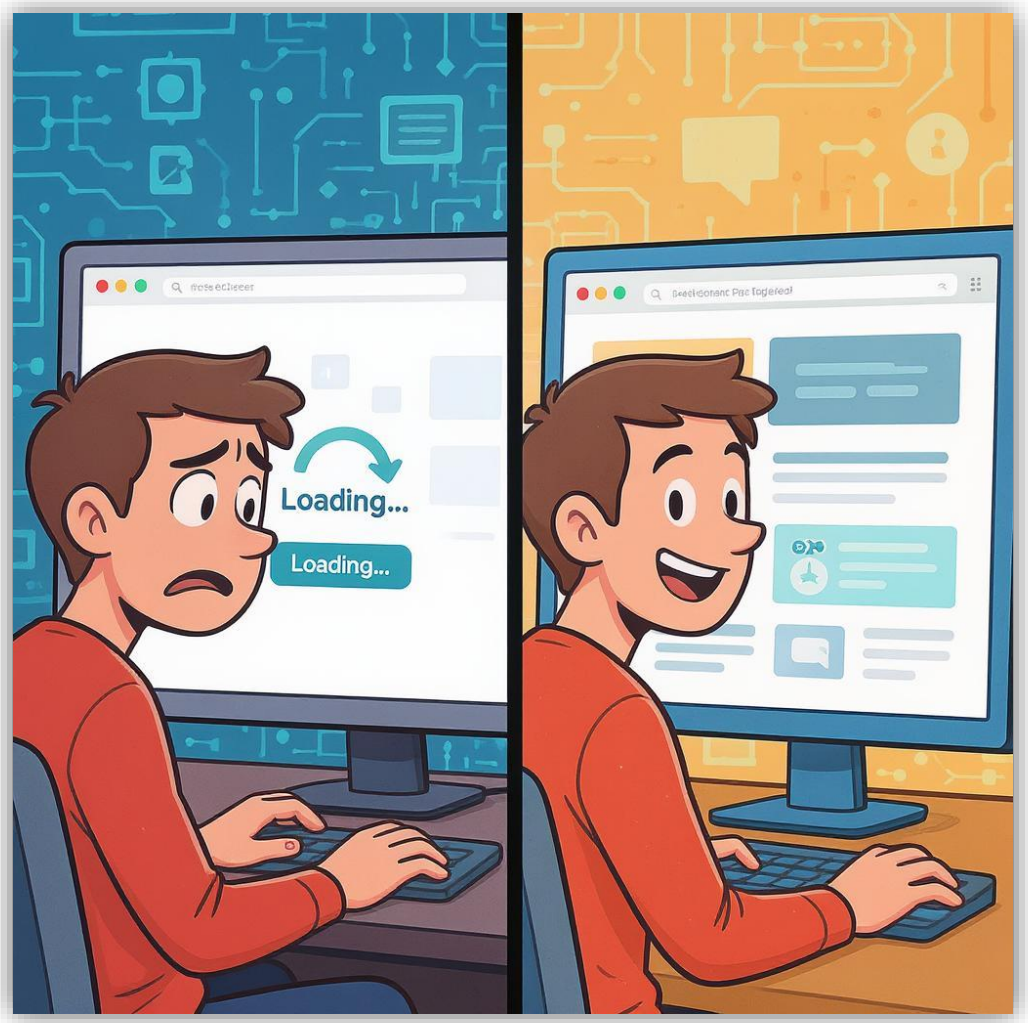
Chapter 7

AJAX

What is in This Chapter ?

This chapter explains a **JavaScript** programming technique called **AJAX** which allows us to send requests to servers and get back responses. The response data can then be used to update portions of a webpage rather than the entire page, thereby speeding things up and giving the user a more interactive feel. We will also discus promises and using the **fetch()**, **.then()**, **.catch()** and **asnyc**/**await** that is common in more modern web programming. At the end of the chapter, we work through a couple of **examples** to show how **AJAX** can be beneficial and then describe a few limitations.



7.1 AJAX Basics

AJAX (Asynchronous JavaScript and XML) is a powerful approach that lets websites communicate with a server behind the scenes, without reloading the entire page. This results in smoother, faster, and more interactive user experiences, such as live search suggestions, updating content in real time, or submitting forms without a full page refresh. Learning **AJAX** helps us build modern, responsive web applications that feel more like apps than static websites. It's a key tool in every web developer's toolkit, especially when working with **API**s, dynamic data, and single-page apps.

To be clear, **AJAX** is not software, nor is it a framework. It is simply a programming method for client-server communication in **JavaScript**, allowing us to send requests and receive responses from within our code. By using the **AJAX** method of programming, we can …

- send requests from our code and get responses back as (plain text, **XML**, **JSON** objects, etc..)
- update just portions of our page, instead of the whole thing

In the 1990s, whenever a website needed new data, it had to make a full request to the server, and the entire page would reload, even if only a small part of the content changed. Here is a bit of history on this, for those of us who are interested:

Microsoft was the first to implement a technique for loading content behind the scenes, by introducing an **ActiveX** control called **XMLHTTP** in Internet Explorer.

This concept was later adopted by other browsers in the form of the **XMLHttpRequest** object, allowing **JavaScript** to send and receive data from a server without reloading the page.

Google popularized this approach in the early to mid-2000s with apps like **Gmail**, **Google Maps**, and **Search Suggest**, helping to define what we now call **AJAX**.

XMLHttpRequest has since become part of the **W3C standard** for browser **API**s, though it has largely been replaced in modern development by the newer `fetch()` **API**.

Originally, **XML** was the common format for exchanging data between the client and server. Now, most modern **AJAX** implementations use **JSON** instead of **XML** since it is lightweight, cleaner and more readable, whereas **XML** is tag-heavy and has a more complex structure. Compare the two formats:

XML	JSON

`<user>`
`  <name>Jane</name>`
`  <age>30</age>`
`</user>`
	`{`
	`  "name": "Jane",`
	`  "age": 30`
	`}`

The term ***Asynchronous*** is used in **AJAX**. This basically means: "This task will start now, but it won't block or pause the rest of the program while it finishes." Perhaps a more accurate word would be ***non-blocking***. This is beneficial because it does not lock up the browser during communications.

Also, the browser updates quicker because it just fetches the data that it needs and when it receives the response from the server … instead of reloading the whole page it updates only the relevant section. Let's compare with simple **HTML** forms…

When using regular **HTML**, we might create a form like this:

```
<form id="searchForm" action="http://someSite.com/" method="GET">
    <input type="text" name="query" placeholder="Enter search term" />
    <button type="submit">Submit</button>
</form>

<div id="result"></div>
```

When the user presses the **Submit** button, a **GET** request is sent to the server along with a query value:

http://someSite.com/?query=entered-value

The browser waits (and is unresponsive) until the server responds with a new full **HTML** page, and then it replaces the entire current page with the new one. The browser automatically re-renders everything … including layout, styles, and scripts … as if we loaded a brand new page. It is similar with a **POST**, but the form data in that case would be in the body of the request.

Now, by using **AJAX** programming, we can write some **JavaScript** code to send the query and get a response. Then we can update just a part of the page by taking the response and placing it withing the result ** **. Here is what the **AJAX** code would look like:

```
document.getElementById('searchForm').addEventListener('submit', function(e) {
    e.preventDefault();             // Prevent the default form submission

    let query = this.query.value;   // Get the input value
    let url = `http://someSite.com/?query=${encodeURIComponent(query)}`;

    let xhr = new XMLHttpRequest(); // xhr is short for XML HTTP Request 😊
    xhr.open('GET', url, false);    // false = synchronous

    try {
        xhr.send();    // This will block the browser until the response is received

        if (xhr.status === 200) {
            // Display result within the <div> </div> tags
            document.getElementById('result').innerText = xhr.responseText;
        } else {
            document.getElementById('result').innerText = `Error: ${xhr.status}`;
        }
    } catch (err) {
        document.getElementById('result').innerText = 'Request failed';
    }
});
```

The first thing that we do in our code is to disable the default behavior for **HTML** form submission by using **`e.preventDefault()`**. Otherwise, the form gets sent as usual with a **GET** or **POST** and the browser reloads the page. In that case, our **AJAX** code gets interrupted or ignored.

Notice that we create an **XMLHttpRequest()** object. The **.open()** method initializes the request with the type (i.e., **GET**), the **URL**, and then whether we want the communication to be asynchronous or not. For this example, we chose synchronous by using **false**.

The **.send()** method sends the request to the server. The code waits until it gets a response, then it checks the response status. If it is **200**, then all is ok and we update the inner text between the ** ** tags with the response text. Otherwise, we indicate the error code between the ** ** tags.

Why are we doing error-checking in two spots? Well, the **catch** block is only triggered for network-level errors, not for **HTTP** errors such as **404 Not Found** or **500 Internal Server Error** … which are valid **HTTP** responses. If the browser gets a response, it doesn't consider it a network-level error. A network-level error refers to a failure that occurs when the browser is unable to reach the server at all, before any **HTTP** response is received (e.g., no internet, **DNS** failure, server unreachable, etc..)

Is this code better than using the simple form … because it sure seems like a lot of extra coding? Well, this code allows the browser to manually insert the response into the **DOM** in between the ** ** tags. The browser only redraws the parts of the page that we changed. But with the regular **HTML** form we had to re-render a whole new page. So, this is better and faster.

However, the code above is *synchronous*. That is, the code blocks on the **send()** call and waits until the server comes back with a reply. The whole browser becomes unresponsive during that time.

A better way to do this is *asynchronously*. To do that, we just need to plug in an event handler that will be called once the result is ready. That way, our browser can continue doing other things while it waits and thus it remains responsive. Here is updated code that is asynchronous:

```
document.getElementById('searchForm').addEventListener('submit', function(e) {
    e.preventDefault();             // Prevent default form submission

    let query = this.query.value;   // Get the input value
    let url = `http://someSite.com/?query=${encodeURIComponent(query)}`;

    let xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);     // true = asynchronous

    xhr.onload = function () {      // callback function for when response has returned
        if (xhr.status === 200) {
            document.getElementById('result').innerText = xhr.responseText;
        } else {
            document.getElementById('result').innerText = `Error: ${xhr.status}`;
        }
    };

    xhr.onerror = function () {     // callback function for when an error occurred
        document.getElementById('result').innerText = 'Request failed';
    };

    xhr.send(); // Send the request (non-blocking)
});
```

The code is very similar, except that we indicate **true** to make it asynchronous and then we split up our handler code into two event handlers. The `onload` callback is called when the full response has been received from the server. The `onerror` callback is called if there was a network-error.

From the code itself, we may not notice the true benefits of asynchronous communication here because there is no code after the `.send()`. But we have to keep in mind that when this form is being submitted, the user can continue interacting with the page (e.g., typing new queries, clicking buttons, scrolling, etc..). Also, because the requests are asynchronous, the user can submit multiple searches in succession, and each result will be handled independently. We could even expand this code to allow cancelling of previous requests or handling results coming back out of order.

It can get even more interesting if there is a lot involved in getting the response from the server (e.g., the particular server is known to be slow at responding). We can do things intermittently at various stages of the requesting process. For example, we could insert some checks for each of the 4 stages involved by checking the `readyState` property of the request object as follows:

```
xhr.onreadystatechange = function () {
    if (xhr.readyState === XMLHttpRequest.OPENED) {
        // Request has been opened, server connection established, but not sent yet
    }
    else if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
        // Response headers received
    }
    else if (xhr.readyState === XMLHttpRequest.LOADING) {
        // Loading / Receiving response
        document.getElementById('result').innerText = 'Loading...'; // show loading indicator
    }
    else if (xhr.readyState === XMLHttpRequest.DONE) {
        // Request finished , response is ready
        if (xhr.status === 200) {
            document.getElementById('result').innerText = xhr.responseText;
        } else {
            document.getElementById('result').innerText = `Error: ${xhr.status}`;
        }
    }
};
```

Notice … instead of setting the callback for `xhr.onload`, we set it for `xhr.onreadystatechange`. That is because `.onload` only gets called once after the response is returned, where as `.onreadystatechange` gets called at each stage of the process. However, for most uses, `.onload` is sufficient.

There are some more useful methods that we can use on **XMLHttpRequest** objects.

- `abort()` - stops the request if it's still in progress. No response will be delivered, and any event handlers such as `.onload` won't get triggered. This is useful to cancel a slow request or if the user changes his mind and wants to send a different request instead (e.g., different search query).

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'http://someUrl', true);
xhr.send();

// Cancel the request after 2 seconds, unless it is done
setTimeout(() => {
    if (xhr.readyState !== XMLHttpRequest.DONE) {
        xhr.abort();
        console.log('Request aborted');
    }
}, 2000);
```

- `getAllResponseHeaders()` - returns all response headers (after the response is received) in a single string, formatted like **HTTP** headers. This is useful for debugging.

```
xhr.onload = function () {
    const headers = xhr.getAllResponseHeaders();
    console.log(headers);
};
```

- `getResponseHeader(name)` - gets the value of a specific response header (once the response has been received). This is useful for checking things like content type, custom headers or rate limit info (i.e., check to avoid sending too many request, which can cause the server to block us).

```
xhr.onload = function () {
    const contentType = xhr.getResponseHeader('Content-Type');
    console.log('Response is of type:', contentType);
};
```

- `setRequestHeader()` - sets custom **HTTP** headers before sending the request. This must be called after `.open()` but before `.send()`. This is handy for adding authentication tokens, telling the server we are sending **JSON** or setting custom headers for **API**s.

```
xhr.open('POST', '/submit', true);
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.setRequestHeader('Authorization', 'Bearer USER123_API_KEY');
xhr.send(JSON.stringify({ name: 'Mark' }));
```

7.2 Promises

To re-cap what we have discussed so far, when we are writing code that talks to a server, we are usually requesting something that might take some time to get a reply. We set up our code to be *asynchronous* so that our program keeps running while waiting for that response. To do this, we needed a way to say: *"when the reply comes back, here's what I want you to do."*.

Traditionally, we used **callbacks** for this (e.g., `.onload` and `.onerror`). The downside is that chaining multiple callbacks can quickly lead to messy, hard-to-read code. Modern **JavaScript** solves this problem with something called ***Promises***.

> ***A promise is a JavaScript object representing a value that will be available in the future, either as a successful result or as an error, allowing our code to react accordingly.***

A promise is a bit like a receipt. For example, imagine we order and pay for a pizza and get a receipt right away. The receipt doesn't give us the pizza immediately, but it guarantees that we will get it eventually.

In **JavaScript**, a promise lets us attach instructions for what to do when the value is ready, or what to do if something goes wrong, keeping our asynchronous code much cleaner and easier to follow.

This leads us to the `fetch()` function, which is a more modern way to make requests to a server in the browser. When we call `fetch()`, it is like ordering and paying for the pizza … it returns a *promise* (i.e., receipt) representing the eventual *response* (i.e., pizza) from the server.

Now, think about the steps in making a pizza and picture them as a series of functions: **getDough()**, **makePizza()**, **bakePizza()**, **packagePizza()** … and then perhaps a **dealWithIssues()** function for handling any problems that arise. Each part of the process returns something that is passed onto the next part of the process:

getDough() → dough
makePizza(dough) → rawPizza
bakePizza(rawPizza) → cookedPizza
packagePizza(cookedPizza) → boxedPizza
servePizza(boxedPizza) → pizza served to customer

In **JAVA**, we could chain such function calls together with the dot operator like this:

```
pizza = getDough().makePizza().bakePizza().packagePizza();
```

Well, when we use the `fetch()` function, we do something similar in that we attach (after the `fetch()` call) a series of `.then()` handlers followed by a `.catch()` handler to process the response or handle errors. Each `.then()` returns a value (i.e., a **promise** … in case it takes a while) that is passed to the next one in the sequence … like what is shown on the left below:

Flow of fetch()/then()/catch()…

```
fetch(url)
    .then(response =>  {
        ...
        return ;
    })

    .then( =>  {
        ...
        return ;
    })

    .then( =>  {
        ...
        return ;
    })

    .then( =>  {
        ...
    })
    .catch(err => { // Handle errors
        console.error("Error: ", err);
    });
```

In this code, each `.then()` receives the result of the previous step as its parameter. The `return` statement passes the value to the next `.then()` in the chain. The `.catch()` at the end catches errors from any previous step.

Using Our Pizza Analogy…

```
fetch('orderPizza')  // order and pay for the pizza
   .then(function(order) {
       // order confirmed, get the dough for the order
       return getDough();
   })
   .then(function(dough) {
       // make the raw pizza from the dough
       return makePizza(dough);
   })
   .then(function(rawPizza) {
       // bake the raw pizza
       return bakePizza(rawPizza);
   })
   .then(function(cookedPizza) {
       // package the cooked pizza
       return packagePizza(cookedPizza);
   })
   .then(function(boxedPizza) {
       // serve the boxed pizza
       return servePizza(boxedPizza);
   })
   .catch(function(issue) {
       // handle any issues that arise
       return dealWithIssues(issue);
   });
```

Alternatively, we could insert the code right into each `.then()` statement instead of making separate functions, and we can also use the => function notation:

```
fetch('orderPizza')  // order and pay for the pizza
   .then(order => {
       let dough = ...;         // code to get the dough depending on the order
       return dough;
   })
   .then(dough => {
       let rawPizza = ...;      // code to make the pizza using dough variable
       return rawPizza;
   })
   .then(rawPizza => {
       let cookedPizza = ...;   // code to bake it using rawPizza variable
       return cookedPizza;
   })
   .then(cookedPizza => {
       let boxedPizza = ...;    // code to box it using cookedPizza variable
       return boxedPizza;
   })
   .then(boxedPizza => {
       let servedPizza = ...;   // code to serve it using boxedPizza variable
       return servedPizza;
   })
   .catch(issue => {
       let problemSolved = ...; // code to handle errors using incoming issue information
       return problemSolved;
   });
```

Think of the `.then()` chain like passing the pizza along a conveyor belt with worker standing at a *station* to handle each part of the process. Every `.then()` returns a **promise**. Even if we return a plain value, **JavaScript** wraps it in a promise automatically. So, the chain itself is always dealing with promises, not raw values. When a `.then()` returns a regular value, the next `.then()` runs immediately with that value. When a `.then()` returns a delayed promise (e.g., baking pizza), the next `.then()` waits for that promise to resolve.

But this "waiting" doesn't block the entire program because **JavaScript** uses an event loop (discussed in the next chapter) to schedule the next `.then()` for when the promise resolves. So, if one station is still baking (i.e., promise is pending), the next station doesn't move until the pizza is done. Meanwhile, the kitchen can handle other pizzas because nothing else is blocked. Once the baking is done, the pizza moves to the next station automatically.

How does this work with **HTTP** Requests and **HTTP** responses? Well, typically, we would chain at least three things together with a `fetch()`:

1. A first `.then()` to handle the raw response object from the server.
2. A second `.then()` to handle the actual data extracted from the response.
3. A single `.catch()` at the end to handle any errors that occur during the request or in any of the `.then()` handlers.

Here is a typical usage template:

```
fetch(url)

    .then(response => {
        if (!response.ok) {  // response object is the HTTP response returned by the server
            // Throw an error to be caught in .catch() below
            throw new Error("HTTP error! " + response.status);
        }
        ...
        // Parse the response body (as JSON, text, or other formats)
        return response.json(); // or response.text() for plain text or html
    })

    .then(data => {
        console.log(data);   // Do something with the data
    })

    .catch(err => {
        console.error('Fetch error:', err);  // Handle errors
    });
```

We can ask the response for certain properties like:

- `response.status` - the status code
- `response.ok` - true if the status is between 200–299

And we can also extract the response's *body* (i.e. the *data*) by using an appropriate function:

- `response.text()` - get the body as plain text or **HTML**
- `response.json()` - parse the body into a **JSON** object
- `response.blob()` - for binary data such as images or files

Let's compare the promise approach with that of the **XMLHttpRequest** approach:

Using **XMLHttpRequest**

```
document.getElementById('searchForm')
            .addEventListener('submit', function(e) {
  e.preventDefault();

  let query = this.query.value;
  let url =
`http://someResource?query=${encodeURIComponent(query
)}`;

  let xhr = new XMLHttpRequest();
  xhr.open('GET', url, true);

  xhr.onload = function () {
    if (xhr.status === 200) {
      document.getElementById('result').innerText =
                                    xhr.responseText;
    } else {
      document.getElementById('result').innerText =
                               `Error: ${xhr.status}`;
    }
  };

  xhr.onerror = function () {
    document.getElementById('result').innerText =
                                   'Request failed';
  };

  xhr.send();
});
```

Using promises with `fetch()`

```
document.getElementById('searchForm')
            .addEventListener('submit', function(e) {
  e.preventDefault();

  let query = this.query.value;
  let url =
`http://someResource?query=${encodeURIComponent(query
)}`;

  // Use fetch() instead of XMLHttpRequest
  fetch(url)

    .then(response => {
      if (!response.ok) {
        throw new Error(`Error:${response.status}`);
      }
      return response.text();
    })

    .then(data => {
      document.getElementById('result').innerText =
                                                 data;
    })

    .catch(error => {
      document.getElementById('result').innerText =
                                      'Request failed';
    });
});
```

Notice that with promises, we no longer need to make a request object, nor do we need the `.open()` / `send()` combination of function calls… and we do not need to define `.onload` and `.onerror` callback functions. We simply use `fetch()` and a `.then().then().catch()` series of calls. I hope you will agree that the code is simpler to read. Notice also that we can even throw an error from within a `.then()` function … which will get handled by the `.catch()` function.

As an option, we can pass a second parameter (i.e., an object) to `fetch()` that allows us to customize the **HTTP** request (e.g., to specify the method, various headers, the body (i.e., not for **GET**)). Here is an example:

```
// Options for our request
const requestOptions = {
   method: 'POST', // 'GET', 'PUT', 'DELETE', etc.
   headers: {
       'Content-Type': 'application/json',
       // add other headers as needed
   },
   body: JSON.stringify({ key: 'value' }) // only for POST/PUT
};

// Call fetch with the URL and the requestOptions variable
fetch(url, requestOptions)
   .then(response => {
       ...
   })
   .then(data => {
       ...
   })
   .catch(err => {
       ...
   });
```

The `.then().then().catch()` sequence works the same way.

We can clean things up by using **`async`** and **`await`** so that our code will look more like **synchronous** code, even though it's still **asynchronous**.

We can place **`async`** before a function name to indicate that the function will be **asynchronous** … meaning it will always return a **promise**. Inside an **`async`** function, we can use **`await`** to pause execution until a promise resolves, effectively "waiting" for an internal asynchronous step to finish. This is especially useful when we need to perform multiple asynchronous steps **in sequence**, like following a recipe where each step depends on the previous one.

Here is a comparison of our pizza-fetching code alongside this new **`async`** / **`await`** approach:

Using `.then()` and `.catch()`

```
function makeAndServePizza() {
  fetch('orderPizza')

    .then(order => {
       // code to get the dough depending on order
       let dough = ...;
       return dough;
    })

    .then(dough => {
       // code to make pizza using dough variable
       let rawPizza = ...;
       return rawPizza;
    })

    .then(rawPizza => {
       // code to bake it using rawPizza variable
       let cookedPizza = ...;
       return cookedPizza;
```

Using **`async`** and **`await`**

```
async function makeAndServePizza() {
  try {
    const order = await fetch('orderPizza');

    // code to get the dough depending on order
    let dough = ...;



    // code to make the pizza using dough variable
    let rawPizza = ...;



    // code to bake it using rawPizza variable
    let cookedPizza = ...;
```

```
    })

    .then(cookedPizza => {
       // code to box it using cookedPizza variable
       let boxedPizza = ...;
       return boxedPizza;
    })
    .then(boxedPizza => {
       // code to serve it using boxedPizza variable
       let servedPizza = ...;
       return servedPizza;
    })
    .catch(issue => {
       // handle issues that arise
       let problemSolved = ...;
       return problemSolved;
    });
}
```

```
    // code to box it using cookedPizza variable
    let boxedPizza = ...;


    // code to serve it using boxedPizza variable
    let servedPizza = ...;
    return servedPizza; // final pizza is ready

  } catch (issue) {
     // handle issues that arise
     let problemSolved = ...;
     return problemSolved;
  }
}
```

Notice how simple the code appears! Here, by using **`await`**, we are saying that we are waiting for the request to be sent (and response to be returned) from the **`fetch()`** call. In reality, the **`fetch()`** call is non-blocking, so it returns right away with a promise … but the **`await`** tells the code to pause until that promise resolves (i.e., until the response has been returned). We can only use **`await`** because we specified that the **`makeAndServePizza()`** function call is asynchronous by using **`async`**.

This code is not quite realistic, however, since each of the individual stages take time to perform, especially the making the pizza and baking it! In the **`let cookedPizza = ...;`** code, the entire system would grind to a halt while the pizza is baking … which is very bad. Instead, we should have functions for performing each stage and then use **`await`** as well on each of these functions so that we do not delay the system from locking up. This is better code:

```
async function makeAndServePizza() {
    try {
        const order = await fetch('orderPizza');            // order and pay for the pizza
        const dough = await getDough();                     // get the dough
        const rawPizza = await makePizza(dough);            // make raw pizza
        const cookedPizza = await bakePizza(rawPizza);      // bake the pizza
        const boxedPizza = await packagePizza(cookedPizza); // box it
        const servedPizza = await servePizza(boxedPizza);   // serve it
        return servedPizza;                                 // final pizza is ready
    } catch (issue) {
        return dealWithIssues(issue);  // handle any issues that occur at any step
    }
}
```

Note that it is vital to use the **`await`** here. For example, if we didn't put it on the **`fetch()`** it would be like grabbing the receipt and trying to eat it!

If we forget to use **`await`**, we are working with the promise object before the response is ready. So, the response and its data are not what we expect. This will lead to logic errors and could even crash our code if we try to access a property or call a method on the unresolved promise that doesn't exist yet.

Let's now compare things with our previous form submission code:

Using `.then()` and `.catch()`

```
document.getElementById('searchForm')
     .addEventListener('submit', function(e) {

  e.preventDefault();

  let query = this.query.value;
  let url =
`http://someResource?query=${encodeURIComponent(query
)}`;


  // Use fetch() instead of XMLHttpRequest
  fetch(url)

    .then(function(response) {
       if (!response.ok) {
         throw new Error(`Error:${response.status}`);
       }
       return response.text();
    })

    .then(function(data) {
       document.getElementById('result').innerText =
                                                 data;
    })

    .catch(function(error) {
       document.getElementById('result').innerText =
                                    'Request failed';
    });
});
```

Using `async` and `await`

```
document.getElementById('searchForm')
      .addEventListener('submit', async function(e) {

  e.preventDefault();

  let query = this.query.value;
  let url =
`http://someResource?query=${encodeURIComponent(query
)}`;

  try {
     // Wait for the fetch promise to resolve
     const response = await fetch(url);


     if (!response.ok) {
        throw new Error(`Error:${response.status}`);
     }
     const data = await response.text();



     document.getElementById('result').innerText =
                                          data;


  } catch (error) {
       document.getElementById('result').innerText =
                                       'Request failed';
  }
});
```

As we can see, the code looks a lot cleaner.

We will halt our discussion of promises for now, but we will pick it up again when we discus the usage of databases in a later chapter.

7.3 Trivia Database Example

Let's look at an example now. There is a website called **opentdb.com** that contains a database of over **4000** trivia questions. The website is a server that will give us (the client browser) some trivia questions if we ask it to.

If we click on the **api** menu item at the top of the page, it lets us pick how many questions we want, a category, a difficulty level and whether we want multiple choice or true/false types. Lastly, we can ask the encoding format. In our example, we will ask for **10** multiple-choice questions from any category at any difficulty level and we will use the default encoding. Clicking the Generate **API** URL button will produce the following **URL** that we will use in our code:
https://opentdb.com/api.php?amount=10&type=multiple

The result that we get back is a **JSON** string that looks like this (only two questions are shown):

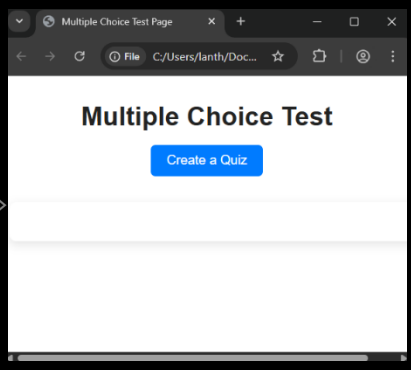
```
{
  "response_code": 0,
  "results": [
    {
      "type": "multiple",
      "difficulty": "easy",
      "category": "Entertainment: Music",
      "question": "Who is the lead singer of Arctic Monkeys?",
      "correct_answer": "Alex Turner",
      "incorrect_answers": [
        "Jamie Cook",
        "Matt Helders",
        "Nick O&#039;Malley"
      ]
    },
    {
      "type": "multiple",
      "difficulty": "easy",
      "category": "Geography",
      "question": "What is the name of the peninsula containing Spain and Portugal?",
      "correct_answer": "Iberian Peninsula",
      "incorrect_answers": [
        "European Peninsula",
        "Peloponnesian Peninsula",
        "Scandinavian Peninsula"
      ]
    },
    ...
  ]
}
```

So, in our code, we will need to take this **JSON** string and convert it to a **JavaScript** object so that we can work with it. Each object will be formatted like this where we can get the key and value for each part of the object:

```
category: "Entertainment: Music"
correct_answer: "Alex Turner"
difficulty: "easy"
incorrect_answers: Array(3)
        0: "Jamie Cook"
        1: "Matt Helders"
        2: "Nick O&#039;Malley"
length: 3
question: " Who is the lead singer of Arctic Monkeys?"
type: "multiple"
```

We have created a simple **trivia.html** page (shown below) that looks as shown in the image:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charse  t="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Multiple Choice Test Page</title>
    <link href="trivia.css" rel="stylesheet">
</head>

<body onload="init()">
    <div id="header-container">
        <h1>Multiple Choice Test</h1>
        <button type="button" id="retrieveTests">Create a Quiz</button>
    </div>

    <div id="questiondiv">
        <!-- Question content will be loaded here -->
    </div>

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



There is a **triva.css** stylesheet associated with this example that allows for nice styling and coloring. Notice that there is a `<div id="questiondiv">` tag that is empty. This is where we will insert our questions after we retrieve them from the trivia server. Also notice that there is an `onload="init()"` in the `<body>` tag and we are including a **trivia.js** JavaScript file at the end of the body. That is where all the work goes, so let's look into it. Here is the start of our code:

```
let questions = [];  // Array of questions

// This will get called after the page loads
function init(){
    document.getElementById("retrieveTests").addEventListener('click', loadQuestions);
}
```

The questions array will get populated from the server response, eventually. The `init()` function gets called as soon as the original page content is loaded … that is … after all **HTML** content, images, scripts, and styles inside the body have been fully loaded by the browser. All we are doing in the function is adding an event handler (called `loadQuestions()` to the **"Create a Quiz"** button).

Now, we could have just included that **addEventListener** line of code immediately below the declaration of the **questions** array, but it is safer practice to wait until the entire page has loaded. Why? Because if our script runs BEFORE the **HTML** elements it references are loaded, then our code will return **null** because that element doesn't exist yet in the **DOM** … so the event listener won't attach.

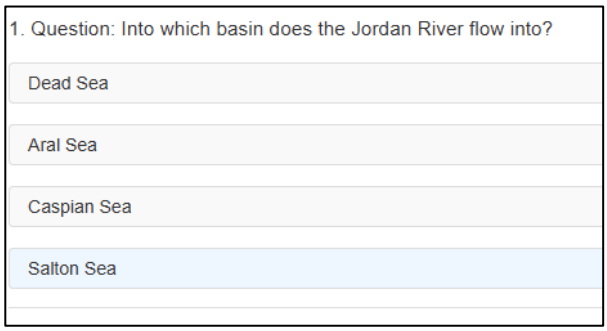
Now what happens when the button is clicked? We need to request the questions from the server and then place (i.e., render) them onto our **HTML** page within the `<div id="questiondiv">` tag.

Let's create an `XMLHttpRequest()` object and handle the `.onload` event when the response comes back with a **200 OK** result. If it comes back with anything else, or if there is a network error, we insert some simple text such as `"Error: 404 Not Found"` or `"Question Database Not Available"`.

If all was ok, then we take the `responseText` (i.e., the returned **JSON** string of questions) and parse it to obtain a **JavaScript** object called `responseObject`. We can then take the results of that (i.e., an array) and store it in our `questions` variable. Then we will insert the questions onto the page with a call to a `render()` function:

```
// This will get the questions from the server
function loadQuestions(){
    let xhr = new XMLHttpRequest();

    // This is only going to get called when the response comes back
    xhr.onload = function() {
        // If the response was successful
        if (xhr.status == 200) {
            // Take the response text (that is in JSON format), parse it into a JS object
            let responseObject = JSON.parse(xhr.responseText);
            // Extract questions from results and update our array
            questions = responseObject.results;
            // Update our page
            render();
        }
        else { // If the response was not successful, show an error
            document.getElementById(questiondiv).innerText = `Error: ${xhr.status}`;
        }

    };

    // Callback function for when a network error occurred
    xhr.onerror = function () {
        document.getElementById('questiondiv').innerText = 'Question Database Not Available';
    };

    //Create and send the request
    xhr.open("GET", "https://opentdb.com/api.php?amount=10&type=multiple", true);
    xhr.send();
}
```

The `render()` function will need to create the **HTML** content to place onto our webpage within the `<div id="questiondiv">` tag. We will go through each question and then show them (numbered from 1-10) and then create a group of 4 answers and put them inside a `<div class="question-block">` tag. Each answer will simply be a label. It will look as shown here:



Here is the code:

```
function render(){
  let content = "";
  let count = 1;

  questions.forEach((question, index) => {
    content += `
      <div class="question-block">
        <p> ${count}. Question: ${question.question}</p>
        <label class="answer" data-correct="true">${question.correct_answer}</label><br>
        <label class="answer" data-correct="false">${question.incorrect_answers[0]}</label><br>
        <label class="answer" data-correct="false">${question.incorrect_answers[1]}</label><br>
        <label class="answer" data-correct="false">${question.incorrect_answers[2]}</label>
      </div>`
    count++;
  });
  document.getElementById("questiondiv").innerHTML = content;
  // … more code is coming soon …
}
```

In the above code, we set `class="answer"` for each answer and set `data-correct` to either `"false"` (for the wrong answers) or `"true"` (for the correct answer). Recall that we can set some `data-` attributes for our **HTML** elements by using `data-*`, where we put whatever attribute name we want in place of the `*`. It allows us to set attributes that we can access later by using `.dataset`. In this case, we can ask the **HTML** element `.dataset.correct` and we will get back either `"false"` or `"true"`. We will see this soon.

Note: The code always places the correct answer first, but we would likely want to place it randomly. The last line is where we actually place the content onto the page.

To complete things, we will now add listeners to each of the labels that will allow us to select one from each question. Correct ones will be highlighted green, incorrect ones will be highlighted red (this is set in our CSS stylesheet as classes `.correct` and `.wrong`). We will also prevent selecting a different answer for each question once the user already selected one.

Here is the template for what we want to do. Place this inside the `render()` function at the bottom:

```
// Add event listeners after rendering
document.querySelectorAll(".answer").forEach(answer => {
  answer.addEventListener("click", function () {
      // After the user clicks on an answer, disable further clicks within the same question block
      // ...
      // If incorrect, set its class so that the stylesheet can color it as incorrect
      // ...
      // Set the class of the correct answer so that it can be shown
      // ...
  });
});
```

The pseudocode explains the three things that we need to do.

Our first step is to disable the clicks for that question so the user cannot keep selecting answers until they find the correct one. To do this, we need to get access to the whole question block full of questions and then find the questions in the block and disable them. We place this in our code above as the 1st piece of pseudocode:

```
// After the user clicks on an answer, disable further clicks within the same question block
let block = this.closest(".question-block"); // search DOM to look for this label's question-block
block.querySelectorAll(".answer").forEach(a => {
    a.style.pointerEvents = "none"; // disable clicks for all questions now
});
```

The word `this` represents the label (i.e., question) that was clicked on. By using `.closest()` function, we search up the **DOM** tree hierarchy, starting from that label's position in the tree, to find the first **HTML** element that has the `question-block` class. Then we can ask that **HTML** element (which is the `<div class="question-block">` tag that contains the question labels) to look through all its descendants that are of the `"answer"` class (i.e., we do not want the question `<p>` tag). Then for each of those elements `a` (i.e., the question labels) we disable the clicks by setting the `style.pointerEvents` attributes to `'none'`.

Now, our style sheet has the following settings for two classes of answers: **correct** and **wrong**:

```
/* Highlight correct answer */
.correct {
  background-color: #d4edda;
  border-color: #28a745;
  color: #155724;
}

/* Highlight wrong answer */
.wrong {
  background-color: #f8d7da;
  border-color: #dc3545;
  color: #721c24;
}
```

So, to show an answer as wrong when clicked, we just set the class for that answer label to `"wrong"` and let the browser handle the rest via the style sheet. Insert this code into the 2nd piece of the pseudocode:

```
// If incorrect, set its class so that the stylesheet can color it as incorrect
if (this.dataset.correct === "false") {
    this.classList.add("wrong");
}
```

Notice that we access the label that was clicked on through the word `this`. Then we access the `dataset` (i.e., the things that we set using `data-*` earlier). Then we access the `.correct` attribute from that set of data and compare it to `"false"` which has been set earlier to identify wrong answers. To highlight it as a wrong answer we simply add a `class="wrong"` to the element (i.e., `this.classList.add("wrong")`) and the browser will do the rest.

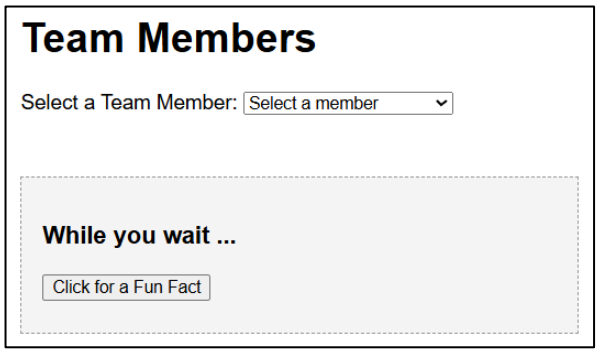
Now, regardless of which answer the user selected, we still want to highlight the correct answer after any click, so we need to add this as the 3rd piece of our pseudocode, which will get the correct answer (i.e., we cannot use `this` anymore because `this` might be a wrong answer label that we clicked):

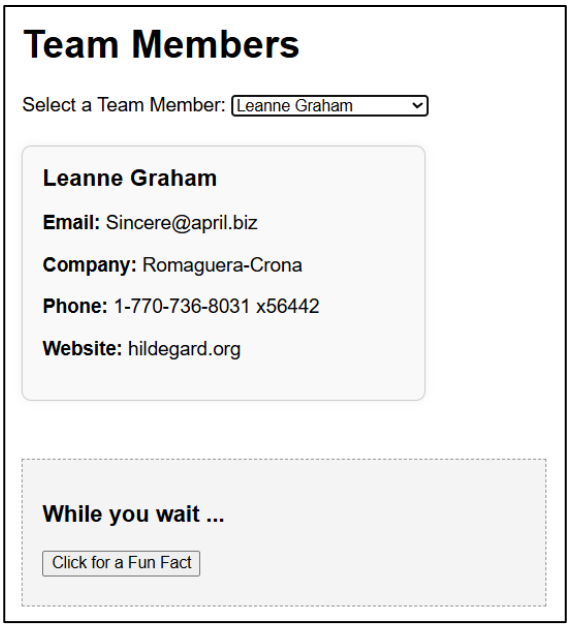
```
// Set the class of the correct answer so that it can be shown
let correctAnswer = block.querySelector('.answer[data-correct="true"]');
if (correctAnswer)
    correctAnswer.classList.add("correct");
```

Notice that we ask the block (from earlier) to find the answer label that had its `correct` data set to `"true"`. Hopefully we had no typos and there is indeed an answer in the list that is **true**. That is why we have the IF statement … to handle the case of **null** for when that situation does occur. Of course, to color it correctly, we add `"correct"` to that label's classes and the browser will color it for us due to the style sheet.

7.4 Team Members Example

Let's do another example that shows the benefits of **AJAX**. We will create a simple page that obtains some team member data from a pretend server (i.e., https://jsonplaceholder.typicode.com/users) and populates a dropdown list of team member names. When first loaded, our page will show as below on the left.





Then when the user selects a member from the list, another request will be made to the server for that member's profile information. When that information comes back, we will display it as shown here on the right.

We will simulate a slow server response by adding in a 3 second delay when requesting a team member's profile data. We include a simple fact widget at the bottom of the page that will display a random fact when the user clicks on a button. As we will see, we can interact with that fact widget while that member's profile is loaded. This will illustrate the asynchronous nature of **AJAX**.

Here is the basic HTML code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Team Member Profile Preview</title>
    <link href="ajax-demo.css" rel="stylesheet">
</head>
<body>
    <h1>Team Members</h1>

    <label for="memberDropdown">Select a Team Member:</label>
    <select id="memberDropdown">
        <option disabled selected value="">Select a member</option>
    </select>

    <div id="loading" style="display: none;">Loading information ...</div>
    <div id="error" class="error"></div>

    <div id="memberCard">
        <h2 id="cardName"></h2>
        <p><span class="field-label">Email:</span> <span id="cardEmail"></span></p>
        <p><span class="field-label">Company:</span> <span id="cardCompany"></span></p>
        <p><span class="field-label">Phone:</span> <span id="cardPhone"></span></p>
        <p><span class="field-label">Website:</span> <span id="cardWebsite"></span></p>
    </div>

    <div id="funFactBox">
        <h3>While you wait ... </h3>
        <button id="factBtn">Click for a Fun Fact</button>
        <div id="factOutput"></div>
    </div>

    <script src="ajax-demo.js"></script>
</body>
</html>
```

Notice that there is a dropdown list (defined by the **<select>** tag). The ** ** section is not displayed upon start (i.e., **display: none**). We will only display that when loading the information. The ** ** section is empty, but it will be filled in with an error message if an error occurred.

The member card has a header and 4 fields. However, it is not displayed at first because our **CSS** file has it as hidden (for now) since there is no information to show. The **"funFactBox"** is shown at the bottom but the ** ** is empty … it will get filled in when the button is clicked.

The **CSS** file is **ajax-demo.css** … we will not go through it. Perhaps though, we can point out that the **#** character allows us to specify the appearance for a specific element id:

```css
#factOutput {  // refers to the <div> with id "factOutput"
    margin-top: 0.5rem;
    font-style: italic;
}
```

We will start our **ajax-demo.js** code with some code to handle the fun fact box:

```
// Fun Fact logic
const facts = [
    "The first website went online in 1991.",
    "JavaScript was created in just 10 days.",
    "JSON stands for JavaScript Object Notation.",
    "AJAX doesn't require reloading the whole page.",
    "XMLHttpRequest was introduced in Internet Explorer 5."
];
let factBtn = document.getElementById("factBtn");
let factOutput = document.getElementById("factOutput");

factBtn.addEventListener("click", function () {
    let randomIndex = Math.floor(Math.random() * facts.length);
    factOutput.textContent = facts[randomIndex];
});
```

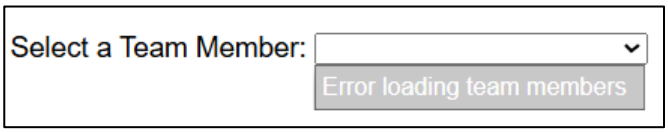
We simply add an event handler to populate the ** ** in the **HTML** file with a random fact. Nothing is new and exciting here.

Now let's look at the code for requesting the team member names and ids from the server and populating the list. This will happen when we first load the page:

```
// Load team members into dropdown
let dropdown = document.getElementById("memberDropdown");
let xhrList = new XMLHttpRequest();
xhrList.open("GET", "https://jsonplaceholder.typicode.com/users", true);

xhrList.onload = function () {
    if (xhrList.status === 200) {
        let members = JSON.parse(xhrList.responseText);
        members.forEach(member => {
            let option = document.createElement("option");
            option.value = member.id;
            option.textContent = member.name;
            dropdown.appendChild(option);
        });
        // Add a non-existent member for testing
        let option = document.createElement("option");
        option.value = 0;
        option.textContent = "unknown";
        dropdown.appendChild(option);
    } else {
        console.error("Error loading team member data:", xhrList.status);
        dropdown.innerHTML = "<option disabled>Error loading team members</option>";
    }
};

xhrList.onerror = function () {
    console.error("Network error loading team member list.");
    dropdown.innerHTML = "<option disabled>Error loading team members</option>";
};
xhrList.send();
```

This code is very similar to our trivia example. The `members` variable ends up being an array of **JavaScript** objects when the response returns ok. We then create an **<option>** element with the `value` being the member's `id` and the `textContent` being the member's `name`. We then add it to the list. For this example, I also added an extra `"unknown"` option at the end of the dropdown list so that we can see what happens later when we ask for data from a non-existent person.

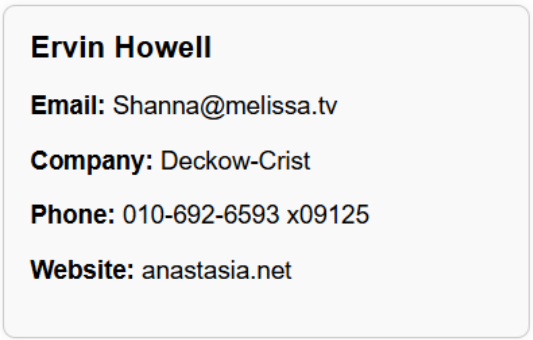
If an error occurs when reading the data from the server, we display something to the console and also put a single non-selectable option to the dropdown list indicating that there was an error as shown here:



Finally, we go ahead and send that request.

Notice that we are not doing anything with all that member's data. We will "pretend" that we only received the names and IDs and that we need to ask the server for the member profile information each time. So, the next step is to handle what happens when the user selects a name from the list.

When that happens, we need to send another request to the server to get the member's profile data and then wait for the response. Then we will display the profile data in a nice box like this (which is defined by the ** ** in our **ajax-demo.html** file):



Before making the request, we should fill in the text for the ** ** tag to inform the user that we are loading something (in case it takes a while) and make that visible (because it started off as hidden). We will also want to hide the member card (if there was one showing) because we are loading new data, as well as any previous errors that were shown. Here is a start to our code:

```
// Load selected member's profile (with delay)
dropdown.addEventListener("change", function () {
    let loadingEl = document.getElementById("loading");
    let errorEl = document.getElementById("error");
    let memberCard = document.getElementById("memberCard");

    let memberId = this.value;
    if (!memberId) return;      // stop if the selected member is somehow invalid
    loadingEl.textContent = "Loading data ... "; // Now show that things are loading
    loadingEl.style.display = "block";           // Make it visible
    errorEl.textContent = "";                    // Hide error until we know if there is one
    memberCard.style.display = "none";           // Hide the card since it is not ready yet
});
```

Now we need to make the request by adding the code below. Notice that we include the **`memberId`** because that is what the server's **api** requires if we want to get a particular member's data.

```
    let xhrmember = new XMLHttpRequest();
    xhrmember.open("GET", `https://jsonplaceholder.typicode.com/users/${memberId}`, true);
    xhrmember.onload = function () {
        if (xhrmember.status === 200) {

        } else {

        }
    };

    xhrmember.onerror = function () {

    };

    xhrmember.send();
});
```

The `.onerror` code is easy. We hide the loading message and add some text to the error element:

```
    xhrmember.onerror = function () {
        loadingEl.style.display = "none";
        errorEl.textContent = "Network error.";
        console.error("Network error during team member fetch.");
    };
```

We do something similar if we don't get a **200** status response. So, we add this inside the **else** of the `.onload` code:

```
        } else {
            loadingEl.style.display = "none";
            errorEl.textContent = "Error loading member data (status " + xhrmember.status + ")";
            console.error("Load team member error:", xhrmember.status, xhrmember.responseText);
        }
```

So, what do we do when the data arrives ok? We just need to fill in the text for the member card, then hide the loading text and show the card. So, we can add this to the **if** of the `.onload` code:

```
        if (xhrmember.status === 200) {
            let member = JSON.parse(xhrmember.responseText);

            // Keep loading visible until after delay
            setTimeout(() => {
                document.getElementById("cardName").textContent = member.name;
                document.getElementById("cardEmail").textContent = member.email;
                document.getElementById("cardCompany").textContent = member.company.name;
                document.getElementById("cardPhone").textContent = member.phone;
                document.getElementById("cardWebsite").textContent = member.website;

                loadingEl.style.display = "none"; // hide only after showing data
                memberCard.style.display = "block";
            }, 3000); // simulate delay
```

Notice the highlighted code. We wrapped the “filling in the member card and showing it” with a 3 second delay to simulate how things will work if the server was slow at responding. This is just for demonstration purposes and is not part of the code.

When we load the page it almost works as expected. But what happens if we select a couple of other users while the server is delaying? It ends up requesting all of them in turn and the behavior seems odd. Instead, let’s adjust the code so that we `abort()` the request if the user selects another member while waiting for the other ones to load.

To do this, we just need a *flag* variable to indicate if a member is being downloaded. In that case, if we select another one, then we call the **abort().** But since we are also simulating a delay, we will also need to cancel the timer when we abort. Below is the altered listener code. The yellow indicates changes that are not part of the actual code … they are just needed because of the fake delay that we added shown in green highlight. The cyan highlights indicate code that we need to add:

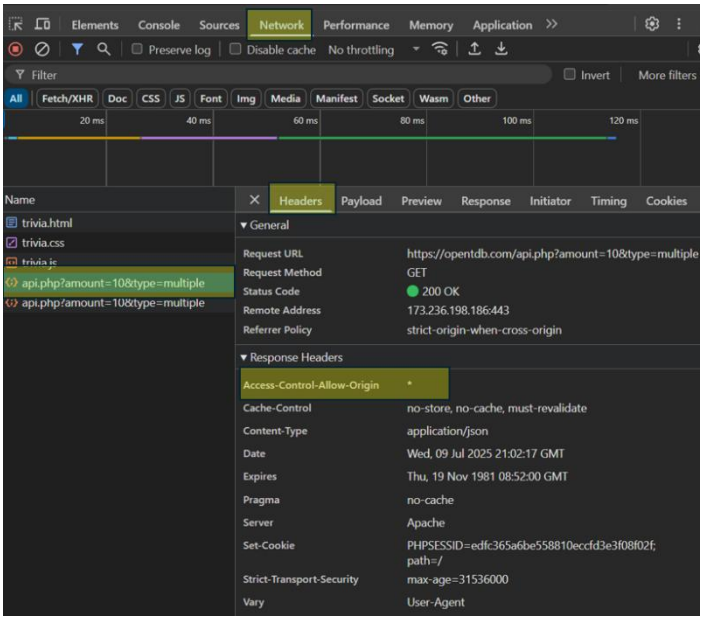
```
// Load selected member's profile (with delay)
let xhrmember = null;  // no member data being requested at this time
let delayTimer = null; // stores the timeout ID

dropdown.addEventListener('change', function () {
    let loadingEl = document.getElementById('loading');
    let errorEl = document.getElementById('error');
    let memberCard = document.getElementById('memberCard');

    let memberId = this.value;
    if (!memberId) return;   // stop if the selected member is somehow invalid

    // If a member's data is already being downloaded, then abort() it before sending new request
    if (xhrmember && xhrmember.readyState !== XMLHttpRequest.DONE) {
        xhrmember.abort();
    }
    if (delayTimer) {
        clearTimeout(delayTimer);
        delayTimer = null;
    }

    loadingEl.textContent = "Loading data ... "; // Now show that things are loading
    loadingEl.style.display = "block";           // Make it visible
    errorEl.textContent = "";                    // Hide error until we know if there is one
    memberCard.style.display = "none";           // Hide the card since it is not ready yet

    xhrmember = new XMLHttpRequest(); // removed the let from before since defined above now
    xhrmember.open('GET', `https://jsonplaceholder.typicode.com/users/${memberId}`, true);
    xhrmember.onload = function () {
        if (xhrmember.status === 200) {
            let member = JSON.parse(xhrmember.responseText);

            // Keep loading visible until after delay
            delayTimer = setTimeout(() => {
                document.getElementById('cardName').textContent = member.name;
                document.getElementById('cardEmail').textContent = member.email;
                document.getElementById('cardCompany').textContent = member.company.name;
                document.getElementById('cardPhone').textContent = member.phone;
                document.getElementById('cardWebsite').textContent = member.website;
```

```
                loadingEl.style.display = 'none'; // hide only after showing data
                memberCard.style.display = 'block';
            }, 3000); // simulate delay
        } else {
            loadingEl.style.display = 'none';
            errorEl.textContent = 'Error loading member data (status ' + xhrmember.status + ')';
            console.error("Load team member error:", xhrmember.status, xhrmember.responseText);
        }
    };

    xhrmember.onerror = function () {
        loadingEl.style.display = 'none';
        errorEl.textContent = 'Network error.';
        console.error("Network error during team member fetch.");
    };

    xhrmember.send();
});
```

7.5 Limitations

AJAX has some limitations. By default, when our **JavaScript** (running in the browser) uses **AJAX** to request data (e.g., with **XMLHttpRequest**) the browser restricts those requests to the same origin. The ***origin*** is defined by the **URL** of the web page that loaded the **JavaScript**, including its protocol, domain, and port. This restriction is a security feature called the ***Same-Origin Policy***. It is designed to prevent malicious scripts on one site from accessing sensitive data from another site (e.g., our bank or private dashboard).

So, if our page is loaded up in the browser from **http://localhost:3000**, we cannot make **AJAX** requests to another domain (e.g., **https://api.example.com**) unless that server explicitly allows it using ***CORS*** (***Cross-Origin Resource Sharing***).



As it turns out, we were allowed to access the opentdb.com trivia database server because it supports **CORS**. We can tell if it supports it by looking at one of the response headers that we got back. If it says `Access-Control-Allow-Origin: *` then it supports **CORS**.

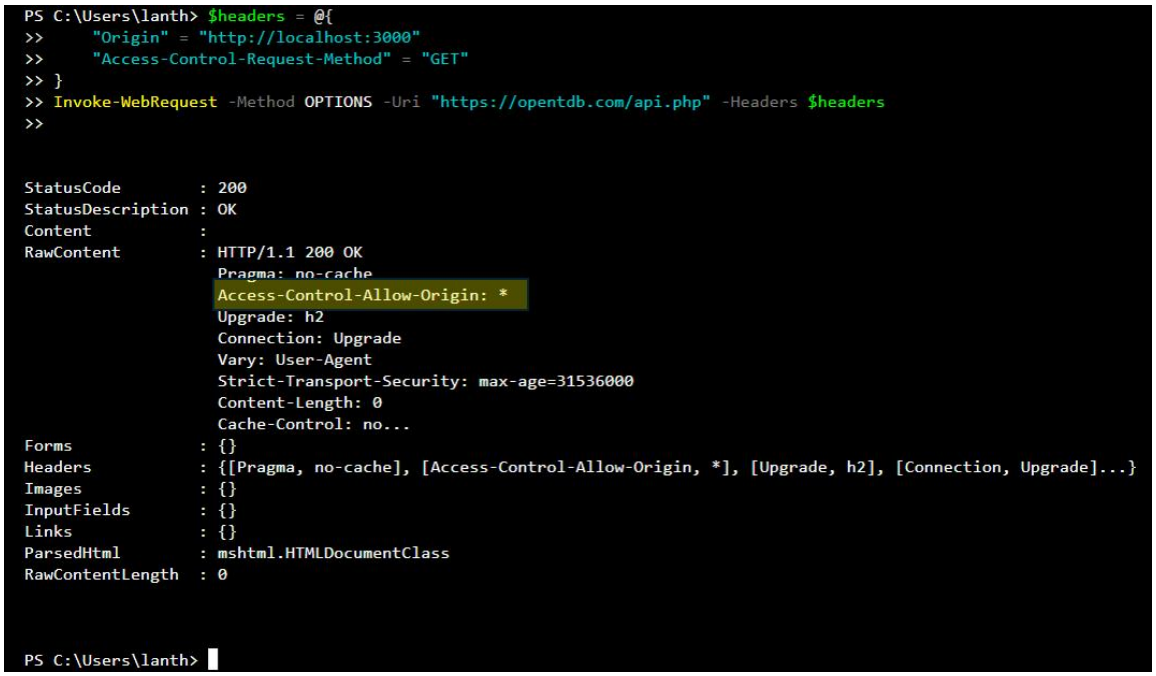
Notice that the request was made to **api.php**. That means it was not just the home page, but the site we need to ask is the **api** portion (i.e., the one that gives us the data), which is:

https://opentdb.com/api.php

Unfortunately, the only way to know if a server supports **CORS** is to send it a request (or simulate one). To simulate a request in a windows powershell (e.g., a **VScode** Terminal), we can type in the following to see if the server supports **CORS**. We can put our own host name as the origin and put in the server's web address to try different ones (note that the newlines are required):

```
$headers = @{
    "Origin" = "http://localhost:3000"
    "Access-Control-Request-Method" = "GET"
}
Invoke-WebRequest -Method OPTIONS -Uri "https://opentdb.com/api.php" -Headers $headers
```

Here is the result from running this in a **VScode** terminal :



For more information on **CORS** we can check out these sites:

- https://www.keycdn.com/support/cors
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Another limitation of **AJAX** is that dynamic page updates can break things like bookmarking and the browser's back button. Since **AJAX** updates the content without changing the **URL**, the browser history doesn't reflect those changes. So, if a user tries to "go back," they return to the previous full page, not the last **AJAX**-loaded content. Similarly, bookmarking the page won't restore the dynamically loaded state.

To address the issue of broken navigation and bookmarking caused by **AJAX**, developers can use the **History API** (`history.pushState()` and `history.replaceState()`) to manually update the **URL** without reloading the page. This allows each **AJAX**-driven change to be reflected in the browser's address bar and history stack. As a result, users can use the **back** button to return to previous states, and bookmarks will point to the correct dynamic content. Alternatively, simpler apps may use ***fragment identifiers*** (e.g., #page2) to track state changes in the **URL** without triggering a page reload.