Computer Science 50
Fall 2009
Scribe Notes

Week 9 Wednesday: November 4, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 9 Wednesday: November 4, 2009
Andrew Sellergren

1 Announcements (0:00–7:00, 13:00–17:00)

- This is CS 50.
- 0 new handouts.
- Only 3 lectures left! Better make the most of them![1]
- Final Project Pre-Proposals are due by Monday at 11:00 AM. Realize that they're fairly lowkey—just a chance for you to touch base with your teaching fellow and run your idea by him or her.
- A special seminar has been added to the list: Software Development for the iPhone and iPod Touch. It will be taught by a representative from Apple.
- Dinner with David tonight at 6 PM in Mather!
- Get a Shuttleboy Card today! Available in two editions—Quad or River—this plastic card has the entire shuttle schedule for weekdays and weekends as well as a visualization of the shuttle map itself.
- Coming soon: the CS 50 Store!
- Charles has been superseded on The Big Board. Others seem to have found that they can trade stock volumes rather than actual stocks and boost their net worth artificially. Charles was good enough to find that because our system was storing shares as an `int`, it only allowed a max of 2 billion to be traded at a time. We've since changed that!
- Problem Set 8 is your last problem set ever for this course! Although you may be tempted to skip it if you haven't yet dropped a problem set, we do encourage you to complete it anyway. In many ways, it's the culmination of all your hard work this semester and it introduces you to several new concepts that will be useful to you for the Final Project and beyond.
- Don't forget the Problem Set 5 Scavenger Hunt! We promise the prize will be worth it. A few hints: the apple tree is where prefrosh go and behave positively, the man scratching his chin in front of a Mac is associated with CS 1.

2 Geek Humor (17:00–19:00)

- Check out George Hutchins for U.S. Congress 2010 and Bella De Soto's Website 12/18/05 for examples of great web design!

[1] I suggest wearing a silly hat to Sanders.

3 The Information Superhighway (7:00–13:00)

- Yes, I did just make that the title of this section.
- If we SSH into any server and run the command `traceroute` followed by a website like `cnn.com`, we can see all the hops in between our current server and the destination server.
- The first such hop will be a server with address 140.247.x.y. This is a Harvard server, as Harvard owns all the IP addresses that begin with those two numbers. All told, Harvard owns 65,536 IP addresses. Unfortunately, we pale in comparison to MIT, who own all the IP addresses of the form 18.x.y.z. This means they have millions at their disposal.
- In David's lecture example, we jumped from Harvard's campus to somewhere in New Jersey, followed by Washington, D.C. and Atlanta. Realize that you probably won't be able to replicate this exact route if you run this command again since packets can follow any number of different paths from source to destination.
- If we see an entry like `* * *`, it means that particular router has blocked information about itself.
- The overall time of the request will be tabulated in the far right column. In class, David's request took 27.457 milliseconds to reach `cnn.com`, which was somewhere in the Atlanta area.
- What's even more interesting is if we run `traceroute` on `cnn.co.jp`, the Japanese version of CNN. Around steps 9 or 10, we see that the request time jumps up significantly. If we examine the names of the servers at these steps, we can infer that this was the jump across the Pacific Ocean. Pretty amazing that it can be traversed in approximately 100 milliseconds.
- Check out this Warriors of the Net video, which is a half-fun, half-serious look at how the internet really works.

4 Ajax (20:00–60:00)

- As we saw last time, it's possible to have a form submit to itself using PHP. In that way, server-side validation can be performed without risking losing all of the information the user has inputted.
- Another way of implementing a seamless user interface is using Ajax. In `ajax1.html`, we can see that entering a symbol and clicking Get Quote causes an alert window to pop up with the stock price. We can see that the URL has not changed at all, so we know the form hasn't actually submitted, yet we are retrieving information. Let's take a look at the source code to see how this is done:

```
<!--

ajax1.html

Gets stock quote from quote1.php via Ajax, displaying result with alert().

Computer Science 50
David J. Malan

-->

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <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)
            {
                // display response if possible
                if (xhr.status == 200)
                    alert(xhr.responseText);
                else
                    alert("Error with Ajax call!");
            }
        }

    // ]]>
    </script>
    <title></title>
  </head>
  <body>

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

Down at the bottom, we see that there's very little XHTML that's needed to implement the actual form. One thing to notice is that the `action` attribute has been left blank. In order for our page to validate, we need to make sure this attribute actually exists, but we don't have to put a real filename there. Clearly if this attribute is blank, the form isn't actually going to submit anywhere. Instead, we have a JavaScript function called `quote()`, specified in the `onsubmit` attribute, which is going to look up the stock price. After this function executes, we're going to return false so that the form doesn't actually submit.

- Ajax is a technology which allows browsers to make additional requests to the server after the web page has already loaded. Unfortunately, browsers never agreed upon how to implement Ajax, so we have to use some muddy syntax in order to ensure cross-browser compatibility. First, we're initializing a global variable named `xhr` by trying to create a new XMLHttpRequest object. Unfortunately, this won't work in Internet Explorer because Microsoft decided that their particular flavor of this object would be called an ActiveXObject. For that reason, we use the try-catch syntax, which attempts to execute the try block and only executes the catch block if the try block fails for some reason.

- After we've initialized `xhr`, we check for null just in case the user is running a browser that doesn't support Ajax. Next we're dynamically creating a URL which we're going to request from the server. In a `GET` variable named `symbol`, we're appending the value the user has entered into the text box. We are accessing this value by invoking a method called `getElementById`, which, as you might've guessed, searches for an HTML element whose `id` attribute we specify. In this case, we've given the text box an `id` of `symbol`, so that's what we're searching for.

- The three lines at the bottom of `quote()` are the ones which actually retrieve the stock quote. First we're telling `xhr` that once its done making its request, call a function named `handler()` that we will write ourselves. The last two lines actually open a connection to the server and send the data. If you wanted to use the `POST` method, you would specify `POST` as the first argument to `open()` and you would pass the actual data as the argument to `send()`, rather than null.

- So let's actually see what this URL will return if we access it directly. If we navigate to `quote1.php?symbol=GOOG`, we get back nothing but a stock quote—no HTML markup, even. It is our `handler()` function which will be manipulating this directly.

- Within the `handler()` function, we are checking two properties of the `xhr` object: `readyState` and `status`. First, we check `readyState` to find if the request has been sent successfully and second, we check `status`, to see if the server has returned a response of OK. If both of those checks

are passed, then we access the `responseText` of the object and display it via an alert window.

- Slightly more sophisticated than an alert window would be to embed the response in the actual XHTML of the webpage. Check out `ajax2.html`:

```
<!--

ajax2.html

Gets stock quote from quote1.php via Ajax, embedding result in page itself.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <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)
            {
                // embed respose in page if possible
                if (xhr.status == 200)
                    document.getElementById("price").innerHTML = xhr.responseText;
                else
                    alert("Error with Ajax call!");
            }
        }

    // ]]>
    </script>
    <title></title>
</head>
<body>

    <form action="" onsubmit="quote(); return false;">
      Symbol: <input id="symbol" type="text" />
```

```
        <br />
        Price: <span id="price"><b>to be determined</b></span>
        <br /><br />
        <input type="submit" value="Get Quote" />
      </form>

    </body>
</html>
```

Where the stock price will go, we have an element of type `span`. This is similar to a `div` in that we can put almost anything inside it, but a `div`, being a block-level element, takes up the whole width of the window whereas a `span`, being an in-line element, does not.

- In this version, when we click Get Quote, the text "to be determined" gets replaced by the actual stock quote. Interestingly, even after the stock quote is displaced, if we view the web page's source, we see that the `span` still contains the text "to be determined." JavaScript can change what's displayed by the browser, but it doesn't change what was originally sent by the server.

- The only difference between the JavaScript in `ajax2.html` and that in `ajax1.html` is the `handler()` function. Now when the Ajax request returns successfully, we're changing the `innerHTML` property of the `price` element rather than popping up an alert window. `innerHTML` initially holds the text "to be determined" (within a `b` tag) which we will clobber with the stock price we just looked up.

4.1 Hiding and Showing HTML Elements

- Ajax stands for *asynchronous JavaScript and XML*. What these buzzwords really mean is that JavaScript is being used to retrieve data that the rest of the page isn't waiting on—hence the asynchronicity.

- While data is being retrieved asynchronously, it's often a good idea to convey to the user, via a progress indicator, that the data is on its way. A quick Google search for Ajax progress bars will yield sites like this that allow you to create your own progress bar.

- Recall that `quote1.php` is a simple ripoff of distribution code from Problem Set 7 which dynamically generates a URL with a stock symbol appended then hits that URL and parses the stock quote from Yahoo's response. Other than this file, we're actually not using any PHP. So far, all of our Ajax files have been nothing but XHTML and JavaScript.

- Let's take a look at `ajax3.html`:

```
<!--

ajax3.html

Gets stock quote (plus day's low and high) from quote2.php via Ajax,
embedding result in page itself after indicating progress with an
animated GIF.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <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 = "quote2.php?symbol="
                 + document.getElementById("symbol").value;

            // show progress
            document.getElementById("progress").style.display = "block";

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


        /*
         * void
         * handler()
         *
         * Handles the Ajax response.
         */
        function handler()
        {
            // only handle requests in "loaded" state
            if (xhr.readyState == 4)
            {
                // hide progress
                document.getElementById("progress").style.display = "none";

                // embed response in page if possible
                if (xhr.status == 200)
                    document.getElementById("quote").innerHTML = xhr.responseText;
                else
                    alert("Error with Ajax call!");
            }
        }

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

```
    <form action="" onsubmit="quote(); return false;">
      Symbol: <input id="symbol" type="text" />
      <br /><br />
      <div id="progress" style="display: none;">
        <img alt="Please Wait" src="pleasewait.gif" />
        <br /><br />
      </div>
      <div id="quote"></div>

      <br /><br />
      <input type="submit" value="Get Quote" />
    </form>
  </body>
</html>
```

In the actual XHTML source, we see that the the progress bar GIF is actually already embedded. But because the `div` which contains it has its CSS property `display` set to `none`, it won't actually be visible when the page is first loaded. If we examine the JavaScript, we see that it's almost identical to `ajax2.html`, except for two lines, one in the `quote` function which sets the `display` property to `block`, and one in the `handler` function which sets this `display` property back to `none`.

- Realize that the GIF animation is not beginning when we click Get Quote. The animation is actually built into the GIF, which has been in the background the whole time. We simply make it visible when we click Get Quote.
- We can see this same show/hide functionality in HarvardEvents. When you click on any given event, there's no request being made to the server. Rather, the paragraph description was downloaded when the page first loaded and clicking simply reveals the description by changing its `display` property to `block`.
- We're also displaying more than just the stock price at this point. Let's take a look at how we do that in `quote2.php`:

```
<?php

    /**
     * quote2.php
     *
     * Outputs price, low, and high of given symbol as text/html, after
     * inserting an artificial delay.
     *
     * Computer Science 50
     * David J. Malan
```

```
     */


    // pretend server is slow
    sleep(5);

    // try to get quote
    $handle = @fopen("http://download.finance.yahoo.com/d/quotes.csv?" .
        "s={$_GET['symbol']}&f=e1l1hg", "r");
    if ($handle !== FALSE)
    {
        $data = fgetcsv($handle);
        if ($data !== FALSE && $data[0] == "N/A")
        {
            print("Price: {$data[1]}");
            print("<br />");
            print("High: {$data[2]}");
            print("<br />");
            print("Low: {$data[3]}");
        }
        fclose($handle);
    }
?>
```

You can see that instead of simply returning a single number, `quote2.php` is actually spitting out some XHTML. This XHTML is what we will dynamically insert into `ajax3.html` when the Ajax call returns successfully.

4.2 Anonymous Functions

- Having a function named `handler` in our previous examples feels a little sloppy because we're only using this function once. As an alternative, we might define it as an anonymous function and assign it directly as the state change event handler for our Ajax request, as we do in `ajax4.html`:

```
<!--

ajax4.html

Gets stock quote from quote1.php via Ajax, displaying result with alert().
Implements handler as an anonymous function.

Computer Science 50
David J. Malan

-->
```

```
<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <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 = function () {
                // only handle loaded requests
                if (xhr.readyState == 4)
```

```
                {
                    // display response if possible
                    if (xhr.status == 200)
                        alert(xhr.responseText);
                    else
                        alert("Error with Ajax call!");
                }
            };
            xhr.open("GET", url, true);
            xhr.send(null);
        }

    // ]]>
    </script>
    <title></title>
  </head>
  <body>

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

This way of structuring our code certainly has its appeal as its more compact and yet still readable. The code within the function is actually identical to what it was in `handler()`.

4.3 JSON

- In `ajax5.html`, we have three `span` elements with unique `id`'s, each of which will be filled with corresponding data from our Ajax request.
- Now that we have three different placeholders, we can't simply insert a chunk of XHTML into our web page. Instead, we'll need to be able to parse the Ajax response. The data structure that will make this possible is familiar from Problem Set 6: a hash table. In `quote3.php`, we'll be creating a hash table—an associative array—with values corresponding to the price, the high, and the low for a given stock symbol. Then we're going to encode this as a JavaScript object using *JavaScript object notation* (JSON):

```
<?php

    /**
```

```
     * quote3.php
     *
     * Outputs price, low, and high of given symbol as JSON.
     *
     * Computer Science 50
     * David J. Malan
     */

    // try to get quote
    $quote = array();
    $handle = @fopen("http://download.finance.yahoo.com/d/quotes.csv?" .
        "s={$_GET['symbol']}&f=e1l1hg", "r");
    if ($handle !== FALSE)
    {
        $data = fgetcsv($handle);
        if ($data !== FALSE && $data[0] == "N/A")
        {
            $quote["price"] = $data[1];
            $quote["high"] = $data[2];
            $quote["low"] = $data[3];
        }
        fclose($handle);
    }
    header("Content-type: text/javascript");
    print(json_encode($quote));
?>
```

Once we do a quick sanity check on the return values from Yahoo's server, we create an associative array with keys of `price`, `high`, and `low`. Passing this array to the `json_encode()` function will create a JavaScript object with properties of the same names. The syntax for this object might look something like the following:

```
{"price":"566.76","high":"568.78","low":"562.00"}
```

In `ajax5.html` we access these values after evaluating the Ajax response text as JSON:

```
var quote = eval("(" + xhr.responseText + ")");
document.getElementById("price").innerHTML = quote.price;
document.getElementById("high").innerHTML = quote.high;
document.getElementById("low").innerHTML = quote.low;
```

The `eval` function takes the JSON and creates a corresponding object in memory.

- Kayak is an example of a site which makes extensive use of Ajax to create a dynamically updated interface. When you search for a flight or hotel on Kayak, the results appear to be added one by one, the lowest price at the top. What's most likely going on behind the scenes is that each of these prices is being retrieved by an Ajax request, a response is being evaluated as JavaScript, and the appropriate content is being inserted into the DOM where it belongs. Applying the filters doesn't change the URL, either, so it would seem that JavaScript is being used to sort the results.

5 Google Maps API (60:00–75:00)

- Using the Google Maps API is as easy as providing an e-mail address and a domain where you will use it and embedding in your website the registration key they give you. After that, the entirety of their code base is available to you. Feel free to browse the API Reference to see just how much they offer.
- `map1.html` isn't very impressive, but it does demonstrate the very basics of embedding a Google Map in a webpage:[2]

```
<!--

map1.html

Demonstrates a "hello, world" of maps.

Computer Science 50
David J. Malan

-->


<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?
       file=api&amp;v=2&amp;key=ABQIAAAA8igYd929VTmOEMLNjNyP1xQBp
       XogqfcSB7goBPOHTPn2IeSNTRQdD9mEVL94XCoVb0q6KdbS4ouIDA"
               type="text/javascript"></script>
    <script type="text/javascript">
    //<![CDATA[

        function load()
```

[2]The `src` attribute of the first JavaScript link has been spread across multiple lines.

```
        {
            if (GBrowserIsCompatible())
            {
                var map = new GMap2(document.getElementById("map"));
                map.setCenter(new GLatLng(37.4419, -122.1419), 13);
            }
        }

    //]]>
    </script>
    <title></title>

  </head>
  <body onload="load()" onunload="GUnload()">
    <div id="map" style="width: 800px; height: 500px"></div>
  </body>
</html>
```

Two new attributes for the `body` element are `onload` and `onunload`. What we're telling the browser to do is to call our custom `load` function when the page first renders and to call Google's `GUnload` function when the page is "unloaded," for example, when the user navigates away from the page.

- The entire content of the page is a single `div`. If the `load` function isn't called, then there's actually no content on the page. Within the `load` function, we first check if the user's browser is supported by Google Maps. Once we've passed this check, a single line of code, instantiating a `GMap2` object, is all it takes to create a Google Map. To its constructor, we simply pass the HTML element in which we want to insert the map, in this case the HTML element with the `id` of `map`.

- Because `map` is now a JavaScript object, it has not only properties associated with it, but also methods, one of which is `setCenter`. We can call this method by using the dot notation just as we did before when accessing the properties of `quote`. The `setCenter` method takes two arguments: a latitude/longitude point and a zoom level. We can tweak either to see where it takes us.

- You'll notice in this basic example, we haven't enabled zoom control. `map3.html` takes care of this, as well as numerous other controls, with a few more lines of code:

```
function load()
{
    if (GBrowserIsCompatible())
    {
        // instantiate map
```

```
        var map = new GMap2(document.getElementById("map"));

        // center map on Science Center
        map.setCenter(new GLatLng(42.376649, -71.115789), 13);

        // add control using a local variable
        var typeControl = new GMapTypeControl();
        map.addControl(typeControl);

        // add another control without using a local variable
        map.addControl(new GLargeMapControl());

        // enable scroll wheel and smooth zooming
        map.enableScrollWheelZoom();
        map.enableContinuousZoom();
    }
}
```

These lines of code are pretty easy to pick up just from reading the manual.

- Another easy feature to add to our map is the familiar red markers which, when clicked, pop up an information window. We do this in `map4.html` using a function called `createMarker`:

```
var map;

function load()
{
    if (GBrowserIsCompatible())
    {
        // instantiate map
        map = new GMap2(document.getElementById("map"));

        // prepare point
        var point = new GLatLng(42.376649, -71.115789);

        // center map on Science Center
        map.setCenter(point, 13);

        // mark Science Center
        createMarker(point);
    }
}

function createMarker(p)
{
```

```
        var marker = new GMarker(p);
        map.addOverlay(marker);

        // associate info window with marker
        GEvent.addListener(marker, "click", function() {

            // prepare XHTML
            var xhtml = "<b>Science Center</b>";
            xhtml += "<br /><br />";
            xhtml += "<a href='http://en.wikipedia.org/wiki/Harvard_Science_Center'
                  target='_blank'>";
            xhtml += "http://en.wikipedia.org/wiki/Harvard_Science_Center";
            xhtml += "</a>";

            // open info window
            map.openInfoWindowHtml(p, xhtml);

        });
    }
```

Now, we save the latitude/longitude (`GLatLng`) object we create as a variable named `point`, which we pass to `createMarker`. Within `createMarker`, we add a `GMarker` object to the map and also add an event listener to it that will handle a user's click. This listener creates some XHTML on the fly and passes it to the `openInfoWindow` method. Know that these lines of code will only be executed when the user clicks the marker, not when the page first loads.

- Take a look at `map5.html`, which combines the Ajax request we used previously in order to dynamically display the stock price of Google within an information window.