7. Promises

Github Repository | Vimeo Video Recording

Topics to cover

- [x] 1. Callbacks (quick review)
- [x] 2. The Async Problem
- [x] 3. Callback Hell
- [x] 4. Introduction to Promises

1. Callbacks

It's very important to consider *why* callbacks are so important in JavaScript:

1. They allow us to make our code much more modular. A Higher Order Function can receive any function as a callback, and that way have the possibility of performing multiple actions. It all depends on the function that has been passed as an argument (*a.k.a. the callback*).
2. They are always binded to events in JavaScript! For instance, when we _click_somewhere on a website, this detonates an *event*, and any event can have a callback associated to it, to do any action like change the background color or display an alert.
3. The most important factor to consider, is that **callbacks can be used to execute code after an asychronous task has finished in the "*background*"**.

This helps us managing async code much better.

2. The Async Problem

As we know so far, JavaScript is always dealing with some sort of asynchronous tasks; this can be using a `setTimeout`, reading or writing files using the `Node.js' FileSystem`, performing `HTTP requests` and many more.

Let's consider a recent example of an asynchronous function

```
const net = require('net');
const server = net.createServer();
server.listen(9876, () => {
    console.log(`Server is now listening.`);
});
```

In this example, we are setting up a TCP server. We tell the server to begin listening to port 9876.

The `listen` function is asynchronous, because in order to complete that task, JavaScript needs to make a request to the operating system - it needs to reserve the port, register as a listener on it.

In this sense, every time we execute an asychronous function, we can execute a callback afterwards to do any extra actions.

Async functions *always* accept a callback!

It's important to recognize that one of the things that all async functions have in common is that they always accept a callback. The callback is the mechanism that we use in JavaScript to delay execution of code until after the async behaviour is complete.

Please note that **not all functions that accept callbacks are async, but all async functions accept callbacks**.

Without a callback, there is no other mechanism for delaying the execution of some code until after the async behaviour is complete.

3. Callback Hell

Let's imagine for a moment that you want to request some data from a remote API, then you want to parse the `json` data in the response into an object, before writing that object to a database.

This operation is made up of three async operations: - 1. Fetching from a remote API - 2. Parsing the JSON into an object (for large JSON payloads, this can be long running) - 3. Writing to a database, and confirming that the result was saved.

```
const request = require('request');
const fs = require('fs');
const readline = require("readline");

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

function fetchCats () {
    request('https://cataas.com/api/tags', (error, response, body) => {
        console.log(body);
        const tags = JSON.parse(body);
        fs.writeFile('cattags.json', tags.join(', '), () => {
            console.log('cattags written');
            rl.question("What type of cat is your favourite", (answer) =>
{
                if (tags.includes(answer)) {
                    request(`https://cataas.com/api/cats?tag=${answer}`,
(error, response, body) => {
                        console.log(`https://cataas.com/cat/$
{JSON.parse(body)[0].id}`);

                    });
                }
                rl.close();
            });
        });
    });
}

fetchCats();
```

Take a look at the depth of our callbacks. **It reaches 6 levels of indentation!**. This awkward and hard to read indentation of multiple callbacks is known as a **Callback Hell**.

In order to have code that is much better to read, follow and mantain, we can incorporate **Promises** into our programs!

4. Introduction to Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It offers an **alternative solution to async programming**.

A promise will be be in one of three possible states: - `pending` : the promise has yet to resolve to a value or reject with an error - `fulfilled` : the promise resolved successfully to a value (calling the `resolve` callback) - `rejected` : the promise was rejected with an error (calling the `reject` callback)

Promises vs. Callbacks

Promises help us to avoid the *callback hell* or _waterfall_we saw just before.

For instance, having code with callbacks that looks like this:

```
// nested callbacks
higherOrderFn((dataOne) => {
  callbackTwo((dataTwo) => {
    callbackThree((dataThree) => {
      callbackFour((dataFour) => {
        // do something
      });
    });
  });
});
```

Can be refactored into much easier to read code, that looks like this:

```
// promises
functionOneReturningPromise()
  .then(() => {
    return functionTwoReturningPromise();
  })
  .then(() => {
    return functionThreeReturningPromise();
  })
  .then(() => {
    return functionFourReturningPromise();
  })
  .then(() => {
    // do something
  });
```

Let us consider the first example above for `fetchCats()` and do a refactor, now using promised-based modules, so we don't have to use callbacks:

```
const request = require('request-promise-native');
const fs = require('fs/promises');
const readline = require("readline/promises");

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

function fetchCats () {
  const catTagPromise = request.get('https://cataas.com/api/tags');

  let tags = [];
  catTagPromise
    .then((body) => {
      tags = JSON.parse(body);
      return fs.writeFile('cattags.json', tags.join(', '));
    })
    .then(() => {
      return rl.question("What type of cat is your favourite? ");
    })
    .then((answer) => {
      rl.close();
      if (tags.includes(answer)) {
          return request(`https://cataas.com/api/cats?tag=${answer}`);
      }
    })
    .then((body) => {
      const bodyJSON = JSON.parse(body);
      const bodyLength = bodyJSON.length;
      const randomCat = Math.round(Math.random() * bodyLength);
      console.log(`https://cataas.com/cat/${JSON.parse(body)
[randomCat].id}`);
    });
}

fetchCats();
```

Take a look at the depth, **it now reaches only 2 levels of indentation**.

Error handling with promises

Without Promises, error handling of async methods can be inconsistent, as normal `try/catch` error handling won't work.

As such, each async method must provide it's own interface for error handling, for example: `js request('[https://cataaas.com/api/tags](https://cataaas.com/api/tags)', (error, response, body) => { if (error) { console.log("ERROR", error); } else { const tags = JSON.parse(body); } });`

With Promises, we can use the `.catch` method in combination `.then` for a consistent error handling interface.

```
request('https://cataaas.com/api/tags').then(body => {
        const tags = JSON.parse(body);

}).catch(err => {
    console.log("ERROR", err);
});
```

Useful Links

- https://www.youtube.com/watch?v=DHvZLI7Db8E&t=135s
- https://www.youtube.com/watch?v=QO4NXhWo_NM
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- https://blog.greenroots.info/javascript-promises-explain-like-i-am-five
- https://nodejs.dev/learn/understanding-javascript-promises

View on Compass | Leave Feedback