University of East Anglia
COMPUTING SCIENCES

Lab 6: JavaScript Async and Fetch

Before attempting this lab, you will need to have completed **Lab 5: JavaScript Events and Forms**.

Learning objectives

By completing this lab, you should be able to:

- Describe Asynchronous and Synchronous Programming concepts
- Explain the needs for Asynchronous in Web programming
- Write Asynchronous JavaScript code using Promise
- Write JavaScript code using fetch API
- Evaluate the difference between Cross-Origin Resource Sharing (CORS) requests vs HTTP requests

Background

Asynchronous JavaScript (MDN) is an advanced topic. In this module I will cover some basic concept as it is important for Web Programming, in particular, how it can be used to effectively handle potential blocking operations, such as fetching resources from a server.

This week lab exercises are building on from we have learned so far, the JavaScript building blocks (MDN) such as conditional statements, loops, functions, and events.

First, we need to understand what is Asynchronous Programming (MDN).

Asynchronous programming is a technique that enables our program to start a potentially long-running task and still be able to handle other events at the same time without having to wait for that task to complete. Once that task has finished, our program is presented with the result. This technique is very important for Web Programming. It is because there are many tasks or operations performed on the Internet that can take a while before we get the results. A good example is downloading a file from the Internet. The operation involves connecting to a web server (this takes some time), one may need to be authorised the access (this takes some time), once granted then the web server searches for that file (this takes some time), if the file exists then the web server retrieves the data from somewhere (this takes some time) and then the server sends the data over the Internet (this takes some time). Other operations could be accessing a user's camera or microphone, or send a file to a printer, as the program waits for external devices to respond.

If we were using Synchronous Programming (MDN) concept on the web then our program would have to wait for an operation to complete before moving on to the next task. This would mean the browser would be frozen every time it had to wait for a task, e.g. file download, to complete.

Other programming language, such as Java, solves this issue by having multi-threaded environment (MDN). **Thread** in computer science refers to the execution of running multiple tasks or programs at the same time. Each unit capable of executing code is called a thread. However, JavaScript is **single-threaded** programming language. Fortunately, it supports Asynchronous Programming concept. **Note**: JavaScript can be Synchronous as well as Asynchronous.

Asynchronous JavaScript is achieved through callbacks (MDN). A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action. ES6 supports Promise (MDN). 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. We will be using Promise in lab exercises. We will explore the use of fetch API (MDN).

Exercises

1. Create a new folder in your student U: directory called `js-async-fetch`. Copy all the files in the `js-events` folder to this new folder.

2. On your text editor, open this `js-async-fetch` folder.

3. Create a new HTML page called `hello.html` – Figure 1.

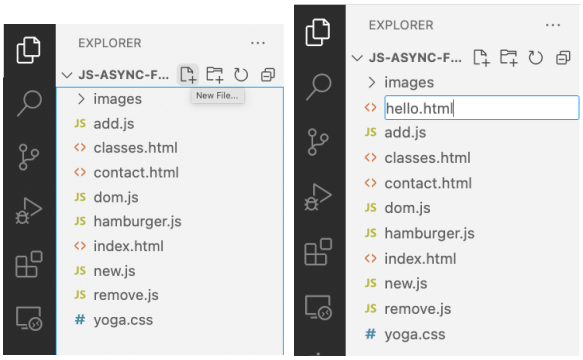


Figure 1. Create a new HTML page called hello.html

4. Copy all (Ctrl `a` on a keyboard) of the content in `contact.html` (Figure 2) and paste it to this `hello.html`

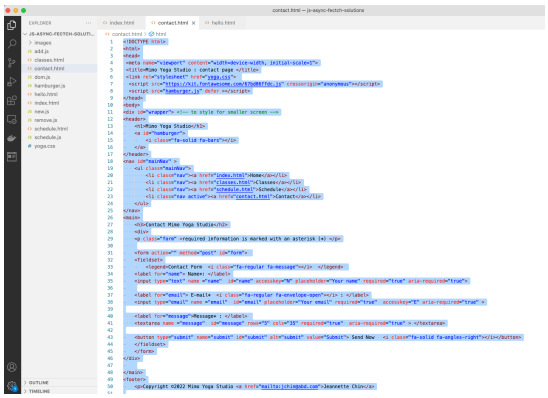


Figure 2. Copy the content in contact.html

5. In `hello.html`, remove everything inside the `<main>` tag (Figure 3)

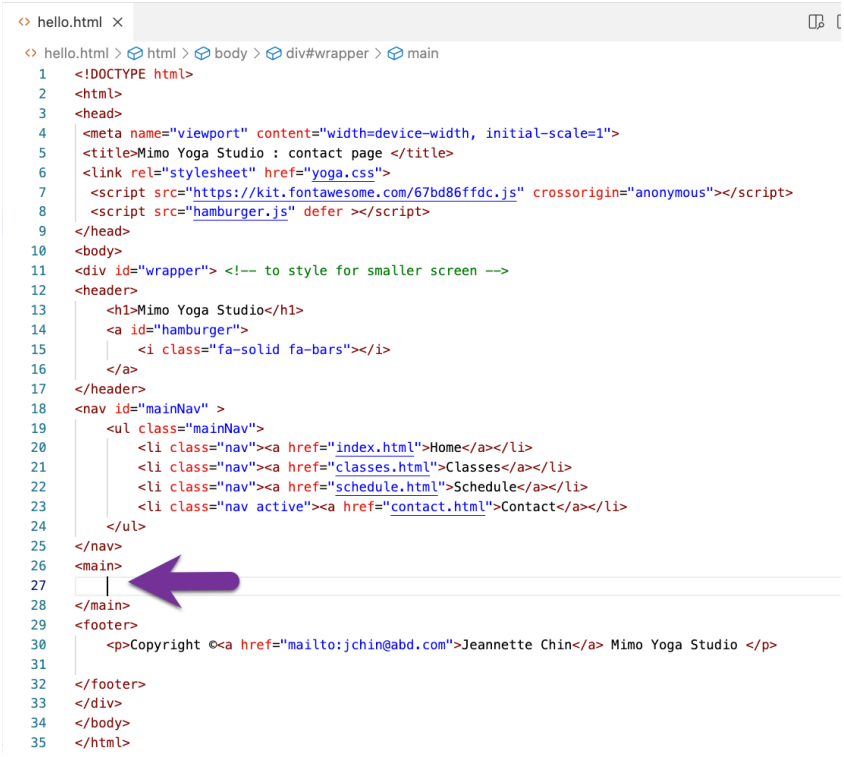


Figure 3. The content in hello.html

6. Now inside this `main` element, create a `div` element with an `id` called “album-view” and another `div` element with an `id` called “modal-view”, as shown in Figure 4.

hello.html

hello.html > html > body > div#wrapper > header > a#hamburger

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>Mimo Yoga Studio : contact page </title>
 <link rel="stylesheet" href="yoga.css">
  <script src="https://kit.fontawesome.com/67bd86ffdc.js" crossorigin="anonymous"></script>
  <script src="hamburger.js" defer ></script>
</head>
<body>
<div id="wrapper"> <!-- to style for smaller screen -->
<header>
    <h1>Mimo Yoga Studio</h1>
    <a id="hamburger">
        <i class="fa-solid fa-bars"></i>
    </a>
</header>
<nav id="mainNav" >
    <ul class="mainNav">
        <li class="nav"><a href="index.html">Home</a></li>
        <li class="nav"><a href="classes.html">Classes</a></li>
        <li class="nav"><a href="schedule.html">Schedule</a></li>
        <li class="nav active"><a href="contact.html">Contact</a></li>
    </ul>
</nav>
<main>
    <div id="album-view"></div>
    <div id="modal-view"></div>
</main>
<footer>
    <p>Copyright ©<a href="mailto:jchin@abd.com">Jeannette Chin</a> Mimo Yoga Studio </p>

</footer>
</div>
</body>
</html>
```

Figure 4. Create two new div elements in hello.html

7. In `hello.html`, add a navigation link to this Hello page. The nav should have an "active" class. Remove the "active" class from `contact.html` link – Figure 5.

```
<nav id="mainNav" >
    <ul class="mainNav">
        <li class="nav"><a href="index.html">Home</a></li>
        <li class="nav"><a href="classes.html">Classes</a></li>
        <li class="nav"><a href="schedule.html">Schedule</a></li>
        <li class="nav"><a href="contact.html">Contact</a></li>
        <li class="nav active"><a href="hello.html">Hello</a></li>
    </ul>
</nav>
```

Figure 5. The navigation links in hello.html

8. Now add a navigation link to this `hello.html` in **all other pages**. Figure 6 below shown the code in my index.html. Do the same for all other pages.

```
<nav id="mainNav" >
    <ul class="mainNav">
        <li class="nav active"><a href="index.html">Home</a></li>
        <li class="nav"><a href="classes.html">Classes</a></li>
        <li class="nav"><a href="schedule.html">Schedule</a></li>
        <li class="nav"><a href="hello.html">Hello</a></li>
    </ul>
</nav>
```

Figure 6. Add a navigation link to hello.html in index.html

9. Save the files. Test the pages using Chrome. Figures 7 – 8. Making sure that all navigations work including hamburger menu (Figure 9)

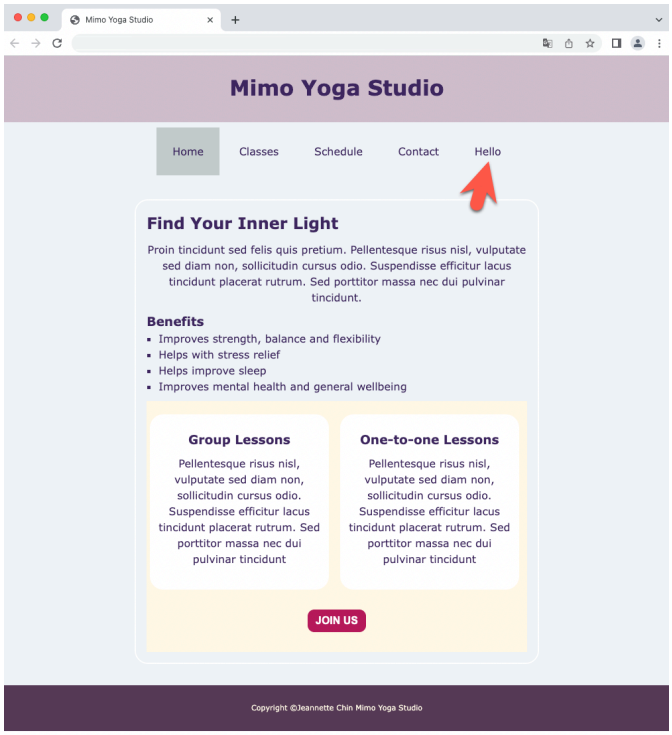


Figure 7. The output of index.html

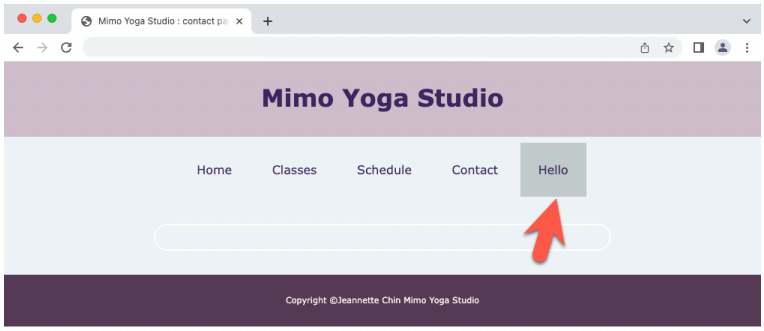


Figure 8. The output of hello.html

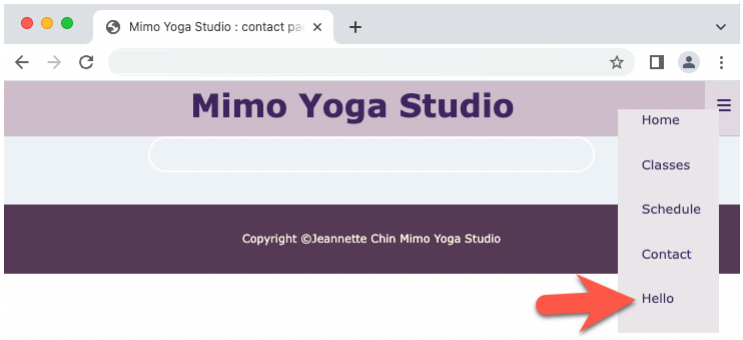
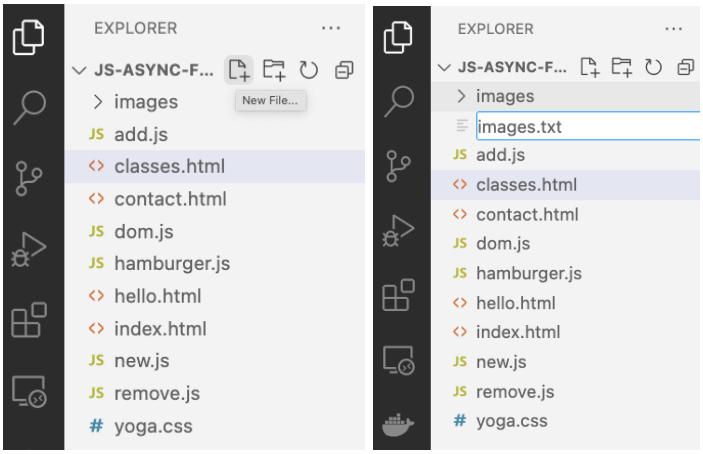


Figure 9. The hamburger menu in hello.html

10. Next, we will grab some images from web service. Gathering images via a web service:
    a) Create a new file called '`images.txt`'



    b) Open the link https://giphy.com/explore/hello and select 6 images of your choice by following these steps:
        i. Right click an image, select "open Link in New Private Tab" (Figure 10) if you are using Private window, or just a New Tab in a normal window.

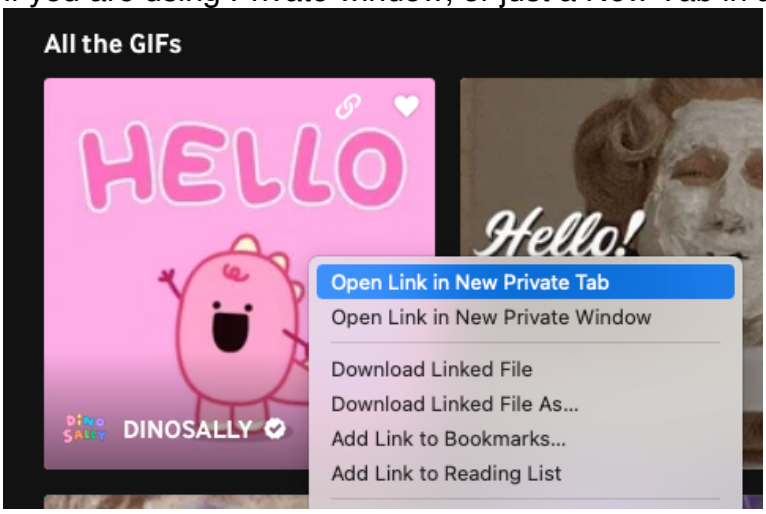


Figure 10. Right click on an image and select Open Link option

        ii. Select "share" option (Figure 11).

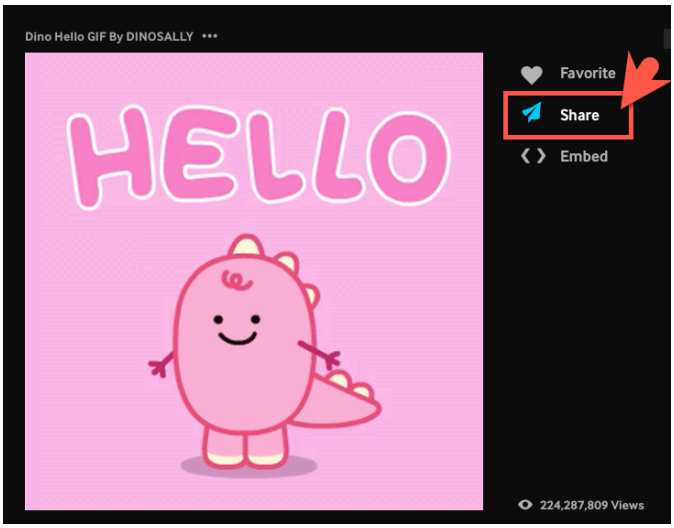


Figure 11. Select Share option

iii. Click “Copy GIF link” button (Figure 12)

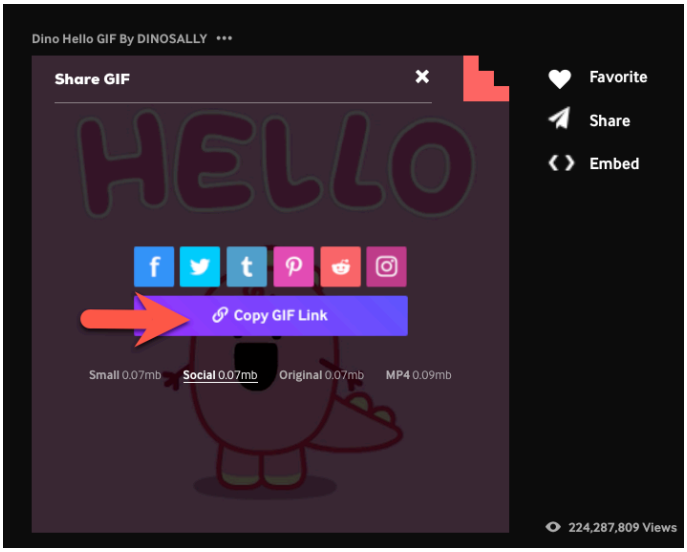


Figure 12. Click the “copy GIF link”

iv. Paste the link in `images.txt` file (Figure 13 and Figure 14)

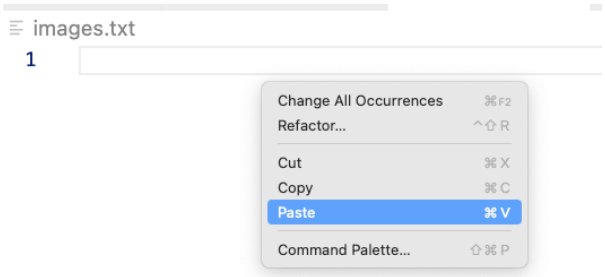


Figure 13. Paste the URL in `images.txt`

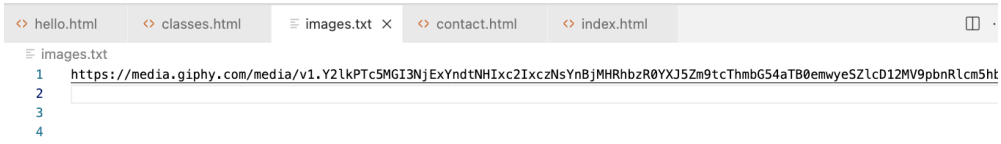


Figure 14. The result

v. Repeat the same procedure for the remaining 5 images you have selected.

vi. Figure 16 below shows the 6 URLs to the images I have selected in my `images.txt` file.

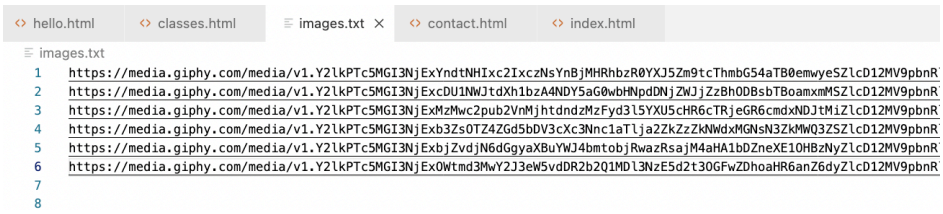


Figure 16. The content of Jeannette's `images.txt` file

11. In your text editor, create a new file named `fetch.js`. (Figure 17)

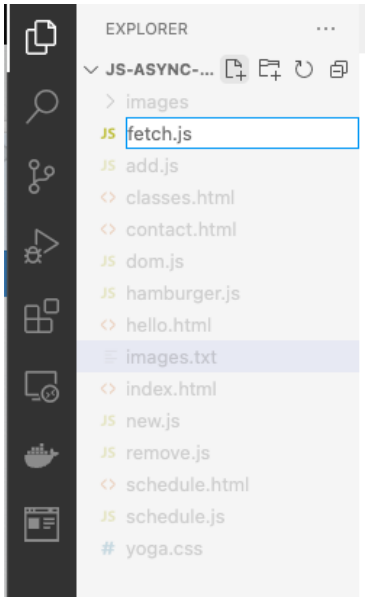


Figure 17. Create a new file using Visual Studio Code editor

12. In `fetch.js`:

a) write `fetch` function to read the `images.text` file we created earlier, as shown in Figure 18. Fetch API returns a Promise (MDN). We will use the `.then()` method to handle the response. In the code below, the handler function "onResponse" will be called if fetch operation is successful, otherwise "onError" handler will be executed if the operation fails. We will write these handler functions later.

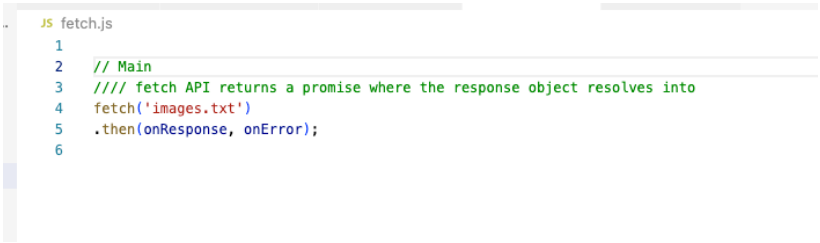


Figure 18. The code in fetch.js

b) now write these 2 handler functions for the fetch call, as shown in Figure 19.

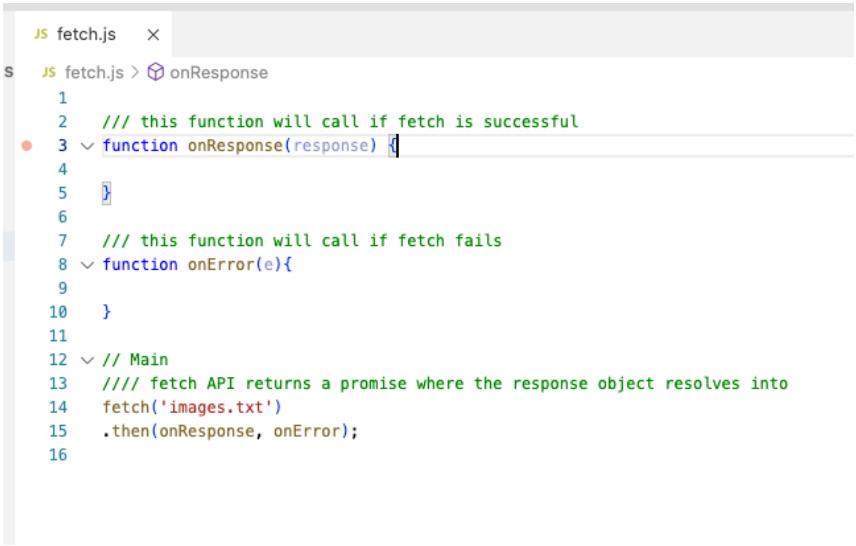


Figure 19. The code in fetch.js

Code: the `onResponse` function will take the response object (Line 3) as argument. The `onError` function will take the error object e (Line 8)

c) Save the file.

d) Now add this `fetch.js` to `hello.html` (Figure 20)

Figure 20. Add fetch.js to hello.html

e) Save the file. Now open the `hello.html` on Chrome, or if the file is already opened, refresh the page.

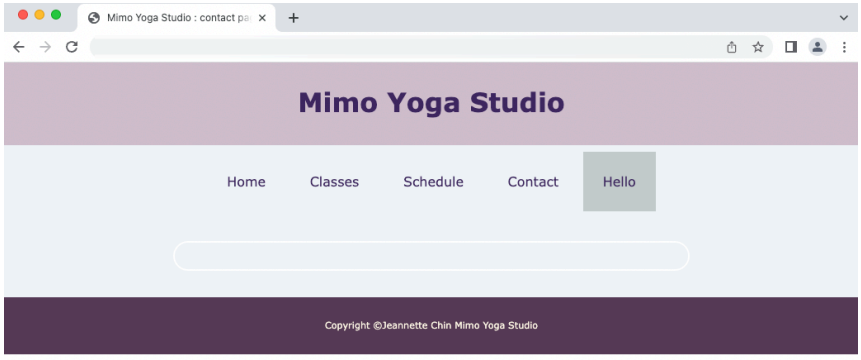


Figure 21. The output of hello.html

f) Now right click the page and select "inspect" option – Figure 22

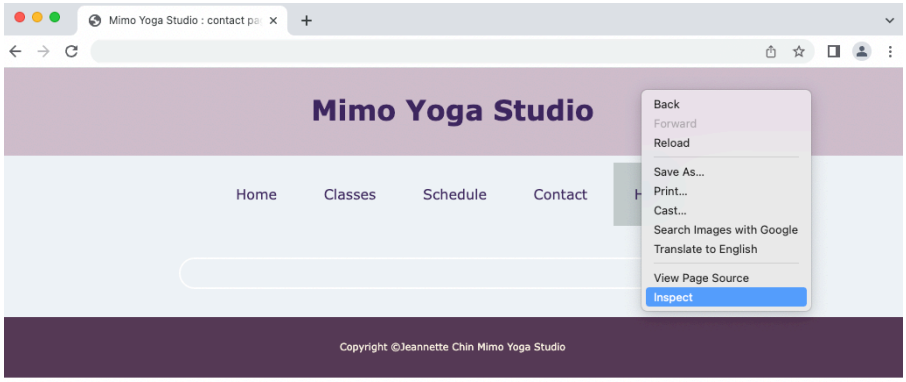


Figure 22. Right click and select inspect option

g) Now select Console Tab – Figure 23, you will see that there are a bunch of errors!

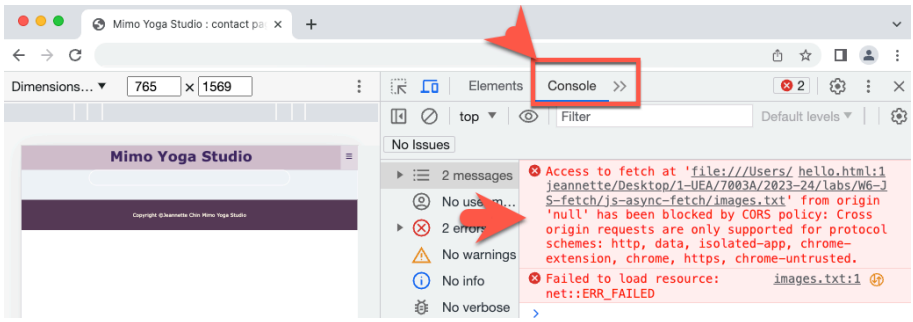


Figure 23. Select Console Tab on Chrome

h) Notice the error message - "….has been blocked by CORS (MDN) policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, chrome-untrusted, https." (Figure 24).

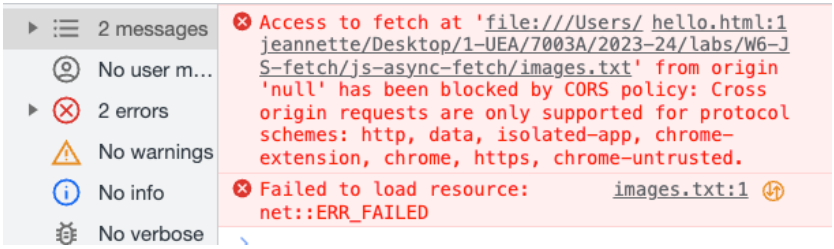


Figure 24. CORS error encountered

i) So how does this violation occur? Observe the output of Chrome, there is this icon called File (Figure 25) indicating we are using the File protocol to view the page.

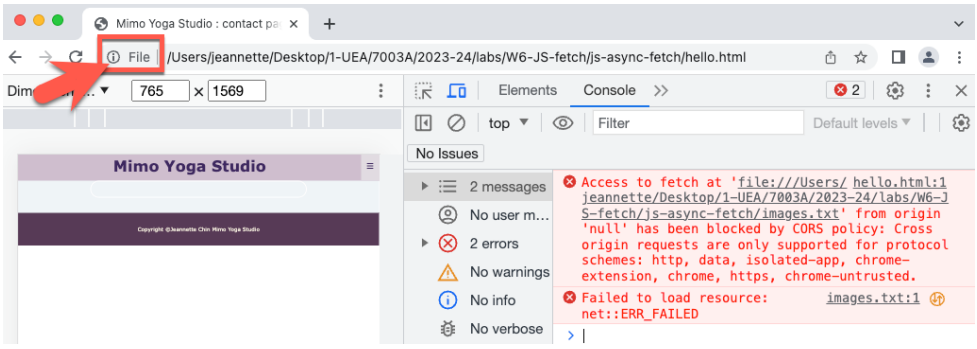


Figure 25. The File icon on Chrome

j) This is the problem. CORS policy prevents any browser using File protocol to access local resources. This is well intended and for security reason. We do not want our browser to access any files store in our computer by somebody from the Internet.

k) To solve this issue, we will run a local server and access this webpage. Type **anaconda powershell prompt**:

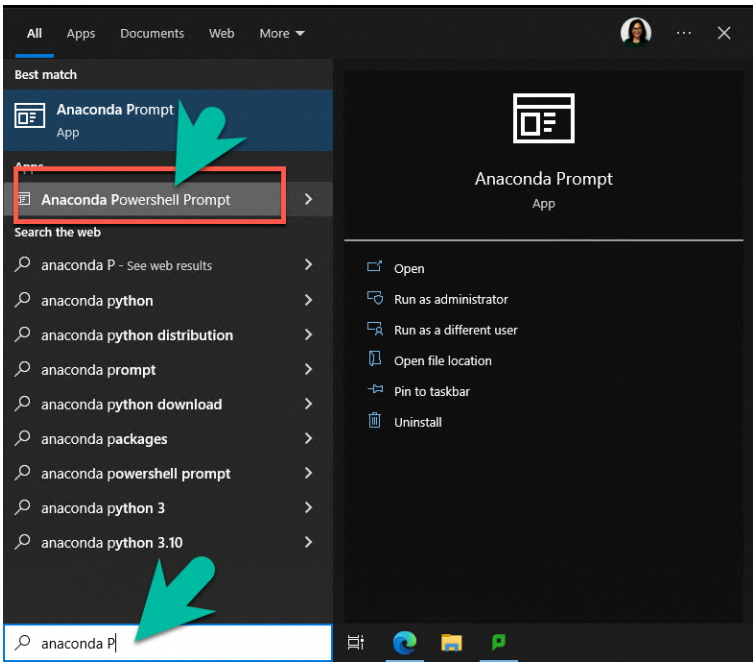


Figure 25a. Type Anaconda Powershell

1. On the **Anaconda Powershell Prompt** navigate to your U: drive (Figure 26), and to your `js-async-fetch` directory (Figure 26a)

   Command line navigation:
   - Change to U drive – type `U:`
   - Change directory – type `cd` then follows by the name of the directory you want to go to

Figure 26. Navigate to U: drive

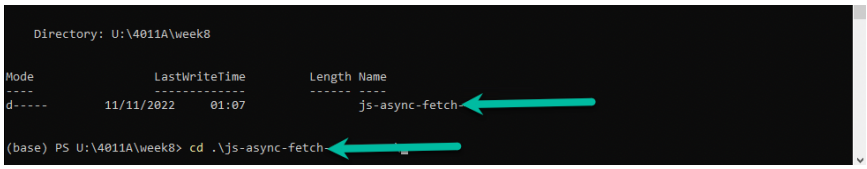


Figure 26a. Navigate to my folder on U: drive

2. Type this command: `python -m http.server` (Figure 27), and hit enter. **Note:** You must use Anaconda Powershell Prompt if you are using a lab machine and must navigate to your `js-async-fetch` directory or otherwise it won't work.

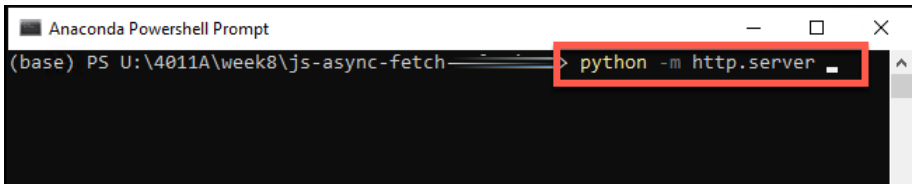


Figure 27. Run a local web server

3. Observe the output on this window (Figure 28). The web server is running on port 8000

Figure 28. The Python server is running on port 8000

4. Now, on Chrome, type this URL: http://localhost:8000/hello.html

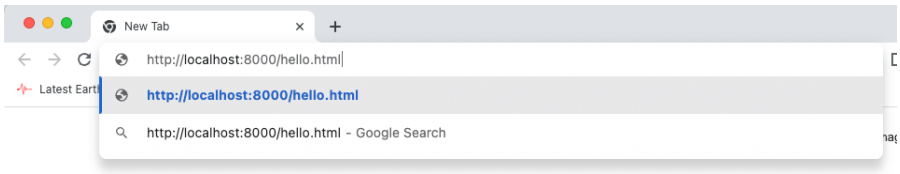


Figure 29. The output of hello.html

5. Verify the output on the console tab and make sure there are no more errors (Figure 29a). We will continue to write code and process the response.

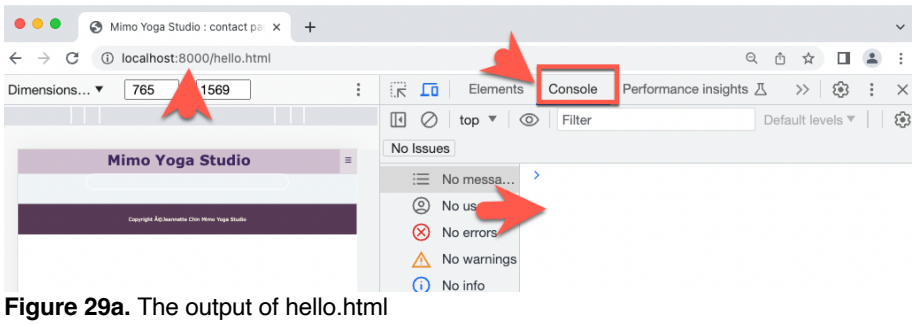


Figure 29a. The output of hello.html

l) back to `fetch.js`, we will write the `onResponse` handler function. But first, let's print the response on the console Figure 30, line 4

```
JS fetch.js
JS fetch.js > onResponse
1
2   /// this function will call if fetch is successful
3   function onResponse(response) {
4       console.log(response);
5   }
6
```

Figure 30. Print the response on the console

m) Refresh the page and inspect the console tab for this response object (Figure 31)

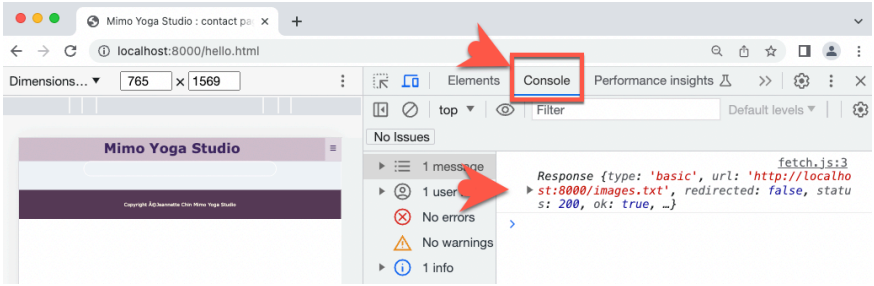


Figure 31. Observe the response object from the console on Chrome

n) Now drill down the response object and look for the "text", as shown in Figure 32. The returning data (from fetch API) will be in this block.

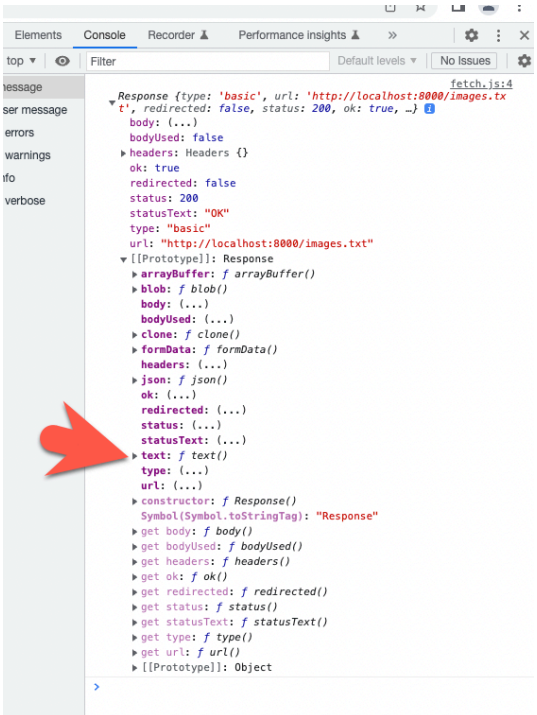


Figure 32. Inspect the response object on Chrome

o) Further drill down the “text” you may see that the length is 0 (Figure 33). Remember fetch is asynchronous, that means the data has not been returned yet. We will handle this data next.

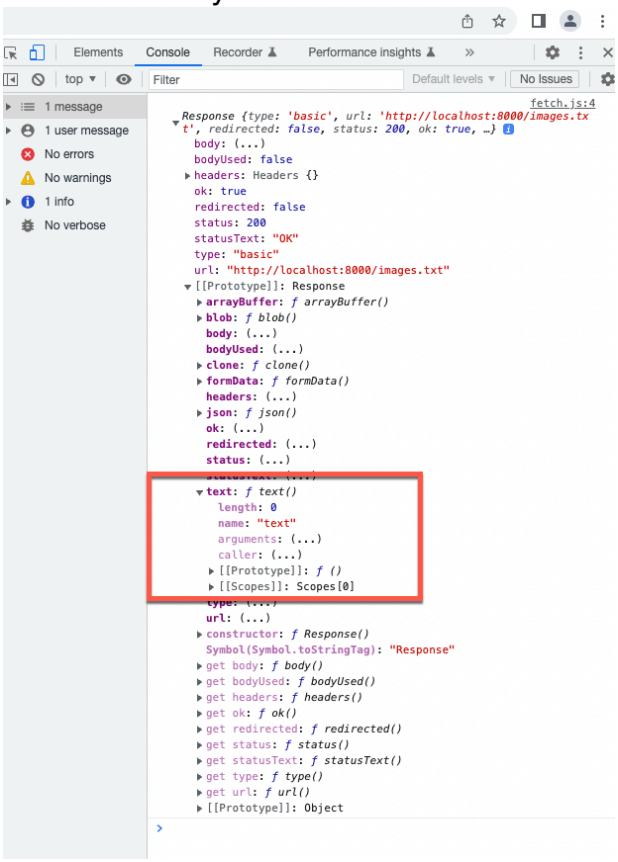


Figure 33. Inspect the response object on Chrome

p) The `.text()` returns a Promise too. We can use the `.then()` method to handle the response, as shown in Figure 34

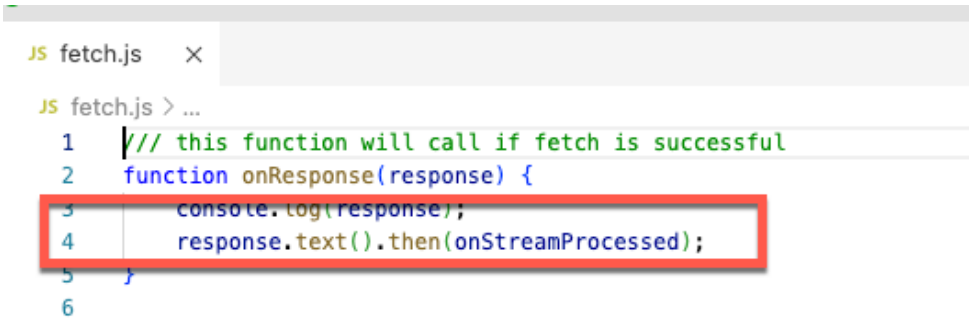


Figure 34. Handling Promise on fetch.js

Note. I pass in a **callback** function called “onSteamProcessed” – line 4. This means when the data is returned from `text()`, this callback function will be called.

q) We will write this **callback** function `onStreamProcessed` (line 3), which takes the text returned as parameter, and print the text out in the console (line 4) – Figure 35.

```
JS fetch.js
JS fetch.js > ...
1
2   //// callback function
3   function onStreamProcessed(text){
4       console.log(text);
5   }
6
7
8   /// this function will call if fetch is successful
9   function onResponse(response) {
10      console.log(response);
11      response.text().then(onStreamProcessed);
12  }
13
```

Figure 35. Code in fetch.js

r) Save the file and refresh the page.

s) Observe the output on Console Tab. You will see the URLs of the images there – Figure 36

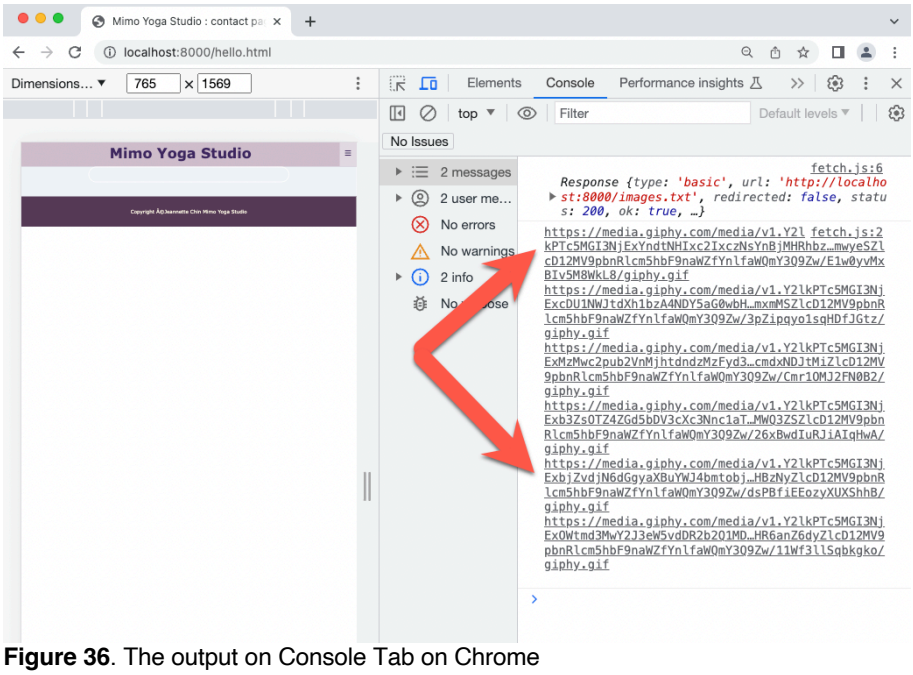


Figure 36. The output on Console Tab on Chrome

Question: where did these URLs come from?

t) Next, back to `fetch.js`, we will process this URLs. For each URL, we will create an image and append the image on the page, as shown on Figure 37.

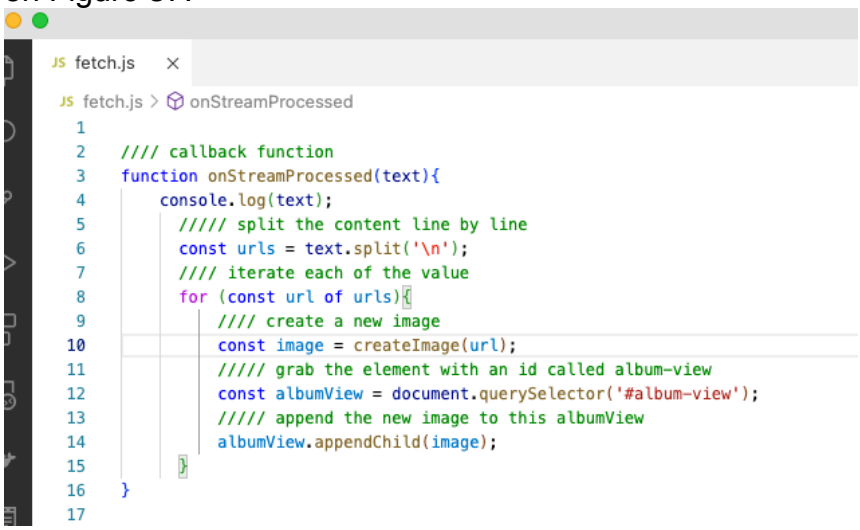


Figure 37. The code in fetch.js

The code is self-explanatory. We split the text to line by line, then iterate each line using a `for` loop. And for each line we create a new image, and append the new image to the element with an ID called `album-view`. Notice that I call a function named `createImage` to create a new image (line 10). We will write this function next.

u) Create a new function called `createImage` which takes a string (which is the URL) `src` as parameter. The function will create a new image using the `src` as URL source, and return the image – Figure 38

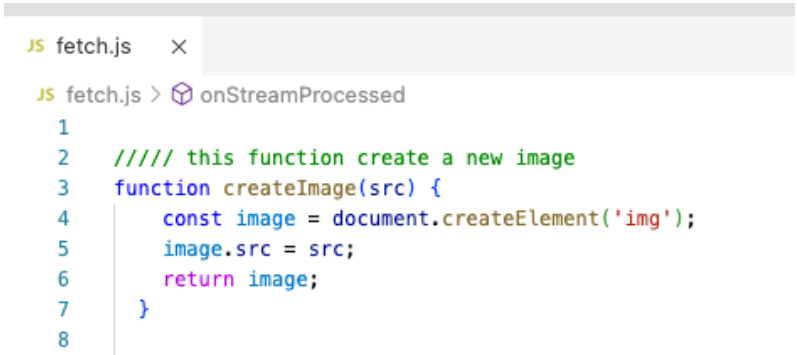


Figure 38. The code on fetch.js

v) Now save the file and refresh the page.

w) Figures 39 and 40 below are my outputs for you to compare.

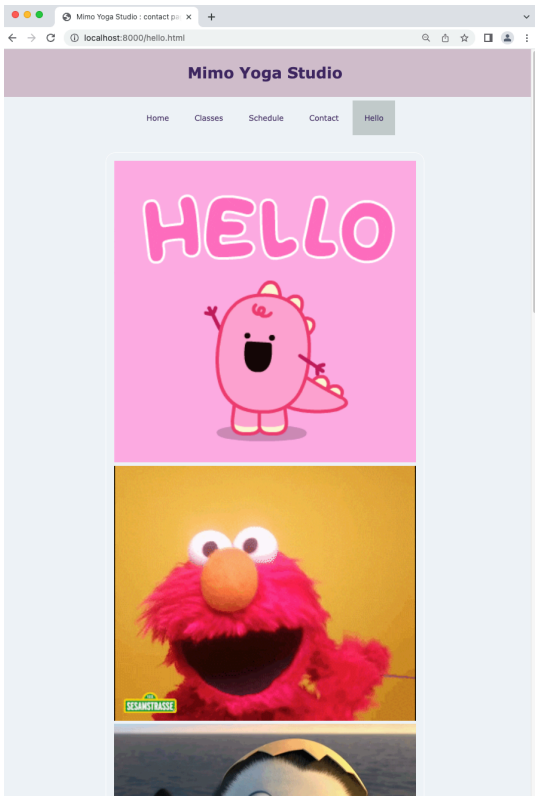


Figure 39. The output of hello.html

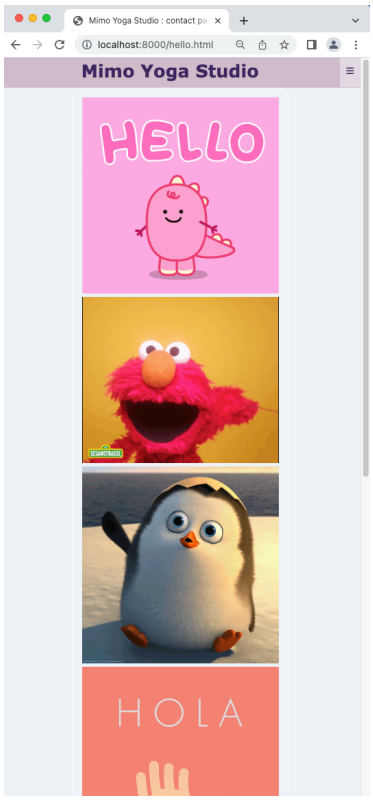


Figure 40. The output of hello.html on screen size smaller than 768px

x) Now, in `yoga.css`, create a CSS rule to style these images so that they have a width 33% and height 150px. Your result should look similar to Figure 41.

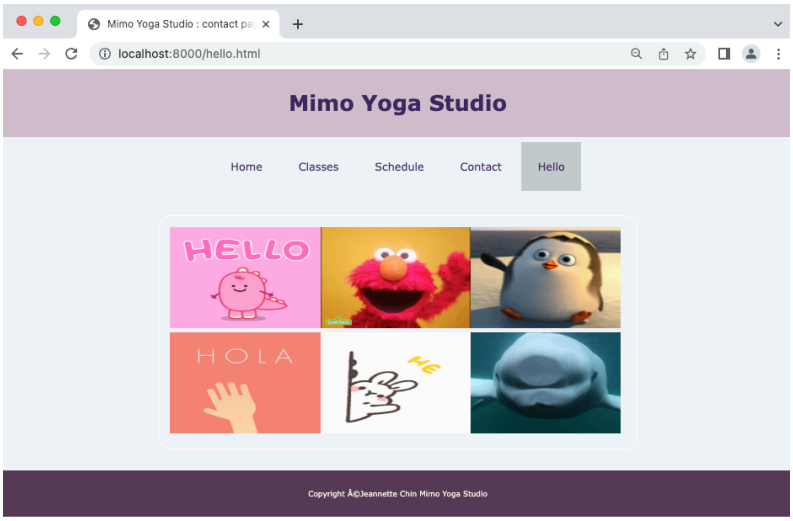


Figure 41. The output of hello.html

y) Next, write a click event handler function such that when a click occurs on any of the images, the same image will appear inside an element with an ID called "`modal-view`" – as shown in Figure 42

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>Mimo Yoga Studio : contact page </title>
 <link rel="stylesheet" href="yoga.css">
  <script src="https://kit.fontawesome.com/67bd86ffdc.js" crossorigin="anonymous"></script>
  <script src="hamburger.js" defer ></script>
  <script src="fetch.js" defer ></script>
</head>
<body>
<div id="wrapper"> <!-- to style for smaller screen -->
<header>
    <h1>Mimo Yoga Studio</h1>
    <a id="hamburger">
        <i class="fa-solid fa-bars"></i>
    </a>
</header>
<nav id="mainNav" >
    <ul class="mainNav">
        <li class="nav"><a href="index.html">Home</a></li>
        <li class="nav"><a href="classes.html">Classes</a></li>
        <li class="nav"><a href="schedule.html">Schedule</a></li>
        <li class="nav"><a href="contact.html">Contact</a></li>
        <li class="nav active"><a href="hello.html">Hello</a></li>
    </ul>
</nav>
<main>
    <div id="album-view"></div>
    <div id="modal-view"></div>
</main>
<footer>
    <p>Copyright ©2022 Mimo Yoga Studio <a href="mailto:jchin@abd.com">Jeannette Chin</a>

</footer>
</div>
</body>
</html>
```

Figure 42. The code in hello.html

z) Your output should be similar to Figure 43 below when the first image is clicked, and similar to Figure 44 when the second image is clicked.

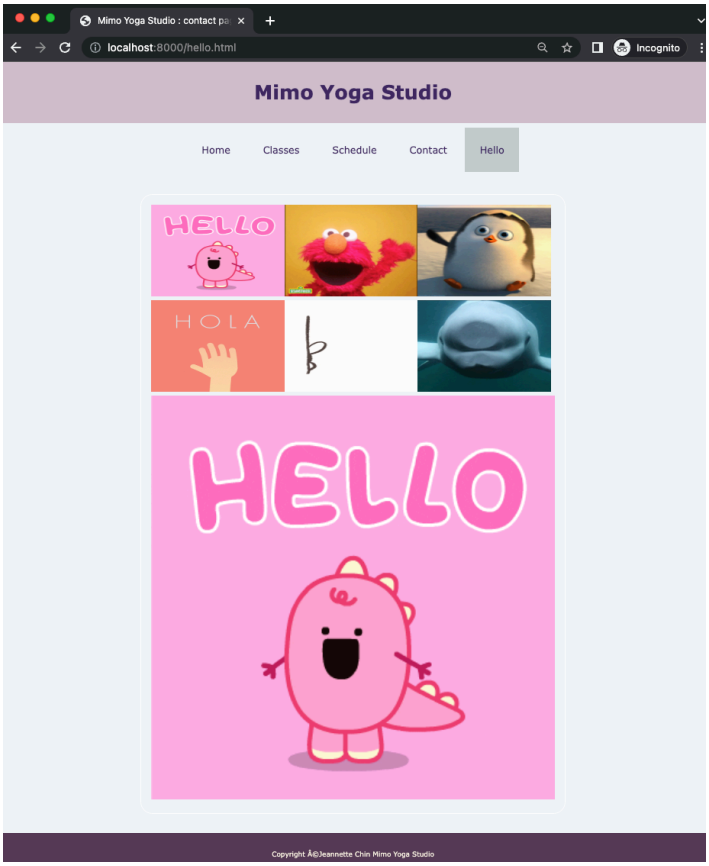


Figure 43. The output of hello.html when the first image is clicked

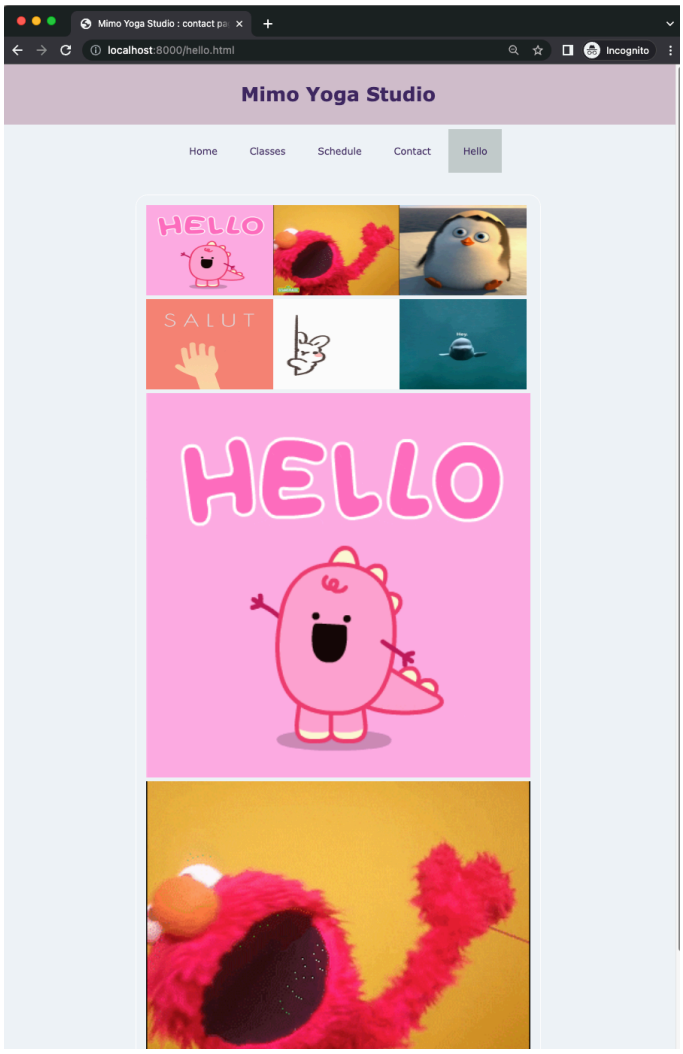


Figure 44. The output of hello.html when the second image is clicked

aa) Once you have got the event handler working, the next task is to style the output images such that they are the same size as the originals but with a `border` 5px solid and colour `gold #c59d5f` as shown in Figure 45. You may resize the image width so that three images can be rendered in one row.

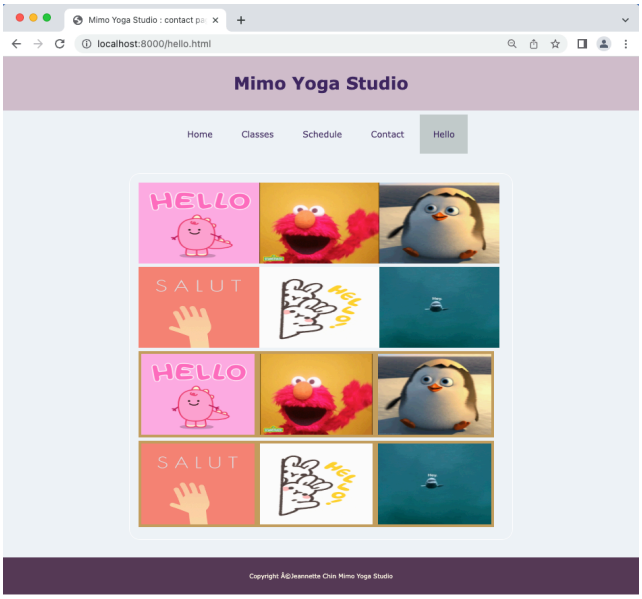


Figure 45. The output of hello.html. The images with a border are the results of click events.

bb) Fix the small screen view as well so that the output is the same as in desktop view [Hint: using media query], as shown in Figure 46

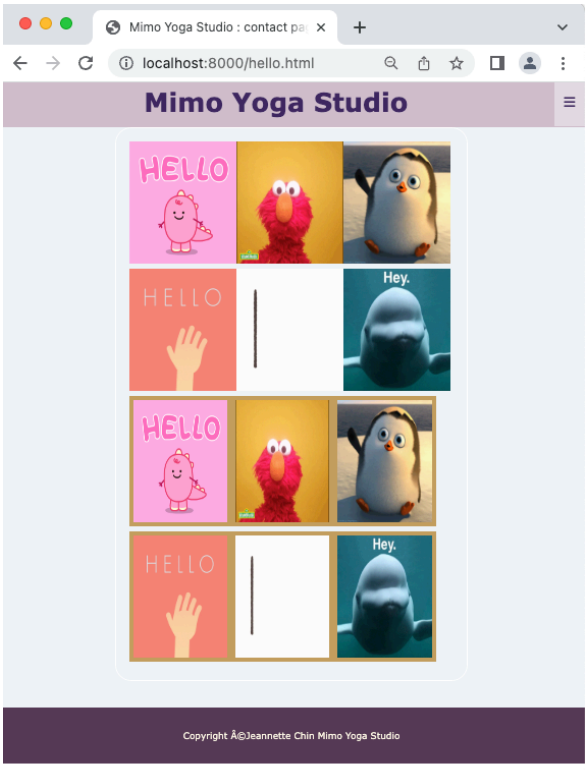


Figure 46. The output of hello.html on a screen size smaller than 768px

Well done for completing this lab!