web222

WEB222 - Week 12

Suggested Readings

- AJAX Guide
- Working with JSON
- Using XMLHttpRequest

AJAX

AJAX is a term coined in 2005 by Jesse James Garrett that refers to an approach to web development that uses dynamic requests to a server to update portions of a page at runtime. Today, the method is so common that it's hard to talk about it not existing. But at the time, it was a game changer.

AJAX stands for Asynchronous JavaScript and XML, which is based on a group of web technologies: HTML (and at the time XHTML), CSS, JavaScript, the DOM, XML, JSON, and a web API called XMLHttpRequest, or just XHR for short.

Before 2005, web browsers lacked a lot of the modern features we take for granted today. Web servers were used to build and serve all (or most) aspects of a web page. Making changes on the page meant a full request/response trip to and from the server, in order to update content. The entire page had to be reloaded for anything of significance to change.

Today we expect "real-time" data to be a part of our web browsing experience. Consider a site like GMail or Google Maps. If we want to see messages in another folder, or navigate to another city, we expect to be able to do that without having to reload the entire page. AJAX makes this possible.

Instead of modifying the entire page (DOM), we instead make background requests for data from servers, and then use that data to update the page's contents live via the DOM's APIs.

Understanding AJAX's "A" (Asynchronous) and "J" (JavaScript)

As we know from previous discussions, web browsers use HTTP/HTTPS to send requests to web servers, which build replies and send back responses (HTML, CSS, images, fonts, JavaScript, etc).

While we don't want to have to reload the entire page in order to get updates from the server, we would like to be able to leverage this communication pattern from within the running page: we need a way to make HTTP requests, wait for responses from the server, and then do something with the data.

Browsers provide a mechanism for doing this in the form of the `XMLHttpRequest` Object, or XHR for short. An XHR object let's us create and send HTTP requests to a server, and get back data responses in various forms (XML, HTML, JSON, text, binary, etc.)

Our XHR requests happen in the background, asynchronously (without blocking the main UI thread in the browser), so user's can continue to work and interact with the page while we wait for a response.

Finally, we work with XHR via JavaScript code. Let's look at a very basic example:

```
// 1. Create a new instance of an XMLHttpRequest Object using its constructor
var xhr = new XMLHttpRequest();

// 2. Define an event handler for the `load` event, which happens when data arrives
xhr.onload = function() {
    // 3. Get the data from the `xhr` object's `responseText` property
    var data = this.responseText;
    console.log('data arrived', data);
};

// 4. Create an HTTP GET request to the given URL
xhr.open("GET", "http://example.com");

// 5. Send the HTTP request to the server, and wait asynchronously for the reply
xhr.send();
```

Example 1: Current Bitcoin Value in USD

To demonstrate a real-world example of what we've been discussing, let's build a simple example. Imagine we want to create a web page that includes information about the current market value of Bitcoin, the most famous of the blockchain-based cryptocurrencies.

The website https://www.blockchain.com provides a number of web services we can use to get this information. In particular, we'll use https://blockchain.info/q/24hrprice, which gives the price in US dollars over the past 24 hours.

Here's an (simplified) example of what it sends when we make an HTTP request:

```
HTTP/2 200
date: Sun, 02 Dec 2018 22:48:35 GMT
content-type: text/plain; charset=utf-8

4200.82
```

In the above response, we have HTTP headers, a blank line, and then the data: `4200.82` .

We want our web page to include this information, but then automatically update it every minute by requesting the current value again over HTTP. Here's one way we could do it.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Bitcoin Value</title>
</head>
<body>
    <p>Current Bitcoin value: <span id="bitcoin-value"></span></p>
    <script>
        function updateBitcoinValue(newValue) {
            // Update the <span> with the new value we get from the server
            var span = document.querySelector("#bitcoin-value");
            span.innerHTML = newValue;

            // Every minute, get the new value and update the page
            var oneMinute = 60 * 1000;
            setTimeout(getCurrentValue, oneMinute);
        }

        function getCurrentValue() {
            var xhr = new XMLHttpRequest();
            var url = "https://blockchain.info/q/24hrprice?cors=true";

            // If/When the request returns successfully, get the value and update DOM
            xhr.onload = function() {
                // Format the raw text we get from the server into a currency string
                var response = this.responseText;
                var currentValue = `${response} (USD)`;
                updateBitcoinValue(currentValue);
            };

            // If the request fails, and we get an error, update the page with an error message
            xhr.onerror = function(e) {
                updateBitcoinValue("unknown (error, unable to get current value)");
            };

            // Open a GET request to the Blockchain API
            xhr.open("GET", url);

            // Send our HTTP request to the server, and wait for a response
            xhr.send();
        }

        window.onload = function() {
            getCurrentValue();
        }
    </script>
</body>
</html>
```

See bitcoin.html for an online version.

By separating the data into a separate web service, it's possible for various applications to all share it, and use it in different ways.

Working with Data: JSON and XML

Our previous example used a very simple data format: a single number. Often we'll need to work with more complex data, which includes both numbers and text, lists, and complex hierarchies. To do this, we need a format that allows us to serialize (i.e., turn into strings) data structures like `Array`s, `Object`s, etc.

The two most popular formats for data exchange on the internet are the JavaScript Object Notation (JSON) and the Extensible Markup Language (XML).

We're going to focus mainly on JSON, but it's good to also know about XML. At one point, a lot of the techniques we will discuss were done with XML (and many languages and services still use it). However, much of the internet has standardized on JSON as a data exchange format.

Let's look at each in turn. First, consider the following food product data:

```
1) Name: Apple
   Price per Pound: $1.29
   Location: Aisle 3

2) Name: Carrots
   Price per Pound: $0.46
   Location: Aisle 2
```

Here's how that data might look using XML:

```xml
<products>
    <product>
        <name>Apple</name>
        <price currency="CAD">1.29</price>
        <aisle>3</aisle>
    </product>
    <product>
        <name>Carrots</name>
        <price currency="CAD">0.46</price>
        <aisle>2</aisle>
    </product>
</products>
```

Look familiar? XML and HTML are both markup languages. In XML, it's possible for us to create our own tags and document structure (i.e., XML Schema). You can think of HTML like an instance of XML, which is what XHTML was trying to do.

Before we look at JSON, let's look at that same data in JavaScript, using an Object Literal:

```js
var products = [
    {
        name: "Apple",
        price: {
            currency: "CAD",
            value: 1.29
        },
        aisle: 3
    },
    {
        name: "Carrots",
        price: {
            currency: "CAD",
            value: 0.46
        },
        aisle: 2
    }
];
```

Here we have two `Object`s in an `Array`. Our `Object`s use `String`, `Object`, and `Number` types to represent the data.

Finally, let's format the same data in JSON:

```
[
    {
```

```
        "name": "Apple",
        "price": {
            "currency": "CAD",
            "value": 1.29
        },
        "aisle": 3
    },
    {
        "name": "Carrots",
        "price": {
            "currency": "CAD",
            "value": 0.46
        },
        "aisle": 2
    }
]
```

Looks familiar, doesn't it? You'll be glad to know that since you already learned JavaScript Object Literals, you already learned 95% of JSON format at the same time.

JSON and JavaScript Object Literals are very similar, but there are some differences. JSON is a subset of JavaScript Object Literal notation:

- All keys must be double-quoted strings: `"key"` vs. `key`
- You don't put comments in JSON
- You can't include function expressions in JSON, only data types:
  - `String`
  - `Number`
  - an (JSON) `Object`
  - an `Array`
  - boolean `true` or `false`
  - the value `null`

JavaScript includes built-in code for converting to/from JSON strings:

```
var products = [
    {
        name: "Apple",
        price: {
            currency: "CAD",
            value: 1.29
        },
        aisle: 3
    },
    {
        name: "Carrots",
        price: {
            currency: "CAD",
            value: 0.46
        },
        aisle: 2
    }
];
// 1. Turn products Object into JSON string
var json = JSON.stringify(products);
// json now equals '[{"name":"Apple","price":{"currency":"CAD","value":1.29},"aisle":3},{"name":"Carrots","price":{"currency":"CA

// 2. Turn JSON string back into a JS Object
products = JSON.parse(json);
// products is now an Object
```

> NOTE: `JSON.parse()` will `throw` an error if the string isn't properly formatted JSON, so it's good to wrap your call in a `try...catch` block.

Example 2: the Dog API

For our next example, let's work with another web service, but this time one that returns more complex data in the form of JSON.

The Dog API is a free web service that uses data from the Stanford Dogs Dataset. This dataset contains images and information about 120 breeds of dogs, and is used for machine learning and artificial intelligence training.

There are a number of endpoints we can use with this API, but we'll focus on these:

- https://dog.ceo/api/breeds/list/all - get a JSON formatted list of all breeds and sub-breeds
- https://dog.ceo/api/breed/hound/images/random/3 - get a JSON formatted list of image URLs for hounds, returning 3 (we can ask for more or less) In other words the URL works like this: `https://dog.ceo/api/breed/{name-of-breed}/random/{number-of-images-to-return}`

Our goal is to do the following:

1. Create a simple web page with an HTML `<form>`
2. Use AJAX techniques to dynamically load all dog breeds into a `<select>` in our form
3. Users can specify how many images they want to load: 1 to 100.
4. When the user selects a breed and clicks a `<button>`, we'll request the JSON list of images
5. Once we get the list of image URLs, we'll start creating `<img>` elements in our page to show those dogs

See the completed code in dogs.html and dogs.js. We'll discuss snippets of the code below.

1. Create a form

Let's start with a basic `<form>`:

```
<form id="dogs-form" action="#">
    <select id="breeds" name="breeds"></select>
    <input type="number" min="1" max="100" value="5">
    <input type="button" id="btn-load" value="Show me dogs!">
</form>
```

Our form is very simple. Notice that it contains no `<option>` elements for the dog breeds. We will load these dynamically once the page is loaded. We include a textbox for entering a number of images to show (default is `5`), and provide a `<button>` to click.

2. Dynamically load dog breeds into a drop-down

Next we need to load our dog breeds from the Dogs API. We'll do that when the page finishes loading, and the DOM is fully created:

```
window.onload = function() {
     loadDogBreeds();
};
```

Our `loadDogBreeds` function needs to create an XHR request to the Dogs API, and parse the JSON we get back:

```
function loadDogBreeds() {
    // See https://dog.ceo/dog-api/documentation/
    var url = "https://dog.ceo/api/breeds/list/all";
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
        var response = JSON.parse(this.responseText);
        var breedList = extractBreedList(response);
        updateBreedList(breedList);
    };
    xhr.open("GET", url);
    xhr.send();
}
```

When our request comes back (i.e., `xhr.onload`), we'll get a JSON string that looks something like this:

```
"{"status":"success","message":{"affenpinscher":[],"african":[],"airedale":[],"akita":[],"appenzeller":[]}}";
```

If we `JSON.parse()` that string, we'll get an `Object` that looks like this:

```
var response = JSON.parse(this.responseText);
/*
{
    status: "success",
    message: {
        affenpinscher: [],
        african: [],
```

```
        airedale: [],
        akita: [],
        appenzeller: [],
        ...
    }
}
*/
```

This data has two main parts:

1. a `status` message, that tells us the server was successful in doing our query
2. a `message` body, which is itself an `Object` of key/value pairs, with the breed name and sub-breeds (if any) in an `Array`.

To get all the dog breeds as a list (i.e. `Array`), we need to extract the `message` property, then call `Object.keys()` on its value to create an `Array` out of all the names:

```
var breedList = Object.keys(response);
/*
['affenpinscher', 'african', 'airedale', 'akita', 'appenzeller', ...]
*/
```

Next we need to take this list of breed name `String`s, and create `<option>` elements that we can dynamically add to our form's `<select>`:

```
// Get a reference to our <select>
var select = document.querySelector("#breeds");

// Given a breed name "beagle", return an <option value="beagle>beagle</option>
function createBreedOption(name) {
    var option = document.createElement("option");
    option.value = name;
    option.innerHTML = name;

    return option;
}

// Loop through each breed name in our Array, call createBreedOption()
// and append the <option> element to our <select>
breedList.forEach(function(breed) {
    var breedOption = createBreedOption(breed);
    select.appendChild(breedOption);
});
```

Our web page now has a drop-down list with all 120 dog breeds.

3. Get dog breed image URLs when the user selects a breed

When the user selects a breed from our list and clicks a `<button>`, we need to make another HTTP request to the server. This time we need to get a list of image URLs for the chosen breed.

First, we need an event handler for the click:

```
var btnLoad = document.querySelector("#btn-load");

btnLoad.onclick = function(e) {
    var breed = document.querySelector("#breeds").value;
    loadBreedImages(breed);
};
```

Our `loadBreedImages` function is nearly identical to `loadDogBreeds`, but we use the `breed` name as part of the URL:

```
function loadBreedImages(breed) {
    // See https://dog.ceo/dog-api/documentation/breed
    // Use the imageCount and breed variables to create our URL
    var imageCount = document.querySelector("#image-count").value;
    var url = `https://dog.ceo/api/breed/${breed}/images/random/${imageCount}`;
    var xhr = new XMLHttpRequest();

    xhr.onload = function() {
        try {
```

```
            var response = JSON.parse(this.responseText);
            var breedImageList = extractBreedImageList(response);
            updateBreedImages(breedImageList);
        } catch(e) {
            showError("Unable to load dog breeds");
        }
    };

    xhr.open("GET", url);
    xhr.send();
}
```

As before, we make a request to the server, and get back JSON, which we parse. The resulting `Object` we get back looks like this:

```
{
    status: "success",
    message: [
       "https://images.dog.ceo/breeds/hound-afghan/n02088094_1003.jpg",
       "https://images.dog.ceo/breeds/hound-afghan/n02088094_1007.jpg",
       ...
    ]
}
```

Once again we get a `status` and a `message` body. This time, however, the `message` is already an `Array` of URLs.

4. Dynamically create elements for all dog breed image URLs

Using the list of URLs for dog breed images from the server, we can easily update our page to display new images:

```
var imagesContainer = document.querySelector("#images-container");

// Clear the imagesContainer if there is anything there now
imagesContainer.innerHTML = "";

// Turn a url String into an <img src="url"> element
function createImgElement(url) {
    var img = document.createElement("img");
    img.src = url;
    return img;
}

// Loop through the image URLs, and create new <img> elements
breedImageList.forEach(function(url) {
    var img = createImgElement(url);
    imagesContainer.appendChild(img);
});
```

We've now got a relatively simple page that can be changed using live data to look completely different, depending on the needs of the user. We didn't have to write HTML for every image, which would have involved hand-writing 120 * 150 = 18,000 `<img>` elements! Using AJAX we can do this with very little code.

Complete code for the example above can be found in the following files:

- dogs.html
- dogs.js
- styles.css

XHR and Cross-Origin Requests

It should be noted that we've been making requests to third-party web servers in the example above. This is something that won't always work for all servers, or all data types.

In the examples above, the servers were configured to allow Cross-Origin Resource Sharing (CORS). By default, browsers maintain a sandbox around resources (scripts, images, data) from one origin, and don't let it mix in unsafe ways with resources from other origins.

In general, it's best to load data from the same origin as your page (i.e., the same web server). The so-called "Same Origin" policy states that you can't load data from other origins. However, using CORS headers, servers and browsers can allow this in some cases. In the examples above, the servers added a header `Access-Control-Allow-Origin: *` to indicate that cross-origin requests were OK.

If ever you are trying to use XHR to request data from a serer, and it won't work, the problem is almost certainly related to CORS.

Other Mechanisms for working with Data

While XHR is a historically popular choice for accessing data from web services, in recent years a number of new APIs have also emerged that offer both similar and enhanced capabilities. Discussing these in any detail is beyond the scope of this course; however, it is important that you're aware of them, and look out for opportunities to learn and use them in future courses and projects.

`fetch()` API

The `fetch()` API provides JavaScript developers a rich set of objects and functions for working with the entire HTTP network infrastructure, including things like requests, responses, the cache, etc. It uses the modern Promise API for handling callbacks.

Here's what it would look like to use `fetch` to request the current Bitcoin value from our first example above:

```
fetch("https://blockchain.info/q/24hrprice?cors=true")
  .then(function(response) {
    var currentValue = `${response.text()} (USD)`;
    updateBitcoinValue(currentValue);
  })
  .catch(function(err) {
     updateBitcoinValue("unknown (error, unable to get current value)");
  });
```

`fetch` offers much greater control over the network, and makes it easier to process data coming from the server.

Server Sent Events

With our Bitcoin example, we needed to constantly poll (i.e., re-request) the updated value from the server. Another approach would be to wait for the server to send us an update (push). One way to accomplish this is with Server Sent Events.

Server Sent Events allow web sites to open a long running connection with a server, and get updates from the server when there is new data. On the browser side, we use the `EventSource` `Object` to process these events, just as we would other types of events in a web page.

Web Sockets

Server Sent Events are great for 1-way communciation from a server to a browser, but sometimes we need to *also* send data back to the server at regular intervals. Consider a chat application, where we need to both receive and send messages.

In such systems, we need a bi-directional, long-running connection to the server. The browser provides this in the form of a `WebSocket`. Web Sockets can be connected to different backends written in any language. Within the browser, we receive events when data arrives, and send data when we need to over the socket.

Modern Front End Development

The concepts we've been discussing above have come to define the modern approach to web development. Browsers have gotten faster and more powerful, and the APIs and tools for building web services more full-featured.

As a result, new approaches to front-end development have taken over. In your follow-up courses you'll learn more about popular front-end frameworks, which build on the low-level ideas we've been using here (HTML, CSS, JS, DOM, JSON), for example:

- React
- Angular
- Vue
- Ember

These frameworks take a data-driven approach to developing on the web, separating an applications data and state from its presentation. The ideas begun with AJAX are taken to another level, and single-page HTML applications are used to create rich interfaces.

Understanding the foundation for how all of this works, from HTML, CSS, and JS to JSON and XHR, will be an important part of taking the next step.
