Asynchronicity And Networking

Async Await

Created by Zain Afzal @zainafzal08 @blockzain

Overview

Client server Interactions

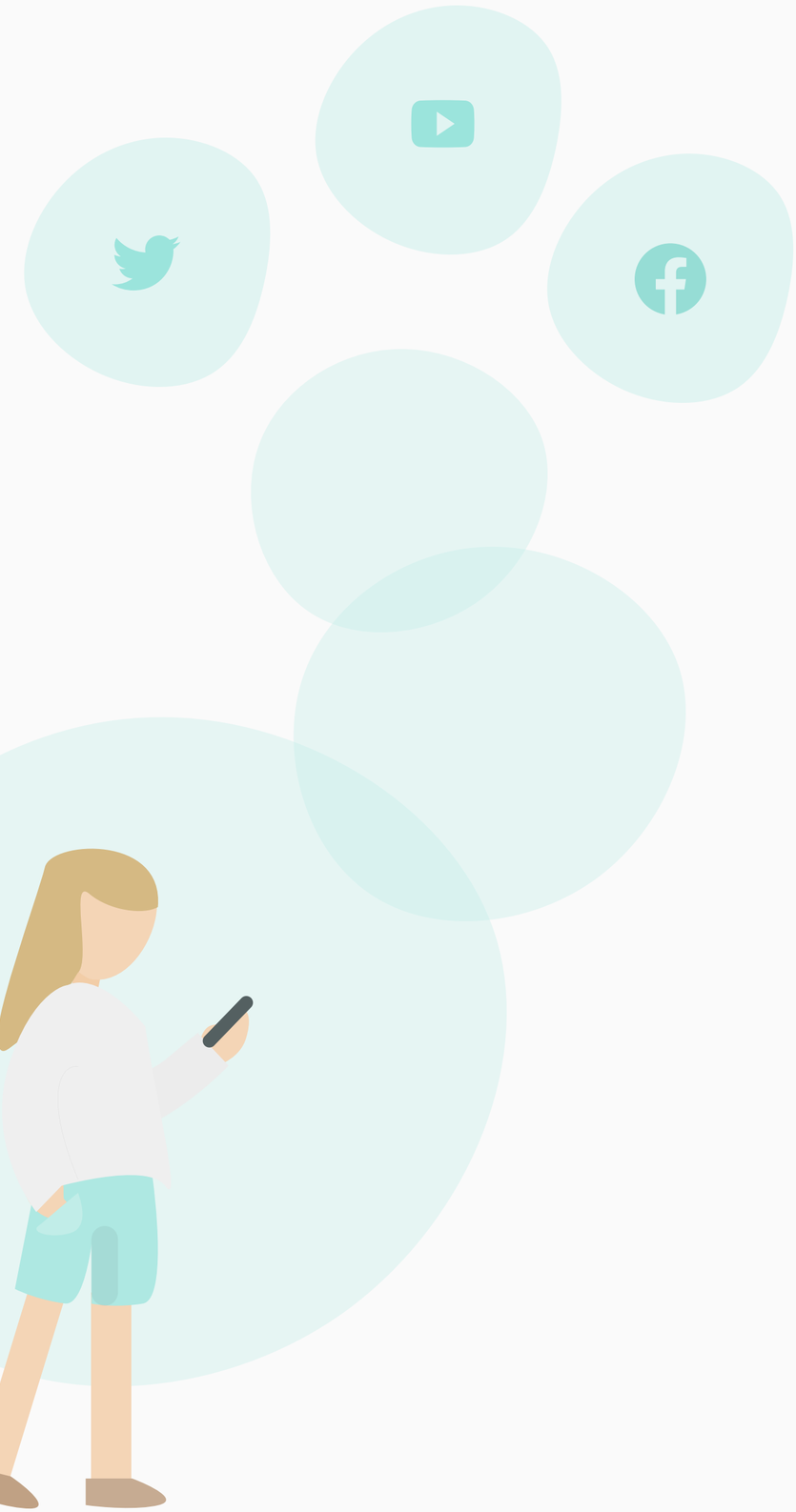
AJAX

Event Loop

Callbacks and XMLHTTP

Promises and Fetch

- Async Await



```
function updateData() {
  loading = true;
  fetch('/data')
    .then(r => r.json())
    .then(data => {
      myData = data;
      loading = false;
    });
}

const myData = null;
updateData();
console.log(loading); // loading still true.
console.log(myData); // myData still null.
```

Recap

Promises are a powerful way to think about async actions and allow us to reason about the order of our code with more confidence. However they still require us to break out of the normal paradigm of thinking about all code in a block running top to bottom.

```
async function updateData() {
  loading = true;
  const r = await fetch('/data');
  myData = await r.json();
  loading = false;
}

```

Asyc Await

Since 2016 many browsers have supported a more pure form of cooperative multitasking by adding the keywords async await. This is actually a way that the language allows us to interact with promises in a more natural way. Promises can be awaited in async functions effectively halting the execution of the function until the promise resolves.

```js
async function myFirstAsyncFunction() {
 // This still returns a promise even
 // though it's a syncronous function
 // it just runs and resolves immediately.
 console.log('hello');
 console.log('world');
}
// This is true.
myFirstAsyncFunction() instanceof Promise;

myFirstAsyncFunction().then(() => {
  console.log('just a promise innit!');
});
```

```js
function sleep(time) {
  return new Promise(resolve => setTimeout(resolve, time));
}

// You can only await in a async function.
async function animate() {
    document.getElementById('output').innerText += 'the...'
    await sleep(1000)
    document.getElementById('output').innerText += 'suspense'
}

// This returns a promise that we ignore.
animate()
// At this point the animation is still running.

```

Asyc Await

Behind the scene any function that is async returns a promise which only resolves when the function completes, any awaits inside of the functions simply stop the execution of the function **yielding** the thread to complete some other task until the thing the function is awaiting has resolved, after which point the function resumes. Note that calling a async function from a non async function (i.e top level scope) does not wait! The function just returns a promise.

```
1 async function foo(v) {
2     const w = await v;
3     return w;
4 }
```

This isn't actually javascript but just some psudocode representing how V8 runs async functions. Read more

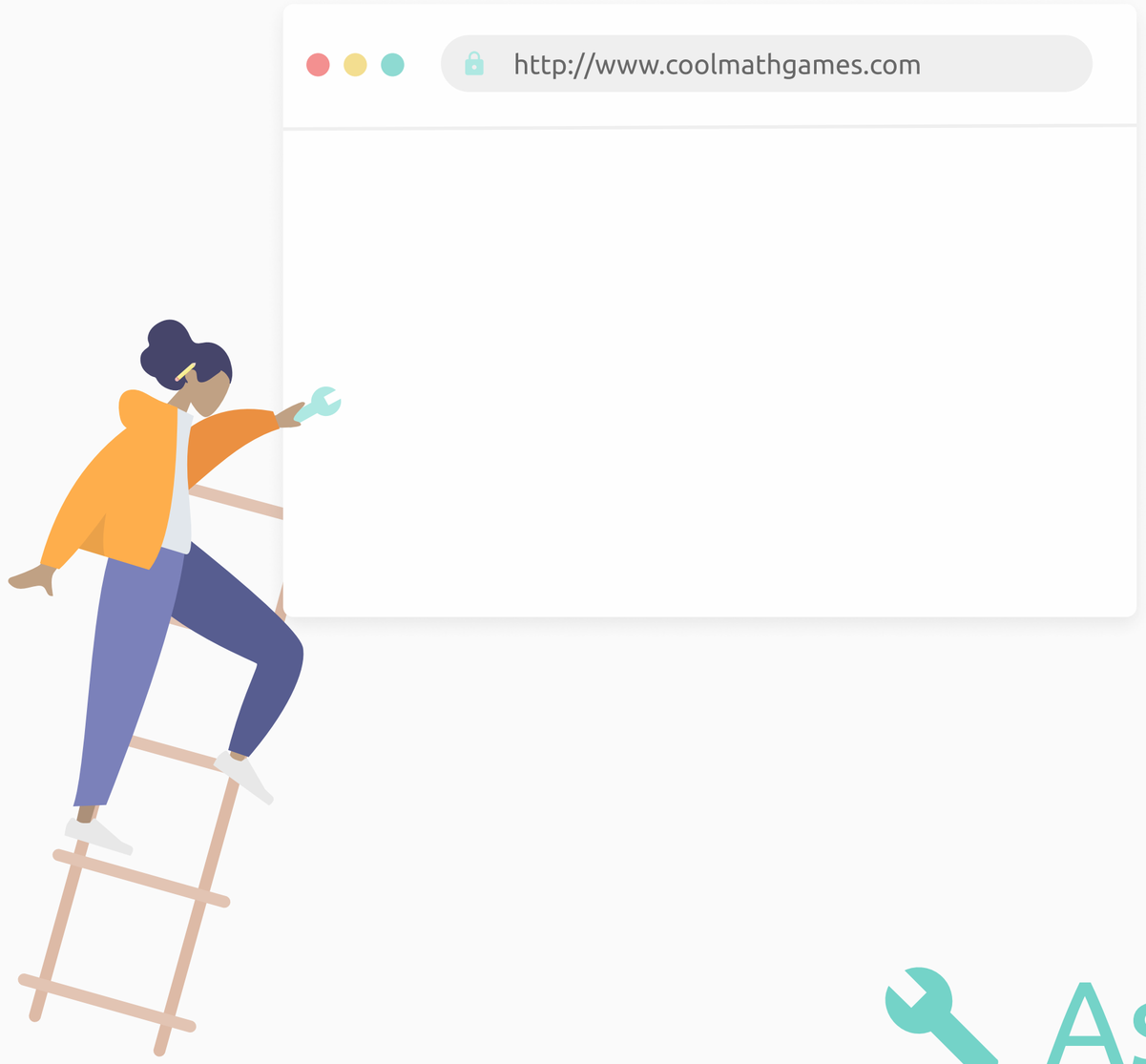
```
1  resumable function foo(v) {
2    implicit_promise = createPromise();
3    // 1. Wrap v in a promise.
4    promise = Promise.resolve(v)
5    // 2. Attach handlers for resuming
6    // foo.
7    promise.then(() => {
8      resume(foo);
9    }, (err) => {
10     throw(err);
11   });
12   // 3. Suspend foo and return
13   // implicit_promise
14   w = suspend(foo, implicit_promise);
15   implicit_promise.resolve(w);
16 }
```

Under the hood

An async function on being called, first creates a implicit promise. When we hit `await v` the runtime wraps `v` in a promise, just in case `v` isn't promise like, then attaches functions to this promise to resume the async function (or throw an error) when the promise settles. Before suspending the function it returns the implicit promise it created earlier. When resume is called the function continues from the point it was suspended at and resolves the implicit promise.

Async Await

Slide 5 of 9



Async Demo

Async Await


```
async function uhOh() {
  throw new Error('oopsies');
}
async function foo() {
  try {

    await uhOh();

  } catch (err) {
    // Prints oopsies.
    console.log(err.msg);
  }
  // Prints oopsies.
  uhOh().catch(err => console.log(err.msg));
}
// Uncaught (in promise) Error.
uhOh();
```

Careful! If you don't await a promise in an async function then it runs on it's own like a promise in a normal function and any error it throws just becomes a unhandled rejection.

Error Handling

Whenever a error is thrown in an async function it gets converted to a promise rejection at the top most async function, however, within an async function you can use the standard try catch synatax in addition to directly using catch.

```
async function foo() {
  const v = await {then: (onFullfill) => onFullfill(42)};
  // Prints 42.
  console.log(v);
}
```

Asyc Await

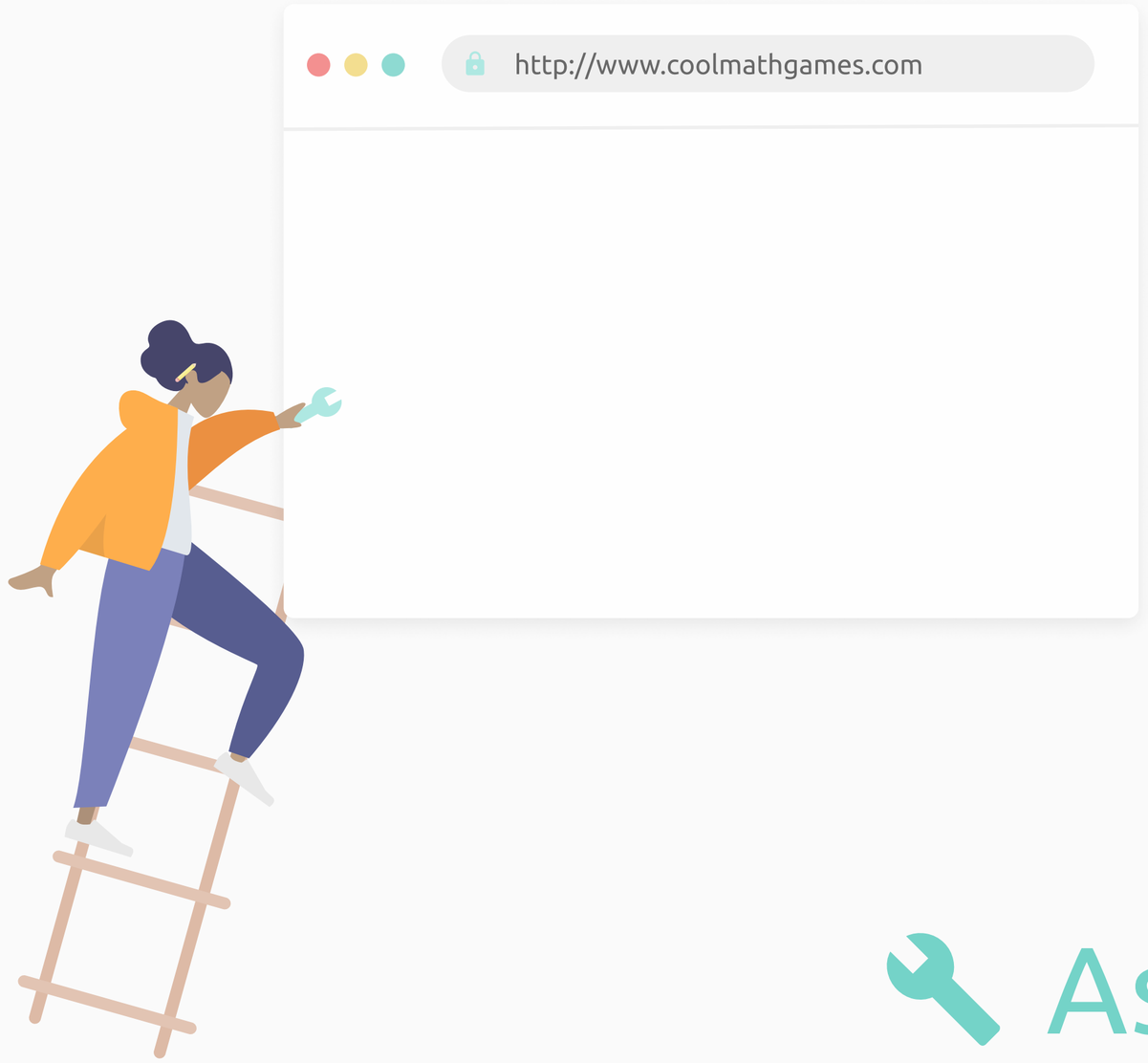
You can just as easily await a thenable object as you can await a promise.

Which one?

If you have a choice between using async await and a promise chain you can do either depending on context. A promise is useful since you can pass it around and attach multiple handlers to it but async await produces easier to read and debug code. Note however that because of way that promises maintain stack traces, **native** async await is a bit more performant then promise.then(). Read more.

Async Await

Slide 9 of 9



Async Demo

main-content/async-error-demo