Intelligrain Code Testing Report

Brandon Hillbom and Dillan Zurowski

April 1, 2024

Table of Contents

Introduction 1
Environment 1
Test Execution Summary 1
Detailed Test Results 2
Test Coverage 2
Findings 2
Conclusion 3

Introduction

The purpose of this report is to assess the code testing for the Intelligrain project, the methods used, and how they helped increase the quality of our code. This report is important in ensuring quality and reliable code for the handoff to Ground Truth Agriculture The scope of our tests includes unit testing with regards to the FastAPI server endpoints, unit testing for frontend shared functions, and widget testing for various widgets on the frontend.

Environment

1. Hardware: List hardware used, like servers and devices.
   1. Android Tablet (Google Pixel 6 & Fire Tablet)
   2. FastAPI (server) via laptop
2. Configurations: Detail configurations used in testing.
   1. FastAPI must be running via localhost port 8000 and the localhost IP must be in the .env file loaded into the device.
3. Software Versions: Mention versions of the software being tested.
   1. Flutter version 3.19.0
   2. Dart version 3.3.0
   3. Python 3.12.1
4. Testing Software:
   1. Python: pytest
   2. Flutter: test and flutter_test

Test Execution Summary

Front End Unit Testing

Our front end unit tests focused on verifying our formatter, type safety, validation, and filtering functions all worked as expected. These were considered the most important unit tests as these functions are used in various places in our app. Filtering is also more logic based so ensuring it worked as expected was very important. In future iterations, we would like to expand our testing to include all written functions.

- Total Test Cases: 47
- Executed Test Cases: 47

- Passed Test Cases: 47
- Failed Test Cases: 0

Widget Testing

We conducted widget testing on various authentication pages to ensure that the expected widgets showed up properly. Unfortunately, we did not get as far as we would have liked and as a result, the majority of widgets have not been properly tested. Additionally, our widget tests were fairly basic as we did not leave enough time to properly test the flow of interacting with the app. However, of the widget tests we did, we found two issues where the rendering of the UI elements extended past the available space. We were able to fix this as a result. To mitigate the lack of widget tests, we implemented rigorous blackbox tests.

- Total Test Cases: 12
- Executed Test Cases: 12
- Passed Test Cases: 12
- Failed Test Cases: 0

Blackbox Testing

This is testing done by a user where the user doesn't look at the code at all. It is all about testing the completeness of the app from the user side. The detailed testing can be found in the Code Testing Google Sheets document on our Github. However, the main takeaways were that the app's core functionality works as expected with no failed cases found. During testing, we did find several bugs that we had not encountered before.

- Total Test Cases: 53
- Executed Test Cases: 53
- Passed Test Cases: 53
- Failed Test Cases: 1
  - Notes: This failed case was the export feature. The export button should check the device for granted storage permissions and ask the user to enable it if it does not have access to the storage. Once access is granted, it will download the necessary files to the device. This feature only worked for one out of two devices we tried due to issues allowing storage access. The test passed for our Fire 7 tablet but failed for the Google Pixel 6, therefore this test is considered a fail.

FastAPI Endpoint Unit Testing

One of the biggest bottlenecks for errors is at the endpoints of the API in our backend. Testing methods include testing endpoint operations and all known reproducible error cases. Ensuring the expected status code and return messages are returned for each endpoint.

- Total Test Cases: 42
- Executed Test Cases: 39
- Passed Test Cases: 39
- Failed Test Cases: 0
    - Notes: Did not run Read from Firestore, Export as shapefile, and Get annotated images. These failed simply due to issues creating mock functions for these endpoint tests

Test Coverage

Due to limited time constraints, we got a score of 68% for our testing coverage for the backend endpoint tests. The missed statements were due to accounting for unexpected errors that we were unable to reproduce.

Findings

Performing the endpoint testing provided insightful feedback on simple user coding errors such as inconsistent error status codes and messages as well as not covering important firebase calls in a try-catch statement which has the potential to cause an error. For example, one of the Firebase calls to update the user profile information was not covered in a try-catch, meaning if an incorrect user id was entered that it would throw an error. These issues have since been fixed and it is less likely that an unexpected error occurs. This also allows for more specific error messages which are beneficial to both the user and developer.

While performing widget testing, we found several issues with UI elements rendering past their available space. This was helpful as we had not found this issue on the screen sizes we were testing with. We were able to fix these issues by adding a SingleChildScrollView to allocate space dynamically.

Blackbox testing and user testing was a major contributor to our bug finds. By seeing how different users use our app, we were able to get fresh perspectives that led to bug finds that we had not encountered. As an example, during Blackbox testing, we found that if a popup is clicked and the user swipes to change the page, the popup can't be exited because the context of that widget no longer exists.

Conclusion

Our tests covered a significant portion and the important aspects of our project including testing for all known and reproducible test cases for all server API endpoints as well as data validation testing before sending data to those endpoints. While we received insightful information from the tests that we performed, we would have liked to have many more tests for both the front-end and backend with a higher test coverage as well as implementing mocking more efficiently. Our app was very user-centered so most of our time and effort was put into making sure the user experience was clean and met all requirements.