Problem Set 8: Mashup

This is CS50. Harvard University. Fall 2014.

Table of Contents

Objectives

- Introduction to JavaScript, Ajax, JSON.
- Exposure to objects and methods.
- Grapple with real-world APIs and libraries.

Recommended Reading

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter to the Administrative Board and the outcome is Admonish, Probation, Requirement

to Withdraw, or Recommendation to Dismiss, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

If you commit some act that is not reasonable but bring it to the attention of the course's heads within 72 hours, the course may impose local sanctions that may include an unsatisfactory or failing grade for work submitted, but the course will not refer the matter to the Administrative Board except in cases of repeated acts.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code online so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Ready

Your (final!) mission for this problem set is to implement "mashup" that integrates Google Maps with Google News with a MySQL database containing thousands of postal codes, GPS coordinates, and more. Quite like this here version by the staff!

http://mashup.cs50.net/

Not only can you search for places via the text box up top, you can also click on and drag the map elsewhere. Scattered across the map, meanwhile, are newspaper icons that, when clicked, provide links to local news!

You may notice that some markers (and labels) overlap others or are otherwise at the wrong coordinates. The GeoNames geographical database is actually imperfect, whereby some places' coordinates are off. For instance, East Boston isn't in Back Bay. And Readville isn't in Boston Harbor. Not to worry if you see those same symptoms in your mashup, assuming the source of the problem is indeed the data itself!

Anyhow, how neat! But where to begin?

Google Maps

Odds are you're already familiar, but surf on over to Google Maps anyway at https://www.google.com/maps. Input **42.3770, -71.1256** into the search box up top, and you should find yourself near Harvard. Interesting! It seems Google Maps understands GPS coordinates (i.e., latitude and longitude). In fact, search for **28.410, -81.584**. Perhaps you'd rather be there? (You might need to zoom out.)

It turns out that Google Maps offers an API that allows you to embed Google's maps into your own web apps. Hey, that's one of the ingredients we need! Go ahead and familiarize yourself with Google Maps Javascript API v3[1] by perusing the three sections below of its Developer's Guide. Read through any sample code carefully, clicking **View example** below it, if present, to see the code in action.

- Getting Started[2]

[1] https://developers.google.com/maps/documentation/javascript/
[2] https://developers.google.com/maps/documentation/javascript/tutorial

- Drawing on the Map
  # Markers[3]
  # Info Windows[4]

Google News

Okay, now we need us some news. If you happen to have a Google Account (e.g., Gmail), head to https://news.google.com/ and click the gear icon at top-right. Below the icon should appear **Personalize Google News**, below which is **Advanced**. Click the latter, and **Add a local section** should appear at right. Input, say, **Cambridge, Massachusetts** into that box, then click **Add**. You should find yourself at the URL below?

https://news.google.com/news/section?pz=1&cf=all&geo=Cambridge,+Massachusetts&ned=us&redirect=true

Not to worry if you don't have a Google Account. Just head straight to that URL.

Interesting, it looks like our input is now the value of an HTTP parameter, `geo`, though there's a bunch of other parameters too. (Recall that `+` is one way a browser can encode a space in a URL. Another way is with `%20`.) One at a time, delete each of those other key-value pairs plus one ampersand (e.g., first `pz=1&`, then `cf=all&`, then `&ned=us`, then `&redirect=true`, hitting Enter after each deletion so as to reload the page via a shorter and shorter URL. You should find that Google still returns news for Cambridge even when the URL is just the below?

https://news.google.com/news/section?geo=Cambridge,+Massachusetts

Nice! +1 for trial and error. Now try changing the value of `geo` to, say, `02138` and then hit Enter again. You should find yourself at the URL below? The articles might change (since Cambridge has more than one postal code), but the news should still be about Cambridge?

https://news.google.com/news/section?geo=02138

Nice. Though the page you're looking at, of course, is written in HTML. And all we want, if the staff's solution is any indication, is a bulleted list of articles' titles and links. How to get

[3] https://developers.google.com/maps/documentation/javascript/markers
[4] https://developers.google.com/maps/documentation/javascript/infowindows

those without "scraping" this page's (surely complicated) HTML? Scroll down to the page's bottom and look for **RSS**. Click that link, and you should find yourself at the URL below?

https://news.google.com/news/feeds?pz=1&cf=all&ned=us&hl=en&geo=Cambridge,+Massachusetts&output=rss

As before, delete any parameters that don't feel core to the mission at hand, leaving only, say, `geo` and, now, `output`. You should find yourself at the (still fully functional) URL below.

https://news.google.com/news/feeds?geo=02138&output=rss

Deleting those parameters probably isn't necessary (and, who knows, their absence might break things eventually), but whittling a URL down to its essence does feel like better design, so let's stick with simple.

Now, what's all this markup that's now on your screen? It looks a bit like HTML, but you're actually looking at an "RSS feed," a flavor of XML (a tag-based markup language). For quite some time, RSS was all the rage insofar as it enabled websites to "syndicate" articles in a standard format that "RSS readers" could read. RSS isn't quite as hip anymore these days, but it's still a terrific find for us because it's "machine-readable". Because it adheres to a standard format, we can parse it (pretty easily!) with software. Here's what an RSS feed generally looks like (sans actual data):

```
<rss version="2.0">
    <channel>
        <title>...</title>
        <description>...</description>
        <link>...</link>
        <item>
            <guid>...</guid>
            <title>...</title>
            <link>...</link>
            <description>...</description>
            <category>...</category>
            <pubDate>...</pubDate>
        </item>
        ...
    </channel>
</rss>
```

In other words, an RSS feed contains a root element called `rss`, the child of which is an element called `channel`. Inside of `channel` are elements called `title`, `description`, and `link`, followed by one or more elements called `item`, each of which represents an article (or blog post or the like). Each `item`, meanwhile, contains elements called `guid`, `title`, `link`, `description`, `category`, and `pubDate`. Of course, between most of these start tags and end tags should be actual data (e.g., an article's actual title). For more details, see http://cyber.law.harvard.edu/rss/rss.html.

Ultimately, we'll parse RSS feeds from Google News using PHP and then return articles' titles and links to our web app via Ajax as JSON. But more on that in a bit.

jQuery

Recall that jQuery[5] is a super-popular JavaScript library that "makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers." To be fair, though, it's not without a learning curve. Read through a few sections of jQuery's documentation.

- $(document).ready()[6]
- Selecting Elements[7]
- jQuery's Ajax-Related Methods[8]

jQuery's documentation isn't the most user-friendly, though, so odds are you'll ultimately find Google[9] and Stack Overflow[10] handier resources.

Recall that `$` is usually (though not always) an alias for a global object that's otherwise called `jQuery`.

typeahead.js

Now take a look at a demo of Twitter's typeahead.js library, a jQuery "plugin" that adds support for autocompletion to HTML text fields. See **The Basics** specifically:

[5] http://jquery.com/
[6] http://learn.jquery.com/using-jquery-core/document-ready/
[7] http://learn.jquery.com/using-jquery-core/selecting-elements/
[8] http://learn.jquery.com/ajax/jquery-ajax-methods/
[9] https://www.google.com/
[10] http://stackoverflow.com/

http://twitter.github.io/typeahead.js/examples/

And now skim the documentation for that same library, which (surprise, surprise) isn't as user-friendly as would be ideal. But, again, not to worry.

https://github.com/twitter/typeahead.js/blob/master/doc/jquery_typeahead.md

Underscore

Finall, skim the documentation for Underscore[11], another popular JavaScript library that offers functions that many folks wish were built into JavaScript itself! In particular, take note of `template`. Admittedly, this documentation isn't very user-friendly either, so not to worry if usage is non-obvious for the moment.

- http://underscorejs.org/#template

Much like jQuery uses `$` as its symbol (because it looks cool), Underscore uses `_` is its symbol. For instance `_.template` means that Underscore has a method (i.e., function) called `template`.

Getting Started

Phew, that was a lot! But think of it this way: that's a lot of functionality you don't need to implement yourself! Indeed, implementing autocompletion alone could be a project unto itself. We just need to figure out how to wire (or, if you will, "mash") all of these components together in order to build our own amazing web app.

Anyhow! Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window and execute `update50` to ensure that your appliance is up-to-date!

Then, download this problem set's distribution code from http://cdn.cs50.net/2014/fall/psets/8/pset8/pset8.zip. Unzip it into `~/vhosts`, so that you ultimately have a `pset8` directory in `~/vhosts`. Then delete `pset8.zip`.

Next, execute `ls` within `~/vhosts/pset8`, and you should see three subdirectories: `bin`, `includes`, and `public`. Ensure that permissions are as follows:

- `700`

[11] http://underscorejs.org/

    # `bin`
    # `bin/import`
    # `includes`

- `711`
    # `public`
    # `public/css`
    # `public/fonts`
    # `public/img`
    # `public/js`
- `600`
    # `includes/*.php`
    # `public/*.php`
- `644`
    # `public/css/*`
    # `public/fonts/*`
    # `public/img/*`
    # `public/index.html`
    # `public/js/*`

(Remember how? Remember why?)

Even though your code for this problem set will again live in `~/vhosts`, let's ensure that it's nonetheless backed up via Dropbox, assuming you set up Dropbox inside of the appliance. In a terminal window, execute

```
ln -s ~/vhosts/pset8 ~/Dropbox
```

in order to create a "symbolic link" (i.e., alias or shortcut) to your `~/vhosts/pset8` directory within your `~/Dropbox` directory so that Dropbox knows to start backing it up.

Next, in a terminal window, execute

```
sudo gedit /etc/hosts
```

in order to run `gedit` as the appliance's "superuser" (aka "root") so that you can edit what's otherwise a read-only file. Carefully add this line at the bottom of that file, which will associate `pset8` with the appliance's "loopback" address (which won't ever change):

```
127.0.0.1 pset8
```

Then save the file and quit `gedit`.

Alright, time for a test! Open up Chrome inside of the appliance and visit http://pset8/.

You should find yourself at a map (without much of anything going on)! (If you instead see Forbidden, odds are you missed a step earlier; best to try all those chmod steps again.) Feel free to click on the map and drag it around. Or try searching for your home town via the text box up top. It won't find it yet! Indeed, the mashup itself doesn't do much just yet!

Head to http://pset8/phpmyadmin using Chrome inside of the appliance to access phpMyAdmin. Log in as John Harvard if prompted (with a username of **jharvard** and a password of **crimson**). You should then find yourself at phpMyAdmin's main page.

In a separate tab (again using Chrome inside of the appliance), visit http://cdn.cs50.net/2014/fall/psets/8/pset8/pset8.sql?download in order to download a file called `pset8.sql`. Once downloaded, open the file in `gedit`, as by clicking its name in Chrome's bottom-left corner or by selecting **File > Open…** in `gedit` and then navigating your way to **Downloads**. You should ultimately see a whole bunch of SQL (i.e., database queries) within `pset8.sql`. Highlight it all, then select **Edit > Copy** (or hit ctrl-c), then return to phpMyAdmin. Click phpMyAdmin's **SQL** tab, and paste everything you copied into that page's big text box (which is below **Run SQL query/queries on server "localhost"**). Skim what you just pasted to get a sense of the commands you're about to execute, then click **Go**. You should then see a greenish banner indicating success. In phpMyAdmin's top-left corner, you should now see link to a database called **pset8**, beneath which is a link to a table called **places**. (The latest version of phpMyAdmin is a bit buggy, though, so you might need to reload the page first.) If you click **places**, you'll find (gasp!) that this table is empty. But we have defined its "schema" (i.e., structure) for you. Click phpMyAdmin's **Structure** tab to see.

Let's now download the data that we'll ultimately import into this table. Using Chrome inside of the appliance, head to http://download.geonames.org/export/zip/, where you'll see a whole bunch of ZIP files, "data dumps" (in `.txt` format) from the GeoNames[12] geographical database, which "covers all countries and contains over eight million placenames that are available for download free of charge." Click `US.zip` to download a dump of postal codes (and more) for the United States. Alternatively, you're welcome to download another country's data, though this spec will assume the US for the sake of discussion. See http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2#Officially_assigned_code_elements if unsure how to interpret the ZIP files' 2-letter "base names." (They're "ISO 3166-1 alpha-2" country codes.)

Next, unzip `US.zip`, which should yield `US.txt`. And then delete `US.zip`.

Per http://download.geonames.org/export/zip/readme.txt, `US.txt` is quite like a CSV file except that fields are delimited with `\t` instead of a comma. To see the file's contents, you're welcome to open it in `gedit` (or use command-line utilities like `head`, `less`, and `more`), but take care not to change it.

Walkthrough

Shall we take a stroll?

http://www.youtube.com/watch?v=ASA8fAEerNo
And now a closer look at the distribution code.

import

Navigate your way to `~/vhosts/pset8/bin` and open up `import` with `gedit`. Not much there yet! Just a shebang and `TODO`. It's in this file that you'll ultimately write a PHP script that iterates over the lines in `US.txt`, `INSERT`ing data from each into `places`, that MySQL table. But more on that later.

index.php

Next navigate your way to `~/vhosts/pset8/public` and open up `index.html` with `gedit`. Ah, there we go. If you look at the page's `head`, you'll see all those CSS and

[12] http://www.geonames.org/

JavaScript libraries we'll be using (plus some others). Included in HTML comments are URLs for each library's documentation.

Next take a look at the page's `body`, inside of which is `div` with a unique `id` of `map-canvas`. It's into that `div` that we'll be injecting a map. Below that `div`, meanwhile, is a `form`, inside of which is an `input` of type `text` with a unique `id` of `q` that we'll use to take input from users.

styles.css

Now navigate your way to `~/vhosts/pset8/public/css` and open up `styles.css` with `gedit`. In there is a bunch of CSS that implements the mashup's default UI. Feel free to tinker (i.e., make changes, save the file, and reload the page in Chrome) to see how everything works, but best to undo any such changes for now before forging ahead.

scripts.js

Navigate next to `~/vhosts/pset8/public/js` and open up `scripts.js` with `gedit`. Ah, the most interesting file yet! It's this file that implements the mashup's "front-end" UI, relying on Google Maps and some "back-end" PHP scripts for data (that we'll soon explore). Let's walk through this one.

Atop the file are some global variables:

- `map`, which will contain a reference (i.e., a pointer of sorts) to the map we'll soon be instantiating;
- `markers`, an array that will contain references to any markers we add atop the map; and
- `info`, a reference to an "info window" in which we'll ultimately display links to articles.

Below those global variables is an anonymous function that will be called automatically by jQuery when the mashup's DOM is fully loaded (i.e., when `index.html` and all its assets, CSS and JavaScript especially, have been loaded into memory).

Atop this anonymous function is a definition of `styles`, an array of two objects that we'll use to configure our map, as per https://developers.google.com/maps/documentation/javascript/styling. Recall that `[` and `]` denote an array, while `{` and `}` denote an object.

The (very pretty) indentation you see is just a stylistic convention to which it's probably ideal to adhere in your code as well.

Below `styles` is `options`, another collection of keys and values that will ultimately be used to configure the map further, as per https://developers.google.com/maps/documentation/javascript/reference#MapOptions.

Next we define `canvas`, by using a bit of jQuery to get the DOM node whose unique `id` is `map-canvas`. Whereas `$("#map-canvas")` returns a jQuery object (that has a whole bunch of functionality built-in), `$("#map-canvas").get(0)` returns the actual, underlying DOM node that jQuery is just wrapping.

Perhaps the most powerful line yet is the next one in which we assign `map` (that global variable) a value. With

```
new google.maps.Map(canvas, options);
```

we're telling the browser to instantiate a new map, injecting it into the DOM node specified by `canvas`), configured per `options`.

The line below that one, meanwhile, tells the browser to call `configure` (another function we've written) as soon as the map is loaded.

addMarker

Uh oh, a `TODO`. Ultimately, given a `place` (i.e., postal code and more), this function will need to add a marker (i.e., icon) to the map.

configure

This function, meanwhile, picks up where that anonymous function left off. Recall that `configure` is called as soon as the map has been loaded. Within this function we configure a number of "listeners," specifying what should happen when we "hear" certain events. For instance,

```
google.maps.event.addListener(map, "dragend", function() {
    update();
});
```

indicates that we want to listen for a `dragend` event on the map, calling the anonymous function provided when we hear it. That anonymous function, meanwhile, simply calls `update` (another function we'll soon see). Per https://developers.google.com/maps/documentation/javascript/reference#Map, `dragend` is "fired" (i.e., broadcasted) "when the user stops dragging the map."

Similarly do we listen for `zoom_changed`, which is fired "when the map zoom property changes" (i.e., the user zooms in or out).

On the other hand, upon hearing `dragstart`, we ultimately call `removeMarkers` so that all markers disappear temporarily as a user drags the map, thereby avoiding the appearance of a flicker that might otherwise happen as markers are removed and then re-added after the maps bounds (i.e., corners) have changed.

Below those listeners is our configuration of that typeahead plugin. Take another look at https://github.com/twitter/typeahead.js/blob/master/doc/jquery_typeahead.md if unsure what `autoselect`, `highlight`, and `minLength` do here. Most importantly, though, know that the value of `source` (i.e., `search`) is the function that the plugin will call as soon as the user starts typing so that the function can respond with an array of search results based on the user's input. For instance, if the user types `foo` into that text box, the function should ultimately return an array of all places in your database that somehow match `foo`. How to perform those matches will ultimately be left to you! The value of `templates`, meanwhile, is an object with two keys: `empty`, whose value is the HTML that should be displayed when `search` comes back empty (i.e., returns an array of length 0), and `suggestion`, whose value is a "template" that will be used to format each entry in the plugin's dropdown menu. Right now, that template is simply `<p>TODO</p>`, which means that every entry in that dropdown will literally say `TODO`. Ultimately, you'll want to change that tvalue to something like

```
<p><%- place_name %>, <%- admin_name1 %></p>
```

so that the plugin dynamically inserts those values (`place_name` and `admin_name1`) or some others for you. In contrast to `<%=`, which Underscore also supports, the `<%-` ensures that the value will be escaped, a la PHP's `htmlspecialchars`, per http://underscorejs.org/#template.

Next notice these lines, which are admittedly a bit cryptic at first glance:

```
$("#q").on("typeahead:selected", function(eventObject, suggestion, name) {
    map.setCenter({lat: parseFloat(suggestion.latitude), lng:
 parseFloat(suggestion.longitude)});
    update();
});
```

These lines are saying that if the HTML element whose unique `id` is `q` fires an event called `typeahead:selected`, as will happen when the user selects an entry from the plugin's dropdown menu, we want jQuery to call an anonymous function whose second argument, `suggestion`, will be an object that represents the entry selected. Within that object must be at least two properties: `latitude` and `longitude`. We'll then call `setCenter` in order to re-center the map at those coordinates, after which we'll call `update` to update any markers.

Below those lines, meanwhile, are these:

```
$("#q").focus(function(eventData) {
    hideInfo();
});
```

If you consult http://api.jquery.com/focus/, hopefully those lines will make sense?

Below those are these:

```
document.addEventListener("contextmenu", function(event) {
    event.returnValue = true;
    event.stopPropagation && event.stopPropagation();
    event.cancelBubble && event.cancelBubble();
}, true);
```

Unfortunately, Google Maps disables ctrl- and right-clicks on maps, which interferes with using Chrome's (amazingly useful) **Inspect Element** feature, so these lines re-enable those.

Last up in `configure` is a call to `update` (which we'll soon look at) and a call to `focus`, this time with no arguments. See http://api.jquery.com/focus/ for why!

hideInfo

Thankfully, a short function! This one just calls `close` on our global info window.

removeMarkers

Hm, a `TODO`. Ultimately, this function will need to remove any and all markers from the map!

search

This function is called by the typeahead plugin every time the user changes the mashup's text box, as by typing or deleting a character. The value of the text box (i.e., whatever the user has typed in total) is passed to `search` as `query`. And the plugin also passes to `search` a second argument, `cb`, a "callback" which is a function that `search` should call as soon as it's done searching for matches. In other words, this passing in of `cb` empowers `search` to be "asynchronous," whereby it will only call `cb` as soon as it's ready, without blocking any of the mashup's other functionality. Accordingly, `search` uses jQuery's `getJSON` method to contact `search.php` asynchronously, passing in one parameter, `geo`, the value of which is `query`. Once `search.php` responds (however many milliseconds or seconds later), the anonymous function passed to `done` will be called and passed `data`, whose value will be whatever JSON that `search.php` has emitted. (Though if something goes wrong, `fail` is instead called.) Finally called is `cb`, to which `search` passes that same `data` so that the plugin can iterate over the places therein (assuming `search.php` found matches) in order to update the plugin's drop-down. Phew.

Notice that we're using `getJSON`'s "Promise" interface, per http://api.jquery.com/jquery.getjson/. Rather than pass an anonymous function directly to `getJSON` (to be called upon success), we're instead "chaining" together calls to `getJSON`, `done` (whose argument, an anonymous function, will be called upon success), and `fail` (whose argument, another anonymous function, will be called upoon failure). See http://api.jquery.com/jquery.ajax/ for some additional details. And see http://davidwalsh.name/write-javascript-promises for an explanation of promises themselves.

Notice, too, that we're using `console.log` much like you might use `printf` in C to log errors for debugging's sake. You may want to do so as well! Just realize that `console.log` will log messages to the browser's console (i.e., the **Console** tab of

Chrome's developer tools), not to your terminal window. See https://developer.mozilla.org/en-US/docs/Web/API/Console.log for tips.

showInfo

This function opens the info window at a particular marker with particular content (i.e., HTML). Though if only one argument is supplied (`marker`), `showInfo` simply displays a spinning icon (which is just an animated GIF). Notice, though, how this function is creating a string of HTML dynamically, thereafter passing it to `setContent`. Perhaps keep that technique in mind elsewhere!

update

Last up is `update`, which first determines the map's current bounds, the coordinates of its top-right (northeast) and bottom-left (southwest) corners. It then passes those coordinates to `update.php` via a GET request (underneath the hood of `getJSON`) a la:

```
GET /update.php?
ne=42.42783050736053%2C-71.00200380859377&sw=42.32612831530431%2C-71.24919619140627
 HTTP/1.1
```

The `%2C` are just commas that have been "URL-encoded." Realize that our use of commas is arbitary; we're expecting `update.php` to parse and extract latitudes and longitudes from these parameters. We could have simply passed in four distinct parameters, but we felt it was semantically cleaner to pass in just one parameter per corner.

As we'll soon see, `update.php` is designed to return a JSON array of places that fall within the map's current bounds (i.e., cities within view). After all, with those two corners alone can you define a rectangle, which is exactly what the map is!

As soon as `update.php` responds, the anonymous function passed to `done` is called and passed `data`, the value of which is the JSON emitted by `update.php`. (Though if something goes wrong, `fail` is instead called.) That anonymous function first removes all markers from the map and then iteratively adds new markers, one for each place (i.e., city) in the JSON.

Phew and phew!

update.php

Now navigate your way to `~/vhosts/pset8/public` and open up `update.php` with `gedit`. Ah, okay, here's the "back-end" script that outputs a JSON array of up to 10 places (i.e., cities) that fall within the specified bounds (i.e., within the rectangle defined by those corners). You won't need to make changes to this file, but do read through it line by line, Googling any function with which you're not familiar. Of particular interest should be `preg_match`, which allows you to compare strings against "regular expressions." While cryptic at first glance, our two calls to `preg_match` in `update.php` are simply ensuring that both `sw` and `ne` are comma-separated latitudes and longitudes.

Oh, and yes, this file's SQL queries assume that the world is flat for simplity.

search.php

Next open up `search.php` with `gedit`. Ah, not much in there now. Just an eventual `TODO`!

articles.php

Now open up `articles.php` with `gedit`. This one we've implemented for you. Read through its lines, again Googling as needed. Notice how it expects a GET parameter called `geo`, which it passes to Google News for localized news, thereafter returning a JSON array of objects, each of which has two keys: `link` and `title`.

You can actually see this file in action. Go ahead and visit URLs like

- http://pset8/articles.php?geo=Cambridge,+Massachusetts
- http://pset8/articles.php?geo=02138

using Chrome inside of the appliance. You should see a (pretty-printed) JSON array of articles!

config.php

Let's now take a quick peek at the file that all those other PHP files have required. Navigate your way to `~/vhosts/pset8/includes` and open up `config.php` with `gedit`. Ah, a file quite like Problem Set 7's own `config.php`, albeit simpler.

constants.php

Next open up `constants.php` with `gedit`. Ah, another familiar sight, albeit with a database called `pset8`.

functions.php

And, lastly, take a look at `functions.php` with `gedit`. In there that same function, `query`, from Problem Set 7, albeit slightly modified. For all intents and purposes, though, you can call it the same as before.

What To Do

Alright, it's time to mash Google's two APIs together.

import

Recall that `places`, that MySQL table you imported earlier, is currently empty. The data that needs to be in it, meanwhile, is in `US.txt`.

Write, in `import`, a command-line script in PHP that accepts as a command-line argument the path to a file (which can be assumed to be formatted like `US.txt`) that iterates over the file's lines, inserting each as new row in `places`. We leave the overall design of this script to you, but be sure to perform rigorous error-checking, leveraging `file_exists`[13], `is_readable`[14], and/or similar. Odds are you'll find `fopen`[15], `fgetcsv`[16], and `fclose`[17] of particular help, along with `query` from `functions.php`. Note that `fgetcsv` takes an optional third argument that allows you to override the default delimiter from a comma to something else.

To run this script, you'll want to execute a command like

```
./import /path/to/US.txt
```

[13] http://php.net/manual/en/function.file-exists.php
[14] http://php.net/manual/en/function.is-readable.php
[15] http://php.net/manual/en/function.fopen.php
[16] http://php.net/manual/en/function.fgetcsv.php
[17] http://php.net/manual/en/function.fclose.php

where `/path/to/US.txt` is indeed the (relative or absolute) path to that file.

Odds are the first several runs of your script won't be quite right, so you'll likely want to empty `places` between runs, as by executing

```
TRUNCATE places
```

in phpMyAdmin's **SQL** tab or by clicking **Empty the table (TRUNCATE)** in phpMyAdmin's **Operations** tab. If you take the latter approach, be sure that you've first selected **places** (as by clicking it in phpMyAdmin's lefthand column) so that you don't truncate some other table. And be sure not to click **Delete the table (DROP)**, else you'll have to re-import `pset8.sql` and re-create any changes you'd made.

Either now or later on, you should probably add one or more additional indexes to `places` in order to expedite searches (for `search.php`). See http://dev.mysql.com/doc/refman/5.5/en/mysql-indexes.html and http://dev.mysql.com/doc/refman/5.5/en/fulltext-search.html (and Google!) for tips. (We defined `places` in `pset8.sql` as using a MyISAM engine so that a `FULLTEXT` index is an option.)

Even though data can sometimes be imported in bulk via phpMyAdmin's **Import** tab, you must indeed (in case wondering!) implement `import` as prescribed!

search.php

Implement `search.php` in such a way that it outputs a JSON array of objects, each of which represents a row from `places` that somehow matches the value of `geo`. The value of `geo`, passed into `search.php` as a GET parameter, meanwhile, might be a city, state, and/or postal code. We leave it to you to decide what constitutes a match and, therefore, which rows to `SELECT`. Odds are you'll find SQL's `LIKE` and/or `MATCH` keywords helpful. Again see http://dev.mysql.com/doc/refman/5.5/en/string-comparison-functions.html and http://dev.mysql.com/doc/refman/5.5/en/fulltext-search.html (and Google!) for tips.

To test `search.php`, even before your text box is operational, simply visit URLs like

- http://pset8/search.php?geo=Cambridge,Massachusetts,US
- http://pset8/search.php?geo=Cambridge,+Massachusetts

- http://pset8/search.php?geo=Cambridge,+MA
- http://pset8/search.php?geo=Cambridge+MA
- http://pset8/search.php?geo=02138

and other such variants to see if you get back the JSON you expect. Again, though, we leave it to you to decide just how tolerate `search.php` will be of abbreviations, punctuation, and the like. The more flexible, though, the better! Try to implement features that you yourself would expect as a user!

configure

Now that `search.php` and your text box are (hopefully!) working, modify the value of `suggestion` in `configure`, the function in `scripts.js`, so that it displays matches (i.e., `place_name`, `admin_name1`, and/or other fields) instead of `TODO`. Recall that a value like

```
<p><%- place_name %>, <%- admin_name1 %></p>
```

might do the trick, perhaps coupled with some CSS.

addMarker

Implement `addMarker` in `scripts.js` in such a way that it adds a marker for `place` on the map, where `place` is a JavaScript object that represents a row from `places`, your MySQL table. See https://developers.google.com/maps/documentation/javascript/markers for tips. But also see http://google-maps-utility-library-v3.googlecode.com/svn/tags/markerwithlabel/1.1.9/ for an alternative to Google's own markers, which add support for labels beneath markers. (Recall that we're already loading `markerwithlabel_packed.js` for you in `index.html`.)

When a marker is clicked, it should trigger the mashup's info window to open, anchored at that same marker, the contents of which should be an unordered list of links to article for that article's location (unless `articles.php` outputs an empty array)!

Again, not to worry if some of your markers (or labels) overlap others, assuming such is the result of imperfections in `US.txt` and not your own code!

If you'd like to customize your markers' icon, see https://developers.google.com/maps/documentation/javascript/markers#simple_icons. For the URLs of icons built-into Google Maps, see http://www.lass.it/Web/viewer.aspx?id=4. For third-party icons, see http://mapicons.nicolasmollet.com/category/markers/.

removeMarkers

Implement `removeMarkers` in such a way that it removes all markers from the map. Odds are you'll need `addMarker` to modify that global variable called `markers` in order for `removeMarkers` to work its own magic!

personal touch

Last but not least, add at least one personal touch to your mashup, altering its aesthetics or adding some feature that (ideally!) no classmate has. Any touch that compels you to learn (or Google!) at least one new technique is of reasonable scope.

How to Submit

Step 1 of 2

When ready to submit, open up a Terminal window and "export" your MySQL database (i.e., save it into a text file) by executing the commands below, inputting **crimson** when prompted for a password. For security, you won't see the password as you type it.

```
cd ~/vhosts/pset8
mysqldump -u jharvard -p pset8 > pset8.sql
```

If you type `ls` thereafter, you should see that you have a new file called `pset8.sql` in `~/vhosts/pset8`. (If you realize later that you need to make a change to your database and re-export it, you can delete `pset8.sql` with `rm pset8.sql`, then re-export as before.) Next create a ZIP (i.e., compressed) file containing your entire `pset8` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset8`, including any subdirectories (or even subsubdirectories!).

```
cd ~/vhosts
```

```
zip -r pset8.zip pset8/
```

If you type `ls` thereafter, you should see that you have a new file called `pset8.zip` in `~/vhosts`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset8.zip`, then create a new ZIP file as before.) * Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.edx.org/submit[18], logging in if prompted. * Click **Submit** toward the window's top-left corner. * Under **Problem Set 8** on the screen that appears, click **Upload New Submission**. * On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear. * Navigate your way to `pset8.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset8.zip`, click it once to select it, then click **Open**. * Click **Start upload** to upload your ZIP file to CS50's servers. * On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.edx.org/submit[19] and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

Head to http://cs50.edx.org/2015/psets/8/ where a short form awaits. (It's a bit longer than usual, so it's okay if you start it before but submit it shortly after the problem set's deadline.) Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 8, your last!

[18] http://cs50.edx.org/submit
[19] http://cs50.edx.org/submit