COMP 1073
Client-Side Scripting

Lesson 6 – Part 1

Communicating with the Server Through Ajax

Georgian

Objectives

In this Lesson we will learn about:

1. **AJAX History**
2. **Creating an AJAX Call**
3. **AJAX Data Formats**
4. **AJAX Accessibility**
5. **Common AJAX Mistakes**

Communicating with the Server Through Ajax

Introduction – Communicating with the Server Through Ajax

- In this lesson, we take a step into the world of AJAX.
- We extend code from previous lessons and take that codebase to the next level by interlacing it with a server communication layer to retrieve the data.
- More specifically, we take the JSON object you've been working with and move it to **an external data file**.
- Then we pull it in with a combination of user events and Ajax calls to create another layer of separation where the data source is external to our normal stack of structure, presentation, and behavior.
- The **data layer** can be in almost any form from JSON to a fully functional database.
- In previous lessons, we mentioned AJAX briefly, but in this lesson you learn exactly how to create and execute an AJAX call.
- You break down the anatomy of each action and discover some new core JavaScript methods in the process.

AJAX History

AJAX History

- AJAX is the concept of refreshing a part of an HTML document **without reloading** the entire page.
- In Lesson 1, we briefly touched on AJAX while going over the history of JavaScript and mentioned how the origins of client-server communications started in 1999 when Microsoft created the **`XMLHttp`** object.
- Microsoft initially brought forward the **`XMLHttp`** object to fulfill a need in its mail client, Web Access 2000.
- They wanted a way for the client to be able to communicate with the server.
- It was first implemented in Internet Explorer 5 as an **`ActiveX`** object.

AJAX History (Cont'd)

- **`ActiveX`** was (and still is) a proprietary Microsoft product made for embedding objects in HTML.
- It was very powerful, but riddled with **security holes**.
- Because Microsoft was taking hold of the **`XMLHttp`** object and a very powerful technology that would later be called **AJAX** a little company called Mozilla stepped into the same path, developed the **`XMLHttpRequest`** object, made it native to the browser (no need for the **`ActiveX`** plug-in) and released it to the world in its first Web browser, Mozilla 1.0.

AJAX History (Cont'd)

- The term AJAX was coined in 2005 by Jesse James Garrett in an article titled *AJAX: A New Approach to Web Applications*.
- The term AJAX itself has evolved as well; initially it was AJAX, an acronym meaning: **Asynchronous JavaScript and XML.**
- This was quickly (in a matter of days) changed when people realized two important flaws in that statement:
  - AJAX can be synchronous or asynchronous.
  - AJAX can use XML, JSON, plain text, or HTML as a data source.
- Rather than changing the acronym entirely, they stuck with it and stopped calling it an acronym, settling on AJAX.

Server Communication

- **Server communication** is at the core of the AJAX technology.
- The goal has always been to be able to send and receive information from the client to the server and create a better user experience in the process.
- Until AJAX came along, all server communication happened on the server, and redrawing portions of a page required either an `iframe` (iframes are awful) or a full-page refresh.
- Neither method provided what I would call a good user experience.
- Ajax offers us two types of server communication:
  - Synchronous
  - Asynchronous

Server Communication - Synchronous

- **Synchronous AJAX** is not very common, but it is perfectly valid. You probably won't use it, but just in case you hit a situation where you need to, it's good to know what it is.
- **Synchronous AJAX** means that the AJAX call happens at the same time as all the other requests in your application.
- There are positives and negatives to this. Making your Ajax calls synchronous will **block the download** of other assets until the request completes, which can create a weird user experience because of some extra-heavy lifting that needs to happen before a page is completely rendered in the browser.
- Many people skip the synchronous AJAX model and parse the data server-side to pick up some speed.
- This is why it isn't used often. Generally speaking, if you can process something on the server, it will be significantly faster than on the client, and you should take advantage of that.

Server Communication - Asynchronous

- **Asynchronous AJAX** is much more common. You'll probably use this 99 out of 100 times you write an AJAX call.
- **Asynchronous** means that the AJAX is not firing at the same time as everything else; it's fairly **independent** and separated from the rest of the assets in a page or Web application.
- These AJAX interactions happen behind the scenes and don't really block anything from downloading.
- Even if you're executing an AJAX call with the **`onload`** event, the timing of that is usually slightly after the normal page load.
- Asynchronous calls can happen at any point and be triggered by any event (**`click`** , **`focus`** , **`blur`** , **`touchstart`** , and so on).
- The point of them is that they don't happen in one large chunk bundled together with the rest of the HTTP requests on a page.

Server Communication – Asynchronous (Cont'd)

- Using an **asynchronous** call removes the blocking nature of the synchronous call because the user can continue to interact with the page in other ways while the request is being processed.
- Many asynchronous AJAX calls go on constantly without the user even knowing that something is happening.

The `XMLHttpRequest`

- The **`XMLHttpRequest`** object is the heart of any AJAX call. We just went over the origins of this object a few paragraphs ago, so I won't get into that again.
- Next, we will be cracking open the contacts search form that you have been building throughout the course of the book and adding AJAX functionality into the autocomplete feature.
- Creating an instance of the **`XMLHttpRequest`** is the first step in making an AJAX call, and it's pretty easy.
- In the following code Listing you can see that we're grabbing an instance of the object and saving it to the **`xhr`** variable so we can use it later.

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

The `XMLHttpRequest` – Cross-Browser Issues

- Remember, from earlier in this lesson, that Microsoft first invented the **`XMLHttp`** object that later became the more popular **`XMLHttpRequest`**.
- This does cause a bit of an issue because the **`XMLHttp`** object is the only AJAX object supported in Internet Explorer 5 and Internet Explorer 6.
- If someone were to visit your Ajax enabled application with IE 6, it wouldn't support the normal **`XMLHttpRequest`** object, but rather the older **`XMLHttp`** object.
- It's a little bit of a pain, but if you don't personally support IE 6 in your development process anymore, don't worry about it.
- It's pretty easy to do a quick support check for the **`XMLHttpRequest`** and move on from there, just in case someone from that rapidly falling market share were to stumble upon your application.

The `XMLHttpRequest` – Cross-Browser Issues (Cont'd)

- In the following code Listing you can see the steps to check for support.
- All we're doing is running a simple **`if` statement** to check for the presence of either the **`XMLHttpRequest`** or the **`ActiveXObject`** and then setting the proper AJAX object to the **`xhr`** variable. This is a form of feature detection.
- Rather than targeting IE 6 directly (because that's very specific), we are instead casting a blanket over a possibly wider audience of **ActiveX** supporters.
- Again, if you don't support IE 6 at all, you don't have to worry about this step.

```
if ( window.XMLHttpRequest ) { // check for support

    // if it's supported, use it because it's better
    xhr = new XMLHttpRequest();

} else if ( window.ActiveXObject ) { // check for the IE 6 version

    // save it to the xhr variable
    xhr = new ActiveXObject("Msxml12.XMLHTTP");

}
```

The `XMLHttpRequest` – Cross-Browser Issues (Cont'd)

- Because this is a very common bit of functionality to have with any AJAX call and will probably be reused over and over, it is best to save it to a **function**.
- You will find similar functions all over the Web with **`XMLHttpRequest` feature detection**.
- The goal of this function is to **detect** for the correct AJAX object and return it for use wherever you need it.
- Because this is a function that passes information into another function, the first step is to initialize the variable, then run the normal **`if` statement** and return the variable so it can be passed into another function.
- In the following code Listing you can see the complete function called **`getHTTPObject()`** and the return value of **`xhr`** at the end.

The `XMLHttpRequest` – Cross-Browser Issues (Cont'd)

```
function getHTTPObject() {

    // initialize the variable
    var xhr;

    if (window.XMLHttpRequest) { // check for support

        // if it's supported, use it because it's better
        xhr = new XMLHttpRequest();

    } else if (window.ActiveXObject) { // check for the IE 6 Ajax

        // save it to the xhr variable
        xhr = new ActiveXObject("Msxml2.XMLHTTP");

    }

    // spit out the correct one so we can use it
    return xhr;
}
```

Creating an Ajax Call

Creating an Ajax Call

- Creating the AJAX object instance is a **separate step** from creating the actual AJAX call.
- All we've done so far is say, "Hey, we're getting ready to work with some AJAX goodness."
- For our application there needs to be a little preparation to convert it for use with AJAX.
- The biggest change will be the **data source**. Previously, the data has been stored inside a **JSON object** within the main JavaScript file.
- Having the data right there with the rest of your JavaScript will provide the **shortest response time** but will prevent your application from scaling from the five contacts we've been using to something like a more real-world address book containing hundreds of people.
- Your main JavaScript file will quickly become difficult to manage.
- It is also pretty rare to have all your data on hand like that; it is generally hosted somewhere else.

Creating an Ajax Call (Cont'd)

- To create a more accurate situation, we're going to move all the contacts' **JSON data** into an external file called **`contacts.json`**.
- The **`.json`** file extension isn't required, but it is best practice to use file extensions that describe the file's contents.
- Because you already have a valid JSON format for the data, it's a pretty easy copy and paste into the external file.
- The only difference is that you'll be removing the **`var`** statement.
- The following code Listing shows you what the contents of your **`contacts.json`** file should look like.

Creating an Ajax Call (Cont'd)

```
{
"addressBook" : [
    {
        "name": "hillisha",
        "email": "hill@example.com"
    },
    {
        "name": "paul",
        "email": "cleveland@example.com"
    },
    {
        "name": "vishaal",
        "email": "vish@example.com"
    },
    {
        "name": "mike",
        "email": "grady@example.com"
    },
    {
        "name": "jamie",
        "email": "dusted@example.com"
    }
]
}
```

Sending a Request to the Server

- Ajax talks to the server, the server doesn't talk to AJAX.
- Because of this, AJAX does two things:
  - Sends a request to the server
  - Processes the data returned from the server
- The first thing you should do after you have created an instance of the AJAX object to work with is to send a request.
- Sending an AJAX request opens up a new property for you to monitor called **`readyState`**, which we will get into in a bit.

Sending a Request to the Server
GET vs POST and performance

- AJAX calls can be in the form of a **GET** or a **POST**.
- You don't use one over the other like using a GET when you're getting information and a POST when you're putting information.
- They work more like a normal HTML **form method**.
- When you're creating a search form in HTML, there is an attribute in the form's markup called **`method`**; you can set that method to GET or POST.
- What it does is on the search results page— all the data is exposed in the URL.
- It's very common because the data isn't sensitive at all.
- The downside of using a GET is that there is a (pretty large) character limit for the data being passed.

Sending a Request to the Server
GET vs POST and performance (Cont'd)

- Not all form methods are set to **GET**, though; some are set to **POST** because the data is more sensitive, such as a username and password.
- The same principles are applied to AJAX **GET** and **POST** methods.
- If you have sensitive data you're sending through an AJAX call, it should be sent via a **POST**, but if you're working with non-sensitive data, it's best to use a **GET** .
- You may be thinking, why not just use a **POST** all the time for AJAX because the URL isn't ever exposed?
- The answer is that a **GET** request **performs better** than a **POST** in most situations.
- Because the data in a **GET** is exposed, less processing is involved, which speeds up the performance of the request.

Sending a Request to the Server – the **`open()`** method

- The **`open()`** method is the second step in getting your AJAX call started. Think of it like a configuration file for the AJAX call.
- It doesn't do any actual work, but what it does is **prepare the statement** to be executed by gathering all the necessary information. It's like having an administrative assistant for your AJAX call.
- The following code Listing shows the AJAX object function we created being saved to a variable called **`request`** and then the **`open()`** method being attached to it, taking three arguments.

```
var request = getHTTPObject();

/* Get all the information ready to go */

request.open("GET", "data/contacts.json", true);
```

Sending a Request to the Server – the `open()` method (Cont'd)

- The **three arguments** in the `open()` method of the following code Listing are
  - The method
  - The file or URL to get
  - A Boolean flag for asynchronous script
- There are also **two other optional arguments**:
  - Username
  - Password

Method

- The first argument in the `open()` method is the method you want to use for your AJAX call.
- This can be set to either **GET** or **POST**

File or URL

- The **file** or **URL** argument is a place for the file path or full `HTTP URL` of the data source you will be pulling in via the AJAX call.
- If it's a local file, like ours is, the path is relative to the HTML document you're using it in, not the JavaScript file.
- This is why ours is set to `data/contacts.json` and not `../data/contacts.json`

Sending a Request to the Server – the `open()` method (Cont'd)

Asynchronous or Synchronous

- The third argument in the **`open()`** method is a flag to tell the AJAX call whether it will be executed as a synchronous call or an asynchronous call.
- As mentioned earlier, this will be set to true almost all the time, because asynchronous AJAX calls usually provide a much better user experience when compared to synchronous calls.

Sending Credentials

- The last two arguments in the **`open()`** method are reserved for a username and a password.
- You would use these arguments when implementing an AJAX call on a sign-in or a registration form.
- Whenever you send password information with this method, it is important to make sure the data is encrypted for better security.
- Even though you would use a **POST** method and the URL wouldn't be exposed publicly, encryption is equally as important as it is when coding on the server.

Sending a Request to the Server – the **`send()`** method

- After all the data for your AJAX call has been properly prepared in the **`open()`** method, you can use the **`send()`** method to ship off the data and **`request`** and **`begin`** waiting for the **`readyState`** property to let you know when the AJAX call data is ready to be used.
- In the following code Listing you can see the AJAX call getting built out.
- We now have the AJAX object, the **`open()`** method gathering the data, and finally the **`send()`** method firing off the actual call.

```
var request = getHTTPObject();

/* Get all the information ready to go */
request.open("GET", "data/contacts.json", true);

/* initiate actual call */
request.send(null);

/* OR - initiate the call with some data */
request.send("hello data");
```

Sending a Request to the Server – the `send()` method (Cont'd)

- You may have noticed that we're passing **`null`** into the **`send`** object, which means that we're not sending any extra data with the AJAX call. We just want the file.
- If you were using any back-end processing on the data URL, you can pass the extra filtering information through the **`send()`** method.
- The following code Listing shows what something like that may look like for a search result.

```
var request = getHTTPObject();

/* Get all the information ready to go */
request.open("GET", "search.php", true);

/* initiate actual call and filter by the term "hill" */
request.send("searchterm=hill");
```

- The request in code listing above would produce the same data as if you were to visit the URL **`search.php?searchterm=hill`**, a normal search results page.

Receiving Data Back from the Server

- After the **request** is sent, the call will return from the server, hopefully with the data you requested.
- As I mentioned a little earlier in the chapter, AJAX opens up a new property called **`readystate`**, which is tied to an event called **`readystatechange`**, which constantly monitors the progress of every AJAX call and reports back to you so you know when the data is available to parse.

readystate

- **`readyState`** is the property in an AJAX call that reports back the status that corresponds with a checkpoint in the processing of that AJAX call.
- Five values get reported back:
  - 0 – The open method hasn't been called (uninitialized)
  - 1 – The open method has been called, but the send method has not (loading)
  - 2 – The send method has been called and the request is being sent (loaded)
  - 3 – The response has started to come back (interacting)
  - 4 – The request is complete (complete)

Receiving Data Back from the Server (Cont'd)

- Each time a **readyState** value changes, the **readystatechange** event is triggered.
- Knowing this, you can attach an **event handler** to this new event and wait for the "4" status to be reached before executing something on the returned data.
- It is possible to listen for each step in the process if you want to provide very detailed feedback to the user, but generally speaking, it's easier to wait for the request to complete by looking for the "complete" value.
- In the following code Listing you can see our AJAX object followed by the event handler for **onreadystatechange**, and then an if statement checking the readyState value.

```
var request = getHTTPObject();

request.onreadystatechange = function(){

    // check if the request is ready
    if( request.readyState === 4 ) {

        // do something

    }
```

Receiving Data Back from the Server (Cont'd)

- Not so tough, right? It looks like a normal function. And it is just a function with an **`if` statement**.
- By checking the **`readyState`** value like this, you can be sure you're not executing code before you have data available to parse.

Server Status

- The readystate property is great, but it only tells you what the step-by-step process of an AJAX call is.
- It doesn't give you any information on whether the request was successful.
- The AJAX object also returns a property called status , which correlates to the server status codes you would normally find on a Web server, like **`404`**, **`200`**, **`304`**, **`500`**, and so on.
- An AJAX call can, in theory, go out to the server and successfully come back, but encounter some sort of failure on the server that prevented the data from being returned.
- Some of the more common codes you can check for are
  - **`404`** – Page not found
  - **`304`** – Not modified
  - **`500`** – Internal server error
  - **`200`** – All is well on the server

Receiving Data Back from the Server (Cont'd)

- You can write **conditional code** for each of these statuses, but for our purposes, we are going to focus on the success status of **`200`** and combine that with our **`readyState`** of **`4`**.
- In the following code Listing you can see the addition to the **`if`** **statement**, which now not only checks for a complete AJAX call, but also for a successful server status code to be returned.

```
var request = getHTTPObject();

request.onreadystatechange = function(){

    // check if the request is ready and that it was successful
    if( request.readyState === 4 && request.status === 200 ) {

        // do something

    }
}

/* Get all the information ready to go */

request.open("GET", "data/contacts.json", true);

/* make the actual call */

request.send(null);
```

Receiving Data Back from the Server (Cont'd)

The Server Response

- Inside the **`onreadystatechange`** event handler, and after you check to make sure the request is complete and successful, you can finally get the data that was returned from the AJAX call.
- In addition to **`readyState`** and status, the AJAX object also returns your data as a property.
- It will either be returned in the form of a string or as XML, depending on which data format you choose to interact with.

As a String

- If your data response in is the form of a **`string`**, it will return as **`responseText`**.
- This is just a string of data that needs to either output as it is returned or be parsed with some of the native objects available in JavaScript. This is the most common form of AJAX response data.
- This format is the most common format because it can contain anything from **plain text**, to **HTML**, to **JSON**, and then be parsed accordingly.
- To access this data inside our AJAX call, you access the AJAX object, then the responseText like this: **`request.responseText`**;

Receiving Data Back from the Server (Cont'd)

As XML

- If you are returning XML data, the response will be in the form of **`responseXML`**.
- The following code Listing shows how you would spit out the retuned data into the JavaScript console to observe the contents of the response.

```
var request = getHTTPObject();

request.onreadystatechange = function(){

    // check if the request is ready and that it was successful
    if( request.readyState === 4 && request.status === 200 ) {

        // spit out the data that comes back
        console.log(request.responseText);

    }
}

/* Get all the information ready to go */

request.open("GET", "data/contacts.json", true);

/* make the actual call */

request.send(null);
```

Receiving Data Back from the Server (Cont'd)

Getting It into a Function

- Now that you have a working AJAX call, you will probably want to use it more than once in your application. In fact, we do want to use it more than once.
- Once for the autocomplete functionality and another time to get all the contacts in a single call.
- To make the **function** a little easier to reuse, we're going to take what we already have and put it into a function, with one small change.
- To be certain that this function can be reused within the context of this application and retrieve the contacts.json file, we are going to pull the reference to **`data/contacts.json`** out of the function and instead use an argument that will be passed when it is called.
- The following code Listing shows our complete AJAX function with a **`dataURL`** argument being passed into it.

Receiving Data Back from the Server (Cont'd)

```
/* define the Ajax call function */

function ajaxCall(dataUrl) {

    /* use our function to get the correct Ajax object based on support */
    var request = getHTTPObject();

    request.onreadystatechange = function() {

        // check to see if the Ajax call went through
        if ( request.readyState === 4 && request.status === 200 ) {

            // spit out the data that comes back
            console.log(request.responseText);

        } // end ajax status check

    } // end onreadystatechange

    request.open("GET", dataUrl, true);
    request.send(null);

}
```

Receiving Data Back from the Server (Cont'd)

Returning the Data

- Right now the data is stuck inside the `onreadystatechange` event handler. This obviously isn't what we want.
- If we were using this function only once, we could add all the data parsing and output right inside the event handler.
- You could build out all your functionality right there inside the `ajaxCall()` function.
- We can make this function a lot more reusable by finding a way to work with the data it's returning while still remaining inside the context of the `onreadystatechange` event handler.
- To accomplish this goal, we need to find a way to call the `ajaxCall()` function and allow it to take an extra argument that is a function.
- With that in mind, what we need to do is allow the `ajaxCall()` function to accept a callback function.
- The first step to doing this is to save the `responseText` to a variable so we can work with it more easily.

Receiving Data Back from the Server (Cont'd)

- The following code Listing introduces a new argument called **`callback`**, which will represent the function we will be passing through to access the data response.
- So we will have two arguments; one will be a **`string`**, and the other will be a **`function`**.
- Because the data is saved to a variable (**`contacts`**), it is easy to pass that data into the callback function.

Receiving Data Back from the Server (Cont'd)

```
function ajaxCall(dataUrl, callback) {

    /* use our function to get the correct Ajax object based on support */
    var request = getHTTPObject();

    request.onreadystatechange = function() {

        // check to see if the Ajax call went through
        if ( request.readyState === 4 && request.status === 200 ) {

            // save the ajax response to a variable
            var contacts = JSON.parse(request.responseText);

            // make sure the callback is indeed a function before executing it
            if(typeof callback === "function"){

                callback(contacts);

             } // end function check

        } // end ajax status check

    } // end onreadystatechange

    request.open("GET", dataUrl, true);
    request.send(null);

}
```

Receiving Data Back from the Server (Cont'd)

- In the bolded code toward the end of the AJAX **status-checking-if-statement** in the previous code Listing, you can see another if statement wrapping the callback function.
- This is using the JavaScript operator **`typeof`**, which can return the type of object you are dealing with.
- It can return things like **`string`**, **`number`**, **`boolean`**, **`undefined`**, and in this case we are looking for it to return "function" before moving forward.
- This is a kind of check and balance we use when programming JavaScript just to make sure everything goes as planned.
- Something else odd you may have noticed about this function is that the **`responseText`** is wrapped in a method called **`JSON.parse()`**.
- As mentioned before, the **response** can come back as either a **`string`** or as XML; in this case, it came back as a string but to be able to work with the data in the way we did while it was an embedded JSON object, it needs to be converted (parsed) back into its **JSON format**.
- Using **`JSON.parse()`** does just that; it's a real life saver for something like JSON parsing.

Receiving Data Back from the Server (Cont'd)

```
ajaxCall("data/contacts.json", function(data){

    /*
    these are the contents of the callback function
    the "data" argument is the contact list in JSON format
    this is where you would loop through the data
    */

});
```

Making Repeat Ajax Calls

- Creating the illusion of **real-time data** is something JavaScript, and particularly AJAX are both very good at.
- We already learned how to **create** and **execute** an AJAX call once, based on either the load of a page or a user-initiated event.
- But you can also create a system that can automatically make AJAX calls over and over.
- You can use the JavaScript object **`setInterval()`** to execute a block of code over and over with a set time in between each call.
- The following code Listing shows how you would use this method to execute a JavaScript alert every 5 seconds.
- The **`setInterval()`** method takes two arguments.
  - The first is whatever **function** you'd like to repeat
  - The second is how much time, in **milliseconds**, you want to pass in between each call.

```
/* alert a message every 3 seconds */

setInterval('alert("fire off an Ajax call', 5000); // alert something every 5 seconds
```

Making Repeat Ajax Calls (Cont'd)

- The same pattern can be applied to our **`ajaxCall()`** function to fire the call off every 5 seconds.
- This process of constantly hitting the server to check for new information is called **polling**.
- If we were pulling data that was being constantly updated, it would be very valuable to poll the server to refresh the data onscreen every so often.
- The following code Listing shows how using **`setInternal()`** with the **`ajaxCall()`** function might look.

```
/* make this Ajax call every 5 seconds */

setInterval('ajaxCall("data/contacts.json",

        function(){

            console.log("made a call");

    })', 5000); // 5000 milliseconds = 5 seconds
```

AJAX Data Formats

AJAX Data Formats

- AJAX can take more than one data format.
- Most commonly, you will be dealing with three specific data formats: **`XML`**, **`HTML`** and **`JSON`**.
- Each has positive aspects and negative aspects; knowing the difference will help you choose the right tool for the job.

XML

- **XML** stands for **eXtensible Markup Language**.
- It is a very flexible data format and immensely popular for use in application data.
- It is very similar to HTML in its anatomy, containing a **DOCTYPE**, **elements**, **tags**, and **attributes**.
- It even adheres to the same document object model as HTML.
- The following code Listing depicts how the `contacts.json` data file would look if it were converted into XML format.

Positives

- Being an extensible format is a huge plus for XML. You're not locked into any predefined data structure because you can define it as you go, as long as it is kept consistent throughout the file.
- Adhering to the DOM standard is another positive aspect of XML. After pulling the data in through an AJAX call, it is parsed the same as a normal HTML document with methods like `getElementsByTagName()`, `getAttribute()`, `parentNode`, `firstChild`, and `lastChild`.
- Not having to relearn any new methods to parse XML makes it a very attractive option.

XML (Cont'd)

Negatives

- Because XML is so similar to HTML and uses the same DOM standards, it can take a lot of code to parse through and build the output.
- One pretty big drawback of using XML is that it cannot be used cross-domain with AJAX.
- All AJAX calls to an XML data file must come from the same domain or the request will fail.
- For this reason, you don't see a lot of public data sources in XML format.
- If they were, you would need to create a server-side proxy to pull in the XML and have the AJAX call reference the data by way of the proxy.

```
<?xml version="1.0" encoding="utf-8"?>
<addressBook>

    <person>
        <name>hillisha</name>
        <email>hill@exmaple.com</email>
    </person>

    <person>
        <name>paul</name>
        <email>cleveland@example.com</email>
    </person>

    <person>
        <name>vishaal</name>
        <email>vish@example.com</email>
    </person>

    <person>
        <name>mike</name>
        <email>grady@example.com</email>
    </person>

    <person>
        <name>jamie</name>
        <email>dusted@example.com</email>
    </person>

</addressBook>
```

HTML

- Working with **AJAX** and **HTML** snippets couldn't be more straightforward.
- You have an HTML file and you consume its contents in full with an AJAX call.
- The following code Listing shows our JSON data in the form of HTML.

```
<ul>
    <li><a href="mailto:hill@example.com">hillisha</a></li>
    <li><a href="mailto:cleveland@example.com">paul</a></li>
    <li><a href="mailto:vish@example.com">vishaal</a></li>
    <li><a href="mailto:grady@example.com">mike</a></li>
    <li><a href="mailto:dusted@example.com">jamie</a></li>
</ul>
```

Positives

- Speed is an important reason to use this method. Unlike the other data formats, no client-side parsing is needed because you're grabbing an entire snippet of HTML and outputting it into the DOM.
- Not having to write a lot of extra JavaScript to parse the incoming HTML will not only save you time as a developer, but it will save processing time for the user.

HTML (Cont'd)

Negatives

- Using HTML as a data source works very well if you are asynchronously updating a single block of content in a document, but it doesn't get you the fine-grained control that XML or JSON will give you.
- Unless you're literally updating an HTML document with a static block of content, which would be a little odd and, frankly, pretty rare, you're going to have to do some server-side processing to get the data you want.
- This is generally the fastest way to do it, but if you like working in server-side code, that might be something to consider as a downside of using HTML as a data format.

JSON

- We have been using JSON as a data format for most of the Course, so you should be pretty familiar with it at this point.
- It's a very human-readable and machine-readable format, which has no structural limitations.
- Each item in a JSON data format can be different from all the others. In our data file, we are using a consistent structure of **name** and **email**, but it doesn't have to be like that because JSON doesn't force any real consistency in its format.
- The following code Listing shows the current JSON data we have been working with so you can compare it to the XML and HTML versions in previous listings.

```
{
"addressBook" : [
    {
        "name": "hillisha",
        "email": "hill@example.com"
    },
    {
        "name": "paul",
        "email": "cleveland@example.com"
    },
    {
        "name": "vishaal",
        "email": "vish@example.com"
    },
    {
        "name": "mike",
        "email": "grady@example.com"
    },
    {
        "name": "jamie",
        "email": "dusted@example.com"
    }
]
}
```

JSON (Cont'd)

Positives

- JSON is very popular because it is native to JavaScript, very fast, flexible, and platform agnostic—meaning almost any programming language plays nice with JSON.
- Unlike XML, JSON can be consumed cross-domain very easily; it has no native domain limitations because it is just a JavaScript data format. This makes it the ideal candidate for API structures.
- Because it is such a flexible format, you will find that the majority of Web services offered are in JSON format.

Negatives

- Although the data format of JSON is very flexible, the syntax is not.
- Every comma, quotation mark, and colon needs to be in the right place for the data to be parsed correctly.
- Some security concerns also exist with using JSON from any third-party Web service, because at its core, it's just JavaScript, and it's very easy to remotely **inject malicious scripting** through a JSON object.
- This can be protected against, but in general you should consume JSON data only from trusted sources.

AJAX Accessibility

AJAX Accessibility

- One often overlooked portion in AJAX development is **accessibility**. We go through a lot of work to make a website accessible to disabled users and yet often fall short when AJAX functionality is layered on.
- When the DOM is initially rendered, a sighted user is able to easily click and see the feedback of a certain region of the page when it's updated, but we tend to forget that there are ways to flag a region with attributes to let visually impaired users know that a portion of the page is going to be updated without a full page refresh so they can come back periodically and check the content of that area.
- This method is called **Accessible Rich Internet Applications** (ARIA), and it is something you should be familiar with before diving too far into the dirty world of AJAX.
- Accessibility is extremely important. Think of it as usability for disabled users.
- The goal of creating a top-notch user experience shouldn't be derailed because a user has trouble seeing or hearing, or any form of disability.
- We create one Web, and that Web should be accessible to everyone, no matter what.

Live Regions and ARIA

- Live regions in HTML exist to indicate to **assistive technologies** (screen readers) that a certain area in the document might possibly change without focus or a page refresh.
- ARIA regions have been around for a few years now, living independently from the rest of the W3C specifications, but it is now getting a lot more attention.
- There are currently four types of live region attributes for ARIA:
  - `aria-atomic`
  - `aria-busy`
  - `aria-live`
  - `aria-relevant`

aria-atomic

- **`aria-atomic`** indicates whether a screen reader should present all or parts of a live region based on the change notifications. This attribute takes **`boolean`** values, either **`true`** or **`false`**.

aria-busy

- **`aria-busy`** is a state reported to a screen reader, which reports back whether a live region is currently being updated. Just like aria-atomic, this attribute also takes either a true or false value. Because this attribute changes based on update status, it needs to be set and updated in the AJAX call.

Live Regions and ARIA (Cont'd)

aria-live

- **`aria-live`** is a way to report how important live changes in the document are. It can take one of three values: **`off`**, **`polite`**, or **`assertive`**.
- Setting the **`aria-live`** attribute to off prevents updates from bubbling up unless the user is directly focused on the element.
- The **`polite`** value will be sure to update the user only when it is courteous to do so. In other words, it will not interrupt the user if something more important is going on, but will update the user at the next convenient moment.
- The last option, **`assertive`**, sets changes to the highest priority and will notify the user immediately of any changes, no matter what. Because this option can be disorienting to a user, it should be used only when absolutely necessary.

Live Regions and ARIA (Cont'd)

aria-relevant

- **`aria-relevant`** will notify the atomic regions as to what type of change has occurred. There are four options for this attribute: **`additions`**, **`removals`**, **`text`**, and **`all`**.
- The most common values for this attribute are **`additions`** and **`removals`**; this means that the region contains items and are both removed and added to. Our autocomplete form does this.
- In contrast, a **`text`** value to this attribute means that text has been added to a DOM node within the region.
- Last, the **`all`** value means everything is going on and rather than listing **`additions`**, **`removals`**, and **`text`**, you can use the all value.
- The following code Listing shows how we are going to apply ARIA live regions to the HTML in our autocomplete form by using **`atomic`**, **`live`**, and **`relevant`**.

```
<!-- adding aria to the HTML output -->
<div id="output" aria-atomic="true" aria-live="polite" aria-relevant="additions
➥removals"></div><!--/#output-->
```

Common AJAX Mistakes

Common AJAX Mistakes

- The number one mistake developers make with AJAX is using it because they think it's cool.
- AJAX is not always the answer. In fact, it is often not the answer at all.
- Use it responsibly and with caution; there is nothing worse than trying to debug an application that piles up its AJAX calls.
- Most in-browser problems can be traced back to a **JavaScript** or an **AJAX origin**, and you will avoid a lot of headaches if you put serious up-front thought into whether you need to or even should use AJAX to accomplish a certain goal.