Bilkent University

Department of Computer Engineering

Verification & Validation

Test-Driver Development of Geolocation Services

Project 3

Maryam Shahid	21801344
Hammad Khan Musakhel	21801175
Murat Sevinç	21702603

April 22, 2022

Table of Contents

1. UML Diagrams

1.1. Use-Case Model

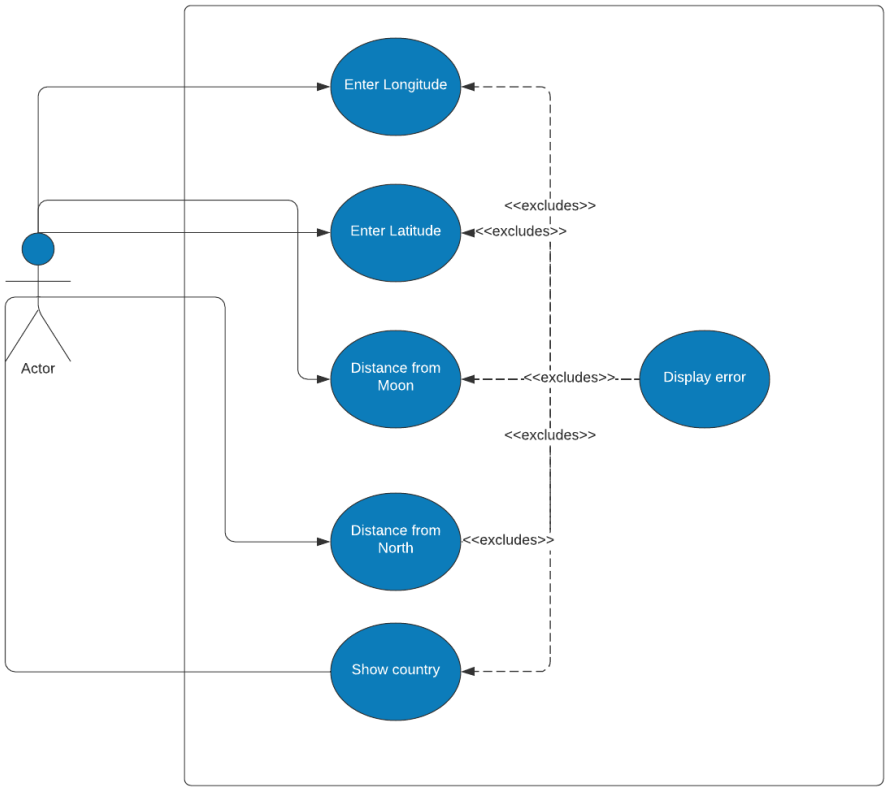

Figure 1: Use Case Diagram for Universal Location Application's functionalities

1.1.1. Enter Longitude

- **Participating actor:** User
- **Entry Condition:**
  - User enters numbers in decimal degrees
- **Exit condition:**
  - Entering a number in decimal degrees ranging from -90 to 90
- **Flow of Events:**
  - User enters the number
  - If correct, along with Latitude, contributes to enabling the 'Show Country' button
  - If incorrect, error message is displayed
- **Special Requirements:**
  - **Should not contain special characters or alphabets, and should always be in decimal format**

1.1.2. Enter Latitude

- **Participating actor:** User
- **Entry Condition:**
  - User enters numbers in decimal degrees
- **Exit condition:**
  - Entering a number in decimal degrees ranging from -180 to 180
- **Flow of Events:**
  - User enters the number
  - If correct, along with Longitude, contributes to enabling the 'Show Country' button
  - If incorrect, error message is displayed
- **Special Requirements:**
  - **Should not contain special characters or alphabets, and should always be in decimal format**

1.1.3. Distance to the Moon's core

- **Participating actor:** User
- **Entry Condition:**
  - User clicks the 'Show distance to Moon' button
- **Exit condition:**
  - Button clicked
  - GPS enabled to allow the application for geolocationing
- **Flow of Events:**
  - User clicks button
  - If GPS enabled, displays distance to moon's core
  - If coordinates entered, with GPS not enabled, displays distance to moon's core
- **Special Requirements:**
  - **GPS must be turned on for automatic location detection OR Coordinates entered**

1.1.4. Show distance to the Geographic North Pole

- **Participating actor:** User
- **Entry Condition:**
  - User clicks 'Show distance to North Pole'
- **Exit condition:**
  - GPS enabled for automatic user location detection
- **Flow of Events:**
  - User clicks 'Show distance to North Pole'
  - If GPS enabled, displays distance to the North Pole
  - If GPS not enabled, no distance shown and error message is displayed
- **Special Requirements:**
  - **GPS enabled**

1.1.5. Show Country

- **Participating actor:** User

- **Entry Condition:**
  - User clicks on ‘Show Country’ button
- **Exit condition:**
  - Coordinates of location entered
- **Flow of Events:**
  - User clicks on ‘Show Country’ button
  - If coordinates entered, displays country
  - If left empty, error message displayed
- **Special Requirements:**
  - **Coordinates entered**

1.2. Activity Model

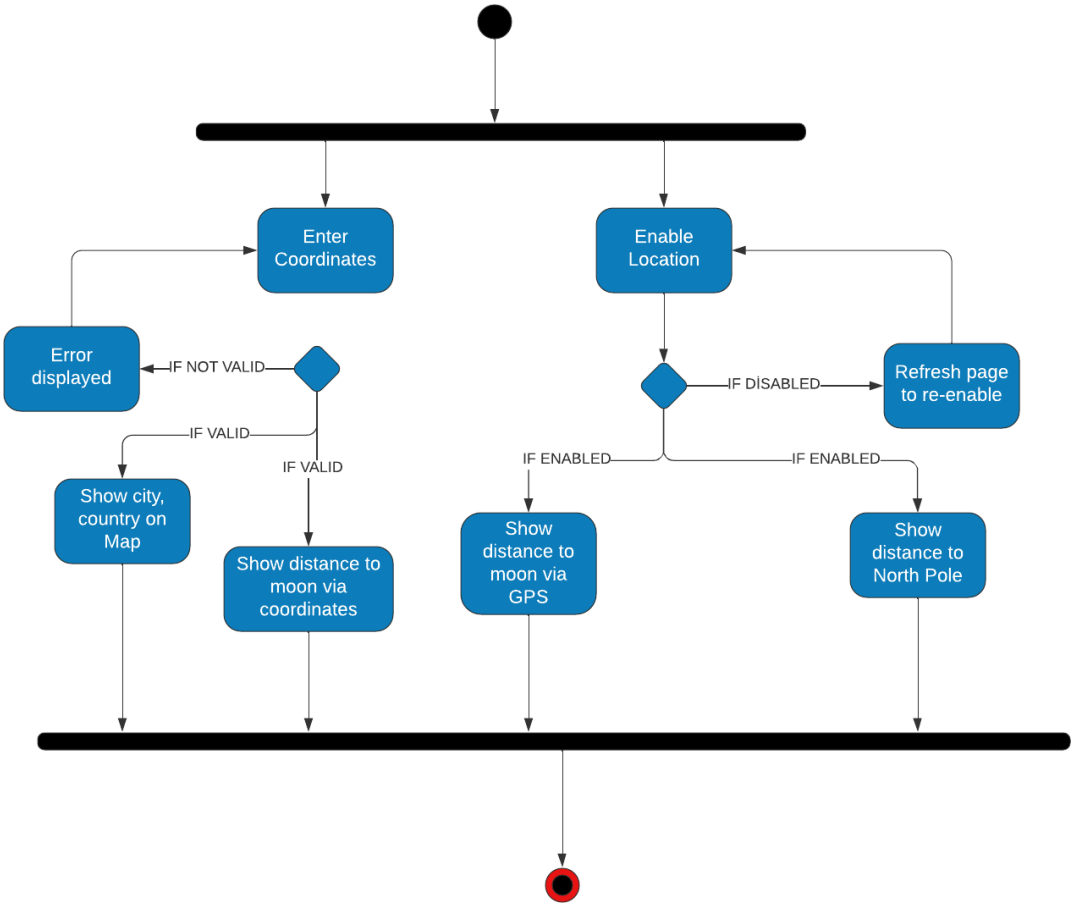

Figure 2: Activity Model For Geolocation Services Application

Once the application is launched, the webpage page prompts the user to enter longitude and latitude coordinates of their location. If the user enters inappropriate data with regards to any of the text boxes, an error message is displayed and he/she can re-enter. Additionally, if the user does not provide information for any information box, upon clicking the ‘Show Country’ button

an error message is displayed. However, if the information is accurate and matches the specified requirements, the ‘Show Country’ button displays the relevant details in form Google Maps. Moreover, if the automatic GPS is enabled the user can easily check the distance from their location to the core of the moon and the north pole. Hence, if the GPS is not enabled, the user can not see the distance to the north pole, and upon clicking the ‘Show distance to the North Pole’ button, an error message is displayed. Moreover, if accurate latitude and longitude values are provided with the ‘Show Country’ button working, the user can still check the distance from that location to the core of the moon by clicking the ‘Show distance to Moon’s core.’ Thus, it can be concluded that to display the country, coordinates are to be entered by the user; to display the distance to the north pole, automatic GPS needs to be enabled; to display the distance to the core of the moon, either GPS or coordinates entered by the user can be used.

1.3. State Model

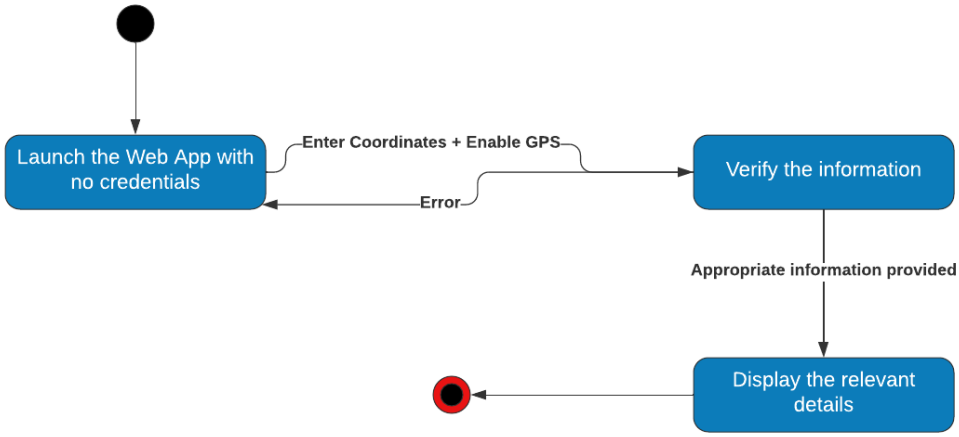

Figure 3: State Model for Geolocation Services Application

Once the application is launched, the system takes the form of a generic web page with buttons and text fields. If the user enters inappropriate credentials with regards to any of the information boxes, the state is not changed. Additionally, if the user provides authentic credentials which are tested against the specifications, the buttons display error and reverts the application to the initial stage.

1.4. Class Model

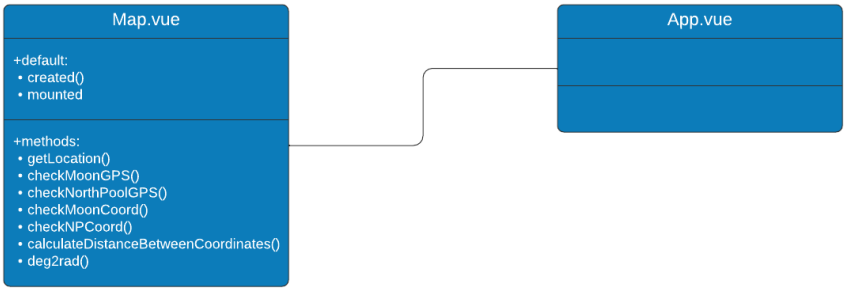

Figure 4: Class Model for Geolocation Services Application

The web application has been developed on Vue.JS, thus, Javascript was used for the logic part. Moreover, in the Map.vue component, multiple methods are used for the different functionalities. The elaborate implementation of these methods will be shown below in the implementation part of the document. The App.vue component is where the instance of Map.vue is called and the web application is displayed.

1.5. Sequence Model

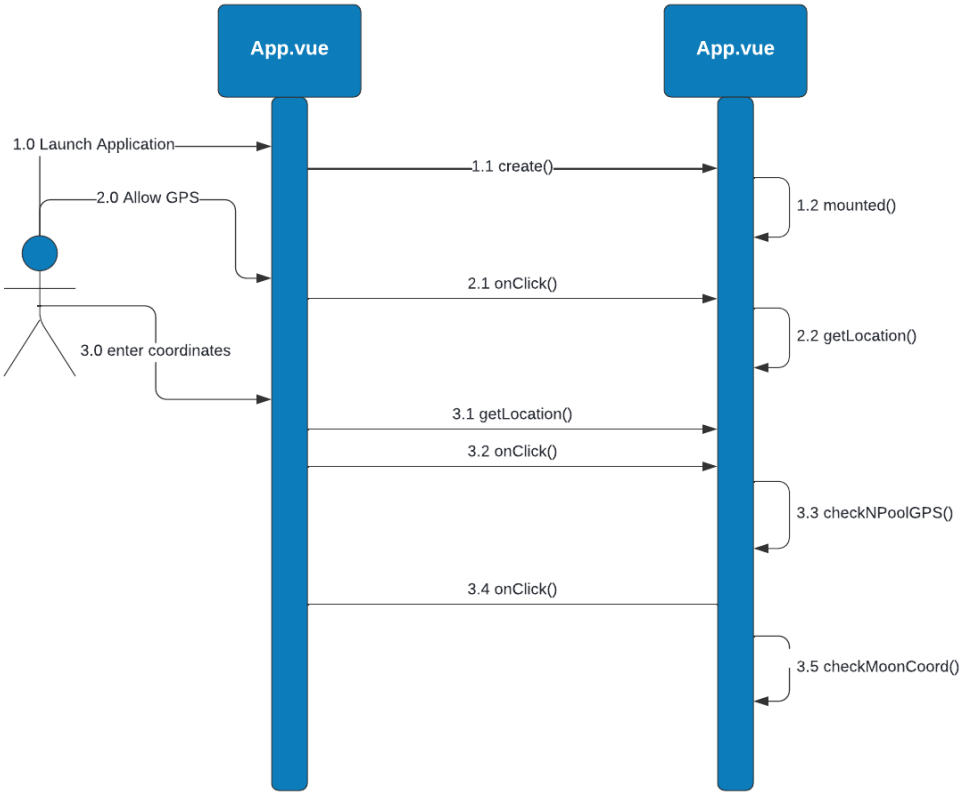

Figure 5: Sequence Model Geolocation Services Application

2. Implementation

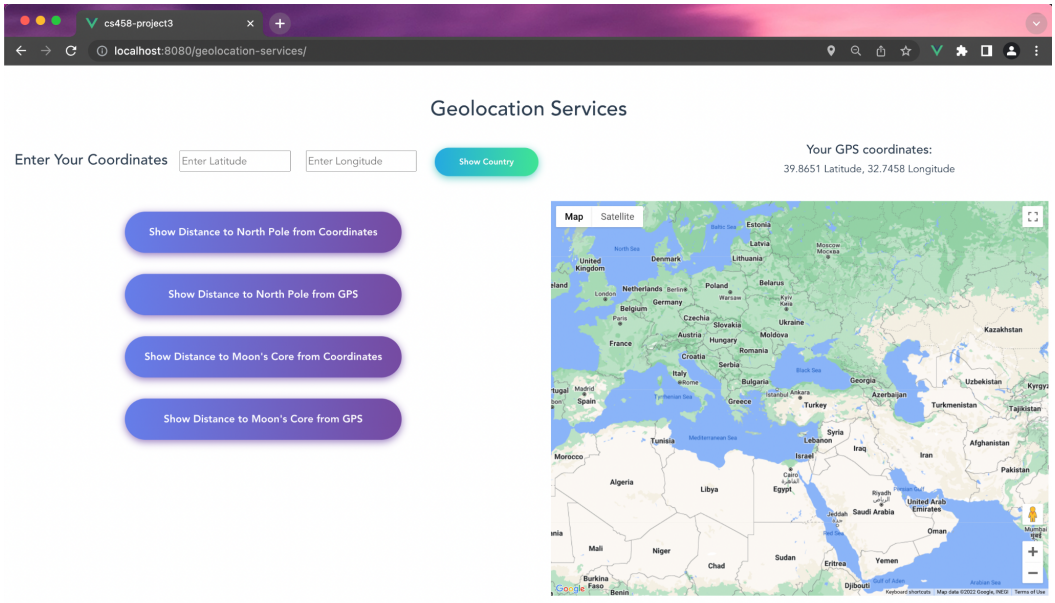

Figure 6: Main page upon launch with GPS allowed to track user

When the application is launched, the user is requested to allow automatic GPS. After that, the main page is loaded with Google Maps API and the user's GPS coordinates are shown in the top right corner.

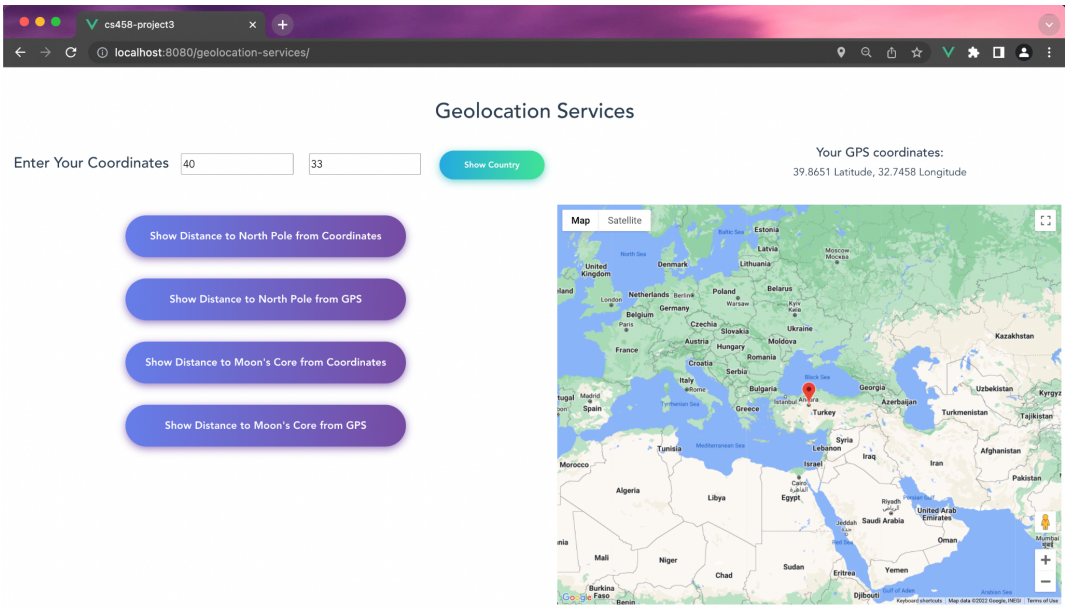

Figure 7: Main page with Google Maps marked

When the user enters appropriate coordinates and clicks on the 'Show Country' button, the map is updated with a marker pointing to the location corresponding to the coordinates entered. The map is updated accordingly with every iteration when coordinates are updated.

```
getLocation() {
  if (this.checkMarker) {
    this.marker.setMap(null);
  }
  // eslint-disable-next-line no-undef
  var myLatlng = new google.maps.LatLng(this.inputLat, this.inputLng);
  if (this.inputLat !== null && this.inputLng !== null) {
    // eslint-disable-next-line no-undef
    var center = new google.maps.LatLng(myLatlng);
    this.map.panTo(center);
  }
  // eslint-disable-next-line no-undef
  this.marker = new google.maps.Marker({
    position: myLatlng,
    title:"Map Marker"
  });
  this.marker.setMap(this.map)
  this.checkMarker = true
},
```

Figure 8: Code snippet of GetLocation() method which points the map to the user's country

The above code snippet manages the location from the coordinates. It centers the Google Map to the coordinates provided and draws a marker on the same coordinates.

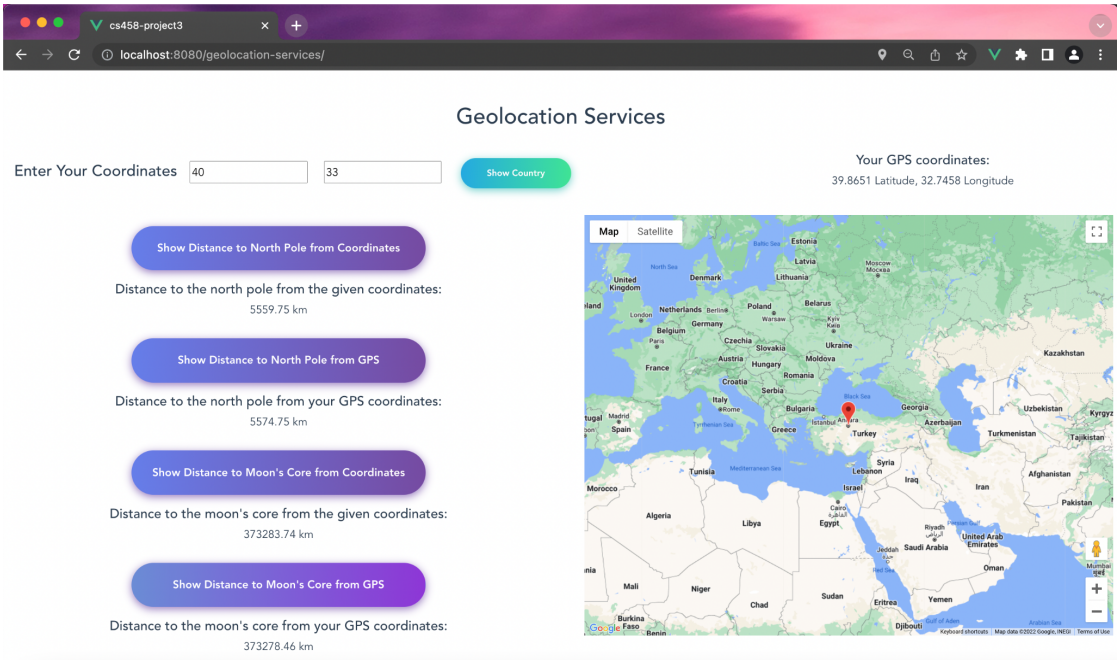

Figure 9: Main page with each of the four buttons on left clicked

From the input coordinates, the user is allowed to calculate the distance to the north pole and the distance to the moon's core by clicking the respective buttons. The user is also allowed to see the distance from their GPS coordinates to the north pole and the moon's care, by clicking its respective buttons. The figure above captures a moment with all buttons pressed and their respective data.

Distance to the North Pole:

The distance to the North Pole is computed using the method below which calculates the distance between two coordinates. Here the application passes either the user's input coordinates or their GPS coordinates along with (90, 0) for the North Pole. The distance calculated is shown in kilometers.

```
calculateDistanceBetweenCoordinates(lat1, lon1, lat2, lon2) {
  var earthRadiusKm = 6371;

  var dLat = this.degreesToRadians( degrees: lat2 - lat1);
  var dLon = this.degreesToRadians( degrees: lon2 - lon1);

  lat1 = this.degreesToRadians(lat1);
  lat2 = this.degreesToRadians(lat2);

  var a =
      Math.sin( x: dLat / 2) * Math.sin( x: dLat / 2) +
      Math.sin( x: dLon / 2) *
      Math.sin( x: dLon / 2) *
      Math.cos(lat1) *
      Math.cos(lat2);
  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt( x: 1 - a));
  return earthRadiusKm * c;
},
```

Figure 10: Code snippet of the method used to calculate distance between two coordinates

Distance to the Moon's Core:

The distance to the moon's core is calculated using the npm module SunCalc [2]. After including the module in the project's package.json file, SunCalc's getMoonPosition function can be used as shown in the following figure. It returns the distance to the moon's core from the given coordinates.

```
this.showMoonCoord = !this.showMoonCoord
var SunCalc = require('suncalc2');
const date = new Date();
this.moonDistanceCoord = SunCalc.getMoonPosition(date, this.inputLat,  this.inputLng).distance
```

Figure 11: Code snippet of SunCalc's getMoonPosition function to calculate distance to the moon

The full code can be accessed in the Map.vue component of the project repository [1] as https://github.com/maryamShahid/cs458-project3/blob/master/src/components/Map.vue

Moreover, as shown in the following figure (figure 12), the application gives an error in case the user tries to use "Show Distance to North Pole from Coordinates" or "Show Distance to Moon's Core from Coordinates" without first inputting the coordinates in their respective input boxes. The application also gives an error in case the user's GPS is disabled and their GPS coordinates are not retrieved.

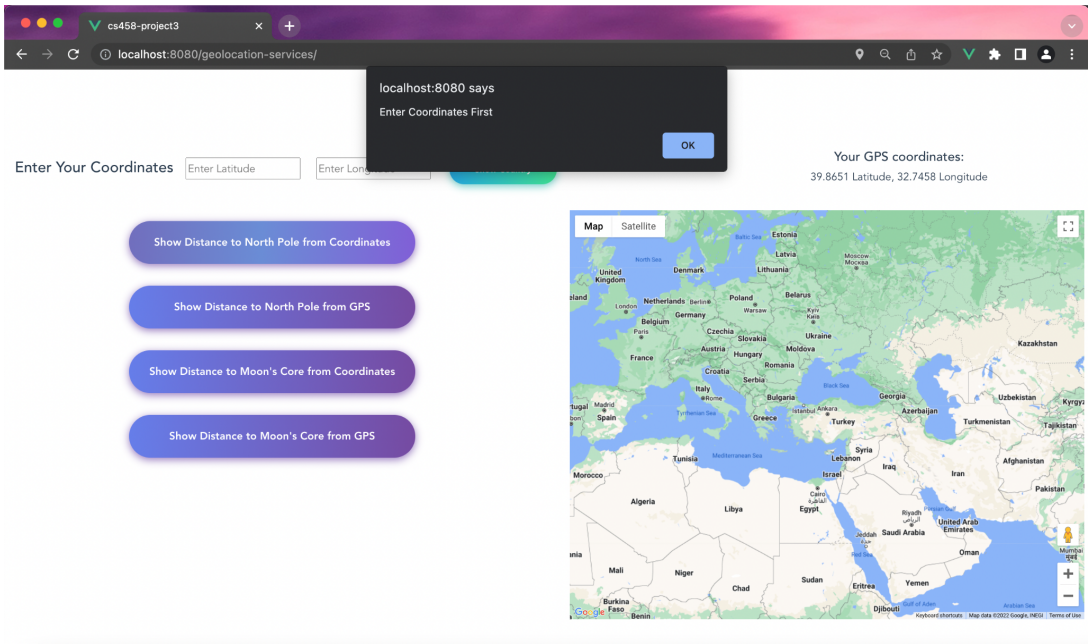

Figure 12: Error displayed for coordinates

3. Test Cases with Code Samples

3.1. Test Case 1

In this test case, coordinates of Iceland are provided (64.9631, -19.0208) and it is expected that the marker on Iceland exists. By doing so, we validate the coordinate inputs and map screen work as expected.

```
@Test
public void testingShowingCountryByCoordinates() throws InterruptedException {
    ChromeDriver driver = createChromeDriver();

    driver.get(projectURL);

    driver.findElement(latitude).sendKeys( ...keysToSend: "64.9631");
    driver.findElement(longitude).sendKeys( ...keysToSend: "-19.0208"); // Coordinates for Iceland.

    driver.findElement(sendButton).click();

    driver.executeScript( script: "window.scrollBy(0,750)", ...args: "");
    Thread.sleep( millis: 1700);

    assertThat(driver.findElement(markerPositionOfIceland).isDisplayed())
            .isTrue();

    driver.close();
}
```

Figure 13: Test Case 1

3.2. Test Case 2

This test case checks a corner case: when longitude input is not provided. By default, we decided that zero should be used when no inputs are provided. Thus, we enter only latitude input and press the “Show Country” button. Coordinates (45, 0) is located in France (near Bordeaux). The application validates this corner case by checking whether or not a marker exists in France.

```
@Test
public void testingDefault0LongitudeWhenLongitudeNotProvided() throws InterruptedException {
    ChromeDriver driver = createChromeDriver();

    driver.get(projectURL);

    driver.findElement(latitude).sendKeys( ...keysToSend: "45"); // 45, 0 is in France

    driver.findElement(sendButton).click();

    driver.executeScript( script: "window.scrollBy(0,750)",  ...args: "");
    Thread.sleep( millis: 500);

    assertThat(driver.findElement(markerPositionOfFrance).isDisplayed())
            .isTrue();

    driver.close();
}
```

Figure 14: Test Case 2

3.3. Test Case 3

This is the case where we test the GPS feature of the application. Basically, the test scenario checks whether or not the string with the location information is user’s current location.

```
@Test
public void testGPS() {
    ChromeDriver driver = createChromeDriver();
    driver.get(projectURL);

    String GPSText = driver.findElement(GPSResult).getText();

    assertThat(GPSText)
            .isEqualTo("39.9671 Latitude, 32.8565 Longitude"); // My Home, coordinates may vary.

    driver.close();
}
```

Figure 15: Test Case 3

3.4. Test Case 4

After pressing the “Show Distance to North Pole from GPS” button, it reads the location information and compares it with the actual distance (5563.4 km).

```
@Test
public void testNorthPoleFromGPS() {
    ChromeDriver driver = createChromeDriver();
    driver.get(projectURL);

    driver.findElement(showNPFromGPS).click();

    assertThat(driver.findElements(distanceResults).get(1).getText())
            .isEqualTo("5563.40 km"); // Distance is from my home, coordinates may vary.

    driver.close();
}
```

Figure 16: Test Case 4

3.5. Test Case 5

This test case checks whether the distance from the moon’s core is calculated with GPS.

```
@Test
public void testMoonFromGPS() throws InterruptedException {
    ChromeDriver driver = createChromeDriver();
    driver.get(projectURL);


    WebElement showDistanceButton = driver.findElement(showMoonFromGPS);

    ((JavascriptExecutor) driver).executeScript( script: "arguments[0].scrollIntoView(true);", showDistanceButton);
    Thread.sleep( millis: 350);

    showDistanceButton.click();

    try {
        assertThat(driver.findElements(distanceResults).get(1).getText())
                .isEqualTo("373207.71 km");     // Distance may vary due to the movement of the moon.
    } catch(AssertionError e) {
        e.printStackTrace();
        driver.close();
    }

}
```

Figure 17: Test Case 5

3.6. Test Case 6

Similar to test case 5, this test case checks the distance from the moon’s core, but this time it is calculated from a specific coordinates provided by the user, not from GPS.

```java
@Test
public void testMoonFromCoordinates() throws InterruptedException {
    ChromeDriver driver = createChromeDriver();
    driver.get(projectURL);

    driver.findElement(latitude).sendKeys( ...keysToSend: "64.9631");
    driver.findElement(longitude).sendKeys( ...keysToSend: "-19.0208"); // Coordinates for Iceland.

    WebElement showDistanceButton = driver.findElement(showMoonFromCoordinates);

    ((JavascriptExecutor) driver).executeScript( script: "arguments[0].scrollIntoView(true);", showDistanceButton);
    Thread.sleep( millis: 350);

    showDistanceButton.click();


    try {
        assertThat(driver.findElements(distanceResults).get(1).getText())
                .isEqualTo("373216.14 km");     // Distance may vary due to the movement of the moon.
    } catch(AssertionError e) {
        e.printStackTrace();
        driver.close();
    }
}
```

Figure 18: Test Case 6

3.7. Test Case 7

This test case checks the error alert functionality of the application. When a user presses the buttons ‘Show distance to North Pole with coordinates’ or ‘Show distance to Moon’s core with coordinates,’ the relative information is provided. However, if no coordinates are provided, an output error ‘Enter Coordinates First’ is displayed. We perform this simulation in this test case.

```java
@Test
public void testErrorWhenCoordinatesNotProvided() throws InterruptedException {
    ChromeDriver driver = createChromeDriver();
    driver.get(projectURL);

    WebElement showDistanceButton = driver.findElement(showMoonFromCoordinates);

    ((JavascriptExecutor) driver).executeScript( script: "arguments[0].scrollIntoView(true);", showDistanceButton);
    Thread.sleep( millis: 350);
    showDistanceButton.click();

    String alertMessage = driver.switchTo().alert().getText();

    assertThat(alertMessage)
            .isEqualTo("Enter Coordinates First");

    driver.switchTo().alert().accept();
    driver.close();
}
```

Figure 19: Test Case 7

4. Refactoring

After writing the unit test for a functionality and implementing it, we sought to refactor our code. Refactoring is the improvisation of the code without changing the external behavior of the code: no other functionality is broken down. It is pertinent to notice that every functionality developed prior to the refactoring a newly developed functionality should hold. After refactoring, testing of existing functionalities is carried out to check if a functionality is broken down.
In the case of Geolocation Services Application, we used refactoring to make our code more readable, coherent, concise, and non-redundant. After every test clearance of a functionality, we improvised the code structure to optimize the app's design without breaking functionalities. It allowed us to remove multiple unwanted elements in different components that were not required.

5. TDD with respect to Velocity & Code quality

TDD is a step-by-step development technique. In TDD, we write the unit test prior to programming; this gives a clear idea to the programmer for what the goal of the functionality needs to be. In the red phase of TDD, we write a failing unit test for the functionality to add. After that, if we successfully pass the test for that functionality with minimal code, we transition into the green phase. After implementation, we can then refactor the code in the refactor phase of TDD. These all phases are aimed at increasing the code quality of the project and to boost the development process. By writing tests before implementation, the programmer has the general idea of the functionality and thus, can focus better. Moreover, the new functionalities implemented need to not break other functionalities increasing code quality and velocity.
We divided our project in three separate classes, as outlined in the project description, for TDD.

6. References

[1]. Project Repository, https://github.com/maryamShahid/cs458-project3.

[2]. SunCalc npm module, https://www.npmjs.com/package/suncalc.