Learning Outcome 8: Use Ajax to dynamically update web pages

# Learning Step 8.1: Describe Ajax and how it works with web pages

Ajax stands for “Asynchronous JavaScript and XML”. What this means is that a JavaScript program on our web page fetches an XML file from the server, parses the XML file, and changes the website based on what’s in the XML file. It does this asynchronously, which means that the website visitor can continue to interact with the web page while the page is waiting for a response to that request.

But XML files are rarely used any more with Ajax. The JSON format is smaller and more efficient, so almost all Ajax is now done with JSON files rather than XML files.

**Server side**

For example: <http://www.someserver.ca/provideXmlData.php?custid=123> would contact the server with a query string of custid=123, run a program on the server to process that data (usually using a database), and return a JSON file for the web page to work with.

**Activity:** Go back to the [www.google.ca](http://www.google.ca) page. What is the name of the page that the HTTP request is sent to? What is one the parameter names that is sent as part of that request?

**Client side**

The web browser does these requests in the background – it automatically sends off the request and allows other stuff to happen on the page while it’s waiting for the response. Actually, it doesn’t wait for a response, but it sets a JavaScript function that gets called when the response is received. When the response is received, this function usually parses the JSON file (or the HTML file or XML file or text file or whatever), then updates the currently displayed page with information from the response.

# Learning Step 8.2: Use Ajax calls to load data

All modern browsers support an object called as “XMLHttpRequest” – this provides a number of methods and properties that we can use to make our Ajax call. Don’t be fooled by the “XML” in the name; we can fetch any kind of page.

To set up an Ajax call, we have to go through the following steps:

* Create an XMLHttpRequest object
* Configure the object, telling it what URL to fetch and how to fetch it (a GET or POST request)
* Set up the event handler on the object to handle the response
* Send off the request
* Go off and do something else until the event handler is executed when the response is received

Note: remember that a security restriction of JavaScript in the web browser is that we can’t make arbitrary connections. Because of this, we are generally limited to making Ajax requests to pages on the same server that the original page was loaded from (unless CORS permissions are set up).

The XMLHttpRequest object includes a state machine that handles the processing of the request. As the request is being processed, the object goes through the following states:

* The HTTP request has been initialized
* A connection to the server has been established
* The server has received the request, but hasn’t begun processing it yet
* The server has finished processing the request, but hasn’t the response back yet
* The response has been received from the server

Each time the object moves to a new state, a “ready state change” event is fired. Typically, we set up our event handler to ignore all states except the last one, because we usually don’t care about the other states.

**Practice:** This practice doesn’t use Ajax or the Fetch API, but it will give us an idea for what we need to do. Grab a copy of the file **instructorinfo.html** – it will display the information for an instructor. For this practice, add a click handler to the first button (updateBtn) so that when the button is clicked, the name, office, and phone number of a CST instructor are displayed in the appropriate span elements. Use jQuery to add the functionality.

Instructor information:

* Name: Joseph Herbert
* Office: Room 348
* Phone: 306-567-5309

**The Fetch API**

The Fetch API provides a method called **fetch()** that we can use to make a request. The **fetch()** method takes an argument which contains the URL of the page or file that you want to request, and an optional argument setting up any custom initialization settings. It returns a *promise* which is fulfilled once the response is available. A promise represents the eventual completion (or failure) of an asynchronous operation. You can think of it as being like a timer which eventually is called when a certain time is reached. In this case, a promise is returned when a file is retrieved from the fetch request.

See “MDN fetch” for <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API> and <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch> and “MDN promise” for <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise> and <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises>.

Promises have the **.then()** method called on them, which has two optional arguments: a callback for success and a callback for failure. We will only add the callback function for success (which may include successfully finding out there is no page!). Add the following code in your **window.onload** function:

document.getElementById("btnUpdateFetch").onclick = function() {  
 *fetch*("data/oneinstructor.json").then(function(response) {  
 window.alert(response.ok);  
 });  
};

**Practice:** Do the following:

1. Add another button to your HTML page. Give the button an ID of “btnUpdateAll” and the text of “Update All Instructors”.
2. Add a click event handler to the new button that will use an fetch call to obtain a JSON document named “allinstructors.json” (from the **data** folder)
3. When the Ajax request has been completed, display an alert box that says **true** or **false** for whether or not the file was fetched.

Make sure that you are running the page from the WebStorm local server or it will not transfer the JSON file.

**Throwing and catching an error**

If a promise is not resolved successfully, you may want to throw an error and catch the error to deal with it. To do so, you add a **throw new Error(*message*);** call, and then you *chain* a **.catch** method at the end of the promise *chain*, which is a list of method calls to deal with the fetch request.

**Practice:** Add the code to throw a new error and catch the error into the event handler for all instructors.

# Learning Step 8.3: Create a web page that is dynamically updated based on retrieved data

So far, we have used the **response.ok** and **response.status** properties of the response to the fetch. Now, we would like to get the actual data and dynamically update our web page! To do so, we chain a **.then** call based on the response’s **.json()** method, which converts the JSON response into an object. Update your code as follows:

*fetch*("data/oneinstructor.json").then(function(response) {  
 if (response.ok) {  
 return response.json();  
 }  
 throw new Error("Network response was not OK: " + response.status);  
}).then(function(jsonObject){  
 window.alert(jsonObject.name);  
}).catch(function(error) {  
 console.log("Problem with the fetch operation: " + error.message);  
});

**Practice:** Update your updateAllInstructors method so that, rather than just displaying an alert message indicating that the file was successfully retrieved, you should display multiple alerts, one for each element of the array. The alert message should display the text of the name property.

**Practice:** Rather than displaying alert boxes for all the instructors, we want to update the Name, Office, and Phone number spans with information on all the instructors. Separate the values with commas. Your results should look something like:

Name: Bryce Barrie, Michael Grzesina, …  
 Office: Room 240A.3, Room 240A.5, …  
 Phone: 306-659-4369, 306-659-4231, …

**Practice:** With the web page open and all of the instructors displayed, add an additional instructor to **allinstructors.json** and without reloading the web page, redisplay the instructors.

**Retrieving information from external servers using CORS**

Generally, web pages have a same-server policy that requires us to get files/information from the same domain as the web page was served from. (See <https://en.wikipedia.org/wiki/Same-origin_policy>)

We can also retrieve information from external servers if they are set up to allow Cross-Origin Resource Sharing (CORS). Example: Google “free json api” (<https://github.com/public-apis/public-apis>)

**Activity:** Get a random dog image or video from **random.dog**’s JSON API

**Practice:** Create a web page that has a button that says “Your answer is…”. When the button is clicked, the web page should display the text answer from the Yes/No JSON API at <https://yesno.wtf>, and display the image from the Yes/No API below the text answer. The API address is just <https://yesno.wtf/api>. *Bonus practice:* add a query string to the API request to force the answer to be “maybe”.

**Practice:** Create a web page that will load the to-do list from <https://jsonplaceholder.typicode.com/todos>/ and display only the to-do items associated with user id 1.

You can also try other APIs on this site such as <https://jsonplaceholder.typicode.com/posts/>

If you want to try additional practices, look at some of the other free JSON APIs such as the Meal Database (<https://www.themealdb.com>).

**Other methods of getting data from external sources**

jQuery provides a **$.getJSON()** method that also makes it easy to get JSON data from external sources using Ajax. There’s actually several different Ajax methods in jQuery, but **$.getJSON()** is probably the easiest to use. In the simplest case, **$.getJSON()** takes two arguments: a string containing the URL of the page that you want to request, and a callback function that is called when the Ajax request has been completed.

The callback function takes two parameters. The first one is an object that contains the JSON object or array that was returned (like the object returned by **response.json()**). The second is a string that indicates the status of the Ajax call – on success, it will contain "success".

Try using it with our **instructorinfo.html.**