NodeJS

LABA

PROMISES

CALLBACKS

A callback is simply a function. But they are called by this special name due to the fact that they are passed as an argument to another function.

This is a really common pattern in javascript, and you definitely used a callback before. Look at this example:

```
document.addEventListener("click", (event) => {
  // this is a callback.
  // Javascript calls this inner function whenever a click happens!
})
```

Solvd NodeJS LABA

CALLBACKS

Breaking that example apart, you could write that code in the following way:

```
function handleClickEvent(event) {
  // do something with the event here
}

document.addEventListener("click", handleClickEvent)
```

Notice that we don’t add parenthesis to the function, as we don’t want to call it. Callbacks are very handy for allowing the callee to add custom behavior to something you expose.

CALLBACK HELL

But callbacks can have some pretty big drawbacks. Such as what we call `callback hell`. Take the following example:

```
fetch("https://httpbin.org/headers").then((response) => {
  response.json().then((data) => {
    saveDataToDatabase(data).then(() => {
      collectMetrics({ /* some data */ }).then(() => {
        // ...
      })
    })
  })
})
```

You can clearly see how this can spiral out of control if we don’t take care.

CALLBACK HELL

Believe it or not, that was how promises were handled before ECMA Script 2016. Which was adopted by browsers in 2017, and many websites still have legacy code that looks exactly like the previous example

That release of ECMA script was a pretty big deal, as it introduced the `await` keyword, which changed how we write async javascript since then.

Lets revisit the last example

CALLBACK HELL

```
const response = await fetch("https://httpbin.org/headers");
const data = await response.json();
await saveToDatabase(data);
await collectMetrics({ /* some data */ });
```

Doesn’t this look much nicer? Of course we are hiding the error handling side of things. But this is much easier to read and understand. A big win.

But what exactly are promises?

PROMISES

Promises are objects that represents the eventual fulfillment of an operation. Whether it fails or succeeds, in an async fashion.

You can construct a Promise like this:

```
const promise = new Promise((resolve, reject) => {
  // code to be executed
  const data = { /* some data */ };
  resolve(data);
  // or reject("message");
})
```

Wait a second, aren’t those callbacks?

PROMISES

But wait a second, aren't `resolve` and `reject` two callbacks? YES!

That is how a promise gives back the data or error it produced after finishing its work. Here are their signatures:

```
function resolve(value) { /* ... */ }
function reject(reason) { /* ... */ }
```

PROMISES

A promise also holds two important internal properties:

State

Pending: Initial state of a promise who was neither rejected or resolved

Fulfilled: The promise finished successfully

Rejected: The promised finished with an error.

Result:

Initially this field is `undefined`, but it changes to either the value produced, in case of a resolution. Or to the error, in case of an rejection

Promise.then()

The `then()` method of a promise takes up to two arguments: callback functions for the fulfilled and rejected cases of the promise.

```
then(onFulfilled);
then(onFulfilled, onRejected);
```

Although you'll very rarely see the latter being used, it does exist.

Promise.catch()

The `catch()` method is the counterpart of the `then()` method, and receives a callback to be called if, or when, the Promise rejects.

```
catch(onRejected);
```

This is preferred over giving `then()` both arguments, and is vastly more utilized.

Solvd NodeJS LABA

Promise.finally()

The `finally()` method receives a callback that will get called wether the promise resolved or rejects. That is, it always runs after the result of the Promise.

```
finally(onFinally);
```

This is not so common. But in some cases it is used!

Solvd NodeJS LABA

Promise.all()

`Promise.all()` is a static method of a promise that takes any Iterable of promises, and returns a single promise.

```
Promise.all([promise1, promise2, ...]);
```

The returning promise will resolve if all promises resolve, or reject if any promise rejects.

Promise.any()

`Promise.any()` also takes an Iterable of promises, and returns a Promise that resolves if any of the inner Promises resolves.

```
Promise.any([promise1, promise2, ...]);
```

That is, it will only ever reject if all promises also rejects.

Promise.race()

`Promise.race()` takes an Iterable of promises, and returns a Promise that resolves with the result of the first promise that settles.

```
Promise.any([promise1, promise2, ...]);
```

That is, the fastest Promise to settle will make this Promise either reject or resolve