SER421 Fall 2018 **Lab5: HTTP, AJAX/Fetch, and APIs**

Due 11/19/18 at 11:59:00pm via online submission to Blackboard

Objectives:

For this lab you will develop two applications. One will be your own API based on a sample project you already have with Fetch, and the second will be an HTML5/Javascript application using AJAX against a real Web API.

Activity 1: Use Fetch with your own API to refactor your SPA (50 points)

For this activity you will port your SPA from Lab 3, Activity 1 (the calculator) to an API-driven application. Specifically, Lab 3 Activity 1 asked you to implement add, subtract, pop, and reset features. You will port these to an API in NodeJS and modify the SPA to consume that API to implement a shared calculator service and application.

R1. (3) Your application should load when the root URL (/) of your server is accessed.

R2. (20) Your application GUI should include:

a. The value of the calculator clearly displayed
b. The current set of operations in the stack (the stack is maintained on the server behind the API, see R3c)
c. A one-line textbox for entering a number for the calculator
d. Buttons for invoking Add and Subtract operations, disabled unless a valid number is entered into the textbox, at which point they should be enabled. These buttons should cause the invocation of Fetch calls to the server API to perform the respective operation. These are specified in Lab 3, Activity 1 R2a and R2b.
e. A button for a Pop operation (Lab 3, Activity 1 R2c) should be provided and should always be enabled. Again, this should cause the invocation of a Pop operation to the API.
f. A button for a Reset operation (Lab 3, Activity 1 R3) should be provided and should always be enabled. Again, this should cause the invocation of a Reset operation to the API.
g. Upon successful completion of API calls for d-f, you should display a new calculator value (R2a) and a new value of the stack as per R2b above. This will require a second API call.
h. The application should detect error messages and display the respective error code and message in the GUI.

R3. (20) Your NodeJS API server should:

a. Listen on port 8008
b. Provide endpoints /add, /subtract, /pop, and /reset. The response for each is derivable from Lab 3 Activity 1 requirements R2a-c and R3. The HTTP verbs supported are given in that lab's requirements. The response to each of these should be a proper status code and a JSON payload reflecting the new calculator value: '{ value : *num* }'.
c. Provide an endpoint /history. This endpoint (which is new) should have the ability to return as JSON the entire history of operations applied to the calculator (the stack). It should only support GET.
d. Lab 3 Activity 1 R4 is still in effect; push is implied but is not an exposed API-accessible behavior.
e. Error situations should be handled appropriately, meaning the proper error response code is returned, with a JSON payload providing error message details where appropriate.

R4. (7) You should provide API documentation (in HTML as a webpage) at /api. This may be a hand-crafted static webpage, though I will be more impressed if you auto-gen the API documentation from your code.

Activity 2: Use AJAX to access a real Web API

www.apixu.com provides a Web API that returns JSON weather data. You can sign up for free access for 5K API calls per month. Once you do you will get example URLs to the current weather and forecast endpoints with your own API key. You will want to save the endpoints and the API key. They also have a nice "Interactive" tab for experimenting with your own API calls.

Write a complete web application that does the following:

R1. (10) Displays 2 cities and their associated 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 (2 rows):

a. The first city should be "Phoenix"
b. A second city name custom to your application. (Paris, London, you pick)
c. For each row, have a column for the city name followed by:
    i. A local (to the city in question) timestamp when the data was last updated.
    ii. Temperature in Fahrenheit, plus "feels like" temperature.
    iii. Humidity – a percentage. Example: 70 means "70% humidity"
    iv. Barometric pressure in inches
    v. Conditions - text

R2. (10) Populate a 3rd city row by selecting from a set of 5 cities in a dropdown. You may populate the dropdown with any 5 cities you like. The website has a list of cities supported. When a new city is selected, you should automatically populate its data in a 3rd row.

R3. (10) At the top of the page, display the following line: "The average temperature is AAA and the hottest city is HHH"

R4. (10) Next to each city name put a "forecast" button that, when clicked, puts text below the table describing the weather forecast for tomorrow *for that city*. The display should include all the values under the “day” and the “astro” (see the JSON), but you can decide the output format (as long as it is clearly readable).

R5. (10) Provide a "Refresh" button at the bottom of the table that causes the data values *for the entire table* to be updated as well as recompute requirement #3 based on the new values. It should also clear the forecast area if there is any text in it.

Hints/Constraints:

- Activity 2 may appear somewhat daunting at first, but spend 15 minutes on the API website with their tools and by inspecting JSON in the browser and you will see it is not that difficult.
- You must use AJAX for this activity, not Fetch!
- Be sure to check for non-200 responses and have basic error-checking – your app should not crash or ruin the rendering of the display table if an API AJAX call is not successful; instead it should show an appropriate error message!
- As before, no CSS!

Extra Credit: (40 points) – NOTE you cannot get EC credit if you have not completed the respective Activity’s requirements!

EC1 (20 points): Add the following requirements to your Activity 1 solution:

1. Keep track of the operations the user in that browser has applied to the calculator and provide a mechanism (UI widget) to display them when requested
2. Provide an ability to retrieve all operations to the server-side stack done by another client since the last calculator operation in *this* client.

***EC2 (30 points)*:** Rewrite Activity 2 using your favorite client-side framework. React, AngularJS, Angular/Core, Vue.js or whatever you like. If you are not sure if the framework is allowed, ask!

Submission:

Submit via a zipfile to Blackboard by the due date. Name the file <asurite>_lab5.zip. It should have subdirectories for the 2 activities named activity1 and activity2. You may add a README.txt in the zipfile if there is anything you want us to take into consideration. If you do the extra credit please submit those in separate directories named EC1 and EC2 respectively (keep the distinct from the activity 1 and 2 solutions).