CSCE 120: Learning To Code

Module 11.0: Consuming Data I
Introduction to Ajax

This module is designed to familiarize you with web services and web APIs and how to connect to such services and consume and process data using JavaScript.

Client-Server Model & HTTP

Many computer applications use the *client-server model* which separates tasks between two entities: a client and a server. This is a very general model; a client can be an installed application on a laptop or mobile device or a browser or even another program that requires no human interaction at all. In general, a server is a provider of services or resources. Clients (we'll mostly focus on web browsers) communicate with servers over a network by sending *requests* and servers respond with data or resources by sending *responses*.

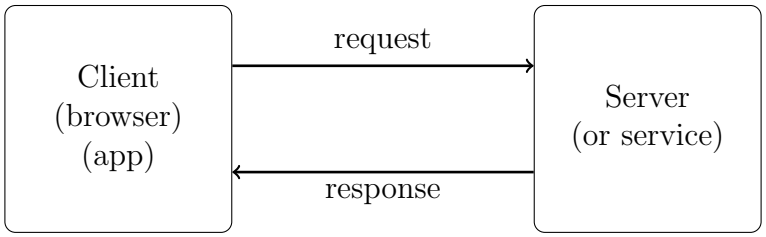

Figure 1: Client-Server Model

In the context of browsers and web servers, the communication between clients and servers is done using the HyperText Transfer Protocol (HTTP). A *protocol* is simply a set of established rules for communication and prescribes how requests and responses are formatted and sent over a network. The HTTP protocol also defines several response codes. Examples would include the 404 (meaning that the page/resource was not found); 500 (indicating an internal server error) or 200 (indicating success. You don't usually the

200 response code in a browser since it means that the request was successful and that the response contains the page which is then rendered.

HTTP is also a *stateless* protocol, meaning that each response/request is independent of each other. This means that things like cookies or login information needs to be stored somehow by the application since it is not handled by the protocol itself.

We won't focus too much on the details of HTTP, but you should be aware of the two main types of requests: a `GET` request and a `POST` request. In general, `GET` requests are used to simply retrieve a resource (a page, an image, or other data) while a `POST` request is used to submit data to the server for processing (webform data for example).

Any type of data can be sent over HTTP including binary data (for images, audio, video, etc.) or textual data. We'll mostly be concerned about making HTTP requests using JavaScript that return JSON data so that we can process that data in our JavaScript code.

Ajax – Asynchronous JavaScript and XML

When a browser requests a page from a web address, the browser acts as a client which requests the page (and any CSS, JavaScript, image, etc. files). The server responds by sending each requested resource. The browser then renders the page and loads and/or executes any scripts.

However, that's not necessarily the end of the client-server interaction. JavaScript can be written to make subsequent requests to the server (or even a completely different server) for additional data.

The most common way to do this is through Ajax (short for Asynchronous JavaScript and XML). Though the acronym refers to transmitting XML data, any type of data can be sent (we'll focus on JSON data). Note the use of the keyword *asynchronous*: recall that some functions in JavaScript are used as callbacks which allows normal execution to continue while some expensive or long running operation executes. The same idea applies here: a request to a server along with sending a response involves quite a bit of communication over a potentially slow network. The nature of Ajax means that we can make multiple, asynchronous requests without blocking the rest of the application (which greatly improves user experience).

Finally, every browser supports the use of Ajax through the `XMLHttpRequest` object (which provides functions that can be used to make connections and retrieve data from a remote server). However, we'll focus on using jQuery's `$.ajax()` function which greatly simplifies the process.

jQuery's `$.ajax()` Function

The `$.ajax()` function takes one argument: an object that contains several settings and parameters necessary to make a connection (full documentation: http://api.jquery.com/jquery.ajax/). This object also contains functions (or references to functions) that will be used to handle the data once it has been retrieved from the remote server.

A simple example:

```
$.ajax({
  url: "http://bulletin.unl.edu/undergraduate/courses/CSCE/155A",
  data: {
    format: "json"
  },
  success: function(data) {
    console.log("Successfully got course data");
    console.log(data);
  },
  error: function() {
    console.log("Error: Was not able to get course data");
  }
});
```

The example above makes a connection to UNL's online course bulletin and gets information about the course CSCE 155A. The browser makes an HTTP connection to the server and submits the `data` as parameters to the request. The server responds with JSON formatted data about the course. The JSON response is presented below (the actual response was transmitted without the extra whitespace, but we've "pretty printed" it for easy presentation).

```
{
   "title":"Computer Science I",
   "description":"<div>Introduction to problem solving with
          computers. Topics include problem solving methods,
       software development principles, computer programming,
       and computing in society.<\/div>",
   "prerequisite":"<div>Appropriate score on the CSE Placement Exam
          or CSCE101; MATH 103 or equivalent.<\/div>",
   "courseCodes":[
      {
         "subject":"CSCE",
         "courseNumber":"155A"
      }
   ]
}
```

Let's examine the properties of the passed argument. Note again that there is only one parameter passed to the `$.ajax()` function (the object defined by the beginning and ending brackets). Each property that we passed as a specific purpose:

- `url` – this defines what server/url (and even protocol) that we want to connect to
- `data` – optionally, this is the data (parameters) that we should pass to the server, basically the input to the service or resource that we're requesting. In this example, we want the response `format` to be `json`. In general, the parameters are specific to the service you connect to.
- `success` – This is a callback function that executes when the response is successfully received. The `data` parameter passed to the function is the response data. For simplicity, we have simply logged it to the console.
- `error` – This is the callback function that executes if the response fails for any reason. There are many reasons why a response would fail and we can handle different types of failure differently, but again the example simply logs an error.

jQuery's `$.ajax()` function actually supports dozens of other parameters (see the full documentation for details). Further, the above connects to a third-party server to get data. However, in general, most Ajax requests are actually made to the same server that the application's page is served from. When connecting to the same server, a full URL does not need to be provided.

Asynchronicity & User Experience

The response/request to the server in the previous example takes a non-trivial amount of time. It may seem fast in many instances if the server connection is fast or if the amount of data is small. However, in many instances, we cannot assume that we'll always have such a quick response. This is where the asynchronous nature of AJAX helps us. If we make such a request synchronously, everything else in the browser may freeze up and "block" until the response/request was completed and processed. If we made *several* such requests synchronously (or even just one large, slow request), each one would be processed in order, one after the other, forcing the remaining requests to wait until each one had completed.

However, since the request/response is done asynchronously, the rest of the browser (and our application) is free to continue processing without having to wait for the request to complete. This greatly improves not only the execution of the program, but is essential for the User Experience (often abbreviated UX). The user could click one button that loads some data asynchronously and while waiting for it to execute, they would not be blocked from performing other actions. In fact, this is the way most user interface-based programs work.

Another thing we could do to improve the UX is to give some visual indication that the

user's request is being processed and then a visual cue on when it is done. This is usually achieved by displaying some "loading" graphic before the `$.ajax()` function is invoked. Then, in the `success` function, we could either replace or remove the loading graphic.

An example:

```
function loadCourseData() {
  $("#loadingDiv").html("<img src='loading.gif'/>");
  $.ajax({
    ...
    success: function(data) {
      //process the data here
      ...
      //remove the loading graphic:
      $("#loadingDiv").empty();
    }
  });
  //note: removing the loading graphic here would be wrong
}
```

As indicated in the comments, it would be wrong to remove the loading graphic after the ajax call. Remember that the ajax call is done asynchronously, so the `success` callback is only executed after the request/response is successfully processed. Control flow immediately resumes after the ajax call, so it would end up potentially removing the graphic *before* the ajax call was done.

Requests to the Same Server

As mentioned before, most ajax calls are to the same server that the page is being served from. That is, if the page is located at `http://cse.unl.edu/myApp.html`, then requests would usually go to `cse.unl.edu`. Of course this assumes that your application has been deployed to a real web server rather than from your local computer using Light Table.

When making a request to the same server, a full URL does not need to be provided. Instead, the URL can be relative to the page. For example, if we want to access a service located at `http://cse.unl.edu/data/getData.php`, from the page indicated above, we would make a request to the URL `/data/getData.php` instead.

In fact, the url need not be a service, it could simply be a static data file. One use case for this would be if we had a large amount of data that we didn't want to embed directly into our page. We could instead, place it into a JSON text file and load it via ajax.

Security Concerns

In general, if an application is making ajax requests to the same server, we can assume that the request/response is safe and contains valid data. However, if we make a request to another server (called a cross-domain request), then the request is not necessarily safe. While we may have complete control over what data our own server responds with, we don't have any control over the data a third party server responds with.

This has several potential security issues. Since we are essentially grabbing raw data from a remote server, that data could be malicious: it could be JavaScript code that silently sends a user's private data (such as cookies or passwords) to another server. It could be code that redirects a user to a malicious website. In general, these are referred to as Cross-Site Scripting Attacks (or XSS).

Since getting untrusted data from a third party site is dangerous, most browsers will not allow a script to connect to a server different from the server that served the script file itself. This is known as a Same-Origin Policy; requests must be made to the same origin and not to services on different servers.

However, there are instances in which we would *want* to or even *need* to connect to a different server. For example, if we wanted to connect to a Google Maps service to compute a travel distance. There are several ways around the Same-Origin policy. First, a server can override this policy and allow cross-domain requests by setting an appropriate header parameter (Access-Control-Allow-Origin). This is known as Cross-Origin Resource Sharing (CORS). One draw back is that the third party server that you want to access may not support this policy. In general, a CORS policy should only be allowed if the data being transmitted is "public", non-sensitive information.

To illustrate, consider the following scenario: a user is logged into their bank site B and is also visiting a malicious page M that makes a request to the bank server B for confidential information. B would allow such requests from any script served from B (same-origin), but does *not* allow such requests originating from a different origin, M. If B allowed such a request from M, the malicious site could hijack the user's bank account.

Another workaround is to use JSON-P (Padded JSON). This technique submits an additional parameter to the server: a callback function name. If the server supports JSON-P, it will respond not just with raw data, but with the raw data wrapped around the function name you provide. For example, if we provide a `callback=foo` parameter, the server may respond with `foo({...data...});` instead of just `{...data...}`. As long as your script contains a function named `foo`, it will invoke this function with the passed data. As with CORS, the server may or may not support JSON-P.

Yet another workaround is to establish a *proxy*. A proxy is a service, usually on server-side that allows you to make a request to a service surreptitiously through the same origin. For example, suppose we wanted to make a request to a service on server B from our script served from server A. If B does not support CORS nor JSON-P, then we could instead make the request to our server A. A process on server A is not a browser and

thus does not have the same-origin policy. It makes the request on our behalf to server B and serves the data back to the script. In this scenario, A is playing the role of a proxy.

API Keys

Another frequent issue with ajax calls to third party servers is that many web services require an Application Programmer Interface (API) Key. Many services allow you to register your application and obtain a free key that then needs to be used to access their services. Some applications do this so that they can track and control how many requests your application makes in a certain amount of time. Google Maps for example allows a certain number of requests for free, but then denies access once that limit is reached. A key allows you to make more requests if you are willing to pay for it. Other applications simply want to prevent abusive usage (like flooding them with millions of requests as in a Denial-of-Service attack).

We won't get into the details of how to use such keys in an ajax call (they would be mostly service-specific anyway). However, it is important to understand that many (if not most and a growing number) "public" APIs do require the use of such a key.