GCISL Full Stack Application

Testing and Acceptance Plan

Washington State University Granger Cobb Institute of Senior Living

GCISL Team

Musa Husseini, Tom Arad, Nathan Bunge

I. Introduction

I.1. Project Overview

The goal of the Cobb Connect app is to allow graduates, students and faculty of the Granger Cobb Institute to communicate with one another as well as provide information on their career paths. When building something from scratch, such as this application it is expected for problems to arise. In order to detect and fix these problems as fast as possible the team needs a solid understanding of the app's functionality as well as testing methods which show that everything is performing as expected. Since the app is under constant development it undergoes changes often. Making it so it has to be tested frequently. Things like third party connections such as the firebase database that the app uses, front end design, and back end functionalities which connect the two have to constantly be checked and maintained.

Major Functionalities:

- Account creation
- Account validation
- Updating account Information
- Feed
- Messaging

I.2. Test Objectives and Schedule

Software testing will be crucial in the development of the Cobb Connect application. It will help the team identify any issues present in the app and as such outline what needs to be fixed so that the application meets the expected standards and quality. In order to achieve this goal the Cobb Connect team will use existing testing tools provided by flutter and firebase as well as tools made by independent developers which were made available to the public such as the firebase fakes.

When handling the front end flutter code the team will use the flutter_test tools to conduct unit testing as well as widget testing. The flutter_test package provided by the flutter SDK allows the user to add test files written in dart which check for logical issues as well as render widgets to the screen. While the flutter_test package is handy when testing individual components of the app, it is also important to test how all of these components interact together. For the front end integration testing the team will be using the integration_test package which is also provided by the flutter SDK. This package allows the user to search for components as well as verify that everything that should be present in the app is indeed there. Similarly to the flutter_test package the testing files for integration testing are written in dart.

Testing the back end of the application is a bit more complicated. It is difficult to use the Firebase libraries to run unit tests since they require an actual device or emulator. A good replacement the team has found for this problem is the Firebase fake/mock packages. These packages implement the API of specific firebase libraries and simulate their behavior. Finally for back end integration testing the team will use the Firebase emulator which uses the cloud firestore as well as other core firebase resources.

The deliverables provided by the team will include our gitlab repository containing our application code, test files, and a README file which will provide all of the necessary information regarding the repository. In addition the team will provide documentation detailing the tests and application in a separate document. We will be using the agile process as our testing strategy. Tests will be written by the developers and implemented at the same time as the code for the application. Milestones: code, unit tests for the code, integration tests for the code, widget tests for the code, integration between flutter and firebase, demo, and user testing the code. These milestones may repeat if there is a need for updates or other additions to the application which require the beginning of a new cycle.

I.3. Scope

This document discusses our plans for testing the Cobb Connect Application. The scope of this document covers the general testing guidelines which the team will follow, as well as the resources and frameworks we will be using to test our code. While this document provides a breakdown of the testing strategy it does not discuss individual test cases as well as the people who will be responsible for writing them.

II. Testing Strategy

For our testing strategy, our team will create tests for all aspects of the system. Everything from the front-end to the back-end will have some level of testing applied to it. This will include testing all landing pages, views, and functionality. To achieve these goals, these are the steps to be taken for our testing life cycle.

Our team will use the CI/CD (Continuous Integration/Continuous Delivery) testing strategy. This process will allow us developers to create and deliver product features in a timely manner. It will introduce automation into the stages of our app development.

1. **Write Test Cases:** All developers will be responsible for writing tests for the code they create. This will be required for all subsystems. Ensuring each developer creates tests for their own work will help keep our system under control.
2. **Run Test Cases:** Each developer will run the tests they have created. It is up to the developer to determine if their test cases have enough coverage and are concise.
3. **Make changes according to test results:** Depending on the result of the test case, the developer will need to take different actions. If the test fails, they need to locate the

source of the bug, revise and fix the bug, and rerun tests until they pass. If all tests pass the developer may move onto the next step.

4. **Push code to remote, and CI runs commit through pipelines:** The CI/CD testing will happen once a developer pushes their code to the repository. The pipeline will run the newly committed code and check to make sure there are no conflicting issues. If the pipeline fails then the developer will need to redo the previous steps. If they pass the developer can move onto the next step.
5. **Make merge request to main:** In this step the developer will make a merge request to main to have their code a part of the main codebase. At this stage it is relatively safe to make this request due to the amount of testing done beforehand.
6. **Request must be approved by another developer:** As a rule of thumb all developers should have their code reviewed by at least one other developer on the team before merging to main. This will ensure that the code being proposed to be merged is being seen from a different perspective and not through automation.
7. **When the branch is merged into main, it will be deployed by the CD**: In this step after the new code is merged into main, it will be deployed by the CI/CD.

If a bug does get through the process described above there are certain steps to be done to fix this bug. The first step is to create an issue in the repository describing the bug. The issue should be assigned to the person who wrote the code, not the person who found the code. This is done so that the person who is the most familiar with the code can trace down the source of the bug. The developer responsible for the bug must not only fix the bug but also add test cases to catch any occurrences of the bug in the future.

III. Test Plans

III.1. Unit Testing

When creating unit tests the goal will be to isolate each function and give it inputs to test against the returned variable. Each unit test should have at least two to three tests. Each unit must test for a lower bound and upper bound (if applicable), an error check, and the expected result. The breakdown goes as so. Each class will be given a test suite. Each function within that class will be tested in their own individual unit. Within each unit the rules as described above will be used for assertions. Along with each test suite, there will be a setup function, which will set up all the required variables needed to test, and a tear down function to deallocate any memory used.

III.2. Integration Testing

Using integration tests for our system will be difficult to do effectively. Our team will develop call graphs to help organize the way our system works within itself. We will create top-down call graphs for classes and create cross class graphs where needed. To integrate a class, we will

follow the call graph created. We will begin with creating mocks and stubbing each function within the class. We then will go through one function at a time, un-stubbing it, and then verifying that the assert passes. The integration test for a class will be considered complete once all functions are un-stubbed and all assertions are passed.

III.3. System Testing

The following types of tests will be performed to test the system:

III.3.1. Functional testing:

Our functional testing will cover each of our four main requirements: Accounts, Posting, Messaging, and Analytics. Each of these were defined in the Requirements and Specification section, which goes into more detail for what needs to be done in each requirement. Testing for accounts will involve both testing account creation and signing in. Multiple fake and real email accounts will be sent for verification, and the site must correctly verify the emails and passwords. Testing for posting will involve adding, deleting, and editing posts. The tests will need to verify that the main page is updated each time. Message testing will be very trivial, it must verify that a message sent to a user is received. Lastly, Analytics will need to be tested with fake user data to determine if the site is calculating correctly.

III.3.2. Performance testing:

Performance testing will not be as straightforward as function testing, as the non-functional requirements are not quantitative in nature. This means that most of the non-functional requirements will be more up to developer discretion. After a prototype site is created, each of the non-functional requirements will be examined and the developers will have to determine if they have met the requirements. Any requirements that are not met will require code refactorization.

III.3.3. User Acceptance Testing:

User acceptance testing will occur multiple times throughout the app development stages. The team will provide a series of activities which will demonstrate the app functionality based on the requirements and specifications as well as what the team managed to get done. These activities will most focus heavily on recent additions to the application. After the activities are performed the customer will be free to explore the application in whatever way they see fit and prove additional comments. Any major bugs or problems found will be fixed by the team and included in the activities of the next demonstration.

IV. Environment Requirements

Unit testing will be performed using dart's testing package [11.] This has no hardware requirements, but all the installation steps outlined in the project readme will need to be performed in order to perform the unit tests. Integration tests will also be performed using dart's testing package. Functional tests will be done with both dart tests scripts, as well as outside black box testing. This means that the software requirements are the same as before; all the installation steps outlined in the readme are required to run the tests.

Performance testing will be done all by developer discretion, as there is no quantitative way to test the requirements. Therefore there are no hardware or software requirements, other than just accessing the website itself. This is the same for user acceptance testing. They will only need a device that has a web browser to access the website.

V. Glossary

Flutter: An open source user interface development kit

Firebase: A hosting service for applications

Front End: The part of the application the user will be seeing and interacting with

Back End: The part of the application that runs the logic and processing

Database: The part of the application that stores all data

API: Stands for application programming interface. It is a type of software interface that offers a service to other pieces of software.

SDK: stands for software development kit, is it a is a set of software-building tools for a specific platform

VI. References

T. Hamilton, "Integration testing: What is, types with example," *Guru99*, 27-Aug-2022. [Online]. Available: https://www.guru99.com/integration-testing.html. [Accessed: 28-Oct-2022].

T. Hamilton, "Unit testing tutorial – what is, Types & Test example," *Guru99*, 27-Aug-2022. [Online]. Available: https://www.guru99.com/unit-testing-guide.html. [Accessed: 28-Oct-2022].

[11] “Test: Dart package,” *Dart packages*, 26-Oct-2022. [Online]. Available: https://pub.dev/packages/test. [Accessed: 28-Oct-2022].