Assignment 2: Callback Yelp

Due Friday, May 15th, at 11:59 PM

Overview:

If you've ever debated with friends about the best and cheapest restaurants in Palo Alto, you're probably familiar with the popular review application, Yelp. In this assignment you'll implement a basic version of Yelp.

Your primary task is to implement create/read/update/delete (CRUD) operations for reviews. You'll also be interfacing with the Google Maps API to enhance your application's UX. We expect you to apply many of the concepts you've learned in class, including JavaScript fundamentals, HTTP methods/requests, AJAX, and APIs. We also expect you to use libraries such as Handlebars and jQuery to simplify your development.

Your previous Piazza assignment had no server component. Since we covered the basics of client-server interaction and AJAX requests in lecture, however, we now require that you communicate with our remote server. You'll be retrieving location entries that were created by both you and your classmates.

Important utilities:

Handlebars - A JavaScript library that allows you to render complex HTML templates and insert them into the DOM. It's a remedy for ugly `element.innerHTML = "[large HTML string]";` expressions.

Reference handout:
<https://docs.google.com/document/d/1W7DrKMonguxaPk03i81urvuaS2tq22N45HDchFRjeHE/edit?usp=sharing>
Further documentation: http://handlebarsjs.com/

jQuery - A JavaScript library that simplifies common JavaScript tasks, such as DOM selection/manipulation and animation.

Lectures on jQuery:
<https://docs.google.com/presentation/d/1wULDV4U4qRnI_N6KqAjcSP7972UDQrSLgfid06wSj04/edit?usp=sharing>
Further documentation: http://jquery.com/

Note that lectures covered the fundamentals of jQuery and AJAX but omitted full API descriptions that may prove necessary for this assignment. Freely consult the linked resources above when confused.

Overview of Starter Files:

We have provided the following starter files to help you get started.

HTML:

index.html:
This file contains the HTML layout of the Yelp application and Handlebars templates that you'll find useful. You may modify this file.

CSS:

css/style.css:
This file contains the CSS styling for the Yelp application. You may modify this file.

Local server:

server.py:
This file contains the logic to start a local development server. Do not modify it.

JavaScript libraries:

js/handlebars.js:
This file contains the Handlebars library, which allows you to create templates and render them. Refer to the Handlebars handout (see "Important utilities" above) for more details. Do not modify this file.

js/jquery.js:
This file contains the jQuery library, which includes helpful DOM traversal functions that'll prove useful for this assignment. Do not modify it.

Custom JavaScript that you're given:

js/main.js:
This file is the entry point to the application. It calls `MainView.render()`.

js/main-view.js:
This file exports the `MainView.render()` function, which delegates to `EntryView.render()`.

Custom JavaScript that you'll be implementing:

js/entry-model.js:
This file exports the `EntryModel` object, which contains functions that create, update, and delete location entries on the server using AJAX.

js/entry-view.js:
This file exports the `EntryView.render()` function, which displays a location entry to the user. It handles click events on buttons and delegates to other views.

js/creating-entry-view.js:
This file exports the `CreatingEntryView.render()` function, which displays an editable form to the user. Once this form is submitted, it creates a location entry with the user-provided information. It then delegates back to `EntryView`.

js/editing-entry-view.js:
This file exports the `EditingEntryView.render()` function, which displays an editable form to the user. Once this form is submitted, it updates a location entry with the user-provided information. It then delegates back to `EntryView`.

js/util.js:
This file exports the `Util` object, which contains common functions that are used across multiple views. If you have replicated logic across views, decompose it into a function in this file. Then, simply call your `Util` function wherever you need the logic.

js/google-map-view.js:
This file exports the `GoogleMapView.render()` function, which submits a Google Maps geocoding request and displays an interactive map for the entry.

Running the local server:

Certain browsers don't allow you to make AJAX requests unless your files are running on a local server. We've provided such a server for you. To run it, simply execute `python server.py` in your terminal. Then, visit the localhost URL that's printed out, and develop as you normally would. Depending on the browser you're using, this may be unnecessary.

Terminology:

Entry: In Callback Yelp, an entry represents a physical place, like a restaurant, grocery store, or bowling alley. Each entry has three fields: the place's name, a short description of it, and an address where the place is located.

python /Users/zhengyuanma/Box\ Sync/CS42/Assignment\ 2\ Yelp/server.py

Abstraction:

Callback Yelp's JavaScript is divided into three types of files: models, views, and utilities.

View: A view handles the logic of one `HTMLElement` and all its children on the page. This constitutes rendering Handlebars templates, adding event listeners, and processing user input with respect to that element. Views can also delegate responsibilities for their child elements to other views, which are called sub-views. Each view effectively has jurisdiction over its element and that element's children.

For instance, the `EntryView` handles logic around the `#entry` element. It renders the `#entry-template` Handlebars template into this element, and it handles click events on the

resulting `.new`, `.edit`, and `.delete` buttons. Furthermore, `EntryView` has a `.map` element, which contains a Google Map centered around the target location. Rather than handling the Google APIs directly, and putting all the logic in one file, `EntryView` creates a sub-view, `GoogleMapView`, that takes care of rendering the map.

Each view contains a `render()` function. The first parameter of the `render()` function is the element which the view has jurisdiction under. In the case of `EntryView`, this element would be the `#entry` div. All further parameters are view-specific, and depend on the specific functionality of that view. The main rule here is that a view must never handle any logic that lies outside of its assigned element.

A view here can be considered a government, and its element is the country being governed. We don't want the government to over-extend its bounds outside its country, and hence a view cannot handle the logic of other elements.

Views will often interface with models when handling events. When the `.delete` button is clicked, the `EntryView` will communicate with the `EntryModel` to delete an entry resource. This brings us to our next topic: models.

Model: A model encapsulates the data for a single resource type. It has methods that create/read/update/delete resources of this type from the server. A "resource", in this context, means some collection of data.

For this assignment, our resource is a location entry, which represents a place in our Yelp application. We've called the associated model `EntryModel`. The `EntryModel` should provide create/read/update/delete operations on entries. Each one of these operations will involve submitting AJAX requests to read/modify data that's on a server. In fact, most all JavaScript models revolve around interfacing with a server in this way via AJAX.

Utilities: Each one of our views are separated into distinct files that are largely opaque to one another. This allows for good separation of logic; each view should be largely independent of other views. Forcing you to create an application in this way makes it really easy to understand what each view is doing, as it doesn't really interact with other views.

But there are cases where views share similar functionality, and you'll see that manifest in Callback Yelp. In these cases, we don't want to rewrite logic in every view. Instead, we create a utility object. This object contains a set of functions that are common to multiple views. Since the utility object is accessible by all views, it effectively decomposes common code.

Requirements:

We'll break down the requirements on a file-by-file basis. We highly recommend implementing the files in the same order that we explain the requirements in.

Entry Model (js/entry-model.js):

You'll need to implement four methods in this model: `add`, `loadAll`, `update`, and `remove`, corresponding to create, read, update, and delete operations on entries, respectively.

For example, the `EntryModel` contains a method `add`, which takes the entry data and callback as parameters. The method should submit an AJAX request directly to the server with the entry information and execute the callback when the server responds to the request (i.e. when the server has successfully added an entry to the database).

The entry data should have the following object form:

```
{
      address: "450 Serra Mall  Stanford, CA 94305",
      name: "Stanford University",
      description: "The best university of all time."
}
```

You will directly communicate with our external server to retrieve and modify resources.

Making requests:

For each method, your code should do the following:

1) Create an `XMLHttpRequest` object
2) Handle the `load` event:
    a) If the status code (`request.status`) is **not** 200 (i.e. an error occurred), call the given callback with the response text (`request.responseText`) as the first parameter. The response text will indicate what error occurred.

    b) If the status code is 200, call the given callback with the first parameter `null`. This indicates that no error occurred. You may also need to pass a second parameter, depending on what method you're implementing.

3) Open up a URL with the correct request type
4) Set the Content-type header if you're making a POST request
5) Send the request. Pass in the appropriate parameters if you're making a POST request.

Passing data in a POST request:

In the below API specification, we mention what parameters you'll need to send to the server when making POST requests. These parameters should be stored in a JavaScript object. To actually send them to the server, you'll need to do two things. First, ensure you set the Content-type header to 'application/json', like so (assuming your XHR object is called

request):

```
request.setRequestHeader('Content-type', 'application/json');
```

Then, assuming your object with parameters is called `params`, pass a JSON string as the request body:

```
request.send(JSON.stringify(params));
```

Setting the header tells the server that our parameters will be passed in the JSON string format. Then, when calling `send()`, we're actually passing our parameters as a JSON string.

In class, we used a different Content-type header and passed parameters in query-string format. Nevertheless, we'd like you to use JSON for this assignment, as it makes it much easier to pass objects around to a server.

Note: Do NOT use jQuery to make AJAX requests. We want to see you using the raw `XMLHttpRequest` object.

API specification:

We'll now cover the API specification for each method.

Load All: To load all entries, submit a `GET` request to the `http://callbackjs.me:4155/entries` endpoint. The server will respond with JSON in the following form:

```
[
      {
            id: 12,
            address: "450 Serra Mall  Stanford, CA 94305",
            name: "Stanford University",
            description: "The best university of all time."
      },
      {
            id: 13,
            address: "101 Forest Ave Palo Alto, CA 94301",
            name: "Philz Coffee",
            description: "The best coffee of all time."
      }
]
```

Parse this JSON and pass the resulting array as the second parameter to the callback.

Add: To add a single entry on the server, submit a `POST` request to the `http://callbackjs.me:4155/entries` endpoint with entry data.

Request body (what you should pass to `request.send()`):

```
{
      address: "101 Forest Ave Palo Alto, CA 94301",
      name: "Philz Coffee",
      description: "The best coffee of all time."
}
```

Notice how you **don't** pass an ID in the request.

Sample response:

```
{
      id: 13,
      address: "101 Forest Ave Palo Alto, CA 94301",
      name: "Philz Coffee",
      description: "The best coffee of all time."
}
```

The server responds to the request with the newly created entry in the JSON format. The entry has a server-generated ID, along with the original fields passed in. Call the given callback with the parsed JSON as the second parameter.

Update: To update a single entry on the server, submit a `POST` request to the `http://callbackjs.me:4155/entries/:id` endpoint. Replace `:id` with the actual ID of the entry you'd like to update.

Request body (what you should pass to `request.send()`):

```
{
      address: "101 Forest Ave Palo Alto, CA 94301",
      name: "Philz Coffee",
      description: "The best coffee of all time."
}
```

Including the ID in the request body is fine, but it'll be ignored (our server uses the ID in the URL).

Sample response:

```
{
      id: 13,
      address: "101 Forest Ave Palo Alto, CA 94301",
      name: "Philz Coffee",
      description: "The best coffee of all time."
}
```

The server responds to the request with the updated JSON entry. You don't need to pass the updated entry to the callback, as it's won't be used.

Delete: To delete a single entry on the server, submit a POST request to the `http://callbackjs.me:4155/entries/:id/delete` endpoint. Replace `:id` with the actual ID of the entry you'd like to delete.

Request body (what you should pass to `request.send()`): nothing

Sample response:

```
Succeeded in deleting the entry with ID: 4
```

There's no need to parse the response text; you don't need to pass a second parameter to the callback.

Server validation:

Note that our server handles all input validation for you. If you make a POST request to `/entries/5` (to update entry 5) when an entry with `id` 5 does not exist, the server will return a `422`, "Unprocessable entity", status code and the following error message:

```
Could not find the entry with ID: 5
```

Additionally, if any properties (`name`, `description`, and `address`) aren't defined, or don't have between 3 and 100 characters, the server will return a `422`, "Unprocessable entity", status code.

For example, when the user executes a create `POST` request to the server with an address that is less than three characters, the server responds with the following error message:

```
The address is too short
```

This implies that you don't need to do any client-side validation. Simply pass the server's error message to the callback, and let the callee handle the rest (later on, you'll display the error message in a `.error` div).

Testing:

Try making some HTTP requests directly to our external server using CURL to understand the server's behavior. For example, execute the following CURL request:

```
curl --data "description=TestDescription&name=TestName&address=TestAddress"
"http://callbackjs.me:4155/entries"
```

What does the server return? What happens if we pass a request parameter that is too short/long?

In order to assess whether your `EntryModel` implementation works, you can go into the JavaScript console and simply try calling your functions. For instance, you could run the following:

```
EntryModel.loadAll(function(error, entries) {
      if (error) {
            console.log('Error:', error);
      } else {
            console.log(entries);
      }
});
```

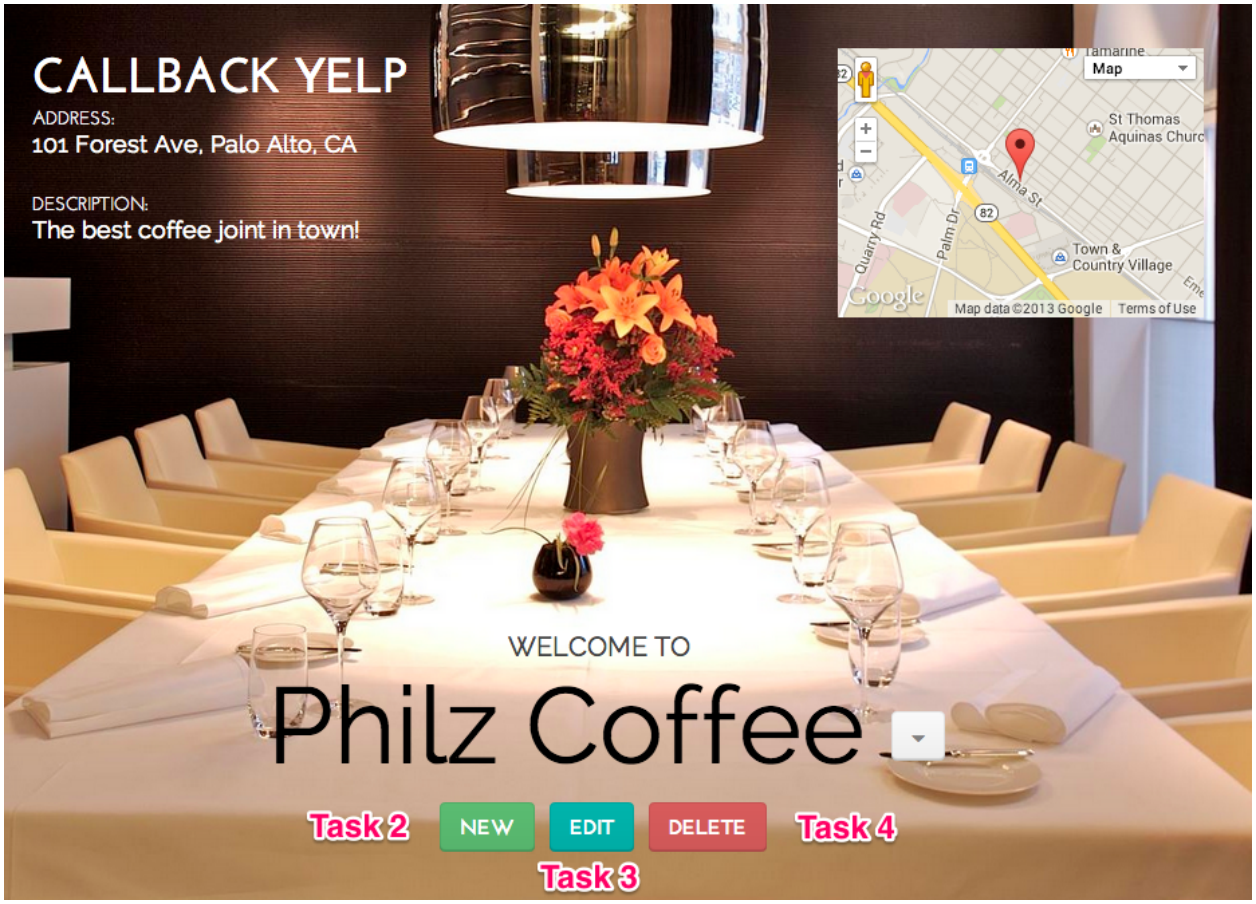
`EntryView` (js/entry-view.js):

The `EntryView`'s main job is to let the user view entries. A user should be able to see the active entry's name, description, and address. The user should also be allowed to switch between entries.

The `EntryView` has jurisdiction under the `#entry` element. Its `render()` function is called when the page loads by the `MainView`, which we have provided for you. `MainView` passes in a jQuery object representing the `#entry` element to this function. Your job is to implement this `render()` function. The required functionality is as follows:

Task 1: Render the `#entry-template` Handlebars template into the `#entry` element. Pass an object to the Handlebars render function in the following form:

```
{
      viewing: true,
      entries: [the list of all entries],
      activeEntryData: [the active entry object]
}
```



Task 2: When the `.new` button is clicked, switch to the `CreatingEntryView`. To do so, call `CreatingEntryView.render()` with the `$entry` element as the first parameter. This transfers control of the element to the `CreatingEntryView`, which will render new contents into the `#entry` element. In effect, we're allowing `CreatingEntryView` to take over and let the user create a new entry (see `CreatingEntryView` section below).

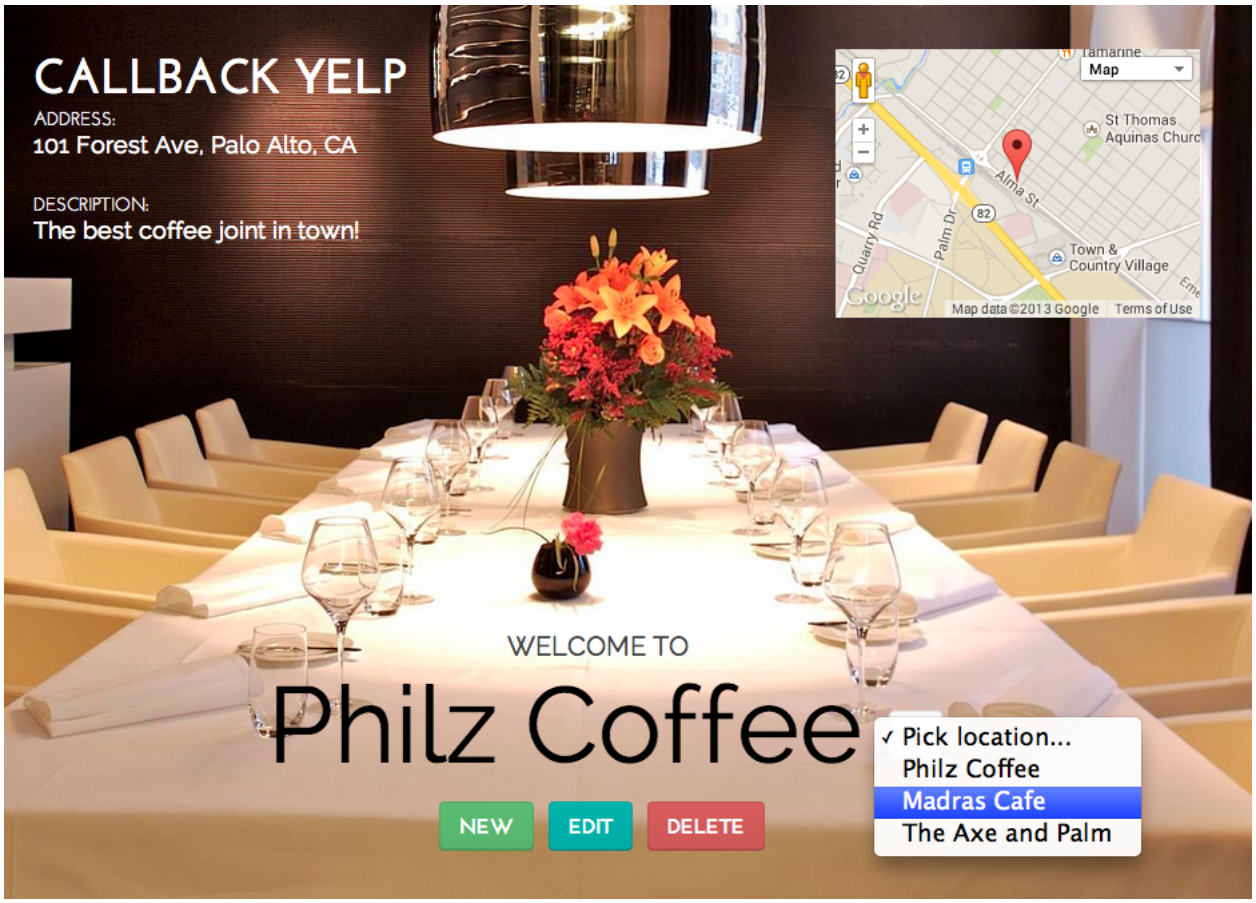
Task 3: When the `.edit` button is clicked, switch to the `EditingEntryView`. To do so, call `EditingEntryView.render()`. This transfers control of the element to the `EditingEntry`, which will render new contents into the `#entry` element. In effect, we're letting the `EditingEntryView` take over and let the user edit the currently active entry (see `CreatingEntryView` section below).

Task 4: When the `.delete` button is clicked, delete the currently active entry. If there are entries remaining, re-render the `EntryView` (call `EntryView.render()`) with one of them as the new active entry. Otherwise, if no entries are left, render the `CreatingEntryView`.

If an error occurs while deleting the entry, display the error message returned by the server in the `.error` div. Do not re-render the `EntryView` in this case.

Task 5: The user can choose which entry he/she would like to view using a dropdown, which is described by the HTML `select` tag. When the value of the `select` tag changes, re-render the `EntryView` so that it shows the entry the user selected.

Attach a `change` event listener to the select tag inside `#entry`, and get the ID of the entry the user selected by using `$select.val()` (assuming `$select` is the jQuery object corresponding to the `select` element). Then, find the corresponding entry with that ID, and re-render the `EntryView` so that entry is active.



Task 6: Render the `GoogleMapView` sub-view. Pass in a jQuery object corresponding to the `.map` div as the first parameter to `GoogleMapView.render()`. `GoogleMapView`'s job is to display an interactive Google Map centered around the active entry's location.

`CreatingEntryView` (js/creating-entry-view.js):

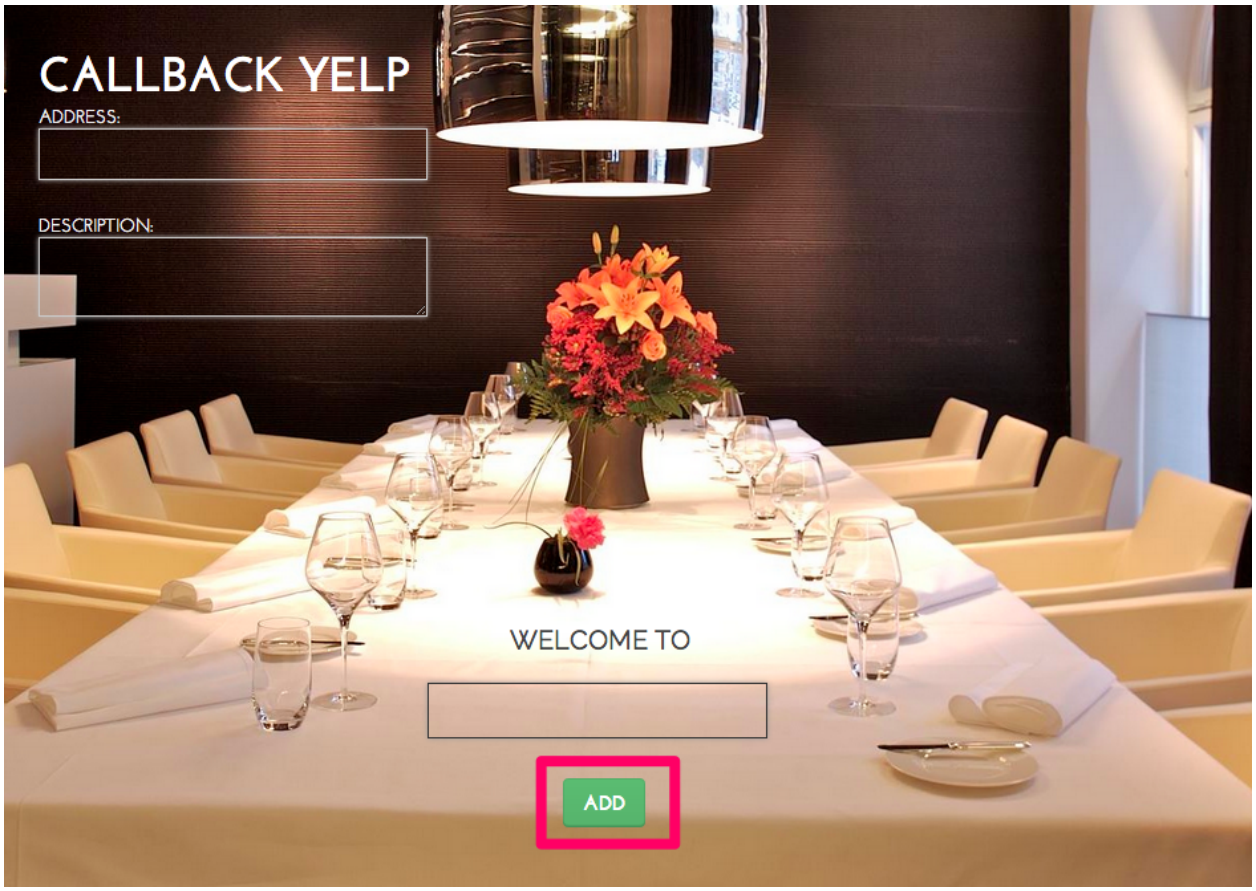
The `CreatingEntryView`'s main job is to let the user create new entries. A user should be able to input an entry's name, description, and address, and have that entry added to the server.

The `CreatingEntryView` has jurisdiction under the `#entry` element. Its `render()` function

is called when the `.new` button is clicked by the `EntryView`. `EntryView` passes in a jQuery object representing the `#entry` element to this function. Your job is to implement this `render()` function. The required functionality is as follows:

Task 1: Render the `#entry-template` Handlebars template into the `#entry` element. Pass an object to the Handlebars render function in the following form:

```
{
      creating: true,
      entries: null,
      activeEntryData: null
}
```



Task 2: When the `.add` button is clicked, create a new entry with the name, description, and address stored in the `input` and `textarea` fields. Once you've done this, switch to the `EntryView` (i.e. render the `EntryView`) and ensure the newly created entry is shown (i.e. the newly created entry is the active entry).

If an error occurs while adding the entry, display the error message returned by the server

in the `.error` div. Do not render the `EntryView` in this case; let the user fix his/her input and try again.

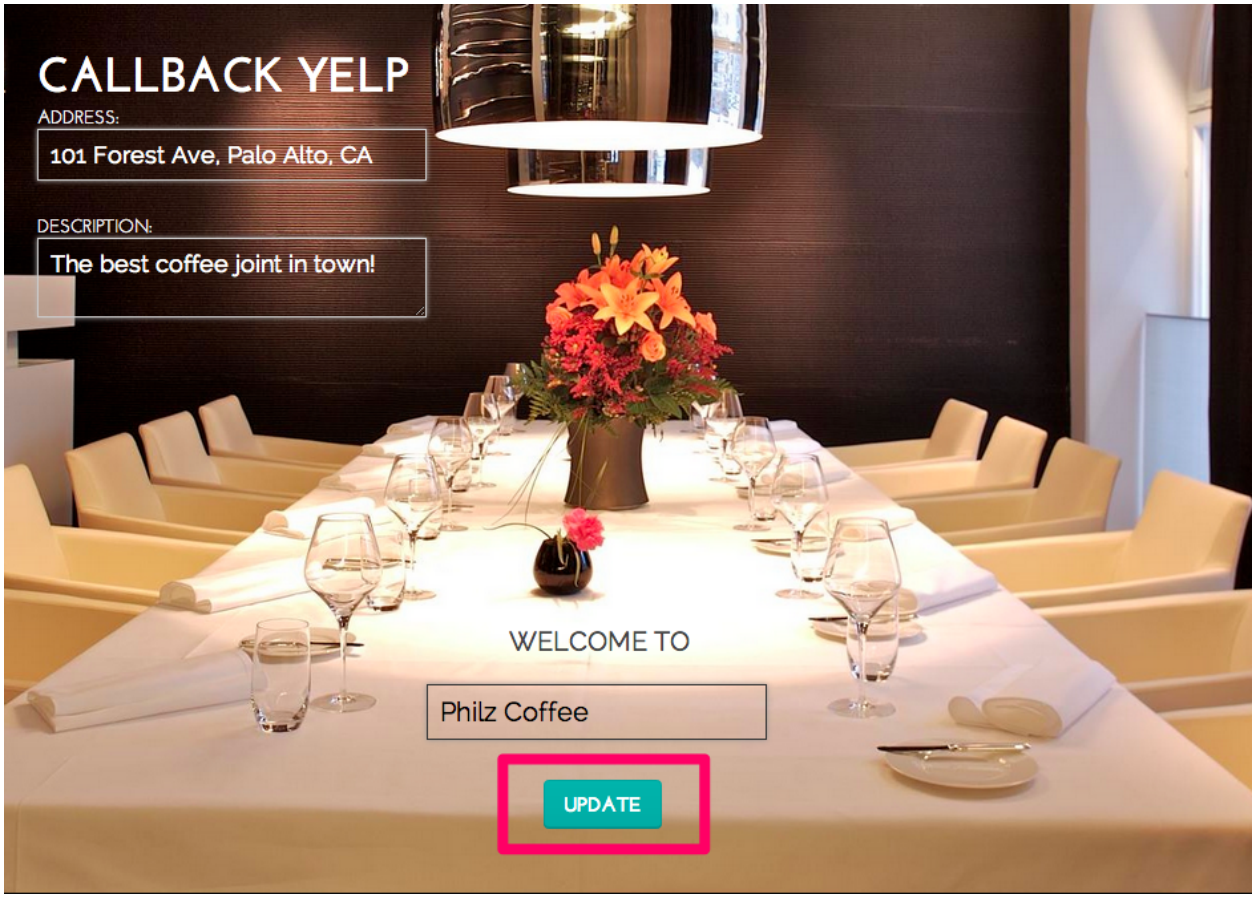
`EditingEntryView` (js/editing-entry-view.js):

The `EditingEntryView`'s main job is to let the user edit an existing entry. A user should be able to edit an entry's name, description, and address, and have that entry updated on the server.

The `EditingEntryView` has jurisdiction under the `#entry` element. Its `render()` function is called by the `EntryView` when the `.edit` button is clicked. `EntryView` passes in a jQuery object representing the `#entry` element to this function. It also passes the entry that the user would like to edit. Your job is to implement the `render()` function. The required functionality is as follows:

Task 1: Render the `#entry-template` Handlebars template into the `#entry` element. Pass an object to the Handlebars render function in the following form:

```
{
      editing: true,
      entries: null,
      activeEntryData: [the active entry object]
}
```



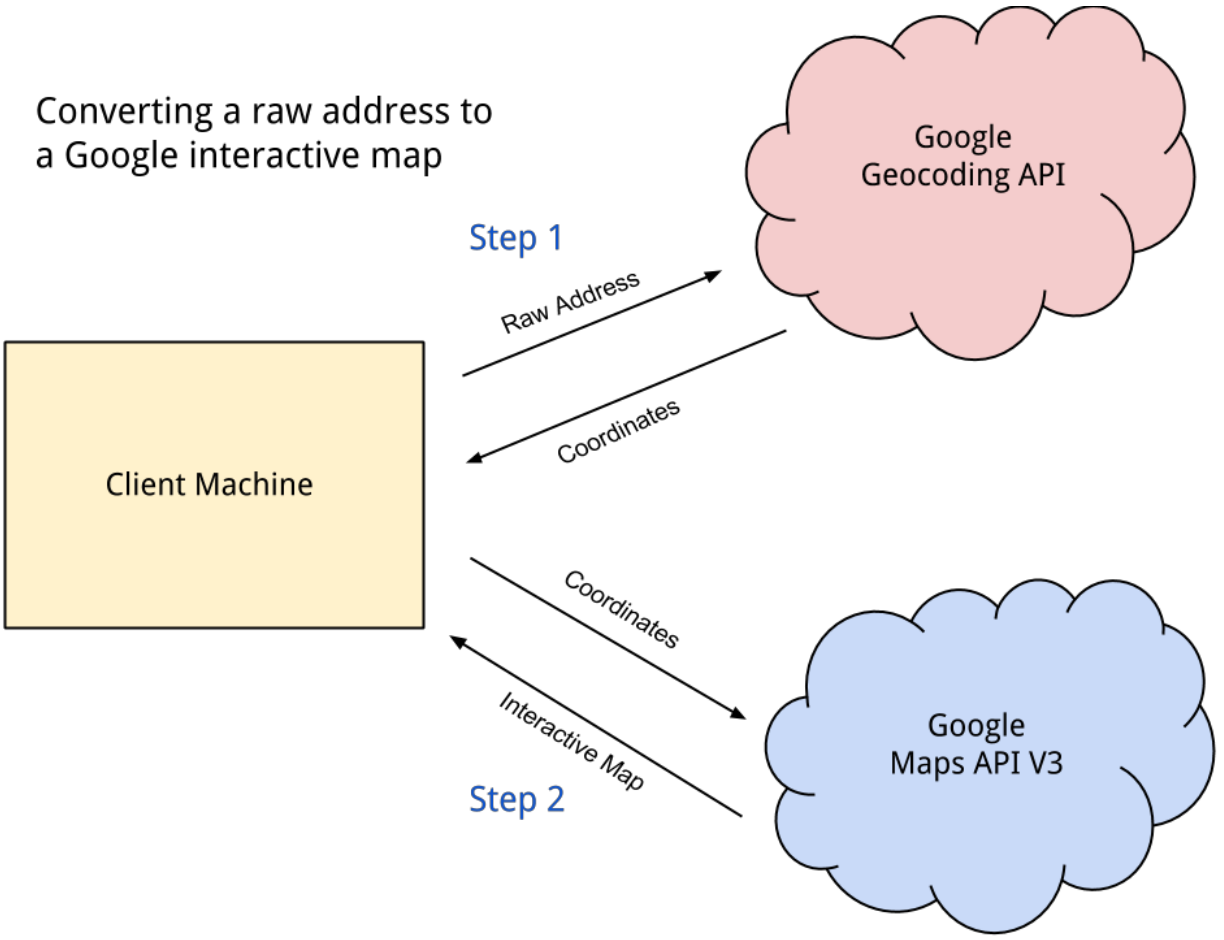
Task 2: When the `.update` button is clicked, update the active entry's name, description, and address with the values stored in the `input` and `textarea` fields. Once you've done this, switch to the `EntryView` (i.e. render the `EntryView`) and ensure the updated entry is shown (i.e. the updated entry is the active entry).

If an error occurs while updating the entry, display the error message returned by the server in the `.error` div. Do not render the `EntryView` in this case; let the user fix his/her input and try again.

`GoogleMapView` (js/google-map-view.js):

We require that you use the Google Maps JavaScript API v3 to display the address of each entry in an interactive map. As a web developer, you will frequently have to use unfamiliar web APIs. Consequently, we'll direct you to the resources you need to create an interactive map, but it'll be your job to interpret them and find how to accomplish this.

The general flow to create an interactive map looks like this:



First, you'll need to pass the address of the entry to the Google Geocoding API. This will return to you (latitude, longitude) coordinates that pinpoint where the address is in the world. Then, you'll need to query the Google Maps JavaScript API v3, giving it the coordinates and the element you want to render a map in. Google Maps will handle the rest, and create an interactive map!

All of your logic for this should go in the `GoogleMapView.render()` function (make sure to decompose, though).

Geocoding API
For the geocoding step, consult the documentation at <https://developers.google.com/maps/documentation/geocoding/>. More specifically, look at the "Geocoding Requests" and "Geocoding Responses" sections. Make sure to retrieve JSON data back so you can parse it.

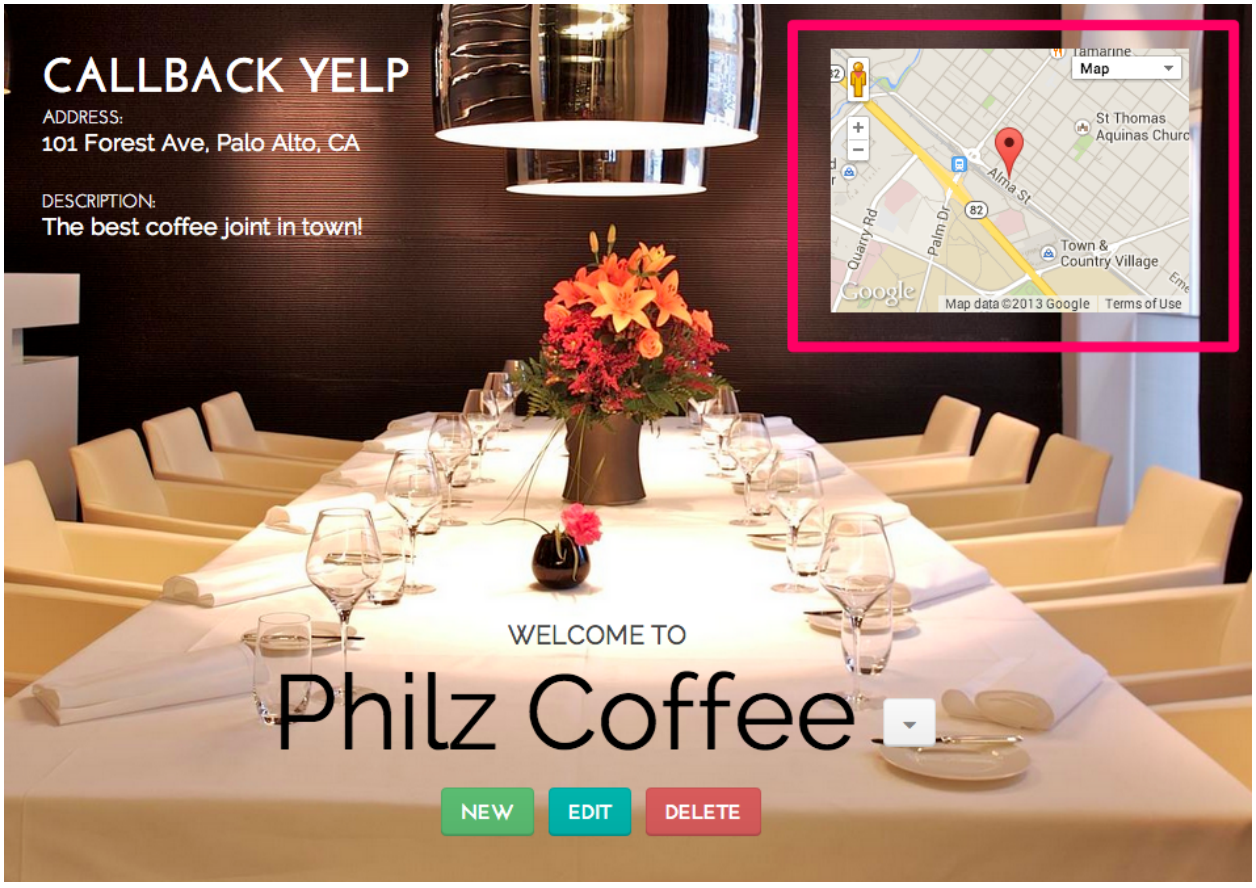
Google Maps JavaScript API v3
Once you have obtained the (latitude, longitude) coordinates corresponding to the raw address via geocoding, you must display an interactive Google map for the location.

We have already loaded the Google Maps JavaScript library as a `script` tag in index.html with the proper API key:

```
<script type="text/javascript"
src="https://maps.googleapis.com/maps/api/js?key=AIzaSyB4NQtL9PBc0KOeCIY2n5rD0N4El6k09zw&sensor=false">
</script>
```

You'll need to consult the documentation at <https://developers.google.com/maps/documentation/javascript/tutorial> to find how to employ the library. We'd recommend paying close attention to the "Hello World Example" section; your code will be very similar. Note that there's no need to add a load event listener; everything has already loaded by the time your `GoogleMapView` is rendered.

In addition to displaying an interactive map, we'd also like you to put a marker at the entry's location. Take a look at the simple marker example at <https://developers.google.com/maps/documentation/javascript/examples/marker-simple> to find how to accomplish this.



Grading:

We will grade your application based on two criteria: the functionality of your application and proper JavaScript coding style.

Functionality (60%):
You must implement all the required tasks above. Follow the design pattern we've laid out for this assignment, and do not violate the model/view abstractions. Decompose all common functionality.

Given that you have done the above and your application works, you should receive full marks. We will deduct points for incomplete/omitted features, lack of robustness, and bad design.

Style (40%):
Your style will largely be graded on consistency. Follow the style already present in the starter files. If you work with JavaScript in industry, the team you'll be working with will likely have a baseline style established. You should always match this style. Consistency is everything.

Make sure to document your code. It's hard for us to stress this enough. The goal is to make your code so understandable that we can read it with ease. Any complex logic that you had to reason through should be clarified with inline comments. All functions should be documented: what does the function do, what are its parameters, what does it return or pass to a callback, and, if applicable, how does it handle errors?

Assignment length

The reference solution is approximately 240 lines of code across all six files (lines are those with semicolons, colons, or opening/closing braces; not including comments). Good solutions will likely range from 200 to 400 lines of code. Exactly how long the assignment will take depends on the person; our estimate is 8-15 hours, but some people may take more, and others may take less.

Frequently Asked Questions (FAQ)

Question: I'm receiving the following error: No `Access-Control-Allow-Origin` header is present on the requested resource. Why am I getting this error?

Answer: Chrome provides very unhelpful error messages for debugging AJAX issues. As such, your issue could be entirely unrelated to submitting cross-origin requests. Before posting on Piazza for further help, quickly run through this list of common fixes.

1. Before sending a POST request, ensure that your `Content-Type` request header is properly set to `application/json`.

2. Ensure that you passed the correct HTTP request method and URL to your `XMLHttpRequest` open function.

3. Ensure that data for GET requests are passed in through the URL. You should not pass any parameters to the associated `XMLHttpRequest` send function.

4. Conversely, ensure that data for POST requests are passed in through the `XMLHttpRequest` send function.

Honor code:

All of your code for this assignment should be your own original work. Do not copy other students' work. If you referenced online sources, cite them as a comment in your code.

Submission:

Visit <https://web.stanford.edu/class/cs42/cgi-bin/yelp.php> and submit a zip file of your entire piazza assignment directory (all starter files should be included in the zip). You may submit as many times as you'd like before the deadline; only your last submission will be retained.