Walk the Hills

College Hill Association and Pullman 2040- Mobile Application

College Hill Association

and

Pullman 2040

Finite Cipher

Adam Wagener;

Mitchell Footer;

Juan Ibarra-Delgado;

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

Walk the Hills is a mobile application being made in collaboration with the College Hill Association and the Pullman 2040 team, meant to make walking the streets of Pullman safer and more accessible to all. The app is being designed for everyone wanting to navigate Pullman, whether they are a local or simply visiting, while focusing on the key design objective of “putting safety first” to create an experience that will increase Pullmans foot traffic while also providing a solution more tailored to the town then applications like Google Maps may allow for. In the app users will be able to filter routes based on safety and rate each route at the end of their session based on criteria like safety or difficulty. Doing this allows Pullman’s local government to collect meaningful information about where the foot traffic of the town currently is at all time, whether it is rainy or sunny, night or day. Users will also be able to report things like light bulb outages or broken sidewalks, providing the city with info on what needs to be fixed, allowing them to get in contact with the people necessary to fix it.

The app is being constructed in Facebook’s react framework using the Google Maps API, which imposes some particular challenges for testing and validating certain API calls. We consider automated tests crucial for the core Google Maps integration that handles the ability for the app to run and select routes for the user dynamically, as well as the save reported information into our database. Most other tests would be beneficial, but we feel that prioritizing a series of automated tests will benefit the app the most coming into the future.

I.2. Test Objectives and Schedule

The primary goal with tests in this style of project is to gain confidence that the code and processes we’ve developed and written will result in a consistently successful execution of the desired behavior and fulfill all necessary requirements at runtime. Automated tests serve the additional purpose of serving to safeguard the project against being broken by future updates. Since we are using the Google Maps API, we know we have at least five years of support for the code we are currently running, which means updates aren’t a huge concern at this time. However, we want the future of this app to remain stable for as long as possible, which is why our automated and manual tests will accomplish both consistent successful execution, and insurance of compatibility, helping to ensure that all core features are operating correctly each time that we, or a future maintainer, deploy a new update to the app. We acknowledge these kinds of tests are especially important to have established as we transition into the second half of our development process next semester. This second development phase will be customer focused. In this phase we will directly interact with users and slowly roll out beta tests for the application. Beta tests like these are important for receiving feedback, which we can use to rapidly iterate on our build. Making sure our tests are of high quality now, will ensure reliable iterations during these beta tests.

Because the project is using the react framework, we will need to adopt a very specific set of testing technologies. For unit testing we will have to use the React Testing Library, a way to “write maintainable tests for your React components” that integrates with the React framework [1]. We can use this framework to develop some isolated level of integration testing since it allows us to “test React components without relying on their implementation details”, meaning we can test that certain components will work before integrating them into our code [1]. Due to the heavy use of another professionally managed API, we will not be able to incorporate full integration testing. However, Google Maps API is an exceptionally well managed highly

professional and well tested API whose functions can be relied on thanks to the efforts of the Google team.

Manual testing procedures will need to be created and documented. We predict a lot of it will be reliant on the browsers console log, allowing us to print certain things within that form of output, where we can manually view what is going on while we are creating or modifying our code. This form of testing will be conducted during our acceptance testing process and will ensure that our product continues to perform all expected services and functionalities as they were designed. Beta testing will allow us to gauge how users interact with certain systems, leading to rapid build iterations based on the feedback received from their experiences.

Our testing process will deliver 3 major deliverables. The first is a suite of unit tests covering all non-trivial non-UI scripts produced when creating the functionality of the application. The

second is a document containing the testing procedure including instructions needed to operate the tests as well as expected results for all manual functional tests and playtests. The final deliverable is a GitHub CI pipeline that will allow for more efficient deployment of future iterations. We will do this using GitHub Actions to assist us with deployment to development platforms.

I.3. Scope

This document discusses our plans for how we intend to test and create test material for Walk the Hills. It covers how we plan to implement the tests, as well as the procedures for said testing including our essential product tests. While this document does not contain every test we plan to implement, along with every procedure, we will include examples within the appendices of the document to offer a sampling of those methods.

II. Testing Strategy

Project testing will be designed to create full automated tests of all core functionality of the app, running these tests through a CI pipeline for continuous integration. While sprints would allow us to easily use CD by deploying changes monthly, we believe CI will be a better practice for our team as we are currently in other classes that may pick up or slow down variably. The major components of the core functionality are the Google Maps integration, and the integration of our database. Our team has worked with our client to identify future stretch goals that may be tested, but these may be run without automated tests due to the time frame and scale of the project. Because React and Google's API automatically catch and handle many errors at runtime, errors in these components will be known about before deployment, and as a result will not compromise the overall operation of the finished system, only certain components that are still in development or being implemented. Our specific testing procedure is broken down below into 2 distinct processes. The first process describes how we develop, run and push code tested through automated unit/integration testing, as well as manual FT and acceptance testing. The second process is a preliminary outline of unit tests during playtesting of our alpha and beta phases, to be finalized later with the help of the College Hill Association and the Pullman 2040 team.

Developer Testing Process:

1. Code is written by Developer: We are not planning on using TDD frequently in this project. Our team has established it will be more beneficial for us to implement white box testing since we are using so many heavily tested implements. The details of the

implementation are more important than the direct input to output relationship tested best with black box testing.
2. Team determines test cases for code: For each core functionality, we will have at least two tests. We know users may interact with our systems in ways we may not intend, so one test will cover unexpected inputs and/or interactions ensuring satisfactory user feedback for proper use, while the other will test the expected interaction for validity. For non-core functionality only one test is expected which will test intended interactions.
3. Run tests: The team will run tests on all current functionalities and features that are present within the app.
4. Inspect test data: The team will discuss the results of the tests, and determine if code needs to be changed. If changes are needed, the specific changes will be discussed and documented before final product release.
5. Fix any code if necessary: If code needs to be changed, the developer will change it to correctly adhere to quality and testing standards, ensuring the functionality functions correctly. This will then restart the testing processes for all affected files.
6. Push code to GitHub: All changes will be pushed to a separate branch on GitHub. A branch will only be eligible for a merge if its most recent commits pass all tests run in the CI.
7. Developer makes a pull request against main: The developer makes a pull request against the main branch, prompting at least one other developer to review the code being pushed. The branch should not be merged until this code review has occurred.
8. Merge branch: Assuming the branch passes all previous CI tests and code reviews, it will be merged with the main branch.

Playtesting Process:

1. Create playtest build: A developer will create a build of the app they are ready to playtest. This can be a fresh build, or one specifically targeting a certain feature needing feedback.
2. Deliver build with time to test: Deliver the build to testers, with at least one week for them to test thoroughly, this time frame may be expanded depending on the number of features needing to be tested.
3. Send out feedback forms: Send out a Google form that will allow for testers to provide both positive and negative feedback. This form should be tailored to what is being tested, asking specific questions and should be remade for each build to ensure the most accurate information is received each test.
4. Collect and record feedback data: Collect the feedback after one week, collect it all in one document used to record the tests, and analyze the feedback received.

Make necessary changes: Using the feedback received, make necessary changes to functionality or user experience to move towards an improved iteration for the next playtest.

III. Test Plans

III.1. Unit Testing

The team will generally follow traditional unit testing procedures, however, the team will diverge in areas that are React specific. The team will be using React Test Library and Jest to deploy accurate and meaningful tests to the code. In order to ensure the code has been effectively tested, the team will be required to test all core functionalities within the app. Core functionalities can be defined as anything critical to the experience of the user such that losing them would result in the app's inability to function, such as navigational services. Additionally, the team will

evaluate the relevance and impact of all non-essential functions. If the developer feels it necessary, more unit tests can be developed for additional functionalities. For this section, we will defer to individual developer discretion when considering the extent of additional unit tests.

III.2. Integration Testing

In the React Testing Overview document, it is said that “the distinction between a ‘unit’ and ‘integration’ test can be blurry” [2]. For this reason, our integration testing will mirror the Unit testing section, allowing for multiple components and functionalities to be integrated and tested.

III.3. System Testing

III.3.1. Functional testing:

The team’s functional testing plan is mainly reliant on manual testing implemented by developers. For this section, the team will primarily focus and refer back to the previously outlined Requirements and Specifications document, which contains developer and stakeholder expectations for project functionality each functional requirement, outlined in the document mentioned above, will be associated with one functional test. This section of the document is also subject to revision if there is an update or restructuring to any of the project's functional requirements. Given the nature of working with the React framework, these tests will be manually validated by the developers. In the case that a functional test should fail, the developer who is testing that component will provide a description of the test conditions and request the original developer to reevaluate and solve the error, documenting what went wrong and how the fix was implemented.

III.3.2. Performance testing:

React has a multitude of tools built in that can help with testing, including performance tools. The team has elected to use the React Performance add on in order to analyze things like wasted time, operations, and other various measurements. This is especially important since Walk the Hills will be a cross platform app, dealing with various different kinds of performance constraints, so ensuring that we make the most of the memory or CPU usage we are allotted will be crucial in later sprints. Additionally, this tool will provide the resources for the developers to monitor the performance of the application under variable loads. In regards to the non-functional requirements specified in the Requirements and Specifications document, our team will manually test the performance of these functionalities.

III.3.3. User Acceptance Testing:

In collaboration with the College Hill Association and the city of Pullman, the team will employ several playtesting iterations before the final release of the app. Our testing strategy for user acceptance testing, will be outlined below with sections for provided resources, instructions for testing groups, and plans for feedback and revision. This outline seen in Appendix A mirrors the Playtesting Process detailed above, however this description focuses more on the developer view.

A. Resources
    a. A build of the application (for each tester)
    b. A google form for feedback (for all testers)
B. Instructions
    a. A small select group of testers will be chosen for this testing iteration.
    b. Each tester in the testing group will be provided with a working build of the Walk the Hills app.

c. A to be determined timeframe will be provided for testing.
d. After the tester reaches the end of the testing time frame, the tester will be provided an exit google form for feedback.

C. Revision
a. The team will receive and review all feedback forms.
b. The team will discuss group feedback and discuss any possible modifications.
i. Documentation will be provided based on any modifications or additional features that are added.

Some of the features we are hoping to include in these playtests are reporting outages and safety ratings, getting feedback on unmarked trails Google may not have, and changing routes based on situations like time of day and how well those work for our users. Our sponsors are ultimately hoping this will be an app that will be able to be involved with anything walking related. If there is a broken sidewalk or streetlamp bulb that is out, they hope the app will be able to send that information to whoever needs it. The city is also hoping to eventually expand the app to feature recommendations for things like restaurants or entertainment curated by locals, however this may be a stretch goal due to our time constraints with the app.

IV. Environment Requirements

Our testing environment will be contained within the React framework. While we are planning on making our app cross platform, using the tools React provides us, we will be able to stay within the React environment when testing these individual platforms both before and during deployment. As previously mentioned, React acknowledges that the difference between unit and integration tests can be blurry, as such they will be treated the same.

When it comes to hardware requirements, users will need a device that is still receiving security updates from its manufacturer. For mobile phones, this is often an indicator that a device is still being supported by it's app store. When it comes to PC's the same applies but is more based on the software manufacturer, such as Microsoft or Apple continuing to support that generation of hardware, however the same general principles apply when discusses current app updates. This will allow us to focus our efforts to keep the app updated for the correct devices.

V. Glossary

Cross platform: the ability for a piece of software to function across multiple computing platforms.

TDD: Stands for Test-Driven Development, a software development approach in which tests are written before the code that needs to be tested. The TDD process typically follows these steps:

API: Stands for Application Programming Interface, a set of rules and protocols that allows a software application to interact with another software application.

Unit testing: 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.

Integration testing: detects faults that have not been detected during unit testing by focusing on

small groups of components

System testing: 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.

CI: Continuous Integration, a software development practice where changes made by developers are automatically merged into a shared repository.

CD: Continues Development, every change in software that passes the designated tests is automatically deployed to the production environment.

VI. References

[1] “Performance tools,” React, https://legacy.reactjs.org/docs/perf.html (accessed Oct. 26, 2023).

[2] “Jest,” Jest RSS, https://jestjs.io/ (accessed Oct. 26, 2023).

[3] “Testing overview,” React, https://legacy.reactjs.org/docs/testing.html (accessed Oct. 26, 2023).

[4] Testing-Library, “Testing-library/react-testing-library: simple and complete react DOM testing utilities that encourage good testing practices.,” GitHub, https://github.com/testing-library/react-testing-library (accessed Oct. 26, 2023).

Appendix-A

Test Name	Aspect Being Tested	How this is tested	Expected Results	Result success criterion	Test case requirements
Add Route Feedback via UI	Can text be entered in real time and stored?	A sample pin drop and text entry will enter in set time window	The full complex stream of characters is returned in order in time.	All character received as plain text in identical ordering.	N/A
Valid and expected route	Can a route be given from two simple locations using autofill.	Two simple strings are entered with autofill requests enabled.	A list of routs displayed functionally getting user from source to destination.	The specific route request is valid and arrives in less distance than specified.	Blank initial map.
Route Clearing	Can a route be cleared when a new route is requested	Two simple strings are entered with autofill requests enabled.	No paths on the old route still appear in the return results.	The list of returned results is void of all old outputs.	A map containing a route from the spark to Safeway was mapped before test ran.
Route Filtering	Will filter routes re-route to reduce the desired constraint.	Two simple strings are entered with autofill requests enabled. Filter is set to elevation.	A route with less total elevation is displayed showing the new elevation	Elevation <= 800 in return.	A route from Beasly to SEL One Building is requested.
Feedback Viewer	Will feedback be displayed to a user upon request	A pin is dropped and a feedback request is issued.	Sample Route feedback should be displayed as entered through test environment.	The exact string is output without any additional cases shown.	Test comment is input into server for test pin drop.