Week 3, Lesson 5 Readings

***Chapter 13: AJAX***

The web of computers known as the internet can be separated into two parts: clients and servers.

JavaScript was originally designed to be a client-side scripting language. AJAX allows JavaScript to request resources from a server on behalf of the client.

Server languages such as PHP, Ruby, Jode.js, or .NET. These are known as server-side languages.

**Asynchronous**

When a request for data is sent, the program doesn’t have to stop and wait for the response. It can carry on running, waiting for an event to fire when a response is received. By using callbacks to manage this, programs are able to run in an efficient way, avoiding lag as data is transferred back and forth.

**JavaScript**

JavaScript was always considered a 'front-end' language, not used to communicate with the server. Ajax enabled JavaScript to send requests and receive responses from a server, allowing content to be updated in real time.

**XML**

When the term Ajax was originally coined, XML documents were often used to return data. Many different types of data can be sent, but by far the most commonly used in Ajax nowadays is JSON, which is more lightweight and easier to parse than XML. (Although it has never really taken off, the term [Ajaj is sometimes used to describe the technique.)](http://en.wikipedia.org/wiki/AJAJ" \t "_blank) JSON also has the advantage of being natively supported in JavaScript, so you can deal with JavaScript objects rather than having to parse XML files using DOM methods.

Fetch API

Fetch() only has one mandatory argument, which is the URL of the resources you wish to fetch.

Example:

fetch('https://example.com/data')

.then( // code that handles the response )

.catch( // code that runs if the server returns an error )

Catch() statements at the end of the code will allow us to deal with any errors that may occur.

Response Interface:

“ok” = property that checks to see if the response is successful.

“status” = is based off the “HTTP status code”.

“200” = will usually be the response if it was successful.

“201” = if a resource was created.

“204” = means the request is successful but no content was returned.

The “ok” property will return “true” if the “status” property is between “200” and “299”.

“if” block checks if the request was successful and throw an error otherwise.

Example:

const url = 'https:example.com/data';

fetch(url)

.then((response) => {

if(response.ok) {

return response;

}

throw Error(response.statusText);

})

.then( response => // do something with response )

.catch( error => console.log('There was an error!') )

* headers – A Headers object (see later section) containing any headers associated with the response
* url – A string containing the URL of response
* redirected – A boolean value that specifies if the response is the result of a redirect
* type – A string value of 'basic', 'cors', 'error' or 'opaque'. A value of 'basic' is used for a response from the same domain. A value of 'cors' means the data was received from a valid cross-origin request from a different domain. A value of 'opaque' is used for a response received from 'no-cors' request from another domain, which means access to the data will be severely restricted. A value of 'error' is used when a network error occurs.

Redirect() = a method to us to redirect to another URL.

fetch(url)

.then( response => response.redirect(newURL)); // redirects to another URL

.then( // do something else )

.catch( error => console.log('There was an error: ', error))

Text() = method takes a stream of text from the response, reads it to completion and then returns a promise that resolves to a USVSting object that can be treated as a string in JavaScript.

fetch(url)

.then( response => response.text() ); // transforms the text stream into a JavaScript string

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

.catch( error => console.log('There was an error: ', error))

File Responses:

The blob() method is used to read a file of raw data, such as an image or a spreadsheet. Once it has read the whole file, it returns a promise that resolves with a blob object.

fetch(url)

.then( response => response.blob() ); // transforms the data into a blob object

.then( blob => console.log(blob.type) )

.catch( error => console.log('There was an error: ', error))

This example is similar to the text example above, but we use the blob() method to return a blob object. We then use the type property to log the MIME-type to log what type of file we have received.

JSON Responses:

JSON is the most common format of AJAX responses.

Json() = method that transforms a stream of JSON data into a promise that resolves to a JavaScript object.

fetch(url)

.then( response => response.json() ); // transforms the JSON data into a JavaScript object

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

.catch( error => console.log('There was an error: ', error))

A constructor function is used to create a new Request object.

const request = new Request('https://example.com/data', {

method: 'GET',

mode: 'cors',

redirect: 'follow',

cache: 'no-cache'

});