COURSEWARE

Asynchronous Programming

Contents

Overview

Normally, a given program's code runs straight along, with only one thing happening at once. If a function relies on the result of another function, it has to wait for the other function to finish and return, and until that happens, the entire program is essentially stopped from the perspective of the user.

You might have experienced the *wheel of doom* whilst waiting for a resource, this is the operating system's way of saying "the current program you're using has had to stop and wait for something to finish up, and it's taking so long that I was worried you'd wonder what was going on".

This is a frustrating experience and isn't a good use of a computer's processing power. There's no sense sitting there waiting for something when you could let the other task chug along on another processor core and let you know when it's done.

This is the basis of **asynchronous programming**.

Tutorial

Threads

The experience of waiting for something to finish up that was mentioned in the Overview generally happens due to a thread chugging along.

A **thread** is basically a single process that a program can use to complete tasks. Each thread can only do a single task at once.

```
Task A --> Task B --> Task C
```

Each task will be run sequentially - a task has to complete before the next one can be started.

JavaScript is traditionally single-threaded. Even with multiple cores, you could only get it to run one task on a single thread, called the **main thread**.

So, we need a way to overcome this hurdle.

Nested callbacks

The ancient solution to synchronize calls were via **nested callbacks**. This was a decent approach for a simple asynchronous task, but wouldn't scale particularly well - the code for three simple tasks would look something like this:

```
doSomething(function (result) {
    doSomethingElse(result, function (newResult) {
        doThirdThing(newResult, function (finalResult) {
            console.log(`Got the final result ${finalResult}`);
        }, failureCallBack);
    }, failureCallBack);
}, failureCallBack);
```

This *Pyramid of Doom* results in nothing short of ***Callback Hell***.

Each function gets an argument, which is another function that is called with a parameter that is the response of the previous action.

Too many people will experience brain freeze just by reading the code above. Having an application with hundreds of similar code blocks will cause even more trouble to the person maintaining the code, even if they wrote it themselves.

Promise

Promises were the next logical step in escaping callback hell. This method did not remove the use of callbacks, but it made the chaining of functions straightforward, and simplified the code, making it much easier to read.

A promise is a *placeholder* for some data that will be available: immediately, some time in the future, or possibility not at all.

JavaScript is executed from the top down, each line of code is evaluated and executed in turn.

What happens if the data needed is potentially not available immediately?

Most commonly we may be waiting for some data to come from a remote endpoint.

We need some way to be able to execute code when the data is available, or deal with the fact that it will never be available - *this is the job of a promise*.

A promise is the representation of an operation that will complete at some unknown point in the future. We can associate handlers to the operation's eventual success (or failure).

It exposes `.then()` and `.catch()` methods to handle resolution or rejection.

With promises in place, the code in our asynchronous JavaScript example would look something like this:

```
doSomething()
    .then(result => doSomethingElse(result))
    .then(newResult => doThirdThing(newResult))
    .then(finalResult => {
        console.log(`Got the final result ${finalResult}`);
    })
    // When a promise rejects - or throws an error - it jumps to the first
.catch()
    // call following the error and passes control to its function
    .catch(failureCallBack);
```

To achieve simplicity, all of the functions used in the example would have to be *promisified*. Let's have a look at how we might update the above code to return a `promise`:

```
const doSomethingElse = function(){
    return new Promise((resolve,reject)=>{
        console.log("Initial");
        resolve();
    }).then(()=>{
        // throw new Error('Something failed') - uncomment me and run & observe
output
        console.log("Do this");
    }).catch(()=>{
        console.log("Do that");
        reject();
    }).then(()=>{
        console.log("Do this after whatever happened before");
    });
}
```

The promise itself performs actions from the method. Now, `resolve` and `reject` callbacks will be mapped to `Promise.then()` and `Promise.catch()` respectively.

Async

The Pyramid of Doom was significantly mitigated with the introduction of Promises. However, we still had to rely on callbacks that are passed on to `.then()` and `.catch()` methods of a `Promise`.

Promises paved the way to one of the coolest improvements in JavaScript. The latest versions of JS brought in syntactic sugar on top of Promises in JS in the form of `async` and `await` statements.

They allow us to write `Promise`-based code as if it were synchronous, but without blocking the main thread, as this code sample demonstrates:

```
async function test(){
    return Promise.resolve("Hello there");
}
test().then(console.log);
```

The word `async` before a function means one simple thing: a function always returns a promise. So, `async` ensures that the function returns and wraps non-promises in it.

Await

The keyword `await` pauses the execution of the `async` function until completion of the promise, and then resumes.

(*note: `await` only works inside `async` functions.*)

```
async function test(){
    let promise = new Promise((resolve, reject)=>{
        setTimeout(()=>resolve('Done'),-1)
    });
    let result = await promise; // wait until the promise resolves
    console.log(result);
}
test();
```

Fetch

`fetch()` requests provide the functionality previously provided by XMLHttpRequests. It greatly simplifies making requests and dealing with responses. `fetch()` requests return promises.

Making a `fetch()` request can be as simple as passing a URL and chaining appropriate `.then()` and `.catch()` methods onto the return.

```
fetch('https://www.qa.com/courses.json')
.then(response => response.json())
.then(myJson => console.log(myJson))
.catch(err => console.error(err))
```

(*note: we don't have to use `JSON.parse`, as response objects have a `.json()` method which returns a *promise* that resolves with the result of parsing the body text of the response by JSON. By default, a `fetch()` request is of type `GET`.*)

A `fetch()` promise does not **reject** on receiving an error code from the server (such as 404).

Instead, it **resolves** and will have a property (`response.ok == false`).

To correctly handle `fetch()` requests, we would need to also check whether the server responded with a `response.ok === true`:

```
fetch(url)
.then(response =>{
    if(response.ok){
        //do things
    } else{
        //handle error
    }
});
```

Exercises

1. Write the following code:
    1. Create three async functions called `asyncFunction<1,2,3>`
    2. Ensure each returns a new promise
    3. Declare a `setTimeout()` method in each
    4. In the body of the timed interval log the name of the function
    5. Set the corresponding time to the functions - AsyncFunction1 = 3 second, AsyncFunction2 = 2 seconds, AsyncFunction3 = 1 seconds
    6. Create a function called `doThings()` and run each AsyncFunction in order and finally print to the screen `All Done!`

    ▶ Solution
2. Now adjust the `doThings()` function to use the `await` keyword so that you get the following output:

```
Async Function 1
Async Function 2
Async Function 3
All done!
```

▶ Solution