**Testing and Validation**

**Introduction**

Slopes is a mobile app that provides users with up to date mountain information to help them make an informed decision about which mountain to board or ski that day. Slopes is being developed with the Ionic framework, and will be deployed onto both iOS and Android. Our app is written in HTML, Javascript, CSS, and has a database in mySQL. Since we are developing in several languages with different components, we will have to test the code in modules as well as a whole system.

**Verification Strategy**

Slopes is a user interface (UI) driven application. The backend is relatively straightforward, so our focus is on making the user interface as clean and usable as possible. Our team is fortunate to be made up of skiers and snowboarders, so we based our initial UI on our individual needs. Since we started with a strong focus on the needs and wants of boarders and skiers, we will find a group of skiing and snowboarding to use as our focus group for feedback.

In order to get feedback, we have developed GUI mock-ups and a very early interactive version of our app. These will give the feedback group a good idea of how everything will be laid out, where they can find the information they’re looking for, what the final product will likely look like, and more. We intend to get feedback from users on a weekly basis. As we step forward in our design, we will have other people using our app to make sure our development is as user-needs driven as possible.

**Non-Functional Testing and Results**

In order to test the performance of our app, we will be putting it through extensive testing on multiple devices, spanning different OSes. We will also be doing network testing to test app performance on good, mediocre, poor, and no connection conditions. To test scalability, we will test adding more information to our database to see how it affects speed and general performance. Additionally, we will be testing how maintainable our databases are via simulation testing (ie a test where a mountain closes, how does our database react).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test #** | **Requirement Purpose** | **Action/ Input** | **Expected Result** | **Actual Result** | **P/F** | **Notes** |
| 1 | Giving the use information they request | User Requests to look at data | Data is loaded onto page |  |  |  |
| 2 | User is able to input personal information | User Requests to update information | Fields where the user can input information are opened |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | App suggests appropriate mountains based on the user’s personal info | User requests to see the suggested mountains | Appropriate list of mountains is displayed according to the user’s personal information |  |  |
| 4 | App refreshes mountain information so it is up to date | App checks if user has refreshed or recently opened app | App refreshes the data from online accordingly |  |  |

**Functional Testing Strategy**

We will be performing unit, integration, system, and manual testing on our app at various intervals. For unit testing, we will individually test each function we create or edit. We will also run unit tests once per sprint for existing functions for redundancy sake. For integration testing, we will run integration tests every time someone finishes a section of code and wants to add it into the master branch. Additionally, we will run integration tests twice a week on a schedule to prevent bugs from propagating. System testing will happen once a week. Manual testing will happen only in the last several sprints, to smooth out any bugs that exist in the UI, not the overall functionality.

Our test cases for unit testing will be developed by the person who writes the function being tested. The integration tests will be a cooperative effort between the people who work on the individual parts being integrated. Manual tests will be ad-hoc by the test group and the developers. System tests will be developed by the entire team, since everyone will have different levels of understanding of the code.

Our team is using the bug tracking software, Jira to keep track of tasks to be completed, bugs that have been found, and suggestions for implementation. The categories of bugs are as follows: High, Medium, Low, and Critical. High priority is for bugs that could stop a user from using the rest of the app, medium priority is for bugs that affect the UI or functionality, but don’t stop the user from using the rest of the app. Low priority is for bugs that are fairly trivial in nature (such as misaligned text, wrong colour font, etc). Critical priority is for bugs that cause crashes and lock the user out of the app. The statuses for our bug priorities are To Do, In Progress, and Done. To Do is for bugs that we haven’t begun to fix, In Progress is for when a developer is working on a bug so multiple developers don’t try to fix the same bug, and Done is for when the bug has been tested and verified as fixed.

**Adequacy Criterion**

Our adequacy criteria are as follows:

* Make sure that every module has been tested on its own, so we know that each individual module does its job correctly so it doesn’t cause problems in the larger piece of software
* Make sure the existing tests provide adequate coverage, so we don’t end up with edge case bugs and crashes
* Make sure that each major component (ie database, displaying weather data, data parsing) of the app is extensively blackbox tested, so each one interacts with prompts and returns the outputs we expect
* Make sure UI is sufficient first with automated UI testing, then with as many third party users as possible
* Make sure that the user interface has been tested by third party, so we can find UI bugs that we didn’t catch in automated testing that a real user might find
* Make sure that the final application has been tested on multiple network qualities, because the average user might be in a poor network area, be offline or have inconsistent network connection and we want to ensure the apps behaviour is consistent