AJAX

Asynchronous Javascript

Goal: explore async http requests using AJAX and see how it works under the hood.

1. AJAX stands for…?
   1. Asynchronous Javascript And XML
   2. XML – Extensible Markup Language
      1. Just carries data
      2. Whereas HTML displays data
      3. Now we use JSON to carry data.
   3. All-encompassing thing that includes all the asynchronous stuff like get and fetch.
2. XML requests under the hood
3. Actually under the hood
4. POSTing data without AJAX?
   1. We would have to specify within the HTML that the method of whatever form is a “POST” method
   2. Once this request is sent, the page will freeze because it is awaiting a response from the server.
      1. Synchronicity
   3. Once the data returns, the page will refresh and reload from the top down.
5. With AJAX
   1. (Async JS and XML)
   2. It allows us to write JS on our front end that can interact with our back-end.
   3. We can send requests and JS can continue running while the servers fulfill our requests.
   4. This makes a single page app (SPA) possible.
   5. A “single page app”
      1. For example, Facebook.
      2. In order to populate, say, a friends list, it would have to send the request
      3. And if the script were synchronous (written without AJAX), the page would freeze up while it retrieved the friends list, then the entire page would have to refresh.
      4. Every time it awaits data from an external library, it would have to stop everything on the page and reload the page.
   6. And it could be used to only update the post.
   7. This is an example of a single-page-app;
      1. All the functionality of the facebook “app” is contained on a single page.
      2. But synchronous code would require us to reload the page twice or more to obtain all info (multiple page)
   8. Synchronous code is “blocking”
      1. The thread can be blocked while something else is being done.
   9. With async, the code is non-blocking; we can set it aside (in the event loop) to be performed by something else (the web API)
   10. AJAX uses JS to make HTTP requests to the server
       1. and JS schedules the request to be handled by an external API
   11. We should start thinking about how a user experiences changes when working in single page apps.
       1. What is the history? The client state/information? Visual feedback?
6. How do we use AJAX?
   1. At the core of an AJAX request is an XMLHTTPRequest object.
   2. This allows us to make an HTTP request and apply callbacks to the response.
7. Example code of using XML to send a GET and POST request.

*//Example of sending a GET request using XML.*

const loadDataUsingGET = () => {

*//create new instance of XHR request.*

const xhttp = new XMLHttpRequest();

*//ready state keeps track of the status of our request.*

*// 0 - uninitialized*

*// 1 - loading (open was invoked)*

*// 2 - loaded 9send was invoked)*

*// 3 - interactive (not finished)*

*// 4 - complete*

*// onreadystatechange is a native method, maybe?*

*// it runs 4 times - each time the readystate is changed from 0 to 1, 1 to 2, etc.*

xhttp.onreadystatechange = () => {

*// when the request is complete, and the status is okay*

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

*// we can change the contents of the HTML*

document.getElementById("demo").innerHTML = xhttp.responseText;

}

*// and we can add catch cases for errors.*

*// Do this when writing Async code. No matter what.*

*//*

}

*// now we can configure a new HTTP request using the open method, passing in args.*

*// "open" a request;*

*// and it sends it to the URL*

*// and if we want the request to be async, we should append "true"*

*// and "true" is a default parameter.*

xhttp.open('GET', 'https://example.com/?sortBy=name', true);

*// then we have to send the HTTP request.*

xhttp.send()

}

const sendDataUsingPOST = () => {

*// create a new XML Http request that we will start to configure.*

const xhttp = new XMLHttpRequest();

*// current readystate is 0; no actual request has been initialized (this happens with the "open" method)*

xhttp.onreadystatechange = () => {

*// and we give it a functionality;*

*// only given*

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

console.log(JSON.parse(xhttp.responseText))

}

}

*// open up a new request and set it to a "POST" request*

*// readystate goes from 0 -> 1*

xhttp.open("POST", "https://exmample.com", true)

*// we set a header for the data to provide the server context for the request. In this case, we are specifying the content type.*

xhttp.setRequestHeader("Content-Type", "application/json")

*// then we have to send the post request.*

*// in this case, the data that we are passing into "send" is called the "request body.""*

*// readystate goes from 1 -> 2*

xhttp.send(JSON.stringify({ fname: "Winona", lname: "Ciancarelli" }))

*// readystate goes from 2 -> 3? Maybe?*

}

1. The above is how we **used** to do things.
2. ***We should NOT use XML http requests. We should use “fetch.”***

*// making a "GET" request using fetch.*

fetch("http://example.com")

.then((resultFromFetch) => {

resultFromFetch.json()

})

.then((jsonedData) => {

console.log(jsonedData)

})

.catch((err) => {

console.log("error fetching data", err)

})

*//This returns a "Promise" object that will store the result of the fetch.*

*// and THEN once it's been completed, we will take the resultFromFetch and do something to it*

*// and THEN once that's been completed, we will log it to data*

*// or we will catch any errors. PLEASE PLEASE PLEASE make sure you add catch cases whenever you use promise chains.*

*// POST request using "fetch"*

fetch({

method: "POST",

url: "https://example.com"

body: JSON.stringify({ fname: Launchpad, lname: Ciancarelli })

})

.then((data) => {

data.json()

})

.then((data) => {

console.log(data)

})

.catch(err => {

console.log('error POSTing data: ', err)

})

1. Cross-site Request Forgery (CSRF)
   1. A victim logs into a bank
   2. The bank assigns a validation token to the victim.
   3. A hacker may send a request to the victim, forged as legitimate comm from the bank.
   4. The sender forwards the request to the bank.
   5. Forged request is executed by the bank using previously assigned validation token.
2. Same-origin policy
   1. Critical security mechanism
   2. Restricts how a document or script loaded from one origin can interact with a resource from another origin.
   3. Origin is defined by a combination of:
      1. Protocol
         1. Http vs Https
      2. Domain
         1. Api.mydomain.com vs mydomain.com are different
      3. Port
         1. Localhost:3000 vs Localhost:8000
   4. Often,
      1. cross-origin writes are typically allowed
      2. cross-origin embedding is typically allowed
      3. cross-origin reads are typically disallowed.
         1. You cannot read AJAX or Fetch calls.
   5. Servers can prevent CSRF by assigning tokens
   6. Or implementing CORS.
      1. Cross-origin resource sharing
      2. This is a relaxation of the same-origin policy, which allows JS on a webpage to consume a REST API served from a different origin.