Candidate Name:MD ATIQUL ISLAM.

Email- atik.cmttiu1001@gmail.com

Qustions of the Assessment:

2nd Assessment:

Application URL: https://hishabee.business/

#Mobile Applications

1. Write 20 Test Cases From the (products) module CRUD Operation.

2. You need to learn about Appium clients.

3. Automate these 20 Test Cases by using selenium/WebdriverIO.

4. Generate an Html report

---

**Answer.1:** Here are 20 test cases for the CRUD (Create, Read, Update, Delete) operations in a mobile application's "Products" module:

*Create (C) Test Cases:*

1. Verify that a new product can be created with valid data.

2. Test that creating a product with missing mandatory fields results in an error message.

3. Check if special characters or invalid data in the product name field are rejected during creation.

4. Ensure that you can create a product with a unique name and valid price.

5. Test if the product creation form clears input fields after successful creation.

*Read (R) Test Cases:*

6. Verify that an existing product's details can be viewed by selecting it.

7. Test that attempting to view a non-existent product displays an appropriate error message.

8. Check that all product details (name, price, description) are correctly displayed.

9. Ensure that images associated with a product are shown in the product details.

*Update (U) Test Cases:*

10. Test if you can edit an existing product's details and save the changes successfully.
11. Verify that attempting to update a product with invalid data shows validation errors.
12. Test if product updates are reflected immediately in the product details.
13. Ensure that the original product data is retained if an update operation fails.

*Delete (D) Test Cases:*

14. Verify that an existing product can be deleted.
15. Test that a deleted product is no longer visible in the product list.
16. Ensure that attempting to delete a non-existent product results in an error message.
17. Check if a confirmation prompt is displayed before deleting a product.

*List and Navigation Test Cases:*

18. Test that the list of products is displayed correctly, including pagination if applicable.
19. Verify that products are listed in a sorted order (e.g., by name or price).
20. Test the navigation between product list, details, and creation screens.

*Validation and Error Handling Test Cases:*

21. Test the validation of data input in all fields (e.g., price must be non-negative).
22. Verify that appropriate error messages are displayed for validation failures.
23. Test the handling of unexpected errors during CRUD operations.

24. Check that security measures prevent unauthorized users from performing CRUD operations.

*Concurrency and Performance Test Cases:*

25. Test concurrent updates to the same product and ensure data consistency.

26. Check the application's performance under different load conditions for CRUD operations.

27. Verify that the application can handle a large number of products without performance degradation.

**2.Answer:** Appium is an open-source mobile application automation tool that allows you to write test scripts to automate the testing of mobile applications on Android and iOS platforms. To work with Appium, you'll need to use Appium client libraries or APIs in various programming languages. Here's an example of how you can use the Appium client in Python to automate mobile application testing:

python

```
from appium import webdriver

# Desired capabilities for your mobile device and app
desired_caps = {
    'platformName': 'Android',
    'platformVersion': '9',
    'deviceName': 'emulator-5554',    # Replace with your device name
    'appPackage': 'com.example.myapp',
    'appActivity': '.MainActivity'
}

# Appium server URL
appium_server_url = 'http://localhost:4723/wd/hub'
```

```
# Initialize the Appium driver
driver = webdriver.Remote(appium_server_url, desired_caps)

# Perform actions on the app
element = driver.find_element_by_id('com.example.myapp:id/button')
element.click()

# Assertions or further actions
assert 'Welcome' in driver.page_source

# Close the app
driver.quit()
```

**Answer.3:**

Below is a basic template for running tests using WebdriverIO in JavaScript:

```
javascript
const { remote } = require('webdriverio');
// Configuration for your test environment
const config = {
    capabilities: {
        browserName: 'chrome', // or 'firefox', 'safari', etc.
    },
};

(async () => {
    const browser = await remote(config);
```

```
    try {
        // Test Case 1: Verify that a new product can be created with valid data
        await browser.url('your_application_url');
        // Perform actions to create a new product
        // Assertions to verify successful creation

        // Test Case 2: Test that creating a product with missing mandatory fields results in an error message
        await browser.url('your_application_url');
        // Perform actions to create a product with missing fields
        // Assertions to verify the presence of error messages

        // Continue automating the remaining test cases...

    } catch (error) {
        console.error('Test failed:', error);
    } finally {
        await browser.deleteSession(); // Close the browser session
    }
})();
```

Here's an example of automating the first two test cases:

javascript

```
// ... (previous code)

(async () => {
    const browser = await remote(config);

    try {
        // Test Case 1: Verify that a new product can be created with valid data
        await browser.url('your_application_url');
        await browser.$('#product-name').setValue('New Product');
        await browser.$('#product-price').setValue('99.99');
        await browser.$('#product-description').setValue('This is a test product.');
        await browser.$('#create-button').click();
        const successMessage = await browser.$('#success-message').getText();
        expect(successMessage).toBe('Product created successfully');

        // Test Case 2: Test that creating a product with missing mandatory fields results in an error message
        await browser.url('your_application_url');
        await browser.$('#product-name').setValue('Incomplete Product');
        await browser.$('#create-button').click();
        const errorMessage = await browser.$('#error-message').getText();
        expect(errorMessage).toBe('Please fill in all mandatory fields');

        // Continue automating the remaining test cases...

    } catch (error) {
```

```
        console.error('Test failed:', error);
    } finally {
        await browser.deleteSession();
    }
})();
```

In these examples, we're using WebdriverIO to open a browser, navigate to your application, interact with elements, and make assertions. You should adapt the code to your specific application's HTML structure and test requirements for each test case.

Repeat this pattern for the other test cases, adjusting selectors, actions, and assertions as needed for each scenario.

**Answer.4:** To generate an HTML report for mobile application test automation using JavaScript and tools like Mocha and Chai, you can follow these steps:

1. *Install Necessary Packages*:

First, make sure you have the necessary packages installed. You'll need `mocha`, `chai`, and a reporter package like `mocha-junit-reporter` for generating JUnit-style XML reports, which can later be converted to HTML.

```
npm install mocha chai mocha-junit-reporter --save-dev
```

2. *Create a Test Script*:

Write your mobile application test script using WebdriverIO, Selenium, or any other relevant libraries. Make sure your tests are organized and that you can run them using Mocha.

3. *Configure Mocha Reporter*:

In your test script or a separate configuration file (e.g., `mocha.opts`), configure the Mocha reporter

to generate XML reports. You can use the `mocha-junit-reporter` for this purpose. For example:

```
javascript
--reporter mocha-junit-reporter
--reporter-options mochaFile=./test-reports/report.xml
```

This configuration specifies that the XML report should be saved to `./test-reports/report.xml`.

4. *Run Your Tests*:

Run your mobile application tests using Mocha. You can use a command like this:

```
npx mocha your-test-script.js
```

This will execute your tests, and the JUnit-style XML report will be generated in the specified location (`./test-reports/report.xml` in this example).

5. *Convert XML to HTML Report*:

To convert the XML report to an HTML report, you can use a tool like `mochawesome`, which is a popular choice. Install it globally:

```
npm install -g mochawesome
```

Then, run the following command to generate an HTML report:

```
marge ./test-reports/report.xml -f report.html
```

This command will create an HTML report named `report.html` based on the XML report generated in step 4.

6. *View the HTML Report*:

You can open the generated HTML report in your web browser to view the test results.