Google Maps with Node

Overview

Mapping applications are very popular on desktop computers, and arguably one of the most popular and frequently used applications on mobile devices (after music, mail and web browsers).

MapBlast (now MSN Maps) was an early web mapping application, and Yahoo's MapQuest is still around, but Google Maps is now the 800 pound gorilla of online maps.

In this assignment, we will be using the Google Maps API and creating a custom node server.

i) You can find a nice history and overview of the capabilities of the Google Maps API at: http://en.wikipedia.org/wiki/Google_Maps

ii) The home page for the google maps API is here:
https://developers.google.com/maps/

iii) The home page for the google maps Web API is here:
https://developers.google.com/maps/web/

iv) The "Hello World" map tutorial is here:
https://developers.google.com/maps/documentation/javascript/tutorial

v) The Google Maps API, which lists properties, methods and events for classes like Map, Marker and Polygon, is here:
https://developers.google.com/maps/documentation/javascript/reference

vi) Lot's of sample code is here:
https://developers.google.com/maps/documentation/javascript/examples/

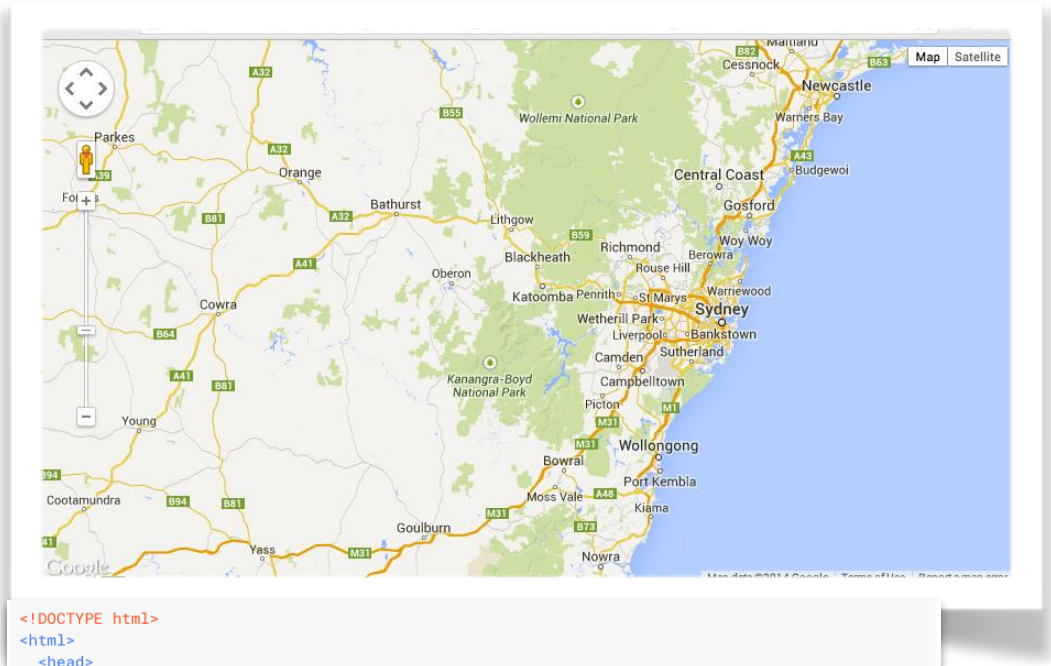
vii) The API signup page is here:
https://developers.google.com/maps/documentation/javascript/get-api-key

Part I. Google Maps - Markers and InfoWindows

1) Go ahead and sign up for the google maps API and get your API key:
https://developers.google.com/maps/documentation/javascript/get-api-key

2) Once you have your API key, go ahead and test it with the Hello, World example from the google site. Name it **map-simple.html**. Sydney, Australia should be visible on the map:

https://developers.google.com/maps/documentation/javascript/tutorial




```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Map</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
      html, body {
        height: 100%;
        margin: 0;
        padding: 0;
      }
      #map {
        height: 100%;
      }
    </style>
  </head>
  <body>
    <div id="map"></div>
    <script>
      var map;
      function initMap() {
        map = new google.maps.Map(document.getElementById('map'), {
          center: {lat: -34.397, lng: 150.644},
          zoom: 8
        });
      }
    </script>
    <script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap"
    async defer></script>
  </body>
</html>
```

The google map docs have a line-by-line explanation of what the code does, go ahead and read that over now.

A) Write below how you can specify the coordinate (latitude -80, longitude 22) as an object literal.

B) Which zoom level will show …

- the World fully zoomed out:

- Landmass/Continent:

- City:

- Streets:

- Buildings:

Part II. The semester is almost over and I need caffeine

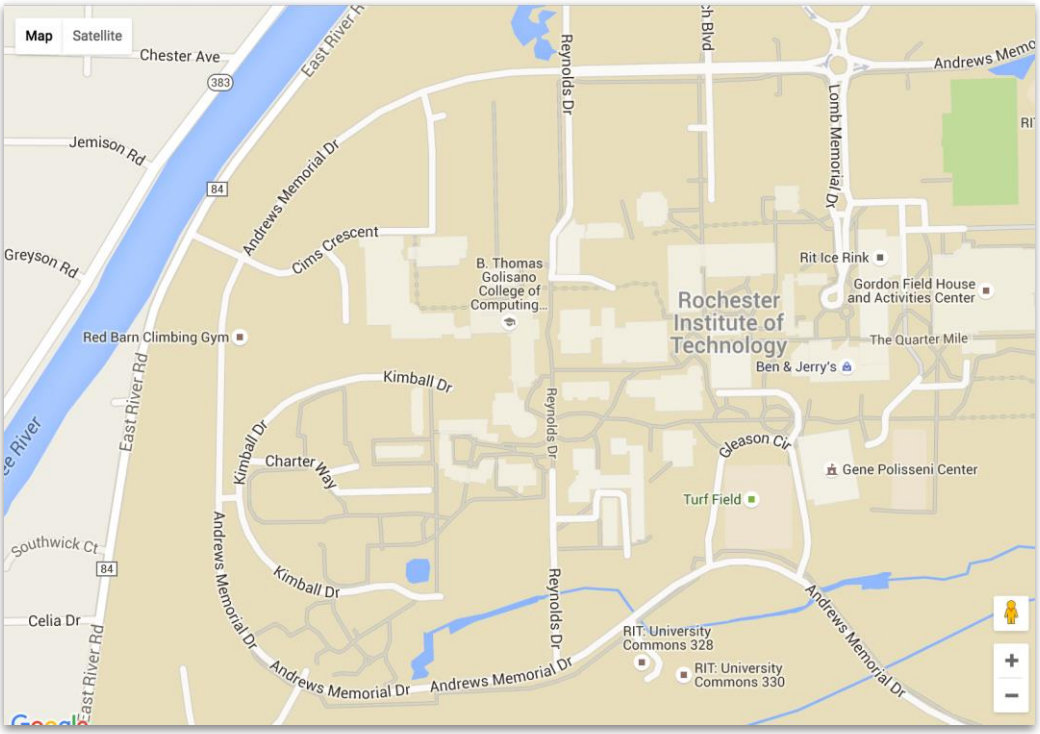
1) Duplicate **map-simple.html** and name the copy **rit-coffee-map.html**

Change the code as follows:

```
var map;
var infowindow;
var markers = [];
var polygons = [];
var baseUrl = 'http://localhost:3000';
var coffeeUrl = '/coffee.json';
var buildingUrl = '/building.json';

function initMap() {
  var mapOptions = {
    center: {lat:43.083848, lng:-77.6799},
    zoom: 16,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  };

  map = new google.maps.Map(document.getElementById('map'), mapOptions);
}
```

Note that we're changing the center property and zoom property of the map, and now specifying a mapTypeID.

Reload the page, it should now be centered and zoomed in on Golisano College.



2) We also could have set these properties directly though the map object. You can see the properties, methods, and events of the Map class here:

https://developers.google.com/maps/documentation/javascript/reference#Map

map.setCenter(), map.setZoom(), and map.setMapTypeId() are the methods you could use.

Let's add some simple controls to the map.

A) Add the following HTML before the <map> tag:

```
<p><button id="worldZoomButton">World Zoom (1)</button></p>
<p><button id="defaultZoomButton">Default Zoom (16)</button></p>
<p><button id="buildingZoomButton">Building Zoom (20)</button></p>
<p><button id="isometricZoomButton">Isometric View (18)</button></p>
<p><button id="updateCoffee">Update Coffee Shops</button></p>
<p><button id="updateBuildings">Update Buildings</button></p>
```

B) Add the following CSS:

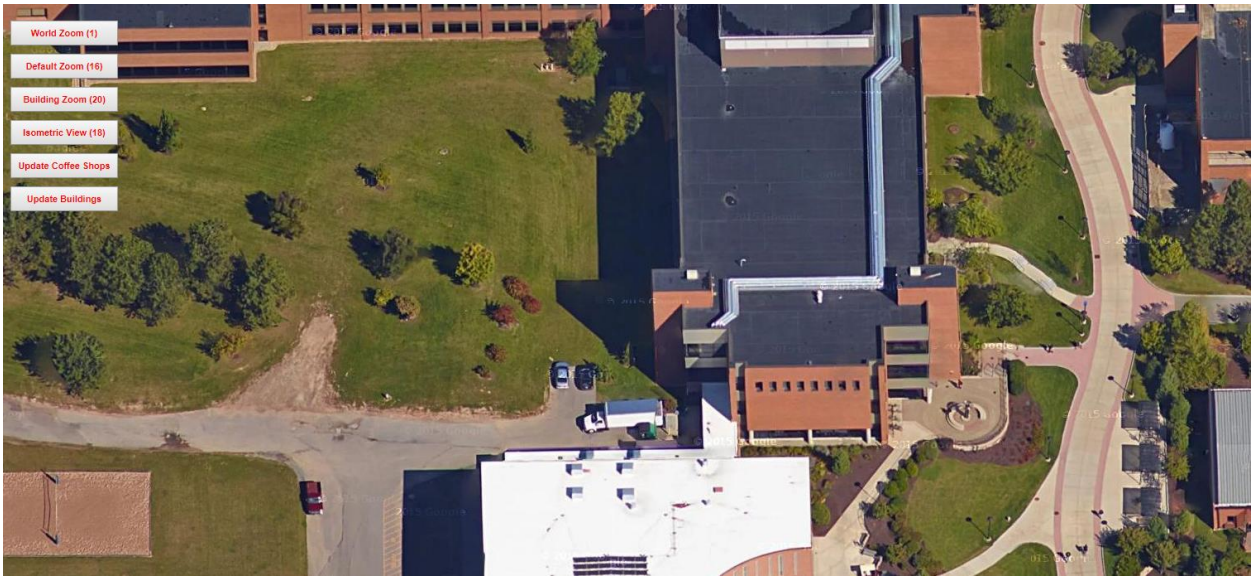
```
button{
  font-size: 11px;
  position: absolute;
  color:red;
  font-weight:bold;
  height:30px;
  width: 130px;
  z-index: 100;
}

#worldZoomButton{
  top:70px;
  left:10px;
}

#defaultZoomButton{
  top:110px;
  left:10px;
}
```

```
#buildingZoomButton{
  top:150px;
  left:10px;
}

 #isometricZoomButton{
  top:190px;
  left:10px;
}

#updateCoffee {
  top:230px;
  left:10px;
}

#updateBuildings {
  top:270px;
  left:10px;
}
```

C) In the initMap function, hook up three of the buttons for zooming.

```
document.querySelector("#worldZoomButton").onclick = function(){
  map.setZoom(1);
};

document.querySelector("#defaultZoomButton").onclick = function(){
  map.setZoom(16);
};

 document.querySelector("#buildingZoomButton").onclick = function(){
  map.setZoom(20);
};
```

The first three zoom buttons should now be functional.



Here we have added standard HTML DOM elements as controls. Google also has their own set of controls that mesh nicely with the maps (scroll down to the controls section of this page for examples):

https://developers.google.com/maps/documentation/javascript/examples/

By the way:
Latitude is 0 at the equator, and gets larger as it moves north.
Longitude is 0 at Greenwich, UK (near London), and gets smaller moving west.

Greenwich's coordinates are 51.4800 N latitude, 0.0000 longitude. Longitude is calculated from this Greenwich Meridian.

4) Let's add a Marker to the map. A Marker is a kind of overlay on the map - you can think of it as a pin that identifies a location on the map. Here are the docs:

https://developers.google.com/maps/documentation/javascript/reference#Marker

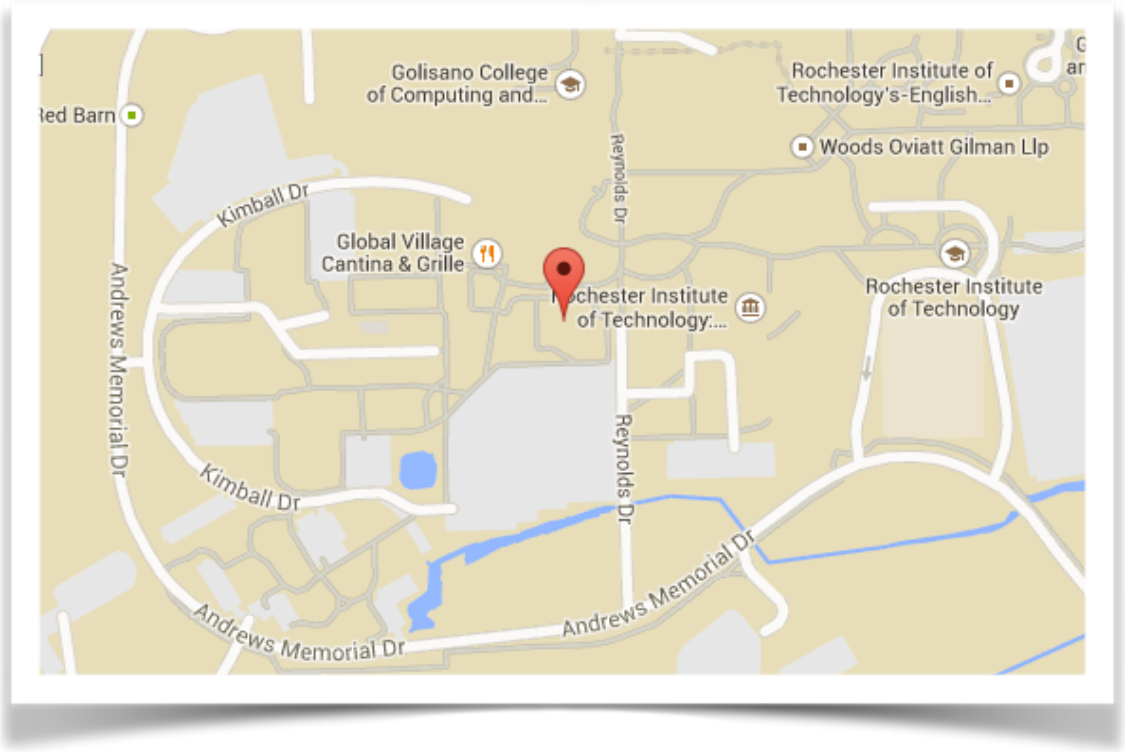
The tutorial also covers Markers very well:

https://developers.google.com/maps/documentation/javascript/markers

A) To put a marker on our map at Crossroads, add the following code to initMap()

```
var position = {lat:43.082634, lng: -77.68004};
var marker = new google.maps.Marker({position: position, map: map});
marker.setTitle("Crossroads");
```

Reload the page, you should see the following:



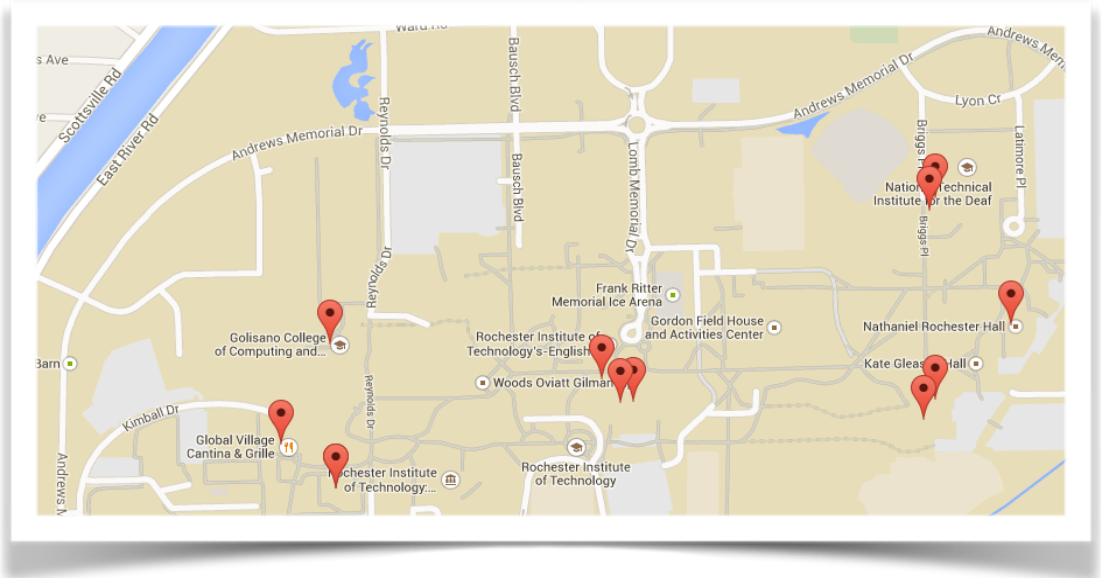
B) Let's go ahead and load in 12 coffee locations on campus. You'll need to import the **coffee-data.js** file (use a <script> tag) This file contains a hard-coded array of objects named coffeeShops. There are 12 objects in the array, each of which represents the location and name of a coffee shop on campus. The objects look like this:

```
{
    latitude:43.084156,
    longitude:-77.67514,
    title:"Artesano Bakery & Cafe"
},

{
    latitude:43.083866,
    longitude:-77.66901,
    title:"Beanz"
},
…
```

C) Delete the previous Marker code, and write the following helper function for creating markers:

```
function addMarker(latitude,longitude,title) {
  var position = {lat:latitude,lng:longitude};
  var marker = new google.maps.Marker({position: position, map:map});
  marker.setTitle(title);

  markers.push(marker);
}
```

D) Now write code in initMap() that loops through coffeeShops and calls addMarker() using the values from the array. When you are done, you should see all 12 coffee locations.



An InfoWindow is bubble-like overlay that we can put text into. It's usually attached to a Marker. The InfoWindow docs are here:
https://developers.google.com/maps/documentation/javascript/reference#InfoWindow

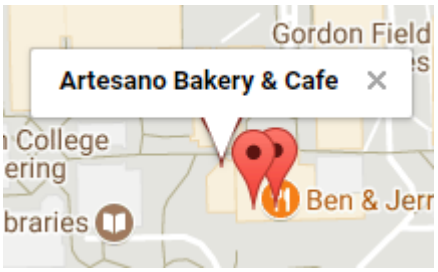
E) Then we'll create another helper function named makeInfoWindow():

```
function makeInfoWindow(position,msg){
  // Close old InfoWindow if it exists
  if(infowindow) infowindow.close();

  // Make a new InfoWindow
  infowindow = new google.maps.InfoWindow({
    map: map,
    position: position,
    content: "<b>" + msg + "</b>"
  });
}
```

F) To add a click event handler to each marker as it is created, add the following to the end of addMarker():

```
marker.setTitle(title);

markers.push(marker);

// Add a listener for the click event
google.maps.event.addListener(marker, 'click', function(e){
  makeInfoWindow(this.position,this.title);
});
```

Reload the page, clicking on a marker should cause a small window to appear that contains the name of the coffee shop.



G) To get 45-degree maps, add the following to initMap():

```
map.mapTypeId = 'satellite';
map.setTilt(45);
```

Not all locations and zoom levels have 45-degree imagery.

H) Now go ahead and add code in initMap to hook up the “Isometric View” to zoom level 18. Zoom level 18 will have a 3D effect.

Part III. Node Server

Now let's hook it up so that we can pull data from a server. This means we could update the coffee data on the fly or search for specific data.

We will build a node server to hold our data and handle requests from our clients.

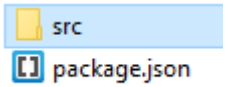
1) If you are on the lab machines, skip to step 2. If you are on your own machine, you will need to install node/npm

   https://nodejs.org/en/ (get current, not LTS).

2) In the folder with your starter files (the client folder), create a new node project using "npm init". If you don't remember how to do this, refer back to *Setting Up a Node Project* on mycourses.

   For a reference for the package.json options, refer to this interactive tool (hover over areas of the package.json on the page to get an explanation of what it does). Not all fields are necessary.
   http://browsenpm.org/package.json

3) In the same folder as your package.json (generated by the previous step), create a folder called src.



4) Inside of the src folder create a file called server.js.

5) Now modify your package.json to the following. Be careful of the syntax.

   Make sure you get the "main" and "script" sections correct.

```
{
  "name": "node_gmap",
  "version": "1.0.0",
  "description": "A very simple server to serve google map data for RIT",
  "main": "server.js",
  "scripts": {
    "start": "node ./src/server.js"
  },
  "author": "Cody Van De Mark",
  "license": "ISC"
}
```

6) Inside of your server.js, add the following.

```
var http = require('http');

var port = process.env.PORT || process.env.NODE_PORT || 3000;
```

So what does this do?

The *require* keyword is used to import modules/files. This is very similar to “using” in C# or “import” in Java & Processing. These modules are either built into Node.js, are installed through NPM (into the node_modules folder) or are files/modules you create.

In the first line we require the http module. This is one of Node’s built-in modules for servers. The HTTP module contains the functionality to create and operate HTTP servers. Similarly, Node has models for UDP, TCP, Websockets and many more depending on what you want to make.

When you give *require* a module name it first checks to see if it’s built into Node. If not, it then checks the node_modules folder to see if it’s installed. Finally, it will check to see if that module is installed on the system globally (uncommon). If any of those are true, Node will pull that module into scope. Node allows you to then scope that import into a variable (which is the most common way you will see it).

What happens when I require a module?

That module is instanced. The module may have support for making new objects, but the module itself is instanced. That means if I import the same module in various files, the functions of that module only get loaded into memory once. This is more performant and prevents circular dependencies. That means file A can require file B, and file B can require file A, without an infinite loop occurring.

What about the port line?

The port is just a number of a specific endpoint on a computer.
https://en.wikipedia.org/wiki/Port_(computer_networking)

There are 65535 ports on a computer. The lower numbers are reserved for specific protocols (21 for FTP, 22 for SSH, 80 for HTTP, 443 for HTTPS, etc).
https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

We want to avoid a reserved port because other applications will probably be listening for network traffic on that port. This would cause some problems with our traffic.

Instead we will use port 3000 (a port for common server development).

What does the port line do?

We are creating a constant for which port our server will listen on. If process.env.PORT is undefined/null, then we set it to 3000.

What is process.env?

The *process* variable is a global in Node describing the running process itself. The *env* variable inside of that is the environment variables. Many servers will set their own environment variable from a config file or from another process. For that reason, we check process.env before setting a manual port. This is what server hosts like Heroku do in order to provide each application a different port on a given server.

7) Now let us start the server and listen for HTTP traffic. Add the following.

```
var onRequest = function (request, response) {
  console.log(request.url);
};

http.createServer(onRequest).listen(port);

console.log("Listening on localhost:" + port);
```

The onRequest function will be the function invoked by the HTTP server every time a new HTTP request comes in. The HTTP module will automatically pass the request and response objects to the function. The request and response objects contain all of the information about the client's request and the server's response. We can then edit the response before sending it.

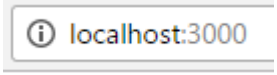
The createServer method of the HTTP module will create a new HTTP server taking in the function we specified. It will then start accepting traffic on the port we specified.

8) Save this file and start your server with *npm start*. Type *npm start* into the terminal you used to create the package.json earlier (double check that it's the right folder). If you need to open this window again, on Windows hold shift, right click in the folder and choose *"open powershell window here".*

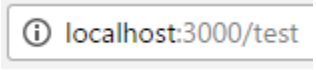
   You should see our console.log fire.

```
Listening on localhost: 3000
```

9) Now open a browser and test it out. Go to localhost:3000. **The page will NOT load because we have told the server to respond yet.**



Try localhost:3000/test (again the page will NOT load).



Now check your Node terminal. You will probably see this. That's because any time a request comes into the server, we are printing request.url (which is the URL after domain:port on the URL bar).

```
Listening on localhost: 3000
/
/favicon.ico
/test
```

When you went to localhost:3000, then the URL is automatically made / . When you went to localhost:3000/test, the URL was /test which you will see in your terminal.

What about favicon.ico or favicon.png?

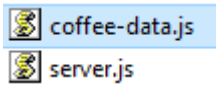
Browsers viciously want their favicon (the little icon that goes onto the page tab). This is standardized to be the URL /favicon.ico or /favicon.png (It's actually a bit more complicated than that).

With most browsers, if the browser does not receive a favicon, it will keep trying on each request (which is why you may see it popped up for localhost:3000 and for /test).

For now, we will ignore the favicon request.

10) Shut down your server for now (you can do this by hitting ctrl + C in the terminal/command window).

11) Now let's add our coffee data to the server. Copy the coffee-data.js file into the src file (with the server.js).



12) Open the coffee-data file and add the following line of code to the bottom.

```
{
    latitude:43.08359,
    longitude:-77.66921,
    title:"Gracie's"
}


];

//add this line to your code
module.exports.coffeeShops = coffeeShops;
```

What does this do?

In Node, each class has private variables/functions by default. We need to mark which ones are public. We do this by creating a module object. Everything in this object is marked public. This is actually using the module pattern (which we learned about earlier in the semester).

Node automatically creates this object for you by default (empty object) and puts it into this module. The object is called exports.

Anything we want to be public, we just add to module.exports.

13) Now that coffeeShops is public, we need to import the file in our server code. In your server.js, add the coffee-data import as coffee.

```
var http = require('http');
var coffee = require('./coffee-data.js');
```

What does the ./ in front of filename do?

As we discussed, if you just put a module name in, Node looks it up in itself, node_modules or globally. However, if you put a ./ in the name and give it the path to a file, it will load that file instead.

The **./** indicates to the *require* command that it should load a file from a path instead of from the modules. The coffee variable will be the public exports from the coffee-data file.

14) Now we can actually send data back to the browser. We could code our server to send the html files, but for this assignment we will just have our html files be served on Banjo. That means our Banjo files will be reaching out to another server with Ajax.

As we know, Ajax cannot go across sites because of Cross-Origin-Resource-Sharing (CORS). Since we have our own server though, we can actually enable this functionality.

In your onRequest method, add the following.

```
var headers = {
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Methods': 'GET',
  'Access-Control-Allow-Headers': 'Content-Type',
  'Content-Type': 'application/json'
};

response.writeHead(200, headers);
```

The response object has all of Node's HTTP Response class's methods and attributes.
https://nodejs.org/api/http.html#http_class_http_serverresponse

The writeHead function allows you to write a status code (200, 404, 401, etc) and a JSON object of the headers to send back. Headers are values that are standardized by a string name that the browser will interpret. Here is a list of standardized response headers.
https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Response_fields

We are setting the Access-Control headers to allow any origin (any web page on any server). We are allowing GET requests to our server and we are allowing the Content-Type header so we can accept requests by type.

The Content-Type header tells the browser what type of data we are sending. In this case, we are sending JSON.

Why application/json? [Media types]

'application/json is a Media type. These are also standardized so that applications can decode data differently. You can find a list of the standardized types here (categories into types starting with application, audio, image, etc). http://www.iana.org/assignments/media-types/media-types.xhtml

json is an application type so it is listed under application. You just look up the media type for whichever media you want to send back.

Please note: Browsers do not support all of the media types listed. This list is just a comprehensive list of the standardized media types for internet software.

15) Next we'll check the request url to see if the browser is requesting coffee.json or something else. If they are requesting coffee.json, then we will write the object to the response and send it (the end method). We need to stringify the coffee object in order to transmit it.

Otherwise we'll send back an error and set a 404 error code.

```
if(request.url === '/coffee.json') {
  response.write(JSON.stringify(coffee));
  response.end();
} else {
  response.writeHead(404, headers);
  response.write(JSON.stringify({'error': 'URL not recognized'}));
  response.end();
}
```

16) Our server should be functional now, but we need to address the client.

Remove the coffee-data.js and building-data.js scripts from the client. We will now get that data from the server.

17) Next remove the loop of coffeeData from your initMap function. Instead we'll get data from the server.

18) Add this function to your client script. This will remove and reset all the markers we placed on the map, allowing us to add new ones.

```
function clearMarkers() {
  markers.forEach(function(marker) {
    marker.setMap(null);
  });

  markers = [];
}
```

19) Now let's make a function to clear the markers from the map and get new ones. This will grab our coffee data from the server as 'data'. Then we can loop through the data as we did before.

```
function getCoffeeData () {
  clearMarkers();

  $("#content").fadeOut(1000);
    $.ajax({
      dataType: "json",
      url: baseUrl + coffeeUrl,
      data: null,
      success: function(data) {
        data.coffeeShops.forEach(function(obj){
          addMarker(obj.latitude, obj.longitude, obj.title);
        });
      },
      error: function(error) {
        console.dir(error);
      }
  });
}
```

20) In initMap, hook up the update coffee button to call your new getCoffeeData function when it's clicked.

21) Now start your node server and test your html in the browser.

When you click the 'Update Coffee Shops' button, it should show markers on the map.

If there are any issues: check the chrome console and your server command/terminal window for errors.

Part III. Heroku – (15% Bonus)

If you would like to, you can upload your files to Heroku (see the pushing to Heroku document on mycourses).

***NOTE*: Once you make your Heroku URL, you'll need to modify the baseUrl variable in your client to go to that instead of localhost:3000.**

You will need to upload everything except the node_modules folder. Test your app on Heroku to make sure the pages work.

You may hand in your Heroku app URL in the submission.

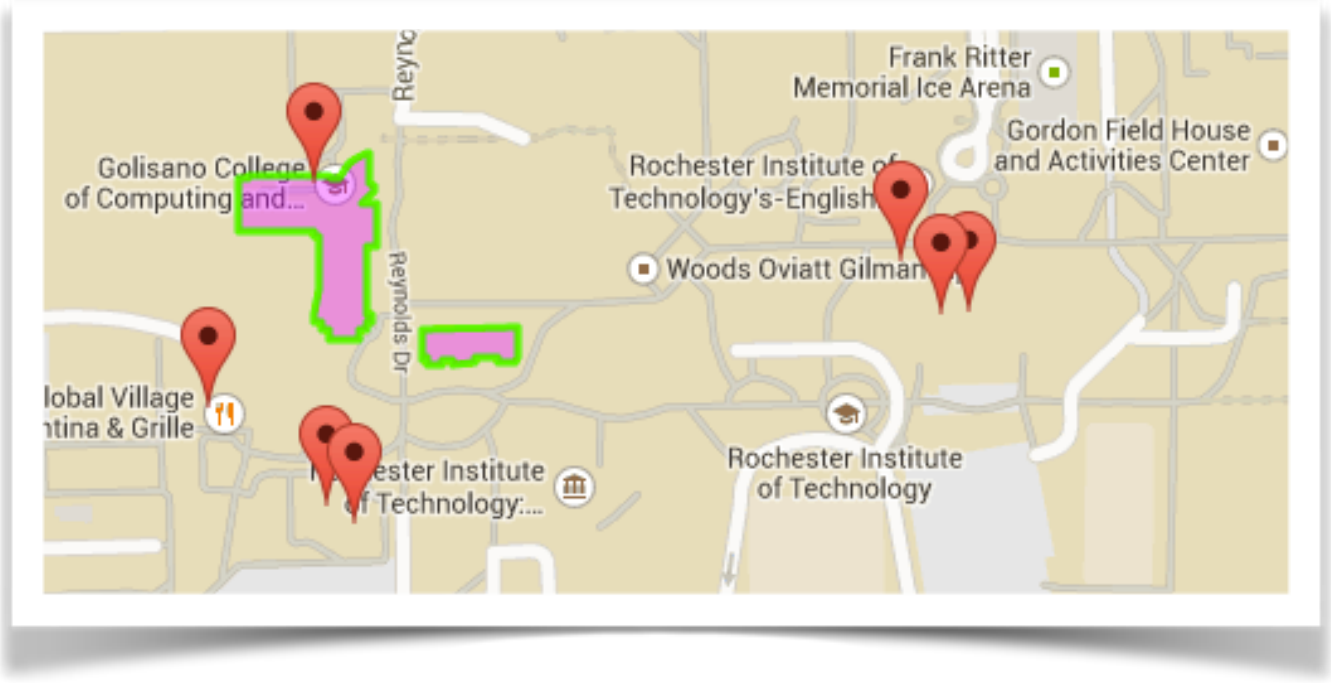
If you have problems with Heroku, please let us know. We can help

Part IV. Polygons – (10% Bonus)

For bonus points on this HW assignment, get Polygons drawing:
https://developers.google.com/maps/documentation/javascript/reference#Polygon

Polygons form a closed loop of coordinates and can be stroked and filled.

- load the **building-data.js** file in Node and export the buildings array
- have a /building.json url in the server to send the building data back
- write a helper function in the client called function drawPolygon(paths,title){…}
- write a function in the client called function getBuildingData () {…}
  that does an Ajax call to the server and loop through the buildings array to draw the building Polygons (there are only 2 in the file, GOL and ORN)
- hook up the 'update buildings' button to call the getBuildingData function



Rubric

DESCRIPTION	SCORE	VALUE %
Has the correct files – All of the files should be appropriately named and in the submission (including the client files, server files and package.json). These will be needed for us to run the code.		**10**
Server correctly sends back coffee json – The server should send back the coffee json correctly when the url /coffee.json is handled. Server does not throw errors and continues to run.		**15**
Server starts correctly without errors – The server starts up correctly without errors.		**10**
Map focuses on RIT – The map should automatically focus on RIT when the page is opened.		**10**
Zoom buttons work – All of the zoom buttons work correctly.		**15**
Pins on coffee shops – Coffee shops are correctly pinned with markers on the map.		**15**
Isometric View – Isometric view works correctly.		**10**
Info Windows on Click – When markers clicked, info window opens correctly.		**15**
Info Window Penalty – If info windows stay open instead of closing, there will be a small deduction.		**-10%**
Polygons Bonus – Polygons correctly displayed over GOL and ORN when the button is clicked. Data is provided correctly by the server.		**+10%**
Heroku Bonus – If your app is functional on Heroku, there will be a small bonus.		**+15%**
Additional Penalties – These are point deductions for run time errors, poorly written code or improper code. There are no set values for penalties. The more penalties you make; the more points you will lose.		
TOTAL		**100%**

Submission:

By the due date please submit a zip of your work to the dropbox complete with all of the files. In the zip include your package.json, src and client. If you did the bonus, please include a link to your working Heroku app (not the dashboard) in the submission comments.