SER421: Web-based Programming SPA Lab

The goal of this lab is to get you working in a browser-based application, using various techniques such as DOM manipulation, event handling, and storage. Submission instructions are at the end, PLEASE FOLLOW SUBMISSION INSTRUCTIONS!

Activity 1: DOM expressions (20%)

Go to duckduckgo.com and perform one search using 3 distinct words of your choosing. Save the resulting page (save the complete page, See Submission Instructions #1 at the bottom).).

Now, for this activity, write DOM expressions that do the following: (5% each)

1. Rename the message "Help Spread DuckDuckGo" in the upper right to say "Help Spread Love!"
2. Output to the console the number of listitem <li> tags in the page
3. Output to the console the value in the search bar (you must get this from the search bar not anywhere else on the page)
4. Make the duck in the upper left corner go away

Activity 2: Implement your own Eliza ON THE CLIENT! (40%)

Since the early days of computing, humans wondered if computers could be made intelligent. In the AI field of natural language processing, the argument was presented that if a computer could communicate like a human, then it possessed human intelligence. To demonstrate this approach, a program named ELIZA was created to emulate a computer talking to a person. (https://en.wikipedia.org/wiki/ELIZA, http://psych.fullerton.edu/mbirnbaum/psych101/Eliza.htm). Eliza was ChatGPT before ChatGPT was! The basic approach behind Eliza is to focus on keywords or substrings a user presents to Eliza, and use it to determine a pseudo-intelligent response from a dictionary of responses. You will use a dictionary structure provided to you on the Canvas site.

Requirements:

Functionally your program must:

R1. (5) Greet the user by asking for her/his name on startup in a simple form. The name should be "remembered" and used anywhere a direct naming of the end user is appropriate. Eliza should then start the conversation with a question. The question is not fixed, but should start with common greetings like "<name> how is your day going?" "<name>, is something troubling you?" or "<name> you seem happy, why is that?"

R2. (12) Provide a one-line web form that allows the user to "talk" to Eliza. You should parse the string the user types in and search for matches to the *key* property of each entry in the dictionary, and on match randomly select one of the *answer* entries and one of the *question* entries for a response. Echo the user's input and Eliza's response above the web form (that is, the one-line web form input should always be at the bottom of the page). *For echoing the input and Eliza's response, you MUST manipulate the DOM of an HTML non-form element – meaning do not use something like a textbox or a textarea!*

R3. (7) You should vary the responses to the same keywords (although the number of responses will of course be finite) by introducing some simple randomization so no 2 sessions follow the same pattern of responses. For randomization, you can use the basic Math.random() built into Javascript.

R4. (6) If no key is matched in R3, then Eliza should respond with a catch-all answer-question pair (think of it as a "default" case of a switch statement). You may choose the answer-question pairings, but they should be done in the same randomized way as R3 and there should be at least 3 pairings

R5. (10) If the user does not respond to an Eliza question within 10 seconds, Eliza should display a dialog box with a message such as "<name>, I'm waiting here!" or "Whatsa matter <name>, cat got your tongue?" or so forth (have some fun with it, but it must include <name>, where <name> is the end user's name). Again, the prompt should not always be the same. To implement this feature, look up the window.setTimeout API in the browser.

Create your own Eliza under the following constraints:

C1. Your application has NO server component whatsoever (*see Special Note end of the document)*.

C2. Your application has NO CSS. I want to be clear that this also includes manipulating any style attributes on a DOM element. The intent is for you to change the structure of the DOM, not hide/unhide elements.

C3. Your application must be a "single page application" – that is, it never reloads a page from a local source (window.location.href) or does a document.write() to simulate a page refresh. The page is in effect your desktop application GUI.

C4. Eliza should display only the previous user response, Eliza's response, and the next question Eliza asks, followed by the prompt for the next end user input. Any prior user/Eliza responses and questions should disappear.

C5. Keep track of the user's name by adding it directly to the DOM somewhere (you design where and how). Do not use session or local storage or cookies for keeping track of the user's name.

C6. Form-based textual input has to be handled on a button event through Javascript in the browser.

Activity 3: Make Eliza Stateful (40%)

Eliza in Activity 2 remembers a user's name but does not remember the conversation. In Activity 3, you should add stateful behavior – namely, Eliza should "remember" previous answers the user has presented and also applies some randomness to avoid deterministic and/or repeated responses. New requirements:

R1. (5) Make it so Eliza displays a running dialogue of the entire conversation. This modifies C4 from Activity 2.

R2. (15) Make your Eliza program stateful by saving the responses it gives to <name>. If the browser closes and restarts, and you come back to Eliza, and enter the same <name> as a prior respondent, then you should be able to restore to the prior conversation.

Note that this is a *per user x browser* requirement, not just a browser requirement, which should hint to the storage mechanism. "Restore" here means you redisplay the older conversation as per R1, and internally reset the conversation to the old state.

R3. (8) Add a special "/clear" operation so that it clears all state of the application for <name>, then returns the app to the start form (how the application shows when first loaded).

R4. (12) Activity 2, R3 asks you to randomize responses based on a keyword. Extend this functionality so that not only is it random, but ensure no response is repeated until all responses are given at least once. (Keep in mind that a "response" is comprised of both an answer and a question, so both have to be tracked independently).

NOTES:

1. You are expected to implement Activity 3 requirements using session or local storage. Using the right storage and managing it properly is considered part of the grading rubric for proper design.

Submission Instructions:

Submit your lab as a single zipfile named <asurite>_ser421labDOM.zip with the following structure:

1. In a subdirectory Activity1, save the complete web page after doing your search but BEFORE running your activity1.js (there will be a subdirectory that is saved with it. Next, do a screen capture showing your original search query with the result page in DuckDuckGo and name it activity1.[jpg|gif|png]. If you need multiple screenshots to get it all just name them 1a, 1b, etc. Have a javascript file named activity1.js that has the sequence of expressions used to answer activity 1. Put comments before each block of code to label it with each step of activity 1. Please screen capture your results of your 4 expressions in the browser console and name it activity1results. .[jpg|gif|png].
2. Your activity 2 will be a combination of html and JavaScript. Save your answer to activity 2 in a directory named Activity2 and name the html file that is the starting point of the SPA as activity2.html. If you factor the JS code out of the html and include it using a link, then name it activity2.js.
3. Your activity 3 will be a combination of html and JavaScript. Save your answer to activity 3 in a directory named Activity3 and name the html file that is the starting point of the SPA as activity3.html. If you factor the JS code out of the html and include it using a link, then name it activity3.js.
4. As always if there is anything you want us to know, put a README.txt|md in the directory.
5. I allow unlimited submissions so there is no reason to be late! Late submissions will not be accepted!

Special Note: *Some folks in the past have reported issues running SPAs from within Google Chrome with no web server. This may require you to a) turn off a security setting (which one has shifted around in versions of Chrome), b) load the initial page (activity2.html or activity3.html) from a locally running server, (again, you are coding so you don't *need* a server, I am suggesting this as a workaround to a security setting, as Chrome will allow your code if served from an http server), or c) run it from Firefox like I do! Do not use a server for any other reason (no module imports or anything like that).*