Computer Science 50   Week 10 Monday: November 8, 2010
Fall 2010   Andrew Sellergren
Scribe Notes

Contents

Computer Science 50   Week 10 Monday: November 8, 2010
Fall 2010   Andrew Sellergren
Scribe Notes

1   Announcements and Demos (0:00–7:00)

- This is CS50.

- 2 new handouts.

- Final Project Pre-proposals were due earlier today. If you haven't gotten in touch with your TF about your ideas, please do so ASAP! If you're looking for ideas or a little help with a new topic, check out projects.cs50.net and the CS50 Seminars.

- This Friday is the deadline for Problem Set 5's scavenger hunt. A special prize will be awarded to the section that finds all of the TFs on campus!

- Be sure to get your hands on some CS50 apparel from the store. A CS50 sweatshirt is the perfect way to brag to your friends and family and to pick up the ladiez and the fellas.

- If you're interested in becoming a TF or CA next year[1], apply by following the link on the course's homepage that will be up before long.

2   From Last Time (7:00–28:00)

- Last time, we began looking at a technology called Ajax which allows for asynchronous HTTP requests to be made to web servers so that content can be updated without refreshing the page. Ajax formerly stood for *asynchronous JavaScript and XML*, but now that the technology has evolved to involve JSON as well as XML, Ajax doesn't stand for anything.[2]
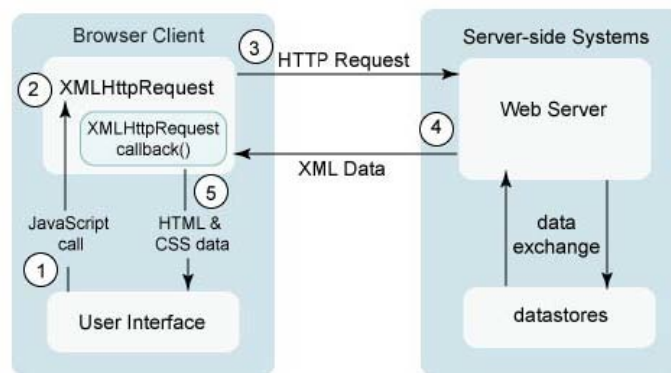
  The magic of Ajax is encapsulated in the XMLHttpRequest object which actually makes the HTTP request and calls the handler when the HTTP response is received.

---

[1]Seriously, who wouldn't be?
[2]Just like the "j" in "djm."

2.1   Overview of the Internet

- In the past few weeks, we also looked at IP, the protocol responsible for the unique addresses assigned to every computer on the internet, as well as TCP, the protocol which ensures reliability by requiring dropped data to be resent. Another protocol called UDP is less reliable than TCP for transmitting messages because it doesn't require dropped data to be resent, but it has its own uses. For example, when watching streaming video, a few dropped frames might not be as important as keeping up with real time. Finally, we talked briefly about routers and switches which enable multiple computers to share the same IP address. All of these terms will be discussed in more depth in Warriors of the Net.

- Information is sent between server and client in packets of a fixed size which contain the sender and receiver's IP addresses as well as what type of packet it is. Taking the client to be your home computer in this case, the packet will be sent out on the local area network (LAN) which encompasses all of the computers in your house. Your home router will direct packets toward the next router on the path toward their destination. If the client here were a corporate computer, it might be behind a proxy server which throttles and filters packets (e.g. blocking traffic to Facebook) as well as a firewall (which is generally built in to your home router) that prevents access from outside and leakage from inside. Once the packet reaches the next router, it will be bounced along to router after router (as we saw with `traceroute`) until it reaches its final destination. At its final destination, it will most likely encounter another firewall which only allows in packets destined for certain ports (e.g. 80 for web traffic and 25 for e-mail). After getting past the firewall, the packet is read by the web server which passes the data to the actual application (e.g. the PHP interpreter) that generates a response. In turn, the response is bundled up into a packet which is transmitted back to the client in much the same way as the request was.

3   Problem Set 8 (28:00–45:00)

- Initially, the plan for Problem Set 8 was to use the Google Charts API to create a website like HarvardEnergy. However, as David was having trouble getting excited about bar charts, he figured you might have trouble as well. So instead, we'll be creating CS50 Shuttle, a video game implemented with the Google Earth and Google Maps APIs. Check out the staff solution!

- The goal of the CS50 Shuttle game is to pick up passengers (your TFs) and drop them off at their destinations on Harvard's campus. Different keystrokes will take different actions such as changing the camera view or actually moving the shuttle and clicking the Pick Up button will fill your

seats. All of this is done with JavaScript! There are even Easter eggs in
CS50 Shuttle. Enter the Konami Code and you'll find that you can fly!

- For Problem Set 8, you'll be given the foundation of the video game itself,
  but with several key pieces missing. For example, you'll need to imple-
  ment the ability to actually pick up passengers, i.e. pluck their name and
  place it into your passenger list. Each of the passengers is actually just a
  JavaScript object.

- Some of the optional features you might implement are the Konami Code,
  autopilot, teleportation, a points system, and a timer for passenger dropoff.

- This problem set will empower you to work with third-party APIs. We
  think you'll find that Google's are quite well documented and the examples
  they provide are fairly easy to follow.

- To get started with Problem Set 8's distribution code, we'll need to run the
  following commands from our `public_html` directory, as the spec details:

```
cp -r ~cs50/pub/src/pset8
chmod 711 pset8/
cd pset8
chmod 644 *.js *.html *.css
```

  These commands will copy the distribution code, make it executable by
  all, and make the JavaScript, HTML, and CSS files readable by all.

- In general, it's a good design practice to separate the form and function
  of your website. You should have separate files for your JavaScript, CSS,
  and HTML.

3.1   index.html

- One line in the distribution code is a `script` tag that references a JavaScript
  file on Google's servers. This is the file that actually implements the map
  itself. The other `script` tags reference files on our own web server:

```
<script src="math3d.js" type="text/javascript"></script>
<script src="buildings.js" type="text/javascript"></script>
<script src="houses.js" type="text/javascript"></script>
<script src="passengers.js" type="text/javascript"></script>
<script src="shuttle.js" type="text/javascript"></script>
<script src="service.js" type="text/javascript"></script>
```

3.2   buildings.js

- Let's take a closer look at `buildings.js`:[3]

---

[3]Note this is just a sample from the actual file, which has several hundred buildings in the
array

```
/*******************************************************************************
 * buildings.js
 *
 * Computer Science 50
 * Problem Set 8
 *
 * Buildings on campus.
 ******************************************************************************/

var BUILDINGS = [
 {
  root: "04558",
  name: "1 Bow Street",
  address: "1238 Massachusetts Ave",
  lat: 42.372341075,
  lng: -71.116148412
 },
 {
  root: "02121",
  name: "1 Brattle Square",
  address: "1 Brattle Sq",
  lat: 42.373523361,
  lng: -71.121240176
 },
 {
  root: "05599",
  name: "Wyss Institute At Clsb",
  address: "3 Blackfan Cir",
  lat: 42.339406779,
  lng: -71.104157549
 }
];
```

BUILDINGS is actually an array of objects, each of which represents a building object that encapsulates its name, address, coordinates, etc. Each building object is really just JavaScript's implementation of a hash table. The keys `root`, `name`, `address`, etc. are mapped to corresponding values.

- `passengers.js` and `houses.js` are very similar to `buildings.js` except that they contain arrays `PASSENGERS` and `HOUSES` that have identifying information for all the CS50 TFs and Harvard houses. `HOUSES` is actually a multi-dimensional object wherein each house name is a key mapped to an object that contains `lat` and `lng` keys. This structure allows us to use the object as a lookup table for the latitude and longitude coordinates of a house. As you can see, creating objects and arrays is as simple as writing curly braces and brackets.

3.3 `index.html` (cont'd)

- The actual source code in `index.html` is actually just a layout for the
  page, namely spaces for the 3D map, the 2D map, and the passenger list.
  For each of these, we use a `div` with certain style attributes specified
  in separate CSS files. To make it easier to manipulate these `div`'s in
  JavaScript, we specify the `id` attribute for each. Having done so, we can
  quickly grab these elements using the `getElementById` function. In fact,
  embedding a Google Earth map in your web page is as simple as creating
  a `div` with an `id` of your choice and then passing your chosen `id` to one
  of Google's JavaScript functions.

- A lot of the magic on our page are triggered by events tied to the `body`
  element:

  ```
  <body onkeydown="return keystroke(event, true);"
        onkeyup="return keystroke(event, false);"
        onload="load();" onunload="unload();">
  ```

  The `onkeydown` and `onkeyup` events are tied to the `keystroke` function
  in `service.js` and the `onload` and `onunload` events are tied to the `load`
  and `unload` functions respectively. Let's take a look at `service.js` to see
  how these functions are implemented.

3.4 `service.js`

- At the top of the file, we declare multiple global variables, which generally
  isn't a great habit to get into but can sometimes be okay as long as you
  don't anticipate your code being portable to other applications. Among
  these global variables we declare are `bus`, `earth`, `map`, and `shuttle`.

- First, we'll tease apart the `load` function:

  ```
  /*
   * void
   * load()
   *
   * Loads application.
   */

  function load()
  {
      // embed 2D map in DOM
      var latlng = new google.maps.LatLng(LATITUDE, LONGITUDE);
      map = new google.maps.Map(document.getElementById("map"), {
       center: latlng,
       disableDefaultUI: true,
  ```

```
      mapTypeId: google.maps.MapTypeId.ROADMAP,
      navigationControl: true,
      scrollwheel: false,
      zoom: 17
    });

    // prepare shuttle's icon for map
    bus = new google.maps.Marker({
      icon: "http://maps.gstatic.com/intl/en_us/mapfiles/ms/micons/bus.png",
      map: map,
      title: "you are here"
    });

    // embed 3D Earth in DOM
    google.earth.createInstance("earth", initCB, failureCB);
}
```

The `new` keyword which we use to instantiate the `latlng` variable in the
first line asks the JavaScript interpreter to give us a fresh instance of an
object. In this case, the object is one of type `google.maps.LatLng`.

- In the second line, we initialize the global `map` variable by omitting the key-
  word `var` in front of it. By making this variable global, we can reference it
  inside other functions without passing it as an argument. To initialize this
  `Map` object, we pass as the first argument the HTML element where we
  want to place it. We grab this HTML element using the `getElementById`
  function. Even though HTML is just text, recall that it is structured in
  such a way that JavaScript can create a hierarchical tree object under
  `document` with which to access its inner elements. The second argument
  to the `Map` initializer is what's called an *object literal*. The syntax is the
  exact same as if we were declaring an object, but we're not giving it a
  name. In this case, the `Map` initializer understands certain keys (as spec-
  ified in the documentation) which represent display options for the map.
  This ability to pass object literals as arguments to JavaScript functions
  is tremendously useful in eliminating optional arguments to functions. In
  this case, as you can see, without the object literal, we would be passing
  6 or more arguments to the `Map` initializer.

- The last line of code in this function is interesting to examine. As the
  second and third arguments, we pass the objects `initCB` and `failureCB`,
  which are actually callback functions. That is, they are functions which
  we'd like JavaScript to execute in the case that the `createInstance` call
  succeeds or fails, respectively. In JavaScript, functions can be treated
  just like any other object and thus can be passed as arguments to or
  even returned by other functions. Notice that when we pass `initCB` and
  `failureCB` as arguments, we do **not** put parentheses after them. If we did,

Computer Science 50   Week 10 Monday: November 8, 2010
Fall 2010   Andrew Sellergren
Scribe Notes

the functions would actually be executed right then and there. However, as you can see in its definition, `initCB` actually takes a single argument:

```
/*
 * void
 * initCB()
 *
 * Called once Google Earth has loaded.
 */

function initCB(instance)
{
    // retain reference to GEPlugin instance
    earth = instance;

    // specify the speed at which the camera moves
    earth.getOptions().setFlyToSpeed(100);

    // show buildings
    earth.getLayerRoot().enableLayerById(earth.LAYER_BUILDINGS, true);

    // prevent mouse navigation in the plugin
    earth.getOptions().setMouseNavigationEnabled(false);

    // instantiate shuttle
    shuttle = new Shuttle({
     heading: HEADING,
     height: HEIGHT,
     latitude: LATITUDE,
     longitude: LONGITUDE,
     planet: earth,
     velocity: VELOCITY
    });

    // synchronize camera with Earth
    google.earth.addEventListener(earth, "frameend", frameend);

    // synchronize map with Earth
    google.earth.addEventListener(earth.getView(), "viewchange", viewchange);

    // update shuttle's camera
    shuttle.updateCamera();

    // show Earth
    earth.getWindow().setVisibility(true);
```

```
        // populate Earth with passengers and houses
        populate();
}
```

Basically, as the API documentation will indicate, `createInstance` plans
to pass that single argument to the function we've provided, so we must
include it in our definition. In this case, it's a reference to the object that
represents the actual Google Earth plugin.

- Within `initCB`, we instantiate a new `Shuttle` object to one of our global
  variables. Roughly speaking, `Shuttle` is a class we've created, though
  technically JavaScript doesn't actually support classes, only prototypes.
  We've decided that the `Shuttle` object should have attributes such as
  `heading`, `height`, `latitude`, `longitude`, and more. The code that defines
  the `Shuttle` object lives in `shuttle.js`.

- The rest is up to you!