## Software Implementation of Planned User Stories

To effectively tackle our planned user stories, we broke down their implementation into three phases. The first phase involved conceptualizing and documenting the user stories. We used Github issues as well as Word documents to detail our user stories, including their difficulty, their descriptions, and their acceptance criteria. Then, we developed the user stories and split the work into frontend and backend. The work was assigned to team members who then took on the task of developing their part of the assigned user story. Finally, we ran unit tests on the implemented code to ensure it met the acceptance criteria and to ensure the code could handle different input.

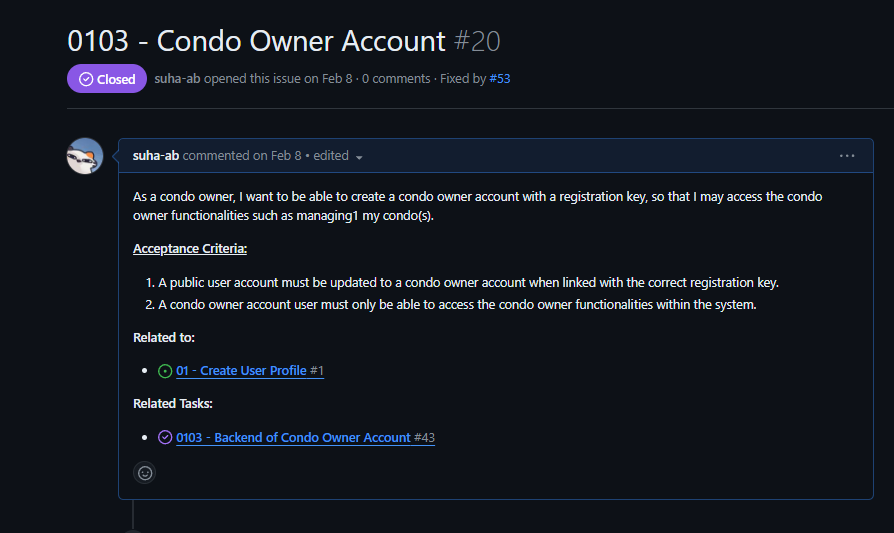


Image 1: Phase 1 of Implementing User Stories

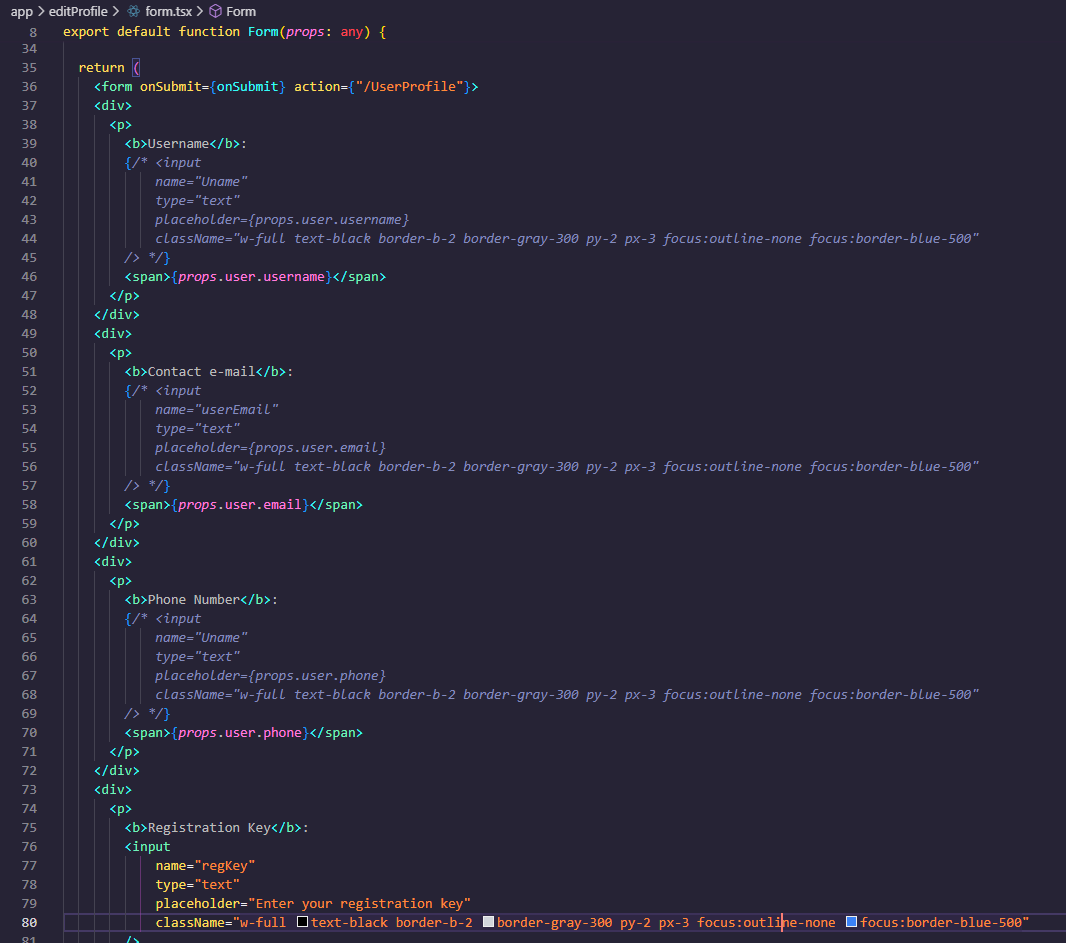


Image 2: Phase 2 of Implementing User Stories

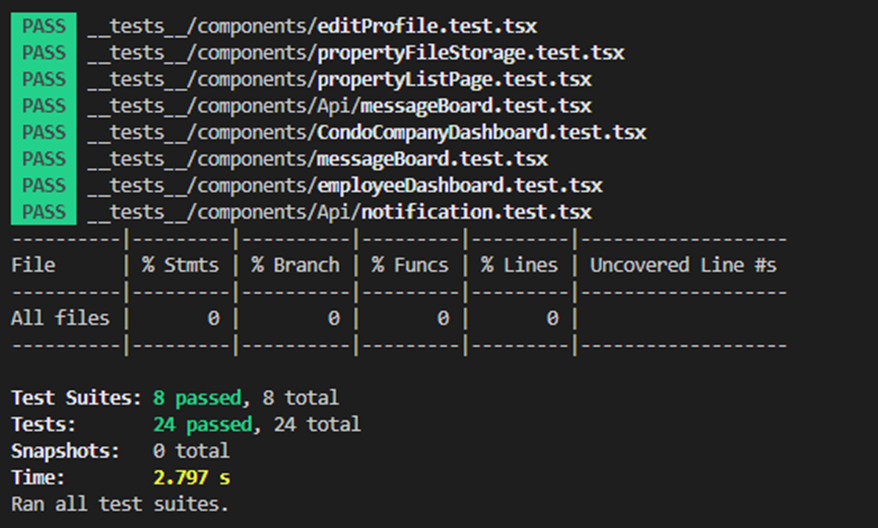


Image 3: Phase 3 of Implementing User Stories

## Bug Reports and Fixing

We are using Github’s issues to track, report, and resolve bugs. When a bug is found during testing or development, a Github issue is opened with the label “bug”.

Bug severity is assessed according to the impact of the bug on the system and its functionality. A blocker or blocking bug is a bug that forces the system or functionality to be temporarily unavailable. Critical bugs are bugs that severely impact the functionality of the system, but do not block it completely. Major bugs are bugs that cause noticeable problems or inconveniences to users, but are not as severe as critical or blocking bugs. Normal severity bugs are those that have a moderate impact on a software’s functionality, are noticeable, but do not impact the core functionality significantly. Normal severity bugs are the most common during development and testing. Minor bugs are bugs with minimal impact on software functionality, but do not affect critical functionalities. Trivial bugs are those that have very little impact on software functionality and are usually not fixed immediately. Finally, enhancements are suggestions for improving the software’s functionality.

| **Bug ID** | 01 ([Github Issue #44](https://github.com/Ryan30012/AnaCondo/issues/44)) | | |
| --- | --- | --- | --- |
| **Originator** | Suha Abubakr | Email: [s\_bubak@live.concordia.ca](mailto:s_bubak@live.concordia.ca) | Signature :Github |
| **Submit Date** | March 17, 2024 | | |
| **Summary** | Navigating to user profile without being logged in throws an error. | | |
| **Severity** | Critical (the software is still usable, but the bug is urgent and should be addressed immediately) | | |
| **Product** | Website (Anacondo) | | |
| **Component** | User Profile Page | | |
| **Version** | 1.0 | | |
| **Platform** | PC | | |
| **OS** | Windows | | |
| **Browser** | Google Chrome | | |
| **URL** | localhost:3000/UserProfile (website is running locally) | | |

Table 1: Bug Report 01

| **Bug ID** | 02 ([Github Issue #46](https://github.com/Ryan30012/AnaCondo/issues/46)) | | |
| --- | --- | --- | --- |
| **Originator** | Suha Abubakr | Email: [s\_bubak@live.concordia.ca](mailto:s_bubak@live.concordia.ca) | Signature :Github |
| **Submit Date** | March 17, 2024 | | |
| **Summary** | Submitting Empty Post Request for Registration Key throws NaN error | | |
| **Severity** | Critical (the registration key functionality is still usable with other values, but the bug is urgent and should be addressed immediately) | | |
| **Product** | Website (Anacondo) | | |
| **Component** | Edit Profile Page, Registration Key Field/Form | | |
| **Version** | 1.0 | | |
| **Platform** | PC | | |
| **OS** | Windows | | |
| **Browser** | Google Chrome | | |
| **URL** | localhost:3000/editProfile (website is running locally) | | |

Table 2: Bug Report 02

## Defect Tracking Tool

For defect tracking, we opted to not use an external tool, because we felt it would over complicate things to have to track the bugs in an external platform. Instead we felt it would be easier to manage bugs if we tracked them directly in GitHub.

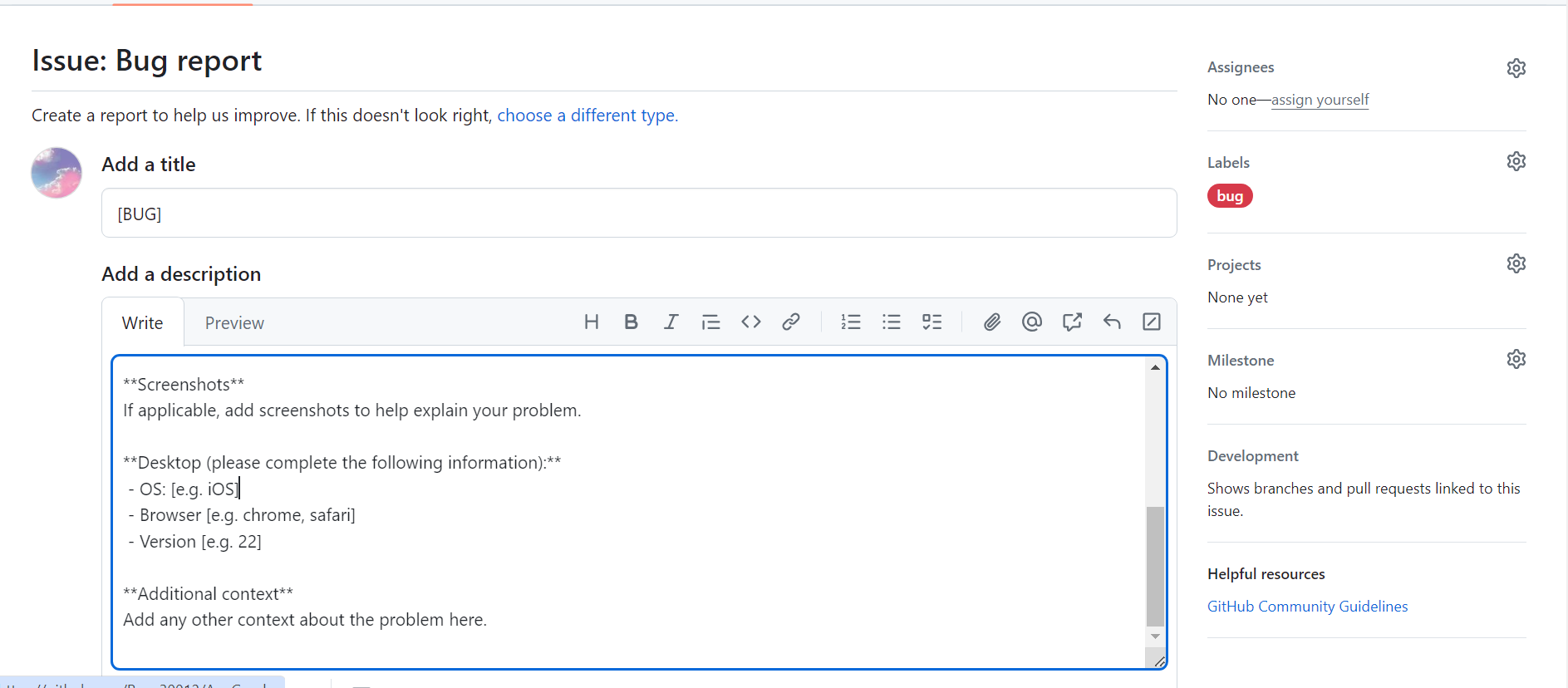
To streamline the bug tracking process in GitHub we have used a bug issue template. A screenshot of the template is below:  


Image 4: Bug Report Template

A proper example of how this template can be used is in the bug: ([Github Issue #46](https://github.com/Ryan30012/AnaCondo/issues/46)), which is shown in the following screenshot:

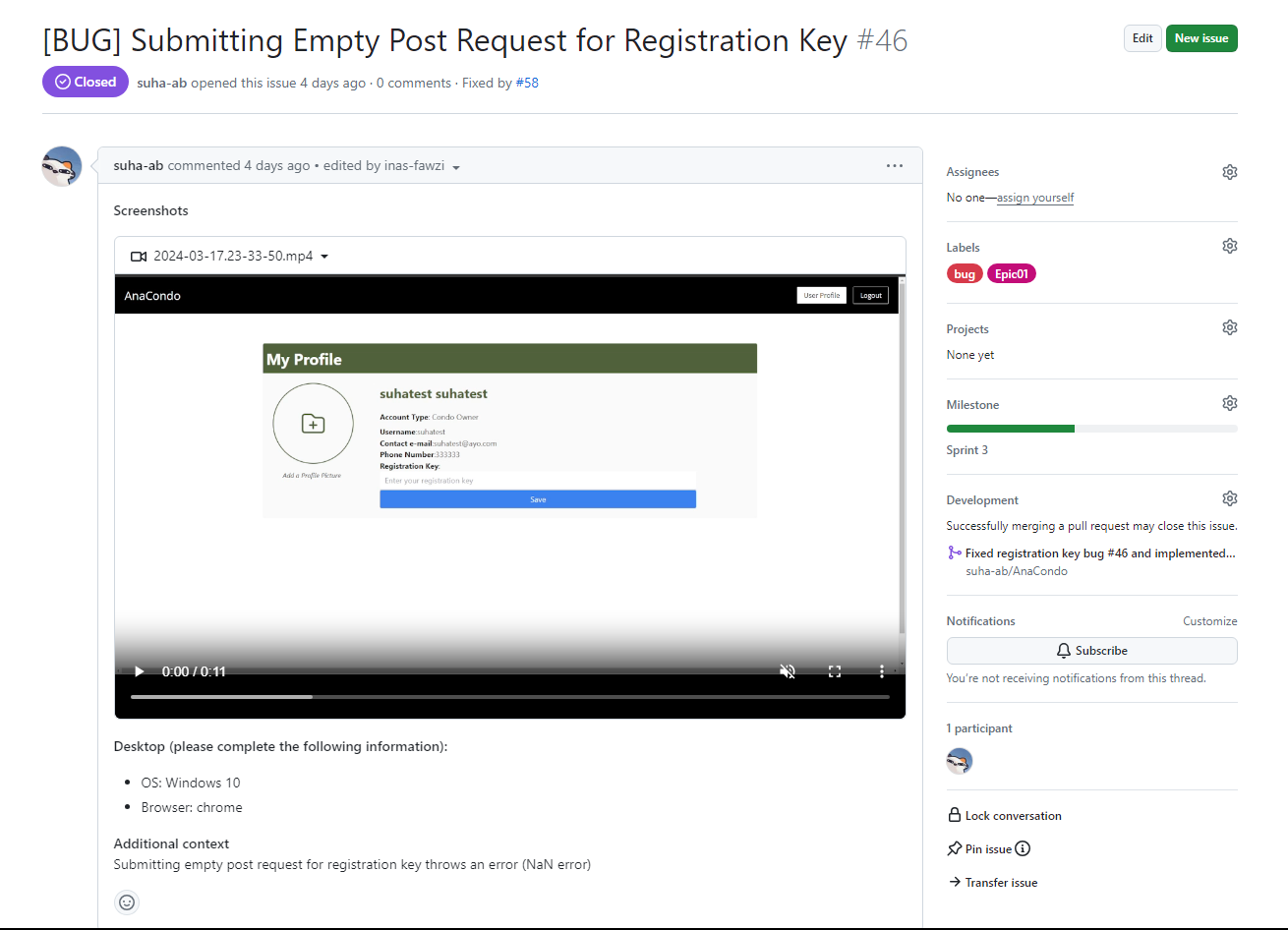


Image 5: Bug Report Example

## Code Review Tools

Our main approaches to code management are pull request reviews and an automated service that evaluates our code, called Codacy.

For every pull request made a reviewer is added and the pull request is not merged until the reviewer has approved the pull request and added any comments they may have. If the reviewer disapproves the pull request the author must amend their code, or further explain their code to the reviewer, in an effort to get their pull request approved.

Aside from pull request reviews, we also occasionally do in person peer to peer code reviews, mainly between individuals working on related pieces of code. During code reviews the person explains their code and makes sure it can be understood by the reviewer. This serves to improve our code but also make sure that developers working on related parts can have a deeper understanding of each others’ work.

Codacy is used to automatically measure complexity and duplication. Codacy will also open issues to annotate pull requests based on problems found in the code.

Below is our code quality report generated by Codacy:

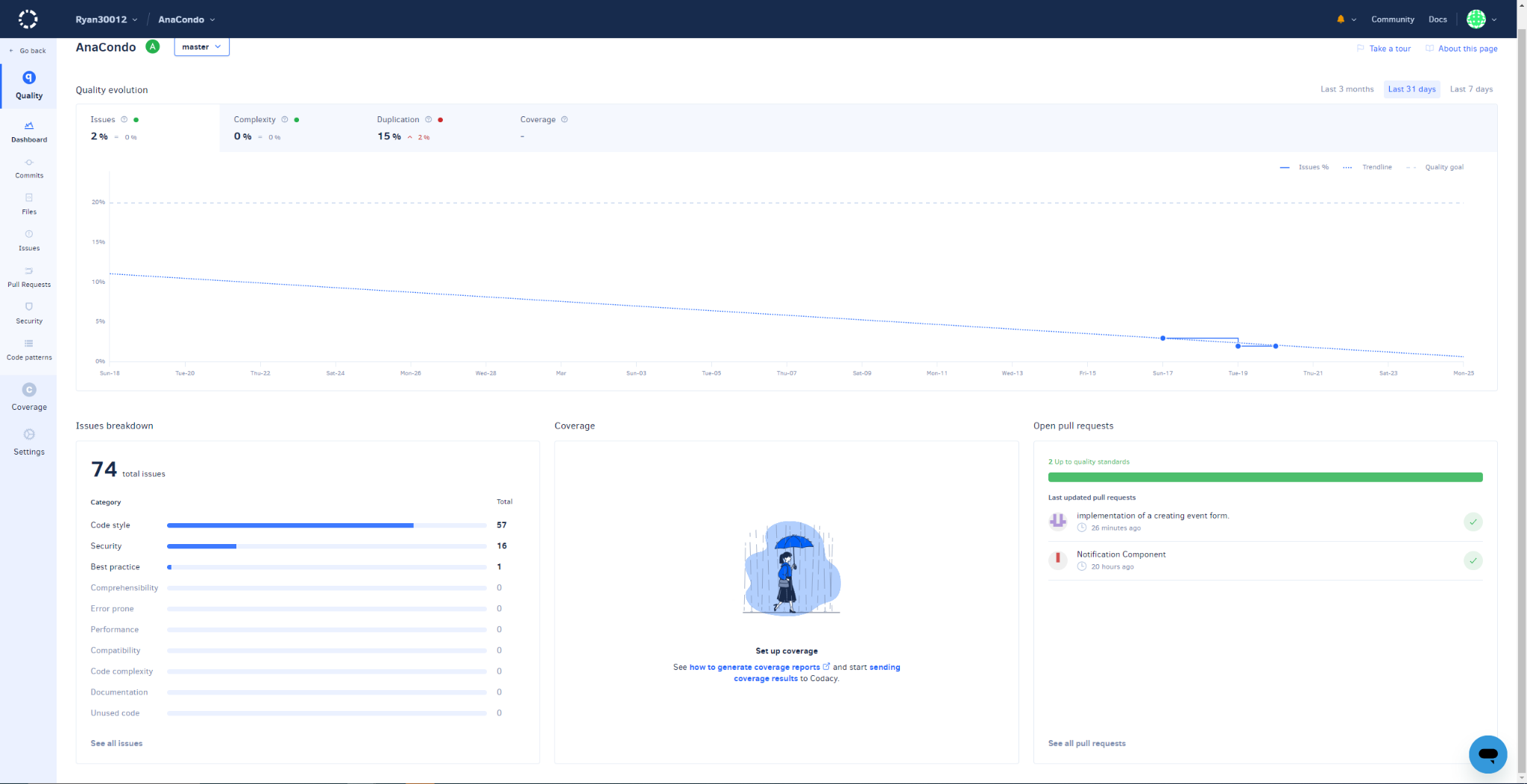


Image 6: Codacy Code Quality Report



Image 7: Codacy Code Quality Report

## Usage of Design Patterns

In terms of design patterns, we have used one when implementing the condo owner dashboard. When creating this view, it requires the incorporation of several components to ensure that this user can view all the necessary components. For instance, it incorporates the components of RentalPropertyCard, RentalFinancialStatus, and RentalSubmittedRequests. It adapts these components into its own structure and adapts them to the user interface seen by the condo owner upon login. We had used the Adapter design pattern to create this structure when designing the condo owner dashboard. Additionally, the prototype pattern was used many times during our project. In fact, there were many cases where we have cloned the prototype due to needing reuse of code. Next, we have implemented the composite pattern for our users. It was necessary to define our user types in a hierarchical way. Finally, we used the strategy pattern to only allow certain features for certain types of users.

## Code Coverage

As of sprint 3, we had aimed to achieve 70% code coverage as to be able to properly manage our upcoming sprints. To achieve this feat, we resorted to user story selection and separation amongst teammates. In fact, we managed a spreadsheet to indicate which team member would be given the responsibility for which user story completion, and have each worked on the completion of that user story. As for test cases, each team member was responsible for the test case related to their own user story in order to increase clarity of separation and team morale. Finally, concerning the database, we have achieved 90% coverage as it is necessary for the implementation of features.

## Design Quality as Measured

Design quality can be further measured by metrics such as lines of code, number of packages, coupling between objects, and cyclomatic complexity. To retrieve our codebase’s lines of code, we used Codacy. Our repository currently has 27,628 lines of code. For our software, we did not have objects. Rather, we had typescript files that represented website routes. The coupling between the files is low since very little of them import each other. The files that require importing of other typescript files are placed in the same directory. If one file is a component, however, it is placed in the components directory and imported. For cyclomatic complexity, that can be measured using tools such as Codacy. Cyclomatic complexity measures the complexity of code by providing a quantitative measure of the number of linearly independent paths through the program's source code.

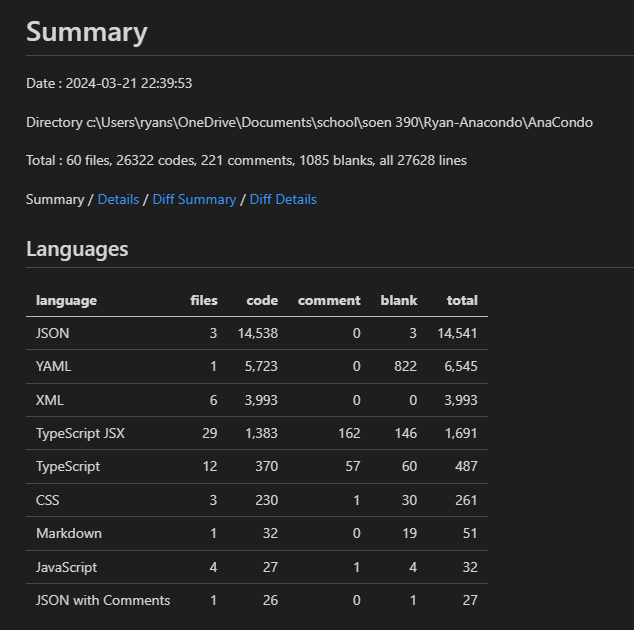


Image 8: Codacy Metrics Report

## Quality of Source Code Documentation

To document our source code, we mainly resorted to inline comments where necessary. This was particularly useful when we needed to ask each other for help in debugging or implementing certain functionalities. An example of inline comments used in our code is provided below. JSDoc comments would be very useful for understanding functions as a whole, however, we currently do not have any JSDoc comments in our source code. Moving forward, we can gradually add more JSDoc and inline comments to improve our source code documentation.

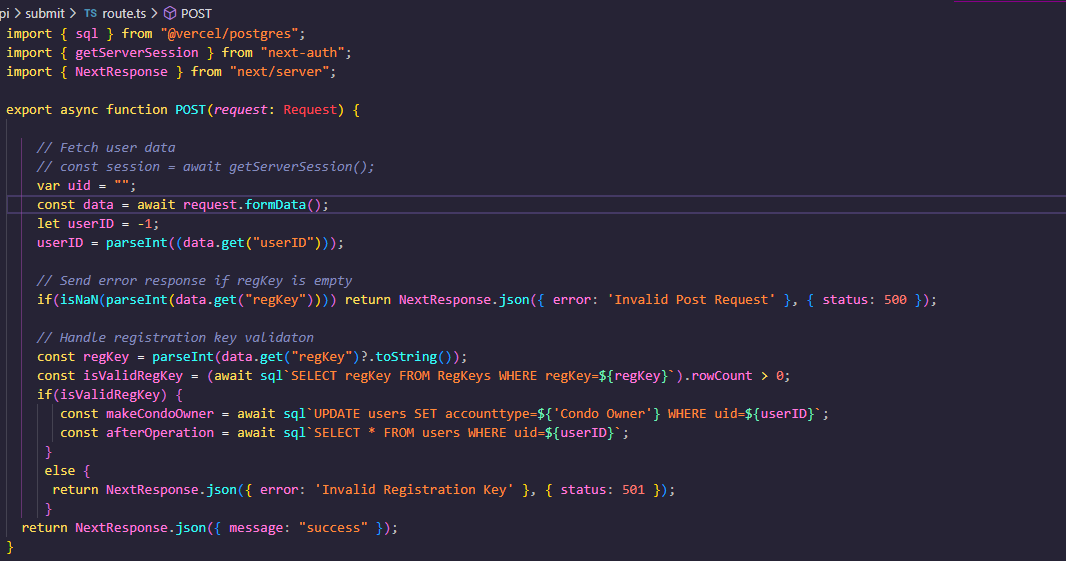


Image 9: Inline Comments in Source Code

## Refactoring Activity Documented In Commit Messages

As we are still in the midst of developing all the features for our product, refactoring has not been at the forefront of our sprint plans. However, there are a few instances of refactoring occurring within the code source storing our project. In this [first instance](https://github.com/Ryan30012/AnaCondo/commit/1bbcb991786b14f476ea689ca2b925074f2f72e5), the biggest trend was the removal of redundant code as well as the addition of shortened segments of code. While original functionality is preserved and improved, these smaller modifications allow for a quicker understanding of the file’s main feature. In this [second feature](https://github.com/Ryan30012/AnaCondo/commit/d95a0866fea3b37b142ec8aa89df048096f90fc8), code no longer deemed useful or relevant to the implemented function was removed from the code and some replacements were made to ensure that the current function works as intended. In this [third feature](https://github.com/Ryan30012/AnaCondo/commit/7cf9e91d118f52029220ae1015ae84790de92286), a focus on providing more readability through comments was made. Parts of the functionality that were not functioning were modified so that they would build. To keep record of the database usage for this feature, the sql commands were commented out for the sake of the future developers that will take over this portion of the code.

## Quality and Detail of Commit Messages

A good commit message should indicate what change was made. Our team has agreed that commit messages should always mention exactly what was changed, and to avoid having commits that contain massive amounts of changes so that the commit message can effectively convey what the changes were. Unfortunately, our team is not consistent yet in making good commit messages. Some of them are better and others are worse. For example, a good commit message would say something like “fixed edit button in user dashboard” rather than “small fixes to user dashboard”. As of now our team has made both good and bad commit messages. We will aim to have more consistent quality to our commit messages as a whole going forward.

## Use of Feature Branches

We have not been using feature branches. Instead each person in the team has their own fork of the repository which they push to, and then perform pull requests between their fork and the main repository. The reasoning behind the fork system is because each member deployed their website on Vercel to gain access to their own database while developing. This allowed us to work with independent databases when needed. It may be difficult for us to implement feature branches at this point due to our forking system. However, a possible way to use feature branches going forward would be to have members of the team create feature branches from within their respective forks. This may however be a redundant practice since the point of feature branches is to isolate specific features from the main repository for multiple people to work on the branch, and then perform pull requests directly to the main repository from the feature branch. Having the forks as a middle point in this process somewhat removes the point of having feature branches since team members do not collaborate in one another’s forks. Moving forward, we aim to use feature branches over our current fork system.

## Atomic Commits

The following formats are to be used when contributing to the repository through pull requests and commit messages to ensure the code quality as well as a means to keep up the code coverage.

### Pull request message format:

## Describe your changes

## Issue number

## Checklist before requesting a review

- [ ] I have performed a self-review of my code

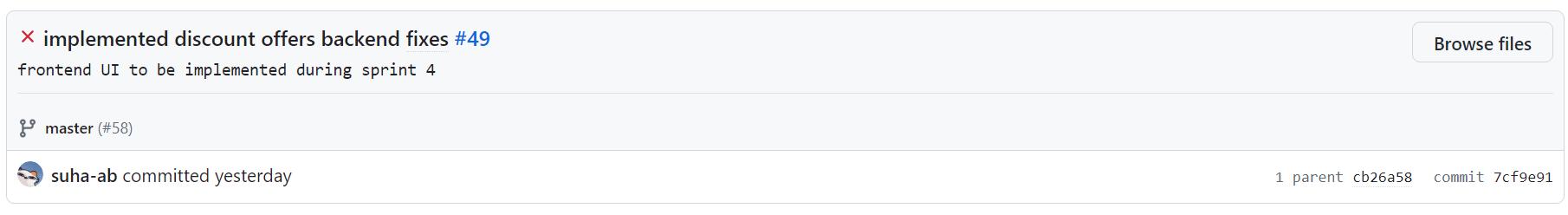
- [ ] There are no code conflicts with my current PR

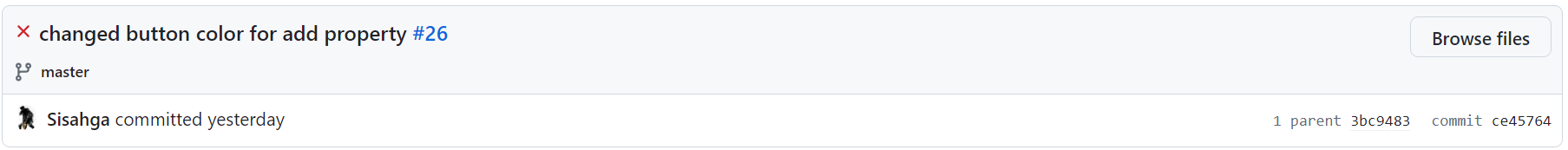
- [ ] There has been at least one unit test added for this feature being implemented/fixed

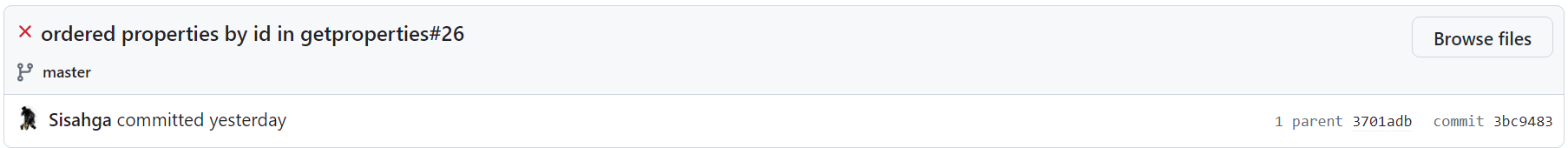
- [ ] I have gotten a peer-review of my code

### Commit message format:

**[verb** eg: implementing/ed, updating/ed, adding/ed, fixing/ed**] [specific work item** eg: edit button, table rows, profile picture, login api…**](***optional:* **in [overarching area of work** eg: user dashboard, profile page…**]) [#issue number]**







Images 10-12: Good Commit Messages

## Linking of Commits to Bug Reports/Features

Ideally, all commits and pull requests should be associated with an issue in GitHub for tracking purposes. This is a part of the process our team has agreed upon for doing commits and pull requests. However, this is another area where we are lacking consistency. Many commits and pull requests are properly linked to an issue, but there are many that are not linked as well. Going forward we will try to become more consistent. The format we are aiming to fully implement is for the commit message to be a description of the changes made to the files followed by the issue number. An example is given below.



Image 13: Linking Commits to Features

## 