QA Technical Assessment – November 2024

Just to document steps taken as I progressed with this assessment:

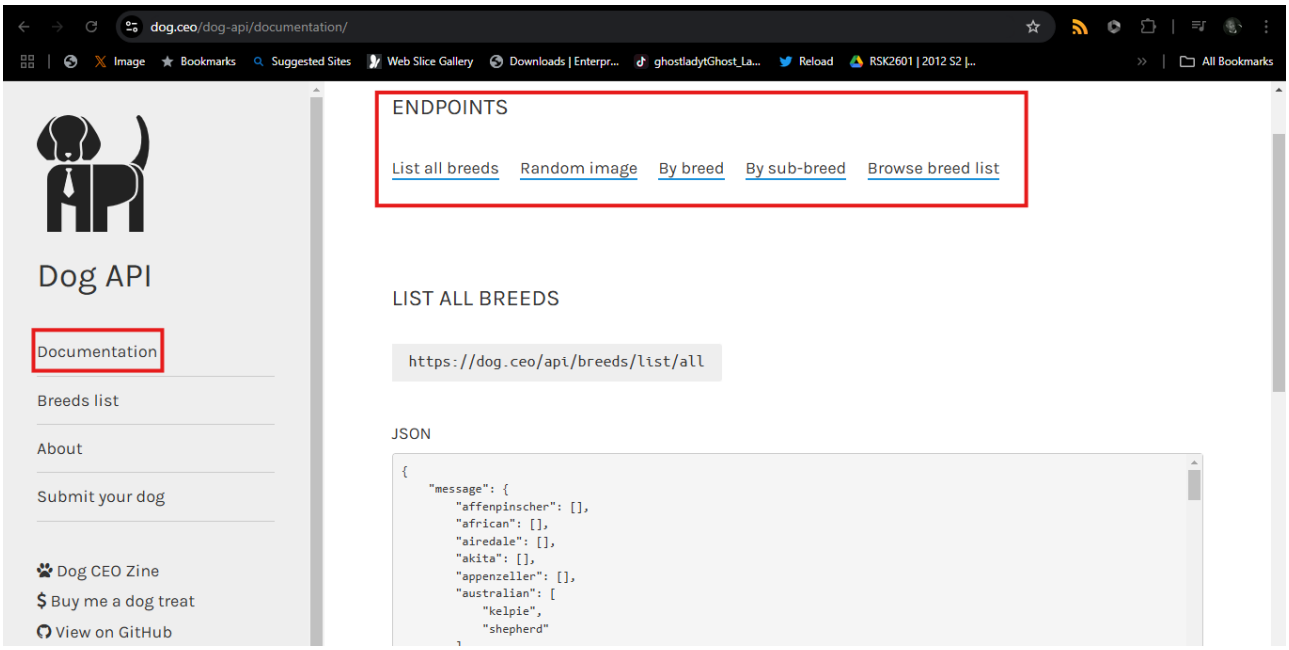
- Read to understand all listed tasks to test cases at hand, used Google for clarity:
  - Hybrid approach with modularization
  - Descripting programming
  - Regular expressions
  - Parameterization
  - 2 ways of storing and utilizing test data
  - Report stores test evidence and results

- Open Source automation frameworks used:
  - Postman for Task 1 API testing
  - Web testing Task 2

- Brushing up on different API request types understanding - Google

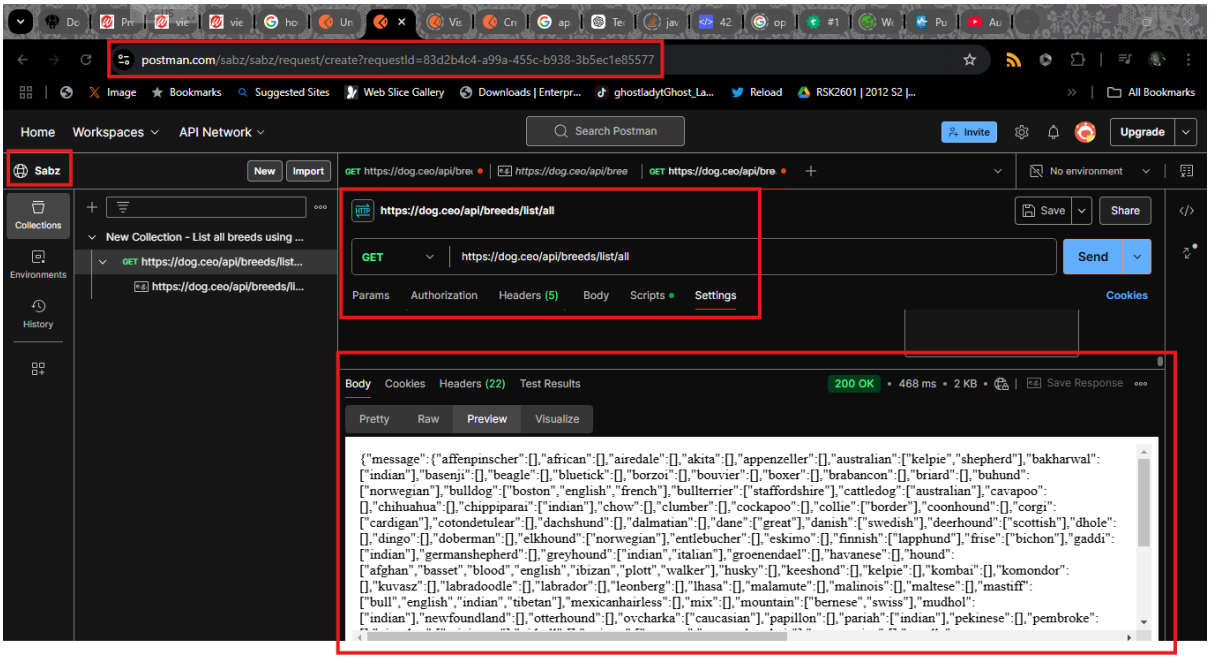
Task 1 – API:

- Its important to firstly note ***all API end points*** utilized by this public site which may be used under the heading ***"Documentation"***:

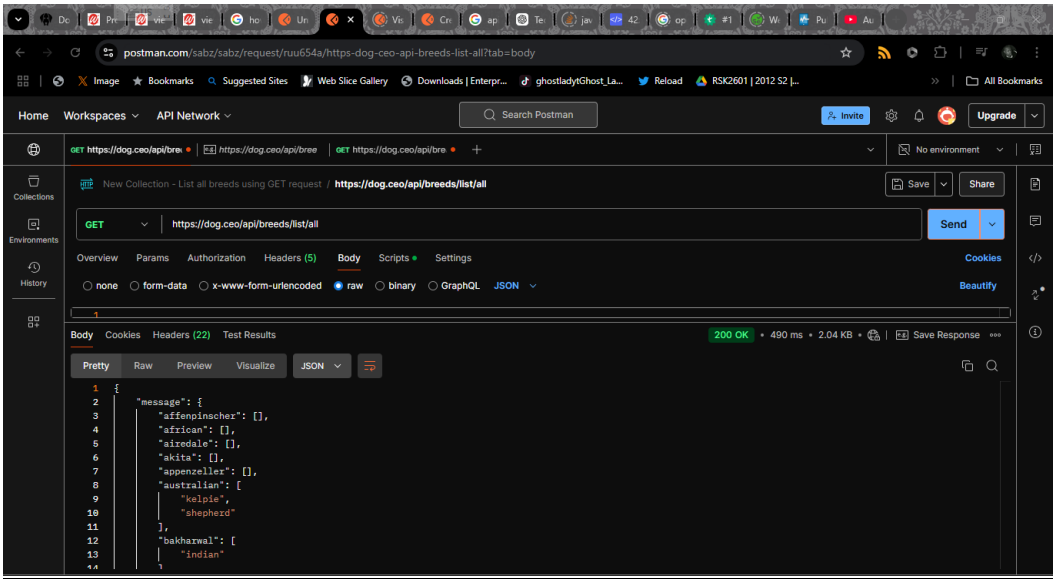


- Performing an API request to produce a list of all dog breeds similar to the output of the assessment document **Diagram 1.**
  - As mentioned above I used my preferred open source API web tool being Postman API Network.
  - Created an open public workspace which im hoping anyone can access and called it Sabz where all my work under API can be found.
  - For this first task, I used the end point https://dog.ceo/api/breeds/list/all which would bring back a list of all dog breeds as requested.
  - Wasn't happy with the display of the JSON data returned with my initial first run of this GET request of all dog breeds as it displayed as follows, which didn't look the same as the supplied "Diagram 1" of the assessment document.
  - Went hunting for a solution online to plug in Java Script code which would reformat the JSON data adding carriage returns making it more readable (and nicer)

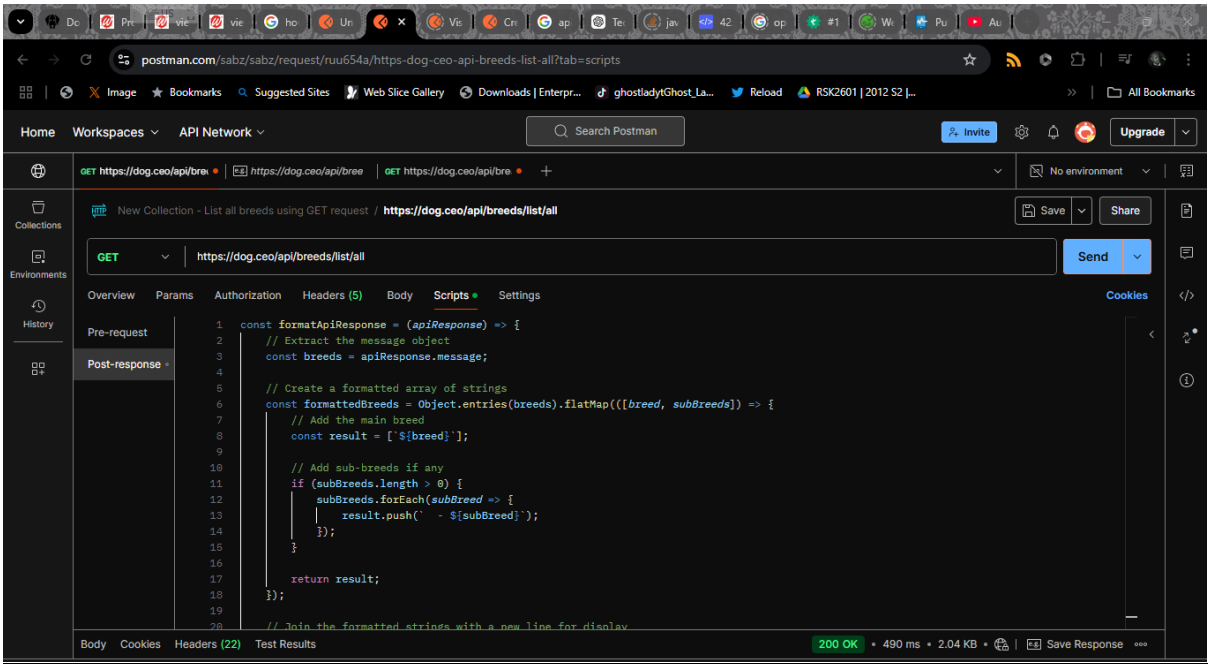
Before adding Java Script to format returned JSON data



After adding Java Script to format returned JSON data



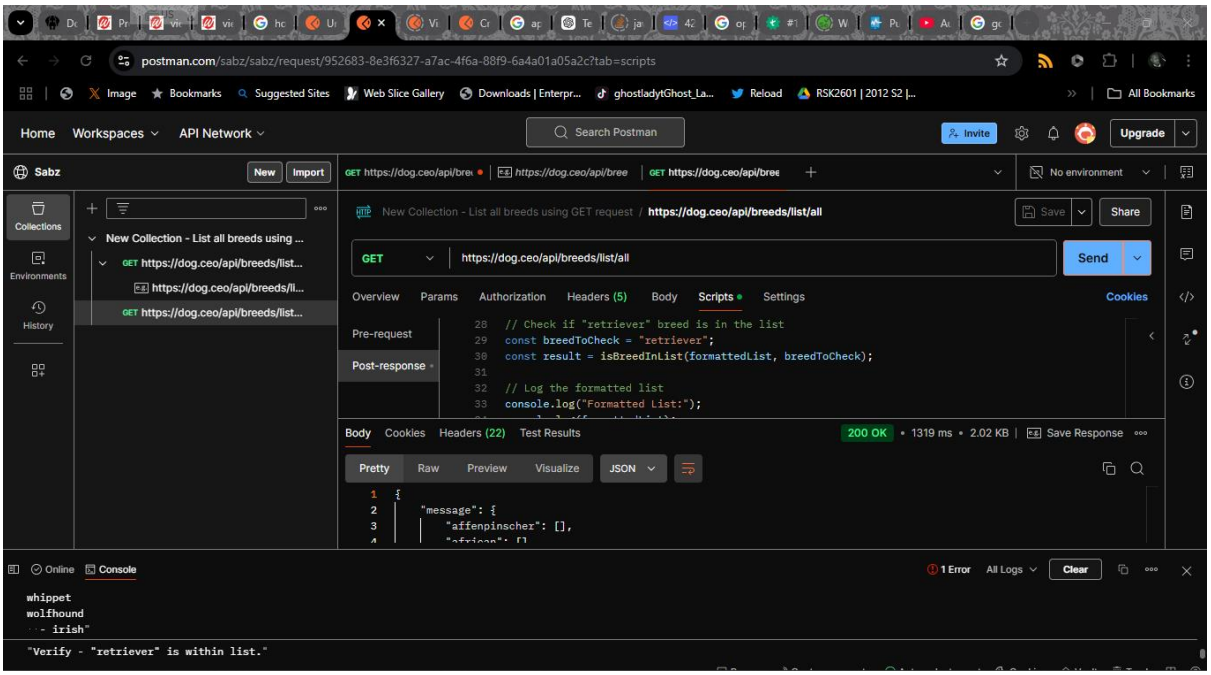
Java script added to format output data



After having gone around on a wild goose chase from being sent from StackOverFlow to other coding forums, finally had to fold and resort to the very platform I criticize programmers of being lazy by turning to it [OpenAI ChatGPT] 😟

```
const formatApiResponse = (apiResponse) => {
  // Extract the message object
  const breeds = apiResponse.message;

  // Create a formatted array of strings
  const formattedBreeds = Object.entries(breeds).flatMap(([breed, subBreeds]) => {
    // Add the main breed
    const result = [`${breed}`];

    // Add sub-breeds if any
    if (subBreeds.length > 0) {
      subBreeds.forEach(subBreed => {
        result.push(` - ${subBreed}`);
      });
    }
    return result;
  });

  // Join the formatted strings with a new line for display
  return formattedBreeds.join('\n');
};

// Call the function and log the result
console.log(formatApiResponse(apiResponse));
```

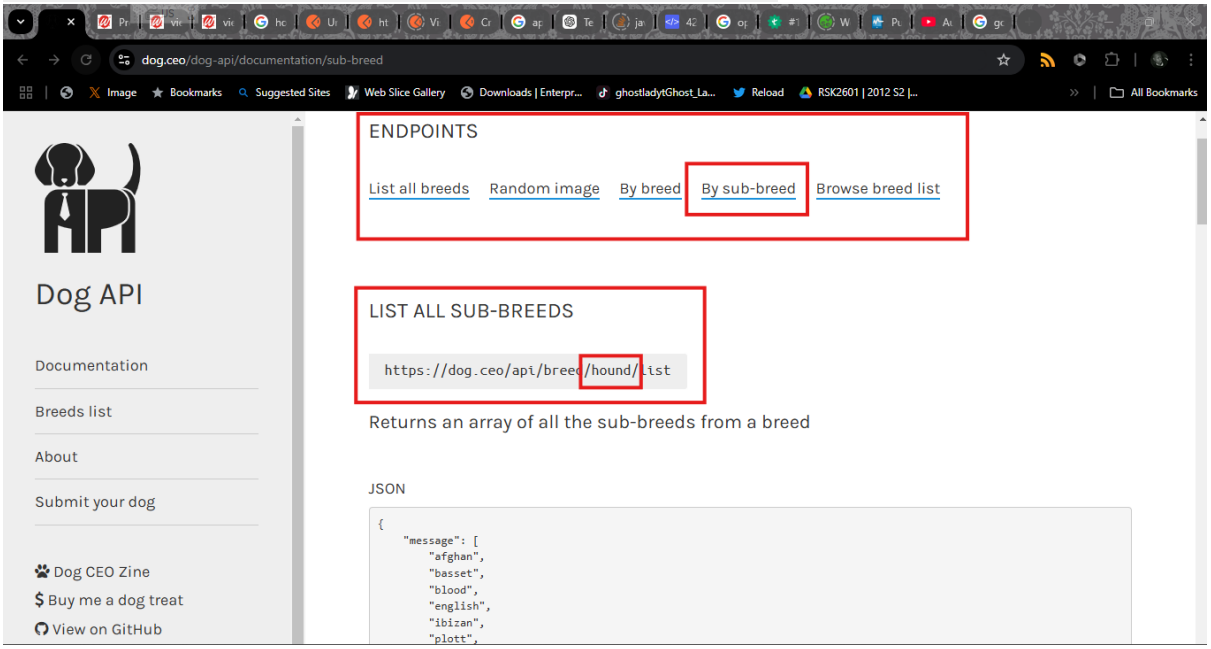
- Using code, verify “retriever” breed is within the above list response like the output found on assessment document **Diagram 2**.
  - Having been helped by OpenAI with the above formatting sample code, it was easier to manipulate it on my own, even though it took me too many tried which led to to again consult the web coding forums ending me on the very same page which helped me in the above task. [OpenAI]
  - This exercise only proved how rusty my Java script coding skills are, with me having not worked on these since pre-covid, frustrating but fun to jump back onto again.
  - Another thing to consider was to combine both scripts, first one formats JSON data putting text items one underneath the other, then the second script would take the same formatted text and check for breed “retriever”
  - The following was my finilised combined script which failed horribly because of scoping { }; needed to ensure certain variables and items were placed correctly within scope of code so it executed correctly and in the right order / sequence
  -



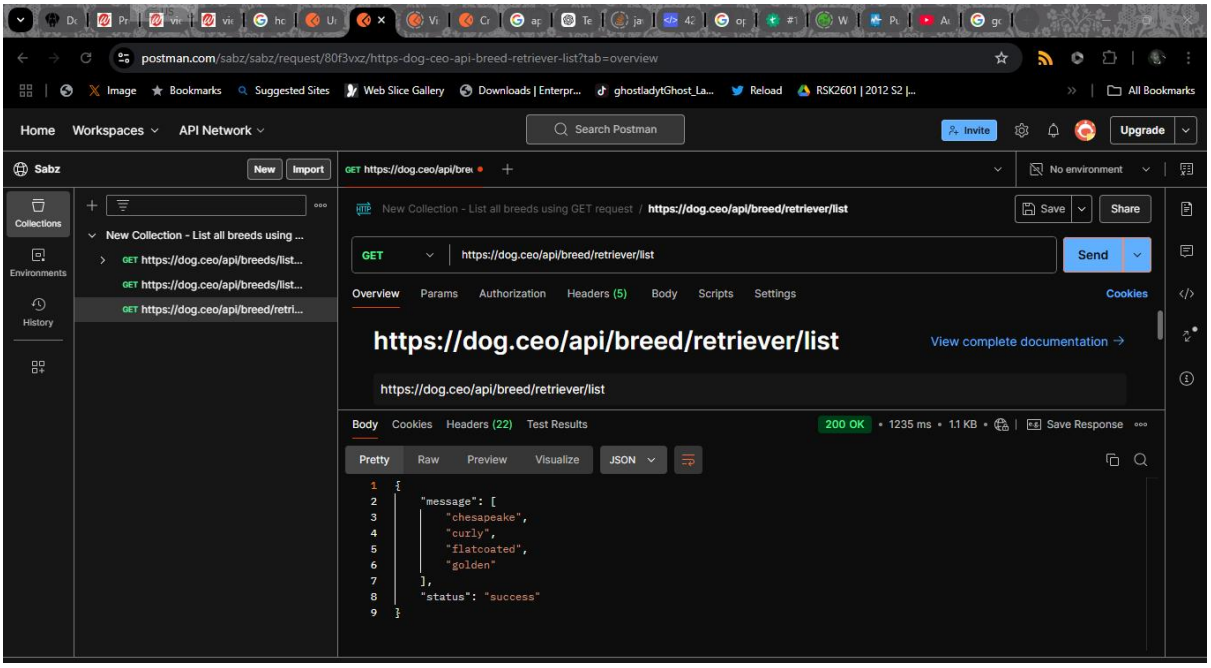

```
// Function to format the API response
const formatApiResponse = (apiResponse) => {
  const breeds = apiResponse.message;

  return Object.entries(breeds)
    .flatMap(([breed, subBreeds]) => {
      const result = [`${breed}`]; // Add main breed
      if (subBreeds.length > 0) {
        subBreeds.forEach(subBreed => result.push(`  - ${subBreed}`)); // Add sub-breeds
      }
      return result;
    })
    .join('\n'); // Combine into a single string with new lines
```

```
};

// Function to check if a breed is in the formatted list
const isBreedInList = (formattedList, breed) => {
  const lines = formattedList.split('\n'); // Split into lines
  return lines.includes(breed); // Check if the breed exists
};

// Extract API response from Postman
const apiResponse = pm.response.json(); // Parse JSON response

// Format the API response
const formattedList = formatApiResponse(apiResponse);

// Check if "retriever" breed is in the list
const breedToCheck = "retriever";
const result = isBreedInList(formattedList, breedToCheck);

// Log the formatted list
console.log("Formatted List:");
console.log(formattedList);

// Log the result of the check
console.log(result ? `Verify - "${breedToCheck}" is within list.` : `Verify -
"${breedToCheck}" is not within list.`);
```

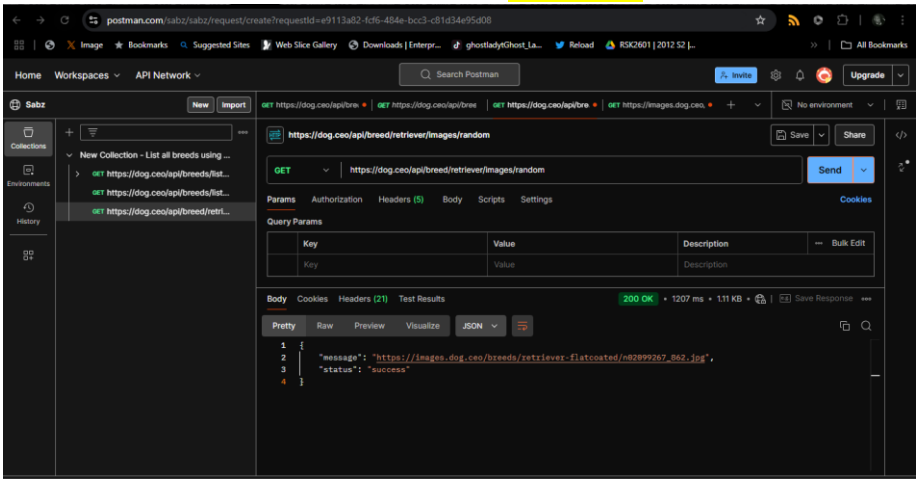
- Performing an API request – produce a list of sub-breeds for “retriever” like the output found on assessment document **Diagram 3**.
  - This would be as simple as demonstrated in the following screenshot, on the https request given on the site ***https://dog.ceo/api/breed/hound/list*** we would change /hound/list at the end of the url to /retriever/list

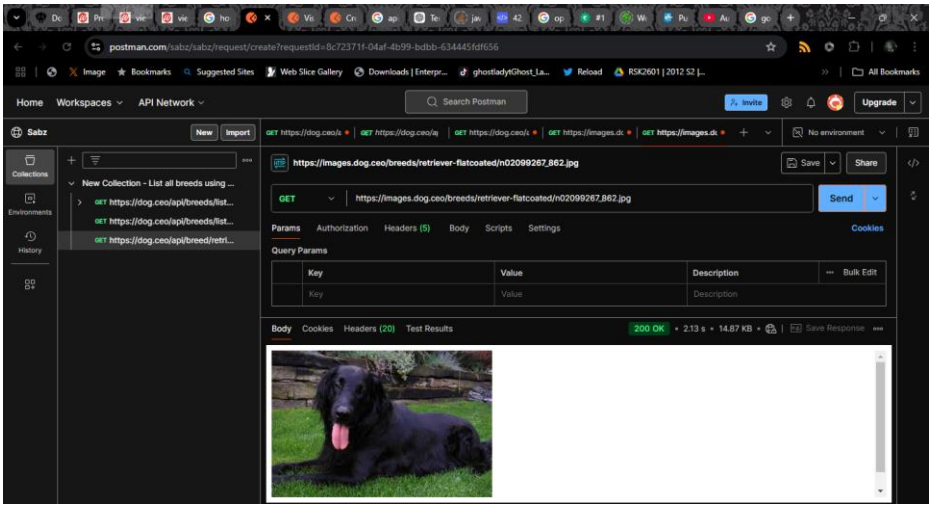


- Result of that slight change being as follows



- Performing an API request – produce a random image / link for sub-breed “golden” like the output found on assessment document **Diagram 4**.





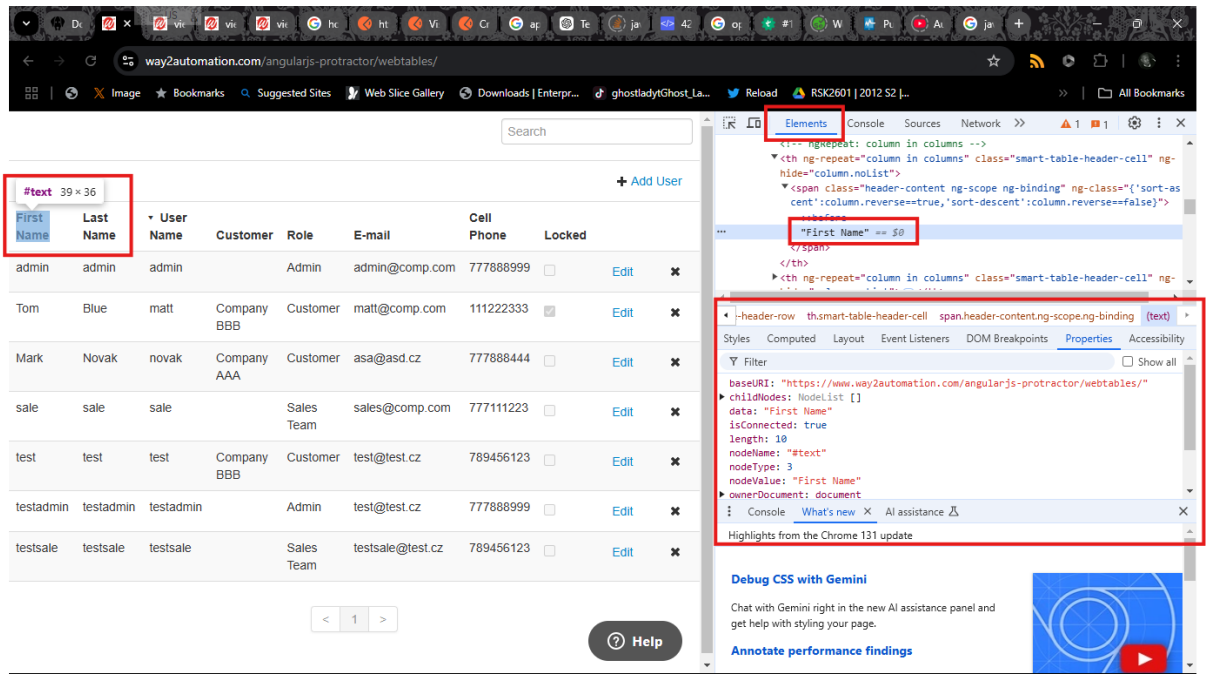
~end

Task 2 – Web Test Case Creation:

- Navigate to webpage
  - Using Java script (again), with an open source web browser automation tool / framework such as the Selenium web driver which Ive used many years ago, and an IDE like IntelliJ / Eclipse / JCreator, I would code these test cases, compile and run them to see if im getting the expected results I wanted. If I remember correctly every new version of the JDK/SDK has a bunch of libraries of methods one can lookup and use pulled into the IDE being used to code. Again, im embarrassingly VERY RUSTY using this language to code at the moment so some practice and work being put in is needed.
  - For this navigate, it was easier to check on the net as I had an idea that wondowDot (window.) and href: "" is something we used back then to navigate and launch a web page, this would be a basic Javascript used in a browser console.
  - In this example given the above, one would use either of the following:
    window.location.href = "webpage" or window.open("webpage") to navigate to the site.
  - Optionally can be completed by a simple "if / else statement" to check that indeed you've landed on the correct site / landing page:

```
// Check if the current URL is correct if (window.location.href === "webpage")
{
        console.log("You are at the correct URL.");

}

Else

{

console.error("You are NOT at the correct URL.");

}
```

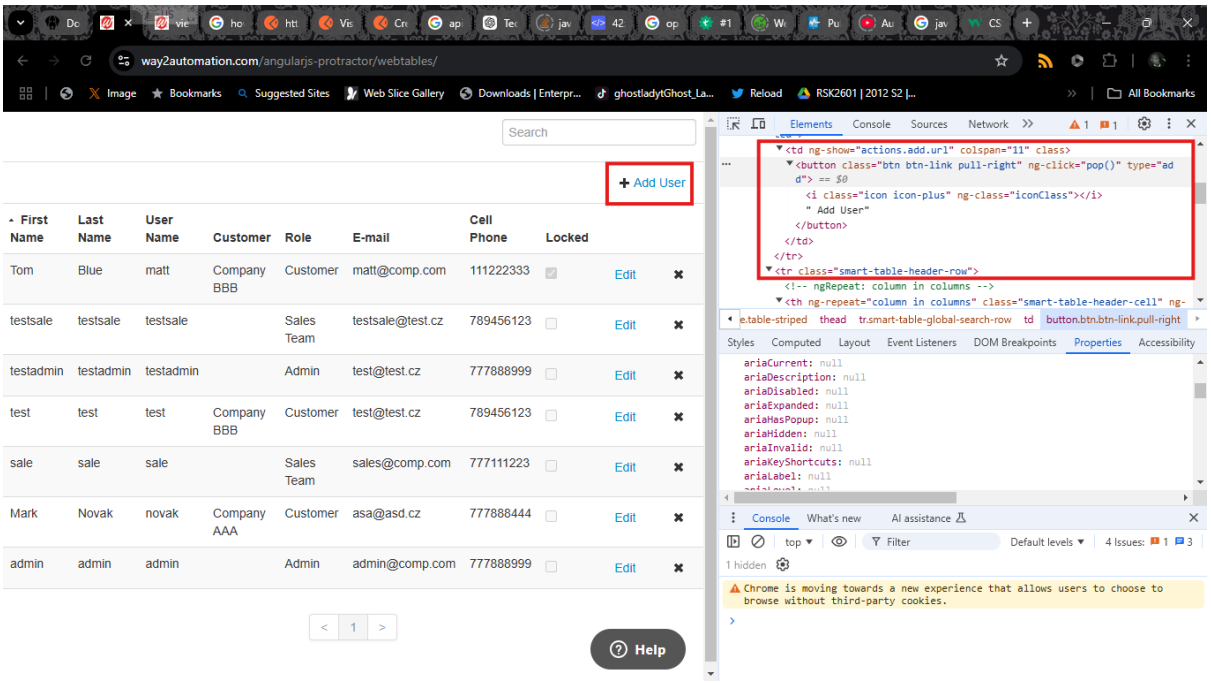
- Validating that I'm on the User List Table
  - When pointing the mouse cursor on the "First Name" heading and selecting "Inspect" > the inspect element window pops up such as on the right side of the below screenshot.
  - Under the heading "Elements" are a whole lot of web elements listed which one can be used to find a specific item, spot within the web browser.
  - In this case, we know that the "User List Table" consists of these 8 heading in bold, "First Name" being one of them therefore if we can have Javascript code in place which locates this element, it means therefore we are indeed on the "User List Table"




```
// Select the element and extract its text content
const element = aSpecificSelector
if (element)
{
  const expectedText = "First Name"; // Text we'd be expecting
  if (text === expectedText) {
    console.log("The inner text matches the expected value.");//Meaning we're within the "User List Table"
  } else {
    console.error(` Expected '${expectedText}', but got '${text}'. `);//Else we are not where we wanted to be
  }
} else {
  console.error("The element was not found on the page.");
}
```

- Clicking on "Add user"
  - Add user as per the following screenshot is a part of the element <button class="btn btn-link pull-right" ng-click="pop()" type="add"> Add User</button>

```
// Code would look something like the following..
const button = aSpecificSelector

if (button)
 {
   button.click(); // Simulates clicking on the AddUser “button”
   console.log("Add user button clicked successfully.");
}
else
{
   console.error("Add user button not found.");
}
```



- Adding users with details specified on the assessment document:
  - “First Name”
  - “Last Name”
  - “User Name”
  - “Password”
  - “Customer”
  - “Role”
  - “Email”
  - “Cell”

// Step 1: Navigate to the webpage as per demo code above from previous steps where either window.location.href = "webpage" or window.open("webpage") is used to navigate to the site.

// Step 2: Click the "Add User" button also as per above code demo.

// Step 3: Fill in the user detail fields

Google search to help me with this part brought back a solution where I need to locate input fields using AngularJS-specific attributes (ng-model) – have heard about AngularJS but have never worked with it before:

First Name: input[ng-model="formData.firstName"]

Last Name: input[ng-model="formData.lastName"]

User Name: input[ng-model="formData.userName"]

Password: input[ng-model="formData.password"]

Customer: input[ng-model="formData.customer"]

Role: input[ng-model="formData.role"]

Email: input[ng-model="formData.emai"]

Cell: input[ng-model="formData.cellNum"]

console.log("Filled in 'First Name'.");//We'd do the same on all fields and log their success captures

// Step 4: Click the "Save" button to add all captured data

The "Save" button is identified with button[ng-click="save(userForm)"].

console.log("User added successfully.");//Log successful data captured and saved, else also log an error should it not.