Code Management

Table of Contents

** updates in document: green

Software Implementation of planned user stories

Phase 2	CMS-47: User Profile Creation Start date: 2024/01/29 End date: 2024/02/28 Effort: 5 CMS-49: Assignment of Registration Keys Start date:2024/02/12 End date: 2024/03/04 Effort: 2 CMS-48: Property Profile Creation Start date: 2024/02/12 End date: 2024/03/04 Effort: 4 CMS-52: Condo File Upload Start date:2024/02/12 End date: 2024/03/04 Effort: 4 CMS-53: Condo Unit Management Start date:2024/02/12 End date: 2024/03/04 Effort: 4 CMS-78: Adding Parking Spots and Lockers to a Property Profile Start date:2024/02/12 End date: 2024/03/04 Effort: 3 CMS-79: Send Registration Keys from Condo Unit to Owner/Renter Start date:2024/02/12 End date: 2024/03/04 Effort: 2 CMS-51: Condo owner dashboard

	Start date:2024/02/12 End date: 2024/03/04 Effort: 6 Total Effort: 30
Phase 3	No new features completed, only unit tests implemented
Phase 4	CMS-110: User Roles Start date:2024/03/22 End date:2024/04/12 Effort: 3 CMS-80: Manager input of fees Start date:2024/03/22 End date:2024/04/12 Effort: 5 CMS-81: Generation of Operational Budget Start date:2024/03/22 End date:2024/04/12 Effort: 5 CMS-82: Generation of Annual Report Start date:2024/03/22 End date: 2024/04/12 Effort: 4 CMS-83: Room Booking Start date:2024/03/22 End date: 2024/04/12 Effort: 6 CMS-84: Room Availability Start date:2024/03/22 End date: 2024/04/12 Effort: 5 CMS-85: Rooms first-come-first-serve Start date:2024/03/22 End date: 2024/04/12 Effort: 3 CMS-86 Assignment of Roles to Employee Users

	Start date: 2024/03/22 End date: 2024/04/12 Effort: 4 CMS-87 Request board Start date: 2024/03/22 End date: 2024/04/12 Effort: 6 Total Effort: 41
Phase 5	CMS-103 Notification Board Start date: 2024/04/12 End date: 2024/05/02 Effort: 5 Total Effort: 5

- The total project effort is 76
- Estimated team velocity is 15 points per iteration

$$Number\ of\ Iterations\ =\ \frac{Total\ Effort}{Estimated\ Team\ Velocity} = \frac{76}{15} = 5.067\ =\ 5\ Iterations$$

Bug reports/fixing

Bug ID	112		
Originator	HICHAM KITAZ	Email:hishamo-hhk@hotmail.com	Signature: hisham-kitaz
Submit Date	February 25, 2024		
Summary	Data fields go blank after saving edit		
Severity	minor		
Product	Progressive web application.		
Version	1.1		
Platform	[PC]		
OS	[Windows]		
Browser	chrome		

Bug reports fixing techniques

To streamline the defect management process for our condo management system project, we will implement a concise and effective strategy based on the structured approach outlined on the guru99.com website, tailored to our specific needs. Here's how we will manage and fix bugs throughout our project lifecycle:

Discovery: Our project team will proactively identify defects through thorough testing before they become apparent to end users. Defects recognized by developers will be accepted for further action.
Categorization: Defects will be categorized based on priority (Critical, High, Medium, Low) to help developers prioritize their tasks efficiently. This ensures critical issues that could severely impact user experience are addressed first.
Resolution: Assigned developers will fix defects based on the assigned priority. This process includes:

- Assigning the defect to a specific developer or technician.
- Scheduling the fix according to its priority.
- Fixing the defect and tracking the progress.
- Reporting the resolution back to the project management team.

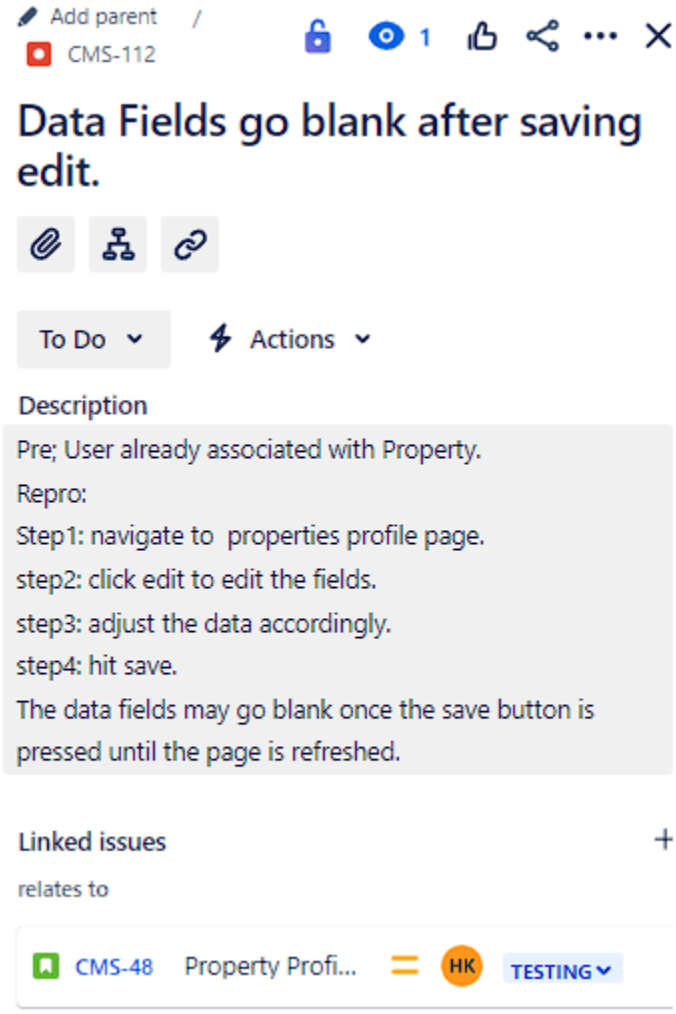
Verification: Post-resolution, our testing team will verify that defects have been properly fixed and that no new issues have arisen due to the fixes.
Closure: Once a defect has been resolved and verified, it will be marked as closed. Any unresolved issues will be sent back to the development team for re-evaluation.
Defect Reporting: We will prepare and share regular defect reports with the management team to provide updates on the defect management process and current defect status. This facilitates better communication, tracking, and management of defects.

By adhering to this structured defect management process, we aim to minimize the impact of defects on our condo management system project, ensuring a high-quality product and satisfaction for our end users.

Defect Tracking Tool for Software Testing

For managing defects and user stories in our software testing process, we have chosen to use JIRA. This decision is rooted in JIRA's comprehensive capabilities as a defect-tracking tool, which offers an intuitive and robust platform for efficiently managing and tracking bugs. JIRA's flexibility allows us to customize workflows, fields, and reporting to suit our specific project needs, making it an ideal choice for our team. Moreover, its integration with a wide range of development tools streamlines the tracking of issues from discovery through to resolution. JIRA's user-friendly interface and powerful features not only enhance our team's productivity but also ensure transparency and effective communication throughout the project lifecycle. By

adopting JIRA, we are equipped to maintain high-quality standards in our software development process, ensuring that all issues are addressed promptly and thoroughly.



Code Review Tools

When it comes to reviewing code for our project, there are several methods we employ to ensure quality and collaboration among our team members. Here are some examples:

1. **Interactive Review Sessions:** An approach we frequently use is live code reviews that allow real-time feedback and discussion. This makes it easy to address any issues or questions as they arise instead of postponing them and creating a delay in our development. Although this approach is usually conducted in person, we have adapted this method for our distributed team using tools such as Visual Studio Live Share.

2. **Pair programming Sessions**: Another approach our team has utilized is pair programming sessions where we pair up team members for code reviews. These sessions involved two members working together at a single workstation, with one writing code and the other providing immediate feedback. While this approach can be time-intensive, we've found it to be highly effective for knowledge sharing and fostering collaboration within our team.

3. **Utilizing Code Review Tools:** We have also leveraged specialized software tools to streamline the code review process. The main tool our team has been using is GitHub's integrated code review tool within its pull request feature. This tool allows us to analyze the differences in code, provide inline comments, and track the history of changes.

4. **Annotating source code before the review:** In our progress, we've implemented a practice of annotating our source code before initiating the review process. This approach has proven to be valuable in guiding us through different changes made, providing clear pointers to the files that require attention, and offering insights behind the modification. Furthermore, it prompts us to catch and address additional errors before the peer review even begins. This proactive error detection has led to a significant decrease in defect density, as we're able to solve issues earlier in the development cycle. Overall, the integration of annotations into our review process has contributed to improved code quality, smoother reviews, and a more efficient workflow within our project.

Correct use of design patterns

The need for better cross-platform compatibility, maintainability, and others played a role in the change of approach from Uno Platform to React for the front end of the Condo Management System (CMS). On the Uno platform, though, it mostly brought along maintainability challenges, not to mention smooth operation on all the targeted platforms. On the flip side, React has a strong ecosystem and boasts broad platform support as a Progressive Web App (PWA) with a very strong community. This switch necessitated a reevaluation and modification of the design patterns used in the development process.

Modified Design Patterns

As for the architecture of the front end, the CMS was rather tidy in placing its codebase. This is to standardize the way components, containers, assets, src, and tests are put, mostly to represent very much a normal composition of a React project, hence keeping it at a high level of modularity, reusability, and maintainability.

- **components**: It is the smallest UI unit of the application, constituting the rendering of the visual parts of the application. They should be designed reusable and encapsulated in a

modular manner so that even the styling and logic could be reused. Containers: Things that take care of how some other things work, and provide data and behavior to exist for presentational or other container components.

- **assets:** These are directories of files that include images, fonts, global style sheets, and others that form part of the visual and functional aspects of the CMS.
- **src:** The root source folder that will provide the structure for the project, housing all the components, containers, services, and utilities responsible for application logic.
- **tests:** Unit and integration tests for every piece of your application, to make sure they are working with their intended behavior in isolation and in combination with other pieces.

At the same time, this architectural pattern is complemented by the general stylesheet for common styles and the relevant styles to components that live inside the folder of the component. In such a way, it allows the customization of every component separately, but at the same time, the views throughout the application are the same in every separate instance of the component.

Backend Architecture

The backend involves the simplified version of the Model-View-Controller (MVC) architectural framework to pave the way for efficiency and ease of development. It leverages ASP.NET capability for building Web APIs with the following clear separation of concerns.

- **Model**: Defines the data structure and is manipulated by services.
- **View:** In this light, the "view" is simply the JSON response presented by the API consumed by the front end.
- **Controller**: It is an intermediary between frontend requests and backend responses. Every entity has an entity-specific controller that, in turn, calls the appropriate service.
- **Service:** This namespace contains business logic, and Entity Framework Core mediates interaction with the database to be able to abstract the access data layer from the controllers. This architecture reduces the number of classes in the backend structure. It suggests removing the classes such as Data Transfer Objects (DTOs) and the repository class, urging some kind of direct interaction of controllers and services with the database. Dependency injection in the back end causes many dependencies to be taken care of to keep components in a loose-coupling relationship.

Pros and Cons of the Modified Design Patterns

Pros:

- **Modularity and Reusability**: The front's structured approach facilitates code reuse and modularity, speeding up development and testing.
- **Simplified Backend Architecture**: The streamlined MVC pattern simplifies the backend, making it more accessible and maintainable.

- **Efficient Data Handling**: Direct service-to-database interactions via Entity Framework Core optimize data handling and reduce boilerplate code.

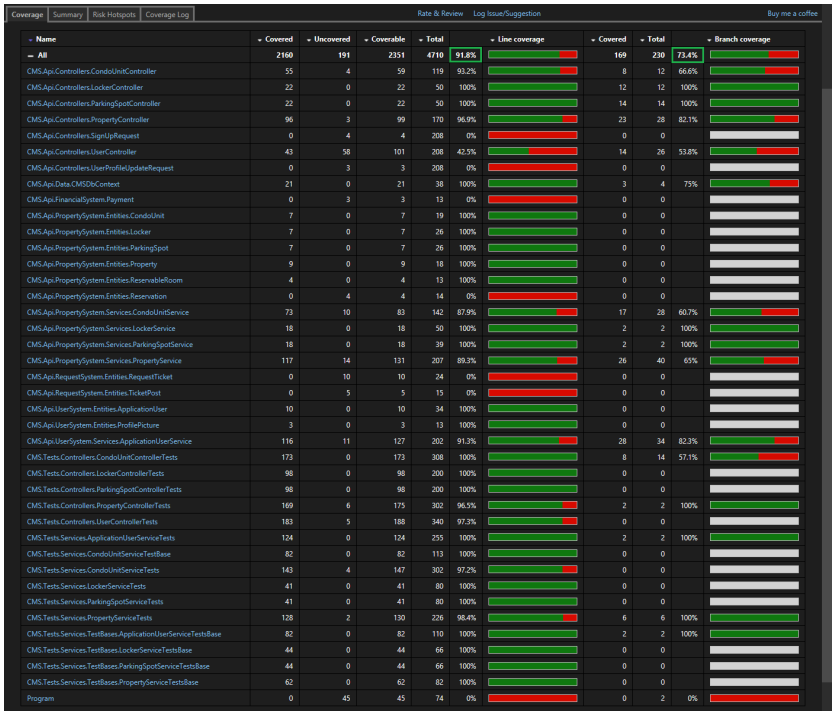
Cons:

- **Potential for Tight Coupling**: The simplified backend might lead to tighter coupling between controllers and services if not carefully managed.
- **Learning Curve**: Team members unfamiliar with React or the specific project structure may face a learning curve.
- **Overhead for Small Projects**: The detailed organization of the front end and the backend architecture might introduce unnecessary overhead for smaller projects.

Essentially, the need to balance efficiency, maintainability, and scalability was an important consideration of the adopted and modified design patterns for the Condo Management System. While simulating the back-end architecture, the project tries to leverage the power of the front-end that React offers and come out with a user-empowered, robust, friendly CMS that would fit well with the stakeholder requirements and at the same time give room for future augmentations.

Code coverage

We switched code coverage tools for sprint 3 from the *coverlet* package used previously to the *Fine Code Coverage* extension for Visual Studio. This provides the team with a better visual of which classes have/haven't been tested and to what degree. Given the shorter duration of this sprint, our focus was purely to catch up on unit testing which was lacking in previous sprints. With this done, our line coverage is now at 91.8%.



Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
All	2160	191	2351	4710	91.8%	169	230	73.4%
CMS.Api.Controllers.CondoUnitController	55	4	59	119	93.2%	8	12	66.6%
CMS.Api.Controllers.LockerController	22	0	22	50	100%	12	12	100%
CMS.Api.Controllers.ParkingSpotController	22	0	22	50	100%	14	14	100%
CMS.Api.Controllers.PropertyController	96	3	99	170	96.9%	23	28	82.1%
CMS.Api.Controllers.SignUpRequest	0	4	4	208	0%	0	0	
CMS.Api.Controllers.UserController	43	58	101	208	42.5%	14	26	53.8%
CMS.Api.Controllers.UserProfileUpdateRequest	0	3	3	208	0%	0	0	
CMS.Api.Data.CMSDbContext	21	0	21	38	100%	3	4	75%
CMS.Api.FinancialSystem.Payment	0	3	3	13	0%	0	0	
CMS.Api.PropertySystem.Entities.CondoUnit	7	0	7	19	100%	0	0	
CMS.Api.PropertySystem.Entities.Locker	7	0	7	26	100%	0	0	
CMS.Api.PropertySystem.Entities.ParkingSpot	7	0	7	26	100%	0	0	
CMS.Api.PropertySystem.Entities.Property	9	0	9	18	100%	0	0	
CMS.Api.PropertySystem.Entities.ReservableRoom	4	0	4	13	100%	0	0	
CMS.Api.PropertySystem.Entities.Reservation	0	4	4	14	0%	0	0	
CMS.Api.PropertySystem.Services.CondoUnitService	73	10	83	142	87.9%	17	28	60.7%
CMS.Api.PropertySystem.Services.LockerService	18	0	18	50	100%	2	2	100%
CMS.Api.PropertySystem.Services.ParkingSpotService	18	0	18	39	100%	2	2	100%
CMS.Api.PropertySystem.Services.PropertyService	117	14	131	207	89.3%	26	40	65%
CMS.Api.RequestSystem.Entities.RequestTicket	0	10	10	24	0%	0	0	
CMS.Api.RequestSystem.Entities.TicketPost	0	5	5	15	0%	0	0	
CMS.Api.UserSystem.Entities.ApplicationUser	10	0	10	34	100%	0	0	
CMS.Api.UserSystem.Entities.ProfilePicture	3	0	3	13	100%	0	0	
CMS.Api.UserSystem.Services.ApplicationUserService	116	11	127	202	91.3%	28	34	82.3%
CMS.Tests.Controllers.CondoUnitControllerTests	173	0	173	308	100%	8	14	57.1%
CMS.Tests.Controllers.LockerControllerTests	98	0	98	200	100%	0	0	
CMS.Tests.Controllers.ParkingSpotControllerTests	98	0	98	200	100%	0	0	
CMS.Tests.Controllers.PropertyControllerTests	169	6	175	302	96.5%	2	2	100%
CMS.Tests.Controllers.UserControllerTests	183	5	188	340	97.3%	0	0	
CMS.Tests.Services.ApplicationUserServiceTests	124	0	124	255	100%	2	2	100%
CMS.Tests.Services.CondoUnitServiceTestBase	82	0	82	113	100%	0	0	
CMS.Tests.Services.CondoUnitServiceTests	143	4	147	302	97.2%	0	0	
CMS.Tests.Services.LockerServiceTests	41	0	41	80	100%	0	0	
CMS.Tests.Services.ParkingSpotServiceTests	41	0	41	80	100%	0	0	
CMS.Tests.Services.PropertyServiceTests	128	2	130	226	98.4%	6	6	100%
CMS.Tests.Services.TestBases.ApplicationUserServiceTestsBase	82	0	82	110	100%	2	2	100%
CMS.Tests.Services.TestBases.LockerServiceTestsBase	44	0	44	66	100%	0	0	
CMS.Tests.Services.TestBases.ParkingSpotServiceTestsBase	44	0	44	66	100%	0	0	
CMS.Tests.Services.TestBases.PropertyServiceTestsBase	62	0	62	82	100%	0	0	
Program	0	45	45	74	0%	0	2	0%

Respect to code conventions

With the majority of the code being in C#, we do our best to adhere to Microsoft's C# Code Conventions and naming conventions. For example, all variables in a defined entity are named using PascalCasing:

```
public class Property
{
    [Key]
    1 reference
    public Guid Id { get; set; }
    17 references
    public string PropertyName { get; set; } = String.Empty;
    14 references
    public string CompanyName { get; set; } = String.Empty;
    14 references
    public string Address { get; set; } = String.Empty;
    14 references
    public string City { get; set; } = String.Empty;
    14 references
    public ICollection<CondoUnit>? CondoUnits { get; set; }
    18 references
    public ICollection<ParkingSpot>? ParkingSpots { get; set; }
    18 references
    public ICollection<Locker>? Lockers { get; set; }
    8 references
    public ICollection<ReservableRoom>? ReservableRooms { get; set; }
}
```

In comparison, instances that are using dependency injection begin with an underscore:

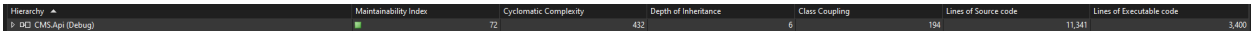
```
private readonly CMSDbContext _context;
private readonly UserManager<ApplicationUser> _userManager;
private readonly PasswordHasher<ApplicationUser> _passwordHasher;
private readonly IPropertyService _propertyService;
```

While regular instances are in camelCase:

```
var emailValidation = "^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$";
var match = Regex.Match(email, emailValidation, RegexOptions.IgnoreCase);
return match.Success;
```

Design quality (number of classes/packages, size, coupling, cohesion)

For assessing and enhancing the design quality of our software—specifically focusing on the number of classes/packages, size, coupling, and cohesion—we opted for Understand by SciTools. Understand is a comprehensive and customizable integrated development environment tailored for static code analysis, equipped with a vast array of visualizations, documentation, and metric tools. Its selection was motivated by the need for an in-depth understanding of our codebase, enabling us to identify and address design flaws systematically. The tool's capability to generate detailed metrics and visual representations of code structure helps us monitor and improve aspects such as class/package organization, code size, and the relationships between different components, aiming for a well-structured, maintainable, and efficiently decoupled architecture. By leveraging Understand, we can pinpoint areas that require refactoring or optimization, ensuring our code adheres to best practices in software design for enhanced cohesion and reduced coupling, ultimately leading to a more robust and scalable software product.



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
CMS.Api (Debug)	72	432	6	194	11,341	3,400

Quality of source code documentation

Importance of Quality Source Code Documentation

- Improving readability: Should the need to refactor code come up, analyzing and changing commented code will be far easier for any of us than if it did not have comments. It is easier to navigate through the program when clear comments give context, explanations, and insights into the purpose of individual components.

- Development acceleration: This happens when there is thorough documentation because programmers can give more effort to writing and improving code rather than interpreting it. Achieving project deadlines and producing high-quality software, results in greater productivity and faster iteration cycles.

- Collaboration: By making sure that all team members have an equal understanding of the application, thorough documentation will increase the cooperation between members of the team. By this we can achieve better collaboration and communication, which will improve the effectiveness of development procedures.

Practices we used for Source Code Documentation on GitHub

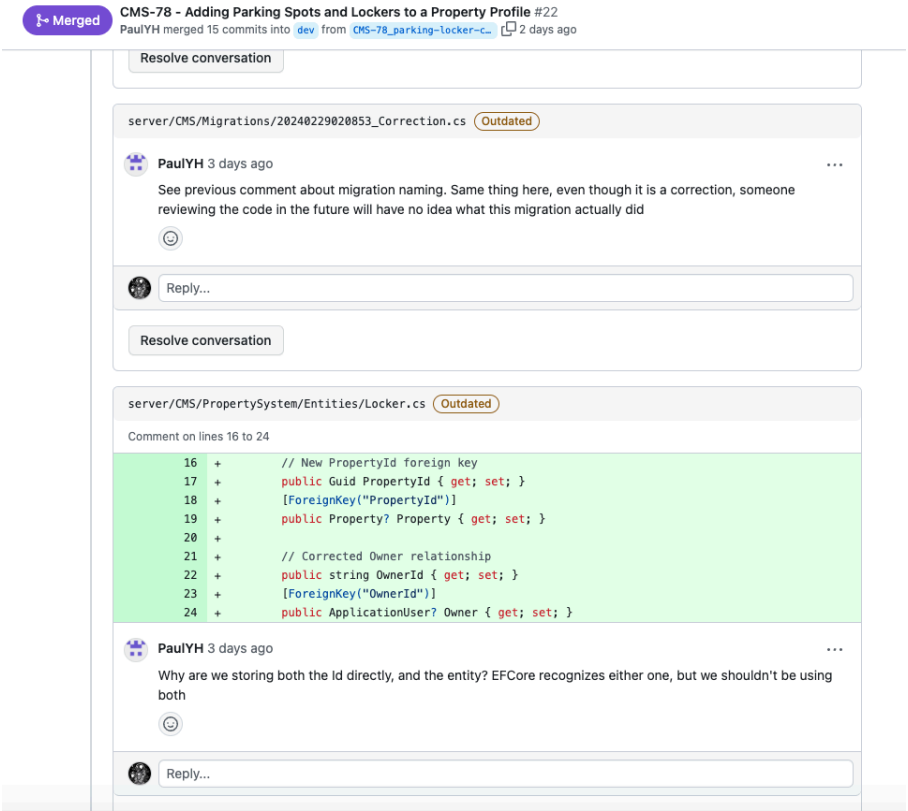
1. Use of descriptive comments: We wrote comments that describe each code function and

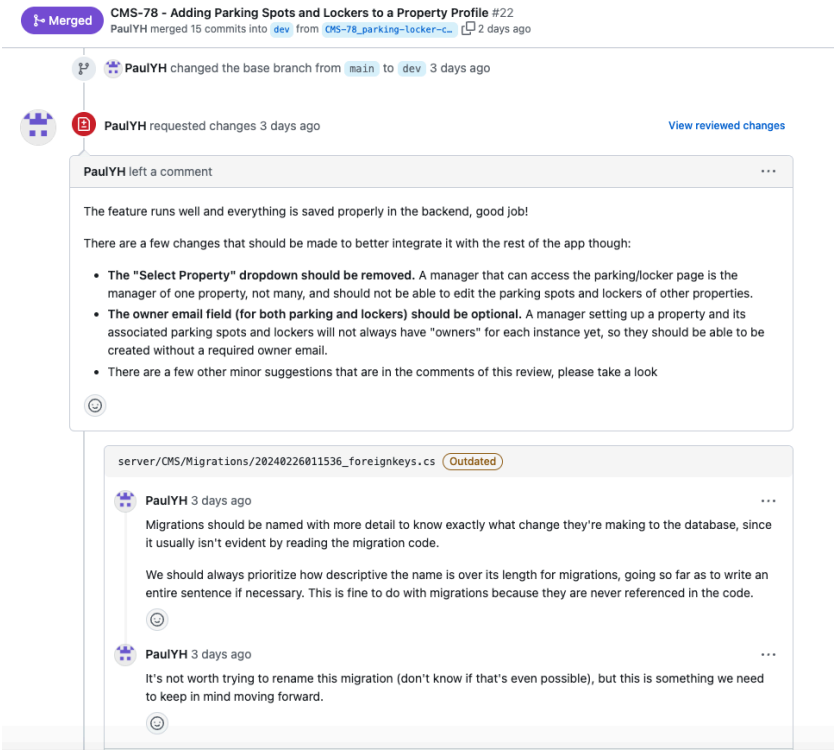
what needs to be improved, or if everything was good with the code. We Stayed free of ambiguity and unnecessary technical terms by using language that is straightforward and brief. When writing comments, we kept the viewpoint of someone who isn't familiar with the codebase in mind.

2. Using Pull Request reviews: To perform thorough code reviews that take into account the documentation's quality, we made use of GitHub's pull request (PR) capability. Encourage team members to evaluate each other's documentation and code, offering helpful feedback and suggestions for enhancement.

3. Consistent commenting: We made sure that all comments were written in a way that they are easy to read and can be understood by all team members. This is done through maintaining consistent commenting throughout the codes.

Examples on next page





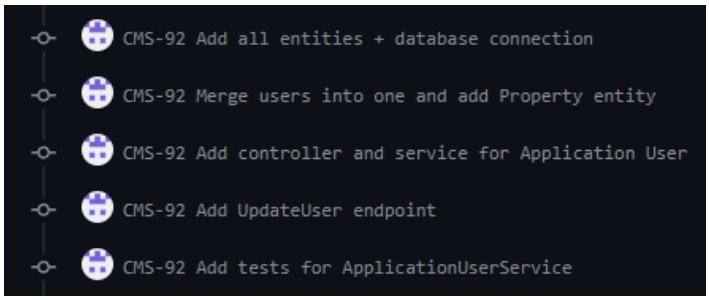
Refactoring activity documented in commit messages

In general, we agreed to use atomic commits, which means we make a commit for every substantial change. That means our commits have a fairly small and defined scope and purpose. For example, when adding a service class for one of our entities, we would make a commit for each method that we would add, unless the class is fairly small in which case we would permit ourselves to add the entire class in a single commit. In terms of refactoring, whenever we would change a class's name or purpose, there would be a detailed explanation in our commit message that explains what the class used to do and what it does now.

CMS-119 Refactor front-end calls to use ReactQuery	Changed basic fetch API calls to more robust and less error-prone calls using React Query npm package
CMS-120 Refactor front-end elements to use NextUI components	Changed HTML elements, primarily buttons, to use the NextUI component library

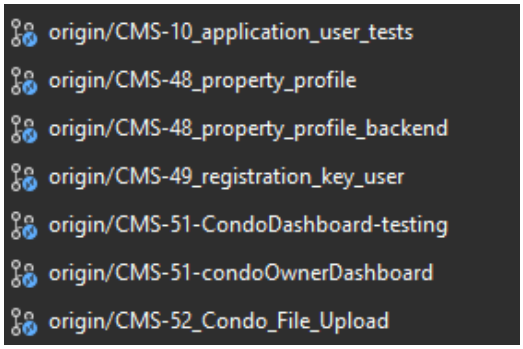
Quality/detail of commit messages

We have done our best to write clear and concise commit messages to convey exactly what is added/modified in a given commit. At the head of each commit message, we include the issue key (CMS-#) which allows Jira to link the commit to the appropriate issue. We also, the majority of the time, write the commits in the present tense as is the convention.



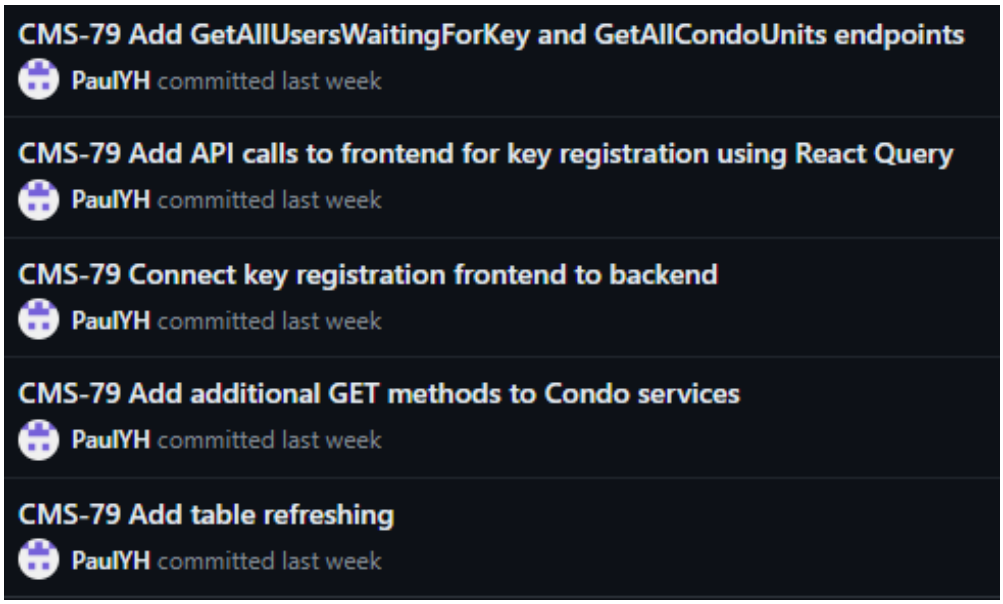
Use of feature branches

We are extensively using feature branches, ensuring that they are named properly to reflect the user story that each aims to implement. At the head of the branch name is the issue key (CMS-#) which allows Jira to link the branch to the appropriate issue.

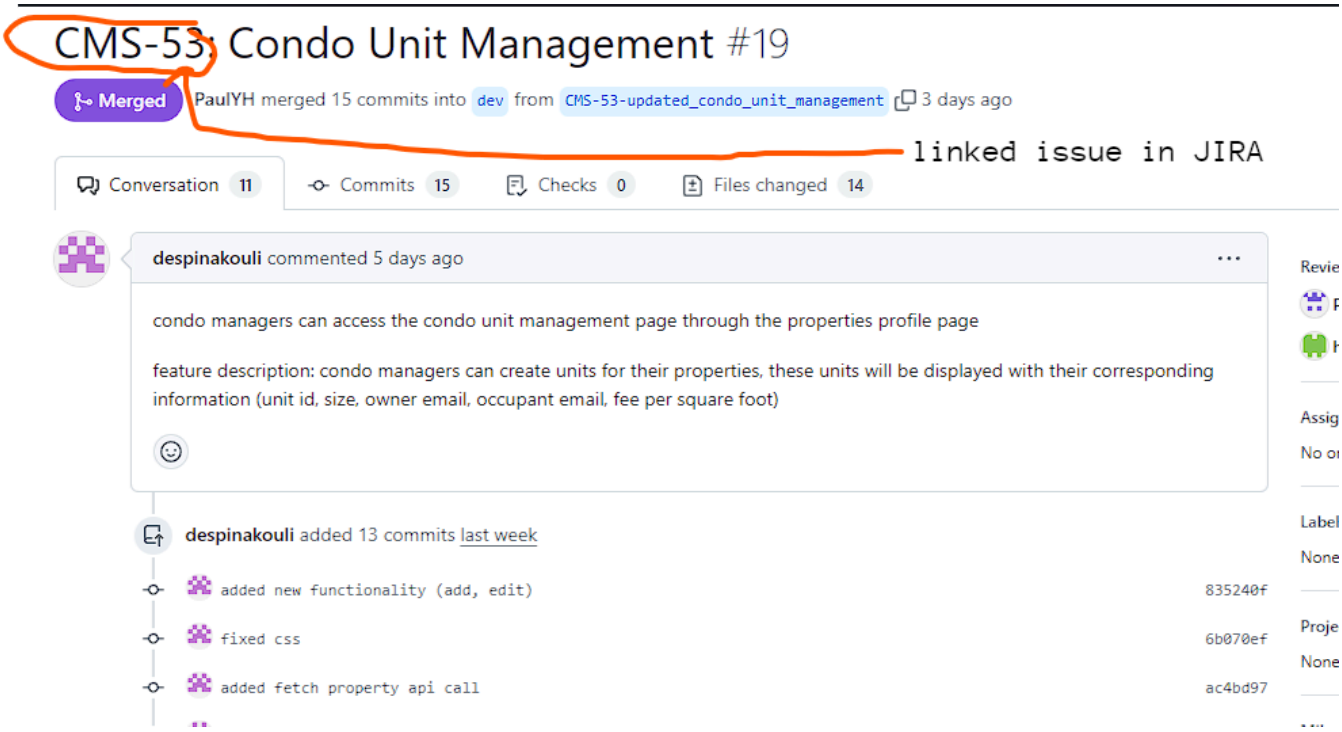


Atomic commits

During Sprint 2 of our project, we adopted the atomic commit methodology to enhance our version control practices. We continued this in Sprint 3. By embracing this approach, we ensure that our commits are granular, focusing on making small, self-contained changes. This means that for every two or three functions added or modified, we proceed with a commit. This strategy not only makes our commit history more intuitive and manageable but also simplifies the process of identifying and reverting specific changes if necessary. Implementing atomic commits has significantly improved the quality and clarity of our codebase, making it easier for the team to collaborate and track progress efficiently.



Linking of commits to bug reports/features



We have linked our GitHub repositories to JIRA, User stories are tracked through CMS-# where # is replaced with the number of the user story.

Coding guides

- React.JS Guide: https://react.dev/blog/2023/03/16/introducing-react-dev
- React.JS Documentation: https://react.dev/reference/react
- Node.JS Documentation: https://nodejs.org/docs/latest/api/

- ASP.NET Guide: https://learn.microsoft.com/en-us/aspnet/tutorials
- ASP.NET Documentation: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0
- Entity Framework Documentation: https://learn.microsoft.com/en-us/ef/
- Microsoft SQL Server Documentation: https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16