Making HTTP Requests

SWE 432, Fall 2017
Design and Implementation of Software for the Web
Today

• Building responsive layouts with CSS grids
• Making HTTP requests
• Events in frontend JavaScript

• Next time: making and responding to HTTP requests
CSS Grids

DOM Manipulation

Multiply two numbers

3 * 4 = 12

```
<h3>Multiply two numbers</h3>
<div>
  <input id="num1" type="number" /> *
  <input id="num2" type="number" />
  =
  <span id="product"></span>
  <br/>
  <br/>
  <button id="compute">Multiply</button>
</div>
```

```
document.getElementById('compute').addEventListener("click", multiply);
function multiply()
{
  var x = document.getElementById('num1').value;
  var y = document.getElementById('num2').value;
  var productElem = document.getElementById('product');
  productElem.innerHTML = '<b>' + x * y + '</b>';
}
```

“Get the current value of the num1 element”

“Set the HTML between the tags of productElem to the value of x * y”

*Manipulates* the DOM by programmatically updating the value of the HTML content. DOM offers accessors for updating all of the DOM state.
Activity: Build an interactive page

• In groups of 2 or 3
  • Build a 4 function calculator page that lets users add, delete, multiply, and divide

• Key code snippets
  • `document.getElementById('compute')`
  • `elem.addEventListener("click", handler);`
  • `inputElem.value`
  • `elem.innerHTML`
AJAX: Asynchronous JavaScript and XML

- Set of technologies to send and receive data from server asynchronously without interfering with behavior of page
  - HTML & CSS
  - DOM Manipulation
  - JSON or XML for data interchange
  - XMLHttpRequest for asynchronous communication
  - JavaScript
- Originally defined for XML. But representation independent, and now used mostly for JSON.
History

- 1998: Microsoft Outlook Web App implements first XMLHttpRequest script
- 2004: Google releases Gmail with AJAX
- 2005: Google Maps with AJAX
- 2006: W3C releases draft of XMLHttpRequest standard

Synchronous vs. Asynchronous Requests

- Classic web apps require user to wait for response to server
- Asynchronous requests enable user to continue to interact with app
Example - Lazy Content Loading

- User changes visible viewport
  - JS code renders new area of map based on updated viewport
- Check tile cache
  - If in cache, load tile from cache
  - If not in cache,
    - request tile from Google Maps Server
Lazy Content Loading

- Advantages:
  - Can have *vast* dataset that the user feels as if they are interacting with in real time
  - Only need to download content that user actually needs
  - Can (sometimes) do computation on client with really simple server that just fetches appropriate part of large data set
Some Uses for AJAX

• Lazily load content only when requested
  • e.g., FB newsfeed, Google Maps tile loading
• Load parts of web page from different hosts
  • e.g., advertisements, embedded Twitter widget, …
• Persist user data
  • In some cases, can do all computation client side
  • Enables building web app without dedicated backend
• Submit form data to server
Single Page Application Site

Browser

```html
<h1>Single Page Application Site</h1>
<pre>
<code>
&lt;html&gt;
  &lt;head&gt;
    &lt;script src='script.js'&gt;&lt;/script&gt;
  &lt;/head&gt;
  &lt;body&gt;
    &lt;div&gt;Welcome to the single page application site.&lt;/div&gt;
    &lt;script&gt;
      function helloWorld() {
        var message = &quot;Hello, world!&quot;
        document.body.innerHTML = message;
      }
      helloWorld();
      &lt;/script&gt;
  &lt;/body&gt;
&lt;/html&gt;
</code>
</pre>
```

events

HTML

HTML elements

Javascript

```javascript
function helloWorld() {
    var message = &quot;Hello, world!&quot;
    document.body.innerHTML = message;
}
```

HTTP Request

HTTP Response (JSON)

Web Server

Presentation tier

Domain logic tier

Persistence tier

Database
Single Page Application (SPA)

- Client-side logic sends messages to server, receives response
- Logic is associated with a single HTML pages, written in Javascript
- HTML elements dynamically added and removed through DOM manipulation
- Processing that does not require server may occur entirely client side, dramatically increasing responsiveness & reducing needed server resources
- Classic example: Gmail
Making HTTP Requests

- Fetch works on the frontend
- Part of browser API, no need to install
- Some minor differences

```javascript
var myImage = document.querySelector('img);

fetch('flowers.jpg').then(function(response) {
    return response.blob();
}).then(function(myBlob) {
    var objectURL = URL.createObjectURL(myBlob);
    myImage.src = objectURL;
});
```

Posting user data

// Retrieve data from controls on page
var userInput = document.getElementById('userInput').value;

// Send data to remote service
fetch("/apiEndpoint",
{
    headers: {
        'Content-Type': 'application/json'
    },
    method: "POST",
    body: JSON.stringify({ "dataProp": userInput })
})
.then(function(res){ console.log(res) })
.catch(function(res){ console.log(res) });
Demo: Simple todo app
Loading pages

• What is the output of the following?

```html
<script>
    document.getElementById('elem').innerHTML = 'New content';
</script>

<div id="elem">Original content</div>
```
Loading pages

- Code in script tags will run in the order in which it is contained in the page.
- Solution: should put script tags at the bottom of the body after elements in the document.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <div id="container"></div>
  <script src="client.js"></script>
</body>
</html>
```
The Event Loop

```javascript
fetch('https://api.wmata.com/Incidents.svc/json/ElevatorIncidents',
    { headers: { api_key: "e1eee2b5677f408da40af8480a5fd5a8" } })
    .then(function(res) {
        return res.json();
    }).then(function(json) {
        global = json;
    });
```

- Event loop is responsible for dispatching events when they occur
- Simplified main thread for event loop:

```javascript
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```
How do you write a “good” event handler?

• Run-to-completion
  • The JS engine will not handle the next event until your event handler finishes

• Good news: no other code will run until you finish (no worries about other threads overwriting your data)

• Bad/OK news: Event handlers must not block
  • Blocking -> Stall/wait for input (e.g. alert(), non-async network requests)
  • If you *must* do something that takes a long time (e.g. computation), split it up into multiple events using Promises (just like on backend)
Event Dispatching

- Each event target can have (0…n) listeners registered for any given event type, called in arbitrary order.

- What happens with nested elements?

What happens when we click **button**?
Event Bubbling

What happens when we click in button?

• Each event target can have (0…n) listeners registered for any given event type, called in arbitrary order.

• What happens with nested elements?

  Listener1: body onClick
  Listener2: form onClick
  Listener3: button onClick

Called

This is the default behavior.
Event Bubbling

- An individual listener can stop bubbling by calling 
  
  `event.stopPropagation();`

- Assuming that `event` is the name of your handler’s parameter

```html
<body>
  <form>
    <button>Listener3: button onClick</button>
  </form>
  <Listener2: form onClick>
  Listener1: body onClick
```

```html
</body>
```
Web Workers

Main JavaScript task

Create worker

Worker

Web Workers represent new threads of execution
Web Workers

- Web Workers allow you to run arbitrary code in the background, without affecting the performance of your page.

- Web Workers:
  - Must be defined in separate files.
  - Can not access `document`, `window`, or `parent` objects (so no DOM manipulation).
  - **Can** use `fetch`.
  - Should mainly be used for performing long, intensive computation (text parsing, image processing, big data).
  - Communicate with the rest of your app with messages.
Web Worker API

• Defining a new worker
  
  ```javascript
  var worker = new Worker('worker.js');
  ```

• Registering a listener to hear results from the worker
  
  ```javascript
  worker.addEventListener("message", function(e){
    console.log("Message from worker: " + e.data + ">");
  });
  worker.addEventListener("error", function(e){
    console.log("Uh oh");
  });
  ```

• Sending data to the worker
  
  ```javascript
  worker.postMessage("Hello");
  ```

• In the worker: registering for messages from the main thread, sending responses
  
  ```javascript
  self.addEventListener('message', function(e) { doSomething(); });
  self.postMessage("Greetings from the Worker");
  ```

• Including additional scripts:
  
  ```javascript
  importScripts('script2.js');
  ```

• Kill a worker:
  
  ```javascript
  worker.terminate();
  ```
Passing Messages with Web Workers

- Can pass string or object
- Objects are *passed by value*
  - Good news: reduces concurrency programming errors
  - Bad news: passing a big (10’s of MB’s) object will be slow
- Alternative: *transfer* an object
  - No longer exists in the original thread
- Syntax:

  `worker.postMessage(myObject, [myObject]);`
Web Workers: Example

Defining a web worker in worker.js

```javascript
self.addEventListener('message', function(e) {
  self.postMessage("Worker is sending back the text:" + e.data);
}, false);
```

Using a web worker in our web app

```html
<script language="javascript">
  "use strict";
  var worker = new Worker('worker.js');
  worker.addEventListener("message", function(e){
    console.log("Message from worker: <" + e.data + ">");
  });
  worker.postMessage("Hello");
  worker.postMessage("How's it going over there, worker?");
  worker.terminate();
</script>
```
When should you use web workers?

- Mainly for computational stuff:
  - Image manipulation
  - Map routing (without going off to server)
  - Numerical methods
- Remember: can *not* interact with DOM in web worker
Web Workers Demo

Calculating Pi iteratively

```javascript
function CalculatePi(loop) {
    var n=1;
    var c = parseInt(loop);
    console.log(loop);
    for (var i=0, Pi=0; i<=c; i++) {
        Pi=Pi+(4/n)-(4/(n+2));
        n=n+4;
    }
    return Pi;
}
```

[https://gmu-swe432.github.io/lecture8demos/public/WebWorkerDemoFinished.html](https://gmu-swe432.github.io/lecture8demos/public/WebWorkerDemoFinished.html)

[https://github.com/gmu-swe432/lecture8demos/tree/master/public](https://github.com/gmu-swe432/lecture8demos/tree/master/public)
Readings for next time

• React Quick Start:
  • https://reactjs.org/docs/hello-world.html
  • https://reactjs.org/docs/introducing-jsx.html
  • https://reactjs.org/docs/rendering-elements.html
  • https://reactjs.org/docs/components-and-props.html
  • https://reactjs.org/docs/handling-events.html