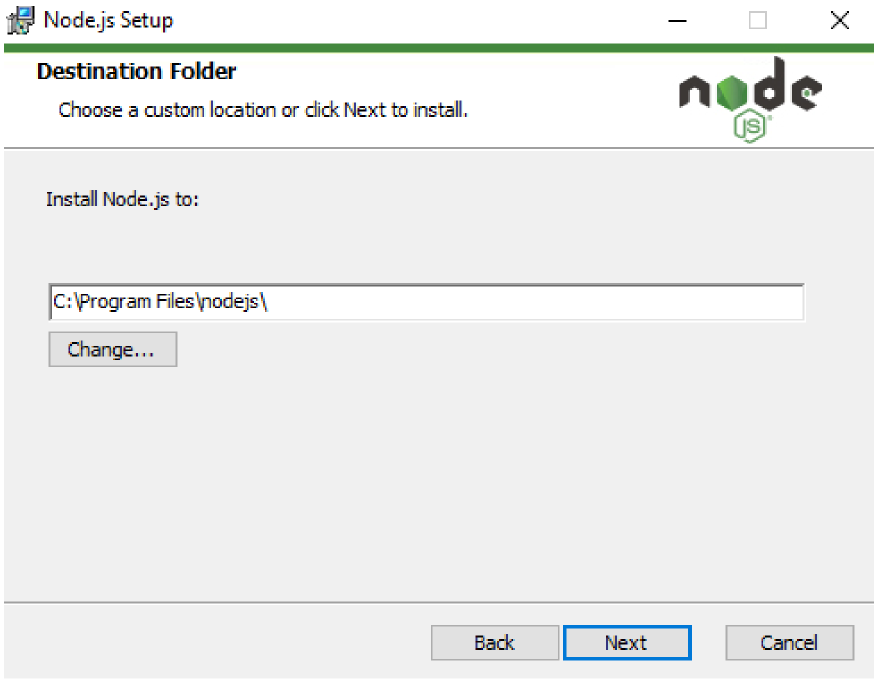
Node.js is licensed for use as follows:

Copyright Node.js contributors. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject

[✓] I accept the terms in the License Agreement

The next step is choosing **where** you would like to install Node. Select the location (or use the default) and click **Next**.



Node.js Setup

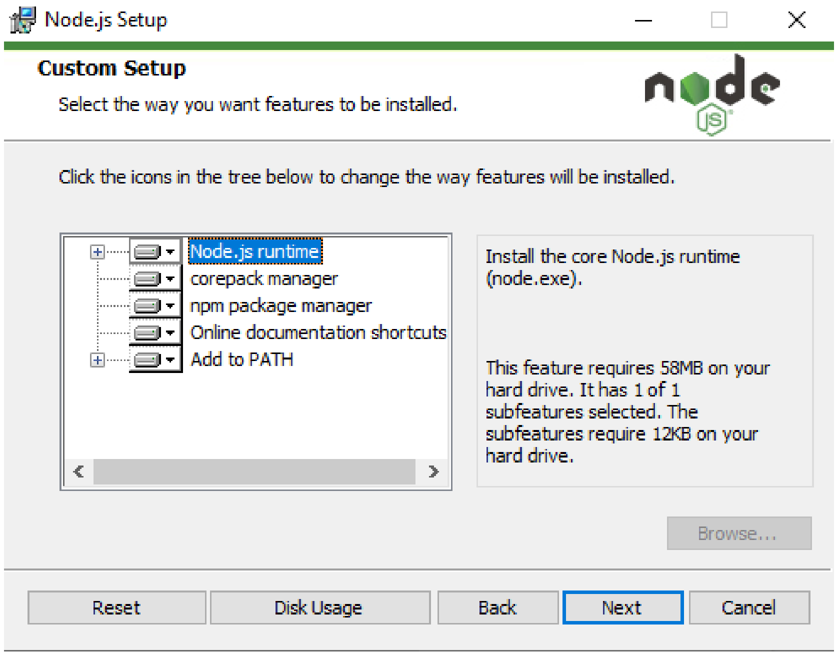
Destination Folder

Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

Once this is done, you'll be presented with a set of complex-looking options. Good news: these are just the defaults: click **Next**:



Node.js Setup

Custom Setup

Select the way you want features to be installed.

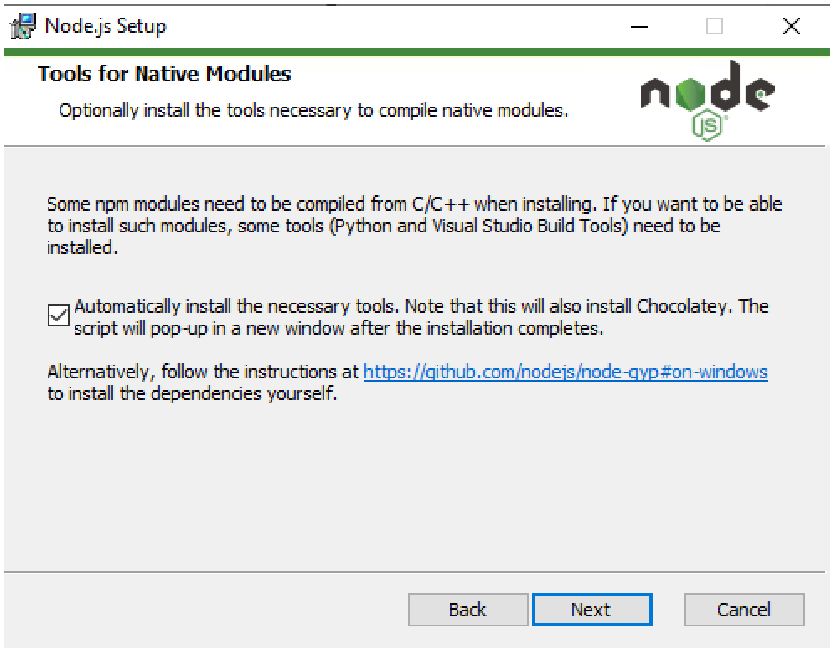
Click the icons in the tree below to change the way features will be installed.

- Node.js runtime
- corepack manager
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

This feature requires 58MB on your hard drive. It has 1 of 1 subfeatures selected. The subfeatures require 12KB on your hard drive.

The final screen, before installation, will inform you that certain dependencies will need to be installed for NodeJS to run properly. It presents you with an option to auto-install them, **check the box** and click **Next**:


Node.js Setup

Tools for Native Modules

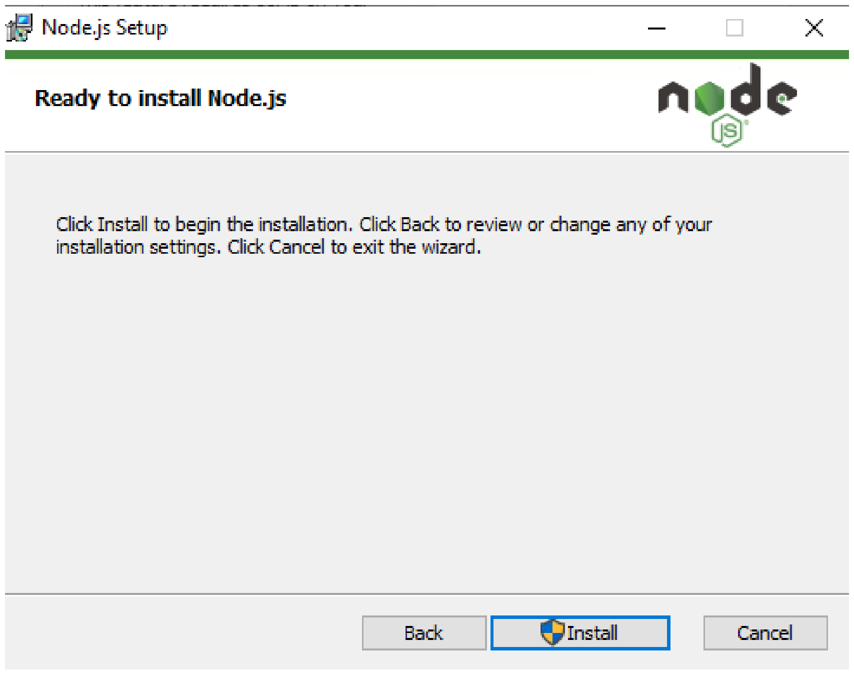
Optionally install the tools necessary to compile native modules.

Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.

☑ Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.

Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.

Now that your set-up is done, you can install Node! This will require some **admin privileges**:


Node.js Setup

Ready to install Node.js

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

FOR MAC USERS

Use the macOS in-built package manager: **brew**. To use this option, open a terminal. To do this, press Command+Space and type in "Terminal".

Follow the instructions : **https://formulae.brew.sh/formula/node**

Once your terminal is open, type in:

```
> brew install node
```

And Brew will install NodeJS for you. Simple!

LET'S GET BACK TO PROMISES - A BASIC RUNDOWN

Let's take a look back on your life. There have been many events that have happened, but one of the most memorable things we tend to have is promises.

In life, when we promise something, we typically have the idea that we are going to do something that someone has requested we do. For example, back when you were in school, you may have had the teacher mention that you needed to complete your homework.

You would often "promise" that you would achieve it by the time they requested it be finished. Of course, sometimes, you may not have been able to fulfil this promise; this, in turn, would have caused you to give your teacher an "excuse" as to why you didn't fulfil your promise.

Promises in programming are precisely the same. Your program is expected to do something, but sometimes it may not be able to complete it, which will return an error.

A widespread practice for this usage is API calls. This task will be going into quite a lengthy amount of detail, so get a nice cup of coffee/tea/hot chocolate and get ready to enter a new but fantastic world of concepts!

WHAT IS AN API?

As you've read in the above extract, we have used a word you may have seen somewhere else, such as movies or a news article about the world of tech and that is "promises". Now let's expand your knowledge even further and give you a new concept that you can use to show off to your friends and family!

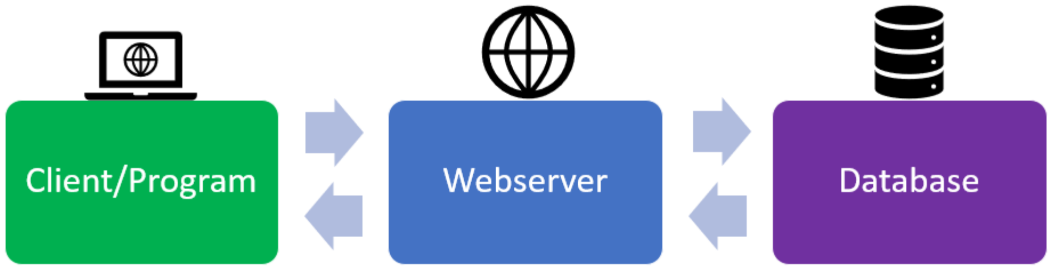
API stands for Application Programming Interface. This may seem like a terrifying term, but we'll explain this in a way that can be applied to real life.

Imagine you go to a restaurant, you sit down, and you get all comfortable. Then you have your waiter/waitress take your order. Once they have your order, they then take it to the kitchen to have your amazing meal cooked. Once it's made and ready to be eaten, it's taken back to you to enjoy!

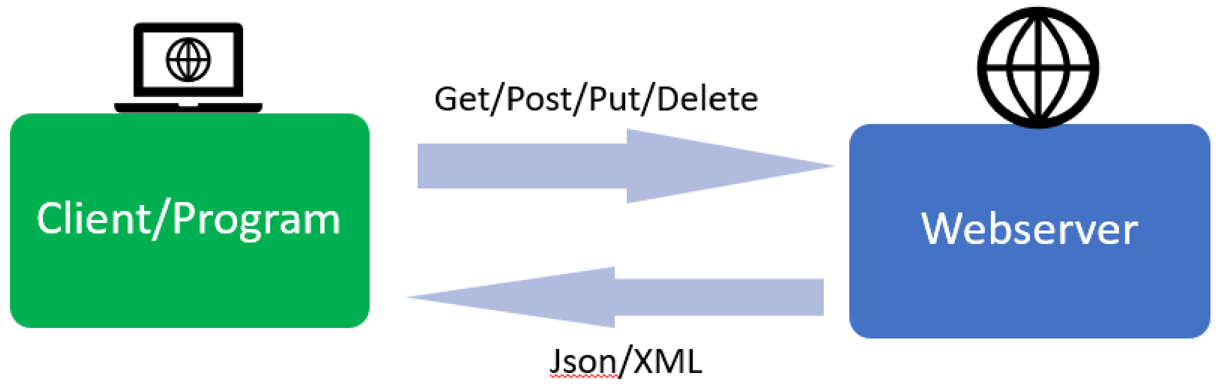
The above example is exactly what an API is! Okay, of course, it's not some random person coming to you to take your order, but the concept is the same!

Let's take the above example and convert it into "programming logic".

Let's imagine that the customer in the above example is your program. Your program is going to send a request to a web server (in the above example, this would be the waiter/waitress). Once you send this request to the server, it takes this request and gets your return value (this would be your food). Once this return value is sent to you, you are able to work with this data and do whatever you want with it! There are many things you can do with APIs, and they open up so many possibilities for a unique program to be created!



As you can see in the above image, each step connects to another to allow for the information to be sent to the server. Let's take a deeper look at the connection between the client and the webserver.



As you can see in the above image, we've added a couple of new words to each arrow. These are also referred to as requests. Requests are used to send your information to the server, asking it to perform a set of tasks. In other words, what you order is what you'll get.

Now let's take a look at each of these requests to understand what they can be used for:

- **GET** - This is going to request or "get" the information that you are looking for. This could be the information from a book, for example.
- **PUT** - Update an existing set of information that's on the server, like changing your password for an account.
- **DELETE** - As the name suggests, this will delete the set of information you are looking for.
- **POST** - Uploads a set of information to the server. Such as changing your profile picture.

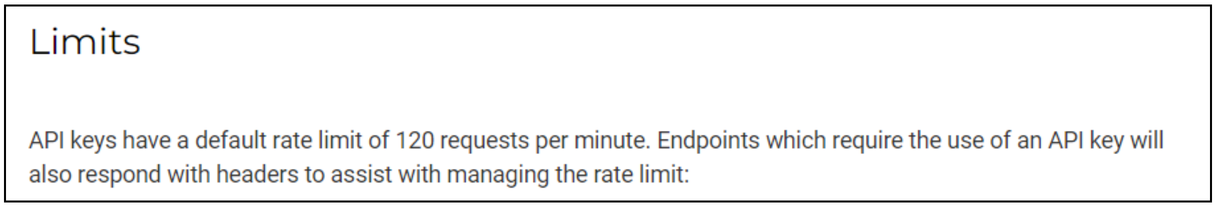
Now I know what you might be thinking… "Well, if I can just delete information from the website using APIs, does that not mean I can easily hack websites?"

Not quite. While you can use APIs for many things, there are still limitations. These are known as **API keys.** Many websites that have a public domain will make use of these.

API keys set the permissions that users can use to make a request to their server. Think of it as permissions; you can only perform specific actions if the API key you are using allows for it to happen (you can't just open every door with your house keys, only your house door will open.)

Typically websites that use public APIs will only give you access to the **GET** command, which will allow you to retrieve information that can be used for your application/website.

It's important to ALWAYS try to read through the API documentation to give you a better understanding of what you are allowed/not allowed to do. Let's look at an example:


Limits

API keys have a default rate limit of 120 requests per minute. Endpoints which require the use of an API key will also respond with headers to assist with managing the rate limit:

As you can see, this website explains how many requests you can make using their API. This is why it's important that you read through any API's documentation to avoid errors caused by limitations you are unsure of.

And that's it! That's what APIs are! Now we can start applying this logic (well, soon). Now that you know what an API is all about, let's look at another interesting concept called ASYNC.

ASYNC vs SYNC

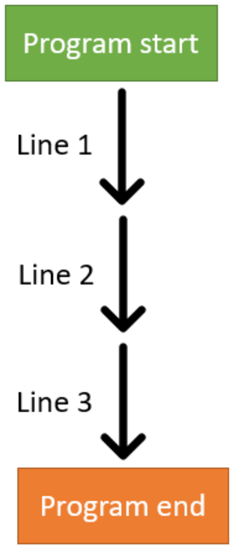
This is another concept you may have come across during your time researching development concepts - Synchronous or SYNC, and asynchronous, or ASYNC, are two important components of development concepts.

The reason this is part of this PDF is that learning about promises involves introducing a new form of programming you're not used to. Because of this, it's vital that we go over related concepts.

Synchronous

Synchronous processing is the form of programming that you have become accustomed to. It's the concept that when your program runs it starts running from top to bottom, left to right. The line below the current line of code won't run until the current line of code is finished running.

This concept is a relatively safe approach to programming as the flow of control will always follow you through the code.



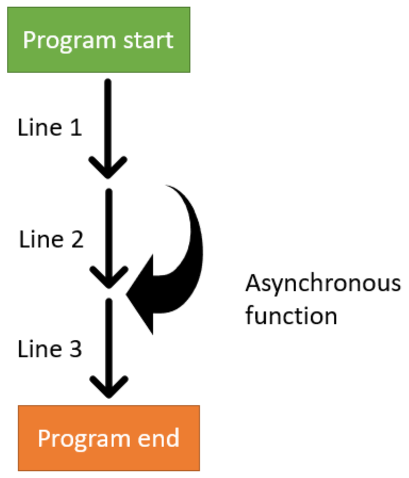
As you can see in this example above, each line runs one at a time from the program start to the program end. This should not seem unusual to you as this is how you've been writing all of your JavaScript code thus far.

Asynchronous

Now we start getting into slightly more complex concepts such as asynchronous processing. The idea behind asynchronous processing is that multiple sets of code are run at the same time. 🤯 That's right, we're able to do that in the world of programming!

However, it's important to remember that even though this is possible, it's suggested that you avoid making use of asynchronous processing where possible - it should only be used where it's safe and where required to do so. Asynchronous programming can cause more errors than synchronous programming if you don't understand why and how you are using it, as you're attempting to run multiple sections of code at the same time and there could be interdependencies, i.e. parts of one set of code may depend on parts of another set of code that is running at the same time.

Take a look at the image below:



The above example represents the same program as the previous example, but now we have an added asynchronous functionality. The asynchronous function is actually running at the same time as the rest of the code.

What's important to remember is that even though, in the example above, the synchronous function ends under line 2, it won't always end after line 2. Many elements (such as the browser and user computer speeds) can cause the function to load quicker or slower.

The general rule of thumb is to only run an asynchronous function if no other code is dependent on it.

You'll get a better understanding of how to work with asynchronous function calls in a later task. For now you only need to understand that **promises are asynchronous** and **run separately** from your normal code.

PROMISES

Now that you have a basic understanding of how APIs work and what synchronous and asynchronous processing is, we can now move onto the actual core content of this lesson: using promises to call an API and use that data in our program.

As mentioned earlier, when our program promises something, it is going to do its best to fulfil the promise by completing the required action. This can be used in many places (such as calling functions!). However, the best way to explain the concept of promises is by going over API calls.

Let’s start off by taking a look at a very basic API call!

Calling an API using a promise

Have a look at the code block below:

```
let items = [];

fetch("https://api.chucknorris.io/jokes/random")
   .then(res => res.json())
   .then((result) => {
      items = result;
      console.log(items);
   }),
   (error) => {
      console.log(error);
   }
```

Don’t worry! It looks like a lot, but we’re going to break down this code line by line to easily help you understand the concept of a promise and APIs.

Fetching
As seen in the first line of the central code block, we have a new keyword, “fetch”, which has a link contained inside it. This is our API link. As you can see, the link is to a random joke generator.

If you were to go to this website, you’d get a display similar to the one below:

```
{
  categories: [],
  created_at: '2020-01-05 13:42:26.766831',
  icon_url: 'https://assets.chucknorris.host/img/avatar/chuck-norris.png',
  id: 'spoCwtffQY-ZiulSUzEcmw',
  updated_at: '2020-01-05 13:42:26.766831',
  url: 'https://api.chucknorris.io/jokes/spoCwtffQY-ZiulSUzEcmw',
  value: 'anything you can do he can do better, Chuck Norris can do
anything better than you'
}
```

It looks like a lot of word vomit. But we will return to this output shortly to explain it! For now, just understand that this is returning a Javascript object!

`.then(res => res.json())`
As you can see there is a **`.`** followed by a **`then`**. Our **`fetch`** keyword is our promise. It's promised that it's going to fetch the data. If that fetch works, then we are going to get a result. Because text can only be sent over the web and we can't send objects, we have to convert it.

Currently, the **`res`** keyword is our **res**ulting text. We then convert it into an object using JSON. This is now a usable object that we have passed through. Now we can start working with it inside our code.

`.then((result) => {`
You may have noticed the second **`then`**. This is actually the return that we can now use in our code. The **`result`** keyword is actually carried over from the previous **`then`** and placed within our arrowhead function.

Once you've completed this, you can start using any normal code between the curly brackets.

You may now be asking, "But we were told not to write code that requires asynchronous functions?"

Correct! However, because this code is all within the synchronous function, each step still needs to be completed inside the promise one at a time; however, while all this happens, the rest of your JS code will continue to run!

The rest of the code
The rest of the code is now just referencing the object we've retrieved from our API; remember the content that looked like a bunch of random text? That's what's currently in our code! (And we got that from some random website, how cool is that?)

To access only the **joke** section we can rewrite the console log to the below:

```
    console.log(result.value);
```

As you can see, all of this code is executed through the promise - it starts off with a promise to fetch the data and then send it back to us!

But wait, what if the website goes offline? Then what? We can't fulfil the promise!

This is where our error comes into play!

Error Handling

```
    (error) => {
        console.log(error);
    }
```

Take a look at this line of code; this will only ever run if our **`fetch`** does not run! The website we're trying to connect to will return an error and print it out to a console.

This is the power of promises. They are a strong tool that every programmer should learn to use!

PROMISES USING NORMAL FUNCTIONS

We're almost done. We just need to go over one more important concept, creating our own promises! Just like an API call using promises, if we create our own function to check for a promise, it will run asynchronously.

Take a look at the code below:

```
let myPromis = new Promise(function(resolve, reject) {
   let x = 0
   if(5 < 1){
      x = 1
   }else{
      x = 0
   }

   if (x === 1) {
      resolve("Resolved Correctly!");
   }
   else {
      reject(Error("The code ran incorrectly."));
   }
});

myPromis.then(function (result) {
   console.log(result)
}, function (error){
   console.log(error)
})

console.log("I'll still be running tho")
```

Once again we're going to break down this code line by line to allow for the most optimal understanding.

`variable myPromis`
At this point, you should be no stranger to OOP, or object-oriented programming. This first line is simply creating a new promise object. This is the default style whenever you create a promise. Within the parameters of the **`promis`** object we create a function that will take two additional parameters.

- Resolve - This will run should the promise run correctly.
- Reject - Vice versa - if the promise does not work correctly, it'll return an error.

Once you have created all of this, you can write any code you may want the action to perform! In this case, we simply check if 5 is less than 1 (which of course it's not!)

What's important to remember is that in every promise you create you need to have both a resolve and reject output (think of these as a return statement for a promise!)

The idea behind the code within the promise function is that it should always only return one value, your resolve and your rejection. This means that the code that runs first should always equate to one of those running.

`myPromis.then(function (result)`
This is our promise now being called. As you can see the object name for our promise is called followed by a function creation.

This function is called depending on what our promise returns. For example, if the return is a **`resolve`** then it will return the message we provide in the **`resolve`** in our promise object. The opposite happens when you have an **`error`** occur.

If you were to copy the above code into your browser you would find that it will print out the words *I'll still be running tho* even though this is the last line of code in our program! That's because the rest of our code will continue to run while our promise runs!

Compulsory Task 1

Follow these steps:

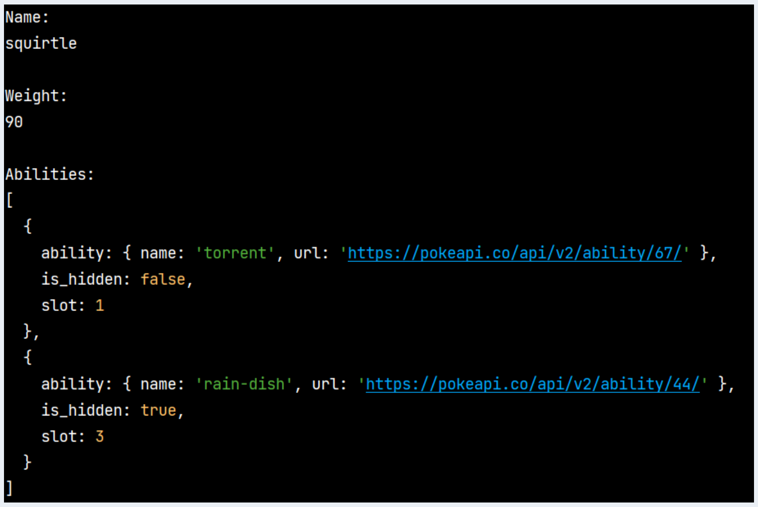
- You're going to use an API call that sends words of affirmation
  - Fetch the API from the following website (**https://www.affirmations.dev/**)
  - Make it so your console outputs a random line for words of affirmation.
  - **Note**: You do not need to create a front end for this task, simply output it to your console.

Compulsory Task 2

Follow these steps:

- You're going to use an API call that gets information about a pokemon.
  - Use this link to call the API: **https://pokeapi.co/api/v2/pokemon/{pokemon}/**
  - Replace the {pokemon} with the name of your favourite Pokemon. Example: **https://pokeapi.co/api/v2/pokemon/absol/**
    - If you don't know any Pokemon you can use this website and find the one you like the most: **https://www.pokemon.com/us/pokedex/**

- Now print out the following about the Pokemon
  - Their name
  - Their weight
  - Their abilities
- The output should look like this



```
Name:
squirtle

Weight:
90

Abilities:
[
  {
    ability: { name: 'torrent', url: 'https://pokeapi.co/api/v2/ability/67/' },
    is_hidden: false,
    slot: 1
  },
  {
    ability: { name: 'rain-dish', url: 'https://pokeapi.co/api/v2/ability/44/' },
    is_hidden: true,
    slot: 3
  }
]
```

- **Note**: You do not need to create a front end for this task, simply output it to your console.