Togetherly

QA and Test Plan

The goal of this quality assurance and test plan is to establish a set of processes and standards that, when followed by the team, will improve our chances of building a functional, high-quality, robust, and maintainable product.

QA Processes

Main and Feature Branches

To prevent accidental conflicts, we have opted to protect the **main branch** of our repository from being pushed to directly. Instead, all work is to be done on **feature branches**, which should generally be named based on the feature or fix for which they were created. In general, only one person should work on a given branch at a time, except where there is very close (i.e., in-person) coordination between developers. Local feature branches should be periodically synced with the corresponding remote branches on the GitHub repository.

Pull Requests and Code Reviews

Merging a feature branch into the main branch must be done via a **pull request (PR)**. A PR should contain a brief yet meaningful title, as well as a concise description that adequately communicates what changes are present and, as necessary, why those changes were made. It is also helpful for the description to contain links to any relevant tasks on Asana.

Before a pull request is merged, it must be approved by another team member. Before giving their approval, that team member should perform a **code review**. At a minimum, a code review involves looking at each of the files that was changed, executing *all* automated tests, and performing manual testing as needed (both forms of testing are described in detail later).

Once a developer has created a PR, they should **request a review** using the “Reviewers” feature on GitHub. In general, the QA lead is responsible for reviewing and approving PRs. However, if they are unavailable to review an urgent change, or if the QA lead needs a review for their own PR, another qualified member of the team may be requested instead. Once a reviewer approves a PR, they should merge it into main and delete its associated branch.

Important: Once a review has been requested, no commits should be pushed to the branch associated with the PR *except by the reviewer*. This is to allow the reviewer to make minor adjustments prior to approving and merging the PR (such as resolving merge conflicts) without having to pester the author of the PR. However, if a reviewer does request changes, they should not push any further commits to that branch until the author has again requested their review.

Test Plan

Automated Testing

Wherever there is non-UI code that involves non-trivial logic (i.e., is more than just interfaces, stubbed methods, getters, or setters), there should generally be **automated unit tests**. The tools and techniques that we will use for automated unit testing are (as of March 23rd) still being evaluated. Once we have settled on those tools and techniques, unit tests should be included as part of any pull request that introduces or meaningfully modifies non-trivial, non-UI code. Unit tests are responsible for evaluating the **functionality** of the system, ensuring that any non-trivial components function as intended**.**

Once we have reached a point where we may begin writing unit tests, reviewers who are evaluating PRs should consider 1) whether unit tests should be present given the changes being made, 2) whether such tests have been included in the PR, and 3) whether those tests appear adequate given the code that was changed.

Assuming that the reviewer deems the tests adequate, **success** for automated testing will be indicated by all unit tests passing when run in the reviewer's development environment. A PR should not be approved without the reviewer first verifying that *all* unit tests pass.

Manual Testing

Because UI code does not lend itself well to automated tests, it will instead be tested manually. **Manual testing** should be performed when reviewing any PR that contains UI changes. Such testing consists of running the app in one's development environment and observing/interacting with each of the new or updated UI components.

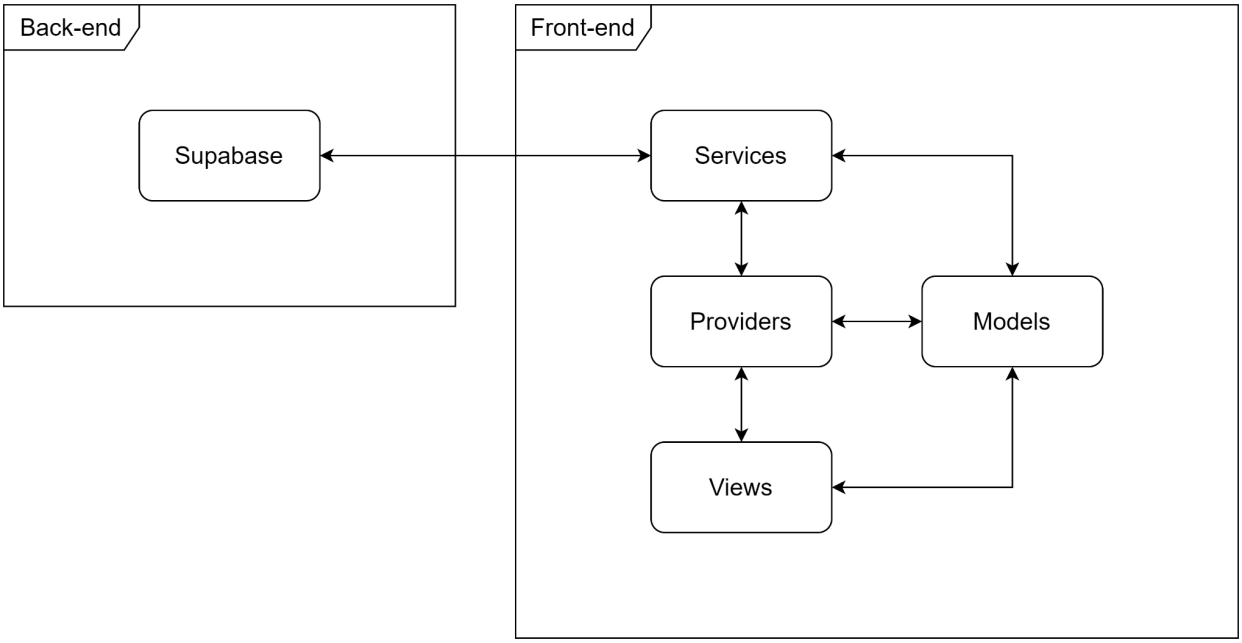
Manual testing of the UI is much more subjective than automated testing. **Success** for manual testing will be determined by the reviewer's satisfaction with the **performance**, **reliability**, **user-friendliness**, and **appearance** of the user interface.

User Acceptance Testing

In preparation for the final demo, a current build of the app should be distributed to a small group of testers to perform **user acceptance testing**. The group of testers may include members of the team, as well as their friends, family, or acquaintances. Testers should be asked to use the app in either a hypothetical setting or, when reasonable, a real-life setting. Following their experience, testers should be invited to share any frustrations or suggestions they might have. Informal user acceptance testing may also occur throughout the development process.

Success for this category of tests will be measured qualitatively based on tester feedback. If testers find the app useful and are not frustrated by any major issues, then this test will generally be considered a success.

Testing of System Modules



Back-end: Supabase

Our Supabase tables and configuration exist outside our GitHub workflow, and as such, will not be reviewed in the same way as other code. Instead, the Database Developer should notify the QA Lead when significant tasks relating to the back-end have been completed. Once notified, the QA Lead will manually check the schemas, configuration, or other work done in Supabase for possible issues. If any issues are found, they will contact the Database Developer over Discord to request clarification or changes.

Front-end

Services

Service classes integrate with Supabase, and as such, testing them will require some degree of mocking. Service classes should accept a SupabaseClient object via their constructor. In production code, this object will be provided via Supabase.instance.client. However, in unit tests, it will be replaced with a mock object.

The following is a non-exhaustive set of Service behaviors that should be covered by unit tests:

- Translation of query results into model objects (e.g., read operations)
- Translation of model objects into query parameters (e.g., create/update operations)
- Translation of method parameters into query parameters (e.g., delete operations and specialized queries)

Providers

Providers act as the bridge between the UI and Service layers. They are also responsible for managing app-wide state. Because they implement most of the app's business logic, Providers are the most critical component to unit test. The range of behaviors that will need to be tested is too broad to list in this document, but it should suffice to say that any non-trivial behavior should be covered.

Note: We failed to make enough time for this portion of our test plan, and as such have set aside plans to write unit tests for our providers.

Views

Views should manage very little state and should implement practically no business logic. While methods do exist for testing aspects of a UI, such testing is excessive given the scope of our project. Instead, changes to the View layer will be tested manually as outlined in the previous section.