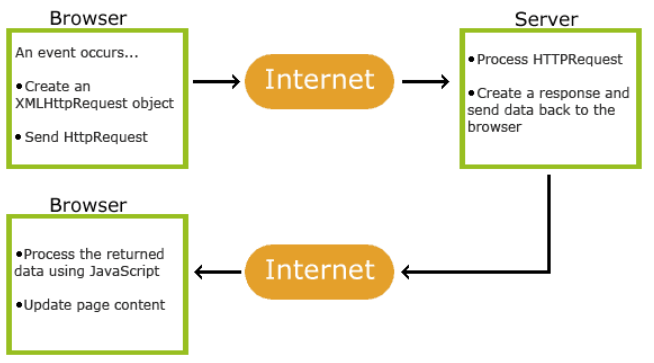
**AJAX**

AJAX (asynchronous JavaScript and XML) is **not a programming language**. It is a **technique** **for accessing web servers** from a web page.

AJAX just uses a combination of:

* A browser built-in XMLHttpRequest object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data)

AJAX allows **web pages to be updated asynchronously by exchanging data with a web server** behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.



1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

So, AJAX is a developer's dream, because you can:

* Update a web page without reloading the page
* Request data from a server - after the page has loaded
* Receive data from a server - after the page has loaded
* Send data to a server - in the background

The data is transferred either by using **XML** or **fetch** techniques, and although **AJAX** has **XML** in the name, **fetch** ended up being more used by the coding community.

**HTTP Methods**

The **Hypertext Transfer Protocol** (HTTP) is designed to enable **communications** between **clients** and **servers**.

HTTP works as a **request-response protocol** between a client and server.

Example: A client (browser) sends an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.

The most used HTTP methods are:

* **Get**: used to request a resource from the server
* **Post**: used to add data to the server (e.g. we have users and we want to add them)
* **Put**: used to edit data on the server.
* **Delete**: used to delete data from the server.

**XML**

**What is XML?**

**Extensible Markup Language (XML)** is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable

**XML Constructor**

The first thing to do is create a new instance by using the built-in constructor **XMLHttpRequest**. It must be called before any other method calls.

const request = new XMLHttpRequest();

**XML Instance Methods**

**Open**

The **XMLHttpRequest** method **open()** initializes a newly-created **request**, or re-initializes an existing one.

**open(method, url**, async, user, password**)**

* **method** - The HTTP request method to use, such as "GET", "POST", "PUT", "DELETE", etc. Ignored for non-HTTP(S) URLs.
* **url** - A string representing the URL to send the request to.
* **async (optional)** - An optional Boolean parameter, defaulting to **true**, indicating whether or not to perform the operation asynchronously. If this value is **false**, the **send()** method does not return until the response is received. If **true**, notification of a completed transaction is provided using event listeners. This must be **true** if the multipart attribute is **true**, or an exception will be thrown.
* **User (optional)** - The optional user name to use for authentication purposes; by default, this is the **null** value.
* **Password (optional)** - The optional password to use for authentication purposes; by default, this is the **null** value.

**onReadyStateChanges**

The **XMLHttpRequest.readyState** property returns the **state** an **XMLHttpRequest** client is in. An XHR client exists in one of the following states:

|  |  |  |
| --- | --- | --- |
| **Value** | **State** | **Description** |
| 0 | UNSENT | Client has been created. open() not called yet. |
| 1 | Opened | open() has been called. |
| 2 | HEADERS\_RECEIVED | send() has been called, and headers and status are available. |
| 3 | LOADING | Downloading; responseText holds partial data. |
| 4 | DONE | The operation is complete. |

XMLHttpRequest.onreadystatechange = callback

Where **callback** is the function executed when the readyState changes.

In this callback function, we want to check for 2 things:

* Check the **readyState**: we only want to run the callback function once we get the data back (once readyState = 4). Until then, we are just going to wait.
* Check the **status**:
* It will return **200** if the request was **successful** (readyState reached 4 – DONE)
* It will return **404** if the server **cannot** **find** the request resource

**Example**

In the example below, we are doing a HTTP request of the file in the './api/sample.txt'. Because the file is correct and the request was successful, we are going to see status: 200 if we console log the object **request**. However, if the file path is wrong, we are going to get status: 400 and **statusText: Not Found**.

const request = new XMLHttpRequest();

console.log(request);

request.open('GET', './api/sample.txt');

request.onreadystatechange = function () {

  if (request.readyState === 4 && request.status === 200) {

    console.log("done")

  } else {

    console.log({

      status: request.status,

      text: request.statusText,

    });

  }

};

request.send();

**JSON**

**JSON** stands for JavaScript Object Notation used to send data from server to web page and vice-versa, which is done in a text format.

There is a rule: **the keys must be in double quotes**.

For example, imagine we have this array of objects and we want to transfer this data to the webpage using JSON.

[

  { "name": "john", "id": 1 },

  { "name": "peter", "id": 2 },

  { "name": "anna", "id": 3 },

  { "name": "susan", "id": 4 }

]

This data is always transferred in text format, which means we have to use **JSON.parse** to convert it to object format. We can then use **map()** to iterate over it and get what we want (e.g. the name of each person).

const url = './api/people.json'; //source file

const request = new XMLHttpRequest();

request.open('GET', url);

request.onreadystatechange = function () {

  if (request.readyState === 4 && request.status === 200) {

    // parse (convert from text to object format)

    const data = JSON.parse(request.responseText);

    // iterate over the array of objects and add the HMTL

    const displayData = data

      .map((item) => {

        return `<p>${item.name}</p>`;

      })

      .join('');

    const element = document.createElement('div');

    element.innerHTML = displayData;

    document.body.appendChild(element);

  } else {

    console.log({

      status: request.status,

      text: request.statusText,

      state: request.readyState,

    });

  }

};

request.send();

**Fetch**

**fetch()** is a **promise based** function provided by the browser and an alternative to XHR.

XHR is not wrong. Fetch is just an alternative approach that has simpler and cleaner syntax. **The end result will be exactly the same**.

**fetch()** takes the **url** that we want to access as parameter.

**It returns a promise**, which means we have to use the method **then** (for the fulfilled case), as well as the catch to get the errors (rejected case).

This then takes the

This **then** **takes the response as an argument** (conventionally named “resp”) which gives as a **response object**, with a bunch of **useful properties and methods**.

|  |  |
| --- | --- |
| const url = './api/people.json';  fetch(url).then((resp) => {    console.log(resp)  }); |  |

One of the methods that you can access from the **response object** is the **json** method, which we can use to transfer data (converts the data into JSON format).

const url = './api/people.json';

fetch(url).then((resp) => {

  return resp.json()

});

**json method returns a promise**, which means we have to chain another **.then** to **access the data**. As you can see below, we accessed the data and it comes already converted in object format.

|  |  |
| --- | --- |
| const url = './api/people.json';  fetch(url).then((resp) => {    return resp.json()  }).then((data) =>{      console.log(data);  }) |  |

Let’s now add a **catch** in case there is an error, as well as make the code cleaner:

fetch(url)

.then((resp) => resp.json())

.then((data) => console.log(data))

.catch((err) => console.log(err))

So, if we want to have the same end result as we had before using XHR to get the name of the people array and display them:

const url = './api/people.json';

const btn = document.querySelector('.btn')

btn.addEventListener('click', () => {

    fetch(url)

    .then((resp) => resp.json())

    .then((data) => displayItems(data))

    .catch((err) => console.log(err))

})

const displayItems = (items) => {

    const displayData = items.map((item) => {

        return `<p>${item.name}</p>`

    }).join('');

    const element = document.createElement('div');

    element.innerHTML = displayData;

    document.body.appendChild(element);

};

**Fetch with async/await**

We can also achieve the same result using **async/await** syntax.

Since we know that **fetch** returns a promise, we can stick **await** before it.

Then, we have to **wait for the promise to be resolved** (either fulfilled or rejected) and convert the data to **json** format (which in this case is then stored in the **data** variable)

const url = './api/people.json';

const btn = document.querySelector('.btn')

btn.addEventListener('click', async () => {

    const resp = await fetch(url);

    const data = await resp.json();

    displayItems(data);

})

const displayItems = (items) => {

    const displayData = items.map((item) => {

        return `<p>${item.name}</p>`

    }).join('');

    const element = document.createElement('div');

    element.innerHTML = displayData;

    document.body.appendChild(element);

};

We can then use **Try/Catch** blocks to **handle the errors**. This **allows for the rest of the code to still run** if what is inside the try block doesn’t run (for example, it’s referencing some variable that doesn’t exist). That way, we don’t ruin the whole application.

const url = './api/people.json';

const btn = document.querySelector('.btn')

btn.addEventListener('click', async () => {

    try {

        const resp = await fetch(url);

        const data = await resp.json();

        displayItems(data);

    }

    catch (err) {

        console.log(error)

    }

})

const displayItems = (items) => {

    const displayData = items.map((item) => {

        return `<p>${item.name}</p>`

    }).join('');

    const element = document.createElement('div');

    element.innerHTML = displayData;

    document.body.appendChild(element);

};