PER SCHOLAS

> 308A.4 - AJAX and Data Fetching

# 308A.4 - AJAX and Data Fetching

## Learning Objectives

By the end of this lesson, learners will be able to:

- Describe the purpose of AJAX and the `XMLHttpRequest` object.
- Use AJAX to make requests from a server without reloading the webpage.
- Use AJAX to receive and work with data from a server.
- Use the `fetch` API to asynchronously communicate with a server.
- Use the `axios` library to asynchronously communicate with a server.

## Asynchronous JavaScript and XML

Asynchronous JavaScript and XML (AJAX) is a term used to describe the use of the XMLHttpRequest (XHR) object to communicate with servers. Despite the naming conventions, AJAX and the XHR object can both be used to retrieve any type of data, not just XML.

"Asynchronous" in the context of web development is the ability to communicate with a server, exchange data (send and receive), and update the local webpage without having to fully refresh it. This allows individual parts of a webpage to update individually of one another, and - importantly - without waiting for the other portions to finish their changes.

## The XMLHttpRequest Object

Using AJAX involves creating a new XHR object to begin your request:

```
1  const request = new XMLHttpRequest();
```

The XHR object methods for communicating with the server are `open` and `send`.

- The first parameter of `open` is the HTTP request method supported by the server you're communicating with. There are many possible request methods, but the most commonly used are `GET`, `POST`, `PUT`, and `DELETE`. As per the HTTP standard, request methods should be entirely capitalized.

- The third parameter of `open` is optional, and defines whether the request is asynchronous (default `true`). Asynchronous requests allow the code on your webpage to continue executing even while waiting for the response from the server (which could take an unknown amount of time).
- The `send` method takes a single parameter containing the data (if any) that you want to send to the server as part of a `POST` request.

```
request.open("GET", "http://www.example.com/myFile.json", true);
request.send();
```

If you want to `POST` data, you may have to set the MIME type of the data being sent. To do so, use the `setRequestHeader` method of the XHR object before your `send` call.

```
request.setRequestHeader(
    "Content-Type",
    "my-MIME-type-here"
);
```

Now that we've learned how to send the request, how do we deal with the response?

The XHR object comes with an event listener called `onreadystatechange`, which we can assign a function to handle the response as appropriate.

This event will fire each time the `readyState` value of the request object changes. The value of `readyState` also allows us to include specific logic within our event handler function cooresponding to the current state of the request, as follows:

- 0 (uninitialized) or (request not initialized). Client has been created. `open()` not called yet.
- 1 (loading) or (server connection established). `open()` has been called.
- 2 (loaded) or (request received). `send()` has been called, and headers and status are available.
- 3 (interactive) or (processing request). Downloading; `responseText` holds partial data.
- 4 (complete) or (request finished and response is ready). The operation is complete.

Each of these values also cooresponds to a static property of the `XMLHttpRequest` object, which can be used to make our code more readable:

- 0: `XMLHttpRequest.UNSENT`
- 1: `XMLHttpRequest.OPENED`
- 2: `XMLHttpRequest.HEADERS_RECEIVED`
- 3: `XMLHttpRequest.LOADING`
- 4: `XMLHttpRequest.DONE`

PER SCHOLAS

```
        // request is complete; do tasks.
    } else {
        // request is not complete.
    }
}

request.onreadystatechange = handleResponse;
```

The two options for accessing the response data are `request.responseText`, which contains the response as a string of text, and `request.responseXML`, which contains the response as an `XMLDocument` object that can be traversed with JavaScript DOM functions.

## Simple AJAX Example

Here, we'll discuss a simple example that demonstrates what we've discussed so far. Feel free to follow along by creating two files **within the same directory**, as described below.

`test.html`

```
This is a test!
```

`index.html`

```
<button id="myBtn" type="button">Test it Out</button>

<script>
let request;

document
    .getElementById("myBtn")
    .addEventListener("click", testRequest);

function testRequest() {
    request = new XMLHttpRequest();

    if (!request) {
        alert("Failed to create an XMLHttpRequest Object.");
        return false;
    }

    request.onreadystatechange = alertResponse;
    request.open("GET", "test.html");
    request.send();
}
```

PER SCHOLAS

```
            alert(request.responseText);
        } else {
            alert("The request returned a status other than 200 OK: " + request.status)
        }
    }
}
</script>
```

Here, we create a button that calls `testRequest` when clicked, which creates a request handled by `alertResponse`. If the reponse returns a status of 200 (which means OK), we send an alert with the text contents of the response.

We've added the extra step of checking the response status in this example, which is good practice. See this reference for a list of potential response statuses. Note that because we are accessing the response status, an exception may be thrown in the event that there is a communication error (like the server being unavailable). You can mitigate this by wrapping your logic in a `try...catch` statement.

## Working with XML

The previous example shows how to easily display contents as text, but most data is more complex. If we converted `test.html` into a proper XML file and named it `test.xml`, we could handle the response with `request.responseXML` and make use of familiar JavaScript DOM methods.

`test.xml`

```
<? xml version="1.0">
<root>XML data test!</root>
```

`index.html`

```
<button id="myBtn" type="button">Test it Out</button>

<script>
let request;

document
    .getElementById("myBtn")
    .addEventListener("click", testRequest);

function testRequest() {
    request = new XMLHttpRequest();

    if (!request) {
        alert("Failed to create an XMLHttpRequest Object.");
        return false;
    }
```

PER SCHOLAS

```
    request.send();
}

function alertResponse() {
    if (request.readyState === XMLHttpRequest.DONE) {
        if (request.status === 200) {
            // Change the way we handle the response data.
            const xmlDoc = request.responseXML;
            const doc_root = xmlDoc.querySelector("root");
            let data = doc_root.firstChild.data;

            alert(data);
        } else {
            alert("The request returned a status other than 200 OK: " + request.status)
        }
    }
}
</script>
```

While this is still a simple example, working with XML gives us the opportunity to navigate more complex data responses with ease.

## Posting Data

Assume you have a form with some data that you want sent to your server for processing. While this can be done with pure HTML, AJAX allows us to process the data, receive a response, and update portions of the page without refreshing it. This provides a much smoother, more responsive user experience.

Let's modify the example we've been working with to send data from a form. First, add the following simple text input field:

`index.html`

```
<label>
    Input:
    <input type="text" id="myInput"  />
</label>

<!-- Previous code omitted below -->
```

We also need to ensure this data gets sent in our request, so we need to modify both the `open` and `send` method calls. For this example, we'll be using a free fake API located at "https://jsonplaceholder.typicode.com/todos." You can click on this link to see how the data we receive will be structured.

```
request.setRequestHeader(
    "Content-Type",
    "application/x-www-form-urlencoded"
);

let inputVal = document.getElementById("myInput").value;
let encodedVal = encodeURIComponent(inputVal);
request.send(`data=${encodedVal}`);
```

Since this is a `POST` request, the status we get back also might be different. If we test our code now, we'll see that receive a status of `201 Created`. This is still a successful response, so we'll want to modify the way our code checks status. While we're at it, we'll also change the way we read the response to `.responseText`.

```
if (request.status === 200 || request.status === 201) {
    alert(request.responseText);
    // omitted other code
```

Another test would reveal that we're getting a response that contains a JSON object that looks like this:

```
{
    "data": "Testing",
    "id": 201
}
```

Our `POST` request has created a new object on the server with a `data` value of whatever we entered into our text field and an unique `id` field. Since this is a fake API, these changes won't actually occur in the external database, but in the case of a real API we would then be able to `GET` our newly created resource, or modify it and existing resources with `PUT`, `PATCH`, `DELETE`, and so on.

If we needed to parse this new object, how might we go about doing so?

No matter what language or technology is being used by a server, the client has the tools to handle requests and responses. It is important that you research any APIs you may access during your projects, as the only thing that is guaranteed is that no two APIs are identical. Formatting, data types, response times... **everything** can vary.

There are some tools and interfaces that can make these processes simpler and more intuitive, including the Fetch API and Axios, which we will discuss in this lesson.

## The Fetch API

The Fetch API is a more powerful and flexible replacement for the `XMLHttpRequest` object. These two options for communication are **not** the only ones, and you will likely run into many other tools, interfaces, and libraries that accomplish similar goals as you continue learning. The fundamentals, however, remain the same.

PER SCHOLAS

```
async function logJSONData() {
    const response = await fetch("http://www.example.com/something.json");
    const jsonData = await response.json();
    console.log(jsonData);
}
```

`fetch()` does not directly return the JSON response body, but instead a `Promise` that resolves with a `Response` object. The `Response` object is a representation of the **entire** HTTP response, so the `json()` method is used to extract the JSON body content from the `Response` (which returns a second `Promise` that resolves with the parsed JSON data).

The `Response` object has other instances methods that you should reference when needed:

- `Response.arrayBuffer()`
  - Returns a promise that resolves with an `ArrayBuffer` representation of the response body.
- `Response.blob()`
  - Returns a promise that resolves with a `Blob` representation of the response body.
- `Response.clone()`
  - Creates a clone of a `Response` object.
- `Response.formData()`
  - Returns a promise that resolves with a `FormData` representation of the response body.
- `Response.json()`
  - Returns a promise that resolves with the result of parsing the response body text as JSON.
- `Response.text()`
  - Returns a promise that resolves with a text representation of the response body.

The second parameter to the `fetch()` method is an options object containing any custom settings that you want to apply to the request. For a full list of the available options, see the MDN reference on `fetch()` parameters.

The options that coorespond to our `XMLHttpRequest` object examples include:

- `method`: The request method, e.g., `GET`, `POST`.
- `headers`: Any headers you want to add to your request, contained within a `Headers` object or an object literal with string values.
- `body`: Any body that you want to add to your request: this can be a `Blob`, an `ArrayBuffer`, a `TypedArray`, a `DataView`, a `FormData`, a `URLSearchParams`, string object or literal, or a `ReadableStream` object.

Here's an example of how we can use `fetch()` to connect to an external API and show a random image of a dog:

```
<img id="dog" src="" height="250" style="cursor: pointer" />

<script>
```

PER SCHOLAS

```
    async function getNewDog() {
        dog.style.cursor = 'wait';
        const response = await fetch("https://dog.ceo/api/breeds/image/random");
        const jsonData = await response.json();
        const url = jsonData.message;

        dog.src = url;
        dog.style.cursor = 'pointer';
    }

    getNewDog();
</script>
```

Here is that example in practice; click! Aren't they adorable?

## Posting Data with Fetch

Many requests also need to send data, so let's use our previous example to see how we can post data to an API with `fetch()`. Here are the two methods side by side:

`XMLHttpRequest`

```
<label>
    Input:
    <input type="text" id="myInput"  />
</label>

<button id="myBtn" type="button">Test it Out</button>

<script>
let request;

document
    .getElementById("myBtn")
    .addEventListener("click", testRequest);

```

`Fetch`

PER SCHOLAS

```
    if (!request) {
        alert("Failed to create an XMLHttpRequest Object.");
        return false;
    }

    request.onreadystatechange = alertResponse;
    request.open("POST", "https://jsonplaceholder.typicode.com/todos");

    request.setRequestHeader(
        "Content-Type",
        "application/x-www-form-urlencoded"
    );

    let inputVal = document.getElementById("myInput").value;
    let encodedVal = encodeURIComponent(inputVal);
    request.send(`data=${encodedVal}`);
}

function alertResponse() {
    if (request.readyState === XMLHttpRequest.DONE) {
        if (request.status === 200 || request.status === 201) {
            alert(request.responseText);
        } else {
            alert("The request returned a status other than 200 OK: " + request.status);
        }
    }
}
</script>
```

As you can see, using `fetch()` alongside `async/await` and promises results in code that is easier to implement and understand.

Fetch is considered the modern replacement for `XMLHttpRequest`, so why do we even teach the latter?

Firstly, foundational tools and concepts are always a good place to start. Secondly, the fetch API is not supported on all browsers by default. As always when working with modern tools, libraries, interfaces, etc., you should verify the compatibility of anything you're uncertain about with Can I Use?. Check the link to see where `Fetch` and the fetch API are supported by default.

Thankfully, fetch is compatible with *almost* all modern browsers. In front-end development, always make sure you understand the compatibility requirements of whatever applications you are working on.

## Axios

Axios is another very commonly used alternative to `XMLHttpRequest` and `fetch()`, though it is important to note that Axios has wider browser support than `fetch()` because it uses `XMLHttpRequest` under the hood. However, Axios is not native to some browsers like `fetch()` is, so it needs to be manually installed or imported into any projects you use it with. For instructions on how to do so, see the Axios documentation on getting started.

PER SCHOLAS

```
1 <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Axios lists the following features within its documentation:

- Make `XMLHttpRequests` from the browser
- Make `http` requests from node.js
- Supports the `Promise` API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Timeouts
- Query parameters serialization with support for nested entries
- Automatic request body serialization to:
  - JSON (`application/json`)
  - Multipart / FormData (`multipart/form-data`)
  - URL encoded form (`application/x-www-form-urlencoded`)
- Posting HTML forms as JSON
- Automatic JSON data handling in response
- Progress capturing for browsers and node.js with extra info (speed rate, remaining time)
- Setting bandwidth limits for node.js
- Compatible with spec-compliant `FormData` and `Blob` (including node.js)
- Client side support for protecting against `XSRF` (cross-site request forgery)

The simplest syntax for making an Axios request is `axios(url)`, but there are many configuration options that can be used to change the behavior of a request. These configuration options can be passing into an `axios()` call through an object containing one or more properties, as described below (taken from the Axios documentation). The only mandatory property is the `url`.

```
{
  // `url` is the server URL that will be used for the request
  url: '/user',

  // `method` is the request method to be used when making the request
  method: 'get', // default

  // `baseURL` will be prepended to `url` unless `url` is absolute.
  // It can be convenient to set `baseURL` for an instance of axios to pass relative UR
  // to methods of that instance.
  baseURL: 'https://some-domain.com/api',

  // `transformRequest` allows changes to the request data before it is sent to the ser
```

PER SCHOLAS

```
// You may modify the headers object.
transformRequest: [function (data, headers) {
  // Do whatever you want to transform the data

  return data;
}],

// `transformResponse` allows changes to the response data to be made before
// it is passed to then/catch
transformResponse: [function (data) {
  // Do whatever you want to transform the data

  return data;
}],

// `headers` are custom headers to be sent
headers: {'X-Requested-With': 'XMLHttpRequest'},

// `params` are the URL parameters to be sent with the request
// Must be a plain object or a URLSearchParams object
// NOTE: params that are null or undefined are not rendered in the URL.
params: {
  ID: 12345
},

// `paramsSerializer` is an optional function in charge of serializing `params`
// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
paramsSerializer: function (params) {
  return Qs.stringify(params, {arrayFormat: 'brackets'})
},

// `data` is the data to be sent as the request body
// Only applicable for request methods 'PUT', 'POST', 'DELETE', and 'PATCH'
// When no `transformRequest` is set, must be of one of the following types:
// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
// - Browser only: FormData, File, Blob
// - Node only: Stream, Buffer
data: {
  firstName: 'Fred'
},

// syntax alternative to send data into the body
// method post
// only the value is sent, not the key
data: 'Country=Brasil&City=Belo Horizonte',

// `timeout` specifies the number of milliseconds before the request times out.
// If the request takes longer than `timeout`, the request will be aborted.
timeout: 1000, // default is `0` (no timeout)

// `withCredentials` indicates whether or not cross-site Access-Control requests
```

PER SCHOLAS

```
// `adapter` allows custom handling of requests which makes testing easier.
// Return a promise and supply a valid response (see lib/adapters/README.md).
adapter: function (config) {
  /* ... */
},

// `auth` indicates that HTTP Basic auth should be used, and supplies credentials.
// This will set an `Authorization` header, overwriting any existing
// `Authorization` custom headers you have set using `headers`.
// Please note that only HTTP Basic auth is configurable through this parameter.
// For Bearer tokens and such, use `Authorization` custom headers instead.
auth: {
  username: 'janedoe',
  password: 's00pers3cret'
},

// `responseType` indicates the type of data that the server will respond with
// options are: 'arraybuffer', 'document', 'json', 'text', 'stream'
//   browser only: 'blob'
responseType: 'json', // default

// `responseEncoding` indicates encoding to use for decoding responses (Node.js only)
// Note: Ignored for `responseType` of 'stream' or client-side requests
responseEncoding: 'utf8', // default

// `xsrfCookieName` is the name of the cookie to use as a value for xsrf token
xsrfCookieName: 'XSRF-TOKEN', // default

// `xsrfHeaderName` is the name of the http header that carries the xsrf token value
xsrfHeaderName: 'X-XSRF-TOKEN', // default

// `onUploadProgress` allows handling of progress events for uploads
// browser only
onUploadProgress: function (progressEvent) {
  // Do whatever you want with the native progress event
},

// `onDownloadProgress` allows handling of progress events for downloads
// browser only
onDownloadProgress: function (progressEvent) {
  // Do whatever you want with the native progress event
},

// `maxContentLength` defines the max size of the http response content in bytes allo
maxContentLength: 2000,

// `maxBodyLength` (Node only option) defines the max size of the http request conten
maxBodyLength: 2000,

// `validateStatus` defines whether to resolve or reject the promise for a given
// HTTP response status code. If `validateStatus` returns `true` (or is set to `null`
```

PER SCHOLAS

```
    return status >= 200 && status < 300; // default
  },

  // `maxRedirects` defines the maximum number of redirects to follow in node.js.
  // If set to 0, no redirects will be followed.
  maxRedirects: 5, // default

  // `socketPath` defines a UNIX Socket to be used in node.js.
  // e.g. '/var/run/docker.sock' to send requests to the docker daemon.
  // Only either `socketPath` or `proxy` can be specified.
  // If both are specified, `socketPath` is used.
  socketPath: null, // default

  // `httpAgent` and `httpsAgent` define a custom agent to be used when performing http
  // and https requests, respectively, in node.js. This allows options to be added like
  // `keepAlive` that are not enabled by default.
  httpAgent: new http.Agent({ keepAlive: true }),
  httpsAgent: new https.Agent({ keepAlive: true }),

  // `proxy` defines the hostname, port, and protocol of the proxy server.
  // You can also define your proxy using the conventional `http_proxy` and
  // `https_proxy` environment variables. If you are using environment variables
  // for your proxy configuration, you can also define a `no_proxy` environment
  // variable as a comma-separated list of domains that should not be proxied.
  // Use `false` to disable proxies, ignoring environment variables.
  // `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and
  // supplies credentials.
  // This will set an `Proxy-Authorization` header, overwriting any existing
  // `Proxy-Authorization` custom headers you have set using `headers`.
  // If the proxy server uses HTTPS, then you must set the protocol to `https`.
  proxy: {
    protocol: 'https',
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'mikeymike',
      password: 'rapunz3l'
    }
  },

  // `cancelToken` specifies a cancel token that can be used to cancel the request
  // (see Cancellation section below for details)
  cancelToken: new CancelToken(function (cancel) {
  }),

  // `decompress` indicates whether or not the response body should be decompressed
  // automatically. If set to `true` will also remove the 'content-encoding' header
  // from the responses objects of all decompressed responses
  // - Node only (XHR cannot turn off decompression)
  decompress: true // default
```

Axios requests can be made in a number of ways using convenient aliases:

- `axios.request(config)`
- `axios.get(url[, config])`
- `axios.delete(url[, config])`
- `axios.head(url[, config])`
- `axios.options(url[, config])`
- `axios.post(url[, data[, config]])`
- `axios.put(url[, data[, config]])`
- `axios.patch(url[, data[, config]])`
- `axios.postForm(url[, data[, config]])`
- `axios.putForm(url[, data[, config]])`
- `axios.patchForm(url[, data[, config]])`

Note that the `url`, `method`, and `data` properties do not need to be specified within the `config` object when using an alias that includes them elsewhere.

## Axios vs. Fetch

Let's compare Axios and `fetch()` to better understand why we would use one or the other.

**Basic Syntax**

- The basic syntax of each is very similar, with mostly small differences.
- Axios automatically serializes data when sending JavaScript objects to the server using `POST`; fetch does not.
- Axios makes the response available as JSON by default, whereas with fetch you much use the `.json()` method to acquire the body of the response.
- Axios will automatically reject a promise if it receives an HTTP status code outside of the range of success (200 - 299), allowing us to handle HTTP errors in `catch()` blocks. Fetch will still resolve unsuccessful responses, which means that HTTP errors need to be handled separately.

Here is our previous `fetch()` example alongside an Axios example that accomplishes the same task. Each has been slightly modified to log JSON data instead of alerting text data.

`Fetch` `Axios`

```
<label>
    Input:
    <input type="text" id="myInput"  />
</label>
```

PER SCHOLAS

```
<script>
document
    .getElementById("myBtn")
    .addEventListener("click", testRequest);

async function testRequest() {
    let inputVal = document.getElementById("myInput").value;
    let requestBody = { data: inputVal };

    const response = await fetch('https://jsonplaceholder.typicode.com/todos', {
        method: 'POST',
        body: JSON.stringify(requestBody),
        headers: {
            'Content-Type': 'application/json; charset=UTF-8',
        }
    });

    logResponse(response);
}

async function logResponse(response) {
    if (response.ok) {
        const jsonData = await response.json();
        console.log(jsonData);
    } else {
        console.log("The request returned a status other than 200 OK: " + response.status);
    }
}
</script>
```

If we log `response` instead of `response.data`, we will see that Axios packages its response object much differently from fetch. It has the following information:

```
{
  // `data` is the response that was provided by the server
  data: {},

  // `status` is the HTTP status code from the server response
  status: 200,

  // `statusText` is the HTTP status message from the server response
  // As of HTTP/2 status text is blank or unsupported.
  // (HTTP/2 RFC: https://www.rfc-editor.org/rfc/rfc7540#section-8.1.2.4)
  statusText: 'OK',

  // `headers` the HTTP headers that the server responded with
  // All header names are lower cased and can be accessed using the bracket notation.
  // Example: `response.headers['content-type']`
  headers: {},
```

```
    // `request` is the request that generated this response
    // It is the last ClientRequest instance in node.js (in redirects)
    // and an XMLHttpRequest instance in the browser
    request: {}
}
```

**Response Timeout**

- Axios has the ability to easily setup a response timeout, while fetch does not.
- Response timeouts are essential in web development. Without an appropriate timeout, the request can hang and slow down your application.

Here is how we easily set up a response timeout with Axios by adding a `timeout` property to our configuration object. The `timeout` property's value is a number in miliseconds.

```
axios.get(url, {
  ...
  timeout: 5000,
  ...
});
```

While it is possible to create this functionality with fetch, doing so is beyond the scope of this lesson.

**Intercepting Requests and Responses**

- Interceptors allow us to create global behavior for all requests and responses.
- Interceptors fire between actions. A request interceptor can modify a request before it is actually sent, and a response interceptor can modify the response before it is handled by a `then()` or `catch()` statement.

This is best described with examples. First, let's look at how to build a simple interceptor that logs a message to the console every time a request is sent:

```
axios.interceptor.request.use(request => {
  console.log('Request sent.');
  return request;
});
```

If we wanted to log messages for our server responses, we could do something similar using the `.response` property:

```
axios.interceptor.response.use(
  (response) => {
    // Success: status 200 - 299
```

PER SCHOLAS

```
  (error) => {
    // Failure: anything outside of status 2XX
    console.log('Unsuccessful response...');
    throw error;
  }
);
```

If you need to dynamically create and remove interceptors within your application, you can. The method for removing an interceptor, `.eject`, requires that interceptor to be assigned to a variable:

```
const requestInterceptor = axios.interceptor.request.use(request => {
  console.log('Request sent.');
  return request;
});

axios.interceptor.request.eject(requestInterceptor);
```

Let's discuss some more practical use cases. What if we wanted to time every request we made? We could create request and response interceptors to incorporate that data into our request and response objects:

```
axios.interceptors.request.use(request => {
    request.metadata = request.metadata || {};
    request.metadata.startTime = new Date().getTime();
    return request;
});

axios.interceptors.response.use(
    (response) => {
        response.config.metadata.endTime = new Date().getTime();
        response.config.metadata.durationInMS = response.config.metadata.endTime - resp

        console.log(`Request took ${response.config.metadata.durationInMS} milliseconds
        return response;
    },
    (error) => {
        error.config.metadata.endTime = new Date().getTime();
        error.config.metadata.durationInMS = error.config.metadata.endTime - error.conf

        console.log(`Request took ${error.config.metadata.durationInMS} milliseconds.`)
        throw error;
});
```

We'll take a moment here to note that while we're using `request` and `response` to make it very clear what we're dealing with, you will often see the variables `req` and `res` to coorespond to these two items in a true development

PER SCHOLAS

our response object instead of in a metadata field. We could then access it easily through object destructuring:

```
axios.interceptors.request.use(request => {
    request.metadata = request.metadata || {};
    request.metadata.startTime = new Date().getTime();
    return request;
});

axios.interceptors.response.use(
    (response) => {
        response.config.metadata.endTime = new Date().getTime();
        response.durationInMS = response.config.metadata.endTime - response.config.meta
        return response;
    },
    (error) => {
        error.config.metadata.endTime = new Date().getTime();
        error.durationInMS = error.config.metadata.endTime - error.config.metadata.star
        throw error;
});

(async () => {
    const url = 'https://jsonplaceholder.typicode.com/todos/1';

    const { data, durationInMS } = await axios(url);
    console.log(`Request took ${durationInMS} milliseconds.`);
    console.log(data);
})();
```

Some other common usages for Axios interceptors include server authentication handling, especially when authentication tokens need to be periodically refreshed without having the user re-authenticate every time; and retrying failed requests a specified number of times. Like most common tasks in programming, the latter has already been packaged into multiple plugins: axios-retry and retry-axios.

It's important to note that **you can do anything with `fetch()` that you can with Axios**. Axios does not create new possibilities, it simply gives you access to pre-made, streamlined solutions to common problems. If you wanted to, you could implement the entirety of Axios's functionality using the fetch API, but it would be more difficult and time-consuming to do so.

## Practicing Asynchronous JavaScript

These concepts are best cemented through practice and practical application:

ALAB 308A.4.1 - AJAX, Fetch, and Axios will give you tasks to complete using the tools discussed in this lesson.


PER SCHOLAS