AJAX

User Interaction

- Our goal is to add more interactivity to our site
- How to have users interact with each other?
    - Form to submit data
    - Page reloads after submission
    - How does a user get updates when someone submits a form? Reload the page?

- We want our sites to update without a refresh

Goal: Chat App

- Let's build a simple chat app
  - Users can send messages
  - All other users can see those messages without taking any action
- We'll need
  - A form to accept chat messages and send them to a path on the server
  - A path to serve the chat history
  - A way to send GET/POST requests without a refresh

AJAX

Asynchronous JavaScript [And XML]

A way to make HTTP requests from JavaScript *after* the page loads

Can make HTTP GET and POST requests

AJAX - HTTP GET Request

```
var request = new XMLHttpRequest();
request.onreadystatechange = function(){
    if (this.readyState === 4 && this.status === 200){
        console.log(this.response);
        // Do something with the response
    }
};
request.open("GET", "/path");
request.send();
```

- Use JavaScript to make an AJAX request
- Create an XMLHttpRequest object
- Call "open" to set the request type and path for the request
- Call send to make the request

JS

AJAX - HTTP GET Request

```
var request = new XMLHttpRequest();
request.onreadystatechange = function(){
    if (this.readyState === 4 && this.status === 200){
        console.log(this.response);
        // Do something with the response
    }
};
request.open("GET", "/path");
request.send();
```

- Set onreadystatechange to a function that will be called whenever the ready state changes
- A ready state of 4 means a response has been fully received
  - In this example, when the ready state changes to 4 and the response code is 200 the response is printed to the console
  - This is where the response would be processed

JS

AJAX - HTTP POST Request

```
var request = new XMLHttpRequest();
request.onreadystatechange = function(){
    if (this.readyState === 4 && this.status === 200){
        console.log(this.response);
        // Do something with the response
    }
};
request.open("POST", "/path");
let data = {'username': "Jesse", 'message': "Welcome"}
request.send(JSON.stringify(data));
```

- To make a post request:
  - Change the method to POST
  - Add the body of your request as an argument to the send method

JS

AJAX - Uses

- We can now make HTTP requests without reloading the page

But why?

AJAX - Uses

Faster page loads

- HTML contains the main structure of the page and very little content
- Any content that takes longer to process is requester via AJAX
    - Request may require database lookups and complex algorithms to generate content
    - Typical when a server is deciding which ad to load
- User sees the page quickly and the content populates as the AJAX responses are sent

AJAX - Uses

Improved user experience (UX)

- Can be disruptive if the page reloads every time you interact with the server
  - Uses bandwidth to repeatedly request all the content
  - Can experience flicker, or worse, when the page reloads
- Allows streaming content

Encodings - Multipart

- As always, we have choices for the format when sending the data of the AJAX request
- We can use multipart formatting by changing the attributes of our forms
  - Add an onsubmit attribute that calls your JavaScript function
    - Add "return false" to block the page reload

```
<form enctype="multipart/form-data" id="myForm" onsubmit="sendMessageWithForm();return false;">
    <label for="form-chat">Chat: </label>
    <input id="form-chat" type="text" name="message"><br/>
    <input type="submit" value="Submit">
</form>
```

Encodings - Multipart

- In JavaScript, create a FormData object using your form element
- Send the FormData object
- Provide the same formatting as submitting the form

```
function sendMessageWithForm() {
    const formElement = document.getElementById("myForm");
    const formData = new FormData(formElement);

    const request = new XMLHttpRequest();
    // onreadystatechange removed for slide

    request.open("POST", "send-message-form");
    request.send(formData);
}
```

Encodings - JSON

- Another option: Manually format the data using JSON
- Don't use the form element
- Create a button instead of a submit input

```
<label for="chatInput">Chat: </label>
<input id="chatInput" type="text" name="message"><br/>

<button onclick="sendMessage()">Send</button>
```

Encodings - JSON

- Manually read the values of any inputs
- Add the values into a JavaScript object or array
- Convert the data to JSON before sending

```
function sendMessage() {
    const chatBox = document.getElementById("chatInput");
    const message = chatBox.value;
    const request = new XMLHttpRequest();

    // onreadystatechange removed for slide

    request.open("POST", "send-message");
    const messageObject = {"message": message};
    request.send(JSON.stringify(messageObject));
}
```

Encodings

- For the assignments
  - You decide how to represent your data

Rendering Content

- There's another new design decision
  - When do we render the content?
  - When do we convert raw data into HTML to be added to a page?

Rendering Content

- We've rendered HTML templates on the server before sending a request
    - Client only sees the final HTML
        - Uses the servers CPU to render content
- This can be restrictive in certain situations
    - What if you want to add a mobile app that doesn't display HTML?

Rendering Content

- Alternative
  - Serve raw data
  - Render it client-side using JavaScript
    - Uses client CPU
      - Increased load times
- Server functionality shifts from hosting the whole web app to hosting an API
  - Serve JSON strings at most paths

Rendering Content

- Again, it's up to for the assignments
  - Render server-side and host HTML
  - Render client-side and host raw data?
- These design decisions must be made for any app

Danger!!

- We will now handling user data and sending it to other users
- This is what we want in order to build features like chat
- But what happens when a user types in chat:
    - "<script>maliciousFunction()</script>"

Danger!!

- "<script>maliciousFunction()</script>"
- This attack is called HTML injection
  - Get scarier when the HTML is a script element (JS Injection)

Danger!!

- To prevent this attack:
  - Escape HTML when handling user submitted data
- Escape HTML
  - Replace &, <, and > with their HTML escaped characters
  - &
  - <
  - >
- These three characters are no longer interpreted as HTML
  - First line of defense against injection attacks

Security Demo