JS Async Functions

Learning Objectives

- Understanding how asynchronous code works
- Working with promises
- Using the `async` and `await` keywords

Asynchronous Code

Asynchronous code is code that runs in the background. This is useful for tasks that can take a long time to complete, but don't need to block the main thread.

JavaScript is a single-threaded language, meaning that only one thing can happen at a time.

Blocking the main thread is bad because it prevents the user from interacting with the page because no other JavaScript code can be executed. Examples of asynchronous code include: network requests, file system access, animations and timers.

Promises

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value. Most of the time it is returned by a function that performs an asynchronous operation.

The Promise object has the following properties and methods:

Property / Method	Description
`state`	the state of the Promise object, can be `"pending"`, `"resolved"` or `"rejected"`
`result`	the result of the asynchronous operation (you'll almost never need to access this directly)
`then()`	a method that takes a callback function that will be called when the asynchronous operation is complete
`catch()`	a method that takes a callback function that will be called when the asynchronous operation fails
`finally()`	a method that takes a callback function that will be called when the asynchronous operation is complete, regardless of whether it was successful or not

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

> 💡 Promises are almost always created for you by other asynchronous APIs, only rarely do you create them yourself. If you create a Promise yourself (`new Promise()`), you either know exactly what you are doing, or you are probably doing something wrong.

Asnyc Functions and the `await` Keyword

Asnyc functions are a syntactic sugar for Promises. Using the `await` keyword, you can write asynchronous code that looks synchronous. Any function can be prefixed with the `async` keyword:

```
async function myAsyncFunction() {
  // ...
}

const myAsyncArrowFunction = async () => {
  // ...
};
```

Inside an async function, you can use the `await` keyword to wait for a Promise to be resolved:

```
async function myAsyncFunction() {
  const value = await functionThatReturnsAPromise();
  console.log(value);
}
```

This is can be easier to read than the Promise syntax especially when you have multiple asynchronous operations that depend on each other.

```
async function myAsyncFunction() {
  const value1 = await functionThatReturnsAPromise1();
  const value2 = await functionThatReturnsAPromise2(value1);
  const value3 = await functionThatReturnsAPromise3(value2);
  console.log(value3);
}
```

Compared to:

```
// avoid doing this:
function myFunction() {
  functionThatReturnsAPromise1()
    .then((value1) => {
      return functionThatReturnsAPromise2(value1);
    })
    .then((value2) => {
      return functionThatReturnsAPromise3(value2);
```

```
    })
    .then((value3) => {
      console.log(value3);
    });
}
```

> 💡 `async` functions always return a Promise. If the function returns a value, the Promise will be resolved with that value. Even if you're not using the `return` keyword the function will return a Promise that resolves to `undefined` when it reaches the end of it's scope. If the function throws an error, the Promise will be rejected with that error.

Handling Errors

When using Promises, you can use the `catch()` method to handle errors. Using `async` functions, you can use the `try`/`catch` syntax.

`try`/`catch`

```
async function myAsyncFunction() {
  try {
    const value = await functionThatReturnsAPromise();
    console.log(value);
  } catch (error) {
    console.error(error);
  }
}
```

The `try` block will be executed, and if an error is thrown, the `catch` block will be executed. The `catch` block has access to the error that was thrown.

If any error is thrown in the `try` block, the `catch` block will be executed immediately aborting the execution of any further code in the `try` block:

```
async function functionThatThrowsAnError() {
  throw new Error("ooops 😱");
}

async function myAsyncFunction() {
  try {
    const value = await functionThatThrowsAnError();
    // The following code will never be executed because
    // `functionThatThrowsAnError()` throws an error.
    // The execution will jump to the `catch` block.
    console.log(value);
    const value2 = await functionThatReturnsAPromise2();
    console.log(value2);
  } catch (error) {
```

```
    console.error(error);
  }
}
```

`finally`

You can also use a `finally` block after any `try` block to run code after the `try` block is resolved, regardless of whether it was successful or not:

```
async function myAsyncFunction() {
  try {
    const value = await functionThatReturnsAPromise();
    console.log(value);
  } catch (error) {
    console.error(error);
  } finally {
    console.log("done");
  }
}
```

`try`/`catch` is not limited to `async` Functions

The `try`/`catch`/(`finally`) syntax is not limited to `async` functions, you can use it with any JavaScript code that might throw an error.

```
function functionThatThrowsAnError() {
  throw new Error("ooops 😱");
}

function myFunction() {
  try {
    functionThatThrowsAnError();
    console.log("this will never be executed");
  } catch (error) {
    console.error(error);
  } finally {
    console.log("done");
  }
}
```

Parallel Promises

`Promise.all()`

If you have multiple asynchronous operations that you want to run in parallel, you can use `Promise.all()`.

`Promise.all()` takes an array of Promises and returns a Promise that resolves to an array of the results of the Promises in the same order.

```
async function myAsyncFunction() {
  try {
    const values = await Promise.all([
      functionThatReturnsAPromise1(),
      functionThatReturnsAPromise2(),
      functionThatReturnsAPromise3(),
    ]);
    console.log(values); // [value1, value2, value3]
  } catch (error) {
    console.error(error);
  }
}
```

This will run all three asynchronous operations in parallel, and wait for all of them to complete before continuing. If any of the asynchronous operations fail, `Promise.all()` will fail as well.

Controlling when to Resolve or Reject Parallel Promises

There are advanced use cases where you might want to control when to resolve or reject a Promise. `Promise.allSettled()` (does not care if Promises are rejected or resolved) and `Promise.any()` (resolves once the first Promise is resolved) are two methods that allow you to do that.

Asynchronous Browser APIs

Here are some examples of asynchronous browser APIs that return a Promise:

- `fetch()` — makes an HTTP request and returns a Promise that resolves to a `Response` object
- `element.animate().finished` — animates an element and returns a Promise that resolves when the animation is complete
- `navigator.getBattery()` — gets the current battery level and returns a Promise that resolves with the battery level

---

Resources

- Thread on mdn
- Asynchronous on mdn
- Using Promises on mdn
- Async functions on mdn
- Promise.all() on mdn
- try...catch on mdn