TASK

Promises, Async, and Await

Visit our website

Introduction

WELCOME TO THE PROMISES, ASYNC, AND AWAIT TASK!

In today's task, we will start by discussing the concept of "promises". Promises allow us to perform operations that may take time, like making requests to an API to retrieve data. Once you understand how promises work without stopping the rest of the code from running, we will move on to another core concept: async/await. This is a simpler and more concise way to create promises, making your code easier to read and manage.

WHAT IS AN API?

Before we look into how promises and async/await are used to handle operations that can take some time to complete, let's first discuss what an API is. **API** stands for **a**pplication **p**rogramming **i**nterface. An "application" refers to any software that interacts with other software or external services, and the "interface" is the point where these interactions occur, allowing different programs to communicate with each other.

Here's an example of an API in action: When you search for a song on iTunes, the application sends a request to the server. The server then finds the song in its database and returns it to you. After receiving it, you can save it to your favourites or delete it. Each step is part of the process of connecting your request to the server.

API in action

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.

Types of requests

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:** This method allows you to **update** existing data on the server, such as changing the image of a product or updating the entire product details.
- **PATCH:** Unlike PUT, which updates the entire resource, PATCH is used to make partial updates to existing data. For example, you can update just the price of a product without affecting other details.
- **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 APIs can be used for many different tasks, there are some limitations, especially when it comes to access and security. Many websites use **API keys** to control who can use their API. An API key is a special code, like a unique password, that you include in your requests to the API. It helps identify you to the API and checks what you're allowed to do. Using an API key only allows you to perform actions that the key allows, and the permissions for the key will be set by the API.

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

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

As you can see in our made-up example above, this documentation explains how many requests you can make using their website's API. By reading carefully through any API documentation, you can avoid errors caused by limitations you are unaware or unsure of.

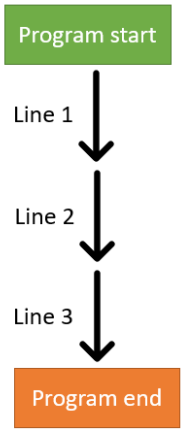
ASYNCHRONOUS VS SYNCHRONOUS

Synchronous (sync) and asynchronous (async) processing are key concepts in development. Understanding them is essential when learning promises and async/await, which introduce a different way of programming. Let's go over these two ideas to build a strong foundation for later tasks.

Synchronous processing

Synchronous processing is a familiar programming approach where code runs in order, line by line. Each line of code waits for the one before it to finish before running. The line below the current line of code won't run until the current line of code is finished running.

This approach is safer because the flow of control moves predictably through the code. Each line runs one after the other from start to finish, which should feel familiar since that's how you've been writing your JavaScript code so far.



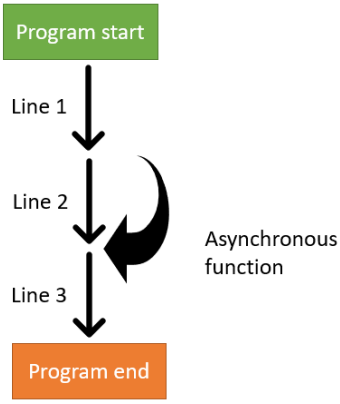
Synchronous processing

Asynchronous processing

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

It's important to avoid using asynchronous processing unless it's necessary and safe. If not used properly, it can lead to more errors than synchronous programming because different parts of the code might run at the same time and also depend on each other, causing issues.

Take a look at the image below:



Asynchronous processing

The above image represents the same program as the previous image, 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 the user's 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 understand how APIs work and the difference between synchronous and asynchronous processing, we can move on to using promises to call an API and work with the 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. Promises can be used in many situations, like calling functions, but the best way to explain them is through API calls.

Calling an API using a promise

Have a look at the code block below:

```
// Fetch a random quote from the API
fetch("https://zenquotes.io/api/today")
  // Convert the response to a JSON object
  .then((response) => response.json())
  // Handle the parsed JSON data
  .then((result) => {
    // Store the fetched data in the items array
    let items = result;
    // Log the stored data to the console
    console.log(items);
  }) // End of the second .then

  // Handle any errors that occur during the fetch
  .catch((error) => {
    // Log the error to 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 code above, we use the keyword `fetch`. The syntax of `fetch` is `fetch(url)`, where `url` is the link to the API you want to get data from. For the above example, the link provided to the `fetch` function is from the **ZenQuotes API**, which provides random quotes.

  When using fetch, it sends a request to the URL and returns a promise, which eventually provides a response. This response is an object that contains the

data from the server. Since only text can be sent over the web, not objects directly, the data is often sent 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 key–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, so it sends a text, or JSON, version.

When fetch calls the ZenQuotes API, it receives the data in JSON format. This JSON text can then be converted back into a JavaScript object, making it easy to work with the data in the code.

- `.then((response) => response.json())`

  After the `fetch` function makes a request to retrieve the data, it returns a promise. The **`.then()`** method is used for chaining actions to this promise, meaning you can specify what should happen next once the promise is fulfilled. 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 focusing on, `.then()` is used to handle the successful completion of the fetch request. It takes the data returned by fetch, referred to as `response` in the code, and calls the `.json()` method on it. This `.json()` method converts the JSON text received from the fetch call into a usable JavaScript object, which allows us to work with the data more easily.

  `.then()` can take up to two different parameters, one for handling successful promises (`onResolved`) and another for handling errors (`onRejected`). In the above example, `.then()` is used with just the first parameter because we're focusing on what happens when the promise is resolved successfully, without specifying an error handler here.

Additionally, the `.json()` method itself returns a promise because converting the data might take some time, and this promise is then passed on in a process known as promise chaining. This allows you to chain additional `.then()` calls to handle the data further, making your code more organised and easier to read. If you want to understand this in more detail, **have a look for more on using promises and chaining**.

- `.then((result) => {`

Now we come to the second `.then()` block. This `.then()` uses only the `onResolved` parameter, as it is meant to handle the data when the promise is successfully fulfilled.

The `onResolved` part specifies what will happen if the promise is kept and the API returns a response. In this case, the code inside the arrow function of the second `.then()` block will execute once the JSON response has been successfully parsed. The arrow function takes the `result` parameter, which is an array of objects returned by the API.

Essentially, `result` holds the actual data we want to work with, and the function processes this data to perform further actions, such as storing it in a variable or displaying it in the console. Remember that this JavaScript object is composed of key–value pairs – if we were to print the `result` out, it would look something like this:

```
[
  {
    q: 'It is not fair to ask of others what you are not willing to
do yourself.',
    a: 'Eleanor Roosevelt',
    h: '<blockquote>&ldquo;It is not fair to ask of others what you
are not willing to do yourself.&rdquo; &mdash; <footer>Eleanor
Roosevelt</footer></blockquote>'
  }
]
```

In our example in this section, the variable `items` has been assigned the value stored within `result`. By doing this, `items` will hold the array of objects that has been received from the API.

Within the arrow function, `console.log(items);` is used to print out this array so you can see what it looks like in the console. This helps to check if the data has been received correctly.

For us to get specific information from the received array we would need to look inside a specific object. For example, to access the quote for the first object within the array the following could be added within the arrow function:

```
console.log(items[0].q);
```

- As you can see, the code begins with a promise to fetch data from the API. It then processes the data through a series of `.then()` methods to handle the response. But what if the website goes offline and we can't fulfil the promise? This is where error handling comes into play using the `.catch()` method.

  The `.catch()` method is specifically designed to handle errors that occur during a promise chain. Unlike the `onRejected` parameter of `.then()`, `.catch()` is its own separate method and is used at the end of the promise chain to catch any errors that might occur at any point within the preceding promises.

  Here is the relevant part extracted from the code example:

```
.catch((error) => {
    // Log the error to the console
    console.log(error);
  });
```

  The above block of code will only run if something goes wrong during the fetch operation or in any of the `.then()` calls. For example, if the website we're trying to connect to is offline, or if there's a problem parsing the JSON, `.catch()` will catch the error and log it to the console.

  Using `.catch()` ensures that any errors that are encountered during the entire process from fetching the data to handling it are appropriately managed.

Let's now learn about how to create our own promises. But before we do, **watch this helpful video explainer** on how JavaScript promises manage asynchronous operations and how async/await provides a concise syntax to handle them.

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. A Promise represents an asynchronous operation.
// It can either succeed (resolve) or fail (reject).
let myPromise = new Promise(function (resolve, reject) {
  // Generate a random number between 0 and 9
  let randNumber = Math.floor(Math.random() * 10);

  // If the random number is greater than or equal to 5, resolve the promise
(success).
  // Otherwise, reject the promise (failure).
  if (randNumber >= 5) {
    resolve("Number was greater than or equal to 5 [RESOLVED]"); // Promise
successfully resolved
  } else {
    reject(Error("The number was less than 5 [REJECTED]")); // Promise rejected
with an error
  }
});

// Handle the result of the promise (either resolved or rejected).
// .then() takes two functions:
// - The first handles the resolved value (success).
// - The second handles the rejected value (error).
myPromise.then(
  function (result) {
    console.log(result); // Log the success message if resolved
  },
  function (error) {
    console.log(error); // Log the error message if rejected
  }
);

// This message runs immediately, demonstrating that the code after the promise
// continues to run while the promise is still pending (asynchronous
behaviour).
console.log("I'll still be running though");
```

Once again we're going to break down this code for you:

- `let myPromise = new Promise(function (resolve, reject) {`

  This line creates a promise and defines the logic that will run when the promise is executed. This is the default style whenever you create a promise. Within the parameters of the `Promise` constructor, we pass a function that takes the parameters `resolve` and `reject`. These two parameters are not just values, instead, they are functions that will be called depending on the result of the asynchronous operation.

  1. `resolve`: This function will be run when the promise successfully completes.
  2. `reject`: This function will be run if the promise encounters an error or doesn't complete as expected.

- Once this has been created, 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. It's important to remember that every promise must have both `resolve` and `reject` functions (think of these as the return statements for a promise). The promise function should return either the `resolve` value or the `reject` value. This means the code inside the promise should always lead to one of these outcomes.

- ```
  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.

When you use `.then()`, you're telling your code what to do after a promise finishes. A promise represents a task that will be completed in the future (either successfully or with an error).

- **If the promise is successful:** You define what should happen after the task is done by providing a function that will run, like saying, "Once this is done, do this."
- **If the promise fails:** You also have the option to handle errors by specifying another function, which is like saying, "If it goes wrong, do this."
  ```

Example using **function keyword** syntax:

```
myPromise().then(
  // Once this is done, print the result.
  function (result) {
    console.log(result);
  },
  // If it goes wrong, print the error.
  function (error) {
    console.log(error);
  }
);
```

Example using the **arrow function** syntax:

```
myPromise().then(
  // Once this is done, print the result.
  (result) => console.log(result),

  // If it goes wrong, print the error.
  (error) => console.log(error)
);
```

If you test the above code in your IDE (e.g., 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 know how to use promises, let's move on to creating asynchronous functions more easily with cleaner and simpler code using the **async** keyword.

In the previous section, creating a promise was lengthy and complex. The **async** keyword lets you convert this into a regular function that does the same thing but in a more straightforward way.

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().then(function (result) {
    console.log(result);
  })
);

// This message runs immediately, demonstrating that the code after the promise
// continues to run while the promise is still pending (asynchronous
behaviour).
console.log("I will still be running");
```

With **async**, there's no need to create a new promise object, making the code simpler and easier to read. Just add the **async** keyword before the function to run it asynchronously. You can write your code and return the value as usual.

Async is often preferred by new programmers, but it's not the only option. Let's explore when to use **async** versus when to use promises.

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 performed inside the same callback that defines the promise.	Any synchronous work can be performed inside the async function along with asynchronous operations.
Error Handling	Promises can handle errors using a combination of `.then()`, `.catch()`, and `.finally()`.	Error handling in async functions is done using a combination of `try`/`catch`/`finally`.

Which one should you use?

It's really up to your personal preference as a beginner programmer. 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.

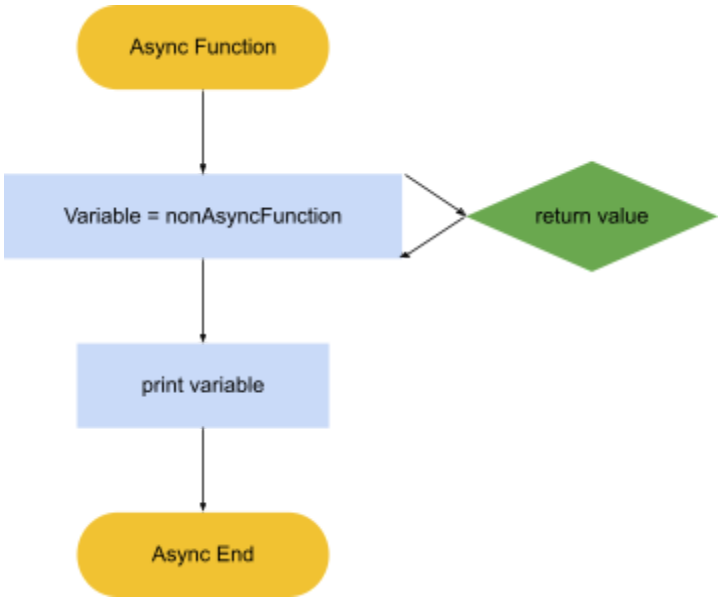
Overall, use promises when you want to work with simple logic that won't require a lot of processing, and use asynchronous functions for more complex code.

USING AWAIT

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

The `await` keyword is comparatively 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:



Async function with a non-async variable

Notice how our variable in the image above 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 below demonstrating `await` using arrow functions and normal functions.

Arrow function

Await:

```
// This arrow function simulates an asynchronous operation
const returnName = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("spinel");  // After 2 seconds, resolve with the name "spinel"
    }, 2000); // 2-second delay
  });
};

const asyncArrowFunction = async () => {
  console.log("Waiting for the name...");  // Indicate waiting starts
  // Awaiting the result from the promise returned by returnName
  let myName = await returnName();
  console.log("Finished waiting!");  // Indicate waiting ends
  console.log(myName);  // Prints "spinel" after the delay
};

asyncArrowFunction();
```

Non-await (possible error):

```
// This arrow function simulates an asynchronous operation
const returnName = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("spinel");  // After 2 seconds, resolve with the name "spinel"
    }, 2000); // 2-second delay
  });
};

const asyncArrowFunction = async () => {
  console.log("Not waiting for the name...");  // Indicate no waiting
  // Not waiting for the promise to resolve
  let myName = returnName();
  console.log(myName);  // Prints the pending promise, not the name
};

asyncArrowFunction();
```

Normal function

Await:

```
// This function simulates an asynchronous operation
function returnNameFunction() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("spinel");  // After 2 seconds, resolve with the name "spinel"
    }, 2000); // 2-second delay
  });
}

async function asyncFunction() {
  console.log("Waiting for the name...");  // Indicate waiting starts
  // Awaiting the result from the promise returned by returnNameFunction
  let myName = await returnNameFunction();
  console.log("Finished waiting!");  // Indicate waiting ends
  console.log(myName);  // Prints "spinel" after the delay
}

asyncFunction();
```

Non-await (possible error):

```
// This function simulates an asynchronous operation
function returnNameFunction() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("spinel");  // After 2 seconds, resolve with the name "spinel"
    }, 2000); // 2-second delay
  });
}

async function asyncFunction() {
  console.log("Not waiting for the name...");  // Indicate no waiting
  // Not waiting for the promise to resolve
  let myName = returnNameFunction();
  console.log(myName);  // Prints the pending promise, not the name
}

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 FROM AN API

When you need to get data from an API (like weather or movie info), one way to do this is by using **`fetch()`**. When fetching data from an API the process takes time, so we use **`async`** functions along with **`await`** to handle the delay. This helps prevent errors caused by trying to use data that hasn't fully arrived yet, making it easier to interact with an API.

Example of using **`fetch()`** (be sure to replace "**`apiLink`**" with a valid API URL):

```
async function apiFunction() {
  // Use fetch() to ask the API for data and wait for it to return
  let response = await fetch("apiLink"); // Replace "apiLink" with a valid URL

  // Convert the response into a usable format (like JSON)
  let data = await response.json();

  // Display the fetched data in the console
  console.log(data);
}

apiFunction();
```

Explanation:

The function begins with the **`async`** keyword, indicating that it contains asynchronous operations. Within this function, we use **`fetch("apiLink")`** to request data from the specified API URL (which should be replaced with a valid link).

Using **`await`** with the **`fetch()`** function pauses the execution of the code until the API responds. This makes it possible to avoid moving forward before the data is ready. This is crucial because if the **`await`** keyword is not used, JavaScript will continue executing the next lines of code, which could lead to errors when trying to access data that hasn't been received yet.

Once the response has been received, `response.json()` is used to convert the raw data into a usable JavaScript object. The `await` keyword is once again used so that the conversion first takes place before proceeding.

Finally, `console.log(data)` is used to display the fetched data in the console. This output will make it possible to confirm that the data has been successfully retrieved and processed. In a real application, this data could be used to perform further operations.

Take note

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click “Request review” on your student dashboard, you will receive a 100% pass grade if you’ve submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you’ve done that, feel free to progress to the next task.

Instructions

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

Auto-graded task 1

Follow these steps:

- You're going to use promises to make an API call that gets information on Pokémon.
- Use this link to call the API: **https://pokeapi.co/api/v2/pokemon/squirtle/**
- Note that you can replace the Pokémon character (Squirtle) with the name of your favourite Pokémon. If you don't know any Pokémon, you can **use this website** and find the one you like the most.
- Now print out the following about the Pokémon:
    - Name
    - Weight
    - Name of an ability, e.g., "Overgrow"

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded 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.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

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.