SER421 Fall 2017 Online Lab5: AJAX
Assigned 11/10/17, due 11/20/17 at 11:59pm via online submission to Blackboard

Objectives:

1. Demonstrate proficiency in constructing a self-contained HTML5/Javascript application with AJAX
2. (Extra Credit) Demonstrate proficiency in client-side MVC using a modern framework.

This lab is to be completed individually. There is a video I will post to Blackboard that gives a demo run through of what you are to emulate.

Activity: Construct a Single-Page Application (SPA) with AJAX

openweathermap.org provides a Web API that returns JSON data (you used this API in your lab 3 as well). The format of the data is described here: http://openweathermap.org/api

While an example is given here: http://api.openweathermap.org/data/2.5/weather?q=London,uk

Write a complete web application that does the following:

1. Displays a list of 3 cities and their associated weather data. The data should be retrieved and parsed out of the JSON at URLs like the above via an AJAX call. The data you should display in a table:
   a. A city name and 2-letter country code. Examples: London,UK and Phoenix,US
   b. A timestamp when the data was last updated.
   c. Temperature in Celsius
   d. Humidity – a percentage. Example: 70 means "70% humidity"
   e. Wind speed – miles per hour
   f. Cloudiness – a percentage. Example: 10 means "10% cloudy"

   When initially loaded, the application should initialize the rows for 2 of the 3 cities, London, UK and Phoenix, US. The 3rd city may be any city and is described next.
2. The 3rd city should be populated by selecting from a set of 5 cities in a dropdown. You may populate the dropdown with any 5 cities you like. When a new city is selected, you should populate its data in the 3rd row.
3. Create a new row underneath each city row that shows how the values have changed from the previous call to the server. Provide a Refresh button that forces a fetch of the 3 cities data. That is, it should show how long ago the previous reading was, how much the temp went up or down, and so on for each data value. Constraint: this functionality should work even if you close the browser and reopen it! To be specific (watch the video too):
   a. The extra row per city that shows up should 1) retrieve fresh values from the API, and 2) compare those values to the values it should already have stored for that city from the last time it called the API.
   b. It may very well be that the values have not changed when you make the new call; in which case the 5 values you display are all 0, as nothing has changed. However, once the values do change, you must store the previous values, display the current values in the first city row, and display the delta of the values in the second city row.
   c. Keep in mind that not every API call will result in you changing the persistent store; only those API calls where the values change will result in you changing the persistence store. The initial time you load the app and use Refresh the deltas will be zero since you just initialized the persistent store the first time; subsequent calls will then proceed as described above.

   Here is a scenario (just one city, Phoenix):
      i. You open the app for the first time at 4pm and it retrieves the weather for Phoenix. The timestamp is 3:50pm, the temp is 20, humidity 5, wind 8, cloudiness 40
      ii. You hit refresh. Your app hits the API and there is no change in any of the values. Your new row is displayed as all zeros.
      iii. At 4:15 you hit Refresh again. The API has new values: timestamp of 4:10pm, temp of 15, humidity 12, wind 5, cloudiness 20. Your refresh row (the 2nd city row) should show:
      iv. minutes ago, -5, 7, -3, -20
      v. You close your browser and come back the next day at 11am and open the app. An API call results in new values: timestamp 10:00am (the next day), temp 12, humidity 0, wind 5, cloudiness 5. Your app should show:
      vi. hours 50 minutes ago, -2, -12, 0, -15
      vii. …and so on.

4. At the bottom of the page, display the following lines:
   a. “The average temperature is AAA and the hottest city is TTT”
   b. “The average humidity is BBB and the most humid city is HHH”
   c. “The city with the nicest weather is XXX”
   d. “The city with the worst weather is YYY”

   Lines c and d should be custom logic you create – my only constraint is that your computation involve all 4 of the values from 1c to 1f above.

Note the data in this Task is in this JSON format, but in some cases you may have to do a conversion (e.g. Kelvin to Celsius temperatures, wind speed in meters per second instead of MPH).

Error Handling:
Your application should handle error response codes from the API server in the most elegant way possible. Detect 4xx and 5xx messages and inform the user as to the specific error (code and message) and what s/he should do by dynamically inserting an error message into the DOM at the top of the page.

Extra Credit (40 points):

You may do only EXACTLY ONE of the extra credit options below as your **individual** work:

Option 1: Convert your weather app to a modern client-side MVC framework
Building on your Activity 1 solution, now have your initial and refresh values for the 3 cities use AJAX calls to openweathermap.org just as in Activity 1, but use a modern client MVC framework to accomplish this. Please convert the GUI application to a modern MVC framework such as Angular (what we are calling Angular4, AngularJS (Angular1), React, Vue, or Ember). Design will be part of the grade criteria.

Option 2: Convert your Clue game to a server-side game with AJAX on the client
Don't want to deal with the learning curve of a client-side framework? Then this option is for you. Take your SPA Clue game from your last lab, and port the game state to the server-side under a NodeJS server (you may use NodeHTTP or Node/Express, your choice. Have your SPA modify gameplay to:

1) Make an AJAX POST every time the user makes a guess. The server should respond with whether the guess is correct or not. NOTE: Make an AJAX call, not a FORM POST.
2) The user still hits "Continue" to toggle between human and computer guesses.
3) "Show History" will need to understand when a game is over to reset it's state, but that state may continue to persist locally.
4) "Show Record" should now store the computer's record on the server side, and when this button is clicked, an AJAX GET should be used to retrieve the Computer's current record.
5) The server should be able to distinguish between multiple users in multiple browsers playing against the Computer at the same time. This means when a new user logs in, effectively starting a new game, a new conversational state must be established to indicate a new game. It is OK to use either a FORM POST Action on login or an AJAX POST on login to accomplish this functionality.

These 5 requirements might sound like a lot, but realize most of what exists in the game on the client and be directly ported into Node (like shuffling, identifying the secret triplet of cards, and so on). You are basically taking your "world model" of the game and moving it to the server while keeping your UI model as an SPA.

Submission:

Submit via a zipfile named <asurite>_lab5.zip to Blackboard by the due date. Name your Activity 1 files <asurite>_lab5_act1.html and <asurite>_lab5_act1.js respectively. Bundle the Extra Credit Option 1 app up as a zipfile from the root directory and call it lab5_ec1.zip. If you choose to do option 2, then bundle all of its files up as a zipfile named lab5_ec2.zip. Then put all of the files into the top-level zipfile named <asurite>_lab5.zip. You may add a README.txt in the root directory of your submission if there is anything you want us to know when grading.