CPSC 319 Zenith Blog Test Plan

Presented By:

Muhammad Saad Shahid
Param Tully
Ruchir Malik
Shawn Zhu
Andrew Liu
Anusha Saleem
Cheryl Yu
Eric Wong
Anthony Baek

Sponsors:

Evan Seabrook
Alvin Madar

TA:

Asem Ghaleb

Version 1.2.1
Date: Mar/8/2023

t

TABLE OF CONTENTS

A. Summary

Our testing approach involves a combination of tools, automation, manual testing, and third-party help. Specifically, we use JUnit 5.9.2 and Cypress 12.1.0 as our primary testing tools. Our test plan consists of automated JUnit test scripts and mostly manual test scripts that rely on manual testers and third-party help from Cypress. We also plan on having a regression pack set in place to ensure that features that worked before don't break as development progresses.

To apply a risk-based approach, we prioritize test scripts based on the risks of features failing from user interaction and the effects of core functionality failing for clients/users. This allows us to focus our testing efforts on the most critical areas and ensure that we catch any issues that could have a severe impact.

We anticipate running 3 test cycles for system acceptance testing. With the amount of time remaining before we have to ship our system, we plan on having around 3 code drops. On our first cycle, the criterion to move forwards would be to just have working core functionalities that don't block the ability to run other test scripts. On our second test cycle, we will add additional requirements to our first criterion such as having at least a majority of core features functioning correctly along with a couple of the more minor features. Finally, with the last test cycle, our criterion would be ideally to have little to no errors. However, since there is a deadline to follow, we will prioritize having no core/critical functionalities failing as a bare minimum.

In summary, our testing approach is a mix of tools, automation, manual testing, and third-party help. We prioritize our testing efforts based on a risk-based approach to ensure that we focus on the most critical areas.

B. Functional Test Plan

User Related Feature Test Plan

1. User Login Component	
Testing technique:	Black box testing + end-to-end testing
Tool used:	Cypress Testing
Risk:	The risk of using black box testing is that some of the internal code could be missed.
Risk Mitigation:	Since login is a simple component and Google sign-in API is well-developed, the risk is minimal.
Scripts(more detail in appendix):	Number of anticipated scripts: 3. Happy path - Logging in with correct credentials Negative path - Logging in with invalid credentials Negative path - Logging in with invalid input format

2. User Registration Component	
Testing technique:	Black box testing + end-to-end testing
Tool used:	Cypress Testing
Risk:	The risk of using black box testing is that some of the internal code could be missed.
Risk Mitigation:	Since registration is a simple component and Google sign-in API is well-developed, the risk is minimal. Manual test scripts can be prone to human error, so careful attention should be paid to the test cases.
Scripts(more detail in appendix):	Number of anticipated scripts: 4. Happy path - Registering a new user successfully Negative path - Registering with an already existing username or email Negative path - Registering with a weak password Negative path - Registering with mismatched passwords

3. After login, do all the pages afterwards maintain login status !!!	
Testing technique:	Black box testing & Manual test scripts will be used for testing
Tool used:	None (manual)
Risk:	Test flakiness: The test may fail intermittently due to factors such as race conditions, network latency, or system load, which could lead to uncertainty and false positives.
Risk Mitigation:	The tests will be conducted multiple times, by different testers and on different devices to minimize such a risk.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

Frontend Commenting Feature Test Plan

4. Whether the frontend reflects an update after editing a post	
Testing technique:	Black box & manual testing
Tool used:	None (manual)
Risk:	If the update functionality is not implemented properly, there is a risk of the update not being reflected in the frontend.
Risk Mitigation:	Backend unit and integration tests will be conducted at the same time. Combining these test results, the actual error will be detected.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

5. **Verify that all comments are visible to other users and that the interaction, such as commenting is supported**	
Testing technique:	White Box & Functional Testing
Tool used:	None (manual)
Risk:	potential data loss or corruption if the blog post record is not removing correctly from the database
Risk Mitigation:	Implement thorough input validation and error handling to catch any issues.
Scripts(more details in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

6. **Rich text editor representing info correctly**	
Testing technique:	Black box & manual testing
Tool used:	None (manual)
Risk:	- Risk of data corruption or loss if the rich text editor does not accurately represent information entered by the user. - Risk of user frustration or confusion if the formatting of their posts is not displayed correctly.
Risk Mitigation:	- Provide clear and concise instructions for using the rich text editor and formatting options - Regularly test and verify the accuracy of the rich text editor with a wide range of input data, including edge cases and special characters
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

Backend Features Test Plan

7. Posting a Comment	
Testing technique:	White box & automatic testing
Tool used:	Junit
Risk:	The risk of using automated testing is that it may not catch all edge cases or unexpected behavior.
Risk Mitigation:	With careful planning and execution of test cases, this risk can be minimized.
Scripts(more detail in appendix):	Number of anticipated scripts: 3. Happy path - Posting a comment successfully Negative path - Posting a comment with invalid input format Negative path - Posting a comment without being logged in

8. Deleting a Comment	
Testing technique:	White box & automatic testing
Tool used:	Junit
Risk:	The risk of using automated testing is that it may not catch all edge cases or unexpected behavior.
Risk Mitigation:	With careful planning and execution of test cases, this risk can be minimized.
Scripts(more detail in appendix):	Number of anticipated scripts: 3. Happy path - Deleting a comment successfully Negative path - Deleting comment without permission Negative path - Deleting a comment that does not exist

9. Creates a new record in the database	
Testing technique:	White box & automatic testing This scenario may involve system testing or acceptance testing, as it involves testing the entire system to ensure that the frontend and backend are integrated and working correctly.
Tool used:	JUnit
Risk:	Some of the risks associated with this scenario include potential data loss or corruption if the new blog post record is not added correctly to the database, as well as potential performance or scalability issues if the database is not able to handle a large volume of new records.
Risk Mitigation:	Implement thorough input validation and error handling to catch any issues with the new blog post data before it is added to the database; cloud-based database service will handle the high volume of new records.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Happy Path - A user creating a new blog post with a title body, and tags Negative Path - A user attempting to log in with invalid or incorrect credentials

10. Removes records from the database using soft deletion	
Testing technique:	White Box & Functional Testing
Tool used:	JUnit
Risk:	- Security vulnerability: If the deleted data is not securely removed from the system, it may lead to a security vulnerability that can be exploited by attackers. - Inconsistency: If the backend does not remove the data consistently after the delete operation, it can

	lead to data inconsistencies, which can cause further issues down the line.
Risk Mitigation:	Implement thorough input validation and error handling to catch any issues with the data removal after it is removed from the database
Scripts(more detail in appendix):	Number of anticipated scripts: 3. Happy path - Backend successfully removes the data associated with the deleted blog post. Negative Path - User verifies that the blog post is still visible on the website. Negative Path - Backend fails to remove the data associated with the deleted blog post.

11. Correctly updates records that have been edited	
Testing technique:	White Box & Database Testing
Tool used:	JUnit
Risk:	- Data inconsistency: The edited post's data may not get updated correctly in the database, leading to data inconsistency. - Data loss: The data in the edited post may get lost during the updating process, leading to data loss.
Risk Mitigation:	Implementing thorough testing: To prevent data inconsistencies or loss, it is important to thoroughly test the system before it is deployed to ensure that all components are working as expected.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Happy Path - The user can see the updated post with the new data Negative Path - The application fails to update the post in the database with the new data

12. CI/CD Slack bot	
Testing technique:	White Box & Automatic Testing
Tool used:	Slack API
Risk:	Security vulnerabilities: If the Slack bot is not properly secured, it may be vulnerable to attacks from malicious actors. This can lead to data breaches and other security incidents. Therefore, it is important to implement appropriate security measures, such as encryption and access controls, to mitigate these risks.
Risk Mitigation:	This project does not have a lot of secure data, and every one of the team members will double check with staff before doing anything that could possibly trigger such a problem.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Happy Path - The Slack bot is properly integrated with the testing framework. Negative Path - Automated tests fail to execute correctly.

C. Non Functional testing

13. Timeout	
Testing scenario	Test the timeout of the blog application when a user is trying to perform a long-running task, such as publishing a post with a large amount of content or uploading a large file.
Testing technique	System & Performance Test
Tool used:	JUnit
Risk:	Performance degradation: Load testing can put a strain on the system resources and impact the performance of the application, causing slower response times or even system crashes.
Risk Mitigation:	Set up time out thresholds to prevent infinite loop or abnormal situations.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

14. Concurrency	
Testing scenario	Blog: Testing whether the blog application can handle multiple users editing and updating posts at the same time without conflicts. CI/CD pipeline: As non-functional requirements state by sponsor. We will have 6 users push their code at the same time to test the concurrency capability.
Testing technique	The testing technique used in this case would be black box testing, as the focus is on testing the functionality of the application from an end-user's perspective.
Tool used:	Manual
Risk:	Performance risks: If the application is not optimized for handling multiple concurrent requests, it may slow down or crash.

Risk Mitigation:	Conduct load testing to identify the maximum capacity of the system and to ensure that it can handle multiple concurrent requests.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

15. Performance	
Testing scenario	Test the performance of the blog application by measuring the response time for various user actions, such as logging in, creating a post, commenting, and browsing the site.
Testing technique	The testing technique used in this case would be performance testing, which involves measuring the response time and resource utilization of the application under different load conditions.
Tool used:	Manual
Risk:	System overload: Load testing can put a strain on the system resources and cause it to become overloaded, leading to degraded performance or even system crashes.
Risk Mitigation:	Conduct load testing to identify the maximum capacity of the system and to ensure that it can handle multiple concurrent requests.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

16. Load Balancing	
Testing scenario	Essentially a performance test. involves simulating a high load on the system to verify that the system can handle a large number of requests by utilizing load balancing.
Testing technique	same as above.

Tool used:	Manual
Risk:	Configuration errors: Load balancing requires careful configuration to ensure that requests are distributed evenly and efficiently among the instances of the application. Configuration errors can result in uneven load distribution or performance degradation.
Risk Mitigation:	Test the load balancing configuration thoroughly before deploying it in a production environment.
Scripts(more detail in appendix):	Number of anticipated scripts: 2. Detailed Steps are available in Appendix.

D. Test scripts

The approaches used for our test scripts for both functional and non-functional tests are included in the respective anticipated scripts section for each component in sections B and C for ease of reading.

Automatic test script link: https://github.com/CPSC319-2022/zenith/blob/main/blog/Dockerfile
To run the automatic unit & integration test, enter ./mvnw test.

All further details have been included in the appendix, which can be found near the end of the document.

E. Apply security testing and Test Data approach

A big portion of our system's security will be offloaded to a third party as we use Google OAuth for user authentication and authorization. Data security regarding the blog will be maintained with TLS encryption. Thus, in general, security testing will not be needed regarding those aspects of the system. We will need to test the administrative function of being able to give a user the contributor status. We will be using a manual testing approach with made up test data (fake users and a fake administrator account).

As for the pipeline, sensitive data will be hidden behind github and GCP. The only areas of information being potentially leaked would be if we were to implement the stretch goal of the live dashboard. However, the dashboard can be kept safe behind a Google OAuth.

F. Regression testing needs

We will be adding scripts that involve posting comments, user login, and creating blog posts to the regression pack as they are high traffic paths and core functionality of our system. As we are still discussing formats for JSON objects being sent from frontend to backend and vice versa, there could be broken functionality as development progresses. Additionally, the login functionality also suffers from a similar issue as further development could involve other third party services, so adding it to the regression pack will help ensure that the key feature continues to function properly. Thus, having scripts related to comments and blog posts in the regression pack will help us detect these issues early on.

Appendix

Functional Tests

User Related Feature Test Plan

1. **User Login Component**

Happy path - Logging in with correct credentials
Precondition: The user navigate to the login page
Input: user entered valid username and password and click the login button
Mock testing user: user name: test_user/ password:abc123456
Output: none
Postcondition: Verify that the user is successfully logged in and redirected to the blog home page

Negative path - Logging in with invalid credentials
Precondition: User navigate to the login page
Input: user enter invalid username and/or password and click the login button
Example: username: wrong_user/ password :wrong123456
Output: error message: login credentials are incorrect
Postcondition: Verify that an error message is displayed indicating that the login credentials are incorrect

Negative path - Logging in with invalid input format
Precondition: User navigate to the login page
Input: user enter invalid username and/or password and click the login button
Example: username: #!$@ abd / password : 42 <random emoji>
Output: error message: Wrong format for username, should only contain number letter or underscore.
Postcondition: Verify that an error message is displayed indicating that the login format are incorrect with correct format guidance

2. **User Registration Component**

Happy path - Registering a new user successfully

Precondition: The user navigate to the registration page
Input: user entered a valid username, email, password, confirm password, and click the register button
Mock testing user: user name: test_user/ email: test_user@example.com / password:abc123456 / confirm password:abc123456
output: none
Postcondition: Verify that the user is successfully registered and redirected to the login page

Negative path - Registering with an already existing username or email
Precondition: The user navigate to the registration page
Input: user entered a username or email that already exists in the database and click the register button
Mock testing user: user name: existing_user/ email: existing_user@example.com / password:abc123456 / confirm password:abc123456
output: error message: Username or email already exists.
Postcondition: Verify that an error message is displayed indicating that the username or email already exists in the database

Negative path - Registering with a weak password
Precondition: The user navigate to the registration page
Input: user entered a weak password (less than 8 characters) and click the register button
Mock testing user: user name: test_user/ email: test_user@example.com / password:abc / confirm password:abc
output: error message: Password must be at least 8 characters long.
Postcondition: Verify that an error message is displayed indicating that the password is too weak

Negative path - Registering with mismatched passwords
Precondition: The user navigate to the registration page
Input: user entered a password and confirm password that do not match and click the register button
Mock testing user: user name: test_user/ email: test_user@example.com / password:abc123456 / confirm password:abcdefg
output: error message: Passwords do not match.
Postcondition: Verify that an error message is displayed indicating that the passwords do not match

3. After login, does all the pages afterwards maintain login status

Happy path

Conditions:

1. User enters valid login credentials and successfully logs in to the blog application.
2. User navigates to various pages of the blog application and remains logged in on each page.

Output: User logs out of the blog application and is redirected to the login page.

Negative path

Conditions:

1. User enters invalid login credentials and is unable to log in to the blog application.
2. User logs in to the blog application, but is redirected to the login page when navigating to a different page.

Output:

User logs out of the blog application, but is still able to access pages without being prompted to log in again.

User tries to log in with special characters or incorrect formatting in the login credentials, and is unable to log in.

Frontend Commenting Feature Test Plan

1. **Whether the frontend reflects an update after editing a post**

Happy Path:

Conditions:

1. Log in to the blog application and navigate to the list of blog posts.
2. Select a post to edit and make changes to the content.
3. Save the changes and verify that the frontend of the blog application reflects the update.
4. Refresh the page and verify that the changes persist.

Output: Repeat the above steps with multiple posts to ensure consistency.

Negative Path:

Conditions:

1. Log in to the blog application and navigate to the list of blog posts.
2. Select a post to edit and make changes to the content.
3. Save the changes and verify that the frontend of the blog application does not reflect the update.

4. Check the backend to ensure that the update was saved properly and investigate any issues.
5. Repeat the above steps with multiple posts to ensure consistency.

Output:

Edit a post and intentionally introduce an error or invalid input into the content.

Save the changes and verify that the frontend of the blog application does not reflect the update.

Verify that the edit is not allowed and that the frontend of the blog application does not reflect the update.

2. Verify that all comments are visible to other users and that the interaction, such as commenting is supported

Happy Path:

Conditions:

1. Log in to the blog application and navigate to a blog post with existing comments.
2. Verify that all comments are visible to the current user.
3. Leave a new comment and verify that the frontend of the blog application reflects the interaction.
4. Log out and log in with a different user account.
5. Navigate to the same blog post and verify that all comments, including the new comment, are visible to the new user.

Output: Repeat the above steps with multiple blog posts and user accounts to ensure consistency.

Negative Path:

Conditions:

1. Log in to the blog application and navigate to a blog post with existing comments.
2. Verify that some comments are not visible to the current user.
3. Investigate any issues with user permissions or visibility settings.
4. Attempt to leave a comment while not logged in or without proper permissions.
5. Verify that the comment is not allowed and that the frontend of the blog application does not reflect the interaction.
6. Leave a comment with invalid input or intentionally introduce an error into the comment.

Output:

Verify that the frontend of the blog application does not reflect the interaction and investigate any issues with error handling or input validation.

3. Rich text editor representing info correctly

Happy Path:

Conditions:

1. Enter text with various formatting options (bold, italic, underline, bullet points, numbered lists, etc.) into the rich text editor.
2. Verify that the post is displayed with the correct formatting on the detail page.
3. Edit the post and make additional changes to the formatting of the text.

Output: Verify that the post is still displayed with the correct formatting on the detail page.

Negative Path:

Conditions:

1. Enter text with invalid or unsupported formatting options into the rich text editor.
2. Verify that the post is not displayed with the incorrect formatting on the detail page.
3. Edit the post and make additional changes to the formatting of the text with invalid or unsupported formatting options.

Output: Verify that the post is not displayed with the incorrect formatting on the detail page.

Verify that the post is displayed with the correct formatting on the detail page, and that it is not excessively slow to load or display.

BACKEND

1. Posting a Comment

Happy path - Posting a comment successfully

Precondition: User is logged in and on a blog post page

Input: User types a comment and clicks "Post Comment" button

Mock testing user: user_id: test_user, post_id: 12345, comment: "Great post!"

Output: Verify that the comment is successfully posted and displayed on the page

Negative path - Posting a comment with invalid input format
Precondition: User is logged in and on a blog post page
Input: User types a comment with invalid format and clicks "Post Comment" button
Example: comment contains special characters or exceeds the character limit
Mock testing user: user_id: test_user, post_id: 12345, comment: "<script>alert('test');</script>"
Output: Verify that the comment is not posted and an error message is displayed indicating that the input format is incorrect

Negative path - Posting a comment without being logged in
Precondition: User is not logged in and on a blog post page
Input: User types a comment and clicks "Post Comment" button
Mock testing user: user_id: null, post_id: 12345, comment: "This post is great!"
Output: Verify that the comment is not posted and an error message is displayed indicating that the user must be logged in to post a comment

2. Deleting a Comment

Happy path - Deleting a comment successfully
Precondition: User is logged in and has previously posted a comment on a blog post page
Input: User clicks the "Delete" button next to their comment
Mock testing user: user_id: test_user, post_id: 12345, comment_id: 67890
Output: Verify that the comment is successfully deleted and no longer displayed on the page

Negative path - Deleting comment without permission
Precondition: User is logged in and another user has posted a comment on a blog post page
Input: User clicks the "Delete" button next to another user's comment
Mock testing user: user_id: test_user, post_id: 12345, comment_id: 54321
Output: Verify that the comment is not deleted and an error message is displayed indicating that the user does not have permission to delete the comment

Negative path - Deleting a comment that does not exist
Precondition: User is logged in and on a blog post page
Input: User clicks the "Delete" button next to a comment that does not exist
Mock testing user**:** user_id: test_user, post_id: 12345, comment_id: 99999
Output: Verify that the comment is not deleted and an error message is displayed indicating that the comment could not be found

3. Creates a new record in the database

Happy Path - A user creating a new blog post with a title, body, and tags

Precondition:

The user has a valid account with the blog application and is logged out.

Input:

The user enters their username and password into the login form to log in. The user enters a title, body, and tags for the new blog post.

Mock testing user:

The mock testing user is a registered user with valid credentials. They enter their username and password into the login form to log in. After successfully logging in, they create a new blog post by entering a title, body, and tags into the appropriate fields on the new post creation form.

Output:

The user is successfully logged in and redirected to the blog dashboard page. The new blog post is successfully created and added to the database. The user is redirected to the blog post listing page, where they can see the newly created post in the list of posts.

Postcondition:

The user is logged in and has successfully created a new blog post. The database has been updated with the new blog post record, and the new post is displayed correctly on the frontend, including the title, body, and tags.

Negative Path - A user attempting to log in with invalid or incorrect credentials

Precondition:

The user is either not registered or is already logged in with invalid credentials. For the delete scenario, the user is attempting to delete a blog post that does not exist in the database.

Input:

For the login scenario, the user enters incorrect or invalid credentials (e.g., wrong username or password) into the login form. For the delete scenario, the user attempts to delete a blog post that does not exist in the database.

Mock testing user:

For the login scenario, the mock testing user is a registered user with valid credentials but enters incorrect or invalid credentials into the login form. For the delete scenario, the mock testing user is a user attempting to delete a non-existent blog post.

Output:

For the login scenario, the user is redirected to an error page indicating that their login attempt has failed. For the delete scenario, the user is redirected to an error page indicating that the blog post they attempted to delete does not exist in the database.

Postcondition:

For the login scenario, the user is still not logged in and cannot access the blog application. For the delete scenario, the blog post has not been deleted, and the user is returned to the blog post listing page. The database has not been modified.

4. **Removes records from the database using soft deletion**

Happy path - Backend successfully removes the data associated with the deleted blog post.
Precondition: The user has created at least one blog post.
Input: A request to delete a blog post created by the user.
Example:

- The user logs in to the blog application.
- The user navigates to a blog post they have created and selects the option to delete the post.
- The system prompts the user to confirm the deletion.
- The user confirms the deletion.
- The system removes the data associated with the deleted blog post from the backend database.
- The blog post is no longer visible on the website.
- The user receives a confirmation message indicating that the deletion was successful.

Output: The backend successfully removes the data associated with the deleted blog post.
Postcondition: The deleted blog post is removed from the system, and the user's data is updated to reflect the deletion.

Negative Path - User verifies that the blog post is still visible on the website.
Precondition: The user has created at least one blog post.
Input: A request to view the blog post that was previously deleted by the user.
Example:

- The user logs in to the blog application.
- The user navigates to a blog post they have created and selects the option to delete the post.
- The system prompts the user to confirm the deletion.
- The user confirms the deletion.
- The system attempts to remove the data associated with the deleted blog post from the backend database.
- An error occurs during the deletion process, causing the system to fail to remove the data.
- The blog post is still visible on the website.
- The user attempts to view the deleted blog post but is unsuccessful.
- The user receives an error message indicating that the deletion was not successful.

Output: The blog post is still visible on the website.
Postcondition: The deleted blog post remains visible on the website, and the user's data is not updated to reflect the deletion.

Negative Path - Backend fails to remove the data associated with the deleted blog post.
Precondition: The user has created at least one blog post.
Input: A request to view the blog post that was previously deleted by the user.
Example:

- The user logs in to the blog application.
- The user navigates to a blog post they have created and selects the option to delete the post.
- The system prompts the user to confirm the deletion.
- The user confirms the deletion.
- The system attempts to remove the data associated with the deleted blog post from the backend database.
- An error occurs during the deletion process, causing the system to fail to remove the data.
- The blog post remains visible on the website.
- The user receives an error message indicating that the deletion was not successful.

Output: The backend fails to remove the data associated with the deleted blog post.
Postcondition: The user may attempt to delete the blog post again or report the issue to the development team for further investigation.

5. **Correctly updates records that have been edited**

Happy Path - The user can see the updated post with the new data
Precondition: The user has successfully edited a post and saved the changes
Input: N/A
Example:

- Precondition: User is logged in to the blog application, the post exists in the database, and the post has been successfully edited and saved
  - Input: N/A
- Output: The application displays the updated post with the new data to the user, and the user can view the changes made to the post.

Output: The updated post is visible to the user with the new changes made by the user
Postcondition: The user can view the updated post with the new data

Negative Path - The application fails to update the post in the database with the new data
Precondition: The user has selected a post to edit
Input: The user makes changes to the post data
Example:

- Precondition: User is logged in to the blog application, the post exists in the database
- Input: User edits the post title to "My Updated Blog Post" and saves the changes
- Output: The application fails to update the post data in the database with the new data, displays an error message "Post update failed," and does not show the updated post with the new data to the user.

Output: The user is not able to view the updated post with the new data
Postcondition: The database contains the original post data, and the changes made by the user are not saved

CI/CD

Slack bot: the results of these tests should be made available to the software development team in a common Slack channel.
Happy Path **-** The Slack bot is properly integrated with the testing framework.
Precondition: The testing framework and Slack bot are set up and configured correctly.
Input: The automated testing framework executes the automated tests.
Example:

- The testing framework executes a suite of automated tests on the blog application.
- The Slack bot receives the test results from the testing framework.
- The Slack bot posts the test results, including information on the number of tests that passed and failed, to the specified Slack channel.
- The development team is able to view the test results in the Slack channel and take appropriate action if any issues are identified.

Output: The Slack bot receives the test results from the testing framework.
Postcondition: The Slack bot has successfully integrated with the testing framework.

Negative Path **-** Automated tests fail to execute correctly.
Precondition: The testing framework and Slack bot are set up and configured correctly.
Input: The automated testing framework executes the automated tests.
Example:

- The testing framework attempts to execute a suite of automated tests on the blog application.
- Due to an issue with the testing framework or the application itself, the automated tests fail to execute correctly.
- The Slack bot does not receive any test results from the testing framework.
- No test results are posted to the specified Slack channel.
- The development team is not informed of any issues with the application and is unable to take appropriate action.

Output: The Slack bot does not receive any test results from the testing framework.
Postcondition: The Slack bot has not received any test results from the testing framework due to the failed test execution.

Non-Functional Tests

Timeout

Happy path:

Condition:

1. User logs in to the blog application and creates a new post with a reasonable amount of content.
2. User submits the post for publishing and it is successfully published within the expected timeout period.

Output: User verifies that the post has been published and is visible on the frontend of the application.

Negative path:

Condition:

1. User logs in to the blog application and creates a new post with an extremely large amount of content.
2. User submits the post for publishing and waits for the process to complete.

Output: If the process takes longer than the expected timeout period, the test fails.
User verifies that the post has been published and is visible on the frontend of the application.

Concurrency

Happy path:

Condition:

1. User A logs in to the blog application as an authenticated user with sufficient permissions to edit posts.
2. User A opens a post for editing.
3. User B logs in to the blog application as an authenticated user with sufficient permissions to edit posts.
4. User B also opens the same post for editing.
5. User A updates the post with new content and saves the changes.
6. User B updates the post with different new content and saves the changes.
7. Both users verify that the changes have been saved in the backend.

Output: Both users reload the page and verify that the updated post appears in the frontend with the changes made by each user.

Happy path:

Input: 6 users push their repo to the pipeline at the same time

Output: The pipeline should function without break

Negative path:

Condition:

1. User A logs in to the blog application as an authenticated user with sufficient permissions to edit posts.
2. User A opens a post for editing.
3. User B logs in to the blog application as an authenticated user with sufficient permissions to edit posts.
4. User B also opens the same post for editing.
5. User A updates the post with new content and saves the changes.

Output:

User B also tries to update the post, but encounters a lock or error message indicating that the post is already being edited.

User B verifies that the changes made by User A have been saved in the backend, but the changes made by User B have not been saved.

User B reloads the page and verifies that the updated post appears in the frontend with the changes made by User A, but not the changes made by User B.

User A and User B repeat the above steps with User B making changes first and User A encountering the lock or error message.

Performance

Happy path:

Condition:

1. Simulate a realistic load on the blog application, such as a certain number of concurrent users or requests per second.
2. Test the response time for various user actions, such as logging in, creating a post, commenting, and browsing the site.
3. Measure the response time and resource utilization of the system for each action.
4. If the response time and resource utilization are within acceptable limits, the test passes.
5. If the response time and resource utilization are better than expected, the team can consider optimizing the system further to improve performance and scalability.

Output: Re-run the tests to verify that the optimizations have improved the performance.

Negative path:

Condition:

1. Simulate an unrealistic load on the blog application, such as an extremely large number of concurrent users or requests per second.
2. Test the response time for various user actions, such as logging in, creating a post, commenting, and browsing the site.
3. Measure the response time and resource utilization of the system for each action.

Output: If the response time and resource utilization are within acceptable limits, the test fails.

If the response time and resource utilization are worse than expected, the team needs to investigate and identify any performance bottlenecks or issues.

Load balance

Happy path:

Condition:

1. Multiple instances of the blog application are running and properly configured behind the load balancer.
2. The load balancer is distributing the requests evenly among the instances.

3. The system can handle a high load of requests without any errors or performance degradation.
4. The system scales up and down dynamically based on the load.

Output: All performance metrics are within acceptable limits.

Negative path:

Condition:

1. The load balancer is not properly configured and is not distributing the requests evenly among the instances.
2. One or more instances of the application are not running or are not properly configured.

Output: The system cannot handle the high load of requests and experiences errors or performance degradation.

The system does not scale up or down properly and cannot handle the load.

One or more performance metrics exceed acceptable limits, such as response time, CPU usage, memory usage, or network bandwidth.

The load balancer fails or becomes unavailable, causing the system to become unresponsive or experience performance degradation.

Pipeline Testing Guide

1. The pipeline is successfully created and set up without any errors or issues.

   Testing Technique: Manual Testing
   Tools used: GitHub Actions
   Risks:
   - Misconfiguration of the pipeline: may fail to run or skip important tests, leading to errors or issues.
   - Integration issues: Other parts of the system may not be even run.

   Risk Mitigation:
   Testing and validation: The pipeline should be tested thoroughly to ensure that all steps are included and configured correctly. Validation should also be performed to ensure that the pipeline is working as expected.

   Happy Path:
   Conditions:
   - There should be no errors while GitHub compiling and running “docker-compose.yml” file.
   - The pipeline is able to access the appropriate testing and deployment environments.

   Output:
   - All necessary test suites are included and run successfully.
   - The pipeline is able to generate accurate and comprehensive reports on the test results.
   - The pipeline is able to deploy the code to the appropriate environment(s) without any issues.

   Negative Path:
   Conditions:
   - The pipeline fails to set up due to errors or issues in the configuration or dependencies.
   - The pipeline is unable to access the necessary testing or deployment environments.

   Output:
   - Some of the necessary test suites fail to run successfully, leading to incomplete or inaccurate test results.
   - The pipeline is unable to deploy the code to the appropriate environment(s) due to errors or issues in the deployment process.
   - The pipeline generates incomplete or inaccurate reports on the test results.

2. Verify that the pipeline is triggered only when a branch is merged and not when a branch is created or updated.

   Testing Technique: Manual & Functional Test
   Tools used: GitHub, JUnit
   Risks: The pipeline may be triggered when a branch is created or updated, which can lead to unnecessary and time-consuming testing.
   Risk mitigation: Configure the pipeline to trigger only when a branch is merged.

   Happy Path (and Output):
   - A new branch is created or updated, but the pipeline is not triggered.
   - A branch is merged and the pipeline is triggered successfully.

   Negative Path (and Output):
   - The pipeline is triggered when a branch is created or updated, but it should only be triggered when a branch is merged.
   - The pipeline is not triggered when a branch is merged, indicating that there may be an issue with the pipeline setup or configuration.

3. Verify that the pipeline runs all the necessary tests and does not skip any tests due to misconfiguration or other issues.

   Testing Technique: Automated Tests
   Tools used: GitHub Actions, JUnit
   Risks: The risk of misconfigured tests that result in skipped or incomplete tests, leading to undetected bugs in the code.
   Risk mitigation:
   - Ensure that all tests are properly configured and thoroughly reviewed before being added to the pipeline.
   - Implement test coverage analysis tools to identify areas of the code that are not adequately covered by tests.

   Happy Path:
   Conditions:
   - The pipeline is configured correctly to run all the necessary tests.
   - All the necessary dependencies and resources are available for the pipeline to execute the tests.

   Output:
   - The pipeline runs all the tests specified in the test suite without skipping any of them.
   - The pipeline completes successfully, indicating that all tests have passed.

Negative Path:

Output:

- The pipeline is misconfigured and does not include all the necessary tests in the test suite.
- The pipeline encounters errors or fails to run due to missing dependencies or resources required for the tests.
- The pipeline skips certain tests due to errors or misconfiguration.
- The pipeline fails to complete or produces inconsistent results, indicating that some tests may have been skipped or failed.

4. Verify that the pipeline is set up to run on the correct repository and branch and does not accidentally run on a different repository or branch.

Testing Technique: Manual Testing
Tools used: GitHub
Risks: Risk of misconfiguration: the pipeline may be set up to trigger on the wrong repository or branch, leading to incorrect test results or wasting computing resources.
Risk mitigation: Double-check the pipeline configuration before running any tests, and ensure that it is set up correctly to run on the expected repository and branch.

Happy path:

Condition:

- The pipeline configuration specifies the correct repository and branch to run on.

Output:

- The pipeline is triggered on the correct repository and branch.
- The pipeline runs all necessary tests and completes successfully.

Negative path:

Output:

- The pipeline configuration specifies the wrong repository or branch to run on.
- The pipeline is triggered on the wrong repository or branch.
- The pipeline skips necessary tests or fails to complete due to misconfiguration or other issues.
- The pipeline runs on a different repository or branch, leading to confusion and potentially interfering with other development work.

5. Verify that the pipeline has proper access to all necessary resources, such as databases.

Testing Technique: Manual & Functional Testing
Tools used: GitHub, JUnit
Risks: The pipeline may not have the necessary permissions to access all required resources, resulting in test failures and inaccurate results.
Risk mitigation:
- Ensure that the pipeline has the necessary permissions to access all required resources.
- Create mock versions of external APIs or services to simulate their behavior during testing.

Happy Path:

Conditions:
- The pipeline has been configured to have access to all necessary resources.
- All necessary resources are available and accessible.

Output:
- The pipeline runs successfully and all tests pass.

Negative Path:

Output:
- The pipeline has not been configured to have access to all necessary resources.
- Database is unavailable or inaccessible.
- Database related functional tests are failing.
- The pipeline fails to run or some tests fail due to missing or inaccessible resources.

6. Verify that the pipeline logs are properly configured and provide sufficient information for debugging in case of failures or errors.

Testing Technique: Manual & Black Box Testing
Tools used: GitHub
Risks: Misconfigured pipelines may lead to missing tests, which can result in defects being missed and not caught early.
Risk mitigation: Ensure that the pipeline is properly configured to run all necessary tests and that there are no configuration errors or omissions.

Happy Path:

Condition:
- The pipeline logs are properly configured and provide detailed information about each step of the pipeline.

Output:

- The logs include information about the status of each test, such as whether it passed or failed.
- The logs provide information about the environment in which the tests were run, including the operating system, browser version, and other relevant details.

Negative Path:

Output:

- The pipeline logs are not properly configured, and do not provide sufficient information for debugging in case of failures or errors.
- The logs are missing important information, such as the status of individual tests or the environment in which the tests were run.
- The logs are difficult to read or understand, making it hard to identify and diagnose issues.

7. Pipeline called when multiple users merging several branches

Testing Technique: Manual & Integration Testing
Tools used: GitHub
Risks: Multiple users merging several branches simultaneously can result in increased complexity and potential conflicts that may affect the pipeline's functionality.
Risk mitigation: Use automation tools such as code review and quality checks to ensure proper branch merging and reduce human error.

Happy path:

Condition:

- Multiple users merge several branches into the main branch.

Output:

- Pipeline is triggered automatically for each merged branch.
- All necessary tests are run successfully without any conflicts or errors.
- Pipeline completes successfully and deploys the updated code to production.

Negative path:

Condition:

- Multiple users merge several branches into the main branch.
- Pipeline is triggered automatically for each merged branch.

Output:

- One or more of the necessary tests fail, causing the pipeline to fail.
- The code cannot be deployed to production, and the changes must be reviewed and fixed before attempting to merge again.
- conflicts and merge issues raised, resulting in code that is not properly integrated and tested.