Lecture 12: AJAX and Security

CS-546 – WEB PROGRAMMING

AJAX

What is AJAX?

AJAX (asynchronous JavaScript and XML) is a series of techniques used to have clients execute JavaScript code in order to request resources without leaving their current page.

This means that you can write code that makes network requests on the client's behalf to access data on your server.

We will be using jQuery to perform our AJAX requests.

How do I use AJAX?

Using jQuery, we would use the *$.ajax* method to create an *XMLHttpRequest* for each resource request you want to make.

See *basic_ajax_with_jquery.js* for an example of how to GET and POST.

We can monitor these requests in our browser's developer consoles, to see what data we send and see the responses.

What does asynchronous mean?

AJAX calls are inherently asynchronous; you do not know when they will complete, so they pass data through callbacks.

- The request is made on a background thread that does not block the UI
- This request can finish anytime after it is sent, so you want to hook into your event listener before the request is sent; a request could theoretically complete before you listen for the response.
- The request may not ever complete successfully

Requesting Data (JSON)

With Express, having a route return JSON is very easy.

You can easily request and use JSON data by:

- Make an AJAX request to a route that sends JSON in its response
- Listen for a callback state change
- Check the data to see what you should do (ie, was it successful? Then render; otherwise, show error)
- Manipulate and use the data as needed

Requesting Data (HTML)

Nothing says you **have** to render your HTML with a full HTML layout -- you can actually just return a portion of markup that is intended on being inserted into a page that already exists.

The basic way to request this data is:

- Make an AJAX request to a route that renders HTML for the requested resource
- Listen for a callback state change
- If successful, insert it into the page!

Manipulating data when you send HTML is hard, but the performance is higher as you don't have to generate HTML on your client; your mileage may vary. Sometimes you want to do one, sometimes the other!

Submitting JSON

We can post JSON, as well! This is particularly useful, as you can finally start sending numbers and booleans as well as strings by sending JSON.

In order to POST data in JSON format, you format your request and pass JSON in the body of the request.

Using the bodyparser middleware allows you to have this JSON easily parsed into our request body field in our Express routes and middlewares.

Benefits of using AJAX

Smaller payloads; you can only send down data that the user cares about.

You can split up your code into a more modular setup.

You can keep updating one page, rather than re-requesting entirely new pages all the time!

- Since one page is updating, you will not have to re-request and re-render the same resources such as stylesheets and JS files, making your application often perform much better.

Downsides of using AJAX

Harder to handle search crawlers

- The more your app requires JavaScript, generally, without a prerendering setup, the worse your SEO becomes.

More edge cases

- The more you rely on the client and their browser, the more combinations of things can go wrong; from them having a Chrome extension that somehow interferes with your page, to them losing internet, once it's in their hands anything can happen

Forces you to manually keep track of the state of your app

Have to rebind your event handlers constantly

- See the advanced jQuery section for a note about this!

When would I use AJAX?

AJAX excels at all situations where you have to send small payloads back and forth between the user and the server.

Single Page Applications essentially require AJAX to function; AJAX allows you to send data without leaving the page. This allows the user to keep do things like click a button to save progress but immediately keep working while the progress is still being stored.

On pages that require real time updates, AJAX allows you to successfully stay on one page and just keep updating small bits of data constantly, delivering a seamless user experience.

AJAX jQuery

Making an AJAX request using jQuery

There are several ways of making AJAX requests, however they are all simply shorthand for the $.ajax method, which you can see in *basic_ajax_with_jquery.js*

AJAX requests return promises, as they are asynchronous! The AJAX request method takes an object that allows you to easily POST data to a server.

If the AJAX detects a JSON response, it will automatically serialize it to a JavaScript object.

Storing data attributes

Sometimes, you need to store arbitrary data on your elements, such as an identifier that corresponds to a database entry for that piece of data.

You can use the $(“selector”).data(“key-name”) to get this data, and $(“selector”).data(“key-name”, “new value”) to set it.

Adding HTML to a page

When your server responds with HTML, you can easily target an element and set its HTML.

This allows you to have server-side rendering that is then placed on the page.

Rebinding Events

One curious thing you may notice is that pieces of data that are added to the page after the page loads may not have their events bound!

This is because when you bind events, it binds them to elements that exist; it literally attaches event listeners to each DOM object!

Therefore, you may have to rebind events after adding new content to the page.

Security Concerns

Types of concerns

Due to the public nature of web applications, there are a number of security concerns we face when dealing with Web Programming.

Some common ones are:

- XSS Attack (Cross Site Scripting Attack)
- DOS / DDOS (Denial Of Service / Distributed Denial Of Service)
- SQL Injections
- Phishing attempts
- Brute Forces
- File Inclusion Vulnerabilities

What can we do about it?

For most attacks, tools and strategies have been developed for handling the attacks as best you can. Some of these attacks target users directly, such as Phishing attempts; some of them attack your own system.

It is your job as a web developer to be mindful of these security issues and preemptively protect against them as best you can.

XSS Attacks

An XSS Attack is an injection attack, where a malicious user manages to inject content(typically, JavaScript) into your website.

You can prevent XSS attacks by never displaying raw input that any user on your site may submit; you must always sanitize it and strip HTML from it.

- It's always safer to deny all HTML except for a whitelisted set of tags and attributes, rather than reject tags and attributes that you think should not be allowed.
- You can use the *xss* package to help protect yourself
  - https://www.npmjs.com/package/xss

We have demonstrated XSS attacks before.

DDOS Attack

A DDOS attack is a Distributed Denial of Service attack.

In general, a DOS is when some form of attack renders a server unable to respond in a timely manner.

A DDOS attack is when many machines are being used simultaneously to access a website / server, causing it to fall under such heavy load that it cannot keep up with any of the requests. This renders the server unusable; all requests will timeout.

You can mitigate a DDOS attack by:

- Temporarily upping your available bandwidth to ride it out
- You can use a service such as CloudFlare to handle incoming traffic and prevent suspected DDOSs.

SQL Injections

An SQL Injection is when you allow for a user to sneak their own SQL statements into SQL you're sending to the server. This allows them to attain unauthorized access to your database.

You can prevent this by:

- Sanitizing all strings used as input
- **Using prepared statements**

In our course, as we use MongoDB, we do not have to worry about SQL Injections.

MongoDB is vulnerable to other injection attacks, but they are much more involved and rely on server exploits beyond the scope of this course.

Brute Force Attacks

A Brute Force Attack is when a user attempts to find information about your system / resources in your system (such as users) by providing a constant stream of values.

User logins are particularly vulnerable to Brute Force Attacks.

- A malicious user would target your login form
- They would write a script to submit to that form with known usernames and the most common 10,000 passwords.
- In time, they would be able to successfully harvest usernames and passwords

You can prevent your system from brute force attacks by:

- Tracking the IP of every form submission and limiting based on that
- Lock accounts after some number of failed attempts to access.
- On password login attempts, you can purposefully stall your responses so that the attacker has to wait longer and longer, reducing the effectiveness of their attack.
- Require CAPTCHA style systems.

File Inclusion Vulnerabilities

When you allow users access to resources that are stored in files, you may be tempted to give them urls that have paths to the files stored in the GET string, like so:

- http://localhost:3000/view_story?file=my_story.txt

This leaves you open to a Local File Inclusion!

- You can change *my_story.txt* to be *../../database_credentials.txt* or something similar to access local files, if you do not allow users to select from a limited set of filenames

You're also vulnerable to a Remote File Inclusion!

- You can change *my_file.png* to be *http://my_hacker_site.com/my_attack_vector.txt* in order to execute your own scripts inside their page, which would allow you to gain access to their system!

You can prevent these types of attacks by:

- Referencing resources by ids, that you then lookup files
- Only allowing known, acceptable resource to be included.

How concerned do we have to be?

You have to be constantly aware of these issues, and many more. You are responsible for your users and their data; you should make your best efforts to be secure.

Making a dynamic page

What is a Dynamic HTML Page?

A Dynamic HTML Page is any page that manipulates itself after the client has received it.

Until now, we have used a small amount of JavaScript in order to interact with the page. We have also been creating APIs that are not particularly useful for a user.

We may not combine these two things to start creating robust, single page applications.

Dynamic pages we're familiar with

When you make a new status on Facebook, you never leave the page; instead it just shows up at the top of your page.

On Tumblr, when you scroll down far enough, the page automatically queries new data to show you.

On reddit, when you upvote or downvote a post, it submits that action without leaving the page.

On Gmail, all actions are done via AJAX! You never actually leave the main page, it's all clever JavaScript.

Setting up a Dynamic Page

In order to create dynamic pages easily, you're going to want to start on your server's side by creating API routes that allow you to easily interact with your application; this is generally anything you would submit a form to, or data that you can request.

After that's done, you setup the parts of your website that don't change, such as the main layout.

You can then write your dynamic parts in a JavaScript file, such as:

- On page load, request data to populate the page.
- On a form submission, validate the form
- On a successful form submission, POST it to the server via AJAX
- Use the response to show an error message if there was an error, and show the error on screen
- If there is no error, query the new data to show it on the page.