AJAX Request- 4 steps:

1. Create an XMLHTTP Request (XHR) object- tells web browser to get ready for AJAX request

**let xhr = new XMLHttpRequest ();**

1. Create callback function (where most of the programming occurs)-

Program to run when server sends back response. To trigger callback, we use a special browser event (event is something that happens in web browser, i.e. a click or submission of form). Add programming to respond to those AJAX events- most important is the *onreadystatechange* event. Event is triggered when there’s an AJAX request.

Function runs when there is a change in the state of AJAX request (i.e. opening, sending):

**xhr.onreadystatechange = function () { }; //also an event handler**

In this callback function, we’re only interested in final change of state when server sends back response so we can update our webpage with it.

* The *XHR object* keeps track of the state using a special property named *readyState* which include number properties (0-4) each representing a state of the AJAX request - *4* represents request is done and server has sent response.
* *XHR object* also has property called *responseText* where it contains the data web server sent back.

To test for that state:

**xhr.onreadystatechange = function () {**

**if (xhr.readyState === 4) {**

**//take AJAX response and post to webpage**

**document.getElementById(‘ajax’).innerHTML = xhr.responseText;**

**}**

**};**

Callback also needs to handle errors (incorrect URL, can’t connect to Db, request crashes)

So, check server status:

**if (xhr.readyState === 4 && xhr.status === 200){ }**

**else if (xhr.status === 404) {** //message saying file not found, contact customer service**}**

**else if (xhr.status === 500) {** //message saying server problem, try again later**}**

or **else { alert(xhr.statusText) };**

* 200: ok, successful HTTP request
* 404: file not found
* 401: not authorized to access URL
* 500: server error (usually server processing request isn’t working)

1. Open a request-

*XHR object* has a method/function called *open* which prepares the browser for sending the request, BUT does not send request yet.

Accepts 2 arguments: HTTP method, URL (where the request is going- i.e file or web server)

**xhr.open(‘GET’, ‘sidebar.html’); //loads this file to the page (‘POST’ to send data)**

1. Send the request-

**xhr.send(); //don’t need to provide send method with any info since we are just requesting html page. But if sending info to server like form data, can pass that data to send method.**

1. Create a button where, when clicked, makes the ajax request by running sendAJAX function:

**<button id="load" onclick="sendAJAX()">Bring it!</button>**

Place the ajax request inside function which also makes button disappear on request:

**function sendAJAX() {**

**xhr.send();**

**document.getElementById(‘load’).style.display = “none”;**

**}**

Send data along with AJAX request (‘POST’ method):

* ‘GET’ method sends data in URL so not safe with sensitive information & there’s a limit to the number of characters allowed in URL
* The part after “?” is the query string at the end of the URL to send additional info to web server to customize its response back. Usually used to search database for info.

Query string- made up of one or more name/value pairs. Separate each name/value pair by ampersand (&) symbol.

[http://website.com/employees.php?**lastname=Jones**](http://website.com/employees.php?lastname=Jones)

* ‘POST’ sends data in body of the request instead of in URL. Need to setup a special header for the request which is an instruction sent to server telling it what kind of data it should expect.

Two common data interchange formats (to exchange data between 2 computers): XML (extensible markup language, similar to HTML) & JSON

* Parsing- the process of breaking a file up into easily accessible parts
* Using XML with Javascript is not as easy that involves analyzing and parsing the document.
* Instead, there is JSON- more Javascript-like data format which has become the popular way to exchange data using AJAX. Two ways to format- array notation or object notation.
* Javascript Object contains key/value pairs
* JSON- Each property must be wrapped in double quotes (not JS objects)
* To make JSON useful for JS, it needs to be parsed- converted from a string (plain text) to JS

**var xhr = new XMLHttpRequest();**

**xhr.onreadystatechange = function () {**

**if(xhr.readyState === 4) {**

//converts the string into an array(JS object) for JS (now we go thru array to display to HTML)

**let employees = JSON.parse(xhr.responseText);**

**}**

**}**

**xhr.open('GET', 'data/employees.json');**

**xhr.send();**

AJAX is limited by a web browser’s same origin policy which controls how Javascript can access content from a web server. In general, you can use AJAX to communicate from one page to another on same web server but not to other web servers. Also won’t allow differences in secure vs insecure servers (http vs https), different port numbers, or different hosts (www vs db) but still have same domain names.

Ways to circumvent same origin policy (to make AJAX requests across domains):

1. Create a web proxy. Web servers aren’t limited by same origin policy so web servers can request data from servers at other domains. So you can set up script on your web server which asks for information from another web server- use AJAX to make a request.
2. Use JSONP (JSON with padding) which relies on the ability to link to Javascript files on other domains (browsers do allow many types of cross domain links). Instead of using AJAX to contact another web server, you load JS file from another site which contains the info you need.

* Example: link to photos, CSS files, JS files on other websites -which is how CDN’s, content delivery networks, like Jquery works. CDN hosts files that other sites can use. (saves space as you don’t need to put library on your own server)

1. CORS (Cross-origin Resource Sharing) which requires some setup on the server which allows it to accept requests from other domains.

FETCH API

*Fetch is a browser API for loading texts, images, structured data, asynchronously to update an HTML page. It is built into the browser so you don’t need to load external library.*

*It's a bit like the definition of Ajax! But fetch is built on the Promise object which greatly simplifies the code, especially if used in conjunction with async/await. One could also use promises with the XMLHttpRequest object, but this, added to Ajax's already more complex code would produce a much larger program.*

Promises have 3 state:

* Pending- value is being prepared
* fulfilled or resolved (.then/.catch)
* rejected- error obtaining the value (there is a reject() method)

Promise object’s methods:

**.then** takes in 2 arguments- result handler (when promise fulfilled) and reject handler.

* Instead of a second argument for reject handler, can use another method called **.catch** where you place the reject handler in.

Example:

**let calculatePromise = new Promise( function(resolve,reject) {** ‘action’ **} );**

Inside ‘action’, if you have something returning like ‘console.log(1+1)’, you can put **resolve(1+1)**. As for an AJAX request- **resolve( JSON.parse(xhr.responseText) );**

**function addTwo(value) {return value + 2};**

**function printFinal(nextValue) {return nextValue};**

Resolve handler: **calculatePromise.then( addTwo );**

The value returned above can be passed again to the resolver of the next promise.

**calculatePromise.then( addTwo ).then( printFinal );**

Can create another promise object **calculatePromise2** with different action ( **resolve(1+2)** ) and see it run asynchronously.

**FETCH API**- uses JS promises to handle results returned from server.

1. To make a request, you use the global fetch method that takes one argument, the path to the resource to fetch. Also optional second argument where you can provide your credentials:

**fetch( ‘myapi.com/endpoint’ )**

1. **fetch** call returns a promise object in which you chain **.then** methods which return promises of their own. These methods get executed sequentially once the previous promise is fulfilled.

* **fetchData( ‘https://dog.ceo/api/breeds/image/random’ )** //using fetch helper func.

**.then( data => generateImage(data.message) )**

//do something with the JSON data (i.e. iterate over and add to HTML page). Here, we access the message property of JSON object (random dog image link)

* helper function (keeps code cleaner and modular):

**function generateImage(data) {** //data represents data.message when passed in

**const html = `**

**<img src=’${data}’ alt>**

**<p>Click to view images of ${select.value}s</p>** //select is drop down menu

**`;**

**card.innerHTML = html;**

**}**

* Add another fetch method to another endpoint to retrieve list of breeds to include as options under select menu:

**fetchData(‘https://dog.ceo/api/breeds/list’)**

**.then( generateOptions(data.message) )**

* helper function (for second fetch call to get list of breeds):

**function generateOptions(data) {**

**const options = data.map(item => `** //create option element for each breed

**<option value=’${item}’>${item}</option>**

**`).join(‘’);** //to get rid of commas showing up in html

**select.innerHTML = options;** //add options to select element

**}**

* Another fetch request to change image when another breed option is selected under select input:

**function fetchBreedImage() {**

**const breed = select.value;**

**const img = card.querySelector(‘img’);**

**const p = card.querySelector(‘p’);**

//get image related to breed selected

**fetchData(‘https://dog.ceo/api/breed/${breed}/images/random’)**

**.then(data => {**

**img.src = data.message;**

**img.alt = breed;**

**p.textContent = `Click to view more ${breed}s`**

**})**

**}**

//event listener on select element to change image

**select.addEventListener(‘change’, fetchBreedImage);**

**Create reusable fetch function** (as you often run multiple fetch requests):

* Runs fetch request, parses and returns JSON, handle errors, check response status, etc.
* **catch** method to handle errors (no response from server) to deal with rejected cases

**function fetchData(url) {** //wrapper function around fetch

**return fetch(url)**

**.then(checkStatus)** //check if request is ok, status 200 (if error, goes to catch)

**.then( res => res.json() )** //response object has several methods on it

do a console.log to see properties of it

**.catch( error => console.log(‘There was a problem’, error) )** //handle errors

**}**

//common practice is to also check for failed http responses (not status 200 or ok property not equal to true) with helper function:

**function checkStatus(response) {**

**if(response.ok === true) return Promise.resolve(response);**

**else return Promise.reject( new Error(response.statusText) );**

//reject promise which activates catch call

//pass in error object (reason for the reject) with error description passed into error obj.- *then* all this passed to catch method

**}**

**Promise.all**-

* put all fetch requests in one promise
* accepts an iterable like an array or string
* waits for all promises to resolve before continuing on (needs to succeed)
* returns an array of values (JSON data)
* Good for fetching similar data such as user profile and lists of posts from user

//used for example purposes only, but in reality should separate them out

1. **Promise.all([**

**fetchData(‘https://dog.ceo/api/breeds/list’),**

**fetchData( ‘https://dog.ceo/api/breeds/image/random’ )**

**])**

**.then(data => {**

**const breedList = data[0].message;**

**const randomImage = data[1].message;**

**generateOptions(breedList);**

**generateImage(randomImage);**

**})**

The default HTTP method for making fetch requests is GET. You can make requests (like **POST**, DELETE and PUT) by passing additional information to **fetch()**.

(in this example, post name and comments from input boxes on click of submit button)

**form.addEventListener(‘submit’, postData);**

**function postData(e) {**

**e.preventDefault();** //prevent default submit button behavior

**const name = document.getElementById(‘name’).value;**

**const comment = document.getElementById(‘comment’).value;**

//api endpoint to test (also preferred to put 2nd argument in separate variable)

let config = **{**

**method: ‘POST’,** //type of request

**headers: {**

**‘Content-Type’: ‘application/json’,** //inform server, data is json

**},**

**body: JSON.stringify( {name: name, comment: comment} )** //post information put here. Covert JS object to JSON string (is ES6, if key and value are same, just put key only)

**}**

**fetch(‘https://jsonplaceholder.typicode.com/comments’, config)**

**.then(checkStatus)**

**.then( res => res.json() )**

**.then( data => console.log(data) )** //logs object with name & comment

**}**