QA Engineer Internship Coding Test Fuad Samadov

Part 1: Test Case Writing

You can find the test cases that I wrote in `part1.py`
The test cases that I wrote assume that there is a function `register_account(username, password, email, subscribe_newsletter)` which processes the form submission and returns a response in the following json format:

```
{
    "status_code": number,
    "msg": "message"
}
```

Part 2: Bug Identification and Reporting

Bug Report: QA Registration Form

Page: `QA Registration Form`

Bug 1: No email format validation

Description: The form validation only checks for the presence of the `@` symbol in the email field but does not check for a complete email with regex (e.g., `user@domain.com`).

Steps to Reproduce:

1. Open the registration form page.
2. Enter an invalid email format, such as `user@`.
3. Fill in the other fields correctly.
4. Click on the **Register** button.

Expected Result: The form should prevent submission and show an alert for an invalid email format (e.g., "Please enter a valid email format.").

Actual Result: No alert is shown, and the form can be submitted with an invalid email format.

Severity: High

Recommendation: Add an email regex that validates the user email.

Bug 2: Missing required field validation

Description: There is no validation to check if the username, password, email fields are empty.

Steps to Reproduce:

1. Open the registration form page.
2. Leave all fields (username, password, email) blank.
3. Click on the **Register** button.

Expected Result:

The form should prevent submission and show alert messages for required fields (e.g., "Username is required," "Password is required," "Email is required").

Actual Result:

An alert is shown, but it does not specify that the fields are empty. Instead, it references the character count.

Severity: Medium

Recommendation: Add checks to ensure that all required fields are filled instead of only referencing the character count.

Bug 3: Weak password validation

Description: The password validation only checks for a minimum length of 8 characters but does not enforce additional security requirements such as letters, numbers, special characters.

Steps to Reproduce:

1. Open the registration form page.
2. Enter a password consisting of only numbers (e.g., `12345678`).
3. Fill in the other fields correctly.
4. Click on the **Register** button.

Expected Result: The form should prevent submission and alert the user to use a stronger password.

Actual Result: No alert is shown, and the form accepts a password with only numbers.

Severity: Medium

Recommendation: Implement password strength validation function to improve account security.

Part 3: Automated Testing Challenge

You can find the script in `part3.py`

Part 4: Performance Testing Scenario

Performance Testing Plan

1. Objectives

The main objectives of the performance test are the following:

- **Load handling**: determine the maximum number of users the application can handle at the same time without evident degradation in performance.
- **Response time under high load**: measure the response time for key functions such as user sign up, user sign in, etc. when the system is under a high load.
- **Stability**: Ensure the application remains stable with a steady response time and minimal error rate at a high usage level.

- **Identify bottlenecks**: identify potential performance bottlenecks and their causes.

2. Tools

Chosen Tool: Python Locust stress testing tool

Justification:

- **Scalability**: Python's Locust stress testing tool is capable of simulating thousands of requests coming from hundreds of different users (threads).
- **Ease of use**: Syntax is very friendly, as well as support for multiple OS (With additional tools for UNIX-based systems).
- **Open Source**: Locust is open-source and it can be seamlessly integrated in any existing testing suite.

3. Test Cases

1. **User sign-up load test**
    - **Objective**: Verify how well the webapp handles a high volume of user sign-up requests.
    - **Description**: Simulate multiple users attempting to sign up at the same time to test the handling of sign-up requests under load.
    - **Expected Outcome**: The application should handle sign-up requests with minimal delay, ideally under a 2-second response time, with an error rate below 0.1%.
2. **User sign-in load test**
    - **Objective**: Measure performance during a high volume of user sign-in requests.
    - **Description**: Simulate sign-in requests from existing users at the same time, mimicking how it could happen in the real-world peak activity.
    - **Expected Outcome**: The application should handle the requests within a response time of 1 second or below with minimal errors.
3. **Spike test for user sign-up and sign-in**

  - **Objective**: Check the system’s ability to manage sudden spikes in traffic.
  - **Description**: Gradually and slowly increase the load of simultaneous users from 1,000 to 10,000 within a short period of time (10 seconds) and monitor system performance.
  - **Expected Outcome**: The application should adapt to sudden increases in traffic with minimal latency while avoiding crashes.

4. Metrics

1. Response Time
2. Error Rate
3. Throughput
4. Simultaneous Users
5. CPU and Memory Utilization
6. Network Latency

5. Test Environment

- **Server Configuration**:
  - Set up the cloud-based service on an environment identical to the one used for production
  - Ensure that the service database and backend are optimized and scaled to handle peak load (e.g., 16 CPU cores, 64 GB RAM, autoscaling and load balancing enabled, etc.).
  - Choose the servers to be located in a distributed setup geographically in order to reflect real-world user distribution.
- **Network Settings**:
  - Use a high-bandwidth network to ensure the test environment can generate sufficient load without being limited by network limitations.
  - Configure network throttling options to reflect various user network conditions like 4G, 3G, 2G, Wi-Fi.

6. Analysis

- **Data collection**: collect test data on the key metrics discussed across the duration of each test.
- **Identify bottlenecks:**
    - Identify where and when response times increase significantly, which can point to whether bottlenecks occur at a server or database level.
    - Check the error logs to see the reason behind certain failures like timeouts, server errors 500, authentication errors, etc.
    - Check CPU, memory, and network usage to see if hardware works at its maximum capacity and limits performance of the application.
- **Make recommendations**:
    - For all bottlenecks uncovered determine whether issues arise from badly written code, shortage in hardware resources inefficiencies, limitations of the network connection, etc.
    - Having found the bottlenecks, recommend optimizations in the areas where they would work towards solving/alleviating the problem.
- **Write a report**: Write a report summarizing the test outcomes.

Part 5: Problem-Solving Questions

Scenario 1

In this situation, my approach would involve a certain mix of analytical steps and efforts to improve the quality of future releases.

First, I would analyze the bug reports. I would categorize them to identify trends, such as bugs happening in specific modules, due to specific functions, and identify common error types. This will help me understand if an issue is coming from certain feature areas.

Next, I would review the development and testing process. I would check again the development specifications to make sure that they are clear and cover all edge cases. If the specifications are not up to standards, software developers might miss the essential requirements or assume certain essential behavior, which sometimes can be even worse.

Afterwards, I would conduct a root cause analysis by working with the development and qa teams. This will help me identify factors like rushed or tight deadlines, specs clarity, bad code areas, etc.

Depending on the result of the above steps, my strategy can go multiple ways. Without a doubt there will be a need for improved documentation and communication across different teams. Also, I might need to start suggesting TDD (test-driven development) in order to catch tests earlier in the development stage.

Lastly, I will establish a process where the dev and qa teams jointly review the requirements before development pipeline sets off. And as a side note, if there are any tests which are conducted manually, I will spend effort to automate them to exclude the possibility of human error.

Scenario 2

Changing from manual testing to automated testing is a big step for a qa team, especially if some members do not feel confident enough in their programming skills. Here are a couple of thoughts that come to my mind on how to ensure a smooth transition while keeping the team motivated:

1. Most importantly, create a supportive learning environment
2. Use low-code or no-code tools where possible
3. Constantly highlight the value of automation
4. Encourage a growth mindset by recognizing the team’s progress over perfection
5. Set clear goals a celebrate their achievement

Part 6: Tool Proficiency

Task 1 (JIRA)

Not completed

Task 2 (Postman)

1. Create a new user

Request Name: Create User

Method: POST

URL: `{{base_url}}/users`

Headers:

Content-Type: `application/json`

Body:

```
{
    "username": "fuadsamadov",
    "email": "fuadsam02@kaist.ac.kr",
    "password": "TestPassword123!"
}
```

Test code:

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});

pm.test("Response body has user ID", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("id");
});
```

2. Retrieve user information

Request Name: Get User

Method: GET

URL: `{{base_url}}/users/{{user_id}}`

Headers:

Accept: `application/json`

Test code:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response contains user details", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("id", {{user_id}});
    pm.expect(jsonData).to.have.property("username");
    pm.expect(jsonData).to.have.property("email");
});
```

3. Update a user

Request Name: Update User

Method: PUT

URL: `{{base_url}}/users/{{user_id}}`

Headers:

Content-Type: `application/json`

Body:

```
{
    "username": "updatedfuad",
    "email": "updatedfuad@gmail.com"
}
```

Test code:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response confirms update", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("username", "updatedfu
    pm.expect(jsonData).to.have.property("email", "updatedfuad@
});
```

4. Delete a user

Request Name: Delete User

Method: DELETE

URL: `{{base_url}}/users/{{user_id}}`

Headers:

Content-Type: `application/json`

Test code:

```
pm.test("Status code is 204", function () {
    pm.response.to.have.status(204);
});

pm.test("Response body is empty", function () {
    pm.expect(pm.response.text()).to.equal("");
});
```