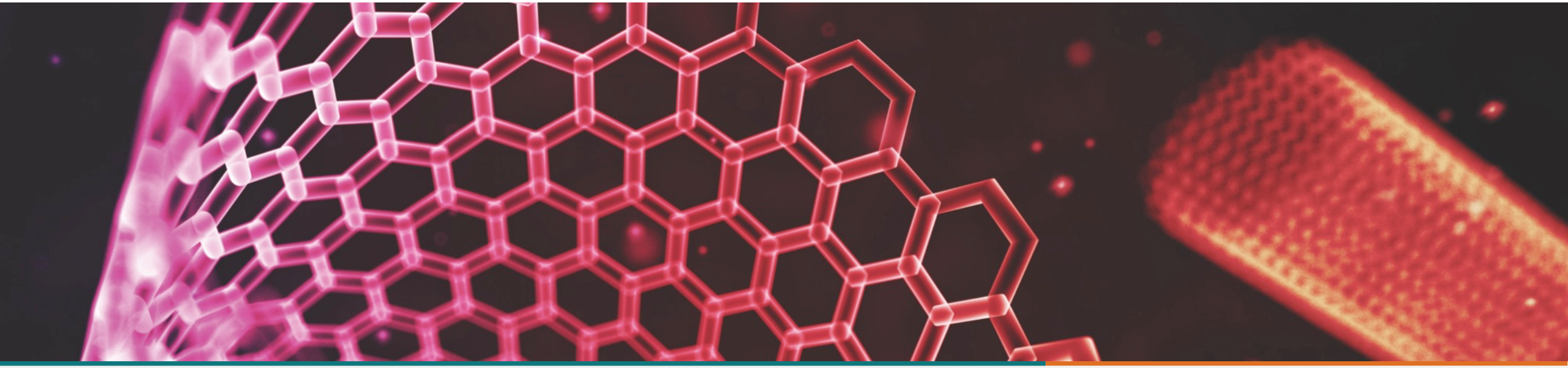
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 546 – Web Programming I
Asynchronous Programming



STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

stevens.edu

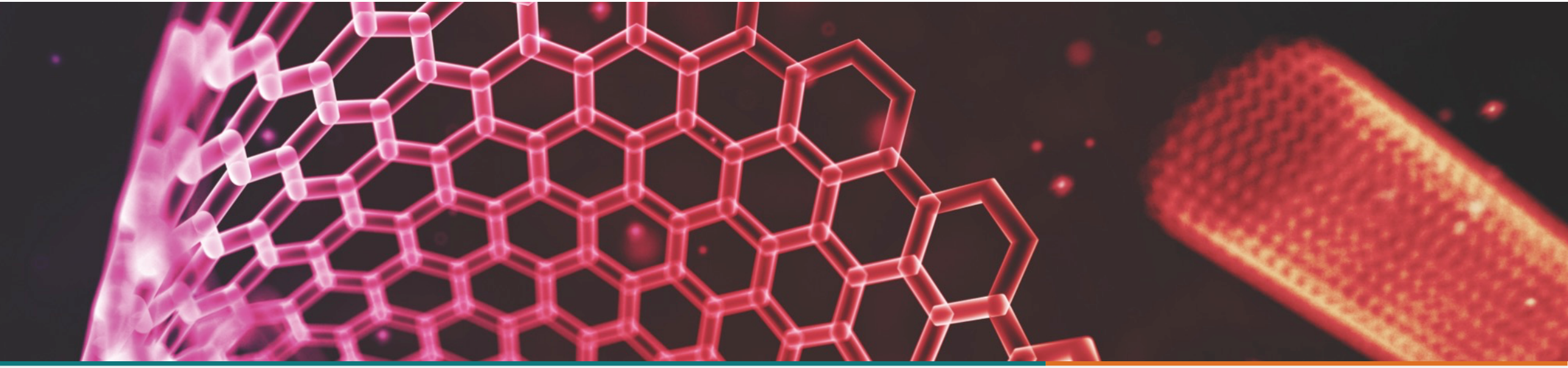
Patrick Hill

Adjunct Professor

Computer Science Department

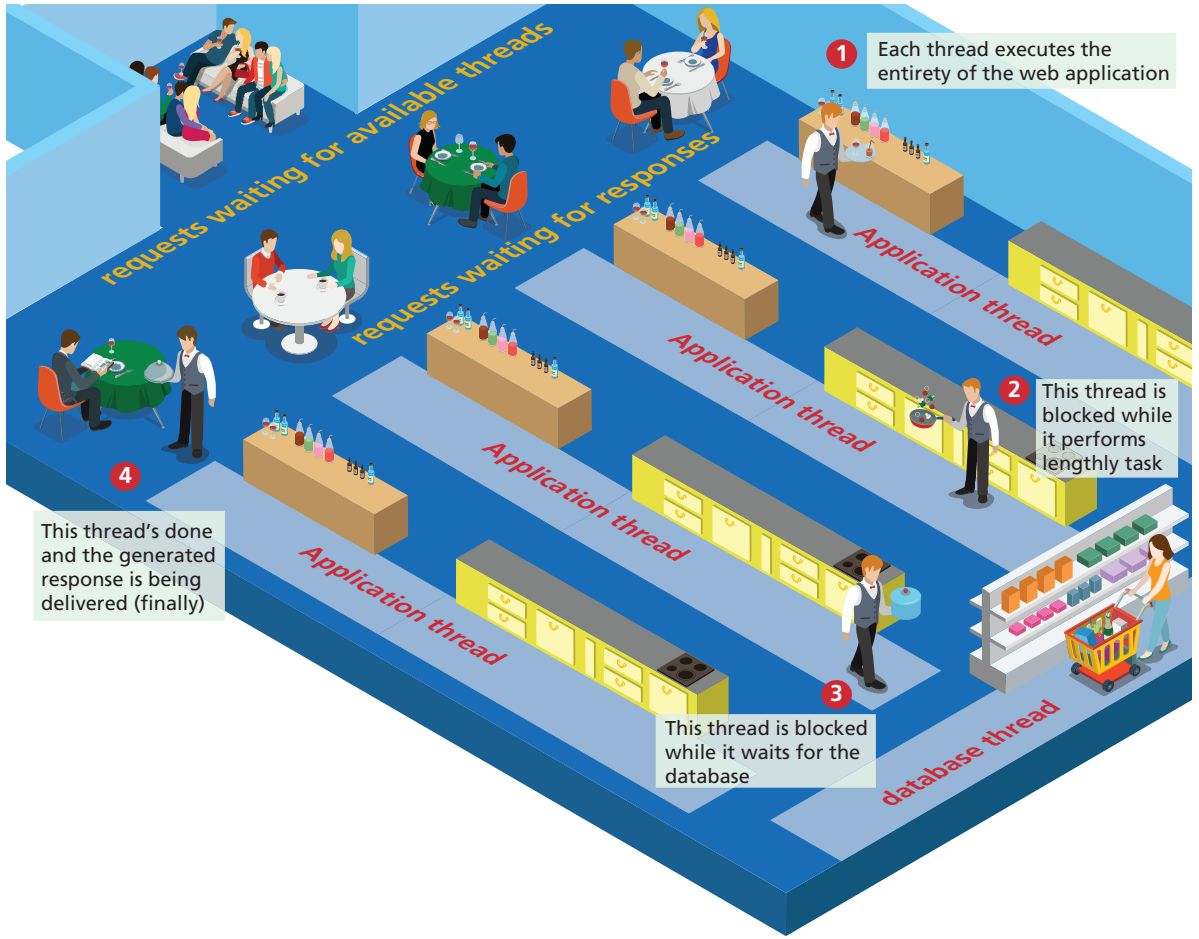
Patrick.Hill@stevens.edu

What is Asynchronous Code?



Thread Blocking

Blocking thread-based architecture (how apache/PHP run)



What How JavaScript Is Run

JavaScript runs off of what is known as an event loop.

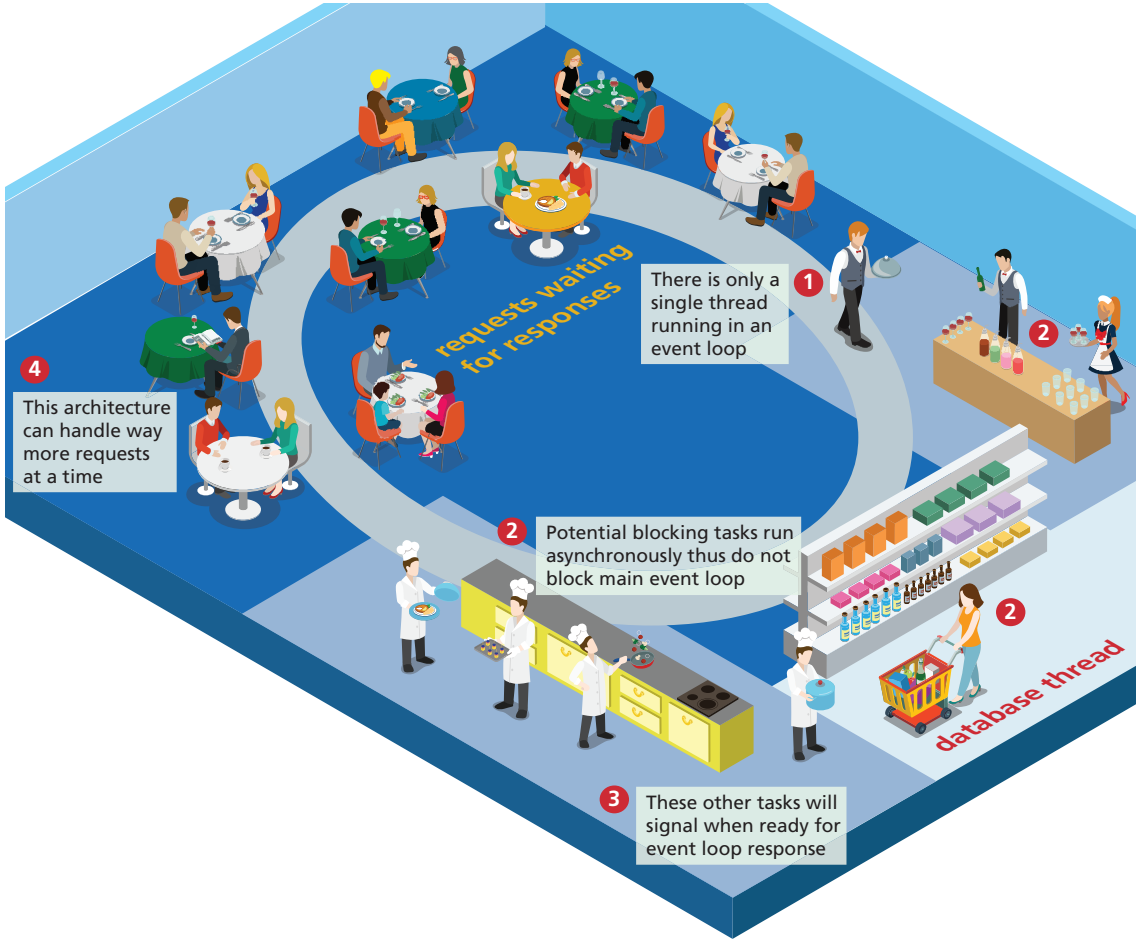
Every statement gets added into a queue of instructions to run, that are processed in order.

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 How JavaScript Is Run

JavaScript event loop:



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.

The key to remembering the difference between asynchronous and synchronous programming is to view all programming as a chain of operations.

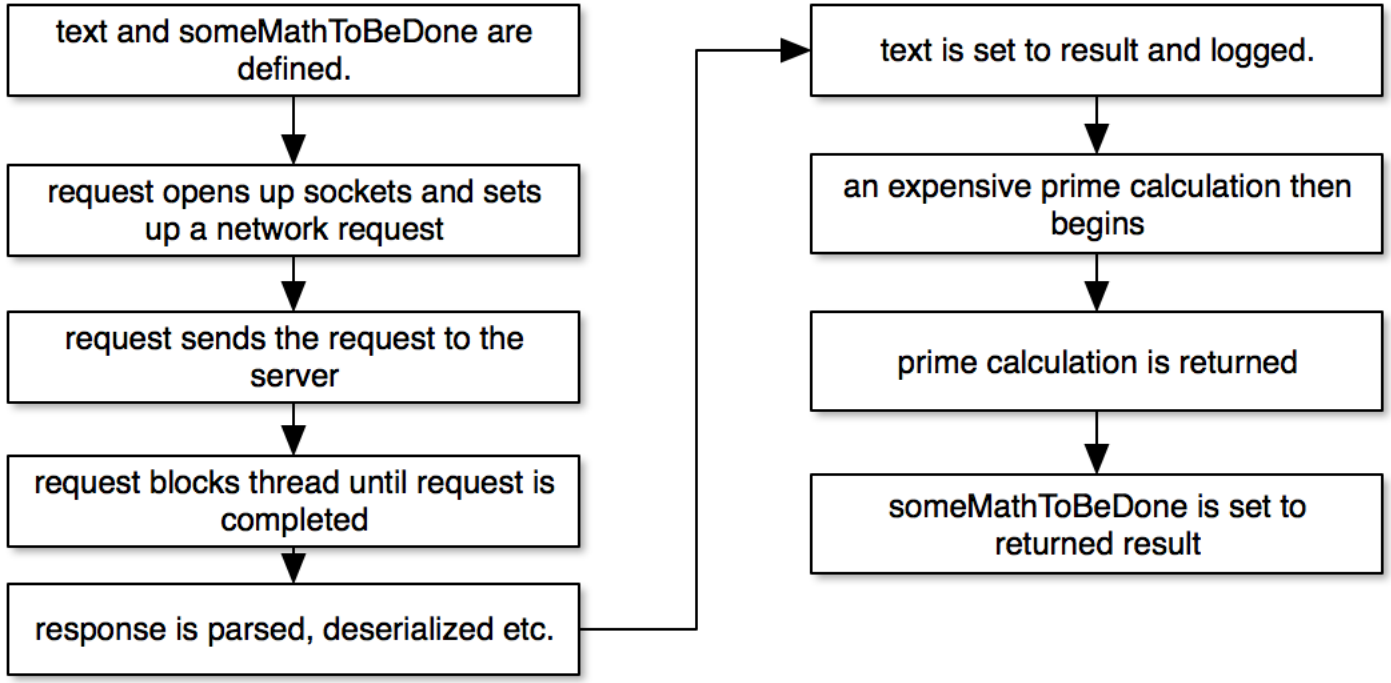
In **synchronous** code, these operations are run **sequentially**. Operations that are written first, will be run first. This will hold true even for functions that run callbacks despite being synchronous, such as **[].map**

In **asynchronous** code, some operations are run **non-sequentially**. Operations that are written first may run before other operations, but operations that passed as parameters to asynchronous functions can get run at any time in the future. No order is guaranteed.

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.

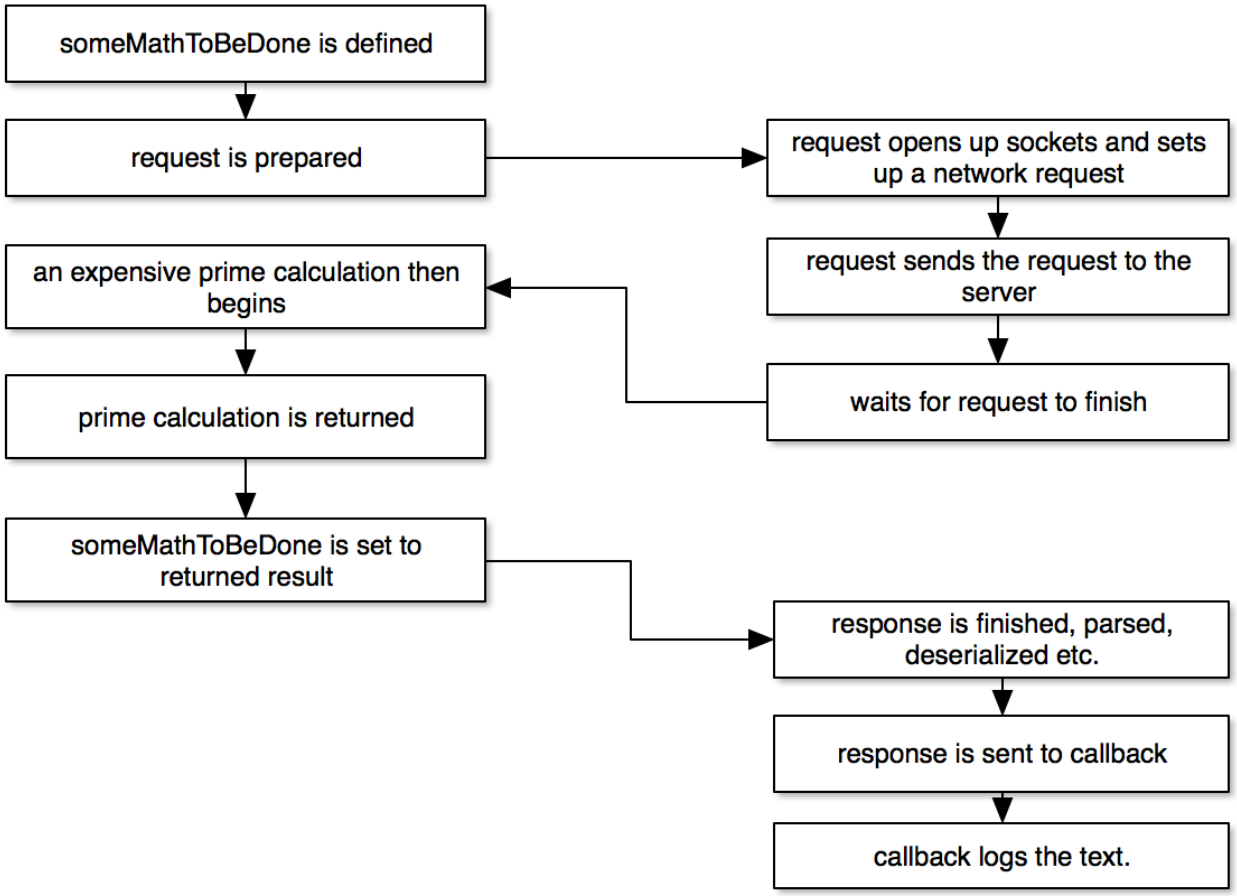
Nowadays, this pattern is abstracted away and many asynchronous functions return promises.

Note: if you have multiple asynchronous functions, there is no guarantee that the first function that is started will finish before the following asynchronous actions. You must therefore often start the second asynchronous actions after the first is completed.

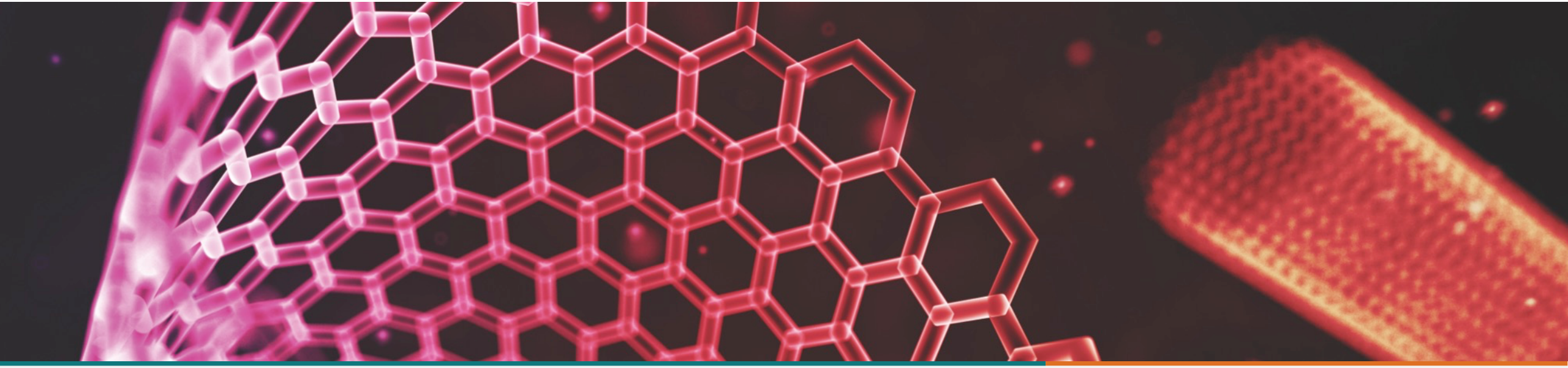
Asynchronous Code

```
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

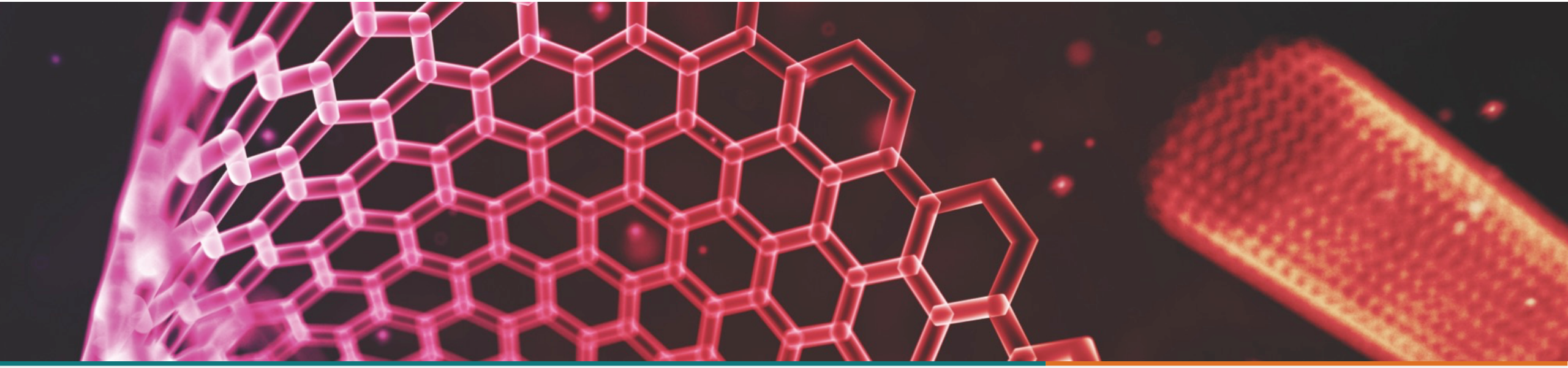
When making a call to a database, you generally use asynchronous methods.

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.

Callback Functions



What Is a Callback Function?

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

This is very useful for asynchronous code, as you can pass a callback function to a long-running function. That way, the result will be passed to the callback once the long-running process is complete. Since the long running function will yield its time to run many times over (since it is asynchronous), we are not able to simply place code after our asynchronous function and assume that it will run in the order we write it.

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

What Is a Callback Function?

```
function doHomework(subject, callback) {
  console.log(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', () => {
  console.log('Finished my homework');
});
```

What Is a Callback Function?

```
function doHomework(subject, callback) {
  console.log(`Starting my ${subject} homework.`);
  callback();
}
function alertFinished() {
  console.log('Finished my homework');
}
doHomework('math', alertFinished);
```

Things to Note

Callbacks are the most basic and simplest form of managing asynchronous operations in JavaScript, but come with a number of difficulties.

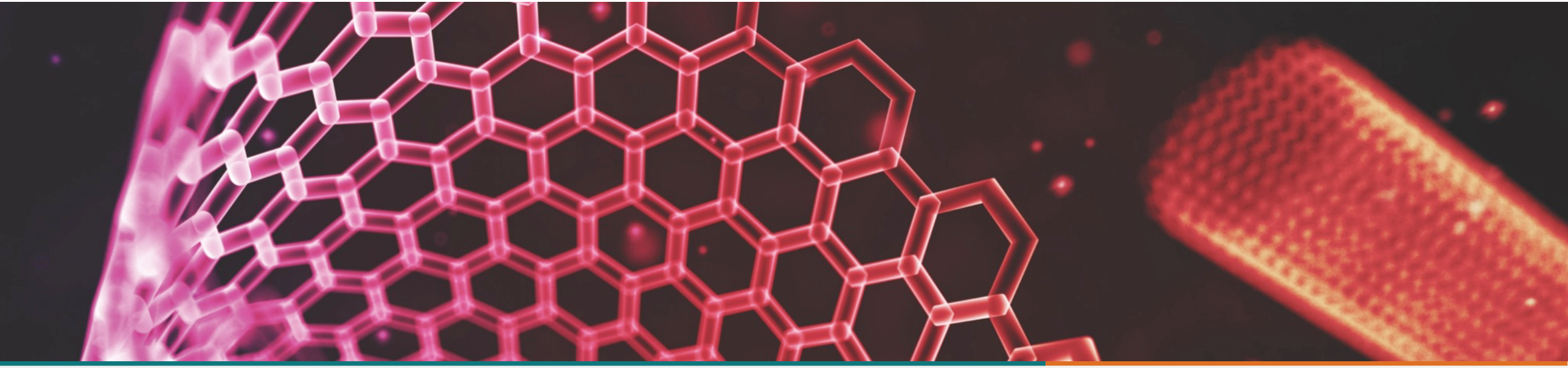
- Code becomes unreadable; you end up with code that goes deep and ends in the middle of your file, rather than code that runs as we're used to (top to bottom).
- Handling errors becomes intensely confusing, as you have to check for them at the start of every callback.
- Callback Hell

We will be taking a look at some examples of using callbacks in our lecture code.

Callback Hell

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Example 1: Reading Files With Callbacks

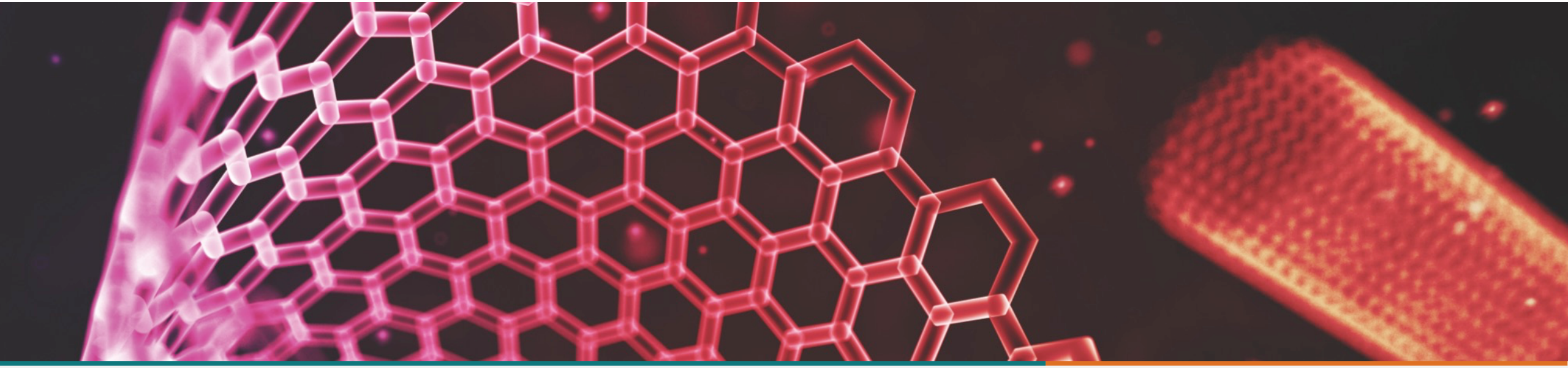


Our Goal

We are going to look at a node script that will perform the following chain of events:

1. It will use prompt to ask the user for the name of a file to open.
2. Once that is complete, it will read the file.
3. Once that is complete, it will reverse the content of the file.
4. Once that is complete, it will save the file again.

Promises



What Is a Promise?

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code.
Multiple callback functions would create callback hell that leads to unmanageable code.
Events were not good at handling asynchronous operations.

Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks and events.

Benefits of Promises

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

Benefits of Promises

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

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!

For example, promise based code:

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_03/code/promises/app.js

Is much easier to follow compared to callback-based code:

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_03/code/callbacks/app.js

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

States of Promises

A Promise has three states:

- **fulfilled**: Action related to the promise succeeded.
- **rejected**: Action related to the promise failed.
- **pending**: Promise is still pending i.e. not fulfilled or rejected yet.

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

Using Promises

Let's look at creating a consuming a promise:

```
let isMomHappy = false;

// Define the Promise
let willIGetNewPhone = new Promise(function(resolve, reject) {
  if (isMomHappy) {
    var phone = {
        brand: 'Samsung',
        color: 'black'
    };
  resolve(phone); // fulfilled
} else {
  var reason = new Error('mom is not happy');
  reject(reason); // reject
}
});

// Consume the Promise
willIGetNewPhone
  .then((fulfilled) => {
    console.log(fulfilled);
  })
  .catch((error) => {
    console.log(error);
  });
```

Using Promises

We can use a shortcut to defining promises:

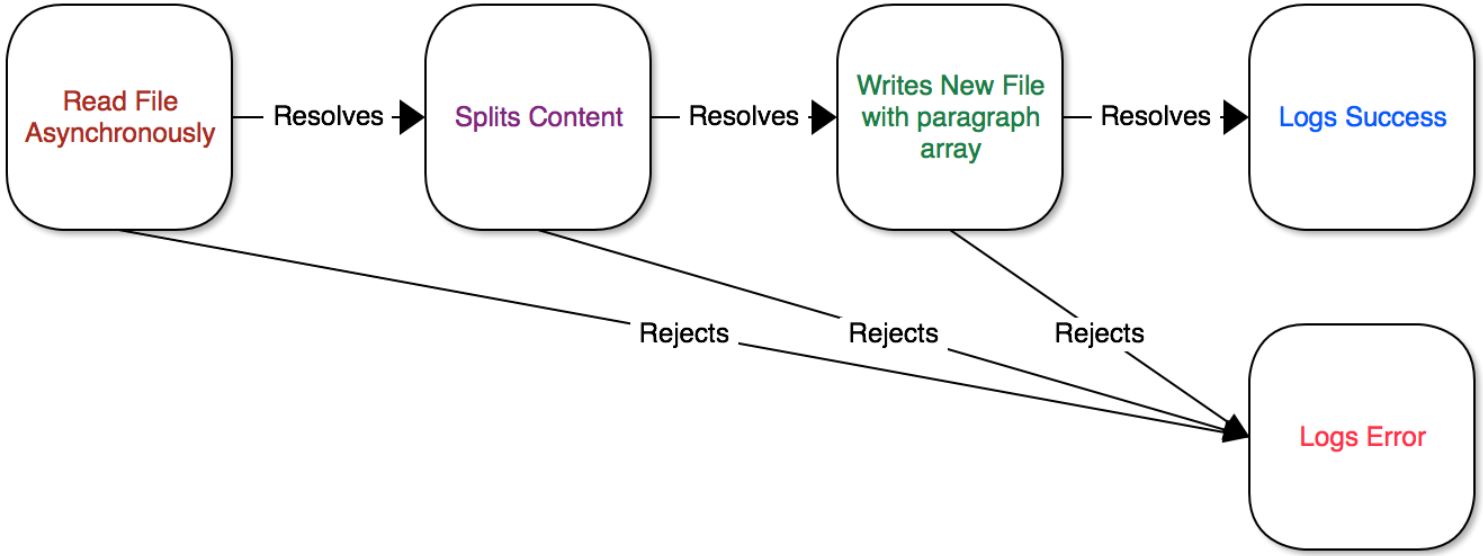
```
// Define the Promise
let willIGetNewPhone = new Promise(function(resolve, reject) {
  if (isMomHappy) {
    var phone = {
        brand: 'Samsung',
        color: 'black'
    };
  resolve(phone); // fulfilled
} else {
  var reason = new Error('mom is not happy');
  reject(reason); // reject
}
});

// We can define the promise like this instead
function willIGetNewPhone() {
  if (isMomHappy) {
    var phone = {
        brand: 'Samsung',
        color: 'black'
    };
  return Promise.resolve(phone);
  } else {
     return Promise.reject('Mom is not happy!');
  }
}
```

Chaining Promises

We can chain promises together using *then*

```
let isMomHappy = false;

// Define the Promises
function willIGetNewPhone() {
  if (isMomHappy) {
    let phone = {
        brand: 'Samsung',
        color: 'black'
    };
    return Promise.resolve(phone);
  } else {
      return Promise.reject('Mom is not happy!');
    }
}

let showOff = function(phone) {
  let message = `Hey friend, I have a new ${phone.color} ${phone.brand} phone`;
  return Promise.resolve(message);
};

// Consume the Promise
willIGetNewPhone()
  .then(showOff)
  .then((showOffResult) => {
    console.log(showOffResult);
}).catch((error) => {
  console.log(error);
});
```

```
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!

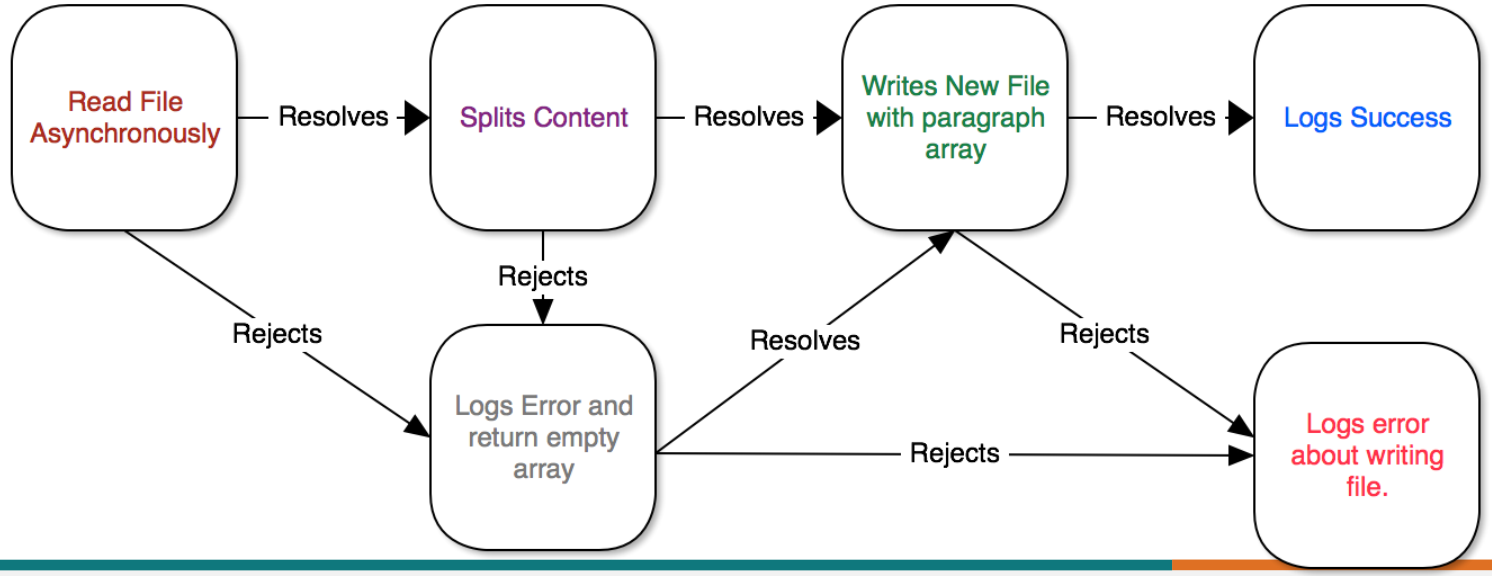
When a promise has an error, all promises that chain off that promise will reject with the same error until it is caught and handled.

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);
})
```



Converting Callbacks to Promises

We will use a node package called **bluebird** to convert methods that take callbacks to methods that return promises.

To convert one method, we use bluebird's **promisify** method; to convert each method in an object, we use **promisifyAll**

This will make a copy of each method, that ends with the term **Async** and returns a promise.

```
1   const bluebird = require("bluebird");
2   const Promise = bluebird.Promise;
3
4   const prompt = bluebird.promisifyAll(require("prompt"));
5   const fs = bluebird.promisifyAll(require("fs"));
6
7   const getFileOperation = {
8     name: "fileName",
9     description: "What file do you want to open?"
10  };
11
12  prompt
13    .getAsync([getFileOperation])
14    .then(function(result) {
```

Converting a Callback to a Promise

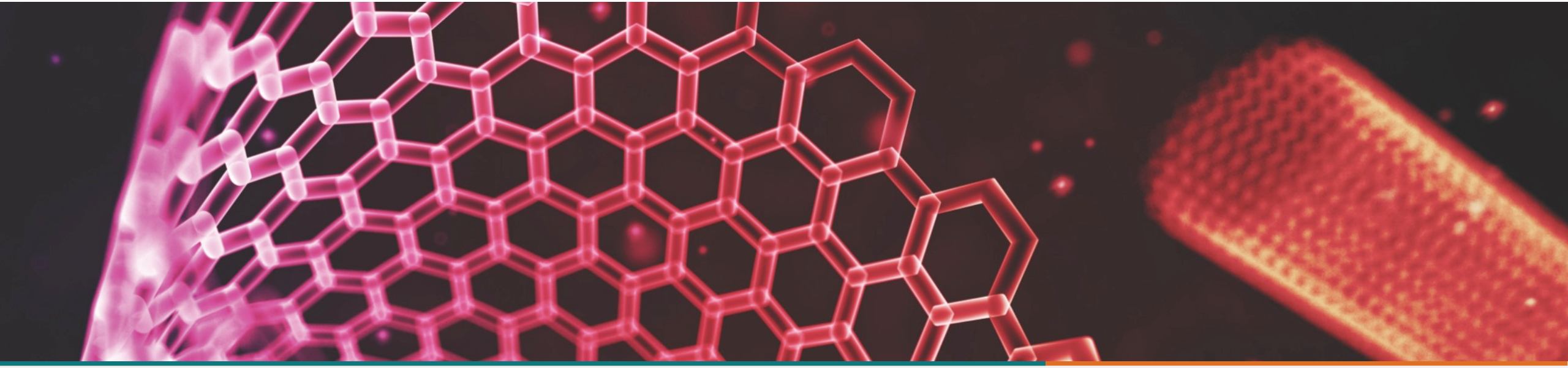
Rather than using callbacks, many people opt to convert their methods to return promises, even if they use code that internally uses callback.

There are two strategies for converting converting callbacks to promises. Let us take, for example, converting ***`fs.readFile`*** from a callback to a promise:

- Manually writing a method that returns a promise, which internally calls **`fs.readFile`**
- Using a promise library, such as bluebird, to make a copy of ***`fs`*** that has methods auto generated to return promises. It will make a method called ***`readFileAsync`***, which returns a promise.

This concept is called colloquially known as promisifying an operation.

Example 2: Reading Files With Promises



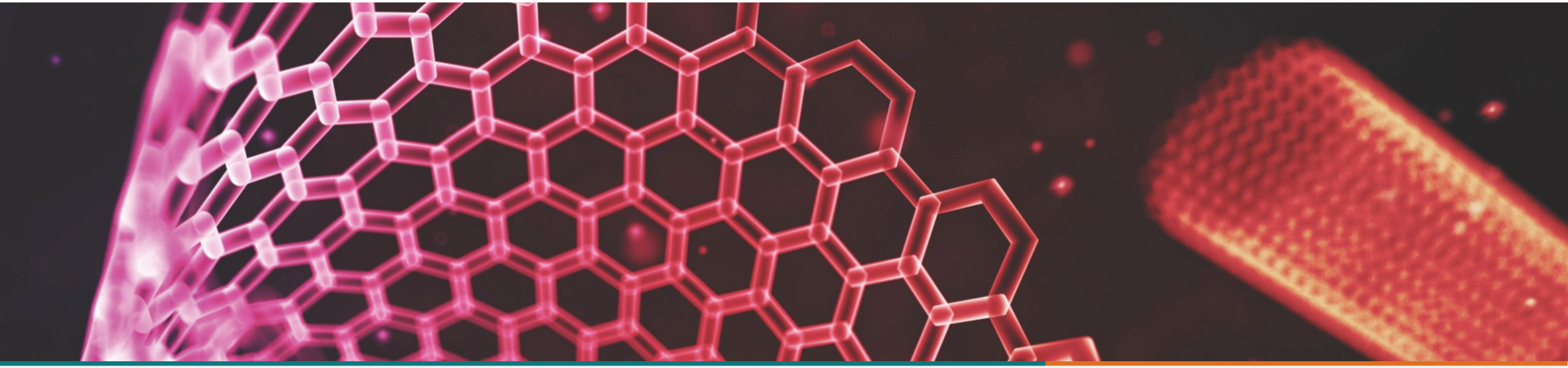
Our Goal

We are now going to convert the previous script to perform the same operation, however this time we will use promises.,

1. We will first use Bluebird to promisify the entirety of ***`fs`*** and *prompt*.
2. We will then perform our goals from before, this time using promises.

You can see this code demonstrated in the **promise** folder.

Async/Await



Asynchronous Code Is Messy

In general, writing asynchronous code is syntactically messy.

- Ends up in deeply nested callbacks
- Creates huge promise chains
- Hard to error check

As a result, over the last several years, the JavaScript language has added a concept of **async functions and await**ing the end of an asynchronous operation.

Async Functions

A function can be defined as an `async function`, which means that the function will automatically return a promise (even if there are no asynchronous operation in that function).

- By marking a function as `async`, you allow the `await` keyword to be used inside of the function.

Besides their ability to `await` promises and their guaranteed returning of a promise, there is no difference between a *function* and an *async function*.

You can ONLY use the `await` keyword inside an `async` function.

Awaiting Promises

The `await` keyword can only be used inside of an `async function`

When you `await` a promise, you will cause the rest of the function to execute after that promise resolves or rejects. If the promise rejects, an error will be thrown on the line that awaits it.

- This allows you to use `try`/`catch` syntax in asynchronous operations!
- The result of an `await` operation is whatever the promise resolves to. If the promise does not resolve to a value, it will have a result of `undefined`.

Example

On the top, we see promises. On the bottom, we see the same code written in ***async***/***await*** notation.

```
12  prompt
13    .getAsync([getFileOperation])
14    .then(function(result) {
15      const fileName = result.fileName;
16      if (!fileName) {
17        throw "Need to provide a file name";
18      }
19
20      console.log(`About to read ${fileName} if it exists`);
21
22      return fileName;
23    })
24    .then(function(fileName) {
25      return fs.readFileAsync(fileName, "utf-8").then(function(data) {
26        return { fileName: fileName, fileContent: data };
27      });
28    })
```

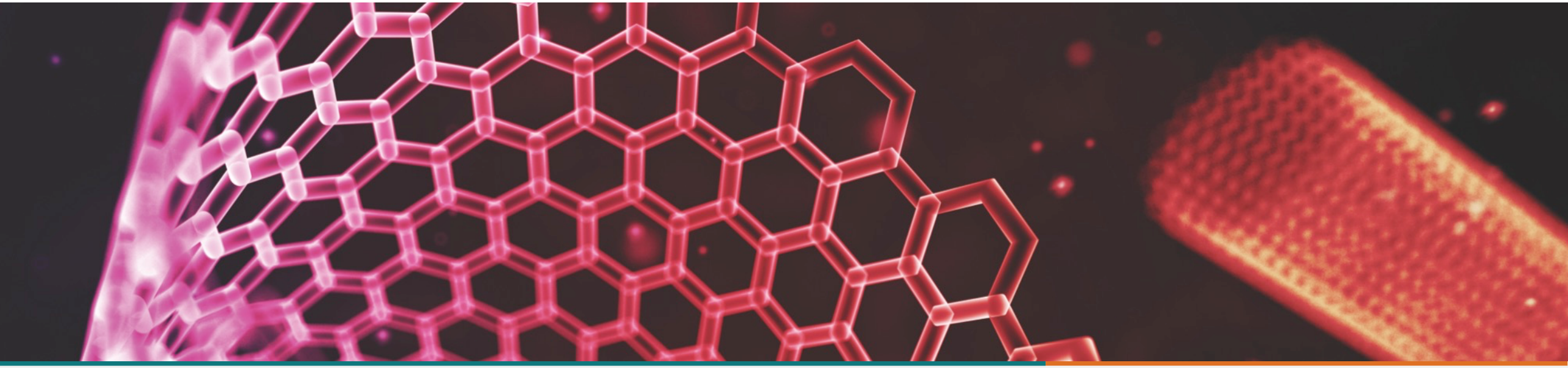
```
8   async function main() {
9     const getFileOperation = {…
12    };
13
14    // Gets result of user input
15    const promptResult = await prompt.getAsync([getFileOperation]);
16    const fileName = promptResult.fileName;
17
18    if (!fileName) {
19      throw "Need to provide a file name";
20    }
21
22    console.log(`About to read ${fileName} if it exists`);
23    const fileContent = await fs.readFileAsync(fileName, "utf-8");
24
```

What Benefit Does It Have?

By using `async` and `await` we can create code that is written and read in the order that which the operations will complete, while still allowing the functions themselves to be asynchronous.

Internally, the functions that run async code will still be constantly giving up execution cycles to perform other operations while they continue their tasks, however your code can abstract over that.

Example 3: Reading Files With Async/Await



Our Goal

We are now going to convert the previous script to perform the same operation, however this time we will use ***async***/***await***.,

1. We will still use Bluebird to promisify the entirety of ***fs*** and *prompt*.
2. We will then perform our goals from before, this time using promises.

You can see this code demonstrated in the **async-await** folder.

Questions?