

SOEN 390 – Sprint 4

Code Management Documentation

1. Quality of source code reviews

The quality of source code review is a very important process in web development. The source code reviews were done for every pull request. At least one reviewer is required to review the changes and inspect code quality such as respecting the coding conventions.

Here is an example of a pull request with source code reviews:

<https://github.com/CONCORDIA-SOEN-390/Condo-Mgmt-Web-App/pull/206>

The reviewer leaves comments on sections of the code that require improvement. The creator of the pull request takes into consideration the comments and refactors the code. The pull request is only merged when the reviewer approves the pull request. For a major pull request, such as code refactoring, the team has a meeting to review all changes.

Figure 1. Source code reviews and comments and the pull request author's refactorings.

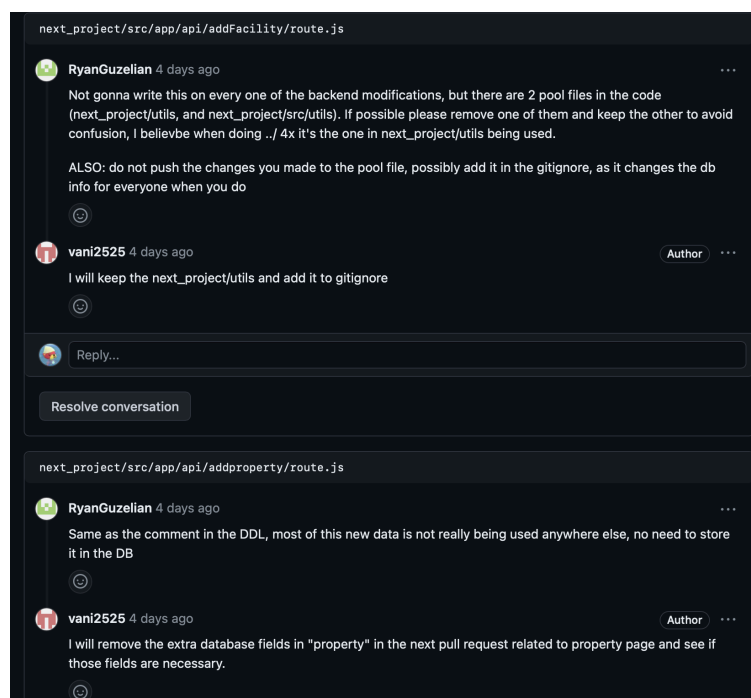
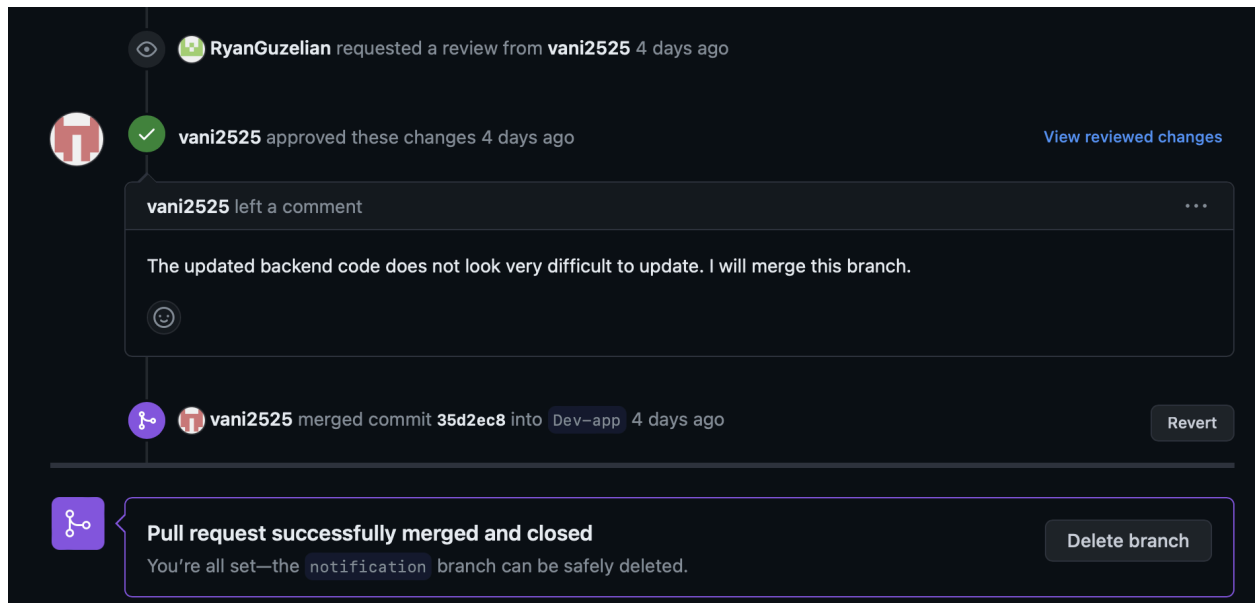


Figure 2. Reviewer's approval of the pull request



2. Correct use of design patterns

In order to follow proper software engineering techniques, and considering our tech stack, we decided to follow the MVC (Model-View-Controller) pattern. This would solidify our use of TypeScript, since under the hood, it relies on JavaScript's prototype-based inheritance, which slightly strays from known OOP concepts.

Model:

The model is represented by our database structure, which uses PostgreSQL to define data models and perform CRUD operations.

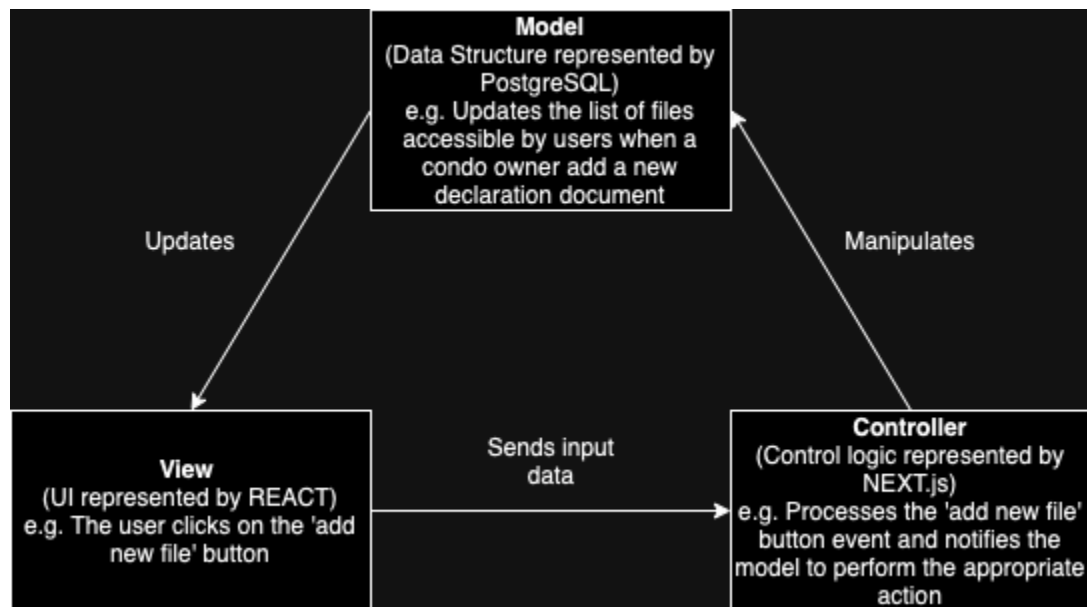
View:

The view is represented by our React components, which present data to the user and handle the interactions with them.

Controller:

The controller is represented by the API routes created using Next.JS, which allow interaction between the application and server-side functions.

Figure 3. Model-View-Controller



3. Respect to code conventions

Coding conventions are programming language-specific guidelines that offer recommendations for keeping your code clean. Code conventions promote code consistency. This is especially important when multiple developers are working on the same project. Thanks to code conventions, different teams can avoid unnecessary confusion and reduce the likelihood of errors caused by inconsistencies in coding styles. Moreover, appropriate conventions and practices ensure that any developers who will work on the application in the future can easily understand the code.

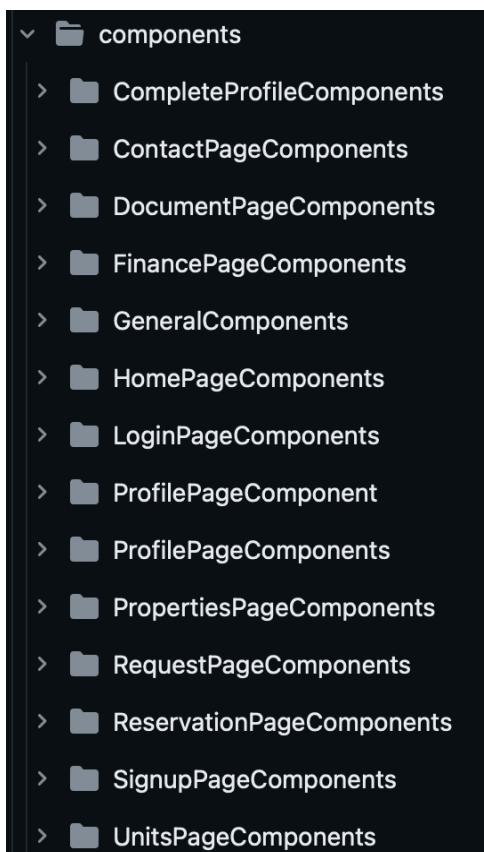
Guideline followed examples:

- Avoid one-letter names
- Nested code should be indented
- Avoid code duplication
- Avoid multiple inheritance
- Lowercase letters represent variables for which you must substitute information or specific values.

Figure 4. Example of code convention: Descriptive lowercase variable names with comments

```
//-----HARDCODED VALUES-----  
const userId = 1; // company user  
const accountType = "company";  
//const userId = 3;  
//const accountType = "reg_user";  
//-----
```

Figure 5. Separate components files for code reusability



4. Quality of source code documentation

Good source code documentation is essential for ensuring that the code written has logic, intent and function. Proper documentation reduces the risk of misinterpretation and costly errors. In addition, it promotes productive and efficient collaboration between different teams when bug fixing, code reviewing, and other maintenance processes.

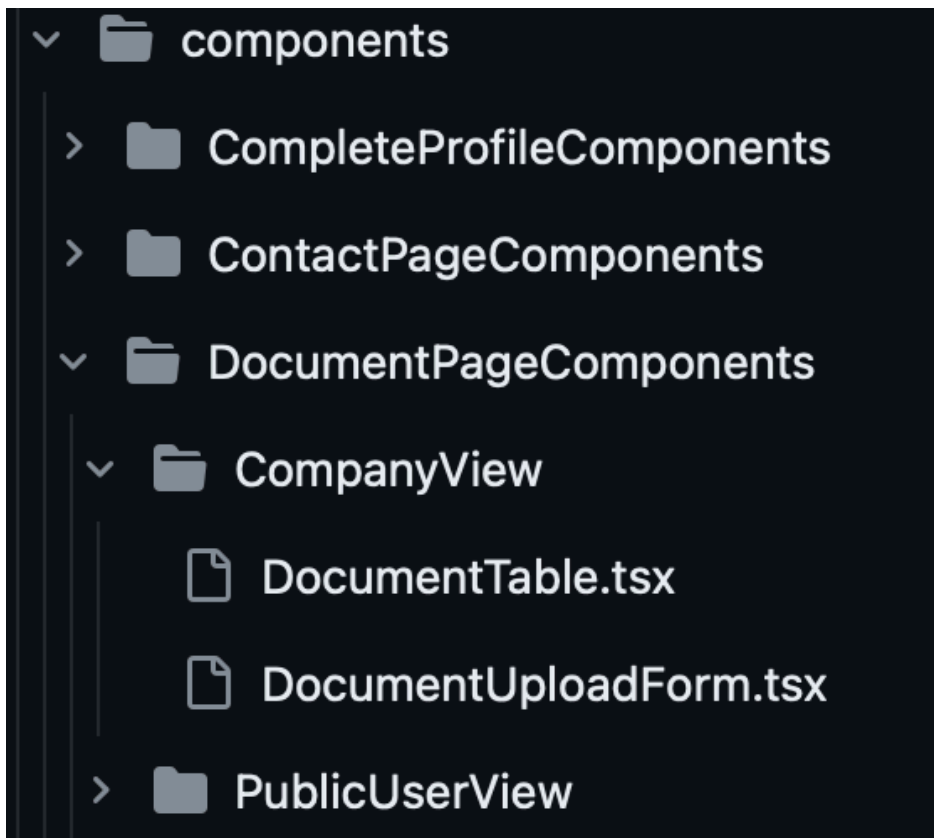
Guideline followed:

- Clearly state the intent and goal of the code, along with any unique functions, classes, or modules.
- Use easy-to-understand language so that both veterans and beginners can understand it
- Use the proper headers, subheadings, and sections to arrange your material.
- Provide references
- Comments

Figure 6. Example of comments

```
// page in progress
//-----FIX PAGE RENDERING HERE -----
const page = 'company';
//const userId = 3 // reg_user
const userId = 1; // company user
//-----
```

Figure 7. Example of clear files names and category names that convey the goal of the code

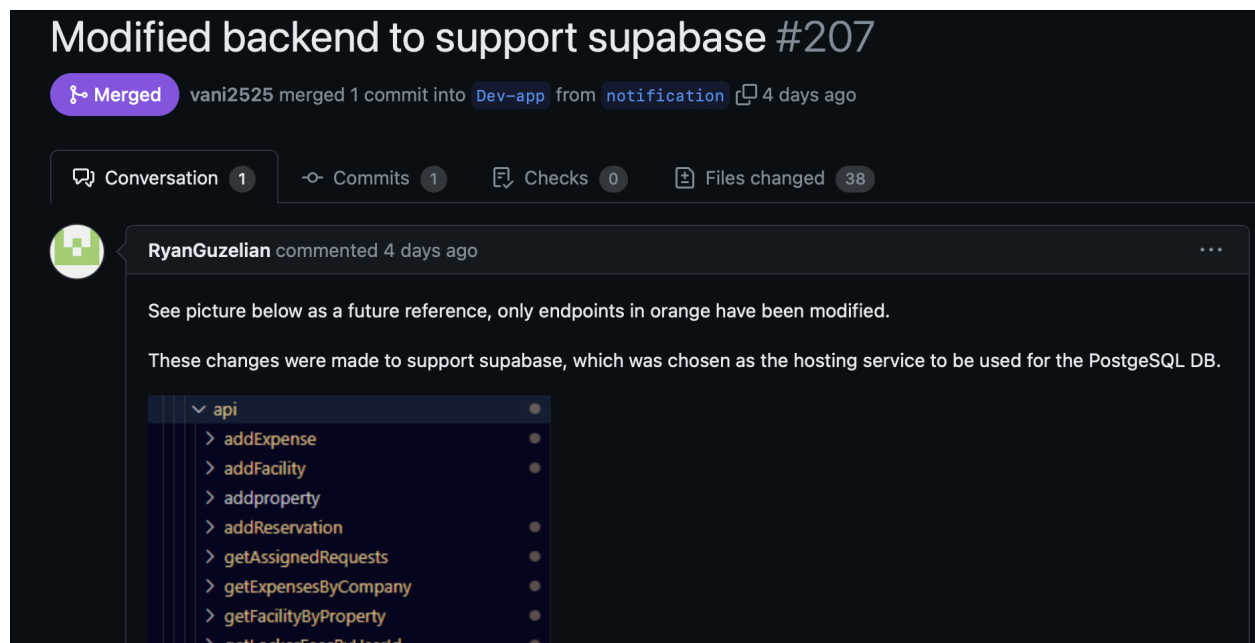


5. Refactoring activity documented in commit messages

The team undertook significant refactoring activities to enhance the project's codebase. Key efforts included file restructuring, standardization of front-end components code, replacing redundant codes with reusable components. These changes were driven by focus on improving the system's maintainability and scalability, providing a clear and structured basis for future development.

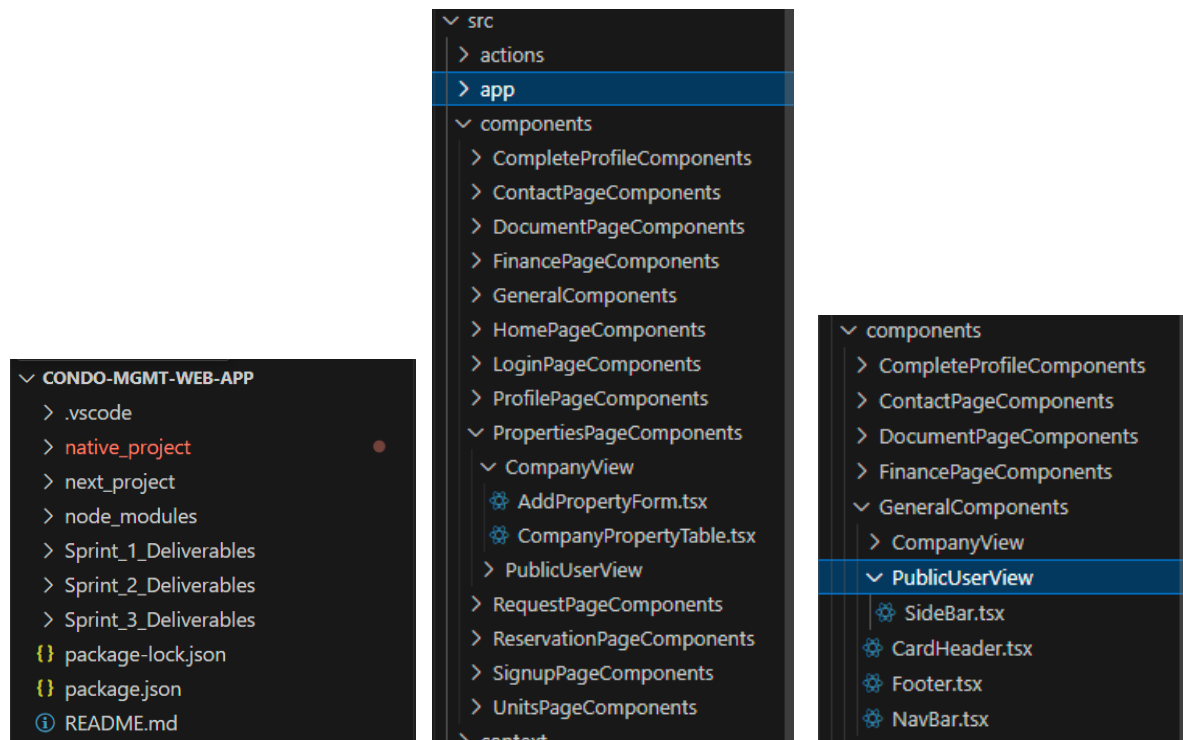
Furthermore, for sprint 4, code refactoring was performed in order to support Supabase as our hosting service for our PostgreSQL database.

Figure 8. Pull request opened to document and review the changes made



<https://github.com/CONCORDIA-SOEN-390/Condo-Mgmt-Web-App/pull/207>

Figure 9. Structure of the project



6. Quality/detail of commit messages

A good commit message provides an essential description of the changes introduced by the commit. Its goal is to explain the change, while providing enough context for others (including future you) to understand the code without having the need to inspect code.

Guideline followed:

- Add a significant title.
- Describe the commit.
- Do not includes personal information, such as passwords
- If a commit solves a bug or closes an issue, specify it.
- Avoid massive sizes. There is no limit to how much text may be stuffed into the commit body. However, a giant message would destroy performance.

Figure 10. Examples of short but descriptive commit titles

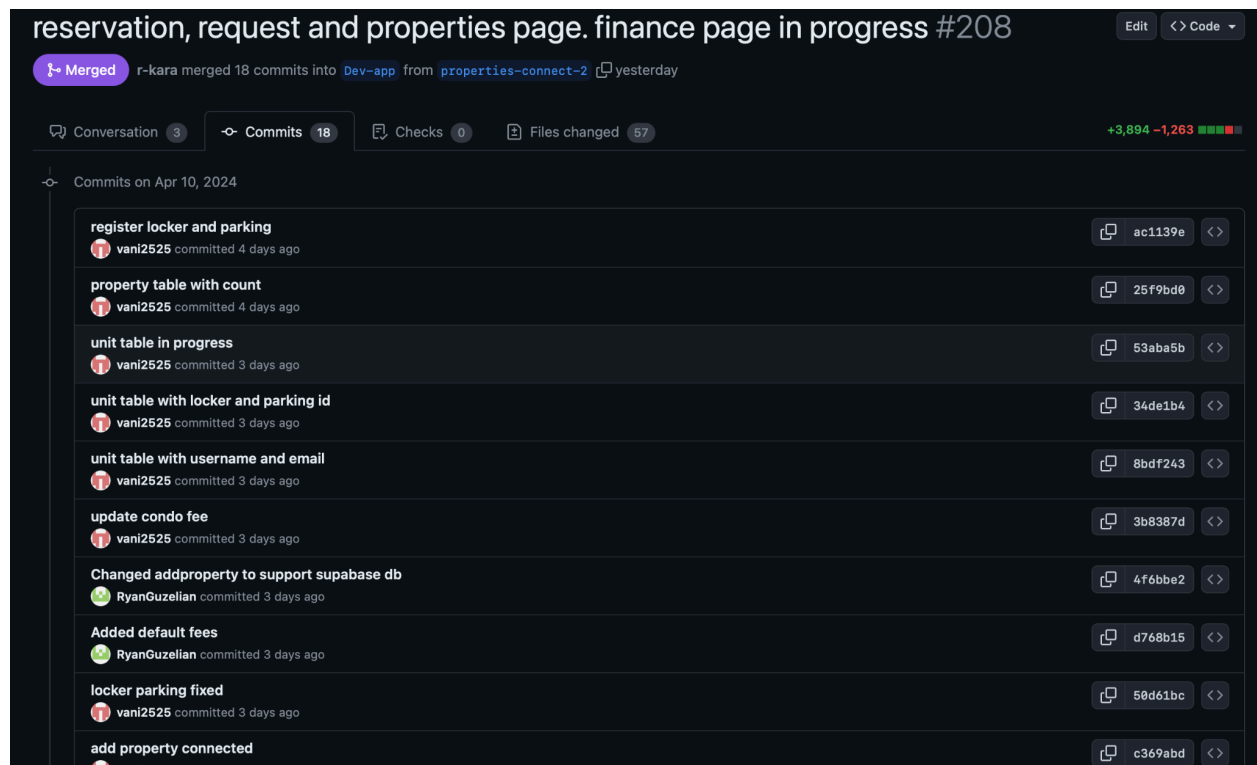
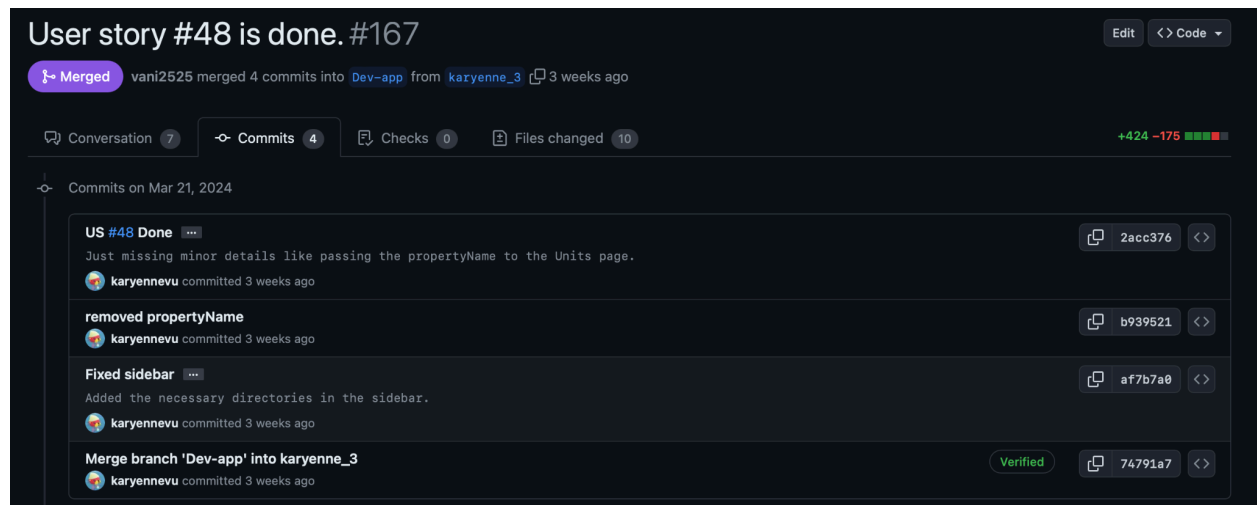


Figure 11. Example of commits with comments to provide further information



7. Use of feature branches

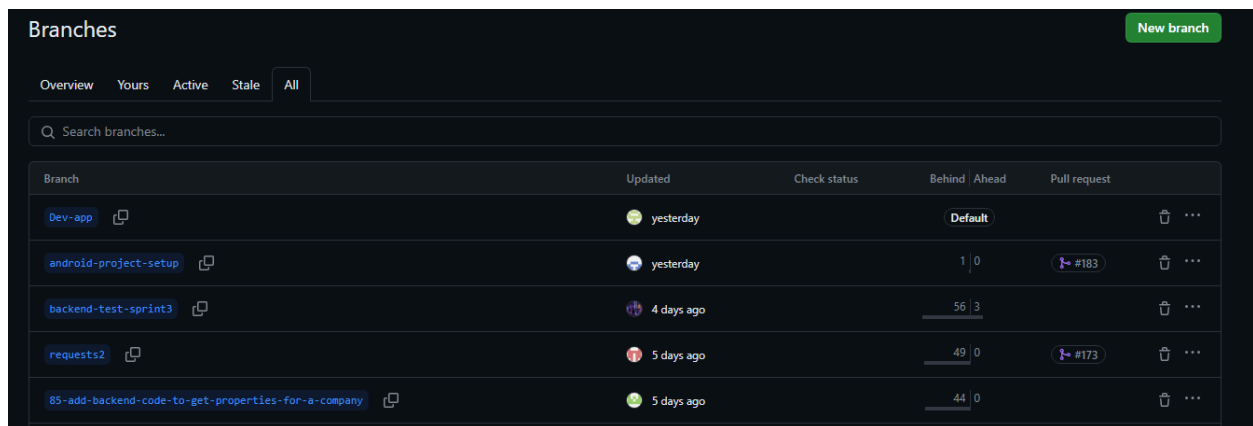
The use of feature branches is an essential example of good software development practice, especially when working in a group setting, to avoid negative results. It allows developers to work on new features of an application or a software without disturbing the progress already made. Without making feature branches, someone could easily delete the changes made and make the team lose all its progress which would consequently delay the delivery of the project.

Additionally, feature branches ensure that the new code has to be reviewed in a pull request first before merging it into the main branch to avoid conflict. The review is also usually done by someone else on the team which makes it so that the person merging their code needs the approval from at least one teammate before merging. This ensures that the person who worked on the feature branch can't just decide things on their own, and it also allows them to receive feedback on their work.

Guideline followed:

- Descriptive feature branch titles
- Existing pull requests for code review before merging
- Different branches for different features

Figure 12. Example of feature branches used for this project on GitHub



The screenshot shows the GitHub 'Branches' page for a repository. It features a dark theme and a 'New branch' button in the top right. Below the navigation tabs (Overview, Yours, Active, Stale, All), there is a search bar. The main content is a table listing several branches. Each row includes the branch name, its update status (with a commit icon and time), a 'Check status' link, a comparison bar showing 'Behind' and 'Ahead' counts, a 'Pull request' link (if applicable), and a menu of actions (trash and three dots).

Branch	Updated	Check status	Behind	Ahead	Pull request	
Dev-app	yesterday			Default		
android-project-setup	yesterday		1	0	#183	
backend-test-sprint3	4 days ago		56	3		
requests2	5 days ago		49	0	#173	
85-add-backend-code-to-get-properties-for-a-company	5 days ago		44	0		

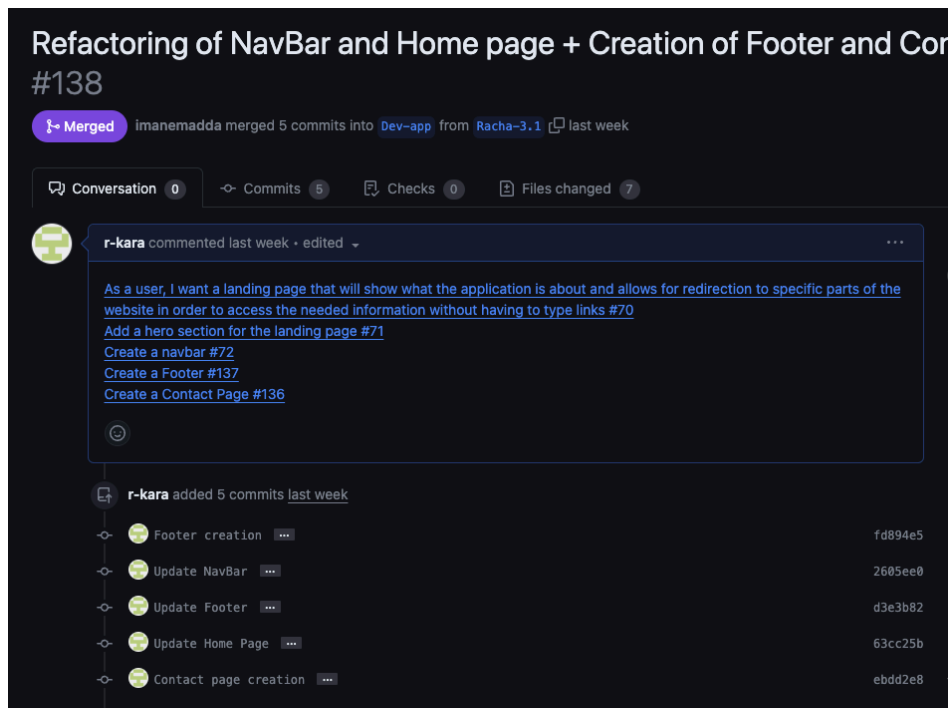
8. Atomic commits

Atomic commits are a widely used good practice among developers. Their main purpose is to ensure that changes are isolated from each other and ultimately reversible if needed. They also make the reviewing process easier since the changes are divided in logical blocks. Overall, a clear history of commits allows for better version control and task management. Although this practice may not seem useful at times, its necessity is highlighted when a bug is detected after merging commits, meaning that it can only be uncovered by reviewing recent changes.

Guideline followed:

- Address one specific task at the time
- Commit all changes made that are related to this task
- If needed, break down a large feature into smaller tasks (should already be done through the issues backlog)
- Avoid unrelated changes as much as possible
- Specify your changes in the commit message

Figure 13. Example of atomic commits that targeted separate tasks



9. Bug reporting

Although formal bug reporting is crucial to the documentation of a project, we initially followed an informal approach by reporting any bugs directly to the developer that worked on the feature, either through text message or in-person. However, we understood the importance of keeping a trace since a bug may emerge again and having a link to the commit that solved it can make us gain a significant amount of time. Therefore, we started reporting bugs, when discovered through testing, through issues directly on GitHub, allowing everyone on the team to be updated at once.

Guideline followed:

- Use a short, descriptive title.
- Add the “bug” tag to the issue.
- Specify the severity and priority.
- Describe the steps needed to reach the bug.
- If possible, add screenshots to the commit message.
- Assign the developer that worked on this feature.

Figure 14. Example of a bug report through a GitHub issue

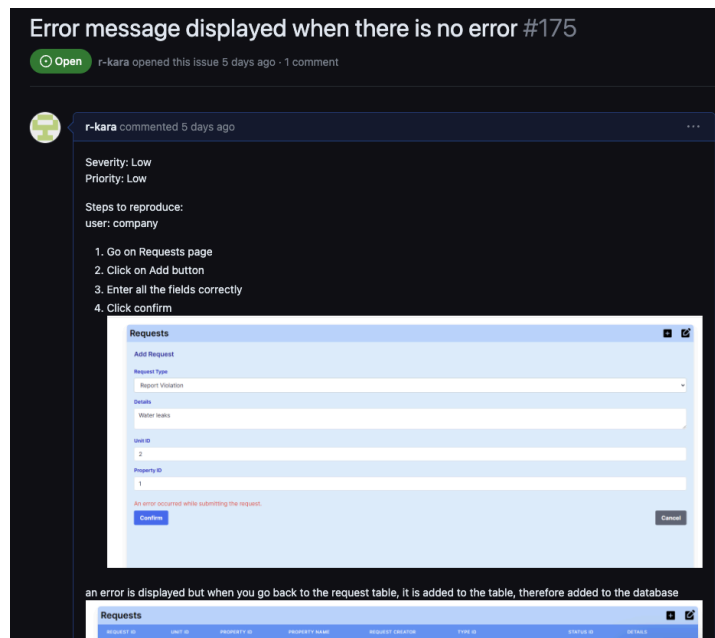


Figure 15. The use of the ‘bug’ label for issues pertaining application bugs on GitHub



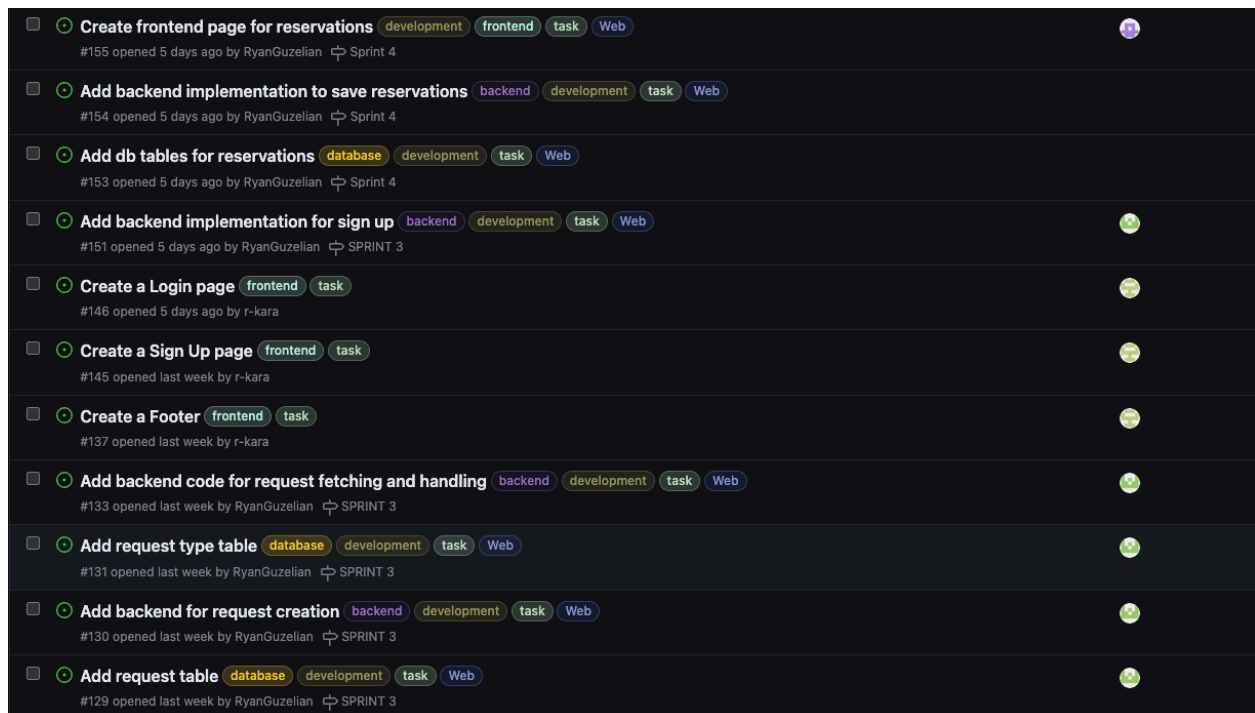
10. Use of issue labels for tracking and filtering

Issue labels are used to identify and organize the HitHub issue tracking system in order to increase understandability and efficient tracking. We defined the different labels at the beginning of the first sprint and consistently used them when defining user stories, tasks and bugs.

Types of Labels:

- Mobile, Web
- Backend, Frontend, Database, Testing
- User Story, Task, Tasks TBA, Bug
- Development, Documentation
- Extra (rarely used): High Priority, Enhancement

Figure 16. Small portion of the issues backlog



11. Links between commits and bug reports/features

Linking each commit to their corresponding bug reports or features may be one of the most essential practices to follow when contributing to a large project. First, it allows the developer to focus on a single issue at the time and make him accountable for this part of the project. It also enables traceability, allowing other developers to know what this commit is solving or contributing to when reviewing it. This way, collaboration within a team is more efficient and progress can be tracked accurately.

Guideline followed:

- Add a significant title.
- Include a link to all related issues in the commit message (user stories, tasks or bugs).
- Include the issue number.
- If a commit solves a bug or closes an issue, specify it.
- If possible, assign a reviewer that worked on these issues.

Figure 17. Example of links between commits and issues

