Analyze and Explore State-of-the-art Legacy System

Legacy Kanban Project Management System Upgrade

Group Members:

1) Amna Mubarak	20k-1695
2) Ehtesham Zafar	20k-1655
3) Hassan Ali	20k-1052

Submitted To: Dr. Syed Moazzam Ali Shah

Submission Date: 5 February 2024

original repo: https://github.com/sldiaz04us/kanban-project-management

our repo: https://github.com/Syed007Hassan/kanban-project-management

Table of Contents

Description:

The Legacy Kanban Project Management System is a web application designed for managing software development projects using the Kanban methodology. However, the current system lacks several crucial features and optimizations that are essential for modern software development practices. The goal of this project is to upgrade the legacy system to meet contemporary standards and address the shortcomings identified in the original implementation.

Stated Technologies in the Project:

1) Frontend:

- Angular: Framework for building single-page applications.
- TypeScript: Superset of JavaScript.
- angular/cdk/drag-drop: Provides drag-and-drop interfaces.

2) Backend:

- NestJS: Node.js framework for building server-side applications.
- MongoDB: NoSQL database..

Stakeholders Involved in the Project:

- **Developers:** Responsible for implementing the upgrades and enhancements.
- **Project Managers:** Oversee the project's progress and ensure alignment with business objectives.
- **Users:** Individuals who utilize the Kanban Project Management System for managing software development projects.
- **Quality Assurance Team:** Responsible for testing the upgraded system to ensure functionality and reliability.
- **IT Operations Team**: Responsible for deployment and maintenance of the system in production environments.**4. Scope:**

The scope of the project includes:

- Identifying and addressing limitations and deficiencies in the current system.

- Implementing an Authentication and Authorization System to enhance security.
- Enhancing accessibility features to improve usability for users.
- Writing unit and integration tests to ensure robustness and reliability.
- Upgrading dependencies and libraries to their latest versions for compatibility and performance improvements.
- Implementing additional features and optimizations based on user feedback and industry best practices.

Features of the Project:

1. CRUD Operations:

- Create, read, update, and delete projects.
- Create, read, update, and delete issues within projects.
- Create, read, update, and delete comments for each issue.

2. Project Management:

- Filter and sort projects.
- Assign users to projects.

3. Issue Management:

- Filter issues.
- Assign issues to users.
- Drag and drop issues onto the Kanban board.

4. Authentication and Authorization:

- Implement user authentication and authorization mechanisms.
- Allow users to register and participate in project management activities.

5. Testing and Quality Assurance:

- Write unit and integration tests to ensure software quality.
- Perform thorough testing to identify and fix any issues.

6. Accessibility Improvements:

- Enhance accessibility features to ensure compliance with accessibility standards.
- Improve usability for users with disabilities.

Functional Requirements:

1. User Authentication and Authorization:

- ➢ Users should be able to register, login, and logout.
- ➢ Each user should have appropriate access rights based on their role within a project (e.g., admin, project manager, team member).

2. Project Management:

- ➢ Users should be able to create, view, update, and delete projects.
- ➢ Projects should have a title, description, and list of associated team members.
- ➢ Projects should be sortable and filterable based on various attributes (e.g., status, creation date).

3. Issue Management:

- ➢ Users should be able to create, view, update, and delete issues within a project.
- ➢ Each issue should have a title, description, status, assignee, and due date.
- ➢ Issues should be sortable and filterable based on various attributes (e.g., priority, assignee, status).
- ➢ Issues should be draggable and droppable within the Kanban board for easy workflow management.

4. Comment Management:

- ➢ Users should be able to add, view, update, and delete comments on each issue.
- ➢ Comments should display the author's name, timestamp, and content.

5. Search Functionality:

- ➢ Users should be able to search for projects, issues, or comments using keywords or specific criteria.

6. Accessibility:

- ➢ The application should adhere to accessibility standards, ensuring that all users, including those with disabilities, can navigate and interact with the interface effectively.

User Stories:

1. User Authentication:

- ➢ As a user, I want to be able to register for an account so that I can access the Kanban Project Management system.
- ➢ As a user, I want to be able to log in and out of my account securely.

2. Project Management:

- ➢ As a project manager, I want to create new projects and add team members to them.
- ➢ As a team member, I want to view a list of projects I'm involved in and their details.
- ➢ As a project manager, I want to update project details such as title, description, and team members.
- ➢ As a user, I want to be able to delete a project if it's no longer needed.

3. Issue Management:

- ➢ As a team member, I want to create new issues within a project.
- ➢ As a team member, I want to view a list of issues within a project and their details.

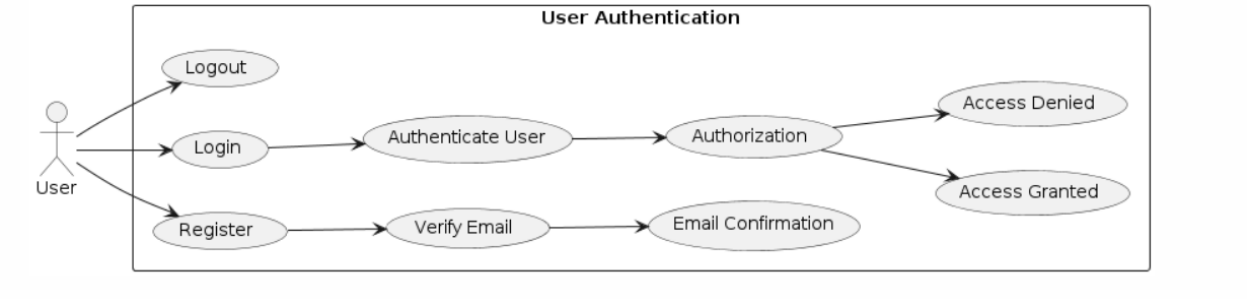
- ➢ As a project manager, I want to assign issues to specific team members.
- ➢ As a team member, I want to update the status and details of an issue.
- ➢ As a user, I want to be able to delete an issue if it's no longer relevant.

4. Comment Management:

- ➢ As a team member, I want to add comments to issues to provide updates or additional information.
- ➢ As a user, I want to view a list of comments on an issue and their details.
- ➢ As a team member, I want to edit or delete comments I've made if necessary.

Use Case Diagrams:

1. User Authentication:

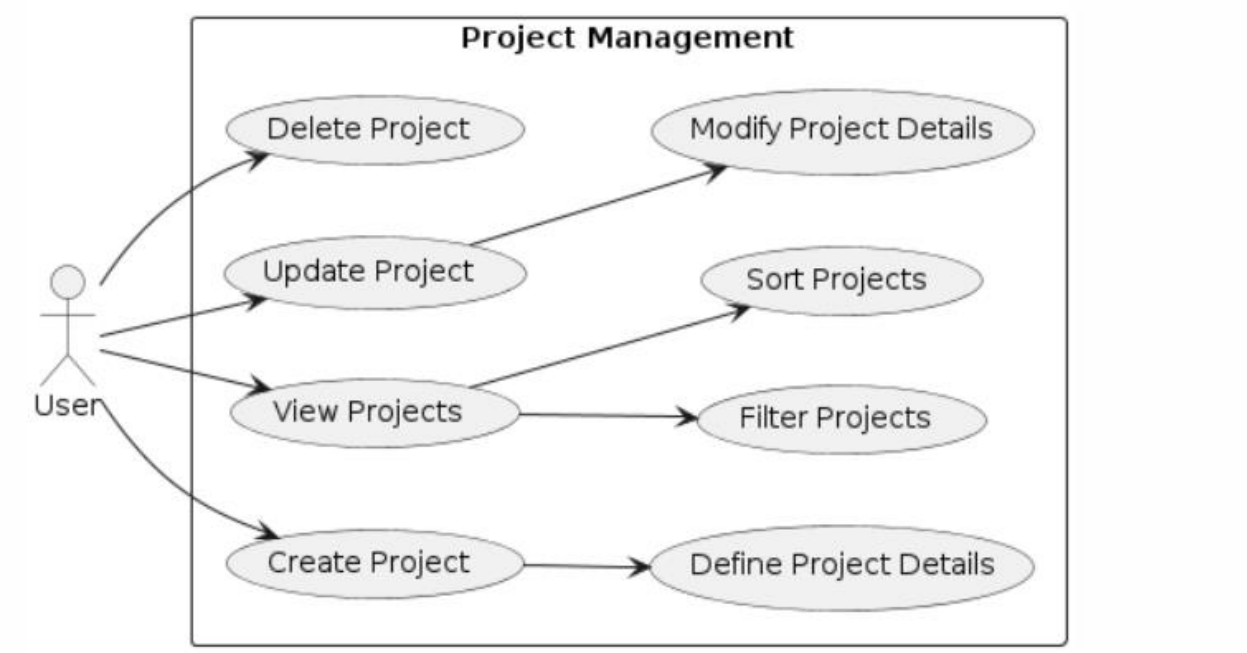


Descriptive UseCase:

Use Case: User Authentication	**Actors: User**
Description:	The User Authentication use case describes the process by which a user can register, log in, and log out of the system securely. It also includes steps for email verification and authorization checks to ensure secure access to system resources.

Preconditions:	1. The user must have internet access. 2. The system must be operational.
Basic Flow:	**1. Register:** User navigates to registration page and fills out required information. System sends a verification email to the provided email address. **2. Verify Email:** User checks email inbox for verification email. User clicks the verification link provided in the email. System verifies the email address and marks it as confirmed. **3. Login:** User navigates to the login page and enters credentials. System validates the user's credentials. 4**. Authenticate User:** Upon successful validation, the system authenticates the user's identity. **5. Authorization:** System checks user's role and permissions. If authorized, access is granted. If not authorized, access is denied. **6. Access Granted:** User gains access to the system and performs authorized actions. **7. Access Denied:** If role or permissions do not allow access, the system denies access and notifies the user.
Alternative Flows:	1. User requests resend of verification email if not received. 2. User retries login or resets password if incorrect credentials entered.
Postconditions:	1. Upon successful authentication and authorization, the user gains access to the system. 2. Users can perform authorized actions within the system. 3. Upon logout, the user's session is terminated and redirected to the login page.

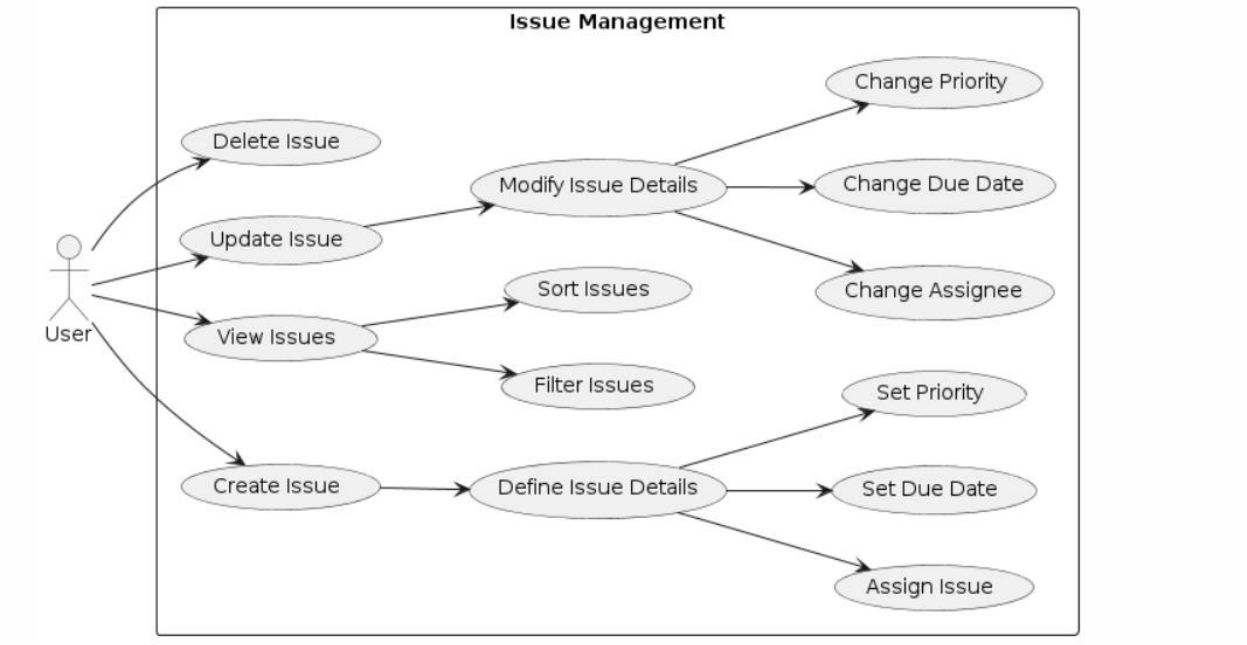
2. Project Management:



Use Case: Project Management	**Actors: User**
Description:	The Project Management use case describes the process by which users can create, view, update, and delete projects within the system. It includes functionalities such as defining project details, filtering and sorting projects, and modifying project information.
Preconditions:	1. The user must be logged in to the system. 2. The system must be operational.

Basic Flow:	**1. Create Project:** User navigates to the create project page. User defines the details of the new project, including title, description, and associated team members. **2. View Projects:** User navigates to the projects page. User views a list of existing projects. Users can filter projects based on specific criteria (e.g., status, creation date). Users can sort projects based on various attributes (e.g., title, last updated). **3. Update Project:** User selects a project to update. User modifies project details such as title, description, or team members. **4. Delete Project:** User selects a project to delete. System prompts the user for confirmation. User confirms deletion, and the project is removed from the system.
Alternative Flows:	If a user attempts to create, view, update, or delete a project without logging in, the system directs them to the login page. If a user attempts to update or delete a project that they do not have permission to access, the system denies the request and notifies the user.
Postconditions:	1. Upon successful creation, updating, or deletion of a project, the system reflects the changes in the projects list. 2. Users can perform actions related to project management seamlessly within the system.

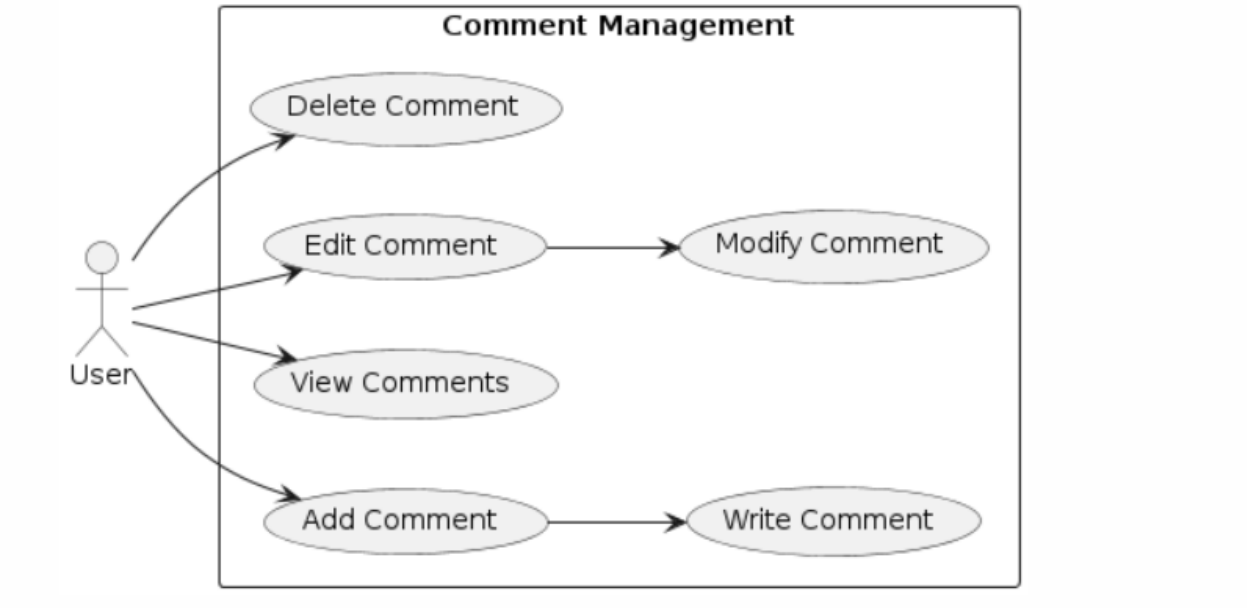
3. Issue Management:



Use Case: Issue Management	**Actors: User**
Description:	The Issue Management use case describes how users can create, view, update, and delete issues from the system. It supports features such as defining issue details, filtering and sorting issues, and modifying issue information.
Preconditions:	1. The user must be logged in to the system. 2. The system must be operational.

Basic Flow:	**1. Create Issue:** User navigates to the create issue page. User defines the details of the new issue, including title, description, assignee, due date, and priority. User assigns the issue to a team member. User sets the due date for resolving the issue. User sets the priority level of the issue. **2. View Issues:** User navigates to the issues page. User views a list of existing issues. Users can filter issues based on specific criteria (e.g., status, assignee, priority). Users can sort issues based on various attributes (e.g., due date, priority). **3. Update Issue:** User selects an issue to update. User modifies issue details such as title, description, assignee, due date, or priority. Users can change the assignee of the issue. Users can change the due date for resolving the issue. Users can adjust the priority level of the issue. **4. Delete Issue:** User selects an issue to delete. System prompts the user for confirmation. User confirms deletion, and the issue is removed from the system.
Alternative Flows:	If a user attempts to create, view, update, or delete an issue without first logging in, the system will redirect them to the login page. If a user attempts to update or delete an issue for which they do not have permission, the system denies the request and notifies the user.
Postconditions:	1. Upon successful creation, updating, or deletion of an issue, the system reflects the changes in the issues list. 2. Users can perform actions related to issue management seamlessly within the system.

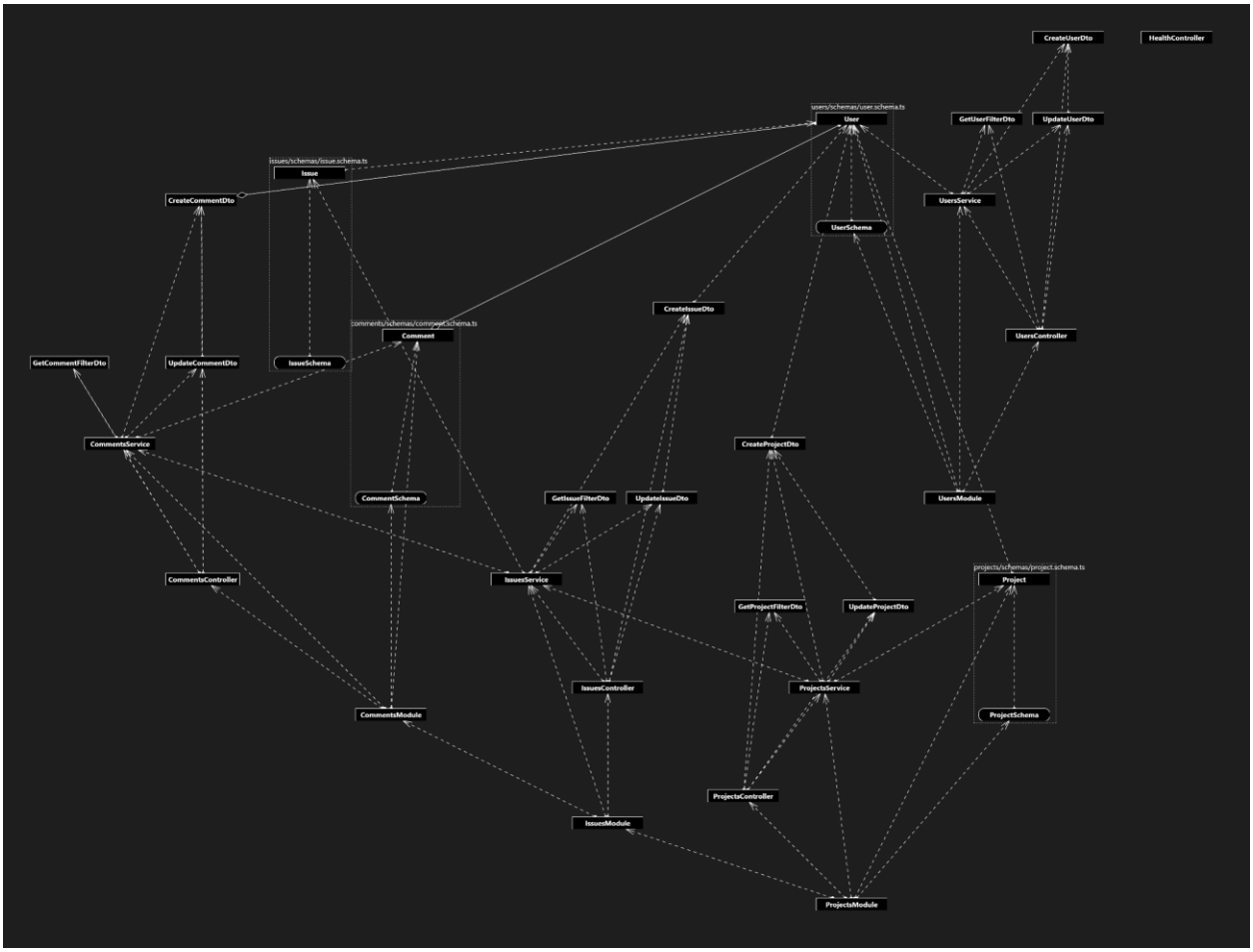
4. Comment Management:



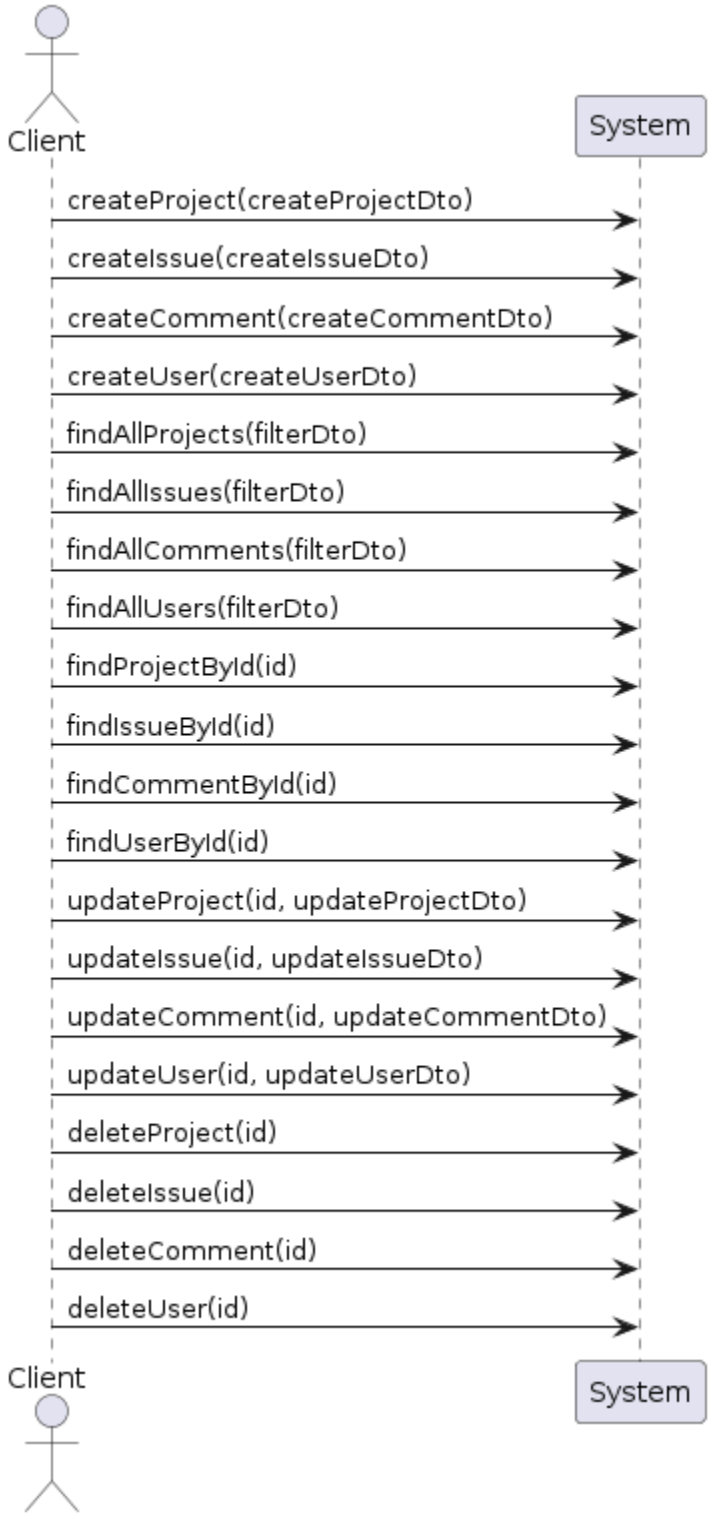
Use Case: Comment Management	**Actors: User**
Description:	The Comment Management use case describes how users can add, view, edit, and delete comments on issues in the system. It supports writing comments, modifying existing comments, and removing comments.
Preconditions:	1. The user must be logged in to the system. 2. The system must be operational.

Basic Flow:	**1. Add Comment:** User views an issue and navigates to the comments section. User writes a new comment and submits it. **2. View Comments:** User views a list of existing comments on an issue. Users can read the content of each comment and see the author and timestamp. **3. Edit Comment:** User selects a comment to edit. User modifies the content of the comment and saves the changes. **4. Delete Comment:** User selects a comment to delete. System prompts the user for confirmation. User confirms deletion, and the comment is removed from the system.
Alternative Flows:	If a user attempts to add, view, edit, or delete a comment without logging in, the system directs them to the login page. If a user attempts to edit or delete a comment they did not create, the system denies the request and notifies the user.
Postconditions:	1. Upon successful addition, editing, or deletion of a comment, the system reflects the changes in the comments section. 2. Users can interact with comments related to issues seamlessly within the system.

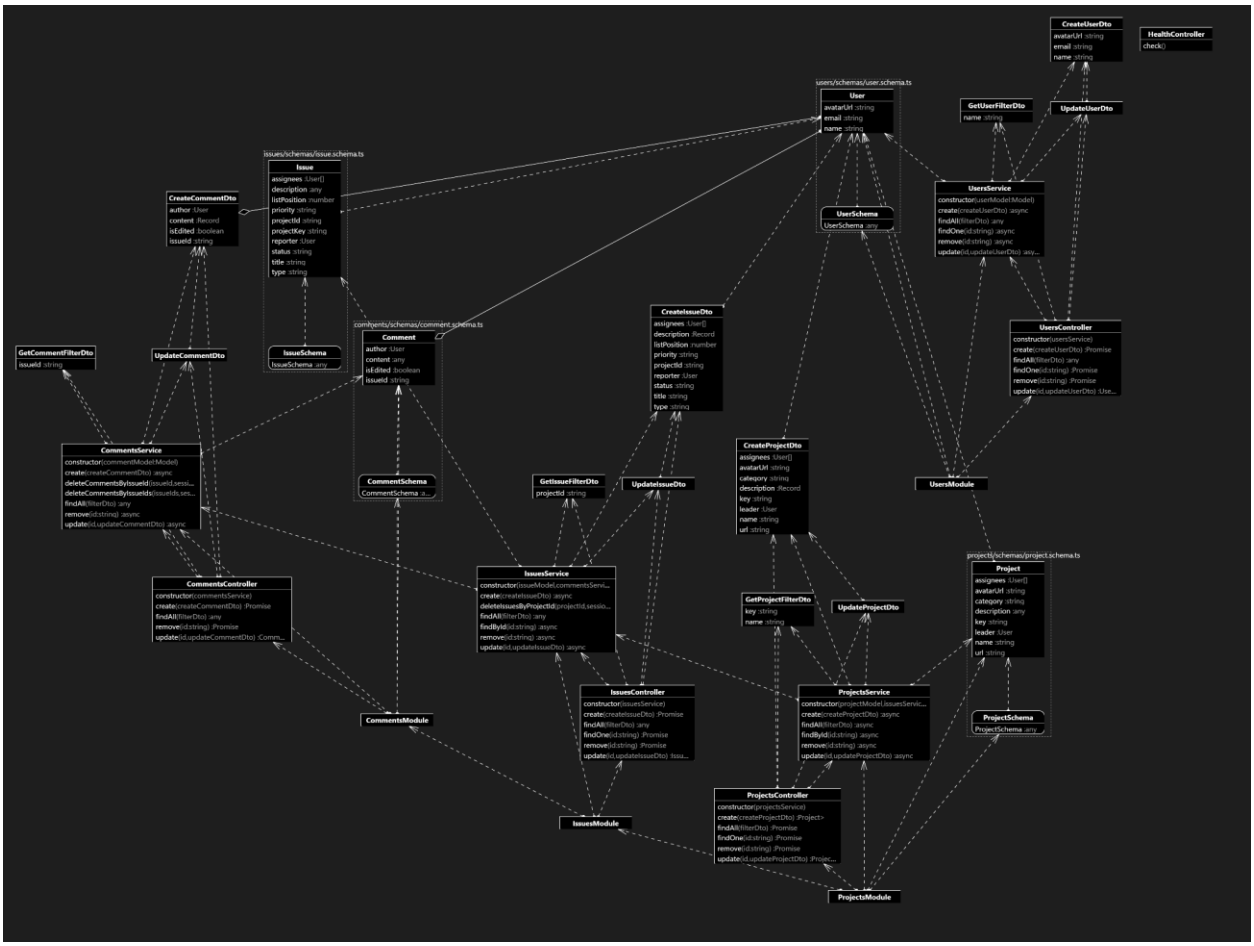
Domain Model:



System Sequence Diagram:

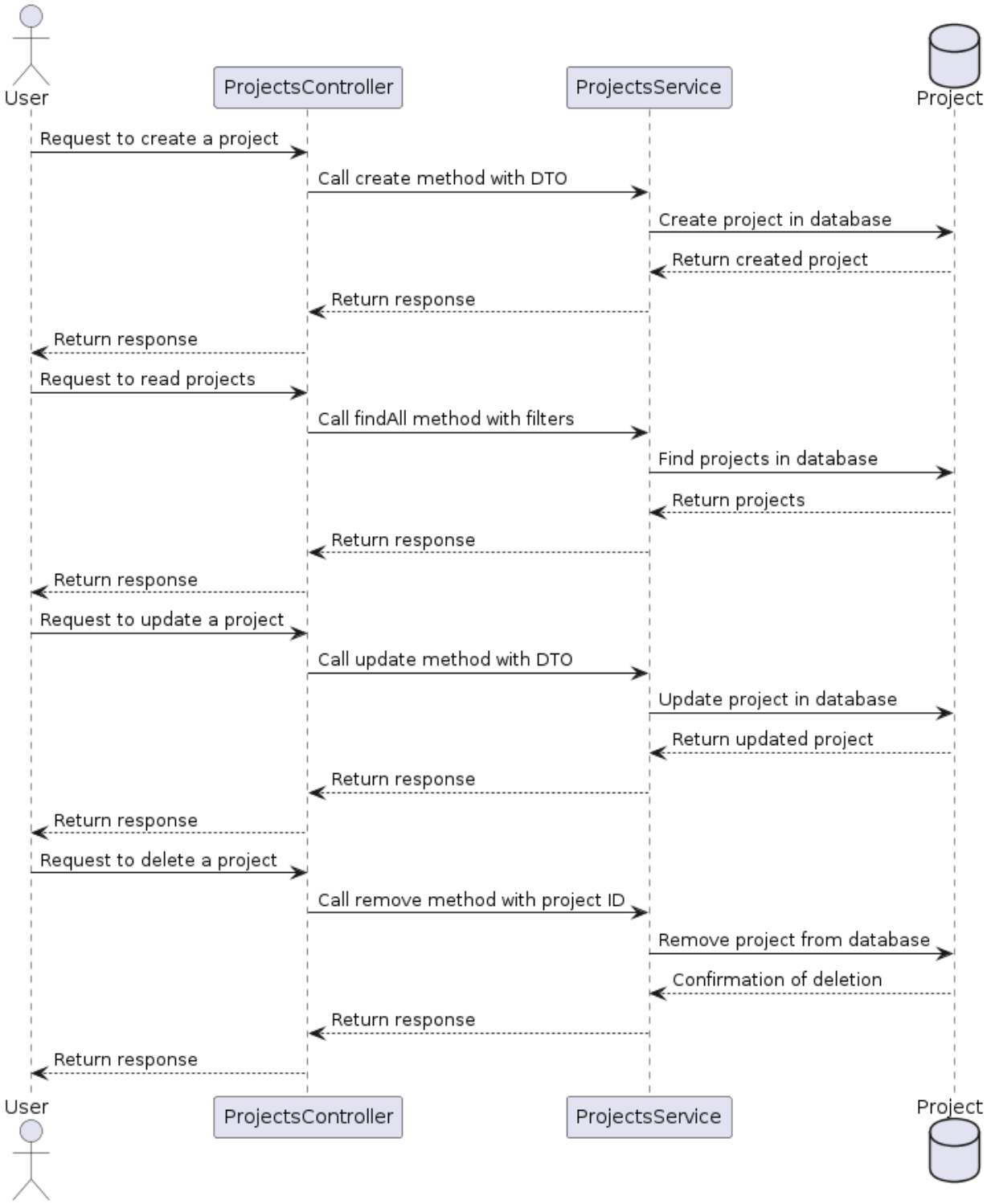


Class Diagram:

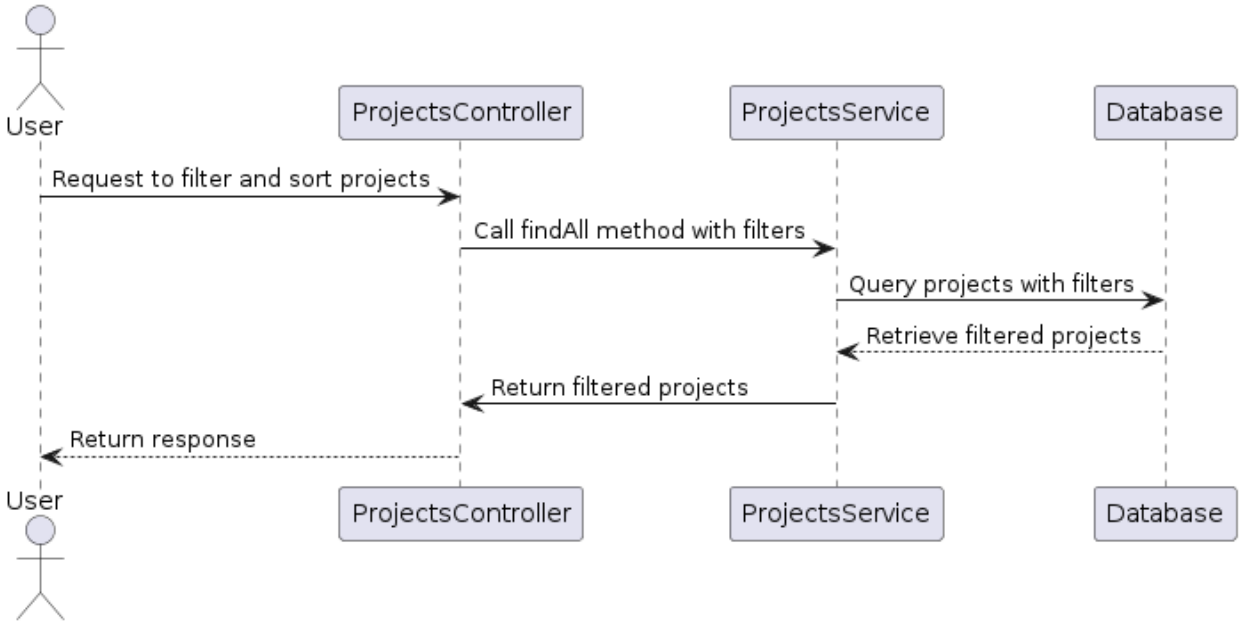


Sequence Diagram

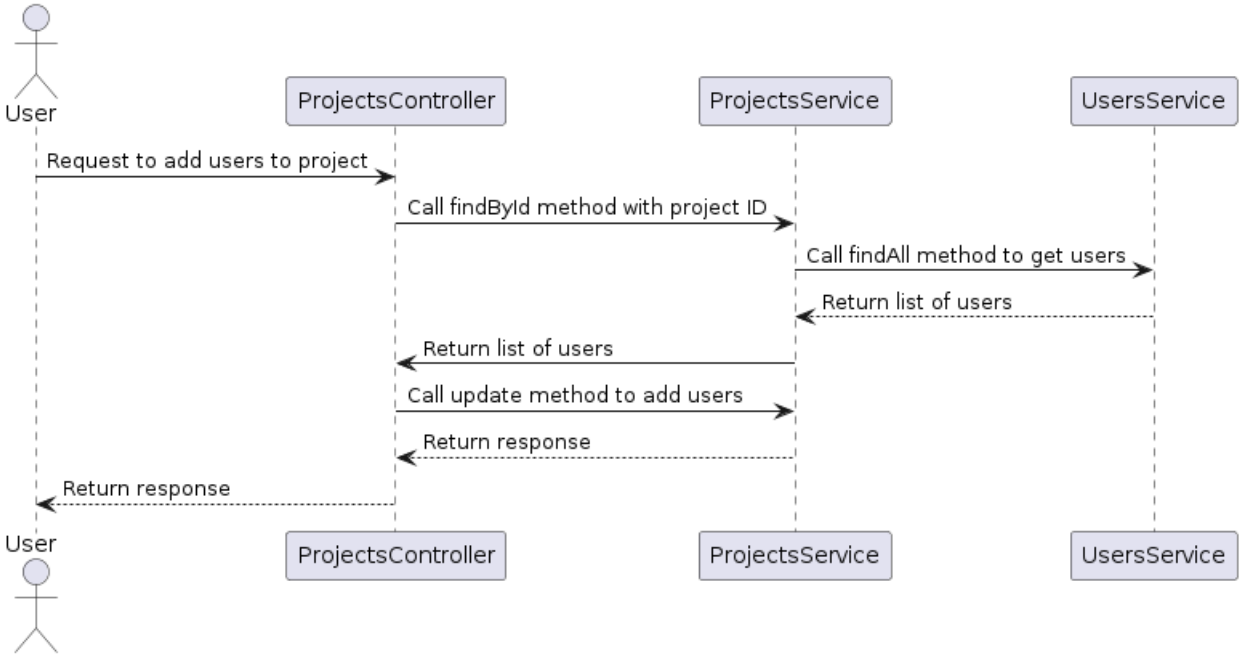
1. CRUD of Projects:



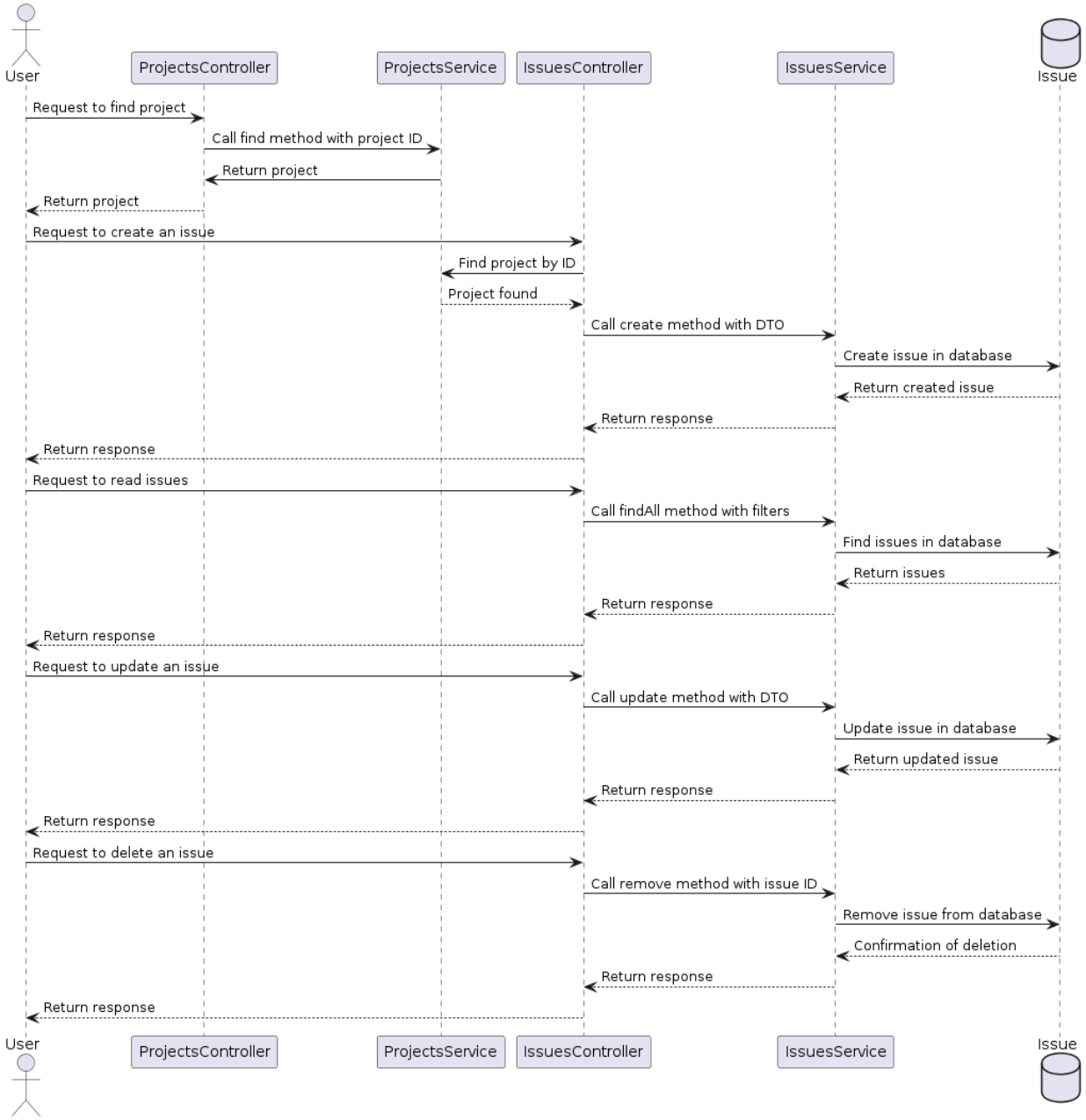
2. Filter and Sort Projects:



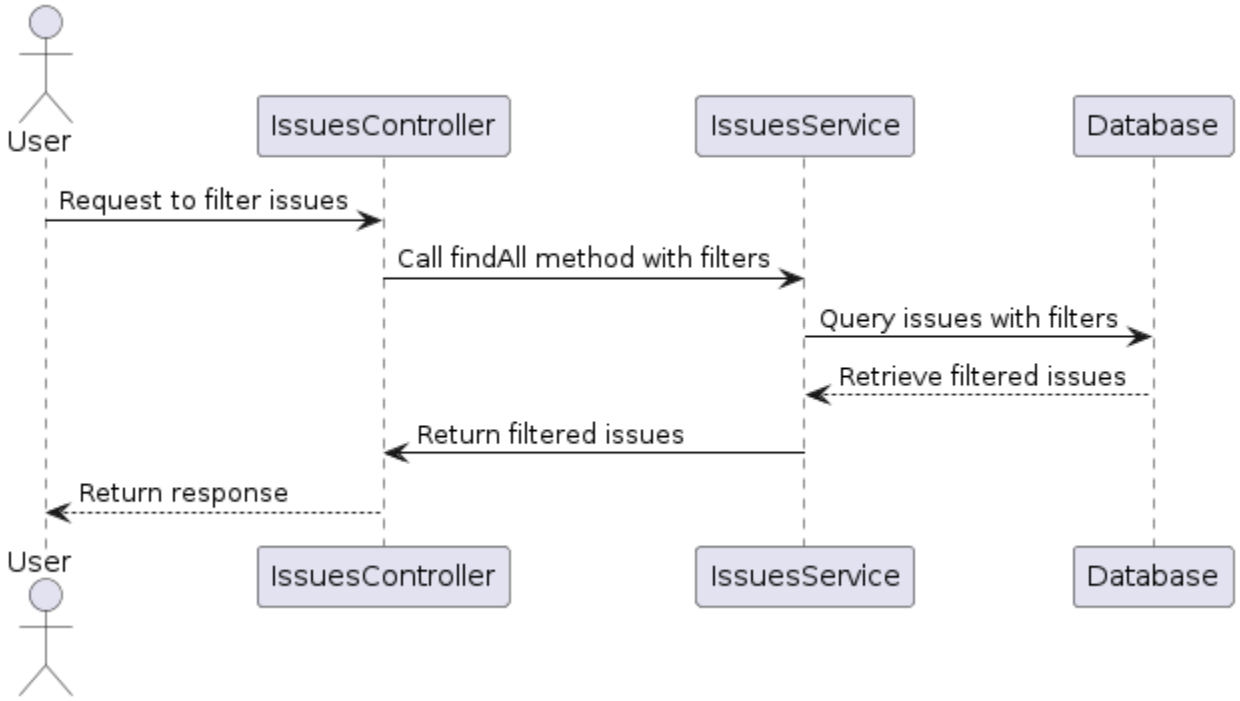
3. Add users to Project:



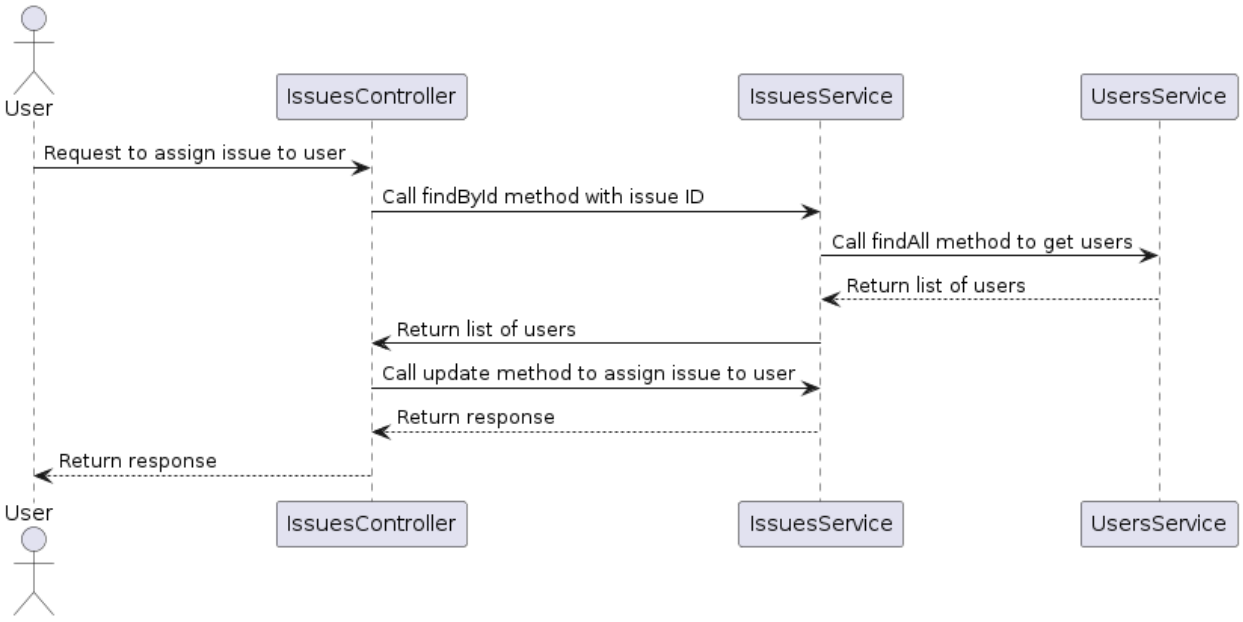
4. CRUD for Issues:



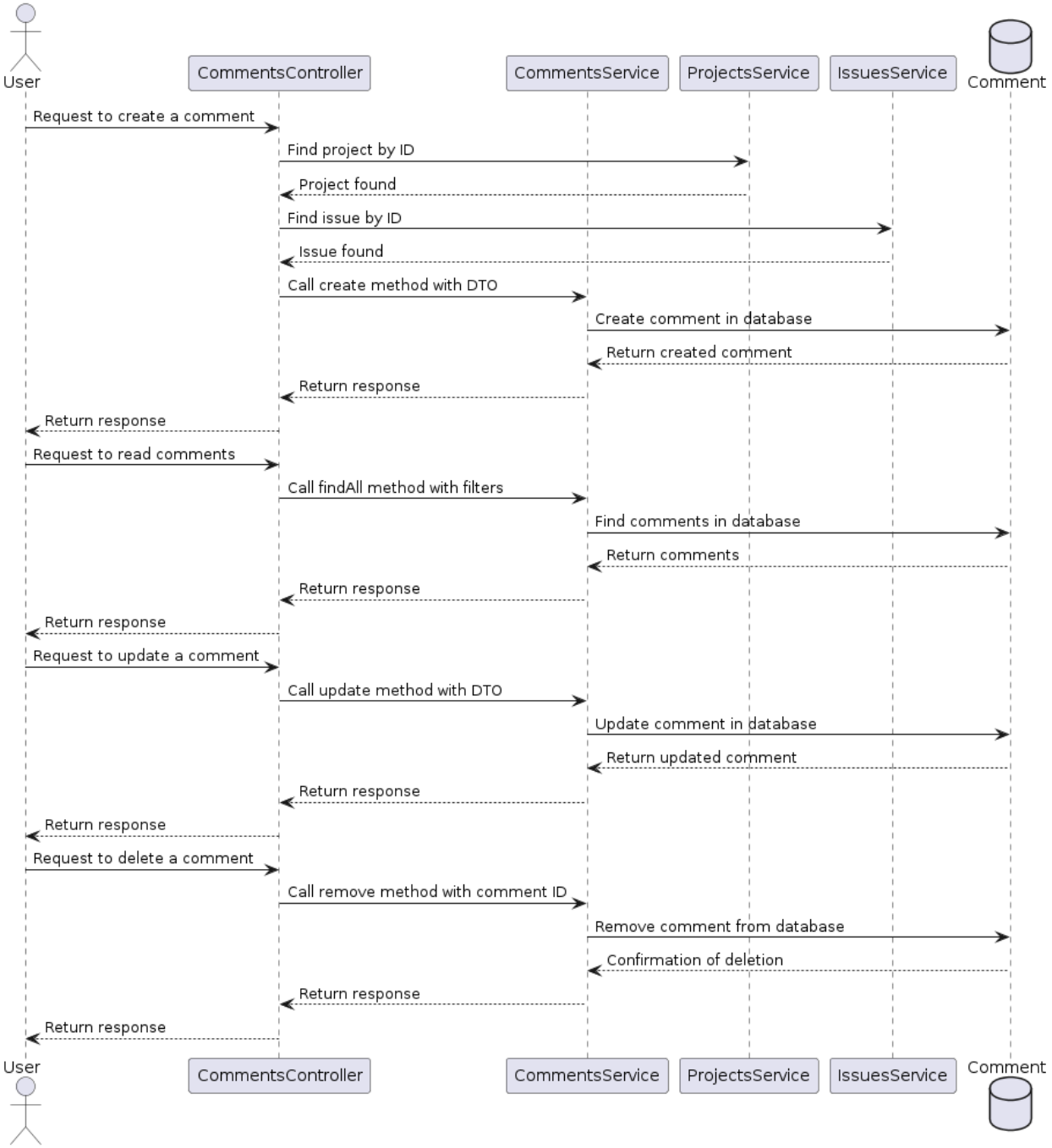
5. Filter Issues:



6. Assign Issues to Users:



7. CRUD for Comments in Issue:



Replicated code, Cohesion and Coupling issues

1. Replicated Code: In the `**issues.ts**` file, the structure of the `**assignees**` and `**reporter**` objects are identical. This is a form of code duplication. Code duplication can lead to higher maintenance costs and potential bugs because changes made to one part of the duplicated code may need to be replicated in all other parts. Here's a more detailed example:

```
// src/database/seeders/issues/issues.ts
const issues = [
  {
    assignees: [
      {
        id: '1',
        name: 'John Doe',
        // other properties ...
      },
    ],
    reporter: {
      id: '1',
      name: 'John Doe',
      // other properties ...
    },
    // other properties ...
  },
  // other issues ...
];
```

To avoid this you can create a function or a class that generates these objects

```
// src/database/seeders/issues/issues.ts
function createUser(id, name) {
  return {
    id,
    name,
    // other properties ...
  };
}

const issues = [
  {
    assignees: [createUser('1', 'John Doe')],
    reporter: createUser('1', 'John Doe'),
    // other properties ...
  },
  // other issues ...
];
```

There's a pattern of duplication in the **findAll** methods of both **ProjectsService** and **CommentsService**. Both methods are creating a query, adding conditions to it based on the input filter, and then executing the query. Here's how the code looks:

```
// src/features/projects/projects.service.ts
async findAll(filterDto: GetProjectFilterDto) {
  const { name, key } = filterDto;
  const query = this.projectModel.find();

  if (name) {
    query.where('name', { $regex: '^' + name + '$', $options: 'i' });
  }

  if (key) {
    query.where('key', { $regex: '^' + key + '$', $options: 'i' });
  }
  return query.exec();
}

// src/features/comments/comments.service.ts
findAll(filterDto: GetCommentFilterDto) {
  const { issueId } = filterDto;
  const query = this.commentModel.find();

  if (issueId) {
    query.where('issueId', issueId);
  }
  return query.exec();
}
```

To reduce this duplication, you could create a base service class that contains a generic **findAll** method. This method would take a filter object and a model as parameters, and then build and execute the query. Here's an example

```ts
// src/common/base.service.ts
class BaseService {
  async findAll(filter: any, model: any) {
    const query = model.find();

    for (const key in filter) {
      if (filter[key]) {
        query.where(key, filter[key]);
      }
    }

    return query.exec();
  }
}

// src/features/projects/projects.service.ts
class ProjectsService extends BaseService {
  /* ... */
  async findAll(filterDto: GetProjectFilterDto) {
    return this.findAll(filterDto, this.projectModel);
  }
  /* ... */
}

// src/features/comments/comments.service.ts
class CommentsService extends BaseService {
  /* ... */
  findAll(filterDto: GetCommentFilterDto) {
    return this.findAll(filterDto, this.commentModel);
  }
  /* ... */
}
```

This way, the **findAll** logic is defined in one place (**BaseService**), and the specific services (**ProjectsService and CommentsService**) only need to call this method with the appropriate parameters.

2. Cohesion

Although the codebase is well-structured and follows the modular design principle, which is a good practice for maintaining high cohesion. However, there are a few areas where improvements can be made to increase cohesion:

A) Environment Configuration Duplication: The environment configuration code is duplicated in both **app.module.ts** and **seeder.module.ts**. This could lead to inconsistencies if changes are made in one place and not the other.

Solution: Extract the environment configuration into a separate module and import it in both **app.module.ts** and **seeder.module.ts**.

B) Seeder Module Responsibility: The **seeder.module.ts** is responsible for seeding different entities like users, projects, issues, and comments. This could lead to low cohesion if the seeding process for these entities becomes complex.

Solution: Keep the responsibility of each seeder module to only seeding its respective entity. If the seeding process becomes complex, consider breaking it down further into more specialized modules or services.

C) Schema Definition Location: The schema definitions for entities like Comment are in the features modules (e.g., comments.module.ts). This could lead to low cohesion if the schema definitions are used in multiple places outside their respective feature modules.

Solution: Consider moving the schema definitions to a common location where they can be imported by any module that needs them. This would increase the cohesion of the feature modules and make the schema definitions more reusable.

D)Violating Single Responsibility Principle: IssuesService and **ProjectsService** classes also have cohesion issues. They both have methods for creating, finding, updating, and removing their respective entities. This is good for cohesion as each service is responsible for operations related to a single entity. But, the IssuesService also has a method **deleteIssuesByProjectId** which seems to be handling a responsibility that might be more appropriate for the ProjectsService, since it involves operations on a project level.

Solution: A potential solution could be to move the deleteIssuesByProjectId method from the IssuesService to the ProjectsService. This way, the **ProjectsService** would handle all operations related to projects, including deleting all issues related to a specific project. This would improve the cohesion of your services.

```
class ProjectsService {
  constructor(
    @InjectModel(Project.name) private projectModel: Model<ProjectDocument>,
    private readonly issuesService: IssuesService,
  ) {/* ... */}

  // Other methods ...

  async deleteIssuesByProjectId(projectId: string, session: ClientSession) {
    // Call the appropriate method from issuesService here
    return this.issuesService.deleteIssuesByProjectId(projectId, session);
  }
}
```

And then remove the **deleteIssuesByProjectId** method from the **IssuesService**.

3. Cohesion

The **ProjectsService class** in backend/src/features/projects/projects.service.ts is coupled to the **IssuesService class**. This means that changes in IssuesService could affect ProjectsService.

The **IssuesService class** in backend/src/features/issues/issues.service.ts is coupled to the **CommentsService class**. This means that changes in **CommentsService** could affect **IssuesService**.

To reduce this coupling, you could consider the following solutions:

Dependency Inversion Principle (DIP): Instead of directly depending on **IssuesService** and **CommentsService**, you could depend on abstractions (interfaces) that these services implement. This way, **ProjectsService** and IssuesService would be shielded from changes in the services they depend on.

Event-driven architecture: Instead of directly calling methods on IssuesService or CommentsService, you could emit events that these services listen to and react upon. This way, the services are decoupled and can evolve independently.

Use a mediator: A mediator object could be used to encapsulate and centralize the interactions between **ProjectsService**, **IssuesService**, and **CommentsService**. This way, the services don't need to know about each other, reducing the coupling.

Restructuring System to the level of Abstraction

Naming Convention: Use clear and descriptive names for your classes and methods. For example, in your **src/database/seeders/** directory, you could name your seeders as **UserSeeder**, **ProjectSeeder**, **IssueSeeder**, etc. This makes it clear what each seeder is responsible for.

Directory Structure: Organize the files in a way that reflects their purpose and relationship. For example, all your database related files (models, seeders, etc.) are in the src/database/ directory, which is good. You could further improve this by having a models/ directory inside src/database/ to separate your models from your seeders.

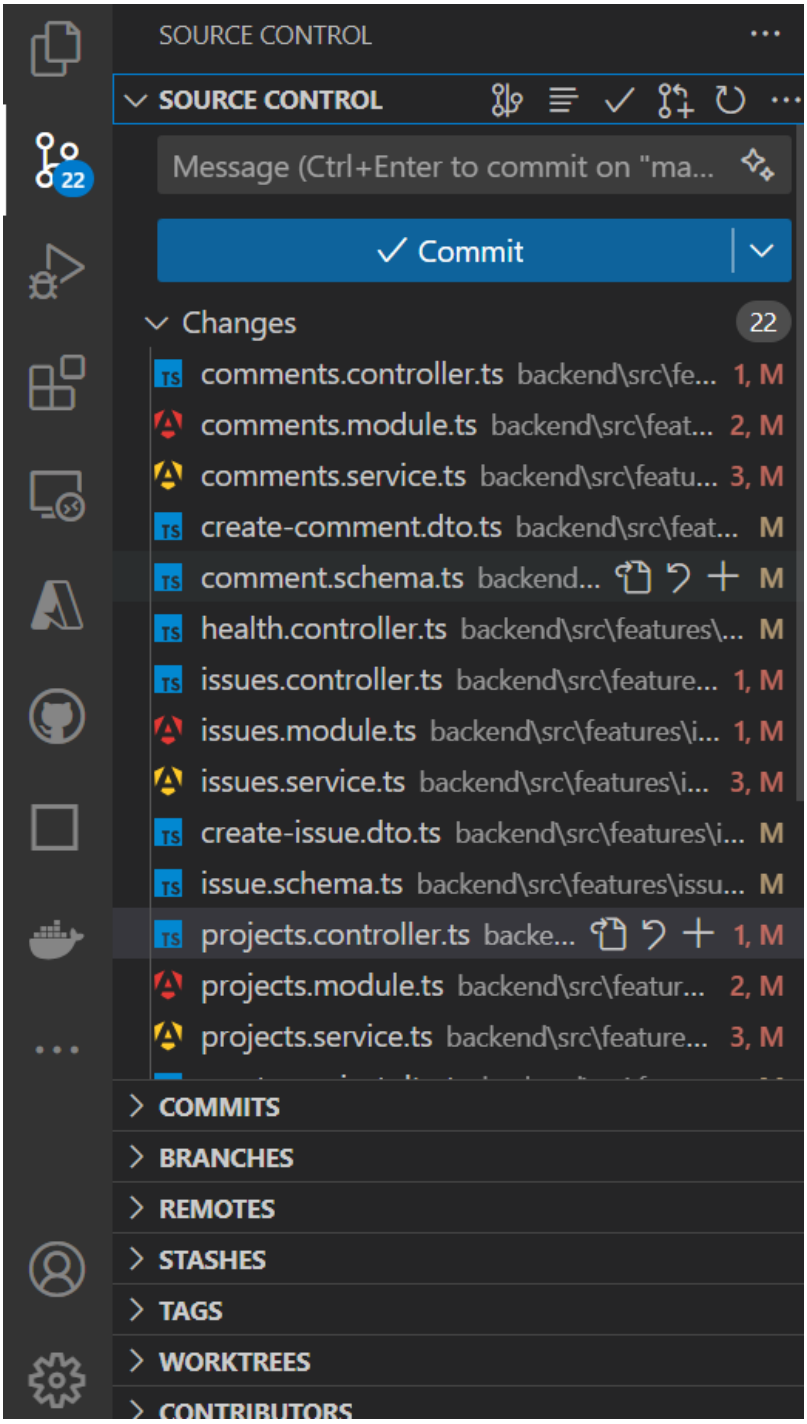
Consistent Analysis and Design Artifacts: Ensure that your codebase aligns with your design documents. For example, if you have a UML diagram that outlines your class structure, make sure it matches the actual structure in your codebase. This helps maintain consistency and makes your codebase easier to understand.

Modularization: Breaking down the code into smaller, reusable modules. For example, in your **src/features/** directory, each feature (**comments, health, issues, projects, users**) could be a separate module with its own controllers, services, and models.

Environment Configuration: Use environment variables for different stages of your application (dev, prod). That is already done in the package.json scripts with cross-env **STAGE=dev** and **cross-env STAGE=prod**, which is a good practice.

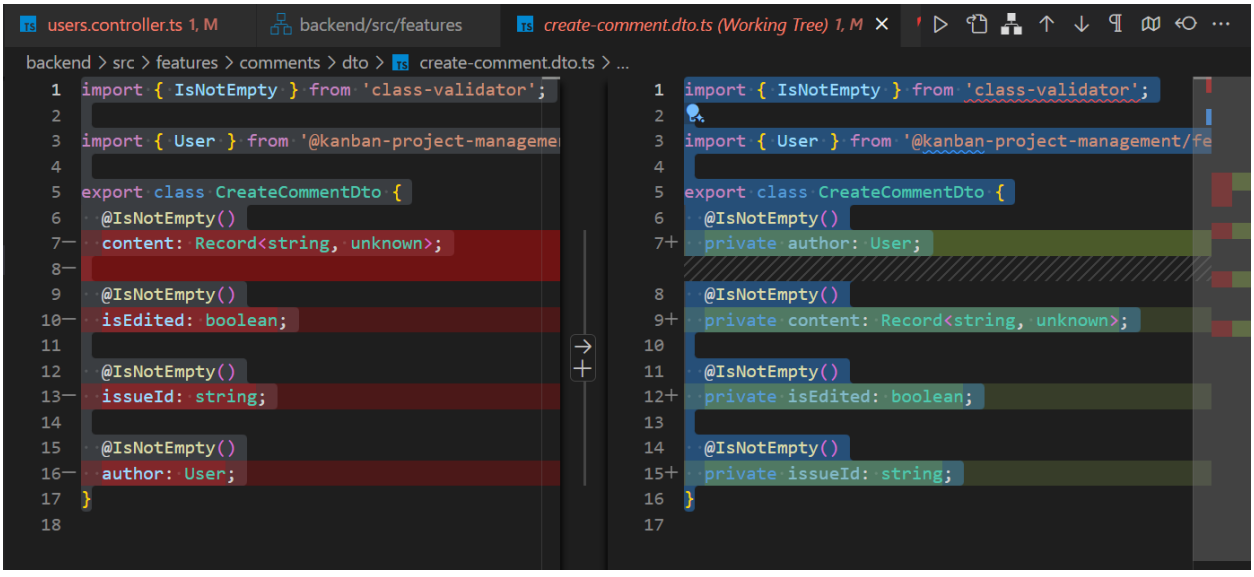
Test Organization: Organize the tests to mirror your application structure. For example, tests for the **src/features/comments/** directory should be in **test/features/comments/**.

All together 22 files are refactored by applying appropriate naming conventions (variable. method and class). Other changes are also made to increase the overall efficiency of the code.

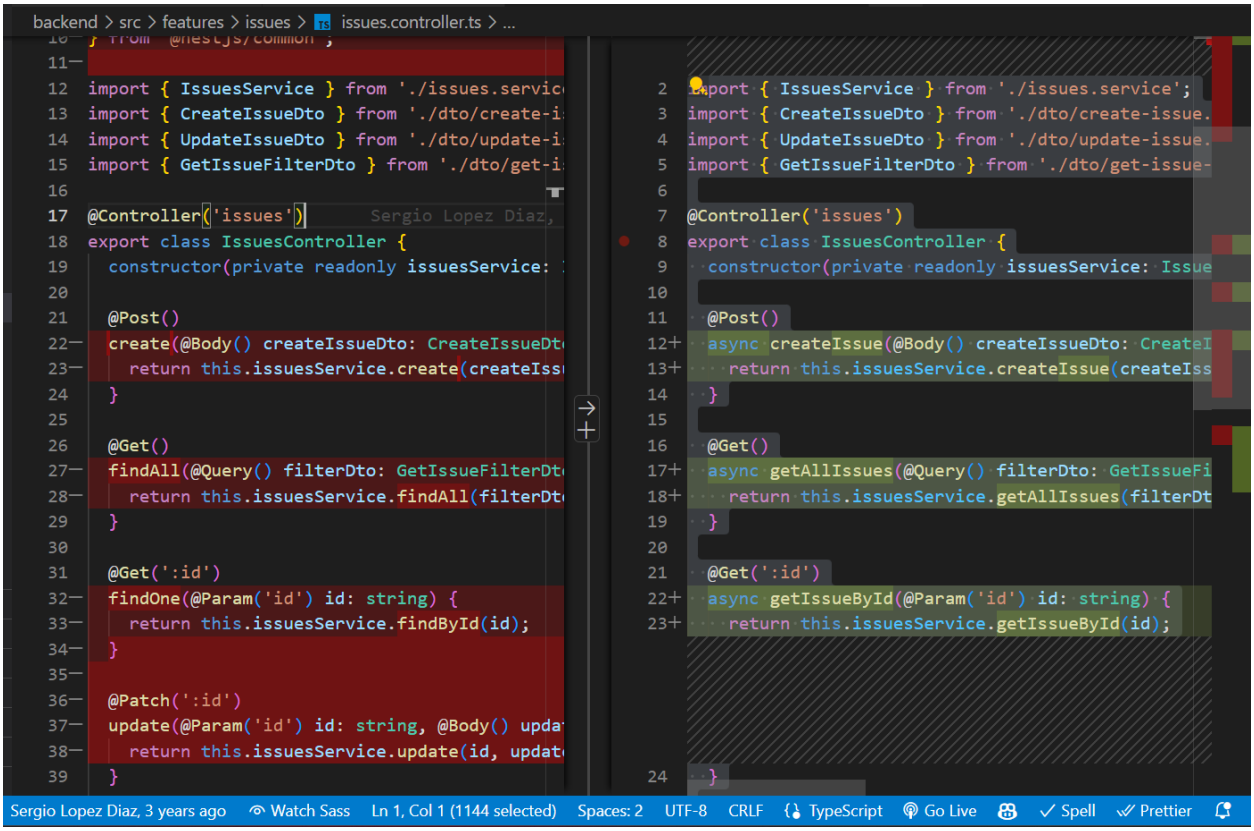


Few of the files changes are displayed here:

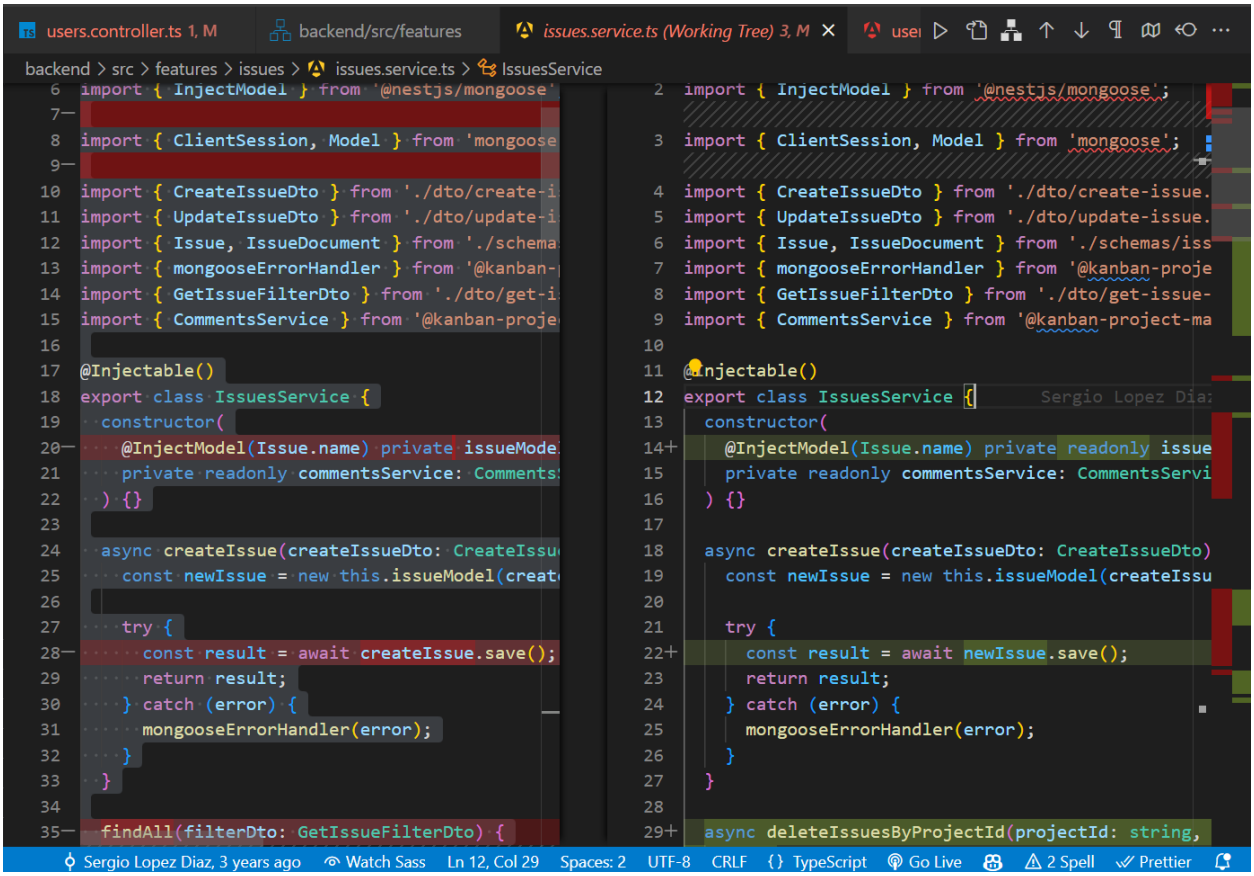
Appropriate naming convention is used for methods in this file.



In the previous code, the properties content, isEdited, issueId, and author were public by default. Making them private can provide better encapsulation and help prevent unintended manipulation of the data from outside the class.



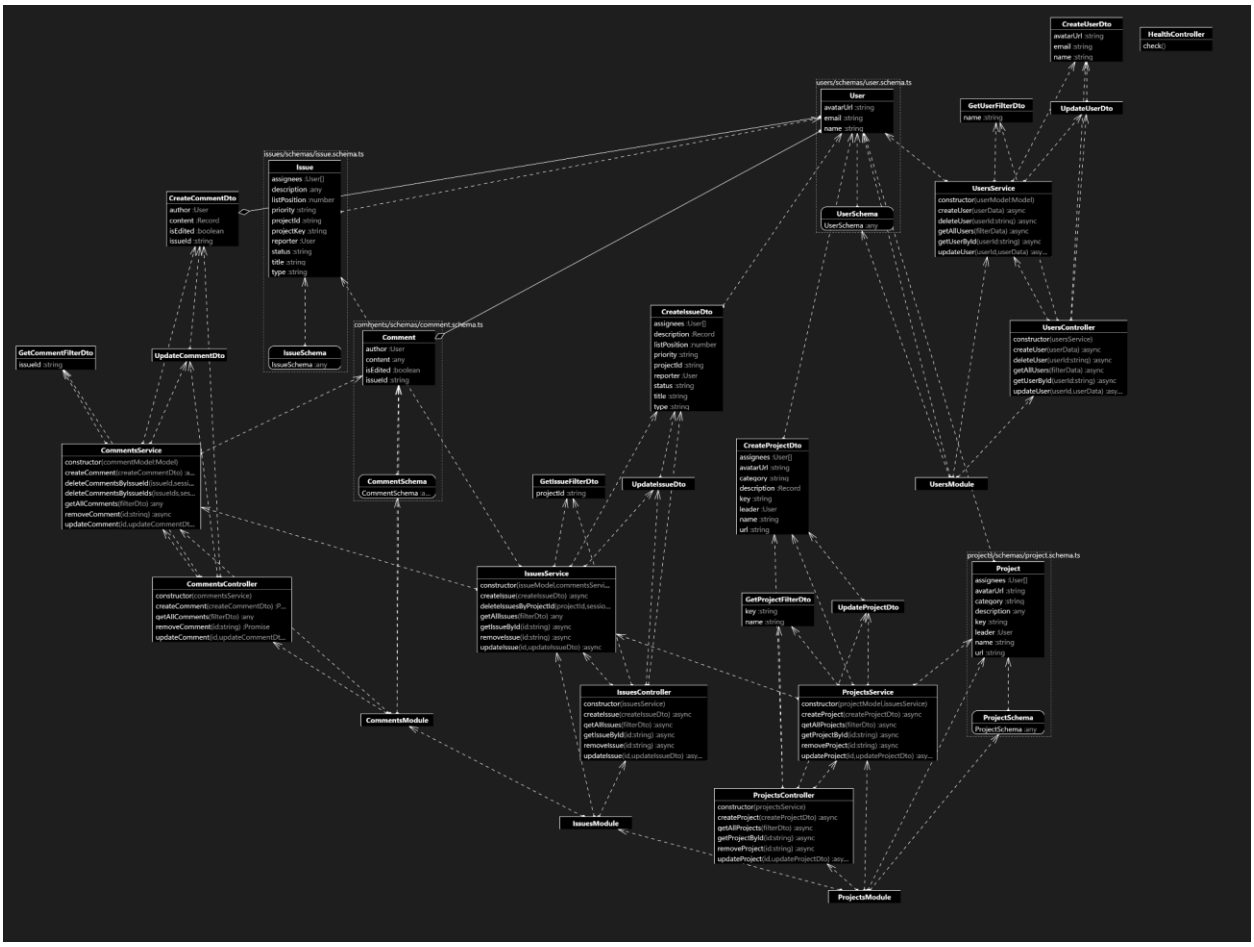
By using the async keyword, you ensure that these controller methods can handle asynchronous operations efficiently, improving the overall performance and responsiveness of your NestJS application.



In the refactored code, the readonly keyword is used in front of the issueModel property in the IssuesService class constructor.

readonly is used to ensure that the issueModel property in the IssuesService class remains immutable once it's initialized, helping prevent unintended modifications and maintaining code consistency.

Refactored Class Diagram:



Tools for Refactoring

1. PlantUml:

PlantUML is a tool used for creating UML (Unified Modeling Language) diagrams from plain text descriptions. It simplifies the process of creating and maintaining UML diagrams, making it a popular choice among developers, software architects, and other professionals involved in software design and development.

In this project we used it to create the System Sequence and Detailed Sequence diagram of the functionalities from the old code to reverse engineer it to artifacts.

2. Classdiagram-ts (VS Code Library):

- Create UML class diagram for a folder or file, display class information: type (class, abstract class or interface), name, members (property and method), and the relationships between classes.
- Open correspondent code when a class or class members in diagram is clicked.
- Diagram is in sync with code, reflects code change immediately.
- Code refactoring. "Tools/Refactor" sorts class members and adds 'private' modifier based on dependency analysis.

In this project, we are using it first for reverse engineering by creating a class diagram of the old code then we use it’s refactor functionality to refactor the code. Afterwards we perform our own refactoring and in the end generate the new class diagram from the refactored code.