## **ASYNC AND AWAIT**

### **Introduction**

Asynchronous code can become difficult to follow when it has a lot of things going on. async and await are two keywords that can help make asynchronous reads more like synchronous code. This can help code look cleaner while keeping the benefits of asynchronous code.

For example, the two code blocks below do the exact same thing, they both get information from a server, process it, and return a promise.

function getPersonsInfo(name) { return server.getPeople().then(people => { return people.find(person => { return person.name === name }); }); }

async function getPersonsInfo(name) { const people = await server.getPeople(); const person = people.find(person => { return person.name === name }); return person; }

The second example looks much more like the kind of functions you are used to writing, however, did you notice the async keyword before the function declaration? How about the await keyword before server.getPeople()?

### **Learning Outcomes**

1. How do you declare an async function?
2. What does the async keyword do?
3. What does the await keyword do?
4. What is returned from an async function?
5. What happens when an error is thrown inside an async function?
6. How can you handle errors inside an async function?

### **The async keyword**

The async keyword is what lets the javascript engine know that you are declaring an asynchronous function, this is required to use await inside any function. When a function is declared with async, it automatically returns a promise, returning in an async function is the same as resolving a promise, likewise, throwing an error will reject the promise.

An important thing to understand is async functions are just syntactical sugar for promises.

The async keyword can also be used with any of the ways a function can be created, said differently: it is valid to use an async function anywhere you can use a normal function. Below you will see some examples that may not be intuitive, if you don’t understand them, come back and take a look when you are done with the assignments.

const yourAsyncFunction = async () => { // do something asynchronously and return a promise return result; }

anArray.forEach(async item => { // do something asynchronously for each item in 'anArray' // one could also use .map here to return an array of promises to use with 'Promise.all()' });

server.getPeople().then(async people => { people.forEach(person => { // do something asynchronously for each person }); });

### **The await keyword**

await is pretty simple: it tells javascript to wait for an asynchronous action to finish before continuing the function. It’s like a ‘pause until done’ keyword. The await keyword is used to get a value from a function where you would normally use .then(). Instead of calling .then() after the asynchronous function, you would simply assign a variable to the result using await, then you can use the result in your code as you would in your synchronous code.

### **Error Handling**

Handling errors in async functions is very easy. Promises have the .catch() method for handling rejected promises, and since async functions just return a promise, you can simply call the function, and append a .catch() method to the end.

asyncFunctionCall().catch(err => { console.error(err) });

But there is another way: the mighty try/catch block! If you want to handle the error directly inside the async function, you can use try/catch just like you would inside synchronous code.

async function getPersonsInfo(name) { try { const people = await server.getPeople(); const person = people.find(person => { return person.name === name }); return person; } catch (error) { // Handle the error any way you'd like } }

Doing this can look messy, but it is a very easy way to handle errors without appending .catch() after your function calls. How you handle the errors is up to you, and which method you use should be determined by how your code was written. You will get a feel for what needs to be done over time. The assignments will also help you understand how to handle your errors.

### **Practice**

Remember the Giphy API practice project? (If not, you should go back and complete the API lesson) We are going to convert the promise based code into async/await compatible code. Here’s a refresher of the code we are starting with:

<script> const img = document.querySelector('img'); fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { return response.json(); }) .then(function(response) { img.src = response.data.images.original.url; }); </script>

Since await does not work on the global scope, we will have to create an async function that wraps our API call to Giphy.

<script> const img = document.querySelector('img'); async function getCats() { fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { return response.json(); }) .then(function(response) { img.src = response.data.images.original.url; }) } </script>

Now that we have a function that is asynchronous, we can then start refactoring from using promises to using await:

<script> const img = document.querySelector('img'); async function getCats() { const response = await fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}); response.json().then(function(response) { img.src = response.data.images.original.url; }); } </script>

Since response is still the same object we have passed to the .then() block at the start, we still need to use the .json() method, which in turn returns a promise. Because .json() returns a promise, we can use await to assign the response to a variable.

<script> const img = document.querySelector('img'); async function getCats() { const response = await fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}); const catData = await response.json(); img.src = catData.data.images.original.url; } </script>

To use this function, we just simply need to call it with getCats() in our code.

<script> const img = document.querySelector('img'); async function getCats() { const response = await fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}); const catData = await response.json(); img.src = catData.data.images.original.url; } getCats(); </script>

This code will behave exactly like the code from the last lesson, it just looks a bit different after refactoring. async/await are very useful tools when it comes to cleaning up asynchronous javascript code. It is important to remember async/await are just promises written in a different way. Do the assignments below, and dive deeper into the understanding of async/await.

### **Assignment**

1. Read [this article](https://javascript.info/async-await) for a solid introduction to async/await. [This article](https://codeburst.io/javascript-es-2017-learn-async-await-by-example-48acc58bad65) also has some good examples of its use.
2. Read [this article](https://pouchdb.com/2015/03/05/taming-the-async-beast-with-es7.html) for a more in-depth look at async/await, including how to handle errors.
3. Watch [this video](https://www.youtube.com/watch?v=9YkUCxvaLEk) for a good overview on async/await and it’s purpose, along with a special trick.

### **Additional Resources**

1. [This video](https://www.youtube.com/watch?v=COKdtOgopWQ) is an example of how you can change callbacks, to promises, to async/await.
2. [This video](https://www.youtube.com/watch?v=vn3tm0quoqE) gives a comprehensive view of Promises, async, and await.