Computer Science 50
Fall 2008
Scribe Notes

Week 9 Wednesday: November 19, 2008
Andrew Sellergren, Anjuli Kannan

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 9 Wednesday: November 19, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–3:00)

- David is advertising for the Asian-American Brotherhood.[1]
- No announcements today![2]
- Harvard's shuttles now serve downtown Boston![3]
- There's still time left to complete Problem Set 5's scavenger hunt! E-mail your Google map's URL to `sysadmins@cs50.net` before lecture on Monday.

2 SQL Injection Attacks (3:00–11:00)

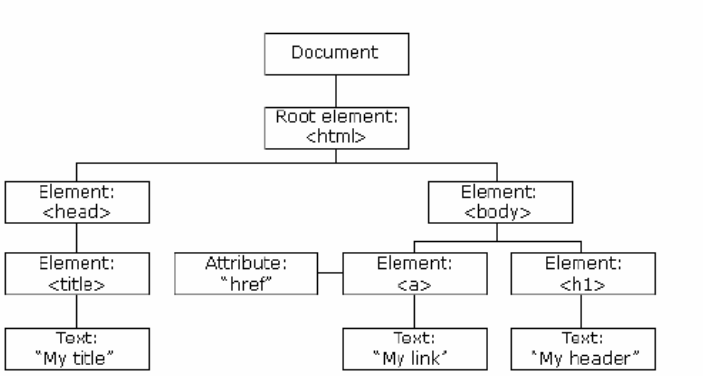
- Why should you follow the example code? SQL injections attacks are one compelling reason.
- Remember the function `mysql_real_escape_string`? Besides being an example of annoying style, it's very useful toward protecting against SQL injection attacks.
- Suppose a malicious user wants to hack into your server and your database. If we pass his input directly to our `mysql_query` function using the `$_POST` array, we've left ourselves vulnerable. What if instead of a password, our user types in a SQL-like string such as `12345' OR '1' = '1` at the login page? Since 1 always equals 1, the user will always be assigned a `uid` even if he doesn't have a valid login. Oops!
- The simple fix is to *escape* any user input that you pass to a database. You can do this by passing the user input to `mysql_real_escape_string` before inserting it into the query to be executed.
- Note that Drew is dominating The Big Board for Problem Set 7 not by any kind of SQL injection attack or malicious input but simply by exploiting the 15-minute delay in CS 50's market prices.[4]

3 DOM and AJAX (11:00–48:00)

- Not to be confused with NOM.
- You should begin to think of every web page as a nested hierarchy of elements, the primary ancestor being the `<html>` tag.

[1] Let it be known that he got to where he is today by being the overeager section guy.
[2] Congratulations to those of you who thought to yourself "Technically, that counts as an announcement!" Click here to claim your prize!
[3] Orly? No, not really.
[4] Too bad for Drew that just as CS 50 has its own imitation market, it also has its own imitation insider trading prison.

- How is a web page represented in memory? That's where the *Document Object Model* comes into play.



  At the top you have the entire Document. Beneath it, the root element, the `<html>` tag. Branching off from that are, as you might expect, the `<head>` and `<body>` tags. And so on.

- Using JavaScript and AJAX and the like, you can manipulate the DOM such that you can grab and update content on demand. Note that Gmail and facebook rarely require an entire window refresh to display new content. They accomplish this using AJAX calls.

- How does Shuttleboy accomplish the animation of the dots representing shuttles? Simply by making a new HTTP request every second or so. We can see this if we use Firefox extensions like Live HTTP Headers or Firebug. What is getting downloaded each time? Only the position of the bus, not the whole map.

- Question: what if the bus moves off the screen? David's implementation punts it to Google's API to decide whether or not to display the red dot.

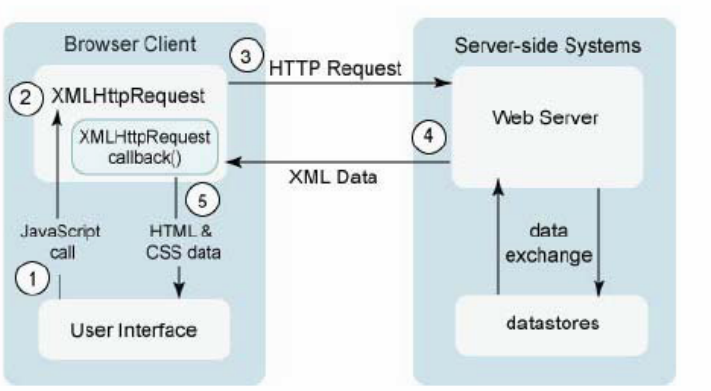
- Why is it that you see a bit of a gray flicker if you move very quickly to new territory on Google Maps? Google has most likely modeled a map as a series of cells or squares. If it hasn't yet downloaded the GIFs or JPEGs for the cells that you've jumped to, it must quickly make a new HTTP request to grab and insert them on the page.

- Question: why did MapQuest and Yahoo originally use the up, down, left, and right buttons? Partly it was that most browsers didn't yet support AJAX but it was also a design decision on the part of Yahoo and MapQuest.

- Question: how could you get rid of the gray flicker? Perhaps you could pre-fetch the cells immediately surrounding the area a user is viewing

because he will most likely move up, down, left, or right within a few seconds. This way, the transition will be seamless. What's the downside of this? More bandwidth is required to achieve this! It's not free (which is probably why Google doesn't do it)!

- Question: doesn't AJAX increase the load on a server? Most certainly, yes! If we all logged in to Shuttleboy, we'd be hitting the FAS server about 300 times per second. This isn't horribly much, but it's not insubstantial.
- It's very easy to make design decisions that result in unmanageable server load. With the CS 50 Office Hours webtool, a query to Google's servers was being made every time the page was refreshed in order to grab the latest OHs data. This ended up hammering the CS 50 server so hard that we had to take it down temporarily in order to improve the implementation.
- Let's take a look at a representation of an AJAX call:



This picture is a bit complicated, but simply consider that the lefthand side represents the client and the righthand side represents the server. If we apply this flow to the Shuttleboy website, we see that it makes a call to a JavaScript function (every second) which uses an XMLHttpRequest object, bundling together HTTP parameters (e.g. the user's desired shuttle stop and destination) in order to make an HTTP request that hits the server, which then talks to its own database, retrieves some data, and returns it to the user interface, in this case the web page itself. Whew.

- Think of the XMLHttpRequest object as a kind of library that has all sorts of functions within it which you can access. Some of those functions, better known as methods in the context of object-oriented programming, are as follows:
  - `abort()`
  - `getAllResponseHeaders()`

  - `getResponseHeader(header)`
  - `open(method, url)`
  - `open(method, url, async)`
  - `open(method, url, async, user)`
  - `open(method, url, async, user, password)`
  - `send()`
  - `send(data)`
  - `setRequestHeader(header, value)`

  This object allows you to open a connection to a server, send data to it, and get a response from it.

- Objects not only have methods associated with them, but also pieces of data, a.k.a. properties. Some of the properties of the XMLHttpRequest object are as follows:

  - `onreadystatechange`
  - `readyState`
    - 0 (uninitialized)
    - 1 (open)
    - 2 (sent)
    - 3 (receiving)
    - 4 (loaded)
  - `responseBody`
  - `responseText`
  - `responseXML`
  - `status`
    - 200 (OK)
    - 404 (Not Found)
    - 500 (Internal Server Error)
  - `statusText`

  The `status` property in particular allows us to check how the HTTP request returned.

- Take a look at `ajax1.html`. Notice that aesthetically, it's a very simple form, but it does manage to accomplish something we haven't yet accomplished: delivering content without a page refresh. If you input a stock symbol and click Get Quote, an alert pops up to display the stock price. If we look at this using Live HTTP Headers, we see that an HTTP request for the file `quote1.php` was made with the stock symbol passed as a `GET` parameter. Take a look at how we achieve this in the body of `ajax1.html`:

```
<body>
  <form onsubmit="quote(); return false;">
      Symbol: <input id="symbol" type="text" />
    <br /><br />
    <input type="submit" value="Get Quote" />
  </form>
</body>
```

Here, we're deliberately telling the form not to submit itself (`return false;`) but rather to make a call to our own function `quote()`. What's the substance of `quote()`? We need only to look to the `<head>` tag to find out:

```
<head>
    <script type="text/javascript">
    // <![CDATA[

        // an XMLHttpRequest
        var xhr = null;

        /*
         * void
         * quote()
         *
         * Gets a quote.
         */
        function quote()
        {
            // instantiate XMLHttpRequest object
            try
            {
                xhr = new XMLHttpRequest();
            }
            catch (e)
            {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
            }

            // handle old browsers
            if (xhr == null)
            {
                alert("Ajax not supported by your browser!");
                return;
            }

            // construct URL
            var url = "quote1.php?symbol=" +
```

```
                document.getElementById("symbol").value;

            // get quote
            xhr.onreadystatechange = handler;
            xhr.open("GET", url, true);
            xhr.send(null);
        }


        /*
         * void
         * handler()
         *
         * Handles the Ajax response.
         */
        function handler()
        {
            // only handle loaded requests
            if (xhr.readyState == 4)
            {
                if (xhr.status == 200)
                    alert(xhr.responseText);
                else
                    alert("Error with Ajax call!");
            }
        }

    // ]]>
    </script>
    <title></title>
</head>
```

Notice we declare a global variable `xhr` before trying to define it as a new instance of an XMLHttpRequest object. We do this using the `try` and `catch` syntax, which is a way of achieving error-handling. In comparison to modern languages, C is somewhat tedious in not having this kind of error-handling mechanism. Consider `copy.c` which was forced to check the return value of almost every function that was called. The `try` and `catch` syntax allows you to sandwich together several lines of code and to deal with any number of different *exceptions* or errors that result.

- What's the deal with two different assignments to `xhr`? Microsoft, liking to be different, has its own version of the XMLHttpRequest object called ActiveXObject. We're kind of abusing the `try` and `catch` syntax here, but effectively we're saying "If instantiating an XMLHttpRequest object fails, assume the user has a Microsoft browser and try to instantiate an

ActiveXObject instead."

- The `NULL` check for `xhr` is in place to ensure that if both attempts to instantiate fail, an appropriate error message will be displayed.

- JavaScript, like PHP, doesn't have strict data typing, so we can simply declare a `var` to hold our URL. To construct our URL, we use the + sign to concatenate whatever the user inputted into the `symbol` form (which we retrieve by calling `getElementById()`) onto the string `quote1.php?symbol=`.

- One advantage of JavaScript over C is *asynchronicity*. When we make a function call in C, we have to wait for it to return before executing the next line of code (unless we implement threading). With JavaScript, however, we can call a function and have it return immediately, calling another function as soon as it returns. Take a look at these lines:

```
xhr.onreadystatechange = handler;
xhr.open("GET", url, true);
xhr.send(null);
```

  What we're doing here is telling the operating environment to call the `handler()` function whenever it's ready to change its state. Then we're telling it to send the `GET` request in an asychronous fashion (using `true` as the third argument to the `open()` method). What does `handler()` do? Notice it makes a call to the `alert()` method (which accomplishes the pop-up) only if the `readyState` property is equal to 4 (which means that the page is loaded) and the overall status is 200 (meaning OK). The argument to `alert()` is the `responseText` property of the XMLHttpRequest object, which, in this case, is the stock price. w00t.

- What about `quote1.php`? Take a look at its code:

```
<?php

    /**
     * quote1.php
     *
     * Outputs price of given symbol as text/html.
     *
     * Computer Science 50
     * David J. Malan
     */

    // get quote
    $handle = @fopen("http://download.finance.yahoo.com/d/
              quotes.csv?s={$_GET['symbol']}&f=e1l1", "r");
    if ($handle !== FALSE)
```

```
    {
        $data = fgetcsv($handle);
        if ($data !== FALSE && $data[0] == "N/A")
            print($data[1]);
        fclose($handle);
    }
?>
```

Here we're calling `fopen()`, which in PHP allows you to open URLs like files and start reading from them. If this call returns true, then we make a call to `fgetcsv()`. CSV stands for *Comma-separated Values*, a very simple way of representing data, separating fields by commas in a flat text file. This type of file can actually be opened by Microsoft Excel such that all the fields will be nicely separated into columns. What Yahoo is doing is distributing their quote data in CSV form so that it can be easily parsed.

- If we access quote1.php?symbol=msft directly from our browser, we get the stock price printed directly to our browser window. However, because we're normally making an HTTP request for this page from within `ajax1.html`, this value won't be printed to our browser window but rather incorporated into an JavaScript alert pop-up thanks to our AJAX call.

4 More Fun with AJAX (48:00–57:00)

- No one likes pop-ups, so how can we achieve this content update in a more elegant fashion? Take a look at `ajax2.html` where it differs from `ajax1.html`:

  ```
  document.getElementById("price").innerHTML = xhr.responseText;
  ```

  Notice this change to the line in `handler()`. Instead of printing the `responseText` property to a JavaScript pop-up, we're going to write it to a DOM element with `id="price"`. We've implemented this element as a `span` rather than a `div` simply because `div`'s include a linebreak: we want our stock price to be displayed in-line. Note that every XHTML element has an `innerHTML` property which refers to any HTML embedded between the open and close tags of that element.

- What's new with `ajax3.html`? So far, we haven't had the best interface because calls to the server might be pretty slow, in which case the user might worry that nothing is happening. It might be nice to implement some kind of animation to let the user know that the page is still working, albeit slowly. We achieve this by creating a new `div` that contains a simple GIF progress bar animation which we display as long as the `readyState` property of the XMLHttpRequest object is not yet 4.

- We're also returning more data from the server, which we access in the `$data` array at indices 1, 2, and 3.

- One syntactic detail: we check `$handle !== FALSE` which invokes the *identical* operator. This operator verifies that a variable is not only equal to a given value but it also has the same type.

5 The Real World (57:00–67:00)

- One of the things you'll be using for Problem Set 8 is Google Maps' API. In the real world, you'll most likely not be writing C, but rather some higher-level language like PHP—and you'll most likely be starting not from scratch but rather from some kind of API.
- While it's true that you pay a performance penalty for writing code in higher-level interpreted languages like PHP, you ultimately might not lose much time given relatively high processor speeds and multiple cores.
- One of the exercises of Problem Set 8 will be to parse a very large dataset in CSV format in order to insert it into a database. This is a very nitty-gritty exercise that mimics everyday tasks in computer science. David, for example, gets data from the registrar every year in CSV format and it's up to him to reformat it so he can insert it in CS 50's database. The quick-and-dirty scripts he writes are meant to save him time, not necessarily to be the fastest implementation possible. Keep in mind that the time it takes to actually write the code needs to be factored in when considering the "efficiency" of the program!
- One of the biggest design decisions you'll make in programming is the very first: what language should I use?
- If we go to the Google Maps Demos, you can see various mash-ups that people have created; for example, a Craigslist functionality that displays all the apartments for rent in a given vicinity.
- What does it take to use Google Maps in our your own website? You sign up for a Google account and receive a developer's key which is embedded in your website so they can keep track of where their code is being used. Then there's nothing to do but start incorporating the basics of their code into your website. One of the goals of introducing JavaScript this week was so that you wouldn't be lost or overwhelmed as you begin to use Google Maps' API Reference.
- If we take a look at the Shuttleboy source code,[5] we see that the array `points[]` gets initialized with all the coordinates of the shuttle stops in latitude and longitude using Google's object `GLatLong`.
- Note that it only took one line of code to actually embed the Google Map in the website:

[5] Jeez, talk about self-promotion.

```
<body color="#c0c0c0" onload="initialize()"
      onunload="GUnload()">
```

What are these functions? These functions are the ones that David wrote himself. `initialize()` begins with a sanity check to make sure the browser is compatible with Google Maps. Then, it checks that there is a `div` with `id="map"` and returns if there isn't. Obviously, if there's no placeholder for the map, then we don't want to continue with our program!

- The code then proceeds through various instantiations which are detailed in the documentation. For each of the shuttle stops, for example, the appearance of the icon is specified and then the method `addOverlay` is called to add them to the map. Simple as that!

6 And Now For Something Completely Different (67:00–76:00)

- Check out this video about the workings of the internet.
- Take your own notes for once! Sheesh, I quit.[6]

[6] David, if you're reading this, know that what I meant to say is: "I most humbly appreciate my lowly role, O Webmaster."