Promises

Learn how ES2015 solves the problem of callback hell and the Pyramid of Doom. Promises are now used everywhere in asynchronous code and are part of modern and future JavaScript.

Preface: A Note

Throughout this lesson and future lessons regarding asynchronous code, several code blocks will be runnable. The code in the blocks is not runnable by itself; if you try to paste it in another JavaScript evaluator, it will not work.

There is hidden code prepended in each block. These blocks are meant only to demonstrate what an actual application using asynchronous code might look like.

Introduction

We come now to a solution to the pyramid of doom from earlier. Promises are still entirely callback-based, but they give us a nice syntax wrapper around the callbacks to make it easier to read asynchronous code.

Standard Callbacks

Starting with callbacks, let's review a common pattern of asynchronous code execution. This is the same code found in the earlier lesson. Here's the situation we're in.

A user launches a request from thier browser. When the server receives the request, the server needs to read data from a file, write it to another file, and then send a response notifying the user when it's all done.

Remember that `onRequest` is called with this as the `request` parameter.

```
request = {
    fileToRead: 'readFrom.txt',
    fileToWrite: 'writeTo.txt'
};
```

```
});
```

```
onRequest(function(request, response) {
    readFile(request.fileToRead, function(data) {
        writeFile(request.fileToWrite, data, function(status) {
            response.send(status);
        });
    });
});
```

Note that when making an asynchronous function call such as `readFile` or `response.send`, we almost always want to wrap the call in a `try`/`catch` block to catch any errors. For simplicity, we'll leave it out for now and show Promise error handling in the next lesson.

First let's transform this using ES2015 arrow functions.

```
onRequest((request, response) =>
    readFile(request.fileToRead, data =>
        writeFile(request.fileToWrite, data, status =>
            response.send(status)
        )
    )
);
```

Now let's get to promises.

Promises

Modern functions give us a new syntax to use to make this even more readable.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(data => writeFile(request.fileToWrite, data))
    .then(status => response.send(status))
);
```

The pattern here is: call some asynchronous function. Chain a `.then` call after it and the callback passed in to `.then` will run after the async operation is completed. It'll receive the return value of the async operation as its parameter.

In this case, `request.fileToRead` is read. Upon completion of the read operation, our callback on line 3 fires. It's an asynchronous function that returns the status of the operation once done. When *that's* done, the callback on line 4 will fire, with the status passed in.

As we can see, this eliminates the pyramid of doom. It's now a plateau of joy™. It's easier to read and no matter how many asynchronous operations we need to perform, we can keep adding `.then` functions. Our eyes can easily go down the page and follow what's happening.

This will only work with asynchronous functions that are promise-enabled. Old code that doesn't support promises will break if we attempt to use `.then`s.

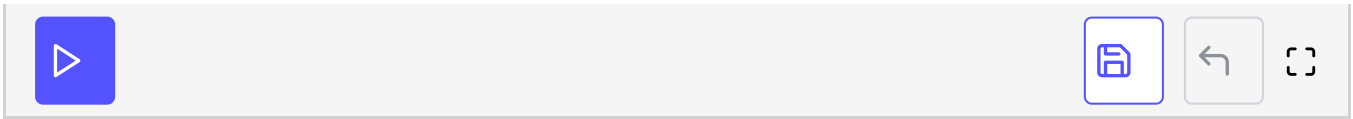
We can, however, convert callback-based asynchronous code to Promises using the `Promise` constructor.

Promisifying

What would we do if the `readFile` function above wasn't Promise-enabled? We could transform it into a Promise ourselves.

```
onRequest((request, response) => {
    const readFilePromise = new Promise(resolve => {
        readFile(request.fileToRead, data => resolve(data));
    });

    readFilePromise
    .then(data => writeFile(request.fileToWrite, data))
    .then(status => response.send(status));
});
```



We can promisify a non-Promise using the `Promise` constructor. `Promise` takes in a callback, which itself takes in a callback parameter `resolve`.

We want to set up the callback we provide to Promise such that it calls its `resolve` parameter when it has the data it needs. So, after the `readFile` operation is complete, we want to call `resolve` with the data we get out of it.

The first `.then` chained to the Promise will receive what was passed in to the `resolve` call. So, `data` in line 7 will be the same `data` in line 3.

Anything returned by a `.then` is automatically promisified. This lets us chain `.then` methods indefinitely.

We'd have to promisify `writeFile` as well if it wasn't Promise-enabled.

That's it for the basics of Promises.

If they seem difficult to understand, you're not alone in thinking so. Both asynchrous code and callbacks are difficult to fully understand, especially for many who never received a thorough education in the nuances of JavaScript.

In the next lesson, we'll learn how to handle errors using Promises.