Final Project: Facebook Friends Mapper+

A Web-Application Mash-Up

Mandy Wong, Raymond Liaw, Chia-Shang Liu (Timm)
Professor Hollan
COGS 121
March 17, 2009

Project Link: http://sdcc15.ucsd.edu/~rliaw/121_final

OUTLINE

I. **Goal & Overview**

II. **Interface Walkthrough**

III. **Design Process**

IV. **Aesthetics**

V. **Project Difficulties & Solutions**

VI. **Lessons Learned**

VII. **Future Improvements**

VIII. **In-depth Source Code Documentation**

DOCUMENTATION

I. Goal & Overview

Our goal for this project was to challenge ourselves and put what we learned about human computer interaction and web design into practice. We began with the intention of creating a Facebook application or Facebook-based website that would allow for a Facebook user to visit his or her friends in their hometowns and eat together with them at a restaurant in the area. We envisioned an application or website that mapped Facebook friends onto Google Maps by hometown and displayed available restaurants in the vicinity along with user ratings from Yelp. The idea was to grab the user's friends' hometown data from Facebook as well as grab restaurant data from Yelp based on hometown location. Then both the hometown data and the restaurant data would be mapped onto Google Maps. Also, we wanted to include some filtering or sorting functions such as filtering the data to display only at the selected city and sorting a list of hometowns alphabetically. This required the creation of a mash-up of the Facebook API, the Google Maps API, and the Yelp API. We felt that this would not only

be a useful and fun website to design, but that it would be a challenging and profitable project that would help us to learn more about human computer interaction programming practically.

II. Interface Walkthrough

The website first checks whether or not a user is already logged into Facebook. If there is no one logged in, the website will direct page to Facebook's login page. Once a user is logged in, a logout button is displayed at the top left and below the button a list of the hometowns of the user's friends are immediately displayed alphabetically in the left pane including a number in parentheses behind each hometown indicating the number of friends from that city. At the same time, the hometowns of the user's friends begin to be mapped onto the Google Maps canvas using Google Maps markers. Once the user clicks on a hometown of choice in the left pane, the Google Maps API zooms into a city-scale zoom level and maps out restaurants (using markers) in a given radius based on a set center point on the map. Also, the user's friends from that city are displayed in a list on the right pane. A filter for restaurant type can also be set so that only restaurants of the type of choice are displayed. Clicking on a restaurant marker pops up a Google Maps notification showing the restaurant name, the rating of the restaurant, a picture of the restaurant, the category or categories the restaurant applies to, the restaurant's address, the restaurant's phone number, and an external hyperlink to Yelp for reading detailed reviews on the restaurant.

III. Design Process

As mentioned in the overview section, we started with the idea that users would be able to map their friends from Facebook and display restaurants from Yelp onto Google Maps. After agreeing on the proposed idea, we decided the steps for the project and set deadlines for when these steps should be completed. The basic structure of our five week schedule was as follows:

- **Week 1:** Learn how to plot markers with clickable tooltip windows on a Google Maps canvas using its API.
- **Week 2:** Learn the Yelp and Facebook API's separately so that we are familiar with plotting filtered Yelp data onto Google Maps and with parsing Facebook friend data from Facebook.
- **Week 3:** Develop a working model and data filter by mashing the three API's together.
- **Week 4:** Finish up the application; fix any last bugs, make small improvements, design the layout for the website, etc.
- **Week 5:** Finalization and documentation of the project.

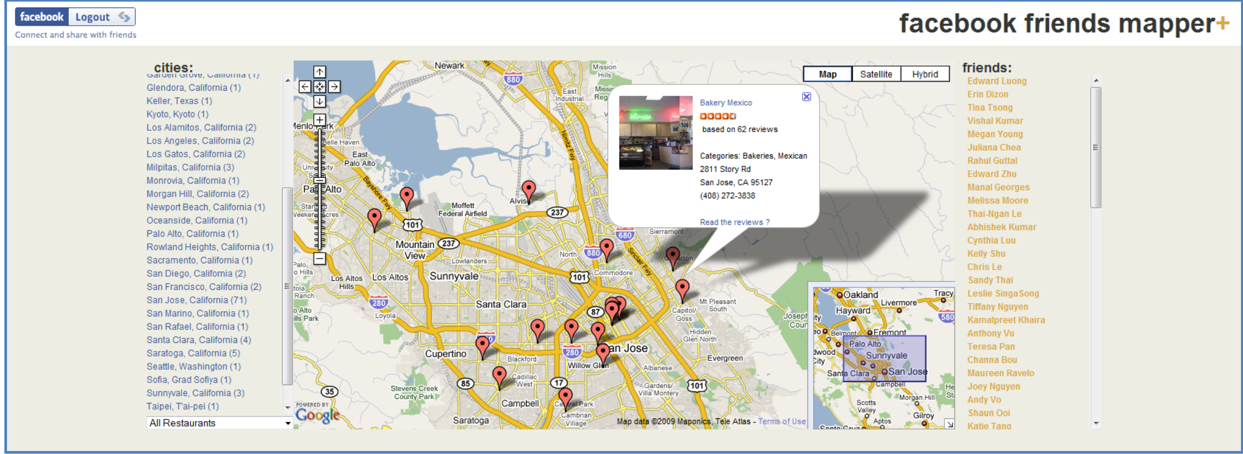
Since most of the design process was learning the different API's and dealing with JavaScript, it makes the most sense for us to discuss the coding process in detail. Our first step was to learn how to use the Facebook API to access a user's friend data as well as the friends' hometown data. To do this, we wrote code according to the Facebook API that performs a batch request to obtain a Facebook user's data.

First, the script for the Facebook login and the feature loader file are executed. The Facebook API key for our application is verified and the cross-domain communication channel file is defined. Using the `user_getInfo()` API call, the application requests for four values from each of the user's friends: UID, first name, last name, and hometown location. Arrays are defined for a list of friends, the names of friends, hometowns, and hometowns with the states they are in. The application checks for whether or not the friends have valid non-null hometown values and then stores these into an array. Hometowns that are unique in the batch are stored in an array and stored as indexers. They are arranged alphabetically while duplicate hometowns are not stored but increase the counter for indexed hometowns. A list of hyperlinks is generated and the alphabetically arranged hometowns are displayed

in a list on the left pane so that the user can click on them. The counters of repeating hometowns are displayed as numbers next to each hometown to indicate how many friends are from that same hometown. Also, the data for non-null hometowns is stored and used for displaying the list of friends from the same city in the right pane of the website.

Using the Google Maps geocoder, the hometown values are converted to GPS coordinates so that they can mapped using map markers on load immediately after login. An `onClick()` event also uses the Google Maps geocoder and converts the clicked city value to a GPS latitude longitude. HTML elements are created and the clicked city coordinate is sent to the `showCity()` function to set the center of map canvas and to zoom to a city-scale view. Using this same city value, the application performs a restaurant query via the Yelp API and pulls restaurant listings from the city, filtering out unrelated business data. Yelp provides GPS coordinates for its restaurant locations so the Google Maps geocoder is not used for getting the coordinates, but we do still use the Google Maps API to plot markers at these coordinates. Also, these markers are click-able and clicking on one brings up a marker tooltip window that displays filtered information about the restaurant (name, star rating, photo, category, address, phone number) and an external hyperlink for reading reviews on Yelp. Below the list of hometowns in the left pane is a drop-down menu with an `onChange` property that applies a restaurant type filter. This changes the type of restaurants requested from Yelp and shows them based on food type (e.g. Chinese or Italian food). Finally, with all the content working correctly, we uploaded the files to UCSD's server which provides each student with personal web space and styled the page using CSS.

IV. Aesthetics



For the aesthetic design of the website, we were shooting for clean and simple. We wanted the user experience to be interactive and visual so we focused on the map by placing the large map canvas in the center of the page. Since users are familiar with and often use navigational menus on the left side of web pages, we conveniently placed the toolbar for choosing friends' hometowns on the left side of the page. We also made the response to selecting a hometown, a list of friends from that hometown, display on the right side of the page. The logout button for Facebook is visibly and intuitively placed at the top left of the page. One possible issue with the interface design is the fact that the initially mapped hometowns use the same type of markers as the restaurants that are mapped after the user selects a hometown. This creates a consistency issue because, besides the fact that the same markers are used to mark different things – cities versus restaurants – the initial hometowns are not clickable while the restaurant markers are clickable for displaying tooltip windows and shifting the center of the map canvas when it is necessary to do so to include the entire tooltip window within view. Overall, the website design style for our application is very simple and to the point. Users who have never used the application should be able to easily recognize the functions and successfully navigate throughout the application.

V. Project Difficulties & Solutions

While the current mash-up website gets the job done, there are a few issues that need work for optimization and there is still room for other improvements. Some of these issues are because of our lack of previous experience with combining different API's, but our goal in doing this project was to be challenged and to get experience in this area. There were two very challenging issues that we encountered during the creation of this application: one regarding asynchronous and synchronous calls in Javascript and the other being the result of an obscure “speed limit” on Google Map’s geocoder.

Initially, we designed the application to loop through a number of individual API calls in order to obtain friends’ data from Facebook. Furthermore, in each iteration of the for-loop, the application was designed so that the mapping functionalities would apply after each Facebook API call. This design flaw was mainly the result of our inexperience with Javascript rather than faulty planning. We soon discovered the amount of problems in this design. First off, there was a tremendous performance issue as making hundred of API calls in sequence individually. Secondly, we hit the frustrating issue of passing data out from nested functions (the result of the complicated asynchronous vs. synchronous calls). After meeting and discussing with Gaston, we were helped in fixing the issue of storing variables between functions. Inadvertently, we also found the solution to our flawed method of making HTML requests.

The structure of the function and the API call was altered to perform in batched mode, where we queue up a sequence of API calls, then group them all into a single HTTP request to the Facebook server. Because only a single HTTP request (versus our previous 100+ requests) is sent, this provided us with an immense performance boost. Additionally, it allowed for us to work with all the needed data in one loop as compared to trying to store and transfer data values outside of multiple for-loops.

Our second roadblock occurred near the end of our project. We had successfully managed to map out the hometown locations of friends in Google Maps along with displaying an alphabetically-

sorted, interactive HTML display of friends' hometown cities. However, it soon became apparent that several hometowns were missing on the map (e.g. Kyoto, Japan). In order to isolate the problem, we passed only a single city (namely one of the missing cities) to the mapping function. To our bewilderment, Google Maps would successfully plot the city. However, by doing so, we were able to isolate where the bug could potentially be coming from.

The two theories behind this issue were: the data from Facebook was erroneous or our method of converting the city name into a map-able GPS coordinate was flawed. The first theory was easily tested and discarded with an echo or alert to the questionable data being obtained from Facebook. This meant that our city-to-coordinate conversion was flawed. After several hours, we learned that the issue arose from Google's geocoder (the API call we were making to convert a city name string into a numerical GPS coordinate). Apparently, Google placed a query limit on the geocoder which caused it to stop responding if there were too many requests in a short period of time. In order to fix this issue, we would have to create a foolproof delay system as we converted our large list of cities.

This was achieved by several steps: storing the cities in an array within our JS file, creating a function to geocode the city as well as tracking array position being geocoded, and creating a mapping function with error detection. Although storing the cities into an array is straightforward (a simple for-loop), the interesting parts are seen in the geocoding and mapping functions. In order to bypass the imposed "speed limit," we utilized the `setTimeOut()` function with a 100ms delay. This created a time gap between each geocode call. Furthermore, we added a counter to determine how far down the array of cities was being geocoded. After geocoding, the mapping function is called. The mapping function required an error check to determine if a successful GPS coordinate was generated before attempting to map the location. With an if-else statement, we set the function to check if there was a proper response from the geocode call. If there was a bad response, then we could decrement the array counter, increment the delay, and re-call the geocode function. Otherwise, a map marker would be

created and rendered on screen. In hindsight, the solution is rather basic; nonetheless, given our beginner skills with Javascript, we still see the solution as rewarding and proof of creative-thinking.

Although there were many other bugs and issues, the group felt that these two issues in particular challenged us the most and improved our logic and Javascript skills the most. Even though some solutions were simply combining relatively simple steps (the “speed limit” problem) and some are still a little ambiguous to us (the asynchronous issues), our method of thinking and tackling problems while programming managed to drastically improved from just these two issues – which is why these two were worthy of mention.

VI. Lessons Learned

Throughout this project, we learned the power of mashing different API’s together into one application. The three API’s we looked at were the Facebook API, the Yelp API, and the Google Maps API. From the Facebook API, we learned how to grab data from Facebook and pass that data onto objects or arrays to be used in Google Maps. We also learned how to sort the Facebook output alphabetically. For the Yelp API, we learned how to filter and obtain restaurant data from Yelp and how to make a filter for the restaurants by restaurant type. As for Google Maps, we learned how to customize the map canvas such as resizing it and how to add in map controls such as controls for changing the zoom level. We also learned how to plot markers on the map and how to display tooltip windows upon clicking the markers. In addition to the technical skills that we learned from doing the project, we also learned how to work in a group with other team members and how to cooperate with one another to meet deadlines.

VII. Future Improvements

There are numerous possibilities for future improvements to our application. One possible

improvement is to work toward cross-browser support. At this point, the application only works perfectly on Mozilla Firefox and Opera. We also tested the application on Google Chrome as well as Internet Explorer 8 and we found that the drop-down menu restaurant filter in the left pane for choosing restaurant types has issues with the onChange JavaScript event. As mentioned in the 'Aesthetics' section, there is a possible issue with marker usage. This problem can be alleviated by making the initially plotted hometown markers clickable and/or by using different colors or types of markers for restaurants so that there is not a consistency issue created by cognitive association. Also, the friends list displayed on the right pane could be expanded to include photos of friends or other options. Aside from being able to view restaurants in the area, we could also include options to display other places of interest. Since we are using the Yelp API and grabbing business data from Yelp, we can include other categories such as Art and Entertainment, Nightlife, Shopping, etc. We also need to enhance the Facebook login and logout functionality. Currently, the logout button successfully logs the current user out of Facebook but does not clear the Facebook data already displayed on the page. Other possibilities include making a version of our application that is incorporated within Facebook itself or to make a mobile version to be used on handhelds such as cell phones and PDA's.

VIII. In-depth Source Code Documentation

The application begins with initializing the Facebook API (using the API key and a cross domain channel receiver). Once initialized, a check for a required login for Facebook is performed:

```
var api = FB.Facebook.apiClient;
api.requireLogin(myFunc);
```

This will prompt the user with a login window if it is detected that the user has not signed in. Upon page load, the Google Map's API is initialized and the map is rendered on the screen.

```
<body onload="initialize()" onunload="GUnload()">
```

```
      function initialize() {
            if (GBrowserIsCompatible()) {
                  map = new GMap2(document.getElementById("map_canvas"));
                  map.enableScrollWheelZoom();
                  map.setCenter(new GLatLng(36.6, -96.8), 4);
                  map.addControl(new GLargeMapControl());
                  map.addControl(new GMapTypeControl());
                  map.addControl(new GOverviewMapControl(new
                    GSize(200,200)));
                  map.setMapType(G_NORMAL_MAP);
            }
      }
```

This specifies the attributes of the map that is to be rendered. A map is rendered in the HTML element 'map_canvas' and it is centered onto the United States on a country-level scale (specified in the `GLatLng` parameters along with the zoom value of 4).

Once the Facebook session is ready, Facebook can begin obtaining a list of friends in order to later obtain their data. The application makes an API call to grab the logged in user's friends:

```
      api.friends_get(null, myFunc);
```

This returns an array of friend UIDs (numbers that identify friend's accounts) as a result which is passed to `myFunc`.

After a small check to determine if the result is not null, the application performs a second API call to obtain the data for the list of friends just previously obtained:

```
      api.users_getInfo(result,
        ['first_name','last_name','hometown_location'], myFunc);
```

The users_getInfo() parameters follow:

```
      users_getInfo(UID, fields, function);
```

As we can see, the result from the `friends_get()` call is passed to the `users_getInfo()` function as a parameter to search through and obtain the specified fields. The result is passed again to another `myFunc`.

Within this embedded function, we have a for-loop that check through all results to determine if the hometown value is not null. Furthermore, it also checks if the city value within the `hometown_location` array is not null:

```
      var names = {};
      var l = 0;

      for(i in result) {
            if(result[i].hometown_location &&
              result[i].hometown_location.city) {
                  names[l] = result[l];
                  l++;
            }
      }

      saveName(names);
```

If the checked results are not null, then the results are saved an Array `names`, which is utilized later to match friend names to city names. We will get to the `saveName(names);` call later.

Next, the application uses a new Array called `friends` and string indexes as a method for checking for duplicate cities. For each duplicate found, the array increments the city value by one (as opposed to storing another duplicate string of the city). If there is no repeat found, a new index is created in the array with the index set to the city name.

```
      var friends = {};

      if(friends[result[i].hometown_location.city]) {
            friends[result[i].hometown_location.city] += 1;
      } else {
            friends[result[i].hometown_location.city] = 1;
      }
```

Additionally, the application stores the city names (after the duplicate filter) in two separate arrays – one for geocoding and one for later HTML output display (in “city, state” format).

```
      var citydb = new Array();
      var cityStr = new Array();
      } else {
            citydb[j] = result[i].hometown_location.city;
            cityStr[j] = result[i].hometown_location.city + ", " +
            result[i].hometown_location.state;
            j++;
            friends[result[i].hometown_location.city] = 1;
      }
      citydb.sort();
      cityStr.sort();
      setAddress(citydb);
```

We perform a `sort()` on the two new arrays in order to alphabetically order its values – which helps with usability when navigating a long list of city names. The `setAddress();` function is defined in the `mapper.js` file.

In the `mapper.js` file, we will see the following three functions:

```
function setAddress(array) {
      for (i in array) {
            address[i] = array[i];
      }
      showLocation();
}

function addAddressToMap(response) {
      if (!response || response.Status.code != 200) {
            nextAddress--;
            delay++;
      } else {
            place = response.Placemark[0];
            point = new GLatLng(place.Point.coordinates[1],
            place.Point.coordinates[0]);
            marker = new GMarker(point);
            map.addOverlay(marker);
      }
      showLocation();
}

function showLocation() {
      if (nextAddress < address.length) {
            setTimeout('geo.getLocations("'+address[nextAddress]+'",
              addAddressToMap)', delay);
            nextAddress++;
      }
}
```

The `setAddress()` function is mainly a method of storing data so we can pass it between functions defined in the JS file. It also initializes the mapping functions: `showLocation()` and `addAddressToMap()`. Within `showLocation`, we see that it loops through the address array defined in `setAddress()` and performs a delayed call on the geocoder `getLocations()`. This is the work around for the set speed limiter on geocode calls set by Google. Additionally, we see that a counter variable is incremented and the result of `getLocations` is passed to `addAddressToMap`. There it performs a response check where, if there's a bad response, it loops back to `showLocation` and decrements the counter back to its previous value while increasing the delay value. If the response was

good, the application utilizes Google Map's API and creates saves the result to a point marker which can be later mapped.

Looking back to the `saveName(names)` call, we will see that the function in the JS file is as follows:

```
var names = [];

function saveName(array) {
      var j=0;

      for (i in array) {
            names[j] = array[i];
            j++;
      }
}
```

This, too, serves as a method to save an array of data in the JS file to pass to other functions. It will become apparent as to why in the next part of the code. In the HTML script, the application begins creating interactive HTML elements:

```
var listDiv = document.getElementById('output');

for(var k=0; k<citydb.length; k++) {
      var displayElement = document.createElement('a');
      displayElement.href = "javascript:showCity('" + citydb[k] + "')";
      displayElement.innerHTML = cityStr[k]+'('+friends[citydb[k]]+')';
      listDiv.appendChild(displayElement);
      listDiv.appendChild(document.createElement('br'));
}
```

Utilizing DOM, we create a list of hyperlinks. The output text is drawn from looping through the `cityStr` array created earlier (the array that stored "city, state"). It's target, however, points to a Javascript call to `showCity()` using the values of Array `citydb` as the passed parameters (recall that since the two arrays were alphabetically sorted, the values that are in equivalent positions between the two arrays are equivalent to each other). The `showCity` function integrates the rest of the Yelp functionality into the application (shown in the result of clicking on a city hyperlink – e.g. map zooms into specified city and restaurants are mapped out).

The `showCity` function involves various straightforward functions also defined in the mapper.js file to successfully create an interactive map of restaurants in a target city. The function itself contains two main parts: one to display friends from the target city and one to create the map marker.

```
	function showCity(city) {
	var friendOutput = document.getElementById('friendbar');
	friendOutput.innerHTML = '';
	g_city = city;

	if (geo) {
		map.setCenter(geo.getLatLng(g_city, function(point) {
			if (!point) {
				alert(g_city + " not found");
			} else {
				map.setCenter(point, 11);
				var marker = new GMarker(point);
				map.addOverlay(marker);
				updateMap();
			}
		}));
	}

	for (var i=0; i<names.length; i++) {
		if (names[i].hometown_location.city == city) {
			friendOutput.innerHTML += names[i].first_name + ' ' +
			  names[i].last_name;
			friendOutput.appendChild(document.createElement('br'));
		}
	}
}
```

The friends display is seen in the for-loop at the bottom of the code snippet. It loops through the `names` array (created at the beginning of the application) and performs an if-search to match friends whose hometown city data matches that of the specified city (passed to the function through the generated hyperlinks earlier). The string of friend names are concated to the `innerHTML` of the HTML element 'friendbar'. On a side note, the `innerHTML` of 'friendbar' is set to a blank string at the start of every showCity() call in order to clear the HTML display when a new city is selected.

The second, mapping functionality is found in the if-statement in the center of the code snippet. The function geocodes the city string (there is no need for the speed limit fix since there is only a single query) and sets the map center as well as assigns a marker to the newly generated point. The call to `updateMap` continues to addition of Yelp data into the application.

```
function updateMap() {
      var yelpRequestURL = constructYelpURL();
      map.clearOverlays();

      var script = document.createElement('script');
      script.src = yelpRequestURL;
      script.type = 'text/javascript';
      var head = document.getElementsByTagName('head').item(0);
      head.appendChild(script);
      return false;
}
```

This function initializes the Yelp API, calls `constructYelpURL`, and clears the map of its overlays (deletes existing map markers, etc.). The function `constructYelpURL` queries Yelp to retrieve the needed restaurant data:

```
function constructYelpURL() {
      var mapBounds = map.getBounds();
      var URL = 'http://api.yelp.com/' + 'business_review_search?' +
        'callback=' + 'handleResults' +
        '&category=' + g_restaurant +
        '&num_biz_requested=20' +
        '&tl_lat=' + mapBounds.getSouthWest().lat() +
        '&tl_long=' + mapBounds.getSouthWest().lng() +
        '&br_lat=' + mapBounds.getNorthEast().lat() +
        '&br_long=' + mapBounds.getNorthEast().lng() +
        '&ywsid=' + YWSID;
      return encodeURI(URL);
}
```

The callback is passed to the `handleResults` function which catches empty responses, saves returned restaurant data, and initializes the `createMarker` function.

```
function handleResults(data) {
      if(data.message.text == "OK") {
            if (data.businesses.length == 0) {
                  alert('No reviews for businesses were found near that
                    location');
                  return;
            }

            for (var i=0; i<data.businesses.length; i++) {
                  biz = data.businesses[i];
                  createMarker(biz, new GLatLng(biz.latitude,
                    biz.longitude), i);
            }

            for (var i=0; i<data.businesses.length; i++) {
                  biz = data.businesses[i];
            }
      } else {
```

```
                alert("Error: " + data.message.text);
            }
    }
```

```
    function createMarker(biz, point, markerNum) {
        var infoWindowHtml = generateInfoWindowHtml(biz);
        var marker = new GMarker(point, icon);

        map.addOverlay(marker);

        GEvent.addListener(marker, "click", function() {
            marker.openInfoWindowHtml(infoWindowHtml, {maxWidth:600});
        });

        if (markerNum == 0) {
            marker.openInfoWindowHtml(infoWindowHtml, {maxWidth:600});
        }
    }
```

The createMarker function takes the passed restaurant data and GPS point and adds a Google Map at the given point value. These markers are assigned a “click” attribute which basically creates a dynamically generated tooltip when the marker is clicked on by the user. The content of the tooltip is generated from the `generateInfoWindowHtml` function initialized in the first line of the function.

```
    function generateInfoWindowHtml(biz) {
        var text = '<div class="marker">';

        text += '<img class="businessimage" src="'+biz.photo_url+'"/>';

        text += '<div class="businessinfo">';
        text += '<a href="'+biz.url+'" target="_blank">'+biz.name+'
                 </a><br/>';
        text += '<img class="ratingsimage" src="'+
                 biz.rating_img_url_small+'"/> based on ';
        text += biz.review_count + ' reviews<br/><br />';
        text += formatCategories(biz.categories);
        if(biz.neighborhoods.length)
            text += formatNeighborhoods(biz.neighborhoods);
        text += biz.address1 + '<br/>';
        if(biz.address2.length)
            text += biz.address2+ '<br/>';
        text += biz.city + ', ' + biz.state + ' ' + biz.zip +
                '<br/>';
        if(biz.phone.length)
            text += formatPhoneNumber(biz.phone);
        text += '<br/><a href="'+biz.url+'" target="_blank">Read the
                reviews ?</a><br/>';

        text += '</div></div>'
        return text;
    }
```

The function is passed the restaurant data from `createMarker` and formats the data into a readable HTML output saved to a variable named text which is returned to `createMarker`. Three other formatting functions are utilized by `generateInfoWindowHtml`: `formatCategories()`, `formatNeighborhoods()`, and `formatPhoneNumber()`. These three functions simply format specific data so its output is in a familiar format (e.g. phone number as (000)000-0000). Looking back at `createMarker()`, we see that this returned data is assigned into the tooltip display value.

With the given code, the application will now: prompt the user for login, map out all hometown cities of valid friends, generate an interactive list of mapped hometowns, zoom into a city and map out restaurants in the vicinity if clicked on, and generate restaurant-specific tooltips for each restaurant marker generated. The main functionality and goal of the application is complete; however, there is still one last function: a restaurant category filter.

In the HTML, there is a drop-down form with an assigned `onchange` attribute:

```
            <form onchange="showRestaurant(this.retaurant.value);">
```

Upon changing the drop-down selection, it calls `showRestaurant` – passing it the value of the selection that was changed to.

```
      if(!g_restaurant) g_restaurant = "restaurants";

      function showRestaurant(restaurant) {
            g_restaurant = restaurant;
            if (geo) {
                  map.setCenter(geo.getLatLng(g_city, function(point) {
                        if (!point) {
                              alert(g_city + " not found");
                        } else {
                              map.setCenter(point, 11);
                              var marker = new GMarker(point);
                              map.addOverlay(marker);
                              updateMap();
                        }
                  }));
            }
      }
```

By default, the global variable `g_restaurant` specifies which type of restaurants to display when `updateMap` is called (default: all restaurants or ‘restaurants’). The `showRestaurant` function simply

changes the value of `g_restaurant` to the value it was passed. This changes the parameters of the Yelp query and, thus, changes the results returned. A new set of restaurants – specific to the passed restaurant type – are then mapped onto the Google Map.

And finally, we have our Facebook Friends Mapper+.