# **Design Patterns**

## Singleton Pattern

We have implemented a theme management strategy that aligns with the Singleton design pattern in our React application. We use React's createContext API to create a Theme Context that centralizes our theme's state and behavior, making it easily accessible to all components in our application. This centralized theme state ensures a consistent visual experience for our users and always maintains a single active theme.

Following the Singleton pattern, our Theme Context design ensures that any updates to the theme are immediately reflected across all components, providing a cohesive user experience. Our ThemeContext.Provider includes the theme's state and a changeTheme function that allows users to toggle between the "light" and "dark" modes. This approach eliminates the complexities associated with handling different theme state instances and establishes a reliable and efficient theme switching mechanism.

This approach is particularly beneficial for complex applications like ours, where multiple components need to respond to theme changes. It simplifies the propagation and synchronization process, leading to improved developer efficiency and user satisfaction.

## Observer Pattern

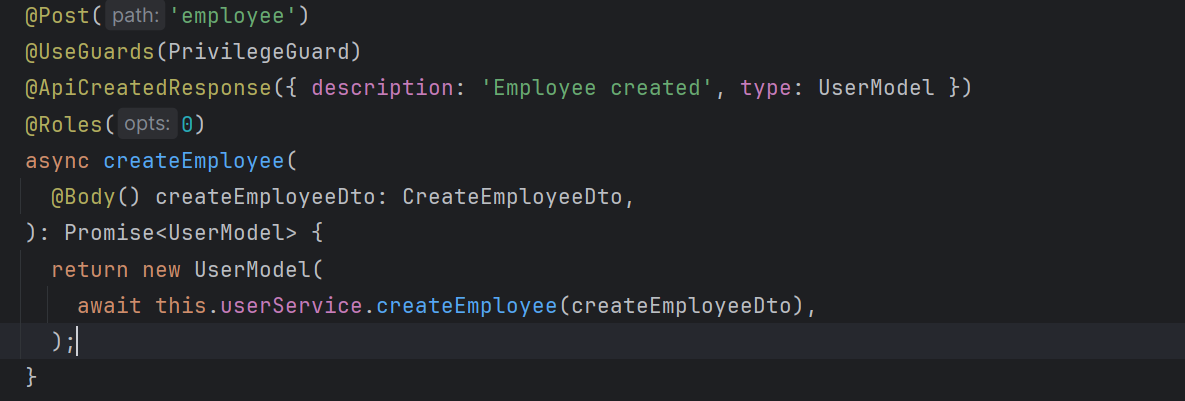
We have decided to use Redux due to its robust state management capabilities that are crucial to the smooth and consistent operation of our application. Redux uses the observer pattern, which is central to our architectural strategy, ensuring that our application's state is always consistent and coherent. We have made the Redux store the cornerstone of our design, as it acts as the single source of truth. It contains all the essential state information, including critical authentication details. This centralization simplifies state management, allowing us to maintain a clean and organized codebase.

Most of the actions that takes place within our application triggers the Redux observer pattern mechanism, which in turn updates the application state through our reducers. For instance, when a user logs in, the relevant reducers are engaged to accurately reflect the user's state change. This mechanism is crucial to preserving the structural integrity of our application and ensures that our user interface accurately mirrors the application's current state.

The flexibility that Redux offers is another reason why it is integral to our application. The ability to integrate additional reducers seamlessly enables us to introduce extra features and functionalities. This adaptability, which is a hallmark of the observer pattern, is what makes Redux a sturdy and versatile state management framework.

## Model-View-Controller (MVC)

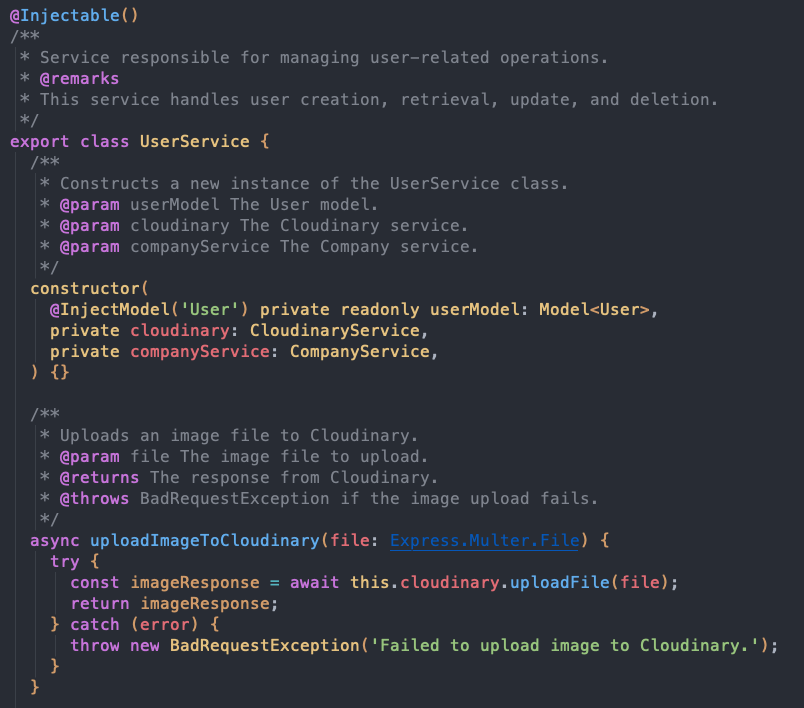
We use the Model-View-Controller (MVC) design pattern in our backend architecture to create a clear separation of concerns and a maintainable codebase. The Controller component orchestrates interactions between our application's frontend and backend layers.



In the provided example, we showcase the role of the Controller. Within our codebase, for example, in the createEmployee function, the Controller receives frontend requests, interacts with the appropriate services, and orchestrates the creation of new employee entries in the system. This demonstrates the Controller's primary responsibility of guiding calls from the frontend to the backend, where the business logic sits.

Furthermore, in our MVC architecture, the View, represented by the frontend, interacts with the Controller by making API requests. By adhering to this pattern, we establish a scalable and flexible architecture that allows for easy integration of new features and enhancements.

# Source Code Documentation

We emphasize the meticulous application of TSDoc standards that contribute to a well-documented codebase, enhancing readability and maintainability. As an example, the UserService class, as part of our application's backend services, is thoroughly annotated following TSDoc conventions. This documentation provides clear guidance on the responsibilities of the UserService, which include user creation, retrieval, update, and deletion. 

Constructor parameters are well-defined, specifying dependencies such as UserModel, CloudinaryService, and CompanyService, which are integral to the UserService's operation. Furthermore, each method within the UserService is accompanied by descriptive comments that outline the purpose, parameters, expected return types, and possible exceptions thrown during execution. For example, the `uploadImageToCloudinary` method description clarifies its role in image handling, while the `createManager` method documentation ensures that developers are aware of the method's input requirements, operational details, and error handling procedures.



The building controller is a crucial component of our backend architecture, it is properly documented to ensure clarity in its role for managing building-related operations. The class is decorated with a @Controller annotation, specifying its route at the 'building' endpoint.



Two primary operations are detailed: create and update. Both methods are accompanied by comments that describe their functionality, accepted parameters, and expected outcomes. The create method is responsible for constructing a new building entity, accepting parameters such as companyId and createBuildingDto, with an additional capability to handle file uploads, as indicated by the @UploadedFile decorator. Similarly, the update method is designed to modify existing building records, as indicated by its parameters and decorators.

By adhering to TSDoc standards, we ensure that our codebase remains accessible and intelligible not only to the current development team but also to any future contributors or stakeholders who may interact with Condofy's code.

**Bug Reports**

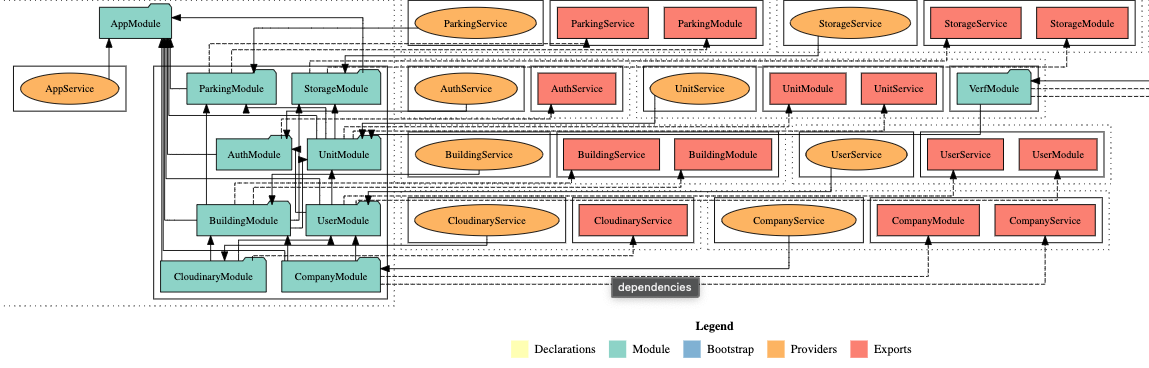
| **Bug ID** | CN-168 | | |
| --- | --- | --- | --- |
| **Originator** | Ramy Attalla | Email: ramysamerattalla@gmail.com | Signature: ramyattalla |
| **Submit Date** | 03/24/2024 | | |
| **Summary** | Backend - Cannot give Roles multiple values;  Cannot give @Roles multiple values to allow for multiple types of users to interact with the endpoint | | |
| **Severity** | Major | | |
| **Product** | Website | | |
| **Component** | API | | |
| **Platform** | PC | | |
| **OS** | Windows | | |
| **Browser** | Chrome | | |

| **Bug ID** | CN-182 | | |
| --- | --- | --- | --- |
| **Originator** | Ramy Attalla | Email: ramysamerattalla@gmail.com | Signature: ramyattalla |
| **Submit Date** | 03/30/2024 | | |
| **Summary** | Allow delete facility;  Managers should be able to delete facilities and this should cascade to related availabilities and reservations | | |
| **Severity** | Major | | |
| **Product** | Website | | |
| **Component** | Front-end | | |
| **Platform** | PC | | |
| **OS** | Windows | | |

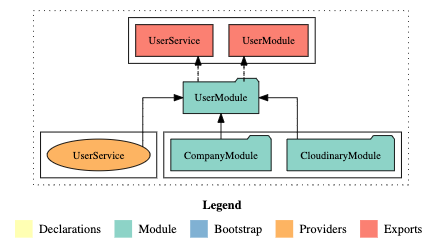
# **Design Quality**

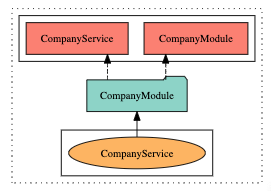
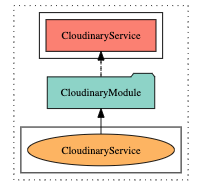
## Modularization

Our application is architected to prioritize high cohesion within modules and low coupling between them, leveraging the robust modular system provided by the NestJS framework. This design approach is evident in how we have organized our controllers and services.



**Cohesion through Functionality-Based Grouping**

By taking full advantage of NestJS's module-based architecture, we have grouped related functionalities into distinct modules. For instance, our UserModule encapsulates all the controllers and services related to user operations, while CompanyModule and CloudinaryModule manage company and media-related functionalities, respectively. This grouping ensures that each module is a high-cohesion unit, focused on a single aspect of the application's functionality.



**Low Coupling with Well-Defined Interdependencies**

The interdependencies between modules are carefully managed and minimized. As depicted in the above diagram, UserService is used within UserModule, demonstrating a direct but contained relationship. This careful structuring ensures that modules interact with each other only when necessary and through well-defined interfaces, which reduces coupling.

**Benefits of Modular Design**

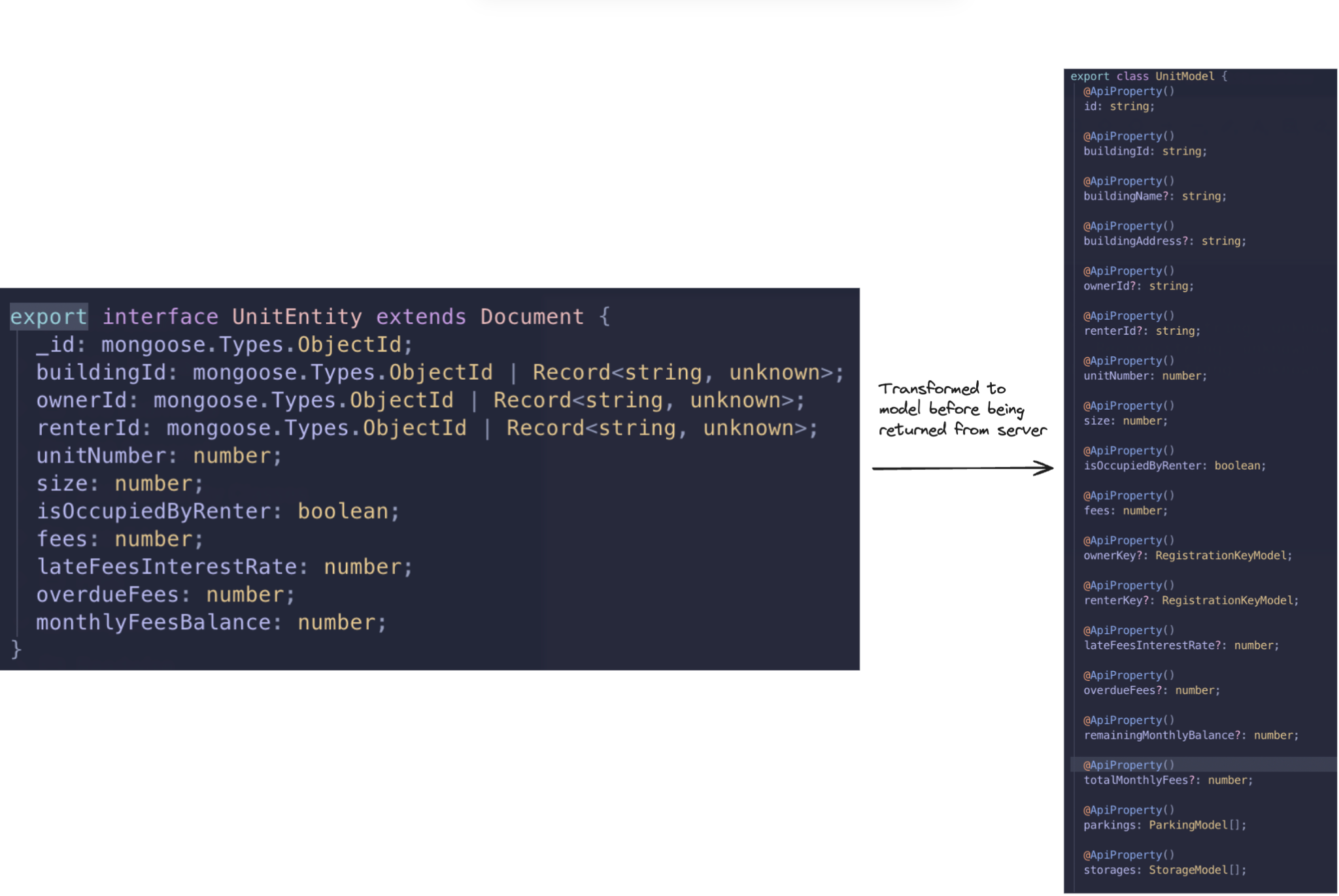
The result of this strategic modularization is a codebase where each module can be developed, tested, and debugged in isolation or in conjunction with others with minimal impact. It enhances the maintainability of the code, as changes in one module have a reduced chance of adversely affecting others. Furthermore, our approach simplifies the process of onboarding new developers, as the functionality-based grouping provides a clear roadmap of the application's structure.

**Scalability and Flexibility**

Lastly, our module-based design offers scalable and flexible solutions to future development. New features can be seamlessly integrated by adding new modules or expanding existing ones, without the need for large-scale refactorings. This strategic approach not only improves current development workflows but also sets a solid foundation for the application's long-term evolution.

## Refactoring Efforts

As part of our comprehensive backend overhaul, we made a strategic decision to separate the core data structures, known as entities, from the user-facing data representations, or models. The previous architecture had these two intertwined, resulting in difficulties in adapting the application, safeguarding data, and customizing the user interface.

In our approach, we utilized an intermediary translation layer, which is illustrated in the above figure. The “UnitEntity”, which serves as the backbone data structure, undergoes a systematic transformation into a UnitModel before being transmitted to the client. This process guarantees that any internal modifications will not affect the user interface, and vice versa.

We improved our data management by implementing design patterns and conversion tools that facilitate efficient data mapping. For instance, we changed the monthlyFeesBalance attribute in the entity to remainingMonthlyBalance in the model. These enhancements not only strengthened the security and flexibility of the app but also made future updates more straightforward. By creating a modular and forward-thinking backend structure, we established a solid foundation for the app's continued development and scalability.

# **Static Analysis & Code Review Tools**

**Importance**

The integration of a static analysis tool that runs on every pull request (PR) plays a pivotal role in maintaining and elevating the overall quality of codebases within software development projects. By systematically examining code for quality, style errors, and ranking issues, this process ensures that code adheres to established best practices, coding standards, and project-specific guidelines. The significance lies in its ability to catch potential issues early in the development lifecycle, fostering a proactive approach to code review and enhancement. This real-time feedback loop not only streamlines the code review process but also cultivates a culture of continuous improvement and collaboration among team members.

Furthermore, the ranking of issues enables developers to prioritize and tackle critical issues first, thereby optimizing resource allocation and maximizing productivity. Ultimately, the integration of static analysis into the PR workflow serves as a linchpin for ensuring codebase integrity, reliability, and maintainability, contributing to the long-term success and sustainability of software projects.

**Tool and Metrics**

For this project, the static analysis tool being used is Codacy. Codacy runs on both of our repos (Web and Mobile)

Codacy is responsible for measuring:

1. Overall code duplication, issues, and complexity

Overall code duplication refers to code snippets that have been copy pasted, instead of abstracted and reused. Issues refers to different categories of issues, code style issues (for example CSS mistakes), error prone (possible undefines that could lead to bugs), best practice (broken best practices that should be respected) and security (API secrets exposure)

A graph with a line

Description automatically generated

1. Issues: Code style issues, error prone issues, best practice issues, and security issues.

As mentioned above the analyzer detects code style issues, error prone issues, best practice issues and security issues. Moreover, if shows more details when one of the categories are selected. Codacy gives a ranking so that developers can prioritize work and avoid spending time on less critical issues.

A screenshot of a computer

Description automatically generated

1. File grades, complexity, and duplication (which effects the overall percentages)

The tool gives a grade to each file and counts issues, measures complexity and counts duplication for each file. It also does the same for each commit.

A white paper with blue lines

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Code patterns.

The tool analyzes code patterns and gives suggestions to improve the code. The issues are also ranked to ensure correct prioritization of tasks,

A screenshot of a computer

Description automatically generated

# 

# **Use of Feature Branches**

All user epics, stories, subtasks, and bugs ("tickets") are meticulously crafted within the user-friendly interface of JIRA, a comprehensive project management tool. JIRA's automatic generation of unique IDs for each ticket, such as "CN-1", serves as a fundamental component for efficient tracking and reference throughout the project's lifecycle. These IDs play a crucial role in monitoring and managing features, allowing for streamlined communication and collaboration among team members.

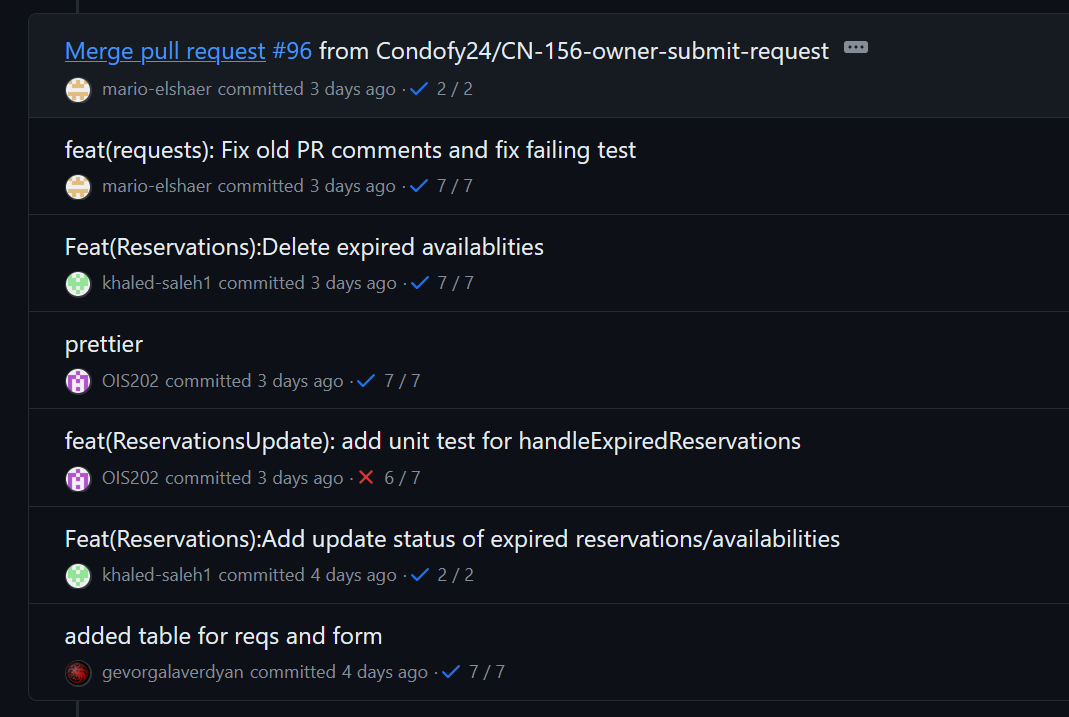
Furthermore, the implementation of a standardized branch naming convention within the version control system contributes to the systematic organization of the development process. Branch names, exemplified by formats like "CN-31-Frontend-Profile-Page", not only include the ticket number but also provide a brief description of the associated task or feature. These branches serve as representations of feature branches, facilitating easy trackability and establishing direct links to user stories in JIRA. Even bug fixes adhere to this convention, ensuring a consistent approach to monitoring the progress of bugs throughout the sprint. This integration of JIRA, ticket IDs, and branch naming conventions enhances the overall project management experience.

# 

# **Commit Messages**

Good commit messages are essential for effective collaboration in Git repositories. They provide clarity about changes made, aiding developers in understanding each commit's purpose. This clarity streamlines code reviews and facilitates navigation through the project's history. Clear messages also speed up identifying and reverting problematic changes in case of issues or bugs. Additionally, they serve as documentation, capturing important details about the development process and facilitating knowledge transfer among team members.

The example below is a screenshot of the commit history of the main branch in our repo. It shows clear commit messages following a standard convention called “Conventional commits”. Conventional commits are a set of rules that can be followed by an entire team to ensure clarity. “Feat” refers to features, “fix” refers to fixes, “refactor” refers to refactoring.



# 

# 

# **Atomic Messages**

Atomic commits refer to breaking commits down to as small as possible and only doing one at per commit. Atomic commits ensure easy code review and high code maintainability. In case of any breaking changes the main branch can be reverted to a specific commit and commits can identify where the breaking change happened.

This screenshot shows an example of atomic commits in one of the sprint 4 commits.



# **Linking of Commits to Features**

Commits are linked to features via commits that have a link to the JIRA ticket.This allows for easy trackability.

