Lecture 3: Asynchronous Code and Promises

CS-546 – WEB PROGRAMMING

What is Asynchronous Code?

How is JavaScript run?

JavaScript runs off of what is known as an *event queue*.

Every statement gets added into a queue of instructions to run, that are processed in order.

However, some operations (such as making network requests, file calls, etc.) are very expensive and would normally use up huge amounts of time and resources in order to complete. This would normally result in blocking execution.

Rather than allowing these to block all execution, we can use asynchronous code in order to continue execution and run the code that relies on the results of expensive operations in callbacks that occur at a later point in time.

What is synchronous code

Synchronous code is code that runs in the order it is written. This is what we are used to as programmers: we write what happens and it gets done in the order expected.

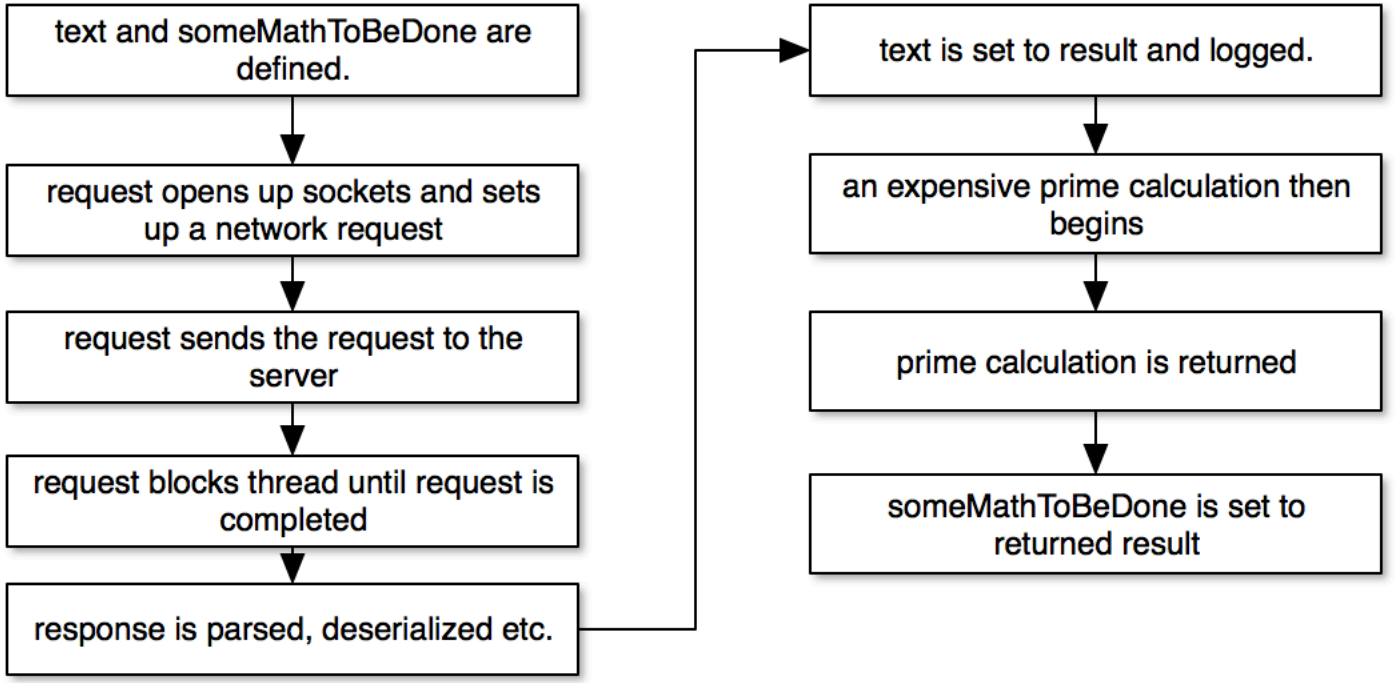
Synchronous Code

```
let text = requestUrlSynchronously("http://someurl.com/peterpan.txt");

console.log(text);

let someMathToBeDone = findPrime(15);
```

Added to event queue and runs in order of arrows



What is asynchronous code

Asynchronous code is code that is not run in the order that which it is written.

Traditionally, asynchronous functions will take callback functions in order to run code that relies on the result of the function.

Nowadays, this pattern is abstracted away and many asynchronous functions return promises.

A callback function?

In JavaScript, you can pass functions as the arguments for other functions.

This is very useful for asynchronous code, as you can run pass a callback function that will execute when a long running process completes. For example:

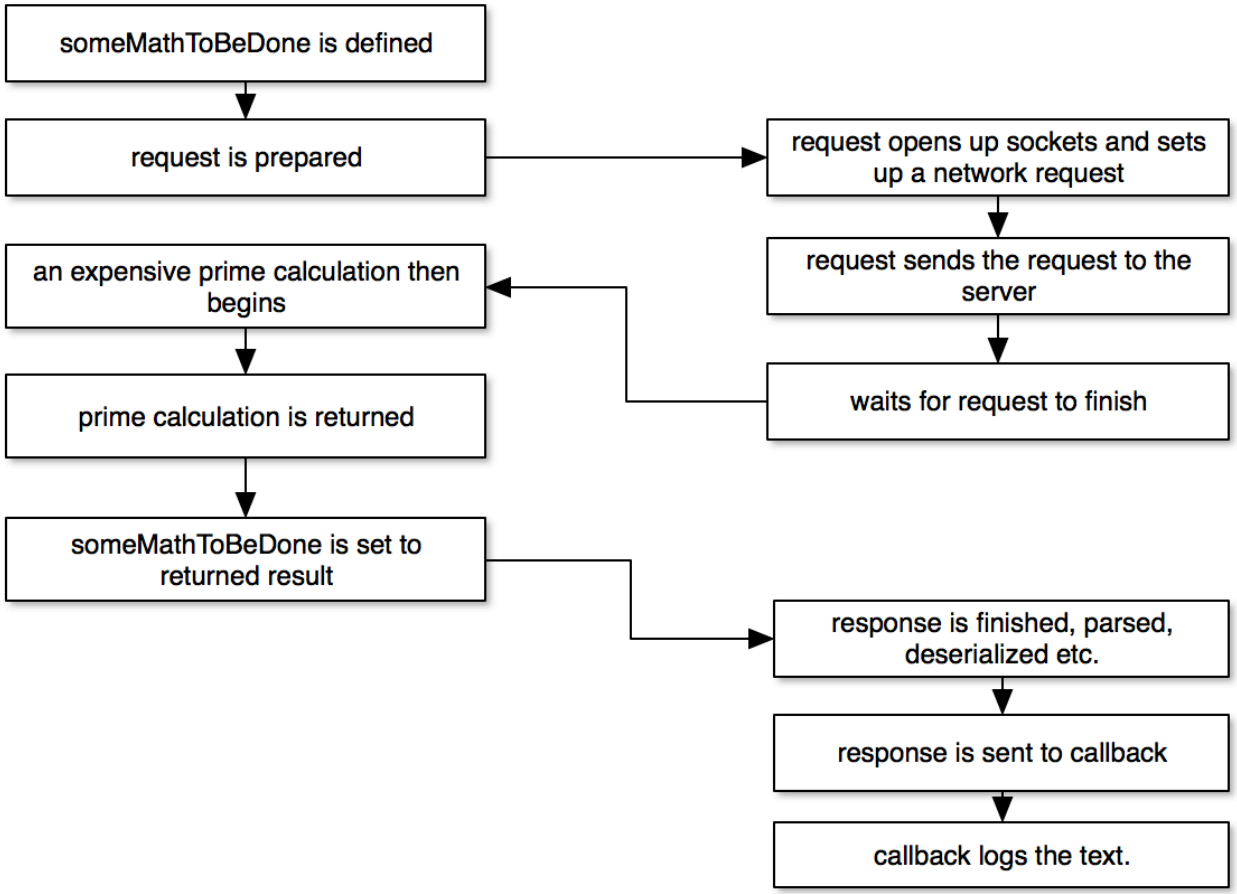
- You can make a database call that runs asynchronously, and when the result comes back pass the result to the callback
- You can make a network request that runs asynchronously, and when the request is completed the response is passed to the callback

Asynchronous Code

```
requestUrlAsync("http://someurl.com/peterpan.txt", function(text) {
    console.log(text);
});

let someMathToBeDone = findPrime(15);
```

Added to event queue and (potentially) runs in order of arrows; actually order may differ



Where will we use asynchronous code?

When working with files

There are many file system functions that are exposed through node's native *fs* module. Most of these are asynchronous, as I/O operations on a computer are notoriously slow.

You can do many things such as

- Read and write files
- Get directory listings
- Create and delete directories
- **Watch for file changes to occur!**
- And more…

You can read about the fs module at

- https://nodejs.org/api/fs.html

Running a Web Server

When you run your own web server, you have no idea when someone will actually access your routes (if they will at all!).

For this reason, all your server code will be asynchronous. Your server code will have to wait for a network request to initiate before sending data back down to the client.

Making an HTTP Request

Making an HTTP request is an asynchronous operation, since it can take a **very** long time for the request to complete.

There are many more parts than you would expect to making an HTTP request, and the responding server (if it exists) could take any amount of time to complete the request.

For this reason, HTTP requests are asynchronous so that the file can be downloaded and such while other operations are completing, rather than blocking and holding up your entire application when a server is responding slowly.

Database Operations

When making a call to a database, you generally use asynchronous methods.

Connecting to a database, finding entries, creating new entries, updating old entries, and deleting entries are **all** asynchronous operations.

There is a great deal of network traffic / inter-process communication that has to occur to perform any database operation, and then a result must be awaited.

For this reason, database operations are asynchronous: there is no need to block the entire node process to wait for the relatively long task of a query. This also prevents node from blocking on a long-running database query.

Promises

What is a promise?

A promise is an object that represents the eventual result of some asynchronous operation

Rather than blocking, an asynchronous method will go through the following steps:

- Immediately return a new Promise object, which takes a callback to run on success and failure
- Prepare the asynchronous call
- Run the asynchronous component of the method; due to how JavaScript's event queue works, this will always begin to run after the current method ends.
- Return a promise that will resolve the request after the asynchronous operation is completed

Why are they useful?

Promises allow us to write code that resembles synchronous code in how it is syntactically written, while actually writing powerful and complex asynchronous code!

- https://github.com/Stevens-CS546/CS-546-WS-Summer-1/blob/master/Lecture%20Code/lecture_3/promises/reading_files.js is much easier to follow compared to callback-based code (https://github.com/Stevens-CS546/CS-546-WS-Summer-1/blob/master/Lecture%20Code/lecture_3/callbacks/reading_files.js)

It is a cleaner way of performing asynchronous operations, rather than infinitely nesting callbacks, which causes for easier development.

Using promises

Promises have a property on them called *then*.

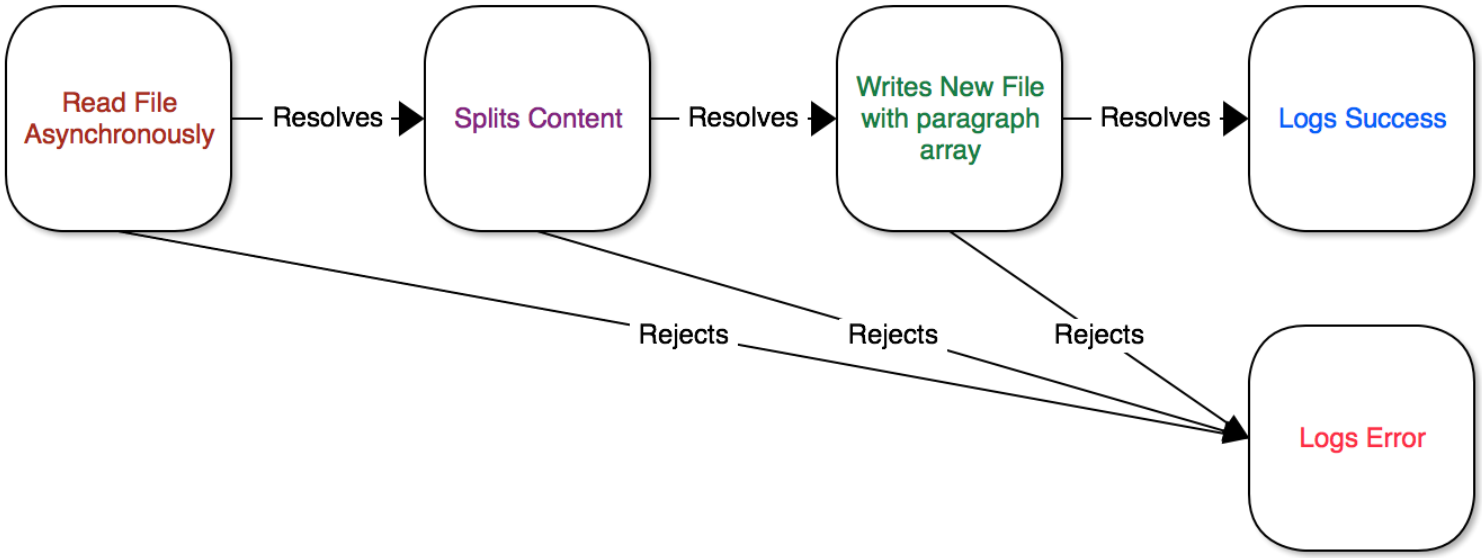
then is a function that takes 1 to 2 parameters: one necessary callback to handle a successful case that will receive the resulting data, and one optional callback for handling errors.

When *then* is run, it returns **another promise**!

The new promise will return the result from the callback run from the first promise! This allows you to keep chaining asynchronous operations

```
let readResult = readFileAndReturnPromiseWithContent("data.txt");

readResult.then((fileContent) => {
 // return string split into an array of paragraphs
    return fileContent.split("\n");
}).then((paragraphArray) => {
    let newFileText = JSON.stringify(paragraphArray);
    return writeFileAndReturnPromise(newFileText);
}).then(() => {
    console.log("Everything has worked!");
}, (error) => {
    console.log("An error has occurred!");
    console.error(error);
})
```



Catching Errors

While promises can chain, they can also have errors!

When a promise has an error, all promises that chain off that promise will reject with the same error until it is caught and handled.

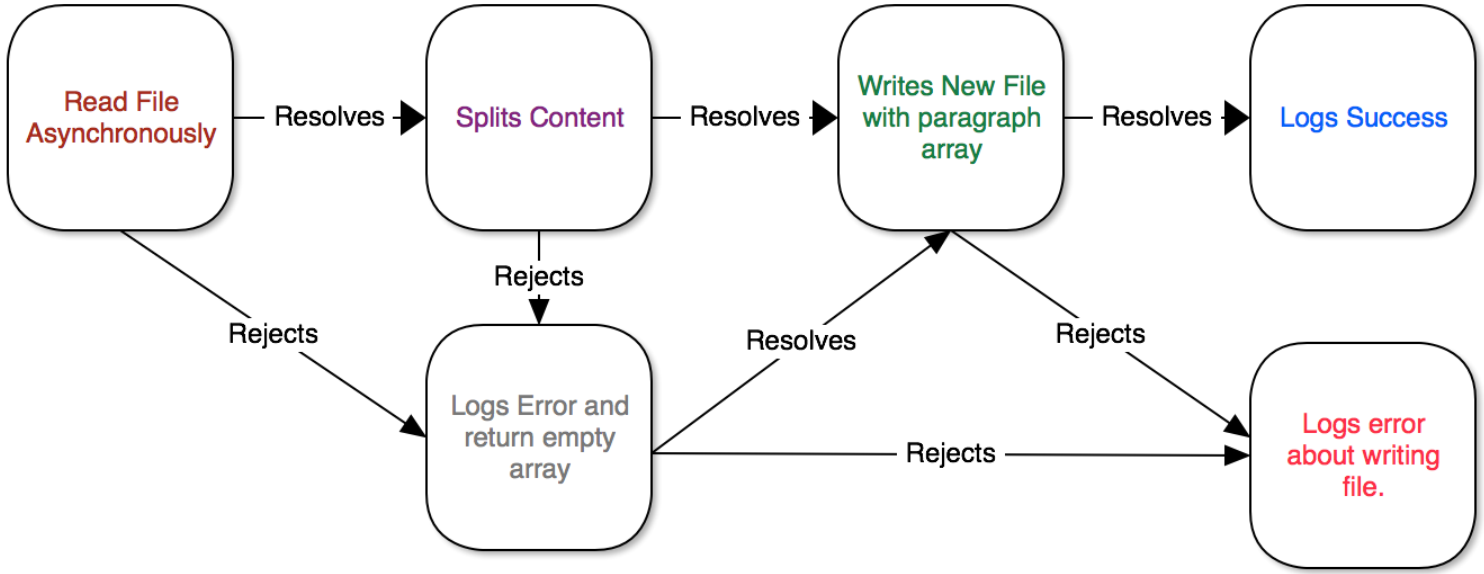
The second callback of *then* allows you to catch any errors that have occurred to that point. From there, you may:

- Log and rethrow the error to keep the chain from being fixed.
- Recover, and return a value (or a promise that will resolve to a value!) that will be used in order to continue the chain successfully from that point

Alternatively, you can use the .catch method on a promise, which is essentially .then(null, (errorHandler) => {})

```
let readResult = readFileAndReturnPromiseWithContent("data.txt");

readResult.then((fileContent) => {
 // return string split into an array of paragraphs
    return fileContent.split("\n");
}).then((paragraphArray) => {
    let newFileText = JSON.stringify(paragraphArray);
    return writeFileAndReturnPromise(newFileText);
}, (readFileError) => {
    console.error("Error splitting text or reading file; returning an empty array");
    return [];
}).then(() => {
    console.log("Everything has worked!");
}, (error) => {
    console.log("An error has occurred writing the file!");
    console.error(error);
})
```



Questions?