Google Workspaces Alternative Application

Project Testing and Acceptance Plan

Wahkiakum School district, Naselle School district, and Wahkiakum 4-H

Team Toto

Albert Lucas

Ben Kaufmann

Daniel Semenko

10/27/2022

TABLE OF CONTENTS

I. Introduction

Wahkiakum School district, Naselle School district, and Wahkiakum 4-H are in need of an application that allows student users to communicate with each other across classrooms, clubs, groups, etc. To satisfy these district needs, our team is planning on analyzing, designing, and building a stand-alone application from scratch to improve student learning and connection.

This document is our Testing and Acceptance plans document, which contains different tests and ways for accepting functions for Team Toto's communication application. It will detail documenting our current strategy and future plans to validate our solution.

For every requirement in our requirements section, this document will detail the one or more tests that go with it. This will also serve as a basis for the agreements between Team Toto, Ananth Jillepalli, our mentor, and Ron Wright, our client.

Our hope with this document is to inform readers of all the ways that we will test the many different aspects and functions of our communication application.

II. Testing Strategy

To achieve the most success in meeting our testing objectives, we will loosely follow a general testing lifecycle, as shown below:

II.1. **Developer writes code:** This will include the implementation of the various subsystems of our software project. These include the Controller, Authentication, Computational Logic, Model, Data Access Logic, Google Cloud SQL, and View subsystems.

II.2. **Developer writes tests for code:** Here, the actual tests for all the functions of the software subsystems are written.

II.3. **Developer runs tests for code:** Test programs are run on the program to assess the effectiveness of the code against the requirements of the project.

II.4. **If tests fail, fix bugs in code and retest, otherwise continue:** If a test case fails, that means the code has bugs, and it is not ready for development. Developers will repeat steps 1-3 until no bugs are found. Once all test cases pass, developers can proceed to the next phase.

II.5. **Developer pushes code to GitHub, where CI/CD testing occurs. CI runs code through its pipelines to test if it is eligible to continue to the next step:** In this next step, Continuous Integration and Continuous Delivery occur, to ensure newly developed functionality or features do not conflict with previous changes. If CI/CD testing fails, developers repeat steps 1-4, until it passes, and can move to the next phase.

II.6. **Developer creates a merge request to the master branch from their development branch, with all team members added as reviewers:** Developer creates a merge request through GitHub, and the other team members are added as reviewers.

II.7. **Requests require at least 1 other team member to approve before the merge can be complete:** The feature branch must be approved by at least one other team member before it can be merged back into the main branch. If no team members approve, the developer must repeat steps 1-6.

II.8. **Upon approval, code is merged into the master branch and deployed by the CD, if there are no pipeline errors:** Code is merged from the development branch into the main branch, and is deployed through CI/CD.

During testing, if a developer encounters a new bug, they must create a GitHub issue describing the bug. The same developer is responsible for fixing the bug and creating new tests relating to the bug. This allows those developers who are most familiar with certain system functions to specialize their efforts into those specific functions.

We will be utilizing the CI/CD development pipelines strategy, in which there is continuous development, testing, and delivery of new code.

III. Test Plans

III.1. Unit Testing

The primary goal of unit testing is to take the smallest unit of testable software in the application, isolate it from the remainder of the code, and test it for bugs and unexpected behavior.

For our application, it seems like the best approach for unit testing is the standard approach. We can have two or more tests for each important/non-trivial function in the code. Potential tests can include valid function usage, and invalid function usage, among other function-specific tests or edge-case tests. Obviously, the more complex the function, the more unit tests should be linked to it. Conversely, trivial functions, such as doing very simple operations or calling other methods, should not require any testing.

We should factor in relevance and complexity when determining how many unit tests per function, thus we will use these two metrics as described: Relevant methods are those used a lot, or that if broken, would result in a severe decrease in Confidentiality/Integrity/Availability of the application. Complex methods are those with many potential uses and many internal execution paths. It will be up to a developer's individual discretion to determine how many tests a function they wrote deserves.

III.2. Integration Testing

Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group.

In regard to our application, integration testing would be more complex than the basic and general unit tests. We will generate integration tests based on entry points. These entry points are the beginning of some application functions, and the test is testing that everything works as

intended. An example would be logging in. The entry point is logging in, yet the tests are much more complex since a multitude of methods and functions are performed to complete the process of logging in.

Choosing where these entry points are, and which entry points are more important (based on relevance and complexity) will be up to the discretion of the development team. A more important entry point may be one where there are multiple tier 0 system requirements being fulfilled.

With the integration tests, it seems more important to test whole classes rather than individual methods. This helps by reducing the complexity of creating test cases, while still promoting high code coverage.

III.3. System Testing

System testing is a type of black box testing that tests all the components together, seen as a single system to identify faults with respect to the scenarios from the overall requirements specifications. The entire system is tested as per the requirements.

During system testing, several activities are performed:

III.3.1. Functional testing:

Test of functional requirements (from requirements specification). The goal is to select those tests that are relevant to the user and have a high probability of uncovering a failure.

In regard to our application, we will rely on our Requirements and Specifications document. In that document, we specified all the functional and non-functional requirements of our application. For functional testing, we will simply take all our functional requirements, and build functional system tests around each of those requirements. This will be logged in a document, including the steps taken in the system to carry out that function. All events of system restructuration that change the operations of functions will be written in this document. All data of failed tests will be recorded in the Results section with relevant information that identifies the problem within the function test. The developer that performed this test should then create an issue within the team's repository and assign a team member who is responsible for that specific task. That team member will engineer a solution towards that task by writing standardized tests to exterminate error results that could occur in the future.

III.3.2. Performance testing:

Performance tests check whether the nonfunctional requirements and additional design goals from the design document are satisfied. In stress testing, the system is stressed beyond its specifications to check how and when it fails.

Performance testing will be executed in a similar manner to functional testing. We will be stressing the speed, scalability, and stability of the system under a specified workload. The tests should be based on specified variables in the Requirement and Specifications document and will meet the non-functional requirements of the system. The developer should be able to analyze the results of these requirements and give feedback using their own intuition. A

document will be created with three sections: Non-functional requirements, steps to test/satisfaction criteria, and results. In the case that a developer feels that the system fails one of these requirements, they must identify what needs to be changed within the application to meet that requirement. Like a functional test, that developer must create an issue within the team repository and assign it to a member that's responsibilities fall under this type of error.

III.3.3. User Acceptance Testing:

User acceptance testing will be exercised constantly throughout the development of the project. It will be similar to black box testing where two or more end users are stressing the application in a practical manner. This will allow the clients to evaluate the system and validate the flow of the application. Developers must also have expectations within mind that are based on the requirements in the Requirements and Specifications documents. The developer will produce a task agenda to ensure that these requirements are satisfied and recorded for improvement or fixing. This testing will also let clients explore new avenues of testing that have not already been performed by the developers. As user acceptance testing continues, the developers should show-and-tell less and let the client use the application until they have a question or request that will be documented. The goal is to have a staff member of a school district be able to train others on how to use the application by themselves. Like the previous testing, if a bug is determined or a request for improvement is made, the developer must create an issue within their team repository and assign it to the most relevant developer. That developer will be responsible for fixing that bug or creating that feature.

IV. Environment Requirements

Our testing environment will utilize a few different tools to enable Team Toto to best test the project. For unit testing, we will make use of Flutter testing packages, including the test and flutter_test packages. These two allow unit and integration tests to be created for any functions needed for testing.

In addition, GitHub has Continuous Integration available which our team will utilize to make sure that all of our tests are run, and passed, prior to merging any commits made by the team.

Finally, the hardware that will be used for the testing environment will be a Chromebook with standard school specifications. Since the application will run mostly on Chromebooks, we can physically test on Chromebooks to ensure that requirements falling within the User Requirements section are satisfied.