Week: 7
Topic: Client-Side Frameworks and Libraries

Intended Learning Outcomes
At the end of the topic students will be able to:

Utilize Popular Frameworks
Understand Ajax and Fetch API for asynchronous communication.
Be able to perform Handling promises and callbacks.

Learning Activities

Introduction to frontend frameworks (React, Vue.Js, Angular)
Asynchronous JavaScript.

1 Vue JS

1.1 Installation

To install the CLI, run the following command in your terminal:

```
npm install --global @vue/cli
```

Or if you'd prefer to use yarn:

```
yarn global add @vue/cli
```

Once installed, to initialize a new project you can then open a terminal in the directory you want to create the project in, and run vue create <project-name>. The CLI will then give you a list of project configurations you can use. There are a few preset ones, and you can make your own. These options let you configure things like TypeScript, linting, vue-router, testing, and more.

We'll look at using this below.

1.2 Initializing a new project

To explore various features of Vue, we will be building up a sample todo list app. We'll begin by using the Vue CLI to create a new app framework to build our app into.

In terminal, cd to where you'd like to create your sample app, then run vue create moz-todo-vue. You can choose the default option Default ([Vue 3] babel, eslint) by pressing Enter to proceed. The CLI will now begin scaffolding out your project, and installing all of your dependencies.

If you've never run the Vue CLI before, you'll get one more question — you'll be asked to choose a package manager which defaults to yarn. The Vue CLI will default to this package manager from now on. If you need to use a different package manager after this, you can pass in a flag --packageManager=<package-manager>, when you run vue create. So if you wanted to create the moz-todo-vue project with npm and you'd previously chosen yarn, you'd run vue create moz-todo-vue --packageManager=npm.

Note: We've not gone over all of the options here, but you can find more information on the CLI in the Vue docs.

1.3 Project structure

If everything went successfully, the CLI should have created a series of files and directories for your project. The most significant ones are as follows:

package.json: This file contains the list of dependencies for your project, as well as some metadata and eslint configuration.
yarn.lock: If you chose yarn as your package manager, this file will be generated with a list of all the dependencies and sub-dependencies that your project needs.
babel.config.js: This is the config file for Babel, which transforms modern JavaScript features being used in development code into older syntax that is more cross-browser compatible in production code. You can register additional babel plugins in this file.
jsconfig.json: This is a config file for Visual Studio Code and gives context for VS Code on your project structure and assists auto-completion.
public: This directory contains static assets that are published, but not processed by Webpack during build (with one exception; index.html gets some processing).
favicon.ico: This is the favicon for your app. Currently, it's the Vue logo.
index.html: This is the template for your app. Your Vue app is run from this HTML page, and you can use lodash template syntax to interpolate values into it.
Note: this is not the template for managing the layout of your application — this template is for managing static HTML that sits outside of your Vue app. Editing this file typically only occurs in advanced use cases.

src: This directory contains the core of your Vue app.
main.js: this is the entry point to your application. Currently, this file initializes your Vue application and signifies which HTML element in the index.html file your app should be attached to. This file is often where you register global components or additional Vue libraries.
App.vue: this is the top-level component in your Vue app. See below for more explanation of Vue components.
components: this directory is where you keep your components. Currently, it just has one example component.
assets: this directory is for storing static assets like CSS and images. Because these files are in the source directory, they can be processed by Webpack. This means you can use pre-processors like Sass/SCSS or Stylus.
Note: Depending on the options you select when creating a new project, there might be other directories present (for example, if you choose a router, you will also have a views directory).

1.4 .vue files (single file components)

Like in many front-end frameworks, components are a central part of building apps in Vue. These components let you break a large application into discrete building blocks that can be created and managed separately, and transfer data between each other as required. These small blocks can help you reason about and test your code.

While some frameworks encourage you to separate your template, logic, and styling code into separate files, Vue takes the opposite approach. Using Single File Components (SFC), Vue lets you group your templates, corresponding script, and CSS all together in a single file ending in .vue. These files are processed by a JS build tool (such as Webpack), which means you can take advantage of build-time tooling in your project. This allows you to use tools like Babel, TypeScript, SCSS and more to create more sophisticated components.

As a bonus, projects created with the Vue CLI are configured to use .vue files with Webpack out of the box. In fact, if you look inside the src folder in the project we created with the CLI, you'll see your first .vue file: App.vue.

Let's explore this now.

1.5 App.vue

Open your App.vue file — you'll see that it has three parts: <template>, <script>, and <style>, which contain the component's template, scripting, and styling information. All Single File Components share this same basic structure.

<template> contains all the markup structure and display logic of your component. Your template can contain any valid HTML, as well as some Vue-specific syntax that we'll cover later.

Note: By setting the lang attribute on the <template> tag, you can use Pug template syntax instead of standard HTML — <template lang="pug">. We'll stick to standard HTML through this tutorial, but it is worth knowing that this is possible.

<script> contains all of the non-display logic of your component. Most importantly, your <script> tag needs to have a default exported JS object. This object is where you locally register components, define component inputs (props), handle local state, define methods, and more. Your build step will process this object and transform it (with your template) into a Vue component with a render() function.

In the case of App.vue, our default export sets the name of the component to App and registers the HelloWorld component by adding it into the components property. When you register a component in this way, you're registering it locally. Locally registered components can only be used inside the components that register them, so you need to import and register them in every component file that uses them. This can be useful for bundle splitting/tree shaking since not every page in your app necessarily needs every component.

```
import HelloWorld from "./components/HelloWorld.vue";

export default {
  name: "App",
  components: {
    //You can register components locally here.
    HelloWorld,
  },
};
```
Note: If you want to use TypeScript syntax, you need to set the lang attribute on the <script> tag to signify to the compiler that you're using TypeScript — <script lang="ts">.

<style> is where you write your CSS for the component. If you add a scoped attribute — <style scoped> — Vue will scope the styles to the contents of your SFC. This works similar to CSS-in-JS solutions, but allows you to just write plain CSS.

Note: If you select a CSS pre-processor when creating the project via the CLI, you can add a lang attribute to the <style> tag so that the contents can be processed by Webpack at build time. For example, <style lang="scss"> will allow you to use SCSS syntax in your styling information.

1.6 Running the app locally

The Vue CLI comes with a built-in development server. This allows you to run your app locally so you can test it easily without needing to configure a server yourself. The CLI adds a serve command to the project's package.json file as an npm script, so you can easily run it.

In your terminal, try running npm run serve (or yarn serve if you prefer yarn). Your terminal should output something like the following:

```
INFO  Starting development server...
98% after emitting CopyPlugin

 DONE  Compiled successfully in 18121ms

 App running at:
 - Local:   <http://localhost:8080/>
 - Network: <http://192.168.1.9:8080/>

 Note that the development build is not optimized.
 To create a production build, run npm run build.
```

If you navigate to the "local" address in a new browser tab (this should be something like http://localhost:8080 as stated above, but may vary based on your setup), you should see your app. Right now, it should contain a welcome message, a link to the Vue documentation, links to the plugins you added when you initialized the app with your CLI, and some other useful links to the Vue community and ecosystem.

default Vue app render, with Vue logo, welcome message, and some documentation links
Making a couple of changes
Let's make our first change to the app — we'll delete the Vue logo. Open the App.vue file, and delete the <img> element from the template section:

HTML
```html
<img alt="Vue logo" src="./assets/logo.png" />
```
If your server is still running, you should see the logo removed from the rendered site almost instantly. Let's also remove the HelloWorld component from our template.

First of all delete this line:

HTML
```html
<HelloWorld msg="Welcome to Your Vue.js App" />
```
If you save your App.vue file now, the rendered app will throw an error because we've registered the component but are not using it. We also need to remove the lines from inside the <script> element that

import and register the component:

Delete these lines now:

```
JS
import HelloWorld from "./components/HelloWorld.vue";
```

```
JS
components: {
  HelloWorld;
}
```

The <template> tag is empty now so you'll see an error saying The template requires child element in both the console and the rendered app. You can fix this by adding some content inside the <template> tag and we can start with a new <h1> element inside a <div>. Since we're going to be creating a todo list app below, let's set our heading to "To-Do List" like so:

```
HTML
<template>
  <div id="app">
    <h1>To-Do List</h1>
  </div>
</template>
```

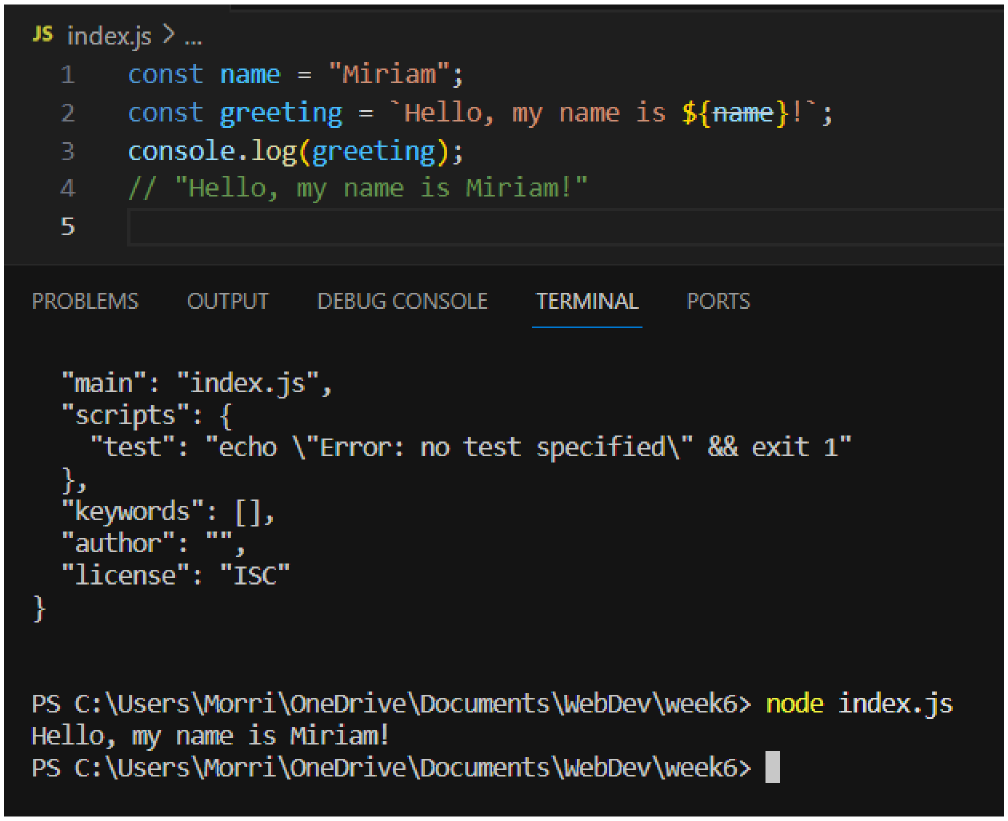
App.vue will now show our heading, as you'd expect.

2 Asynchronous JavaScript

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

2.1 Synchronous Programming

Consider the following code:



index.js > ...

```javascript
const name = "Miriam";
const greeting = `Hello, my name is ${name}!`;
console.log(greeting);
// "Hello, my name is Miriam!"

```

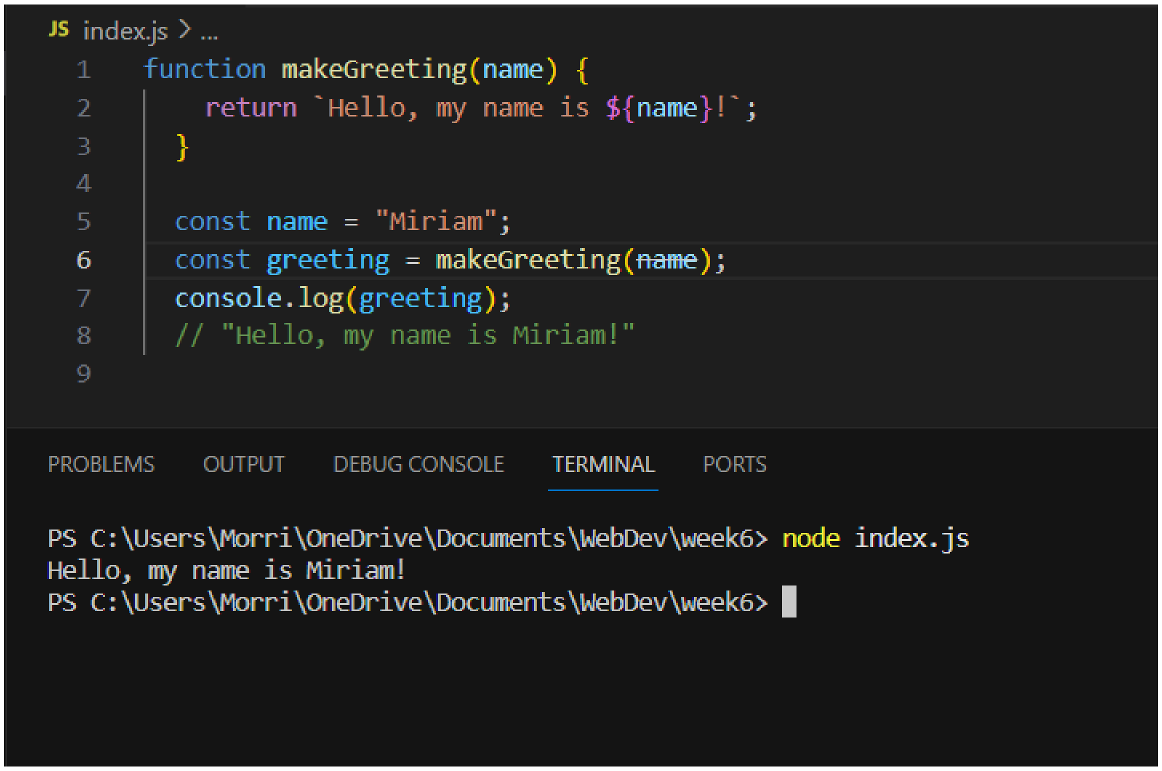
```
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


PS C:\Users\Morri\OneDrive\Documents\WebDev\week6> node index.js
Hello, my name is Miriam!
PS C:\Users\Morri\OneDrive\Documents\WebDev\week6>
```

This code:

i. Declares a string called name.
ii. Declares another string called greeting, which uses name.
iii. Outputs the greeting to the JavaScript console.

We should note here that the browser effectively steps through the program one line at a time, in the order we wrote it. At each point, the browser waits for the line to finish its work before going on to the next line. It has to do this because each line depends on the work done in the preceding lines.

That makes this a synchronous program. It would still be synchronous even if we called a separate function, like this:



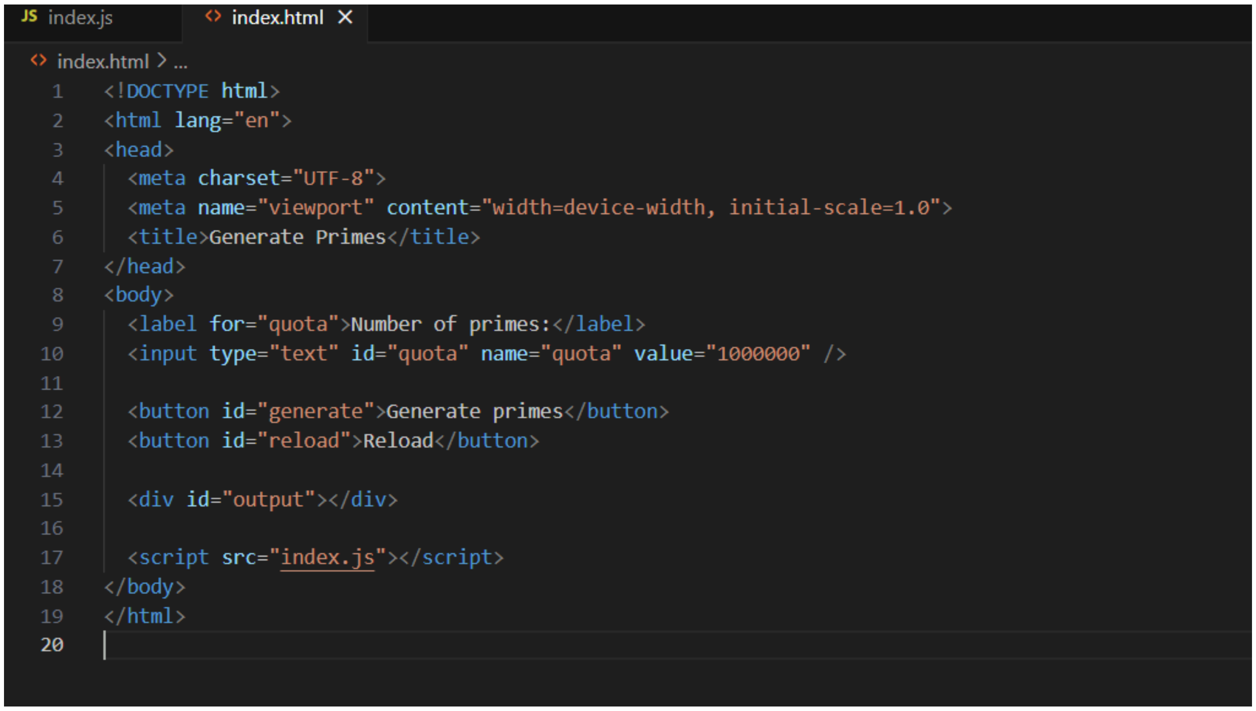
```javascript
function makeGreeting(name) {
  return `Hello, my name is ${name}!`;
}

const name = "Miriam";
const greeting = makeGreeting(~~name~~);
console.log(greeting);
// "Hello, my name is Miriam!"
```

```
PS C:\Users\Morri\OneDrive\Documents\WebDev\week6> node index.js
Hello, my name is Miriam!
PS C:\Users\Morri\OneDrive\Documents\WebDev\week6>
```

Here, makeGreeting() is a synchronous function because the caller has to wait for the function to finish its work and return a value before the caller can continue.

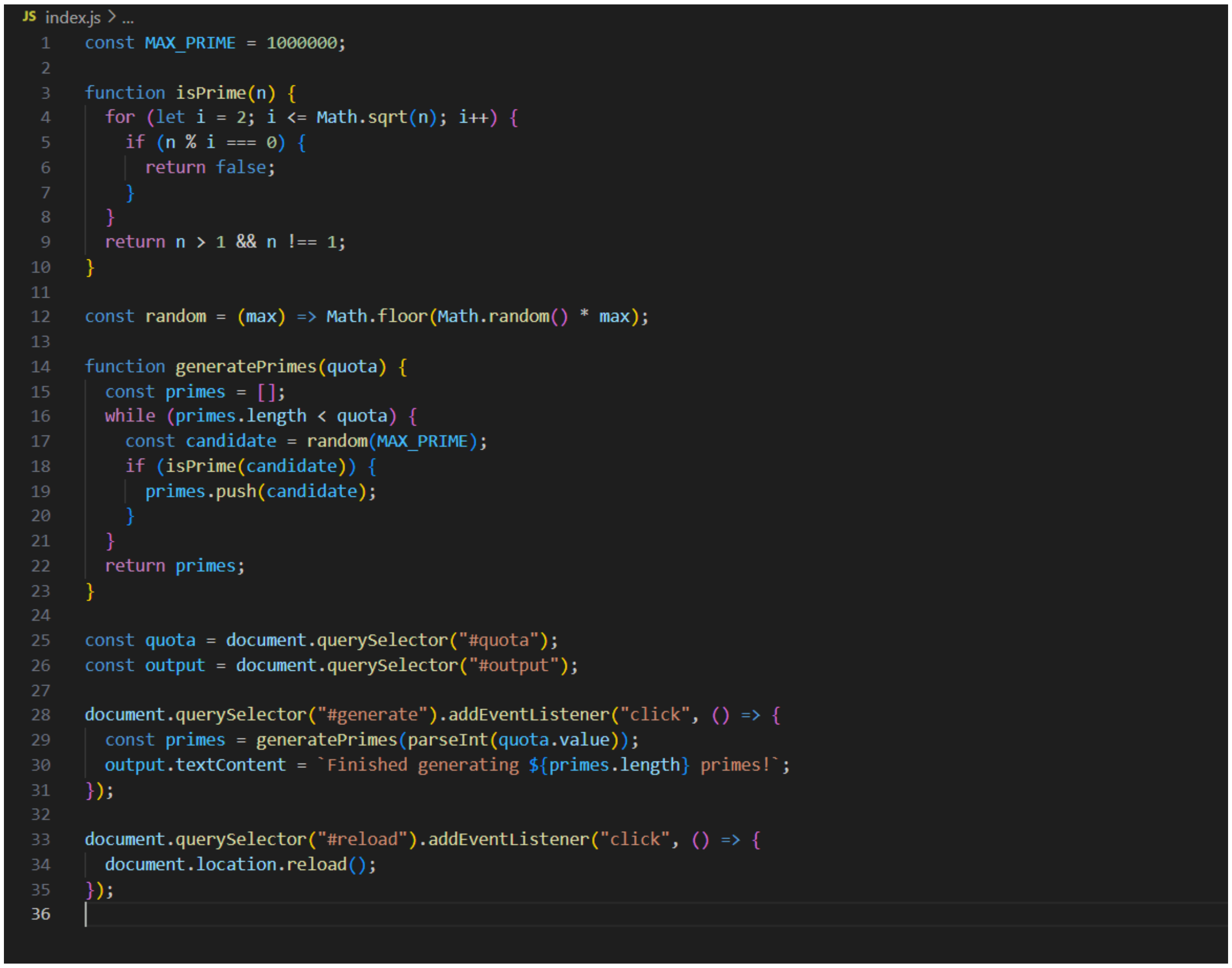
2.1.1 Single threaded long running synchronous code

The program below uses a very inefficient algorithm to generate multiple large prime numbers when a user clicks the "Generate primes" button. The higher the number of primes a user specifies, the longer the operation will take.
HTML Code



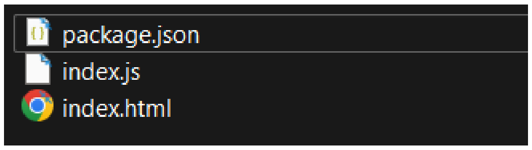
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Generate Primes</title>
</head>
<body>
  <label for="quota">Number of primes:</label>
  <input type="text" id="quota" name="quota" value="1000000" />

  <button id="generate">Generate primes</button>
  <button id="reload">Reload</button>

  <div id="output"></div>

  <script src="index.js"></script>
</body>
</html>

```

JS Code



```
index.js > ...
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1 && n !== 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}

const quota = document.querySelector("#quota");
const output = document.querySelector("#output");

document.querySelector("#generate").addEventListener("click", () => {
  const primes = generatePrimes(parseInt(quota.value));
  output.textContent = `Finished generating ${primes.length} primes!`;
});

document.querySelector("#reload").addEventListener("click", () => {
  document.location.reload();
});
```

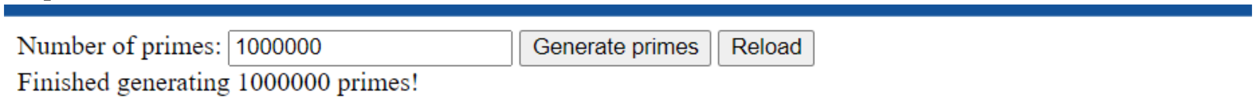
Run index.html



```
package.json
index.js
index.html
```

Try clicking "Generate primes". Depending on how fast your computer is, it will probably take a few seconds before the program displays the "Finished!" message.

Output



Number of primes: 1000000

Finished generating 1000000 primes!

2.1.2 Single threaded long running synchronous code with user input

The next example is just like the last one, except we added a text box for you to type in. This time, click "Generate primes", and try typing in the text box immediately after.

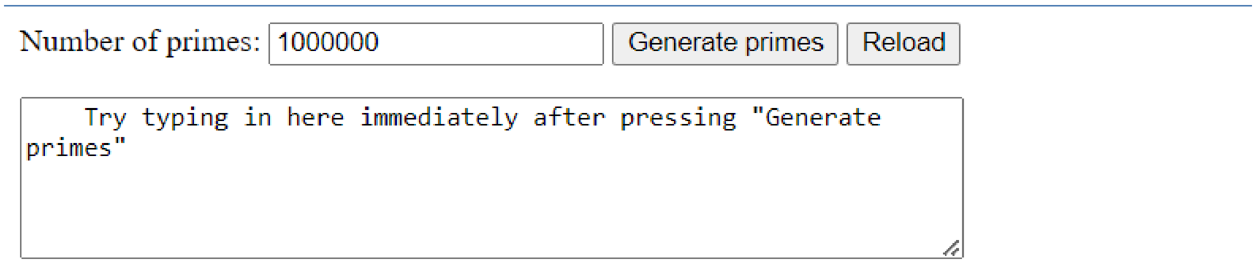
You'll find that while our generatePrimes() function is running, our program is completely unresponsive: you can't type anything, click anything, or do anything else.

HTML Code

```html
<!DOCTYPE html>
<html>
<head>
  <title>Prime Number Generator</title>
  <style>
    textarea {
      display: block;
      margin: 1rem 0;
    }
  </style>
</head>
<body>
  <label for="quota">Number of primes:</label>
  <input type="text" id="quota" name="quota" value="1000000" />

  <button id="generate">Generate primes</button>
  <button id="reload">Reload</button>

  <textarea id="user-input" rows="5" cols="62">
    Try typing in here immediately after pressing "Generate primes"
  </textarea>

  <div id="output"></div>

  <script src="index.js"></script>
</body>
</html>
```

JS Code

```js
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}

const quota = document.querySelector("#quota");
const output = document.querySelector("#output");

document.querySelector("#generate").addEventListener("click", () => {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value} primes!`;
});

document.querySelector("#reload").addEventListener("click", () => {
  document.location.reload();
});
```


Number of primes: 1000000

```
    Try typing in here immediately after pressing "Generate
primes"
```

The reason for this is that this JavaScript program is single threaded. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

What we need is a way for our program to:

i. Start a long-running operation by calling a function.
ii. Have that function start the operation and return immediately, so that our program can still be responsive to other events.
iii. Have the function execute the operation in a way that does not block the main thread, for example by starting a new thread.
iv. Notify us with the result of the operation when it eventually completes.

That's precisely what asynchronous functions enable us to do. The rest of this module explains how they are implemented in JavaScript.

2.2 Promises

Promises are the foundation of asynchronous programming in modern JavaScript. A promise is an object returned by an asynchronous function, which represents the current state of the operation. At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.

2.2.1 Using the fetch() API

In this example, we'll download the JSON file from https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json, and log some information about it.

To do this, we'll make an HTTP request to the server. In an HTTP request, we send a request message to a remote server, and it sends us back a response. In this case, we'll send a request to get a JSON file from the server.

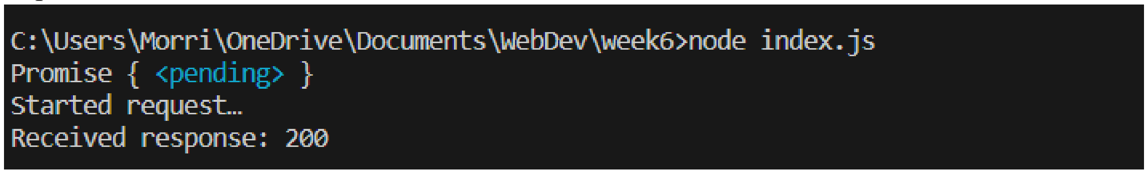
JS Code

```
const fetchPromise = fetch(

"https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log(`Received response: ${response.status}`);
});

console.log("Started request…");
```

Here we are:

i. calling the fetch() API, and assigning the return value to the fetchPromise variable
ii. immediately after, logging the fetchPromise variable. This should output something like: Promise { <state>: "pending" }, telling us that we have a Promise object, and it has a state whose value is "pending". The "pending" state means that the fetch operation is still going on.
iii. passing a handler function into the Promise's then() method. When (and if) the fetch operation succeeds, the promise will call our handler, passing in a Response object, which contains the server's response.
iv. logging a message that we have started the request.

Ouput



```
C:\Users\Morri\OneDrive\Documents\WebDev\week6>node index.js
Promise { <pending> }
Started request…
Received response: 200
```

2.2.2 async and await

The async keyword gives you a simpler way to work with asynchronous promise-based code. Adding async at the start of a function makes it an async function:

JS
```
async function myFunction() {
  // This is an async function
}
```
Inside an async function, you can use the await keyword before a call to a function that returns a promise. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown.

This enables you to write code that uses asynchronous functions but looks like synchronous code. For example, we could use it to rewrite our fetch example:

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be
settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(

"https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/prod
ucts.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to
be settled
    // the `response.json()` call will either return the parsed JSON object or
throw an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```
Here, we are calling await fetch(), and instead of getting a Promise, our caller gets back a fully complete Response object, just as if fetch() were a synchronous function!