Solution of Distribusion Assessment

Test Strategy

Following the important key factors involved in test strategy.

1) Scope:

The testing scope encompasses the verification of critical functionalities related to Gist creation, retrieval, editing, and deletion through both the API and web interface.

2) Testing Objectives:

- Ensure the successful creation, retrieval, editing, and deletion of Gists.
- Validate the correctness of responses from both the API and web interface.
- Verify proper handling of edge cases and negative scenarios.

3) Test Levels:

- **Unit Testing:**

  Validate individual components responsible for Gist creation, retrieval, editing, and deletion.

- **Integration Testing:**

  Confirm seamless interaction between Gist functionalities and external components.

- **System Testing:**

  Verify end-to-end Gist management, including API and web interface interactions.

4) Test Types:

Functional Testing:

Ensure the intended functionalities of Gist creation, retrieval, editing, and deletion.

- **Test Case: Verify Gist Creation**
  Verify that a Gist is successfully created with valid parameters.

Test Case ID	GI_01	**Test Case Description**	Verify Gist Creation		
Created By	Raza	**Reviewed By**		**Version**	

QA Tester's Log

Tester's Name	Raza	Date Tested	29-10-2023	Test Case (Pass/Fail/NotExecuted)	Pass

S #	Prerequisites:
1	Ensure a valid GitHub API token is available.
2	Navigate to the Gist creation endpoint.
3	
4	

S #	Test Data
1	Github Token
2	

Test Scenario Verify filter condition is correctly displaying

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed / Suspended
1	Send a POST request to the Gist creation endpoint with valid parameters and get 201 response.	User should able to do request	User is performing post request with valid parameter and get 201 response	Pass
2	Retrieve the created Gist using the Gist ID from the response.	System should create GIST id from response.	User should be able see GIST id from response.	Pass
3	Validate that the retrieved Gist matches the expected format with accurate information like description and content.	User is able to get matches result with the detail information	User is able to validate actual and expected result with accurate content information. New Gist is created with expected result.	Pass

- **Test Case: Confirm Gist Retrieval**
  Validate that the retrieved Gist matches the expected format and contains accurate information.

  Preconditions:

- An existing Gist ID is available.
  Steps:

- Send a GET request to the Gist retrieval endpoint with the existing Gist ID.
- Verify that the response status code is 200.
- Confirm that the retrieved Gist matches the expected format.
- Validate that the Gist contains accurate information.

  Expected Result:

- The Gist is successfully retrieved with accurate information.

  Actual Result:

- User is able to retrieved the result with actual information.

- **Test Case: Validate Gist Editing**
  Validate the ability to edit an existing Gist and see changes reflected in both API responses and the web interface.

  Preconditions:

- An existing Gist ID is available.
- Prepare updated data for the Gist.
  Steps:

- Send a PATCH request to the Gist editing endpoint with the existing Gist ID and updated data.
- Verify that the response status code is 200.
- Retrieve the edited Gist using the Gist ID.
- Confirm that the retrieved Gist reflects the applied changes.
- Validate that the Gist contains accurate information, including the updated details.

  Expected Result:

- The Gist is successfully edited, and changes are reflected in the retrieved data.

- **Test Case: Test Gist Deletion**
  Verify the deletion of a Gist to ensure proper handling by the API and web interface.

  Preconditions:

- An existing Gist ID is available.
  Steps:

- Send a DELETE request to the Gist deletion endpoint with the existing Gist ID.
- Verify that the response status code is 204.
- Attempt to retrieve the deleted Gist.
- Confirm that the response status code is 404.
  Expected Result:

- The Gist is successfully deleted, and subsequent retrieval results in a 404-status code.

Non-Functional Testing:

Validate system performance, security, and reliability under varying conditions.

Performance Testing:

Performance testing is conducted to assess the system under varying conditions. This includes creating and retrieving a large number of Gists with parallel requests. Specific performance metrics will be defined, such as response times and system resource usage.

- **Test Case: Assess System Performance**
  Validate the system's performance by creating and retrieving a large number of Gists.

  Preconditions:

- A valid GitHub API token is available.
- The system is in a stable state.
  Steps:

- Use a loop to iteratively create a significant number of Gists (e.g., 100 or more) using parallel requests.
- Record the time taken for Gist creation in each iteration.
- Verify that the Gists are successfully created without errors.
- Use a loop to iteratively retrieve the created Gists in parallel.
- Record the time taken for Gist retrieval in each iteration.
- Verify that the retrieved Gists match the expected format.

Expected Result:

- The system can handle a large number of parallel requests for Gist creation and retrieval without significant degradation in performance.

- **Test Case: Verify Authentication Mechanism**
  Verify that the authentication mechanism restricts unauthorized access effectively.

  Preconditions:

- A valid GitHub API token is available.
- An invalid token is generated for unauthorized access.
  Steps:

- Attempt to create a Gist using a valid API token.
- Verify that the Gist creation is successful.
- Attempt to create a Gist using an invalid or expired API token.
- Verify that the request is denied with an appropriate authentication error.
- Attempt to retrieve a Gist using a valid API token.
- Verify that the Gist retrieval is successful.
- Attempt to retrieve a Gist using an invalid or expired API token.
- Verify that the request is denied with an appropriate authentication error.
  Expected Result:

- The authentication mechanism allows access with a valid token and denies access with an invalid or expired token.

- **Test Case: Evaluate System Reliability**
  Evaluate system reliability by stress-testing Gist deletion with multiple parallel requests.

Security Testing:

Security testing is expanded to cover proper authorization for Gist operations, testing for common security vulnerabilities like SQL injection, and verifying the system's behavior when faced with unauthorized access.

Regression Testing:

Confirm that new updates or features do not negatively impact existing functionalities.

- **Test Case: Re-run After Updates**
  Re-run functional tests after an update to ensure Gist creation, retrieval, editing, and deletion remain unaffected.

- **Test Case: Verify Existing Functionalities**
  Verify that existing functionalities work as expected after introducing new features.

5) Testing Techniques:

Utilize boundary value analysis to assess the system's behavior at the limits.

- **Test Case: Minimum Content Length**
  Create a Gist with the minimum allowed content length.

- **Test Case: Maximum Content Length**
  Attempt to create a Gist with content exceeding the maximum allowed length.

Equivalence partitioning to test Gist functionalities with different input scenarios.

- **Test Case: Valid and Invalid Characters**
  Test Gist creation with valid and invalid characters in the content.
- **Test Case: Retrieving a Non-Existing Gist**
  Check the system's response when attempting to retrieve a non-existing Gist.

6) Test Environment:

I have tested this feature on Stagging environment and I have ensured that all the bugs are fixed and deployed on this environment. After regression testing I have performed the release UAT testing on UAT environment. UAT is replica of the production. I ensured that before releasing to production all the bugs, scenarios are fixed and covered. Then will release to the production for actual users to use this feature.

7) Testing tools & Automation:

I have used Jira, Zephyr, X-rays for writing efficient test cases. For automation, I have used Mocha framework with node module depenedencies.

Automation testing can increase the depth and scope of tests to help improve software quality. Test automation can easily execute thousands of different complex test cases during every test run providing coverage that is impossible with manual tests. In this project proper estimated number of user test case (end to end flow) to be automated so that in every regression QA need to execute this test case for improving the quality project.

These are the tool I have used while automating this feature,

- Visual Studio
- npm
- Node.js Environment
- Mocha and Axios API Framework

Thanks!