

# Exploration — Writing Asynchronous Code

## Introduction

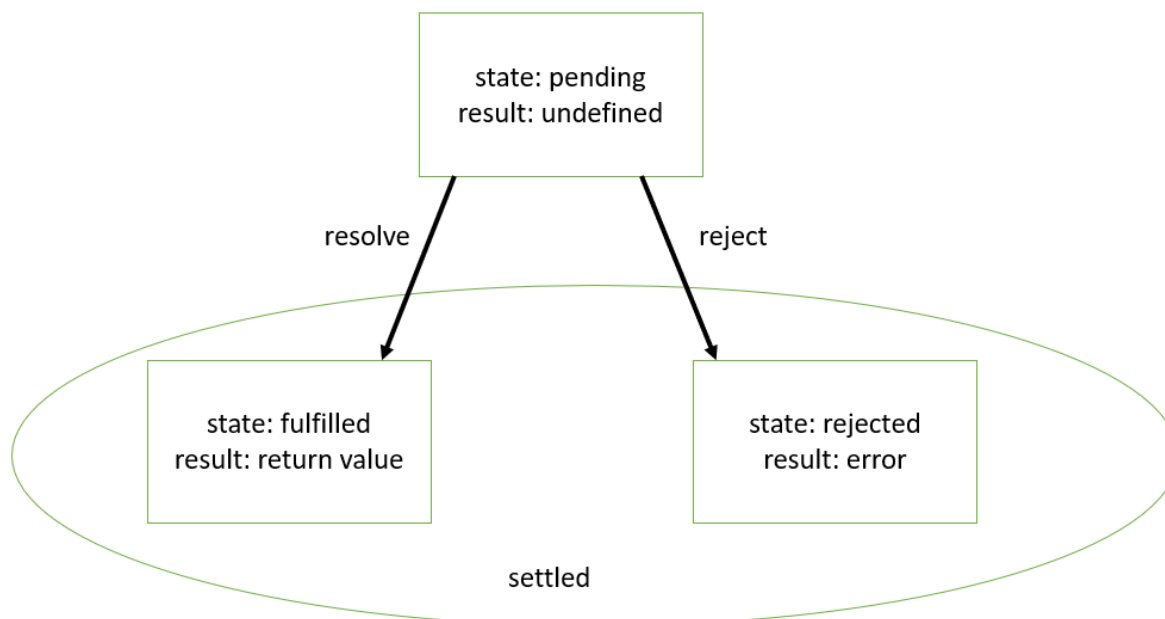


In this exploration, we look at two important language features provided by JavaScript to help write asynchronous code, **Promises** and **await/async**.

## Promises

JavaScript introduced the feature of promises to make it easier to write asynchronous code. A `Promise` is a JavaScript object that represents the result of an asynchronous operation whose results may not be available as yet. The core idea is this:

- The asynchronous function returns a `Promise` object.
- The initial state of this `Promise` object is `pending` meaning it does not yet have a result.
- When the asynchronous function successfully finishes, it fills in the `Promise` object with the result and sets its state to `fulfilled`. The promise is now said to have been **fulfilled**. The `Promise` is said to have resolved to this result value.
- When the asynchronous function fails due to an error, it will not produce a value. In this case, the state of the `Promise` object is set to `rejected`. The promise is said to have been **rejected**.
- Once the state of the `Promise` objects is set to either the `fulfilled` or `rejected`, it is said to have been **settled**.

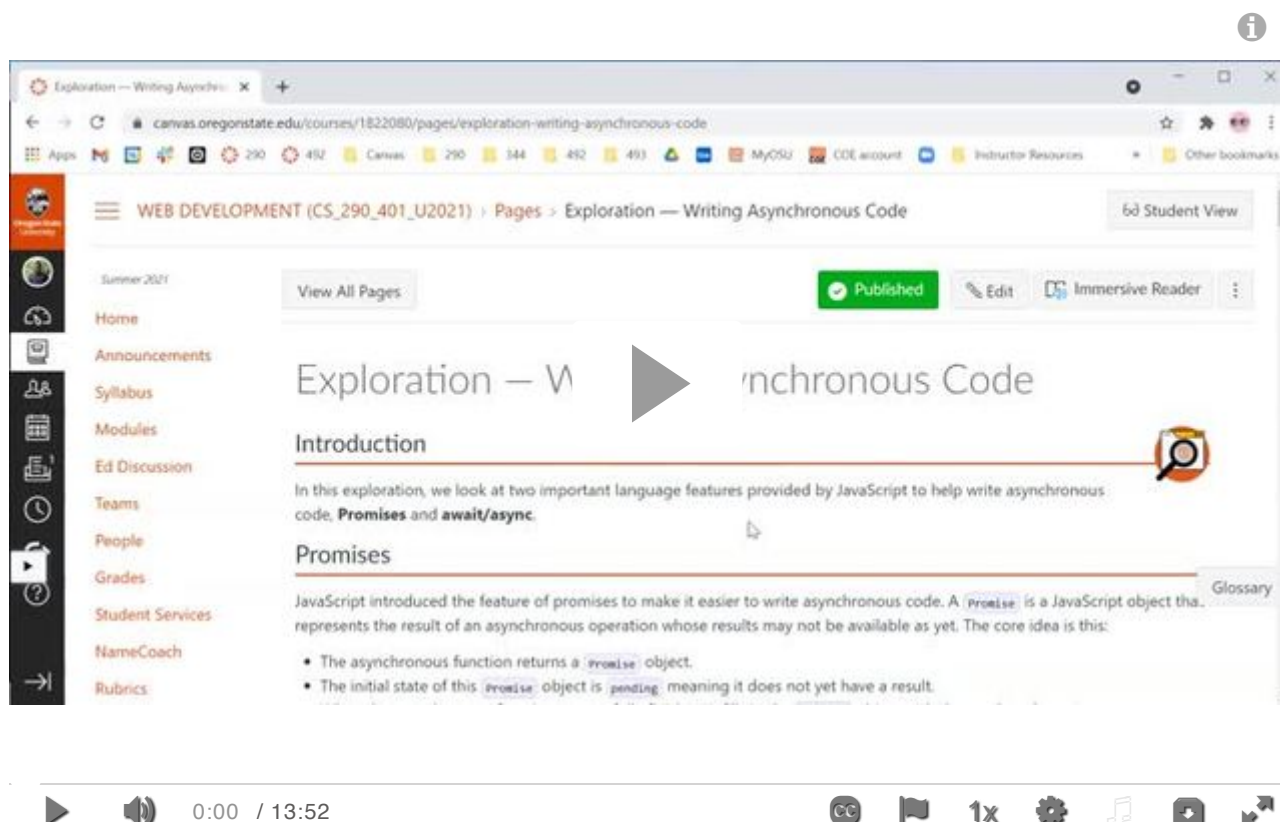


We can create `Promise` objects using the constructor `Promise`. However, it is much more common that we will write code that uses library functions that return promises, rather than us writing functions which return promises. For this reason, let us focus on how we call functions which return promises.

## Obtaining Promise Results

When we call a function which returns a promise, we provide functions that will be executed when the promise settles.

- Using the `then()` method on the promise, we provide a function that will be called when the promise returns a result, i.e., it is fulfilled. If the function provided to `then()` returns another promise, we can process its result by calling the `then()` method again.
- Using the `catch()` method on the promise, we provide a function that will be called when the promise fails, i.e., it is rejected.

A screenshot of a web browser displaying a Canvas LMS page. The page title is "Exploration — Writing Asynchronous Code". The URL is "canvas.oregonstate.edu/courses/1822080/pages/exploration-writing-asynchronous-code". The page shows a sidebar with navigation links like Home, Announcements, Syllabus, Modules, Ed Discussion, Teams, People, Grades, Student Services, NameCoach, and Rubrics. The main content area has a header "WEB DEVELOPMENT (CS\_290\_401\_U2021) > Pages > Exploration — Writing Asynchronous Code" and a "6d Student View" button. Below this is a "View All Pages" button and a "Published" status. The main content area has a large heading "Exploration — Writing Asynchronous Code" with a play button icon. Below this is an "Introduction" section with the text "In this exploration, we look at two important language features provided by JavaScript to help write asynchronous code. Promises and await/async." and a "Promises" section with the text "JavaScript introduced the feature of promises to make it easier to write asynchronous code. A Promise is a JavaScript object that represents the result of an asynchronous operation whose results may not be available as yet. The core idea is this:" followed by two bullet points: "• The asynchronous function returns a Promise object." and "• The initial state of this Promise object is pending, meaning it does not yet have a result." There is also a "Glossary" link on the right.

## Example: Obtaining Promise Results

In the past, the `XMLHttpRequest` object was widely used to make asynchronous HTTP requests in client-side code. However, modern browser support the [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) ([https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)) which is a lot more powerful and easy to use than `XMLHttpRequest`. Therefore, modern client-side JavaScript code typically uses the Fetch API instead of using `XMLHttpRequest`. With the Fetch API, we make an HTTP request by calling the

function **fetch** [\\_ \(https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch\)](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch)).

- The function `fetch()` has one required parameter identifying the resource for which an HTTP request would be sent.
  - In our example, we will provide a *URL* for this parameter
- By default, `fetch()` sends a request with the `GET` method and that's what we will use in our example.
  - However, we *can* provide an additional parameter to configure the request. We can use this additional parameter to send a request with a different HTTP method, such as `POST`, to set request headers, or to send a request body, etc.
- `fetch()` returns a promise.
  - If the request is successful, i.e., a response is received, the promise is fulfilled and the value of the promise is resolved to the HTTP response.
  - If the HTTP request fails, e.g., due to network issues or a malformed URL, then the promise is rejected and an error is returned.

In the following example, we modify our previous example and replace `XMLHttpRequest` with `fetch()` and remove the checkbox from the page. The changes are discussed below the repl.

▶ Run

open in  repl.it



index.js ×



```
1 'use strict';
2
3 const PORT = 3000;
4 const express = require("express");
  const app = express();
5 app.use(express.static('public'));
6
  const items = ['MongoDB', 'Express', 'React', 'Node'];
7
```



https://m612--coecs290.repl.co



Console Shell

Show alert

Send request to server



Let us examine the files that comprise the web app:

- `index.js`
  - This is exactly the same program as used in the previous example.
- `index.html`
  - This page is slightly modified from the previous example.
  - In the previous example we had a checkbox that controlled whether the request is to be sent synchronously or asynchronously.
  - This checkbox has been removed because `fetch()` returns a promise and the request to the server program will always be asynchronous in the current example.
- `fetch.js`
  - Similar to the previous example, submitting the form in `index.html` results in calling the handler `callServer()`.
  - Again, the function `callServer()` calls the function `getData()` two times to make the actual HTTP requests.
  - The crucial change is in `getData()` which now calls `fetch()` to make the HTTP request.

Let us look at the code of the function `getData()` which is reproduced below:

```
function getData(url) {  
  return fetch(url)           // line 1  
    .then(response => response.text()) // line 2  
    .then(data => addData(data)) // line 3  
    .catch(error => console.error(error)); // line 4  
}
```

- In line 1 of the function we call `fetch()` which returns a promise.
- In line 2 we call the `then()` method on this returned promise.
  - We pass one parameter to the method `then()`.
  - This parameter is the function that will be invoked when the promise is fulfilled.
  - The fulfilled promise will resolve to a **response object** [\\_\(https://developer.mozilla.org/en-US/docs/Web/API/Response\)\\_](https://developer.mozilla.org/en-US/docs/Web/API/Response).
  - Our function will receive this response object as its argument.
  - Our function calls the method `text()` on the response object.
  - The method `response.text()` returns another promise.
- In line 3 we call the `then()` method on the promise returned by `response.text()`.
  - The fulfilled promise will resolve to a **text** [\\_\(https://developer.mozilla.org/en-US/docs/Web/API/Body/text\)\\_](https://developer.mozilla.org/en-US/docs/Web/API/Body/text) object.
  - Our function will receive this text object as its argument and will call the function `addData()` passing it the text object.
  - `addData()` which is a synchronous function, as before, adds this text to the DOM tree.
- In line 4 we call the `catch()` method.
  - If either of the two promises, i.e., the one returned by `fetch()` or the one returned by `response.text()`, is rejected, the code jumps to catch and the function we have passed to

the `catch()` method will be called.

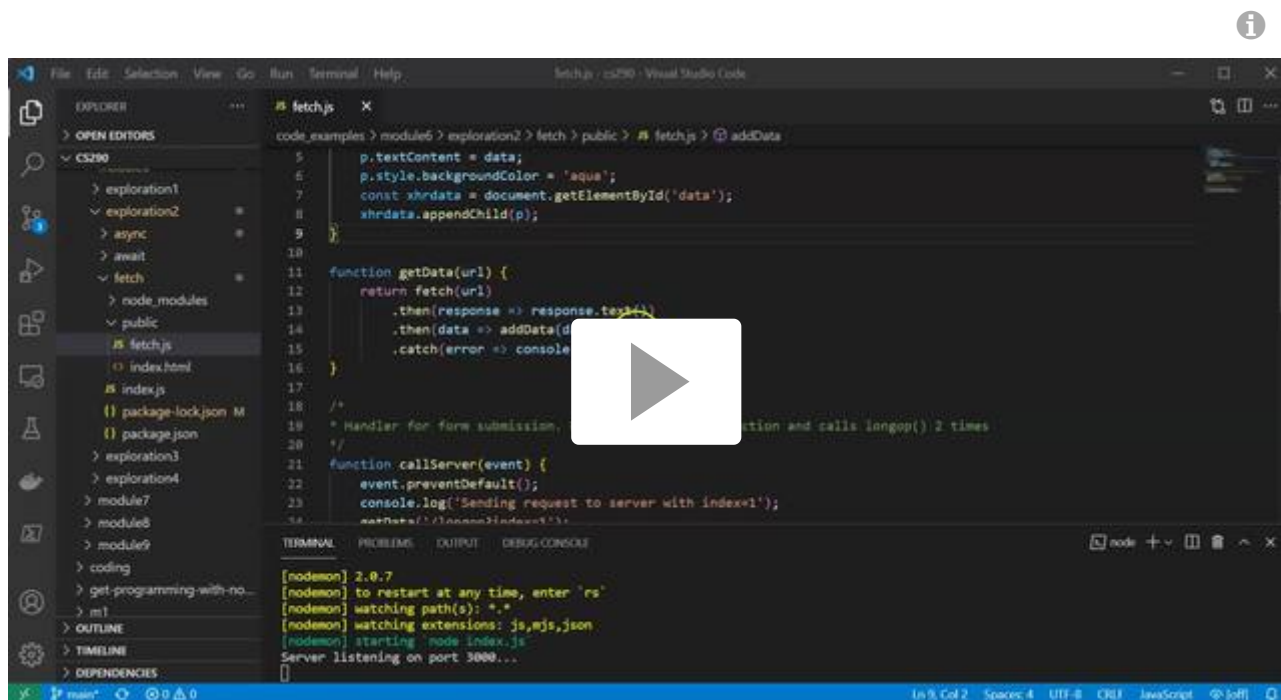
- This function receives the error object as an argument and just logs the error.

## await/async

In the previous section, we saw that when one asynchronous function returns a promise which is then passed to another asynchronous function that in turn returns a promise, we can chain these function calls by using multiple `then` methods. This pattern of coding is sometimes called building a "pipeline" of promises. While such pipelines of promises are often used in JavaScript code, this pattern is a lot less familiar than how we write conventional synchronous code. To make working with promises simpler, JavaScript added two new keywords `await` and `async`. Here is how we use these keywords to work with promises:

### await

- The `await` keyword takes a promise and waits for the promise to be settled.
- If the promise is fulfilled, the resolved value is returned.
- If the promise is rejected, an exception is thrown with the rejection value of the promise.



### Example: await

Let us look at the following example where we code the functionality of calling `fetch()` and then `response.text()` using the `await` keyword.

```
try {  
  const response = await fetch(url); // line 1  
  const data = await response.text(); // line 2  
  addData(data); // line 3  
} catch (error) { // line 4  
  console.error(error) // line 5  
}
```

- The function `fetch()` returns a promise.
- The expression in line 1 uses `await` to wait for the promise to settle. If the promise is fulfilled, the `response` object returned by the fulfilled promise will be assigned to the variable `response`.
- After the promise in line 1 is settled, line 2 will be executed.
- The method `response.text()` also returns a promise.
- The expression in line 2 uses `await` to wait for this promise to settle. If the promise is fulfilled, the text returned by `response.text()` will be assigned to the variable `data`.
- If either of the promises in line 1 or line 2 is rejected, an exception will be thrown by that line.
- Any exception will be caught in line 4 and the error will be logged to the console in line 5.

## async

We are now using a coding style which is the same as we use for synchronous code. But if we are waiting for a JavaScript function to complete, haven't we just gone back to completely synchronous code, which as we saw is a bad idea? The answer to that question is a firm "no." The reason is that we can only use `await` keyword in a function that is declared with the `async` keyword. Doing otherwise will result in a compilation error and JavaScript will reject our code.

### Example: await and async

In the following example, we have written the function `getData()` from our earlier example using `await` and `async`.

```
async function getData(url) {  
  try {  
    const response = await fetch(url);  
    const data = await response.text();  
    addData(data);  
  } catch (error) {  
    console.error(error)  
  }  
}
```

A complete Node application using the above function is available in the following repl.it. Notice that it uses the `document.addEventListener('DOMContentLoaded', () => {})` method, which will be handy for the assignment.

 Runopen in  repl.it

index.js ×



```
1 'use strict';
2
3 const express = require("express");
  const app = express();
4 const PORT = 3000;
5
6 const items = ['MongoDB', 'Express', 'React', 'Node'];
7
  // Decrease the value of count to make the wait shorter, increase it
```

<https://m613--coecs290.repl.co>

Console Shell

Show alert

Send request to server

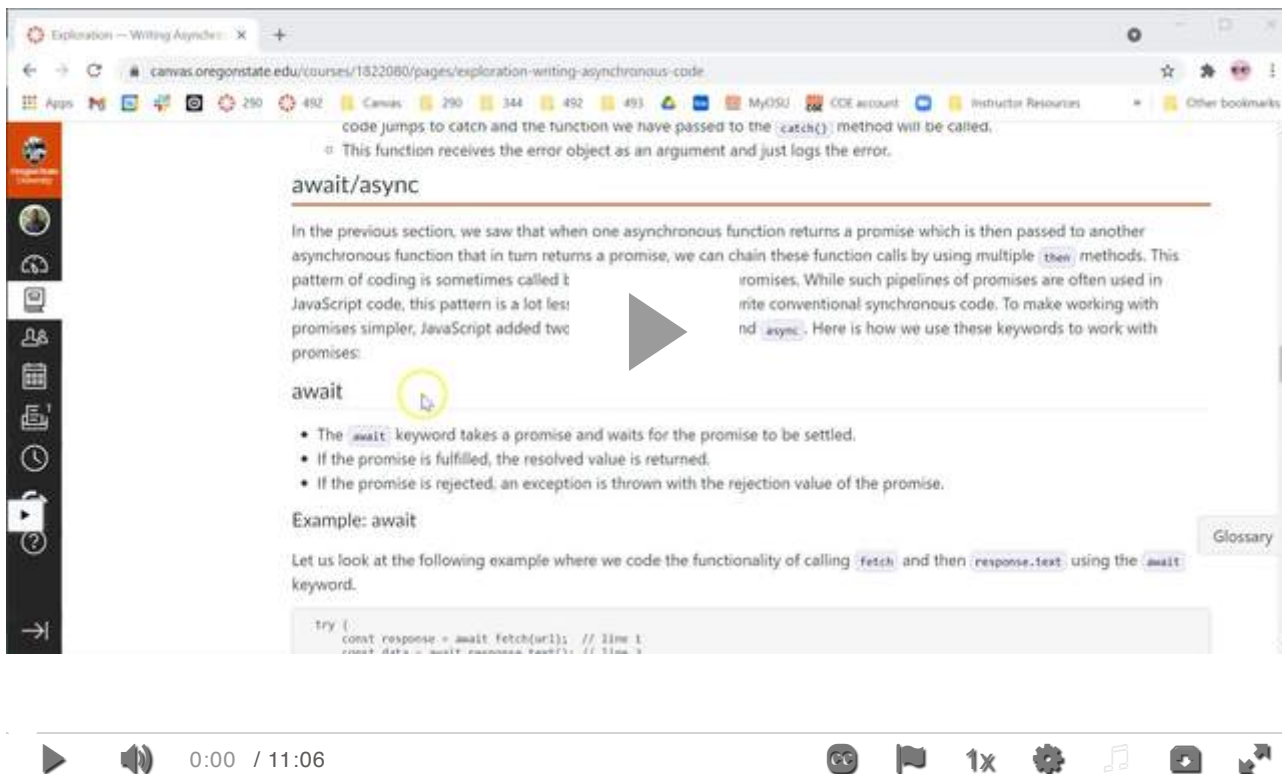


## Return Values from async Functions

An async function is executed asynchronously which means that the call to the async function returns immediately. An async function always returns a promise. If the calling function wants to use the value returned by the async function, then there are 2 options:

- Call the `then()` method on the returned promise to get the result.
- Use `await` keyword when calling the function and wait for the promise to settle.
  - But since the `await` keyword can only be used in an async function, this means that we will need to add the `async` keyword to the calling function as well!





code jumps to catch and the function we have passed to the `catch()` method will be called.

- This function receives the error object as an argument and just logs the error.

### await/async

In the previous section, we saw that when one asynchronous function returns a promise which is then passed to another asynchronous function that in turn returns a promise, we can chain these function calls by using multiple `then` methods. This pattern of coding is sometimes called `then` chaining. While such pipelines of promises are often used in JavaScript code, this pattern is a lot less simpler, JavaScript added two new keywords: `await` and `async`. Here is how we use these keywords to work with promises:

#### await

- The `await` keyword takes a promise and waits for the promise to be settled.
- If the promise is fulfilled, the resolved value is returned.
- If the promise is rejected, an exception is thrown with the rejection value of the promise.

**Example: await**

Let us look at the following example where we code the functionality of calling `fetch` and then `response.text` using the `await` keyword.

```
try {
  const response = await fetch(url); // line 1
  const data = await response.text(); // line 2
}
```

## Example: Return values from async functions

In the following replit, we have modified our running example so that:

- The function `getData()` is no longer responsible for updating the DOM tree with the data returned by our server program.
- Instead `getData()` returns a promise for the data returned by `response.text()`.
- The function `callServer()` now uses the method `then()` on the promise returned by `getData()` and then calls `addData()` to update the DOM tree.



Run

open in repl.it



index.js ×



```

1  'use strict';
2
3  const express = require("express");
   const app = express();
4  const PORT = 3000;
5
6  const items = ['MongoDB', 'Express', 'React', 'Node'];
7
   // Decrease the value of count to make the wait shorter, increase it

```

<https://m614--coecs290.repl.co>

Console Shell

Show alert

Send request to server



Here is the code for `getData()`. Note that the function returns the promise returned by `response.text()`.

```

async function getData(url) {
  try {
    const response = await fetch(url);
    return await response.text();
  } catch (error) {
    console.error(error)
  }
}

```

Here is the code for `callServer()`. Note that it now calls `then()` on the promise returned by `getData()` and then passes the resolved value to `addData()`. Because we use `then()`, we do not need to declare `callServer` as an `async` function. If we were to use `await` inside `callServer()`, then it would be required that we declared `callServer()` as an `async` function.

```

function callServer(event) {
  event.preventDefault();
  console.log('Sending request to server with index=1');
  getData('/longop?index=1')
    .then(data => addData(data))
    .catch(error => console.error(error));
  console.log('Sending request to server with index=2');
  getData('/longop?index=2')
    .then(data => addData(data))

```

```
}  
    .catch(error => console.error(error));  
}
```

## Summary

---

There is a lot more functionality that JavaScript provides related to promises. This includes executing multiple promises, racing multiple promises, running concurrent awaits. We also did not study how to write our own promises. However, the material covered in this exploration provides us with the tools necessary to understand and write modern asynchronous JavaScript code.

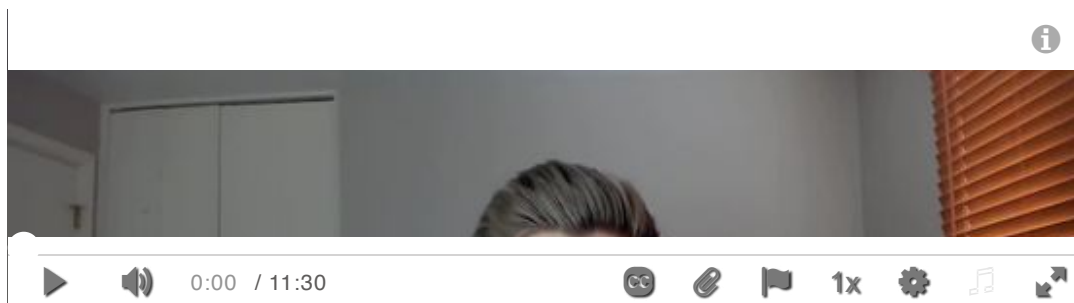
## Additional Resources

---

Here are some references to learn more about the topics we discussed in this exploration.

- MDN has a very good explanation of **promises** [\\_\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise\)\\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).
- MDN's discussion of **async functions** [\\_\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async\\_function\)\\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function).
- If you are interested in how to create promises, here is a good Youtube video

### JavaScript Promises In 10 Minutes (1:20)



- Another Youtube video this time on using `await/async`.

## JavaScript Async Await (7:30)

