National University
of computer and emerging sciences

Assignment #3

Course:

Software Quality Engineering

Topic:

LAYP
(API Testing Automation Framwork)

Submitted by:

Roll No.	**Name**
22F-3714	Faizan Tariq
22F-3722	Muhammad Abdullah

Submitted to:

Mr. Jawad Khalid

Contents

LAYP-API framework employs Jest for unit and integration testing and k6 for load testing. This combination ensures that the API is robust, performs well under load, and meets expected functionality.

Prerequisites

Before you start, ensure you have the following installed on your machine:

- **Node.js**: The JavaScript runtime for executing the API and tests.
- **npm**: Node package manager, which comes with Node.js.
- **Jest**: A JavaScript testing framework for unit and integration testing.
- **k6**: A modern load testing tool that provides performance testing capabilities.

Basic Setup

1. **Clone the Repository**
    1. Use Git to clone the LAYP-API repository to your local machine:

➢ git clone https://github.com/whis-19/LAYP.git

2. **Navigate to the API Directory**
    1. Move into the LAYP-API directory:

➢ cd LAYP/LAYP-API

3. **Install Dependencies**
    1. Install the required Node.js packages, including Jest:

➢ npm install

4. **Environment Configuration**
    1. Set up any necessary environment variables by creating a **.env** file in the root directory. This file may include database connection strings, API keys, and other configuration details.

Jest Testing Framework

Configuration

- Jest can be configured through a **jest.config.js** file in the root directory. This file allows you to specify various settings, such as the test environment and verbosity of the output.

Writing Tests

- Tests can be organized in a **__tests__** directory or can be placed alongside the code in files fileName**.js**. Each test file should focus on a specific functionality or module within the API.

Test Table

Test ID	Test Description	Category	Inputs	Expected Result	Result
1	Verify deletion of a non-existing To-Do	Negative Test	todoId = null, DELETE request	Returns 404 status	Pass
2	Verify retrieval of a non-existing To-Do	Negative Test	todoId = null, GET request	Returns 404 status	Pass
3	Attempt to create a To-Do with missing required fields	Validation Test	POST request, body = {}	Returns 422 status	Pass
4	Attempt to update a non-existing To-Do	Negative Test	todoId = null, PUT request with body = { title: "", status: "" }	Returns 404 status	Pass
5	Attempt to create a To-Do with a duplicate title	Validation Test	POST request, title = "Test To-Do"	Returns 422 status	Pass
6	Attempt to create a To-Do with invalid status	Validation Test	POST request, status = "invalid"	Returns 422 status	Pass
7	Attempt to create a To-Do without authentication	Security Test	POST request without Authorization header	Returns 401 status	Pass
8	Attempt to create a To-Do with an empty title	Validation Test	POST request, title = ""	Returns 422 status	Pass
9	Attempt to create a To-Do with an invalid user ID	Validation Test	POST request, user_id = 99999	Returns 422 status	Pass

10	Attempt to create a To-Do with too long title (256 characters)	Boundary Test	POST request, title = "a".repeat(256)	Returns 422 status	Pass
11	Attempt to create a To-Do with empty status	Validation Test	POST request, status = ""	Returns 422 status	Pass
12	Attempt to create a To-Do with invalid status (random string)	Validation Test	POST request, status = "random"	Returns 422 status	Pass
13	Successfully create a user	Functional Test	POST request, valid name, gender, email, status	Returns 201 status, user details returned	Pass
14	Retrieve details of an existing user	Functional Test	userId, GET request	Returns 200 status, user details match	Pass
15	Update details of an existing user	Functional Test	PUT request, name = "Updated Test User", email = new unique email	Returns 200 status, updated user details returned	Pass
16	Retrieve updated user details	Functional Test	userId, GET request	Returns 200 status, updated user details match	Pass
17	Delete an existing user	Functional Test	userId, DELETE request	Returns 204 status	Pass

18	Verify deletion of a user	Negative Test	userId, GET request	Returns 404 status	Pass
19	Attempt to create a user with missing required fields	Validation Test	POST request, body = {}	Returns 422 status	Pass
20	Attempt to update a user with missing fields	Validation Test	PUT request, body = { name: "", email: "" }	Returns 404 status	Pass
21	Attempt to create a user with duplicate email	Validation Test	POST request, email = "duplicateemail@example.com"	Returns 422 status	Pass
22	Attempt to create a user with invalid email format	Validation Test	POST request, email = "invalid-email-format"	Returns 422 status	Pass
23	Attempt to create a user without authentication	Security Test	POST request without Authorization header	Returns 401 status	Pass
24	Successfully create a user with inactive status	Functional Test	POST request, valid name, gender, email, status = "inactive"	Returns 201 status, user details returned	Pass
25	Attempt to create a user with too long name (256 characters)	Boundary Test	POST request, name = "a".repeat(256)	Returns 422 status	Pass
26	Attempt to create a user with minimum length email (5 characters)	Boundary Test	POST request, email = "a@b.c"	Returns 422 status	Pass
27	Attempt to create a user with	Boundary Test	POST request, email = "a".repeat(318) + "@example.com"	Returns 422 status	Pass

	maximum length email (320 characters)				

You can use this table as a reference for your test results and update the **Actual Result** column after running each test case.

Running Tests

- To execute the tests, you can use the command:

➢ npm test
OR
➢ jest

- This command will run all the tests defined in your project and provide a summary of the results, including which tests passed or failed.

k6 Load Testing

Configuration

- k6 requires minimal configuration. You can create a JavaScript file (e.g., **load-test.js**) where you define your load testing scenarios, including the number of virtual users and the duration of the test.

Writing Load Tests

- Load tests in k6 allow you to simulate multiple users interacting with your API simultaneously. This helps to assess the performance and reliability of your API under stress.

Scenarios

Scenario 1: GET Users

Description

This scenario tests the ability of the API to handle multiple concurrent requests for retrieving a list of users.

Code Overview

- **Virtual Users (VUs)**: 30
- **Duration**: 20 seconds
- **Endpoint**: **/users**
- **Authorization**: A Bearer token is used for authentication.

Objectives

- Verify that the API returns a successful response (HTTP status 200).
- Ensure that the response time is within acceptable limits (less than 500 milliseconds).

Performance Checks

- **Status Check**: Confirms that the response status is 200.
- **Response Time Check**: Ensures that the response time is less than 500 milliseconds.

Scenario 2: DELETE User

Description

This scenario tests the API's ability to handle concurrent delete requests for a specific user.

Code Overview

- **Virtual Users (VUs)**: 30
- **Duration**: 20 seconds
- **Endpoint**: **/users/{userId}** (where **userId** is a placeholder for the actual user ID)
- **Authorization**: A Bearer token is used for authentication.

Objectives

- Verify that the API successfully deletes a user and returns a successful status code.

Performance Checks

- **Status Check**: Confirms that the response status is in the range of 200 to 299, indicating a successful deletion.

Scenario 3: POST User

Description

This scenario tests the API's ability to handle concurrent requests for creating new users.

Code Overview

- **Virtual Users (VUs)**: 20
- **Duration**: 15 seconds
- **Endpoint**: **/users**
- **Authorization**: A Bearer token is used for authentication.
- **Payload**: A JSON object containing random user data (name, gender, email, and status) is sent in the request body.

Objectives

- Verify that the API successfully creates a new user and returns a status code of 201.

- Ensure that the response contains the newly created user's name and email.

Performance Checks

- **Status Check**: Confirms that the response status is 201, indicating successful creation.
- **Response Content Check**: Verifies that the response contains the **name** and **email** fields.

Scenario 4: GET Posts

Description

This scenario tests the API's ability to handle multiple concurrent requests for retrieving a list of posts.

Code Overview

- **Virtual Users (VUs)**: 25
- **Duration**: 15 seconds
- **Endpoint**: **/posts**
- **Authorization**: A Bearer token is used for authentication.

Objectives

- Verify that the API returns a successful response (HTTP status 200).
- Ensure that the response time is within acceptable limits (less than 400 milliseconds).
- Confirm that the response body contains data.

Performance Checks

- **Status Check**: Confirms that the response status is 200.
- **Response Time Check**: Ensures that the response time is less than 400 milliseconds.
- **Content Check**: Verifies that the response body contains content (i.e., it is not empty).

Running Load Tests

- To run your load test, use the command:

➢ k6 run file-name.js

- This command will execute the load test as defined in your JavaScript file and provide real-time feedback on the performance of your API.

Results Interpretation

After running your tests, both Jest and k6 will provide output that summarizes the results:

Jest Results

- The output will indicate the number of tests passed, failed, and skipped.
- Detailed information about any failed tests will be provided, including the expected and actual results.

k6 Results

- k6 will output metrics such as:
    - **Requests per second**: Indicates how many requests were handled by the API per second.
    - **Response times**: Shows the average, minimum, maximum, and percentiles of response times.
    - **Error rates**: Displays the percentage of requests that resulted in errors.
- These metrics help you understand how well your API performs under load and identify potential bottlenecks.