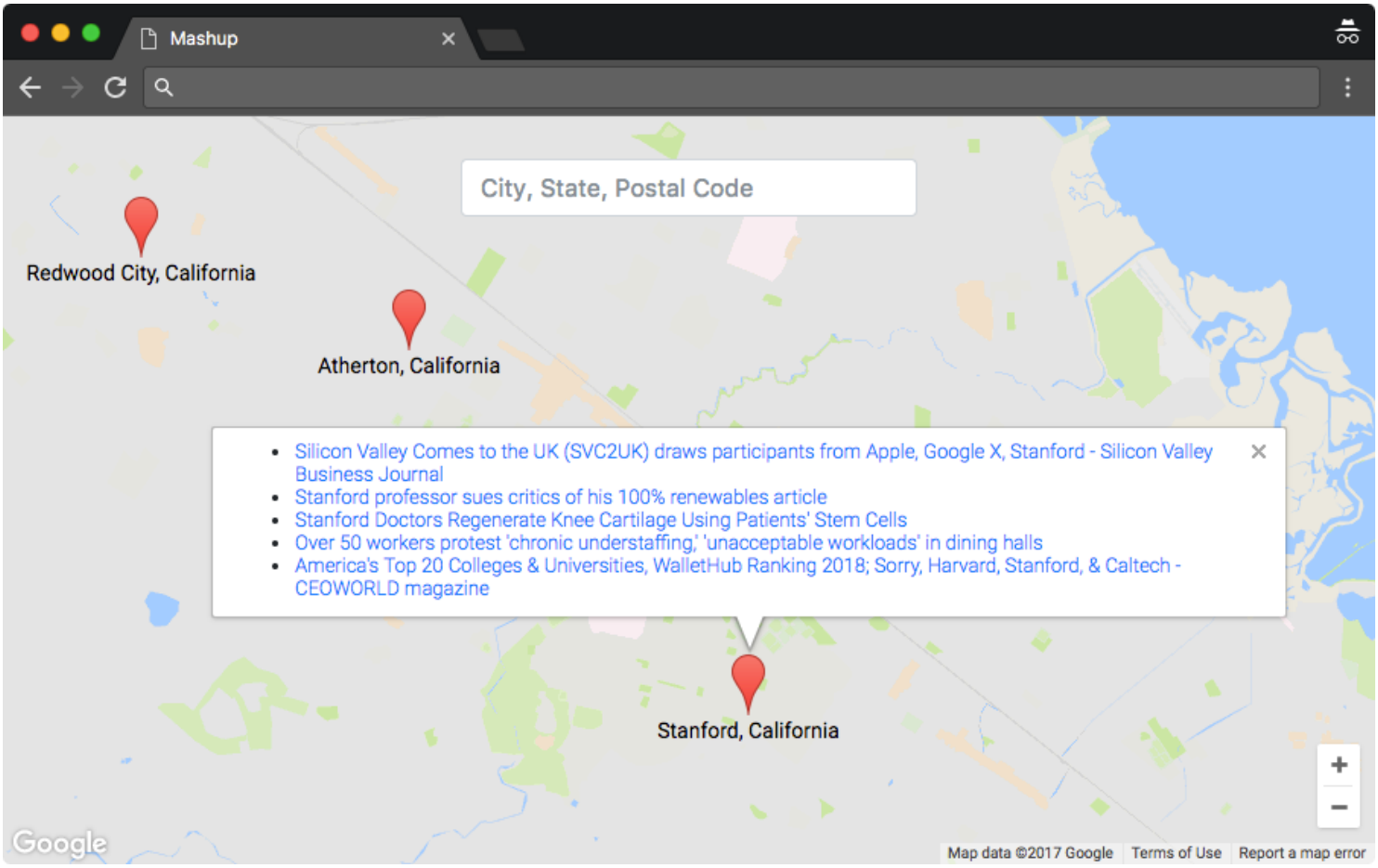
Table of Contents

- tl;dr
- Background
  - Google Maps
  - Google News
  - jQuery
  - typeahead.js
- Distribution
  - Downloading
  - Running
  - Understanding
- Specification
  - `mashup.db`
  - `application.py`
  - `scripts.js`
- Walkthroughs
- Testing
  - Correctness
  - Style
- Staff's Solution
- Hints
- FAQs
  - CREATE TABLE places(…) failed: duplicate column name
  - ImportError: No module named 'application'
  - OSError: [Errno 98] Address already in use
- CHANGELOG

Mashup

tl;dr

Implement a website that lets users search for articles atop a map, a la the below.



Background

A "mashup" is a web app that combines data or functionality from multiple sources. In this here mashup, you'll combine data from Google News with functionality from Google Maps!

Google Maps

Odds are you're already familiar, but head to Google Maps anyway at https://www.google.com/maps (https://www.google.com/maps). Input **42.374490, -71.117185** into the search box up top, and you should find yourself at Harvard. Input **41.3163284, -72.9245318**, and you should find yourself at Yale.

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 (https://developers.google.com/maps/documentation/javascript/) 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 (https://developers.google.com/maps/documentation/javascript/tutorial)
- Drawing on the Map
  - Markers (https://developers.google.com/maps/documentation/javascript/markers)
  - Info Windows (https://developers.google.com/maps/documentation/javascript/infowindows)

Google News

Okay, now we need us some news. Fortunately, Google News offers just that! In fact, if you visit the URLs below, you should see stories pertaining to Cambridge, Massachusetts and New Haven, Connecticut, respectively:

- https://news.google.com/news/local/section/geo/Cambridge,%20Massachusetts (https://news.google.com/news/local/section/geo/Cambridge,%20Massachusetts)
- https://news.google.com/news/local/section/geo/New%20Haven,%20Connecticut (https://news.google.com/news/local/section/geo/New%20Haven,%20Connecticut)

Notice how the spaces in those towns' names have been "URL-encoded" (i.e., escaped) as `%20`. Notice, too, that you can query for news by postal code:

- https://news.google.com/news/local/section/geo/02138 (https://news.google.com/news/local/section/geo/02138)
- https://news.google.com/news/local/section/geo/06511 (https://news.google.com/news/local/section/geo/06511)

The pages you're seeing are, of course, written in HTML. But all we want, if the staff's solution is any indication, is a bulleted list of articles' titles and links. How to get 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 URLs like the below, perhaps with some HTTP parameters after a question mark?

- https://news.google.com/news/rss/local/section/geo/02138 (https://news.google.com/news/rss/local/section/geo/02138)
- https://news.google.com/news/rss/local/section/geo/06511 (https://news.google.com/news/rss/local/section/geo/06511)

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 https://cyber.law.harvard.edu/rss/rss.html (https://cyber.law.harvard.edu/rss/rss.html).

Ultimately, we'll parse RSS feeds from Google News using Python 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 (http://jquery.com/) is a 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 if you'd like.

- $(document).ready() (http://learn.jquery.com/using-jquery-core/document-ready/)
- Selecting Elements (http://learn.jquery.com/using-jquery-core/selecting-elements/)
- jQuery's Ajax-Related Methods (http://learn.jquery.com/ajax/jquery-ajax-methods/)

jQuery's documentation isn't the most user-friendly, though, so odds are you'll ultimately find Google (https://www.google.com/) and Stack Overflow (http://stackoverflow.com/) 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 some examples of Twitter's typeahead.js library, a jQuery "plugin" that adds support for autocompletion to HTML text fields. Play with **The Basics**, **Custom Templates**, and **Scrollable Dropdown Menu** in particular.

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

And now skim the documentation for a "fork" (i.e., someone else's version) of that same library:

https://github.com/corejavascript/typeahead.js/blob/master/doc/jquery_typeahead.md (https://github.com/corejavascript/typeahead.js/blob/master/doc/jquery_typeahead.md)

Note that Twitter hasn't updated their own version of the library for quite some time, so take care to rely on github.com/corejavascript/typeahead.js (https://github.com/corejavascript/typeahead.js), not github.com/twitter/typeahead.js (https://github.com/twitter/typeahead.js).

Distribution

Downloading

```
$ wget http://cdn.cs50.net/2018/x/psets/8/mashup/mashup.zip (http://cdn.cs50.net/20
$ unzip mashup.zip
$ rm mashup.zip
$ cd mashup
$ ls
application.py  mashup.db         static/
helpers.py      requirements.txt  templates/
$ wget http://cdn.cs50.net/2017/fall/psets/8/mashup/US.zip (http://cdn.cs50.net/201
$ unzip US.zip
$ rm US.zip
$ ls
application.py  mashup.db   requirements.txt  templates/
helpers.py      readme.txt  static/           US.txt
```

Running

1. Start Flask's built-in web server (within `mashup/`):

```
flask run
```

Visit the URL outputted by `flask` to see the distribution code in action. You won't be able to search for news, though, just yet!

2. Via CS50's file browser, double-click **mashup.db** in order to open it with phpLiteAdmin. Notice that it doesn't have any tables yet! (That's where you come in.) Here on out, though, if you'd prefer a command line, you're welcome to use `sqlite3` instead of phpLiteAdmin.

Understanding

index.html

Open up `templates/index.html`, which will be your app's one and only HTML page. If you look at the page's `head`, you'll see all those CSS and JavaScript libraries we'll be using (plus some others). Included in HTML comments are URLs for each library's documentation if curious.

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

Next open up `static/styles.css`. 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

Next open up `static/scripts.js`. 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" Flask routes 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 (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/3.exp/reference#MapOptions (https://developers.google.com/maps/documentation/javascript/3.exp/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

Ah, 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/3.exp/reference#Map (https://developers.google.com/maps/documentation/javascript/3.exp/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).

Below those listeners is our configuration of that typeahead plugin. Take another look at https://github.com/corejavascript/typeahead.js/blob/master/doc/jquery_typeahead.md (https://github.com/corejavascript/typeahead.js/blob/master/doc/jquery_typeahead.md) if unsure what `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 one key, `suggestion`, whose value is a "template" that will be used to format each entry in the plugin's dropdown menu. That template is created by a call to `Handlebars.compile`, a method that comes with Handlebars (http://handlebarsjs.com/), a templating language for JavaScript similar in spirit to Jinja for Python. Right now, that template is simply `<div>TODO</div>`, which means that every entry in that dropdown will literally say `TODO`. Ultimately, you'll want to change that value to something like

```
<div>{{place_name}}, {{admin_name1}}, {{postal_code}}</div>
```

so that the plugin dynamically inserts those values (`place_name`, `admin_name1`, and `postal_code`) or some others for you.

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(suggestior
    ...
    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) {
    info.close();
});
```

If you consult http://api.jquery.com/focus/ (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/ (http://api.jquery.com/focus/) for why!

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` two additional arguments, the last of which (`asyncResults`) is a "callback" function that `search` should call as soon as it's done searching for matches. In other words, this passing in of `asyncResults` empowers `search` to be "asynchronous," whereby it will only call `asyncResults` 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` asynchronously,

passing in one parameter, `q`, the value of which is `query`. Once `/search` responds (however many milliseconds or seconds later), `asyncResults` will be called and passed `data`, whose value will be whatever JSON that `/search` has emitted. The plugin can then iterate over the places therein (assuming `/search` found matches) in order to update the plugin's drop-down. Phew.

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` via a GET request (underneath the hood of `getJSON`) a la:

```
GET /update?ne=37.45215513235332%2C-122.03830380859375&q=&sw=37.39503397352173%2C-1
```

The `%2C` are just commas that have been "URL-encoded." Realize that our use of commas is arbitary; we're expecting `/update` 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` 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` responds, the anonymous function passed to `done` is called and passed `data`, the value of which is the JSON emitted by `/update`. (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!

application.py

Now open up `application.py`, which contains four routes!

`index`

All this route does is render `index.html`, the app's sole template.

`articles`

Not much in here yet, just a `TODO`!

`search`

Not much in this route yet either, just another `TODO`!

`update`

Ah, okay, here's the "back end" 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 route, but do read through it line by line, Googling any function with which you're not familiar.

And yes, this file's SQL queries assume that the world is flat for simplicity.

`helpers.py`

Finally, take a look at `helpers.py`. In this file we've defined just one function, `lookup`, which queries Google News for articles for a particular geography, falling back on The Onion if none are available.

Specification

`mashup.db`

Per `readme.txt`, `US.txt` is quite like a CSV file except that its fields are delimited with `\t` (a tab character) instead of a comma. Conveniently, SQLite allows you to import CSV files (https://www.sqlite.org/cli.html#csv_import) and, as it turns out, TSV (tab-separated values) files as well. But you first need a table into which to import such a file.

Using phpLiteAdmin or `sqlite3`, create a table in `mashup.db` called `places` that has these twelve fields, in this order:

1. `country_code`
2. `postal_code`
3. `place_name`
4. `admin_name1`
5. `admin_code1`
6. `admin_name2`
7. `admin_code2`

8. `admin_name3`
9. `admin_code3`
10. `latitude`
11. `longitude`
12. `accuracy`

See `readme.txt` (or `US.txt` itself) for clues as to appropriate types for these fields. Don't include an `id` field (else you can't do what we're about to do!).

Rather than `INSERT` the rows from `US.txt` into your newly created table, let's now import them in bulk as follows:

```
$ sqlite3 mashup.db
.separator "\t"
.import US.txt places
```

If you see any errors, odds are your schema for `places` isn't quite right, in which case you'll want to `ALTER` (or `DROP` and re-`CREATE`) it accordingly. To confirm that an import's successful, execute

```
wc -l US.txt
```

to count how many rows are in `US.txt`. (That command-line argument is a hyphen followed by a lowercase L.) Then execute a query like

```
SELECT COUNT(*) FROM places;
```

in `sqlite3` or phpLiteAdmin. The counts should match!

`application.py`

`articles`

Complete the implementation of `/articles` in such a way that it outputs a JSON array of objects, each of which represents an article for `geo`, whereby `geo` is passed into `/articles` as a GET parameter, as in the staff solution, below.

- http://mashup.cs50.net/articles?geo=02138 (http://mashup.cs50.net/articles?geo=02138)
- http://mashup.cs50.net/articles?geo=06511 (http://mashup.cs50.net/articles?geo=06511)
- http://mashup.cs50.net/articles?geo=90210 (http://mashup.cs50.net/articles?geo=90210)

Odds are you'll want to call `lookup`! To test `/articles`, even before your text box is operational, simply visit URLs like

- `https://ide50-username.cs50.io/articles?geo=02138`
- `https://ide50-username.cs50.io/articles?geo=06511`
- `https://ide50-username.cs50.io/articles?geo=90210`

and other such variants, where `username` is your own username, to see if you get back the JSON you expect.

`search`

Complete the implementation of `/search` 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 `q`, as in the staff solution below.

- http://mashup.cs50.net/search?q=02138 (http://mashup.cs50.net/search?q=02138)
- http://mashup.cs50.net/search?q=Cambridge (http://mashup.cs50.net/search?q=Cambridge)
- http://mashup.cs50.net/search?q=06511 (http://mashup.cs50.net/search?q=06511)
- http://mashup.cs50.net/search?q=New%20Haven (http://mashup.cs50.net/search?q=New%20Haven)

The value of `q`, passed into `/search` as a GET parameter, 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`. It suffices to support searching by postal codes only, but try to support searching by city and/or state as well. Odds are you'll find SQL's `LIKE` keyword helpful. If feeling adventurous, you might like (but are not required) to experiment with SQLite's support for full-text searches (https://www.sqlite.org/fts3.html).

For instance, consider the query below.

```
db.execute("SELECT * FROM places WHERE postal_code = :q", q=request.args.get("q"))
```

Unfortunately, that query requires that a user's input be exactly equal to a postal code (per the `=`), which isn't all that compelling for autocomplete. How about this one instead? (Recall that `+` is Python's concatenation operator.)

```
q = request.args.get("q") + "%"
db.execute("SELECT * FROM places WHERE postal_code LIKE :q", q=q)
```

Notice how this example appends `%` to the user's input, which happens to be SQL's "wildcard" character that means "match any number of characters." The effect is that this query will return rows whose postal codes match whatever the user typed followed by any number of other characters. In other words, any of `0`, `02`, `021`, `0213`, and `02138` might return rows, as might any of `0`, `06`, `065`, `0651`, and `06511`.

If you'd like to support searching by more than just postal codes, keep in mind that SQL supports `OR` and `AND`!

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

- `https://ide50-username.cs50.io/search?q=02138`
- `https://ide50-username.cs50.io/search?q=Cambridge+MA`
- `https://ide50-username.cs50.io/search?q=Cambridge,+MA`
- `https://ide50-username.cs50.io/search?q=Cambridge,+Massachusetts`
- `https://ide50-username.cs50.io/search?q=Cambridge,+Massachusetts,+US`

or

- `https://ide50-username.cs50.io/search?q=06511`
- `https://ide50-username.cs50.io/search?q=New%20Haven+CT`
- `https://ide50-username.cs50.io/search?q=New%20Haven,+CT`
- `https://ide50-username.cs50.io/search?q=New%20Haven,+Connecticut`
- `https://ide50-username.cs50.io/search?q=New+Haven,+Connecticut,+US`

and other such variants, where `username` is your own username, to see if you get back the JSON you expect. Again, though, we leave it to you to decide just how supportive `/search` will be of such variants. The more flexible, though, the better! Try to implement features that you yourself would expect as a user!

Feel free to tinker with the staff's solution at http://mashup.cs50.net/ (http://mashup.cs50.net/), inspecting its HTTP requests via Chrome's Network tab as needed, if unsure how your own code should work!

`scripts.js`

First, toward the top of `scripts.js`, you'll see an anonymous function, inside of which is a definition of `options`, an object, one of whose keys is `center`, the value of which is an object with two keys of its own, `lat`, and `lng`. Per the comment alongside that object, your mashup's map is currently centered on Stanford, California. (D'oh.) Change the coordinates of your map's center to Cambridge (42.3770,

-71.1256) or New Haven (41.3184, -72.9318) or anywhere else! (Though be sure to choose coordinates in the US if you downloaded `US.txt`!) Once you save your changes and reload your map, you should find yourself there! Zoom out as needed to confirm visually.

As before, feel free to tinker with the staff's solution at http://mashup.cs50.net/ (http://mashup.cs50.net/), inspecting its HTTP requests via Chrome's Network tab as needed, if unsure how your own code should work!

`configure`

Now that `/search` 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

```
<div>{{place_name}}, {{admin_name1}}, {{postal_code}}</div>
```

might do the trick.

`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`. See https://developers.google.com/maps/documentation/javascript/markers (https://developers.google.com/maps/documentation/javascript/markers) for tips. Note that the latest (experimental) version of Google's API allows markers to have labels (https://developers.google.com/maps/documentation/javascript/3.exp/reference#MarkerOptions).

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` outputs an empty array)!

Not to worry if some of your markers (or labels) overlap others, assuming such is the result of imperfections in Google's API or `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 (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 (http://www.lass.it/Web/viewer.aspx?id=4). For third-party icons, see https://mapicons.mapsmarker.com/ (https://mapicons.mapsmarker.com/).

removeMarkers

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

Walkthroughs

Testing

Correctness

No `check50` anymore! Be sure to try to "break" your own site, as by

- searching for cities that don't exist,
- clicking markers for cities that don't have any articles,
- dragging and zooming in and out to update your map's markers, and
- searching with potentially dangerous characters like `'` and `;`.

Style

```
style50 application.py
```

Staff's Solution

You're welcome to stylize your own app differently, but here's what the staff's solution looks like!

http://mashup.cs50.net/ (http://mashup.cs50.net/)

It is **reasonable** to look at the staff's HTML and CSS. It is **not reasonable** to look at the staff's JavaScript.

Hints

- You're welcome center your map on some country other than the United States, downloading some other ZIP file (http://download.geonames.org/export/zip/) instead of `US.zip`. See Wikipedia (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2#Officially_assigned_code_elements) if unfamiliar with ISO 3166-1 alpha-2 codes.
- Know that you can use `console.log` much like you might use `eprintf` 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 (https://developer.mozilla.org/en-US/docs/Web/API/Console.log) for tips.

FAQs

CREATE TABLE places(...) failed: duplicate column name

If you see this message upon running `.import` in `sqlite3`, odds are you haven't run `sqlite3` in the same directory as `mashup.db`. If so, exit `sqlite3` with `.exit`, `cd` to your `mashup` directory, and then re-run `sqlite3 mashup.db`.

ImportError: No module named 'application'

By default, `flask` looks for a file called `application.py` in your current working directory (because we've configured the value of `FLASK_APP`, an environment variable, to be `application.py`). If seeing this error, odds are you've run `flask` in the wrong directory!

OSError: [Errno 98] Address already in use

If, upon running `flask`, you see this error, odds are you (still) have `flask` running in another tab. Be sure to kill that other process, as with ctrl-c, before starting `flask` again. If you haven't any such other tab, execute `fuser -k 8080/tcp` to kill any processes that are (still) listening on TCP port 8080.

CHANGELOG

- 2018-06-03
  - Updated the distribution code, updated the Search function, removed the need for student to have own API key and added another Hint about console.log