Hyperiondev

TASK

Promises, Async and Await

Visit our website

Introduction

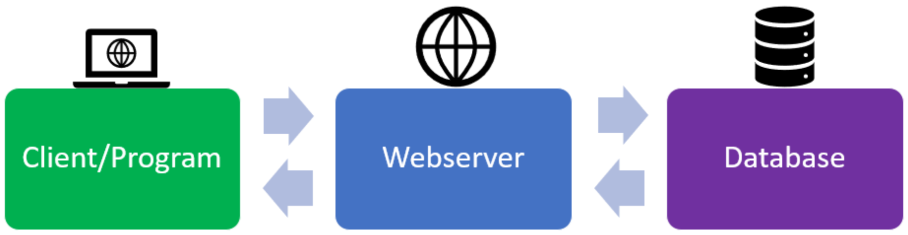
WELCOME TO THE PROMISES, ASYNC AND AWAIT TASK!

In today's task, we will start off by discussing the concept of “promises”. Promises simply help us to perform asynchronous operations such as making requests to an API to retrieve data. Once you understand how promises work asynchronously from the rest of the code, we will move on to another core concept of asynchronous functions; async/await. This is a much easier and more concise approach to creating your own promises while keeping the code easily readable and maintainable.

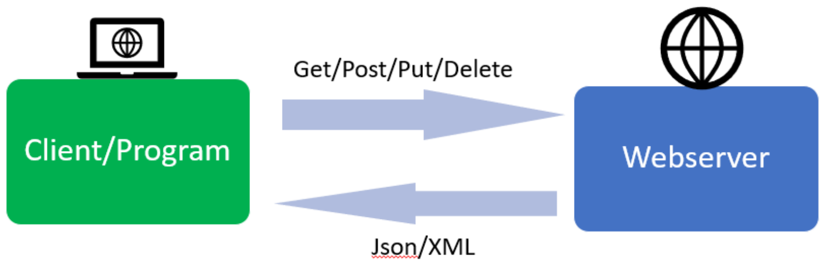
WHAT IS AN API?

Before we delve into how promises and async/await are used to handle asynchronous operations, let’s first discuss what an API is. API stands for **A**pplication **P**rogramming **I**nterface. “Application” refers to any software that interacts with external services and the “Interface” is the point of interaction between the software programs.

Let’s explore an example of an API in action to understand the programming logic. Imagine that you want to search for your favourite song from iTunes. The iTunes application will send this request to the web server. Once the request is sent to the server, the server will find the song in the database and return the song you have requested. Once received, you can work with the data in ways such as saving the song to your favourites list or deleting it from the list. As you can see in the image below, 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 web server. As you can see in the image below, 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.



Imagine that you are an administrator of an online shopping website. Let's take a look at each of the requests to understand what they can be used for in the context of the shopping website:

- **GET** - This is going to request or **read** data that exists within the database. This could be the information about the products on sale.
- **POST** - We may want to **create** new products to sell on our website. Therefore, we will add new details of the product such as the name, price, etc.
- **PUT** - You can also **update** existing data that's on the server, like changing the image of the product.
- **DELETE** - As the name suggests, this will **delete** the product or product item from the website.
- You will later learn more about these CRUD (Create, Read, Update and Delete) operations when you build the server side of your application.

While you can use APIs for many things, there are still limitations. Many websites that have a public domain will make use of **API keys**. API keys set the permissions that users can use to make a request to their server; you can only perform specific actions if the API key you are using allows for it to happen.

Therefore, it's important to always read through the API documentation to get a better understanding of what you are allowed/not allowed to do. Let's look at an example, an extract from typical API documentation for a website:

> ***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. By reading carefully through any API documentation, you can avoid errors caused by limitations you are unaware or unsure of.

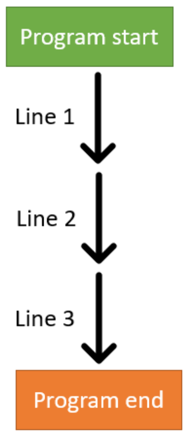
ASYNCHRONOUS VS SYNCHRONOUS

Synchronous or SYNC processing, and asynchronous or ASYNC processing, are two important development concepts. The reason this is part of the content is that learning about promises and async/await involves introducing a new form of programming you're not used to. Let's go over these two concepts, ensuring you have the foundation necessary to understand the core concepts of this task later.

Synchronous Processing

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 the example below, 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.

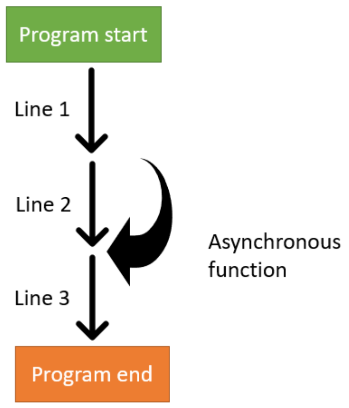


Asynchronous Processing

Asynchronous processing is slightly more complex, as it involves multiple sets of code being run at the same time.

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 to use it properly, 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 that 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 asynchronous function ends under line 2, it won't always end after line 2. Many elements (such as the browser and users' computer speed) 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. **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 are, we can move on to using promises to call an API and use that data in our program. 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.

Calling an API using a promise

Have a look at the code block below:

```
// Create an empty array to store quotes
let items = [];

// Fetch random quotes from the API
fetch("https://api.quotable.io/random")
  //Fetch returns an object which is called res by default
  //Parse the response as a JSON object
  .then((res) => res.json())
  // Callback used to parse the data
  .then((result) => {
     // Assign and store the data in the items array
     items = result;
     // Return the data in the console
     console.log(items);
  } //end arrow function
  ), // end .then
  // Error handling is executed if fetch fails
  (error) => {
    // Return an error in the console
    console.log(error);
  };
```

Don't worry! It looks like a lot, but we're going to break down this code line by line to help you easily understand the concept of a promise and fetch data from an API.

- **`fetch()`**
  In the first line of the central code block, we have a new keyword, "`fetch`". The syntax of `fetch` is `fetch(file)`, where file is the link to the API. `fetch` returns a *promise* which resolves to a response *object*. The link provided to the fetch function in the example above is a link to a random quote generator by **Quotable**. Only text can be sent over the web - we can't send objects. `fetch` thus returns text in a format called JSON (JavaScript Object Notation). JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays. In other words, the fetch call to the API wants to return a data object but can't send that object over the web, and so it sends a text, or JSON, version.

- **`.then((res) => res.json())`**
  As `fetch` keyword is a promise (it promises that it's going to fetch the data), it follows that `then()` is a method of promise instances. The basic syntax of `then()` is:

```
.then(
    (onResolved) => {
        // Some task on success
    },
    (onRejected) => {
        // Some task on failure
    }
) // end of .then
```

In the line of code we're focussing on (**`.then((res) => res.json())`**), `then()` takes the JSON "text" that has been returned by `fetch`, and which we've called **`res`** in the code example. We call the `json()` method on res (**`res.json`**) to convert the JSON into a usable JavaScript object that will allow us to access and work with the data. The **`json()`** method returns this JavaScript object. Here, **`then()`** doesn't take a second parameter (the **`onRejected`** part of the basic syntax above), but returns a promise which we call **`result`**, in a manner known as promise chaining. If you want to understand this in more detail, **have a look here for more on promises and chaining**.

- **`.then((result) => {`**
  Now we come to the second **`then()`**. This uses both of the basic syntax parameters this time (i.e. it has code for `onResolved` as well as `OnRejected`). Let's look at the `onResolved` part first.

  `onResolved` is what will happen if the promise is kept and the API returns a response. In this case the code for the arrow function in this second then will run, with the JavaScript object, **`result`**, that we were just talking about in the previous point being passed into the arrow function. Remember that this JavaScript object is composed of key-value pairs - if we were to print it (**`result`**) out, it would look something like this:

```
{
  _id: 'Ilvlt1O0thk7',
  content: 'We know what we are but know not what we may be.',
  author: 'William Shakespeare',
  tags: [ 'Famous Quotes' ],
  authorSlug: 'william-shakespeare',
  length: 48,
  dateAdded: '2019-09-08',
  dateModified: '2023-04-14'
}
```

The `result` object is passed as a parameter into the expression of the arrow function that gets evaluated; this expression starts from the opening curly bracket and looks as follows :

```
{
   items = result;   // Assign and store the data in the items array
   console.log(items); // Return the data in the console
 }
```

Here, the array called `items` is assigned the contents of the `result` object, with each key-value pair becoming an element in the array. If you were to run the entire code example we initially provided, you'd get output similar to that shown in the code example immediately above, depicting the content of the `items` array.

When you know the keys in the key value pairs (which we do, now, having worked out how to output them to the console) you can access any value in the format `objectname.keyname`. For example, to access only the **quote** key-value pair and output the value (the actual text of the fetched quote) to the console log, we could do as below:

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

- 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 sends it back to us. But what if the website goes offline and we can't fulfil the promise? This is where our error handling comes into play, i.e. the second component of the `then()` method, the `onRejected` part. Take a look at the last few lines of code.

```
  (error) => { // Error handling is executed if fetch fails
  console.log(error);    // Return an error in the console
 };
```

These lines of code will only ever run if `fetch` does not return a result. In the case that the website we're trying to connect to is actually offline, `fetch` will return an error and print it out to a console.

PROMISES USING NORMAL FUNCTIONS

We will now go over one more important concept, creating our own promises. As with an API call using a promise, if we create our own function to check for a promise, it will run asynchronously.

Take a look at the code below:

```
// Create a new Promise object taking in the resolve and reject parameters
let myPromise = new Promise(function (resolve, reject) {
  // Store random number in a variable
  let randNumber = Math.floor(Math.random() * 10);

  // Promise is resolved if the random number is greater than or equal to 5.
  // Else it will be rejected
  if (randNumber >= 5) {
    resolve("Number was greater than or equal to 5 [RESOLVED]");
  } else {
    reject(Error("The number was less than 5 [REJECTED]"));
  }
});

// Return the resolved or rejected results
myPromise.then(
  function (result) {
    console.log(result);
  },
  function (error) {
    console.log(error);
  }
);

// Return message to indicate the code is still running
console.log("I'll still be running though");
```

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

- **`myPromise`**
  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 **`Promise`** object, we create a function that will take two additional parameters.

1. `resolve` - This will run should the promise run correctly.
2. `reject` - If the promise does not work correctly, it'll return an error.

- Once you have created this, you can write any code you may want the action to perform! In this case, we simply check if the `randNumber` is greater than or equal to 1. 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.

- **`myPromise.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`.

If you test the above code in your IDE such as Visual Studio Code, you will find that it will print out the words "`I'll still be running though`" 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.

ASYNC

Now that you understand how to create and use promises, we can now move on to learning about how to easily create asynchronous functions with a lot less effort using a neater and more understandable code base. This can be achieved using the `async` keyword.

You may recall our creation of a promise in the previous section to check whether the return value was accepted or rejected. You may notice how lengthy it was to create. It can also be quite difficult to read and understand without having to read over it a couple of times. What if we told you this could easily be converted into a normal function that performs the exact same thing? That's exactly what async does!

Take a look at the code below:

```
// Define an asynchronous function
async function myAsyncFunction() {
  // Store random number in a variable
  let randNumber = Math.floor(Math.random() * 10);

// Create a condition to check if the random number is greater than or equal to 5
  if (randNumber >= 5) {
    return "Number was greater than or equal to 5 [RESOLVED]";
  } else {
    return "The number was less than 5 [REJECTED]";
  }
}
// Return the result in the console
console.log(myAsyncFunction());
```

Notice how there is no need to create a new promise object and the concept is a lot easier to read and understand. The only difference you may have noticed is that we have the `async` keyword just before our function. This is the only thing we need to let JavaScript know that we want to run this function asynchronously from the rest of the code. From there you can write any code you may want and just return the value you wish to return, as usual.

This is typically why async is the preferred choice for new programmers. Note that this does not mean that you should only ever use async, or always choose it over promises. Let's consider how to decide when to use which.

PROMISES VS ASYNC

While these concepts are very similar to one another, there are a couple of things to keep in mind when deciding which one to use. Take a look at the table below, which will give you a quick overview of the key differences between the two concepts.

	Promise	**Async**
Scope	Only the original promise itself is asynchronous.	The entire function itself is asynchronous.
Logic	Any synchronous work can be completed inside the same callback. Multiple promises make use of the `promise.all()` method.	Synchronous work needs to be moved out of the callback. Multiple promises can be assigned to variables
Error Handling	A combination of then/catch/finally	A combination of try/catch/finally

Which to use?
It's really up to your personal preference at this point in time. As you start going through different API calls and running asynchronous code you will find specific use cases for each. Nonetheless, here is some general advice.

When to use a promise
Promises are a good option should you want to quickly but also concisely grab results from a promise. Promises are a good choice to avoid writing multiple wrapper functions inside an async when a simple `.then` will suffice.

When to use Async
You'll typically make use of asynchronous functions when you are using complex code that needs to run separately from the rest of the code. The simple syntax of asynchronous functions and the fact that this is more familiar to people makes it easier to follow the code.

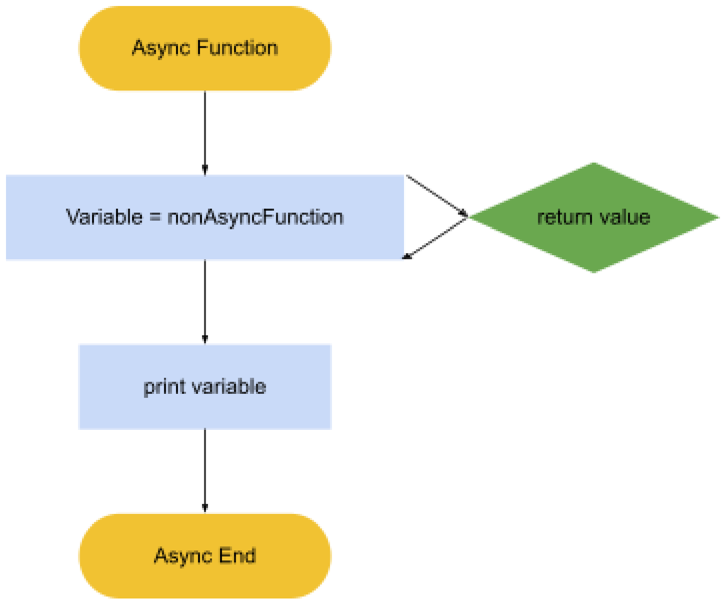
Overall
Use promises when you want to work with simple logic that won't require a lot of processing, and use asynchronous functions for more complex code.

USING AWAIT

Now that you better understand the difference between asynchronous functions and promises, we can introduce the second concept in this task, the `await` keyword.

The await keyword is rather simple to understand. Because an asynchronous function runs code separately from the rest of our code, there is a chance that we may need to have our code wait for another function to run before we can continue.

Take a look at the image below:



Notice how our variable is assigned to the "`nonAsyncFunction`" which is a separate function that returns a new value. We want to end up printing that value out to our program using our async function. However, because of the nature of async functions, it will attempt to print the variable before we have an assigned value for it. This, of course, would cause our code to crash. We can solve this by using the **`await`** keyword.

The await keyword basically tells an async function to wait for the other function's process to complete before running the rest of the code. This is perfect if you require a set of content to continue the process.

Take a look at the code provided in the table below explaining await using functions and arrow head functions:

	Arrow Function	Normal Function
Await	`const returnName = () => {` `  return "spinel";` `};` `const asyncArrowFunction = async` `() => {` `  let myName = await returnName();` `  console.log(myName);` `};` `asyncArrowFunction();`	`function returnNameFunction() {` `  return "spinel";` `}` `async function asyncFunction() {` `  let myName = await` `returnNameFunction();` `  console.log(myName);` `}` `asyncFunction();`
Non-await (possible error)	`const returnName = () => {` `  return "spinel";` `};` `const asyncArrowFunction = async` `() => {` `  let myName = returnName();` `  console.log(myName);` `};` `asyncArrowFunction();`	`function returnNameFunction() {` `  return "spinel";` `}` `async function asyncFunction() {` `  let myName =` `returnNameFunction();` `  console.log(myName);` `}` `asyncFunction();`

The reason that the non-Await may not always produce an error is that some machines/servers process information a lot more quickly than other devices. This is why it's important to always consider different devices and speeds and always account for the usage of await.

FETCHING APIS

As you have learned, asynchronous functions are a form of promises, and as such, can fetch APIs. Take a look at the below code. Notice how it's exactly the same as a promise, just without the `.this` keyword. Notice the word `apiLink`, this is where you would place the API URL. Just don't forget the `await` keyword, or else you will run into issues.

```
// Define an asynchronous function
async function apiFunction() {
  // Await to fetch data from the API
  let item = await fetch("apiLink");
  // Return results the console
  console.log(item);
}
```

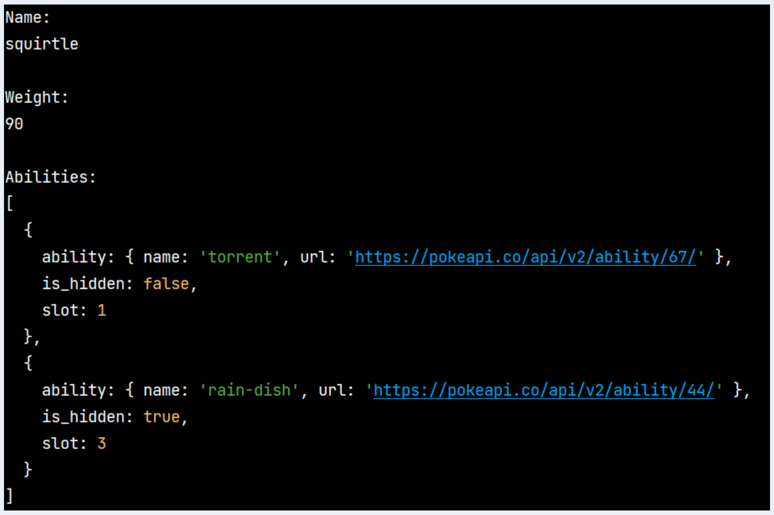
Instructions

In these tasks, you will be fetching data from two different APIs, first using promises and thereafter using async/await. Please note that you **do not** need to create a frontend for these, simply output the data to the console.

Compulsory Task 1

Follow these steps:

- You're going to use promises to make an API call that gets information about a Pokemon.
- Use this link to call the API: **https://pokeapi.co/api/v2/pokemon/squirtle/**
- Note that you can replace the pokemon character {squirtle} with the name of your favourite Pokemon. 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
  }
]
```

- Please remember to submit your answer code.

Compulsory Task 2

Follow these steps:

- Use async/await to fetch data from a URL to randomly generate cat GIFs.
- Use the following API URL to fetch a random GIF of a cat: **http://thecatapi.com/api/images/get?format=src&type=gif**
- Please only output the image URL in the console.
- Please remember to submit your answer code.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.