Testing Strategy Document

UI and End to End Testing

Our approach with UI testing is using a combination of Jest and the React testing library. We will particularly make sure that essential UI components (e.g. titles, descriptions, links, etc.) are in place, mainly using the *expect* API from Jest. On later deliverables where we have finished implementing all the features for our UI we would use snapshots to make sure that the visual aspect of the webpage remains as expected.

Regarding end to end testing we will use the Playwright Library. We will navigate through the pages testing the main function of them, for example being able to drop files, click buttons or links and ensuring that these work as expected. In this sense we will test the Beap Engine API, making sure that the data uploaded is transformed correctly into a .csv file and that the user can use the available ML models to get their predicted files.

Logging

This section briefly describes the motivation for logging in our project, our tools used, and recommends common best practices for developers and quality assurers.

Logging will be used in our project to address two primary concerns which are to (1) understand common (intended and unintended) use case scenarios from real users and (2) to record failures and help identify faults.

We plan to use Rollbar and Crashlytics. Rollbar is a multilevel logging framework that provides a familiar logging API for several languages and technologies including Java Spring, React, and Node.js. A different logging framework will be needed for R scripts.
Crashlytics will be used to record program crashes. This can be deployed on Digital Ocean. Crashlytic reports include logging that shows a user's path and stack trace leading up to the point of the crash.

Logging Severity Levels

Logging messages are expected to be written by both the dev team and the QA team.

Debug - These logging messages should be placed in most meaningful function and function calls. It is recommended to write debug messages with the same frequency and in similar places as assertions. The message types are usually only used in failure analysis.

Info - Record user events and API calls. For instance, “a controller of your authorization API may include an INFO log level with information on which user requested authorization if the authorization was successful or not” (Kuc, 2023).

Warning - This message type needs to be used anytime something *unexpected* occurs even if no functionality is impacted.

Error - This message type needs to be used when there is a failure that impedes one or more functionality but does not cause the program to crash.

Critical - This message type is used when there is a failure that causes the system to crash or prevents the system from running.

What work needs to be done:

Assertions and Debug messages need to be able to be written in only one line of code. This will promote logging and assertion use.

References for logging level section:

Isaiah, A. (2023, November 23). *Log levels explained and how to use them*. Better Stack Community. https://betterstack.com/community/guides/logging/log-levels-explained/

Kuc, R. (2023, November 24). *Logging levels: What they are & how to choose them*. Sematext. https://sematext.com/blog/logging-levels/

Mendes, E., & Petrillo, F. (2021, December 7). *Log severity levels matter: A multivocal mapping*. arXiv.org. https://arxiv.org/abs/2109.01192

Usage

source https://docs.rollbar.com/docs/javascript

```
// Caught errors
try {
  doSomething();
} catch (e) {
  Rollbar.error("Something went wrong", e);
}

// Arbitrary log messages. 'critical' is most severe;
'debug' is least.
Rollbar.critical("Connection error from remote Payments
API");
Rollbar.error("Some unexpected condition");
Rollbar.warning("Connection error from Twitter API");
Rollbar.info("User opened the purchase dialog");
Rollbar.debug("Purchase dialog finished rendering");

// Can include custom data with any of the above.
// It will appear as `message.extra.postId` in the
Occurrences tab
Rollbar.info("Post published", {postId: 123});

// Callback functions
Rollbar.error(e, function(err, data) {
  if (err) {
    console.log("Error while reporting error to Rollbar:
", e);
  } else {
    console.log("Error successfully reported to Rollbar.
UUID:", data.result.uuid);
  }
});
```

Test Coverage Reports

We are using Jest and Playwright, to generate test coverage reports through the CI pipeline. Testing matrix will ensure all features are tested and test coverage reports need to ensure that there is at least 70% statement coverage. This 70% is a metric recommended from CMPT 470 lectures. The test Coverage Reports will be done mainly in code from the backend that we create/modify in the backend and that deals with the main functionality

Note on the use of Mocking:

Starting from ID 2, we will have some mocking pages where we change some items and basic functionality from the original pages. These pages would also contain introduced errors, to test the robustness and correctness of our tests.

Mocking did on ID can be found in the branch: test/101-mockPageFileDropzone

Notes on the use of Mockups:

We have created different Mockups for the pages that we are going to design, so we can be sure that their design is correct and we can ensure usability aspects. These Mockups are part of ID2.

Note on testing of R modules

Since one of the possible features asked to be implemented by the stakeholder was the adding support to upload data from Garmin watches, we may need to test the R module that would interact with this data. This would mainly consist of unit tests done manually. However, this feature would only be done if time allows to develop it.

Note on Spike Prototypes

We would invest in spike prototypes whenever we feel unsure of how the different technologies that we are using would interact between each other. Our first spike prototypes consist of a program that uses React as its frontend component and Java as its backend.

For ID2, we created spike prototypes on the use of playwright and rollbar. We integrated these spike prototypes into our main project.

Note on Smoke Test and Our different kind of testings:

Smoke Tests done for ID1 and ID 2 have basically been manual testing in which we have ensured that the pages that we created are rendered and that one can navigate through the different pages of the app. Additionally, for UI test smokes we rely on the automated tests that we have created using playwright and jest.

Regarding test smokes for functional features we will perform manual tests for example to make sure that we can upload files to the page and then download it as a .csv. In this case the first smoke test would be to ensure that the file that we uploaded gets into our database. In the same way we will test that the predicting ML algorithms actually produce different files of data.

As we have mentioned before, one of our main investments in testability is using a combination of manual and automated testing, which is currently done on the frontend(the main part where we have done changes). We will perform the same thing once we refactor the backend of our program with the new modules that we create. However, we have also invested in making our pipeline safe. Automated testing is part of the CI/CD pipeline because for every time we commit or create a pull request, all our unit and end to end tests will run. Finally, we have also invested in containerization, since we have moved our complete system to Docker.