COMP4110 - Project Description

Sherif Saad

January 14, 2023

Contents

1 System Overview and Requirements

Microblog is a web application built using the Flask framework in Python that allows users to create and share their blog posts and interact with other users. The application has a user subsystem that enables users to register, log in, and recover forgotten passwords. The purpose of a Microblog is to provide a platform for users to share their thoughts and experiences on different items/products with a community of like-minded individuals and to make it easy for users to interact and connect.

1.1 Key Requirements of the Systems

Here are the key requirements of the Microblog app

1. User registration and login: Users should be able to create an account and log in to the app to access its features.
2. Profile management: Users should be able to view and edit their profiles, including their name, profile picture, and bio.
3. Posting reviews: Users should be able to create and post reviews for different items/products.
4. Viewing reviews: Users should be able to view reviews posted by other users and filter them by product category.
5. Commenting: Users should be able to add comments to reviews posted by other users.
6. Notifications: Users should receive notifications when someone comments on their reviews or when someone likes their reviews.
7. Search: Users should be able to search for reviews and products by keywords.
8. User management: Admins should be able to manage user accounts, including the ability to block or delete accounts if necessary.
9. Follow/Unfollow: Users should be able to follow and unfollow other users and see their reviews on a personalized timeline.
10. Like/dislike: Users should be able to like and dislike reviews and see the likes/dislikes count for each review.
11. Sharing: Users should be able to share reviews on social media platforms, such as Facebook, Twitter, and LinkedIn.
12. Bookmark: Users should be able to bookmark reviews and products, so they can easily find them later.
13. User-generated tags: Users should be able to add tags to their reviews to make them more discoverable by other users.

1.2 Git Repository

The Micorblog web app Git repository is hosted on Github, you can access it by going to:
https://github.com/miguelgrinberg/microblog[1]

To start with the project, first fork the repo to create your own project. make sure to keep your repo private to your group members only and the course TAs and Instructor.

During the course you should follow the GitHub flow branching model and some of the best-practices related to task management. In practice, that means every feature or bug fix should have its own GitHub issue ticket with one feature branch created for each ticket being worked on with. When merging back to the master branch, create a Pull Request (PR) so that at least one other member of the group can perform a code review of the proposed changes. The reviewer(s) should evaluate the PR and propose improvements or alternate implementation if necessary. After the review process is done and at least one reviewer has approved the PR, it can be merged to the master branch and the GitHub issue can be closed.

Note: that no commit should be added directly to the master branch except for merges from feature branches as described above.

2 Project Learning Outcome and Deliverables

The project centers on applying advanced testing techniques and tools for checking and improving the quality of a web application system. The focus will be continuous integration (DevOps), automated functional testing, performance testing, and security testing.

The project's main objective is to learn, by applying the quality control techniques seen in class and your software engineering background (both in terms of design and coding), how to expose and fix reliability, performance, scalability issues and software vulnerabilities in modern software systems.

The objective of this document is to give an overview of the basic functionality of the system under testing as well as the amount of work that needs to be done to complete the different steps of the project. Below is a list of the project activities and tasks along with the due dates of the various deliverables.

3 Project Execution and Deliverable

The project/lab will be performed in teams of 5-6 students. Students must form their team at the beginning of the term and notify the lab TAs by sending an email to both TAs. Here are the TAs' emails:

- Nour Elkott (elkott@uwindsor.ca)
- Karan Kashyap (kashyapk@uwindsor.ca)

Deliverables are due on the dates listed below by 11:59 pm. The deliverables must be submitted through Brightspace. The labs will be marked by course TAs. You might be asked to discuss/present your submission in the lecture.

Important Note: Team formation is entirely the responsibility of the students. Neither the instructor (Dr.Saad) nor the TAs will be involved in dispute resolution between teammates. If you are unhappy with your teammates, you can change the group at the end of each deliverable. However, we will not arbitrate any disagreement over performance issues among teammates.

Project Deliverables

- (Due Jan 28, 2023) Requirements Specification (5%)
- (Due Feb 15, 2023) Continuous Integration Setup (10%)
- (Due Mar 2, 2023) Automated Testing & Continuous Integration (15%)
- (Due Mar 29, 2023) Performance & Scalability Testing (15%)
- (Due Apr 10, 2023) Security Testing (15%)

[1] This repo/project is originally created as educational tool to teach flask and web development. It is supplement with very clear blog and video tutorials

4 Part 1: Requirements Specification & Backlog Building

At the end of this phase, your group is expected to deliver a requirement specification report that documents the **user stories** of the application and to build a backlog of known issues based on the report and your knowledge acquired from the process.

This phase is divided into two tasks, one for each of the goals mentioned above. However, before starting the process, you should fork the project repository to serve as your group's Git repository.

4.1 Task 1: Requirements Specification

Provide a requirement specification report that documents the user stories of the application, which can assist in its quality control process. Once you can clone your fork of the project repository, open it using your favourite IDE, and run the application. Read the README.md file for detailed instructions.

Once the application runs, start interacting with it by trying the different functionalities offered and identifying all its features. Then separate each feature into one or more user stories that collectively describe every application capability. Each user story should be written in **Gherkin** following the best practices presented in class.

Note: that the user stories represent the system's expected behaviour regardless of any defect that might be present. Therefore, wear your Product Owner hat and describe features as you think they should be (given the time constraints of the course) and not as they are implemented. You don't need to mention any bugs and poorly designed aspects of the application in the report, but you should keep track of them for Task II below.

Finally, for each key requirement state if the requirement implemented (even if partially implemented or contain bugs) or it is missing from the Microblog Web App. Now think of each requirement as an epic that has 2 or more user's story.

4.2 Task 2: Backlog Building

This task aims to build a backlog of known issues based on your requirement specification report.

The process consists of creating a GitHub issue for every bug or poorly designed aspect of the application identified in Task I. As described above, each issue should have a description, a priority and an estimation. You should use the related stories from the report as the description since the stories describe the application's expected behaviour after the defect is fixed.

For the priority, consider the scope and importance of the defect and how much value will the fact of fixing the ticket provide to the users.

After filling the backlog with known issues, it is time to add tickets for new desirable application features. Create GitHub issues for the following tasks:

- **Two-Factor Authentication:** User must login using a two-factor authentication method (a PIN code send to the user mobile phone or a verification link send to his email address).
- **Archive Favourite Posts:** A user can archive a post by adding it to their favorite list, and if the original poster decides to delete the post, it will be deleted for all users except those who have it in their favorite list. Favorite lists are private and only visible to their owners.
- **A Missing Requirement:** propose a new feature for the Micorbog Web App This requirement can be small but should not be too trivial since it will be implemented by the group in Part 3 of the project.

Note: As before, the issues should be described by user stories. As for their priority, since those features are considered highly desirable, mark them as a high priority.

At this point, your backlog should have several tickets with varying priorities. In Part 3, you'll pick the most important ones to work on.

4.3 Deliverable

- The requirement specification report containing the user stories of both the already implemented and proposed features.
- The GitHub issues forming the backlog

5 Part 2: Continuous Integration Setup

The two main goals of this part are to configure Docker to facilitate development and set up a Continuous Integration (CI) server in anticipation of adding new features, creating automated tests and fixing defects in the application (in the next part). You will need to set up Jenkins and SonarQube servers as part of your CI setup.

5.1 Setup CI Pipeline

The next task consists of setting up a Jenkins CI Multibranch Pipeline to do the following after each push to the group's Git repository

- Build the production application.
- Scan the application using SonarQube.
- Run the integration tests, which involve two parallel tasks:
  - Deploying the application;
  - Running the Selenium integration tests.
- Notify the group should any step fail.

Note:: that the Jenkins server should already include every plugin required to accomplish the tasks below

1. Add a deploy key to Github and Jenkins;
2. Create an initial Jenkinsfile that only builds the application;
3. Create a Jenkins Multibranch Pipeline configured to pull the code from the GitHub repo and to use the application's Jenkinsfile;
4. Configure GitHub to securely notify the Jenkins server of every push to the server using an integration webhook resulting in the Pipeline running for the appropriate branch;
5. Add a SonarQube scan stage to the Pipeline, so the application's source code is analyzed after the build – **Note:** that at this stage, the SonarQube scan results don't need to be analyzed yet;
6. Add a stage to the Pipeline to deploy the application – **Note:** that by default, the Pipeline job will never finish while the application is running;
7. Add a stage to the Pipeline to run the application's integration test suite (Selenium tests) after the application finishes deployment but while it is still running;
8. Add a mechanism to finish the Pipeline job with the correct build result (that is, success if the application is deployed correctly and the tests pass or failure if either fails) once the tests are finished – **Note:** that the application doesn't need to continue running afterwards unless you want to test it afterwards;
9. Add a notification mechanism to inform at least one group member when any deployment steps fail. This can be accomplished in several different ways, such as by sending an email, posting to any external notification system or integrating with the group's Slack, if the group has one

5.2 Deliverable

- A step-by-step demonstration of the Docker container being deployed with screenshots;
- An explanation of (1) what are Docker Containers and Images, (2) what is the difference of purpose between the Docker and the Docker-compose tools and (3) how is the Microblog app image created, including which tools are involved in the process;
- A step-by-step description of the Jenkins and the GitHub setup with screenshots and an explanation for each configuration choice and action taken;
- he description of every change made to the application files in order to support the Continuous Integration process, including the Jenkinsfile and the Docker file(s);
- Screenshots of the SonarQube report and issues pages;

- Screenshots showing the build history, trends and other Jenkins reports;
- An example of a notification when a build, scan, deployment or test fails;
- A commentary on the utility of Continuous Integration (CI) for software development;
- A small proposal (one or two paragraphs) on how the CI structure could be improved or otherwise augmented in this scenario. For instance, mention how stages could be changed or reorganized and which tools could be added or better used, so the CI process is improved;
- A commentary on the GitHub flow model and a comparison with other commonly used branching models

6 Part 3: Automated Testing & Continuous Integration

his part consists of 2 separate sets of tasks. Although it is not required, you may conduct Tasks I & II in parallel. In this case, you can split the team so that some members can start fixing bugs and implementing the new features while the remaining members can work on the testing. During this phase, you will use the CI server previously set up to maintain your code integrated into your testing environment.

You must also track the time spent working on each issue and update the respective ticket with this data before closing it. At the end of the assignment, calculate the following according to the chosen estimation method:

- If story points: The team's velocity, as an average of the number of points completed per week;
- If hours: The accuracy of the estimation, as the quotient of the actual time spent vs. the estimated time

Finally, consider how the value would change (if it would change at all) with time if the team becomes more experienced and familiar with this project.

6.1 Task 1: Automated Functional Testing

The goal of this task is to improve upon the initial automated integration test suite provided by creating new test cases that cover all the user stories related to:

The test cases must be implemented using the Selenium WebDriver library in either Java, JavaScript (Node.js) or Python. Document every test case by explicitly identifying the user story it covers.

Each test case must be self-contained and test only a tiny facet or variation of the user story. This means that to cover each user story, at least one, but usually several, test cases will have to be created. Include both positive and negative test cases when necessary, and test every possible variation of the features listed above.

Note: that some bugs may prevent some test cases from passing. You should prioritize those bugs for Task III so that, in the end, all your test cases pass. If you find more bugs during this task, add them as GitHub issues and follow the same process.

Every day, collect test execution results, and in the end, plot the defect arrival graph (i.e., the Number of Defects vs. Day) and comment on it by discussing the different quality characteristics of the released application before and after this task was performed.

6.2 Task 2: Implement New Functionalities and Fix Bugs

The goal of this task is to simulate the day-to-day development of an application. As such, you are tasked to implement new features and fix bugs in the application. In practice, this means picking from the backlog the tickets that have the highest priority and which collectively fit the time available.

This includes implementing the three new features as well as fixing the most critical defects of the system and, if there is time, a few lower priorities but quick-to-fix issues.

Note: that a proper feature PR (Pull Request) consists of the feature code as well as new integration test cases as described on Task I
Before delivering the assignment, generate a release of the Microblog app. This is the version which will be marked. **Note:** that the version number should be bumped to match the release tag generated. Be sure to push everything to the GitHub repository.

6.3 Deliverable

- The release (tag) available on the group's repository containing the updated code and the test suite;
- The GitHub issues and Pull Requests;
- A report containing:
    1. The test cases with test data values;
    2. A description of the uncovered bugs;
    3. The defect arrival graph;
    4. Commentary on different quality characteristics of the released application before and after this part of the assignment;
    5. A description of how the new features were implemented;
    6. The new user stories;
    7. Screenshots showing the Jenkins' build history, trends and reports;
    8. Description of the pros and cons of using Story points or Hours for task estimation and the definition of which one was chosen;
    9. A table with the priority, estimation and actual time spent with each ticket created;
    10. The velocity or estimation accuracy calculation;
    11. The consideration of whether and how the value would change with time;
    12. A commentary on whether doing CI provided any benefit or hindrance during this assignment that would be different in a real professional environment.

7 Part 4: Performance & Scalability Testing

This phase aims to perform load and stress testing of the web application by progressively varying the workload and measuring response time, throughput and CPU utilization; and then plotting and analyzing the obtained results.

7.1 Task I: Load Test

Load Test the server-side functionalities by creating a test plan simulating the behaviour of a user while interacting with the system following this scenario:

Each virtual user should start by visiting the application's home page and logging into the system only once. Then, at each iteration, the user performs different actions with different likelihoods:

1. Navigate to the accounts page 100% of the time;

Run the scenario above in a loop over 60 minutes starting with one user and adding a new user every second until there are 200 concurrent virtual users, and collect the following

1. Average, min and max response times per action (sampler) and aggregate;
2. A chart of the response times over time per action (sampler) and aggregate;
3. The error rate per action (sampler) and aggregate;
4. A chart of the response codes over time;
5. A chart of the number of active threads over time;
6. Average, min and max CPU usage;
7. Average, min and max memory usage.

Based on the above test results, calculate the software availability using the aggregate model presented in class.

Using the measured aggregate response time, the selected think time, and the maximum number of (tested) virtual users, calculate the average system throughput (based on the response time law).

Using the measured resource utilizations, calculate the maximum achievable system throughput (based on the Throughput Bound Law).

Provide a detailed description of the system's behaviour as the number of users increases and after it stabilizes according to the data collected. Report if you notice any abnormal behaviour and how would you further test the performance of these functionalities.

Finally, you should **Note:** that:

1. Each virtual user has its login information, which is not shared among users, and it will only log in once at the beginning of the test;
2. Each virtual user acts sequentially, meaning it will only perform the following action after the previous request is finished;
3. The action probabilities are independent;
4. There should be a realistic think-time between each action;
5. Since this test is only concerned with the server performance, each simulated action should be mapped to the relevant back-end calls. For instance, when requesting a new account, the user has to retrieve the list of branches before actually making the new account request;
6. The JMeter application should run in a different machine than the target server. The test should run in non-GUI mode collecting only the results file (.jtl). After the test, load the result file into the JMeter test plan to generate the statistics and charts. This is done to reduce the influence of the JMeter application execution and collection process on the test results;
7. The machines should be on the same LAN to reduce network interference in the results.

7.2 Task II: Stress Test

Using the same test plan as above as a basis. Modify the test plan so that one virtual user is added every 2 seconds up to a total of 500 virtual users with a timeout of 20 seconds; set up a remote test the environment with two or more JMeter servers (slaves) and answer the following:

1. How many concurrent users are necessary to increase the average response time to more than 2 seconds?
2. How many concurrent users are necessary to increase the error rate to more than 5%?
3. How many concurrent users are necessary to degrade the system's performance enough so it becomes unusable for most users? Can this even be achieved, and if so, how does the system behaves in that scenario? Does it enter into an unrecoverable state, hang or crash, or return to normal after the stress is reduced?
4. Do you believe the system's performance allows it to be released into production? Consider the hardware, network, user base vs. peak concurrent users, variations in user behaviour, performed actions, think-time etc.

Include screenshots or charts to corroborate your answers.

As with the load test, the target server should be on a different machine than each JMeter server, but they should all be on the same LAN. The JMeter master can share the machine with 1 slave; therefore, 3 machines will be required.

All those restrictions are in place in order to reduce external interference on the test results and because of the risk of saturating the client machine instead of the target server, so it is important to follow them thoroughly.

Remember that each virtual user should have its credentials; this includes virtual users of different slave machines.

7.3 Deliverable

- The JMeter test plans created;
- The result files (.jtl);
- In the report:
  - Document the aforementioned tasks;
  - Comment the obtained results;
  - Explain your test organization (test elements and structure) and configuration choices;
  - Describe the steps taken, configuration changed, and commands executed to run the load and stress tests. Include screenshots when possible.

  A single report can be provided for both sets of tasks (i.e. stress and load tests)

8 Part 5: Security Testing

This phase aims to conduct threats and vulnerability scans for the application. Different tools will be used to cover different types of security tests. The following tasks must be performed:

1. Construct a threat/vulnerability matrix based on the template presented in class. Start by listing the high-level features of the application. Then identify threats of each feature and possible mitigation solutions, and describe how they can be verified. Finally, build the table listing the feature, threat name, brief description, mitigation solutions, verification cases and status.
2. Analyze the SonarQube reported issues to identify potential vulnerabilities or weaknesses in the application's source code. Audit the scan result by analyzing every vulnerability and security hotspot issue (bugs and code smells can be ignored) reported by the tool in light of the threat model (i.e. whether they are relevant or applicable) and/or by trying to exploit them.
   (a) First, analyze whether each issue is either a false alarm or a real vulnerability and mark it accordingly – **Note:** that exploitable Security Hotspot issues will be turned into vulnerabilities when set to "Detect". To help analysis, click on the ellipsis icon to the right of the issue's title to bring the details panel. Read the information there thoroughly to learn more about the general specificity of each issue type. This is especially important when dealing with possible security issues since analyzing them is a complex task;
   (b) Then, for the real issues, update each issue type and severity according to the Impact/Likelihood matrix. This is important because the tool might misclassify some issues.
   (c) Add the appropriate commentary to each issue to describe the reasoning behind each action performed.
3. Use the Zed Attack Proxy (ZAP) tool to pentest the application and uncover vulnerabilities not previously identified. An adequate pentest consists of automated scans interwoven with proxied explorations and directed attacks;
4. Fix all high severity (that is, high, critical and blocker) vulnerabilities identified in steps 2 and 3 Lower-severity vulnerabilities don't need to be fixed. Don't forget to follow the same protocol used in Part 3 (GitHub flow). **Note:** once again that some issues might be misclassified by the tools, so classify every issue according to its impact and likelihood;
5. After fixing the high-severity issue, re-scan the updated version of the application with ZAP to make sure the vulnerabilities were fixed correctly and that no other vulnerability was introduced;
6. After merging the fixes to the master branch, analyze the updated SonarQube report to be sure every reported issue was fixed and no new critical issue was reported. Mark issues as "fixed" if required;
7. Based on the above testing activities/outcome, update the Threat/Vulnerability matrix by updating the test coverage/status;
8. Before delivery, release a new version of the application as done in Part 3

8.1 Deliverable

(a) The release (tag) available on the group's repository containing the updated code;

(b) The actions and commentary available at the group's SonarQube project;

(c) A zip file comprising:

   i. The ZAP session(s) containing all the scans performed;

   ii. A report containing:

      A. Screenshots of some reports generated by SonarQube showing changes in the number, type and/or severity of issues through time;

      B. A summary of the pentest process with screenshots and descriptions of scans and tests performed;

      C. A description of the highest severity vulnerabilities found by ZAP;

      D. A short comparison of vulnerabilities found by the two tools and an explanation of why some of these are different and what role each has, if any, in a thorough security evaluation of an application;

      E. The Threat/Vulnerabilities matrix;

      F. A small paragraph commenting on what other tools and/or techniques could be used to further test the application's security.