UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

HelpDesk Management System

Submitted by:

Dhruthan M N (PES2UG22CS181)

Sanket B. Muttur (PES1UG23CS846)

Veeresh Amaragatti (PES1UG23CS847)

Vineet Goel (PES1UG22CS697)

6th Sem -L Section

Sowmya Shree P

January - May 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement:

In many organizations and online communities, addressing technical issues and sharing knowledge efficiently can be challenging due to the lack of a centralized platform. Existing systems often lack flexibility, are difficult to manage, or do not offer proper moderation and collaboration tools. There is a need for a robust, secure, and user-friendly platform that enables users to report problems, seek solutions, and contribute to a growing knowledge base—all while maintaining administrative control and content integrity.

The **HelpDesk Management System** is designed to solve this by providing a centralized, feature-rich web application where users can post technical issues, receive help from the community, and contribute to the resolution of problems through a structured and moderated platform.

Key Features

Major Use Cases

1. **User Authentication and Profile Management:**
    - Registration
    - Login (Normal user/Admin)
    - View Profile
    - Update Profile

    This is a major use case because it's fundamental to the system - users need to authenticate before accessing most features, and profile management is essential for user identity.
2. **Post Management:**
    - View Posts
    - Create Posts
    - Edit Posts
    - Delete Posts
    - Search Posts (by Title/Category/Tag/Content)

Post management represents the core functionality of the HelpDesk system. It's the primary way users interact with the platform and share information.

3. **Comment and Interaction System:**
    - Write comments on posts
    - Reply to comments
    - Upvote/Downvote posts and comments

    This is a major use case as it enables user engagement and interaction, which is central to a community-based help desk system.

4. **Admin Content Moderation:**
    - View/Hide/Delete Posts
    - View/Hide/Delete Comments
    - Manage Categories
    - Handle Reports

    Content moderation is critical for maintaining quality and appropriate content on the platform, making it a major use case for administrators.

Minor Use Cases

1. **User Role Management:**
    - Change user roles
    - Suspend users
    - Ban users

    While important for administration, these features are used less frequently than the core content management functions.

2. **Reporting System:**
    - Report posts
    - Report comments

    The reporting system is a supporting feature that helps maintain content quality but is used less frequently than the main interaction features.

3. **Category and Tag Management:**
    - Create categories
    - Edit categories
    - Delete categories
    - View posts by category/tag

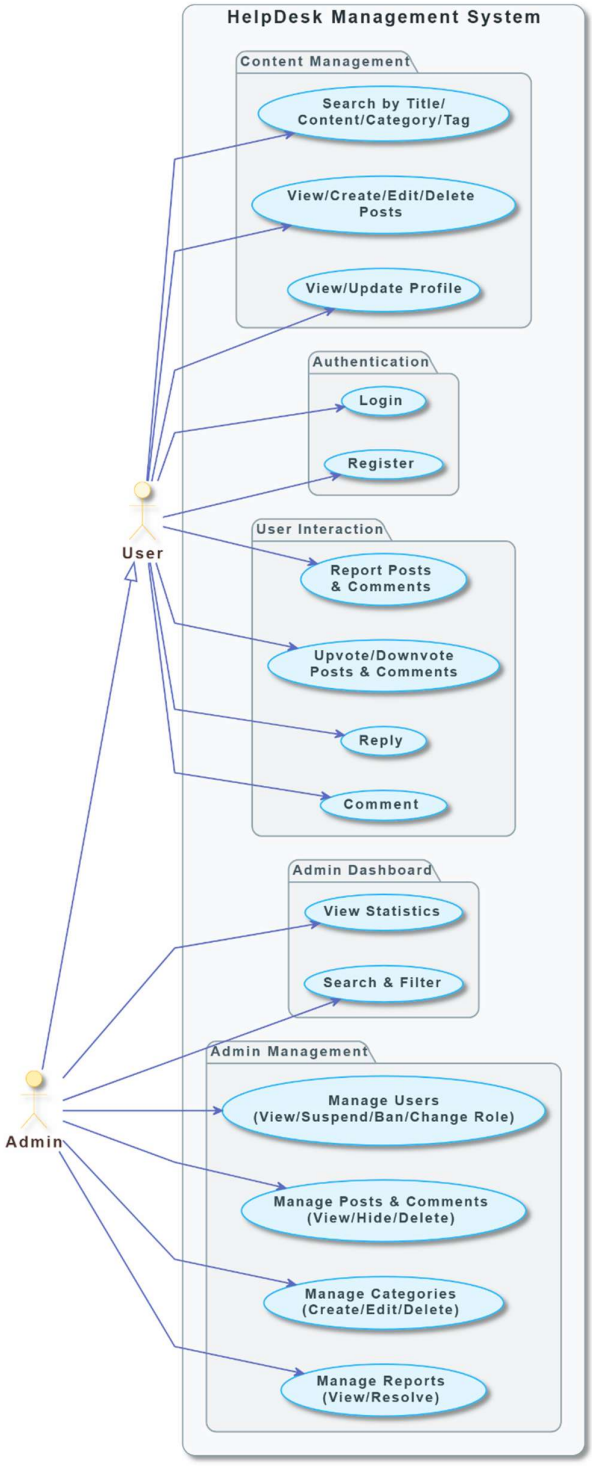
    This is classified as minor because it's primarily an organizational feature that supports the main content system.

4. **Admin Dashboard and Analytics:**
    - View statistics (users, posts, comments, reports)
    - Search and filter
    - Categorize handling of system entities

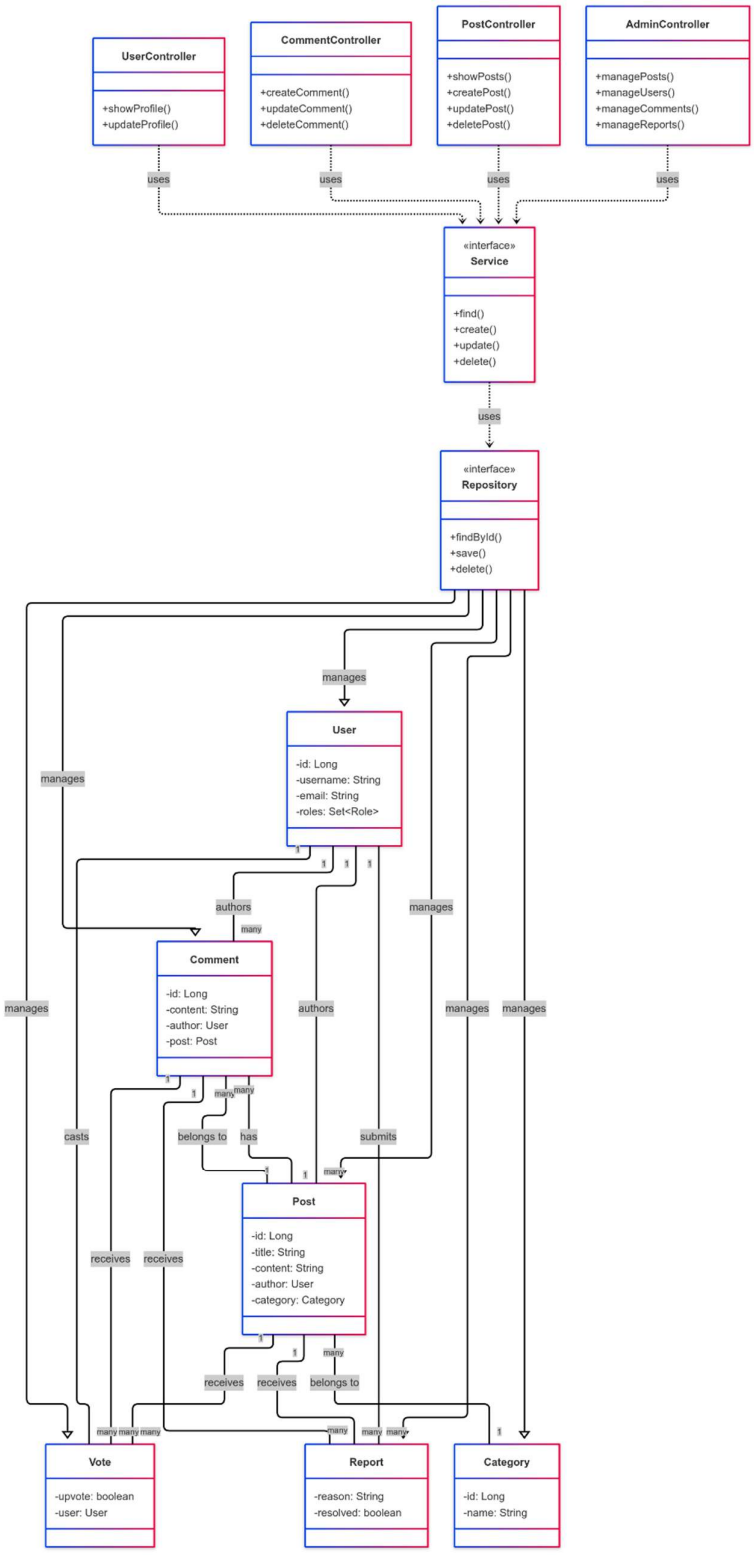
    While valuable for administrators, these analytical features support decision-making rather than providing core functionality.

Models

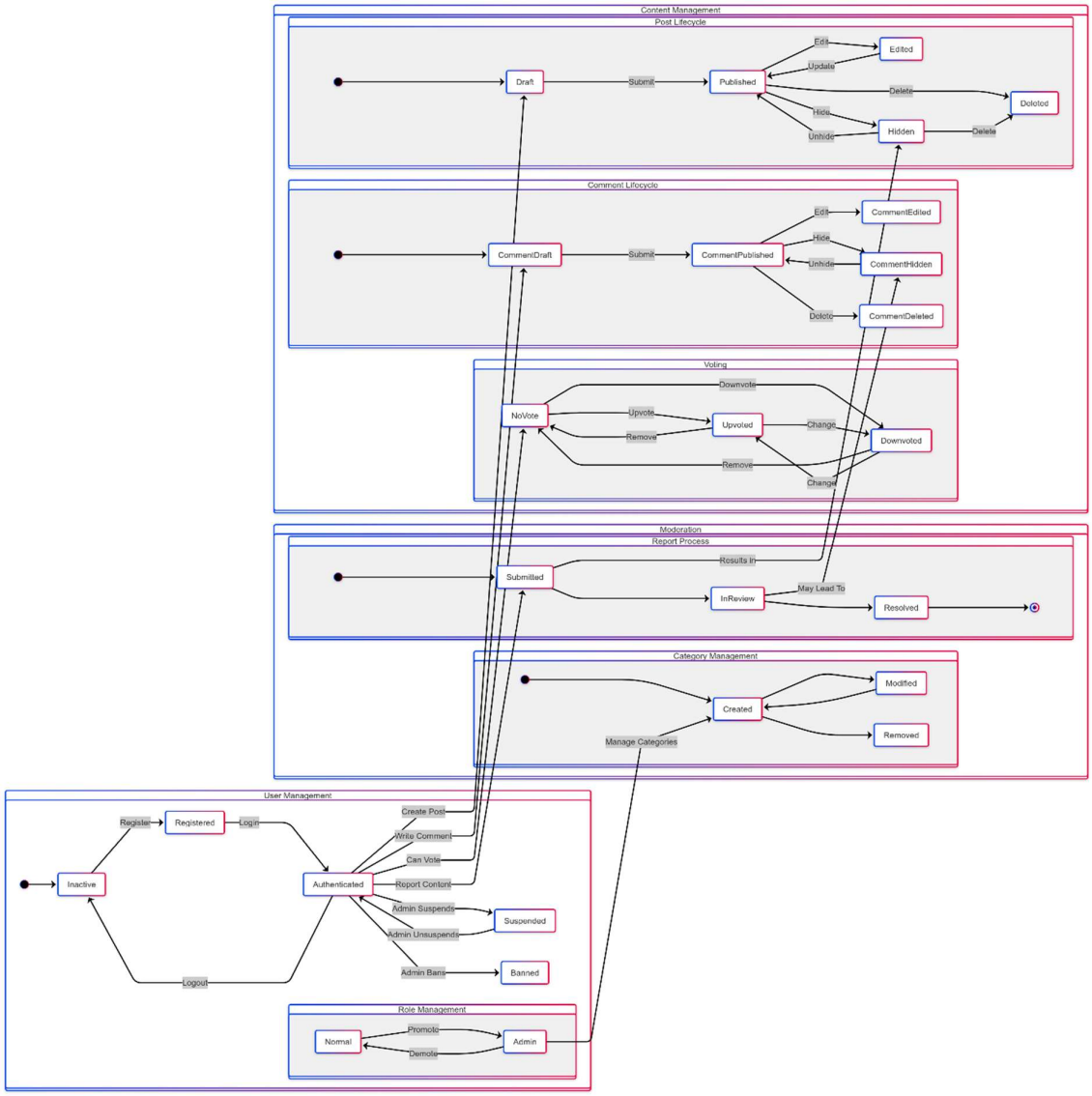
Use Case Diagram:



Class Diagram:

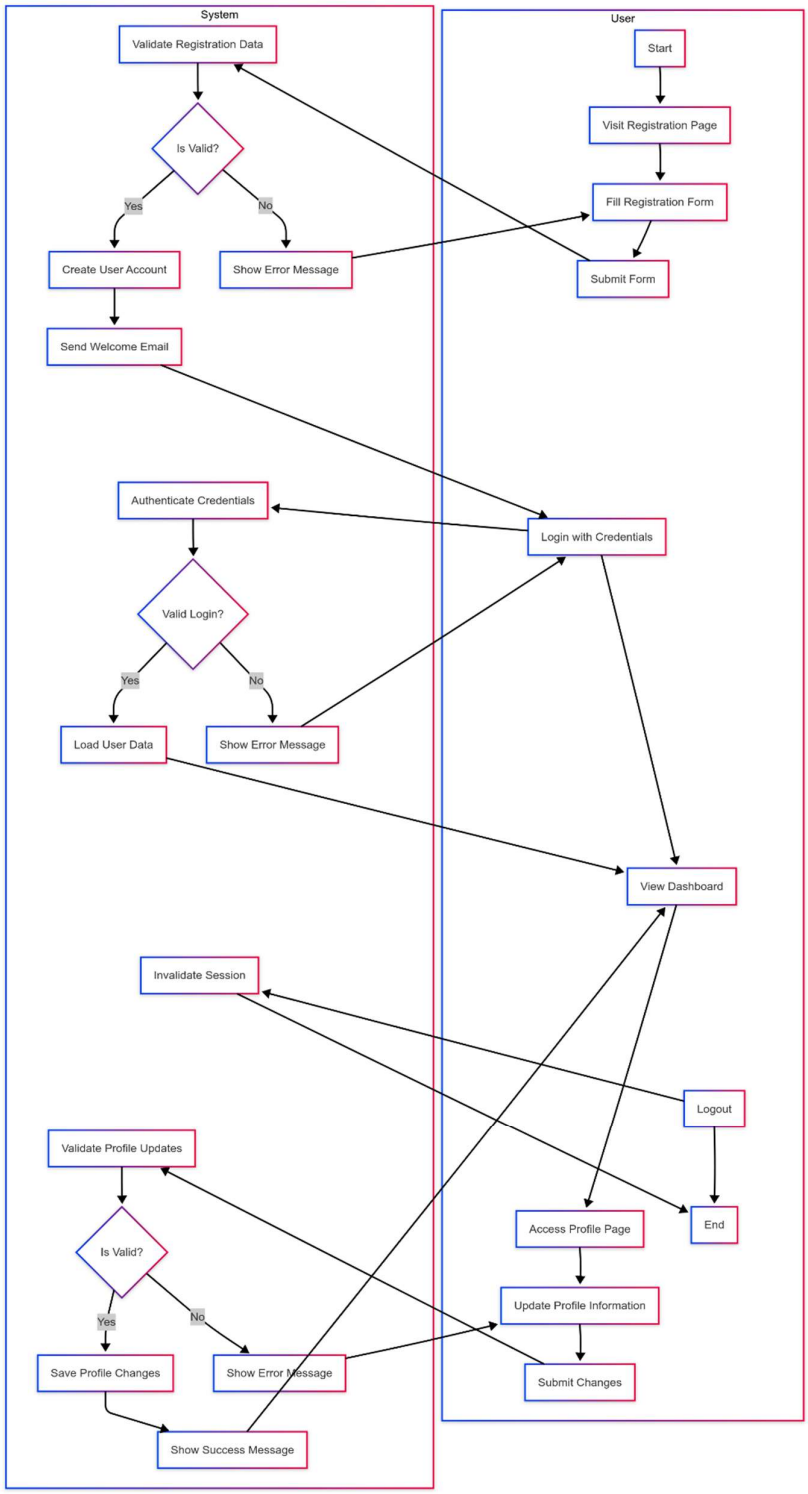


State Diagram:

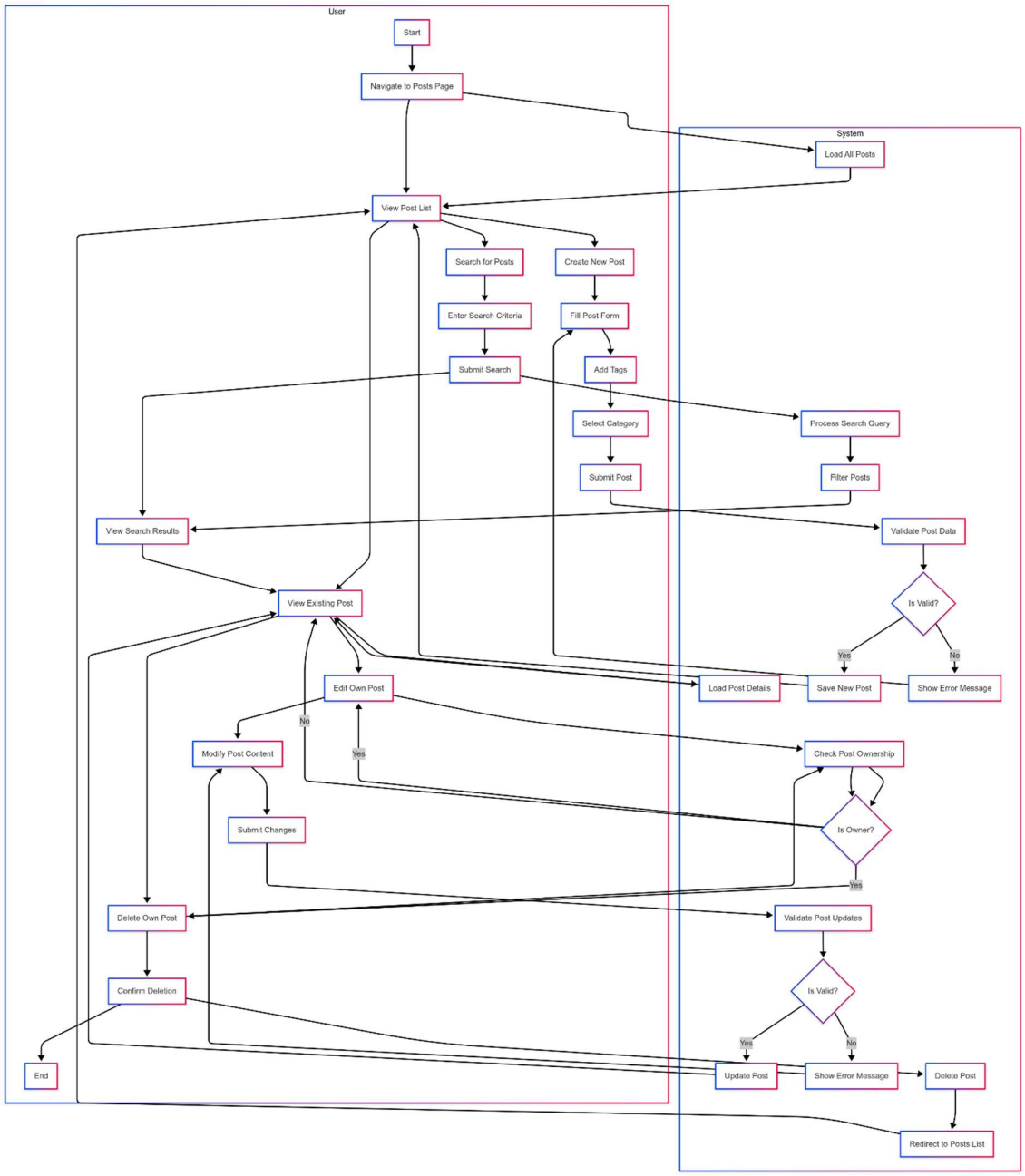


Activity Diagrams:

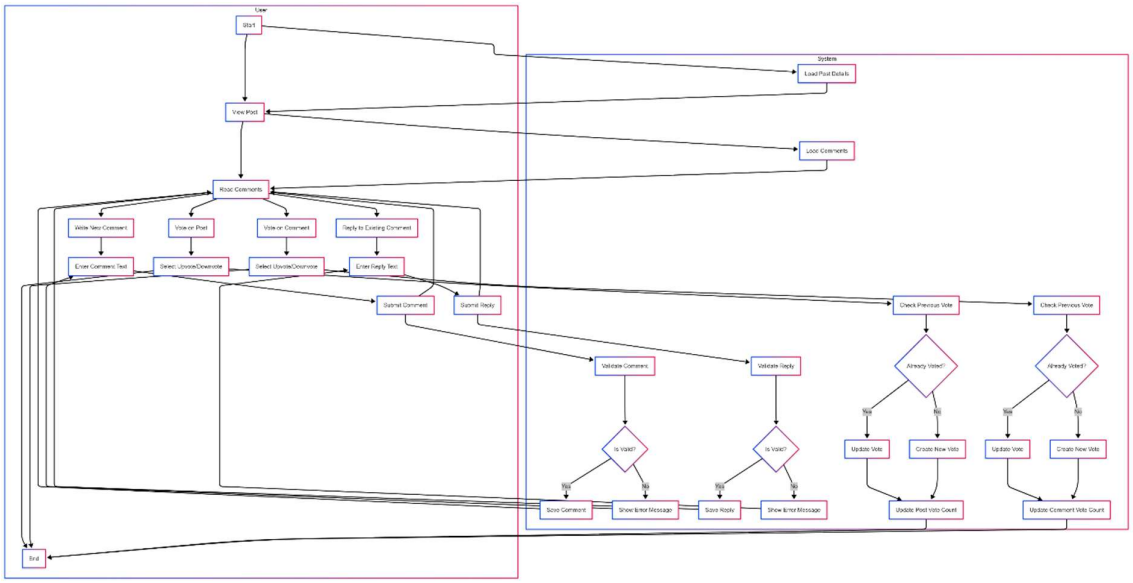
Major Use case 1 - User Authentication and Profile Management:



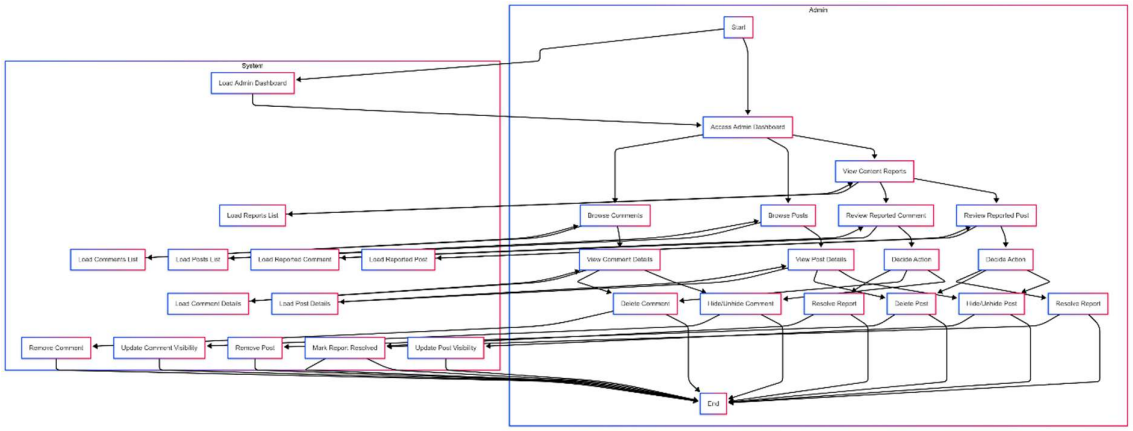
Major Use case 2 - Post Management:



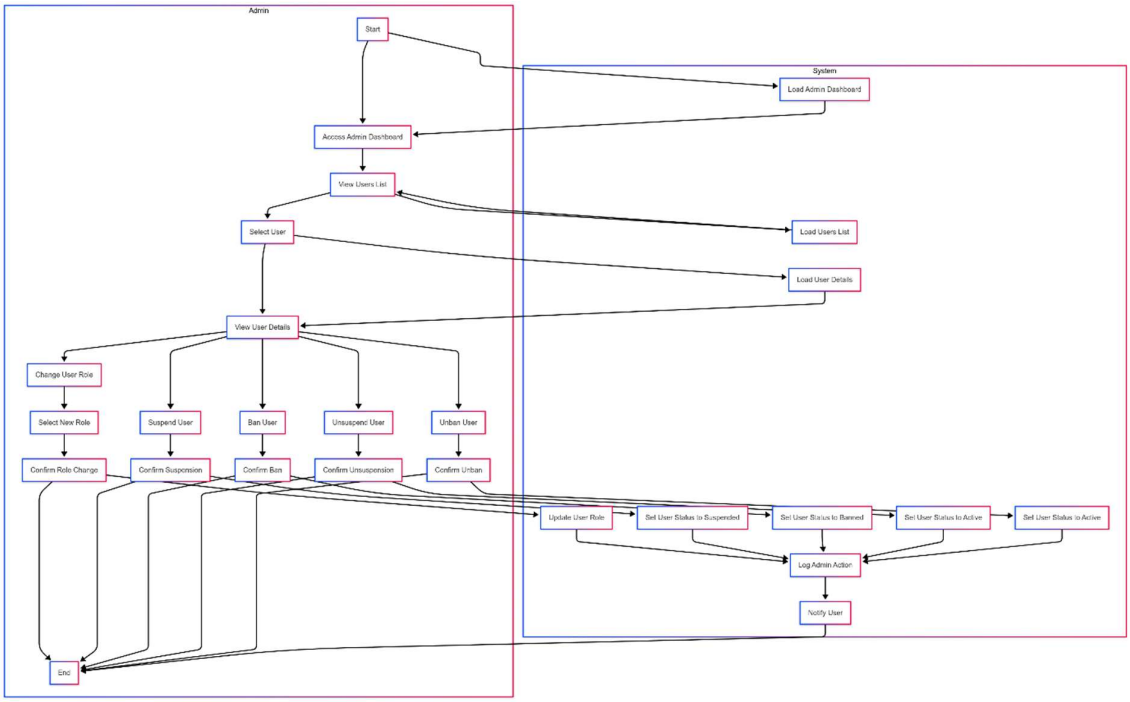
Major Use case 3 - Comment and Interaction System:



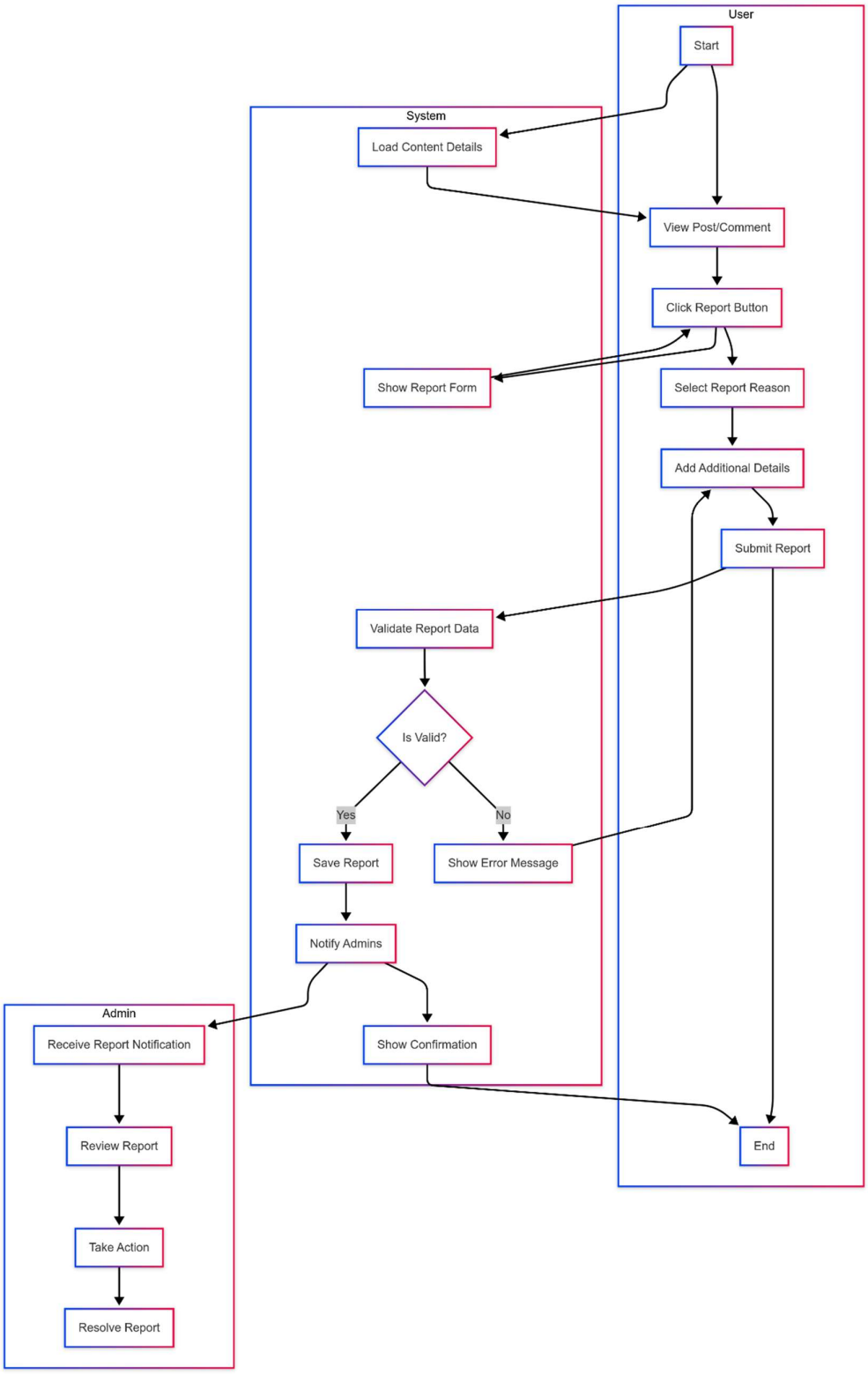
Major Use case 4 - Admin Content Moderation:



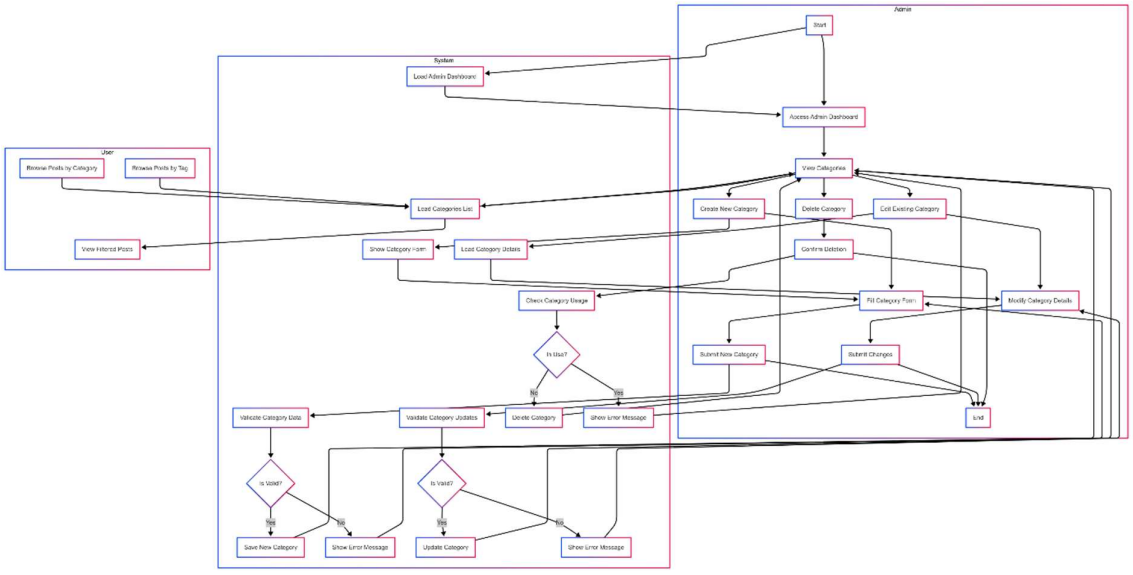
Minor Use case 1 - User Role Management:



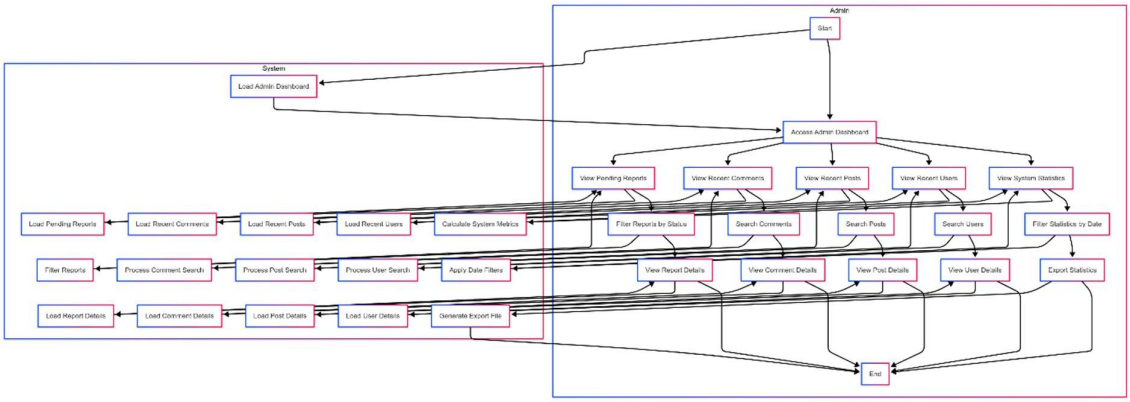
Minor Use case 2 - Reporting System:



Minor Use case 3 - Category and Tag Management:



Minor Use case 4 - Admin Dashboard and Analytics:



Architecture Patterns: Model-View-Controller (MVC) in HelpDesk Management System

Overview of MVC Architecture

The HelpDesk Management System implements the Model-View-Controller (MVC) architectural pattern, which separates the application into three interconnected components. This separation enhances maintainability, scalability, and testability of the application while promoting code reusability and parallel development.

Components of MVC in the HelpDesk System

1. Model Layer

The Model layer represents the application's data structure and business logic. It is responsible for retrieving, processing, and storing data.

Key Components:

1. **Domain Entities**: Java classes that represent the core business objects.
    - **User**: Represents system users with attributes like username, email, and roles
    - **Post**: Represents help desk posts created by users
    - **Comment**: Represents user comments on posts
    - **Category**: Represents post categories for organization
    - **Vote**: Represents user votes on posts and comments
    - **Report**: Represents content reports submitted by users
    - **Role**: Represents user roles for authorization
2. **Repositories**: Interfaces that extend Spring Data JPA's **JpaRepository** to provide data access operations.
    - **UserRepository**: Manages user data persistence
    - **PostRepository**: Handles post-related database operations
    - **CommentRepository**: Manages comment data
    - **CategoryRepository**: Handles category data
    - **VoteRepository**: Manages voting data
    - **ReportRepository**: Handles report data
    - **RoleRepository**: Manages role data

3. **Services**: Classes that implement business logic and act as an intermediary between controllers and repositories.
    - **UserService**: Handles user-related operations like registration and profile management
    - **PostService**: Manages post creation, retrieval, and moderation
    - **CommentService**: Handles comment operations
    - **CategoryService**: Manages category operations
    - **VoteService**: Handles voting functionality
    - **ReportService**: Manages content reporting
4. **Data Transfer Objects (DTOs)**: Objects used to transfer data between subsystems.
    - **UserDto**, **UserProfileDto**, **UserRegistrationDto**: Transfer user data
    - **PostDto**: Transfers post data
    - **CommentDto**: Transfers comment data
    - **CategoryDto**: Transfers category data
    - **VoteDto**: Transfers vote data
    - **ReportDto**: Transfers report data

2. View Layer

The View layer is responsible for rendering the user interface and presenting data to users. In the HelpDesk system, this is implemented using Thymeleaf templates.

Key Components:

1. **Thymeleaf Templates**: HTML templates with Thymeleaf expressions for dynamic content rendering.
    - **login.html**, **register.html**: Authentication views
    - **dashboard.html**: Main user dashboard
    - **profile.html**: User profile view
    - **posts/list.html**, **posts/view.html**, **posts/create.html**, **posts/edit.html**: Post-related views
    - **admin/dashboard.html**, **admin/users.html**, **admin/posts.html**: Admin views
2. **Static Resources**:
    - CSS stylesheets for visual presentation
    - JavaScript files for client-side functionality
    - Images and other media assets

3. Controller Layer

The Controller layer handles user requests, processes them (often by interacting with the Model layer), and returns the appropriate View.

Key Components:

1. **Web Controllers**: Spring MVC controllers that handle HTTP requests.
    - **AuthController**: Manages authentication operations
    - **UserController**: Handles user profile operations
    - **DashboardController**: Manages dashboard views
    - **PostController**: Handles post-related operations
    - **CommentController**: Manages comment operations
    - **CategoryController**: Handles category operations
    - **VoteController**: Manages voting functionality
    - **ReportController**: Handles content reporting
    - **AdminController**: Manages administrative functions
2. **REST Controllers**: Controllers that provide RESTful API endpoints.
    - API versions of the controllers for programmatic access

Benefits of MVC in the HelpDesk System

1. **Separation of Concerns**: Each component has a specific responsibility, making the codebase more organized and maintainable.
2. **Code Reusability**: Models and business logic can be reused across different views and controllers.
3. **Parallel Development**: Different team members can work on models, views, and controllers simultaneously.
4. **Testability**: Components can be tested in isolation, facilitating unit testing.
5. **Flexibility**: Changes to one component have minimal impact on others, allowing for easier modifications and enhancements.

Design Principles in the HelpDesk Management System

1. SOLID Principles

Single Responsibility Principle (SRP)

The HelpDesk system demonstrates SRP by ensuring each class has only one reason to change:

- Model Classes: Each entity class (User, Post, Comment, etc.) represents a single domain concept
- Repository Interfaces: Each repository is focused on data access for a specific entity
- Service Classes: Services handle specific business logic for their respective domains
- Controllers: Each controller manages requests related to a specific feature area For example, the UserRepository focuses solely on user data access operations, while the UserService handles user-related business logic.

Open/Closed Principle (OCP)

The system is designed to be open for extension but closed for modification:

- Repository Interfaces: Spring Data JPA repositories allow extending functionality without modifying existing code
- Service Interfaces: Services are defined through interfaces, enabling different implementations
- DTOs: Data Transfer Objects facilitate extension of data structures without affecting core entities

Liskov Substitution Principle (LSP)

The system adheres to LSP through proper inheritance hierarchies:

- User Roles: Different user roles (Admin, Regular User) can be substituted without affecting the system's behavior
- Repository Interfaces: All repositories extend JpaRepository, ensuring consistent behavior

Interface Segregation Principle (ISP)

The system implements ISP by providing focused interfaces:

- Service Interfaces: Each service interface contains only methods relevant to its domain
- Repository Interfaces: Repositories define only methods needed for their specific entity

Dependency Inversion Principle (DIP)

The system follows DIP through dependency injection and abstraction:

- Constructor Injection: Dependencies are injected via constructors
- Interface-based Design: High-level modules depend on abstractions, not concrete implementations

2. Separation of Concerns (SoC)

The HelpDesk system clearly separates different concerns:

- Presentation Logic: Controllers handle HTTP requests and responses
- Business Logic: Services implement domain-specific operations
- Data Access Logic: Repositories manage database interactions
- Domain Logic: Entity classes encapsulate domain concepts and rules This separation makes the system easier to maintain and test, as changes to one concern don't affect others.

3. Don't Repeat Yourself (DRY)

The system avoids code duplication through:

- Common Base Classes: Shared functionality is extracted into base classes
- Utility Classes: Common operations are centralized in utility classes
- Template Fragments: Thymeleaf template fragments for reusable UI components

4. KISS (Keep It Simple, Stupid)

The system maintains simplicity through:

- Clear Naming Conventions: Classes, methods, and variables have descriptive names
- Focused Components: Each component has a clear, single purpose
- Straightforward Workflows: User journeys follow intuitive paths

5. YAGNI (You Aren't Gonna Need It)

The system avoids unnecessary complexity by:

- Implementing Required Features: Only features specified in requirements are implemented
- Minimal Dependencies: Only necessary libraries are included
- Focused Functionality: Each component provides only what's needed

6. Law of Demeter (Principle of Least Knowledge)

The system follows the Law of Demeter by:

- Encapsulated Access: Objects interact only with their immediate collaborators
- DTOs for Data Transfer: DTOs provide a clean interface between layers
- Service Abstraction: Controllers don't access repositories directly

7. Composition Over Inheritance

The system favors composition over inheritance:

- Service Composition: Services use other services through composition
- Feature Modules: Functionality is composed of smaller, focused components

8. Programming to Interfaces

The system implements programming to interfaces:

- Repository Interfaces: Data access is defined through interfaces
- Service Interfaces: Business logic is accessed through interfaces
- Dependency Injection: Components depend on interfaces, not implementations

9. Model-View-Controller (MVC)

As previously discussed, the system follows the MVC architectural pattern:

- Model: Entity classes, repositories, and services
- View: Thymeleaf templates
- Controller: Spring MVC controllers

10. Defensive Programming

The system employs defensive programming techniques:

- Input Validation: User inputs are validated before processing
- Error Handling: Exceptions are properly caught and handled
- Null Checks: Potential null values are checked before use

11. Fail-Fast Principle

The system implements the fail-fast principle:

- Early Validation: Inputs are validated as early as possible
- Immediate Error Reporting: Errors are reported immediately when detected
- Constraint Enforcement: Database constraints enforce data integrity

12. Principle of Least Surprise

The system follows predictable patterns:

- Consistent Naming: Similar operations have similar names
- Standard Workflows: Common operations follow standard patterns
- Familiar Interfaces: User interfaces follow common web patterns

13. Secure by Design

The system incorporates security principles:

- Authentication: Spring Security for user authentication
- Authorization: Role-based access control
- Input Sanitization: User inputs are sanitized to prevent attacks
- CSRF Protection: Cross-Site Request Forgery protection

14. Persistence Ignorance

The domain model is designed to be persistence-ignorant:

- Clean Domain Model: Entity classes focus on business concepts
- JPA Annotations: Persistence details are added through annotations
- Repository Pattern: Data access is abstracted through repositories

15. Convention over Configuration

The system leverages Spring Boot's convention over configuration:

- Standard Project Structure: Following Spring Boot conventions
- Default Configurations: Using sensible defaults
- Annotation-Based Configuration: Using annotations instead of XML

Conclusion

The HelpDesk Management System demonstrates a comprehensive application of modern software design principles. By adhering to these principles, the system achieves:

1. Maintainability: Code is organized, focused, and follows clear patterns
2. Extensibility: New features can be added with minimal changes to existing code
3. Testability: Components can be tested in isolation
4. Scalability: The system can grow without significant architectural changes
5. Security: Security concerns are addressed throughout the design
6. Usability: The system follows intuitive patterns for users These design principles work together to create a robust, flexible, and maintainable system that meets the needs of both users and administrators while providing a solid foundation for future enhancements.

Design Patterns in the HelpDesk Management System

1. Repository Pattern

The Repository pattern is extensively used to abstract the data access layer from the rest of the application.

Implementation:

- Each entity has a corresponding repository interface that extends JpaRepository
- Examples include UserRepository, PostRepository, CommentRepository, etc.

Benefits:

- Centralizes data access logic
- Provides a clean API for domain objects
- Facilitates unit testing through mock repositories
- Decouples business logic from data access details

2. Dependency Injection Pattern

Spring's dependency injection is used throughout the application to manage component dependencies.

Implementation:

- Constructor injection in controllers and services
- Autowired dependencies

Benefits:

- Loose coupling between components
- Easier unit testing through dependency mocking
- Centralized configuration of components
- Promotes the use of interfaces over concrete implementations

3. Data Transfer Object (DTO) Pattern

DTOs are used to transfer data between subsystems of the application.

Implementation:

- DTO classes for various entities (UserDto, PostDto, CommentDto, etc.)

- Lombok annotations (@Data, @NoArgsConstructor, @AllArgsConstructor) to reduce boilerplate

Benefits:

- Decouples the presentation layer from the domain model
- Allows for different data representations for different use cases
- Prevents exposing sensitive domain information
- Facilitates API versioning and evolution

4. Service Layer Pattern

The Service Layer pattern is used to encapsulate business logic.

Implementation:

- Service interfaces and their implementations
- Business logic contained within service classes

Benefits:

- Centralizes business logic
- Provides a clear API for controllers
- Facilitates transaction management
- Promotes separation of concerns

5. Front Controller Pattern

Spring MVC implements the Front Controller pattern through DispatcherServlet.

Implementation:

- Controller classes with request mapping annotations
- Centralized request handling

Benefits:

- Centralizes request handling logic
- Provides consistent handling of cross-cutting concerns
- Simplifies security implementation
- Facilitates view resolution

6. Model-View-Controller (MVC) Pattern

The entire application follows the MVC architectural pattern.

Implementation:

- Models: Entity classes and DTOs
- Views: Thymeleaf templates
- Controllers: Spring MVC controllers

Benefits:

- Separation of concerns
- Parallel development of components
- Improved maintainability
- Enhanced testability

7. Template Method Pattern

The Template Method pattern is used in service implementations for operations that follow a similar sequence but differ in specific steps.

Implementation:

- Base methods that define the algorithm structure
- Specialized methods that implement specific steps

Benefits:

- Reuse of common algorithm structure
- Customization of specific steps
- Reduced code duplication
- Consistent behavior across similar operations

8. Observer Pattern

Spring's event system implements the Observer pattern for loosely coupled event handling.

Implementation:

- Event publishers
- Event listeners
- Application events

Benefits:

- Loose coupling between components
- Simplified event-driven programming
- Scalable event handling
- Separation of concerns

9. Factory Method Pattern

Factory methods are used to create objects, particularly in service implementations.

Implementation:

- Methods that create and return objects
- Encapsulated object creation logic

Benefits:

- Encapsulation of object creation logic
- Centralized object creation
- Flexibility in object creation process
- Consistent object initialization

10. Builder Pattern

Lombok's @Builder annotation implements the Builder pattern for complex object construction.

Implementation:

- @Builder annotation on entity and DTO classes
- Fluent API for object construction

Benefits:

- Simplified construction of complex objects
- Improved readability of object creation code
- Optional parameters handling
- Immutable object creation

11. Singleton Pattern

Spring beans are singletons by default, implementing the Singleton pattern.

Implementation:

- Spring-managed beans with singleton scope
- Single instance per application context

Benefits:

- Controlled access to shared resources
- Reduced memory footprint
- Consistent state across the application
- Centralized configuration

12. Composite Pattern

The comment system implements a form of the Composite pattern for handling comment hierarchies.

Implementation:

- Comments can contain child comments (replies)
- Recursive processing of comment hierarchies

Benefits:

- Unified treatment of individual and composite objects
- Simplified client code
- Natural representation of hierarchical structures
- Extensibility for new types of components

13. Strategy Pattern

Different strategies for handling various types of content (posts, comments) implement the Strategy pattern.

Implementation:

- Different service implementations for different content types
- Common interfaces for similar operations

Benefits:

- Runtime selection of algorithms
- Encapsulation of algorithm details
- Simplified client code
- Extensibility for new strategies

14. Decorator Pattern

Spring Security filters implement the Decorator pattern to add security functionality to HTTP requests.

Implementation:

- Security filters that wrap HTTP requests
- Additional functionality added to base behavior

Benefits:

- Dynamic addition of responsibilities
- Flexible alternative to subclassing
- Composable behavior modifications
- Separation of concerns

15. Proxy Pattern

Spring's transaction management and security use the Proxy pattern to add cross-cutting concerns.

Implementation:

- Transaction proxies for service methods
- Security proxies for protected resources

Benefits:

- Transparent addition of functionality
- Separation of cross-cutting concerns
- Controlled access to resources
- Simplified client code

Conclusion

The HelpDesk Management System effectively implements numerous design patterns that contribute to its maintainability, flexibility, and adherence to best practices. These patterns work together to create a robust architecture that:

1. **Separates concerns**: Each component has a clear responsibility
2. **Promotes reusability**: Common patterns are applied consistently
3. **Enhances testability**: Components can be tested in isolation
4. **Facilitates maintenance**: Changes can be made with minimal impact
5. **Supports extensibility**: New features can be added without major restructuring

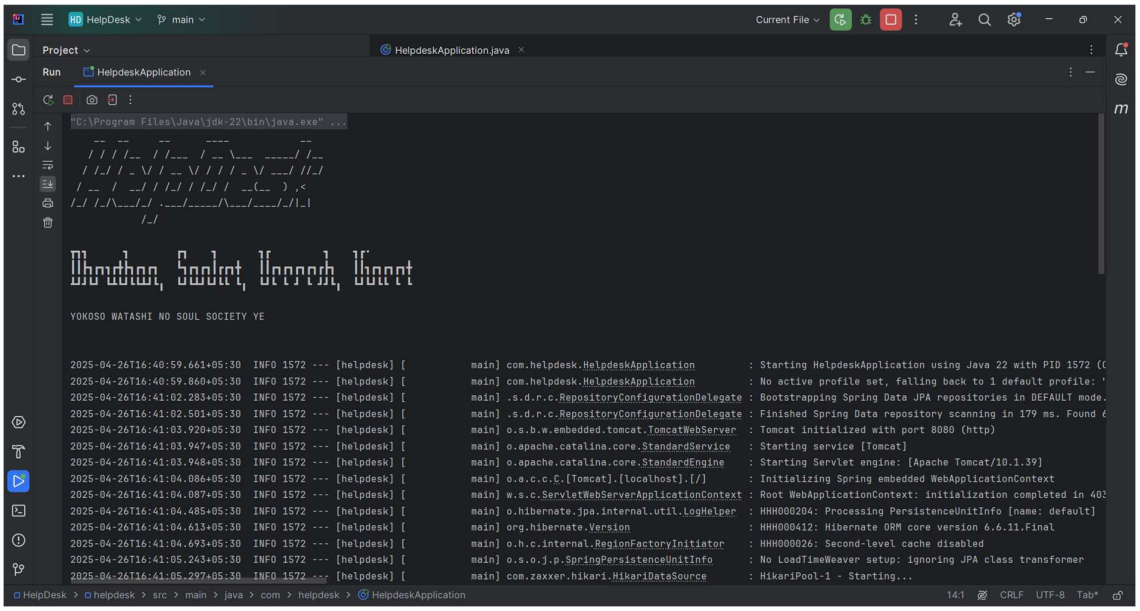
The consistent application of these design patterns demonstrates a well-architected system that follows industry best practices for enterprise application development.

GitHub Repo Link

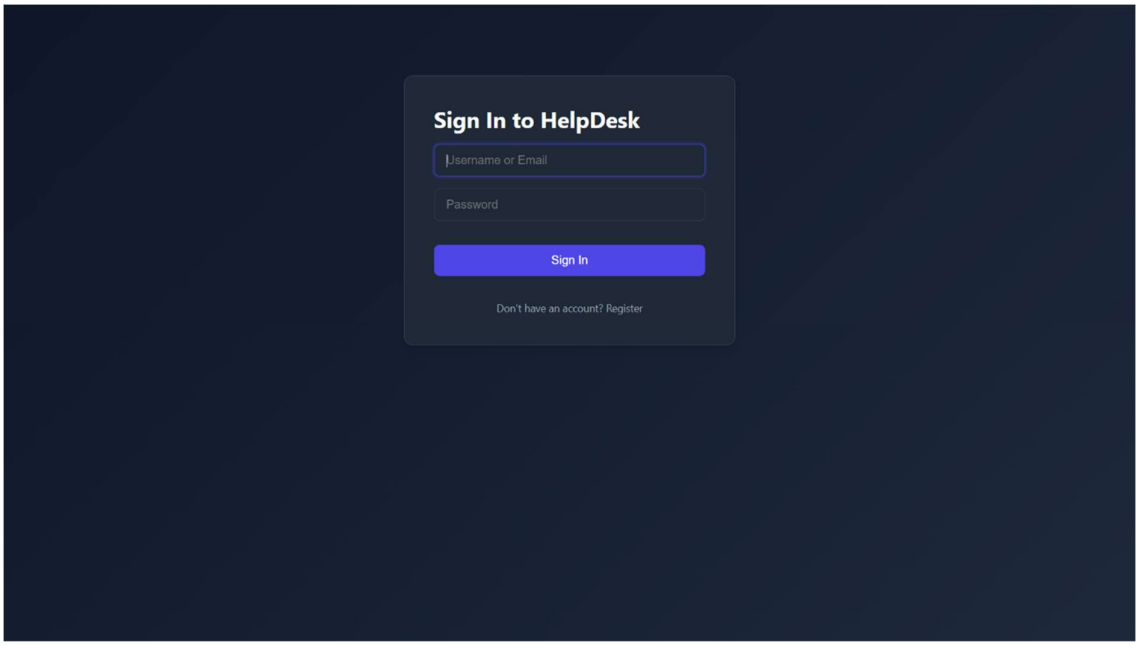
GitHub *link* for HelpDesk Management System

Screenshots

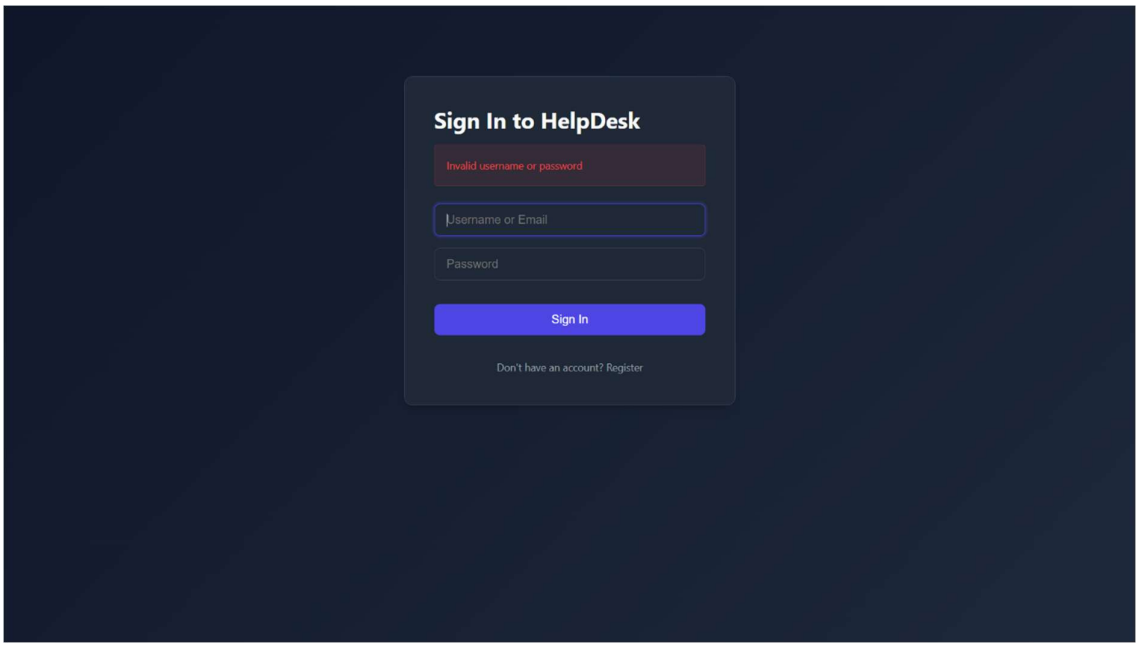
Starting the Application:



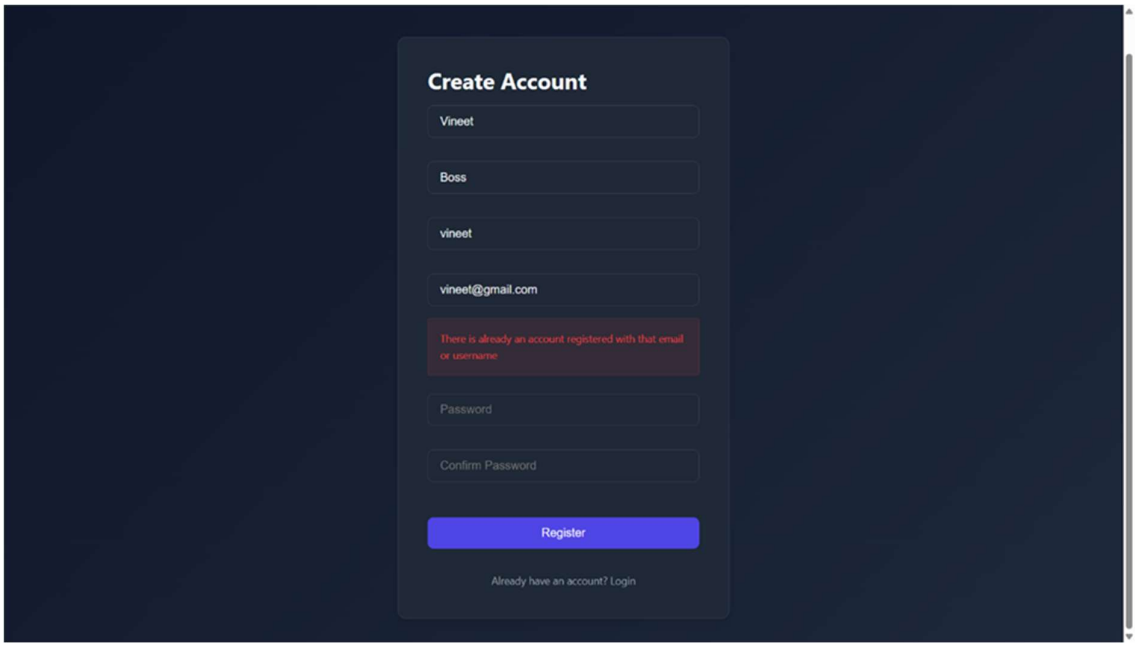
Sign In



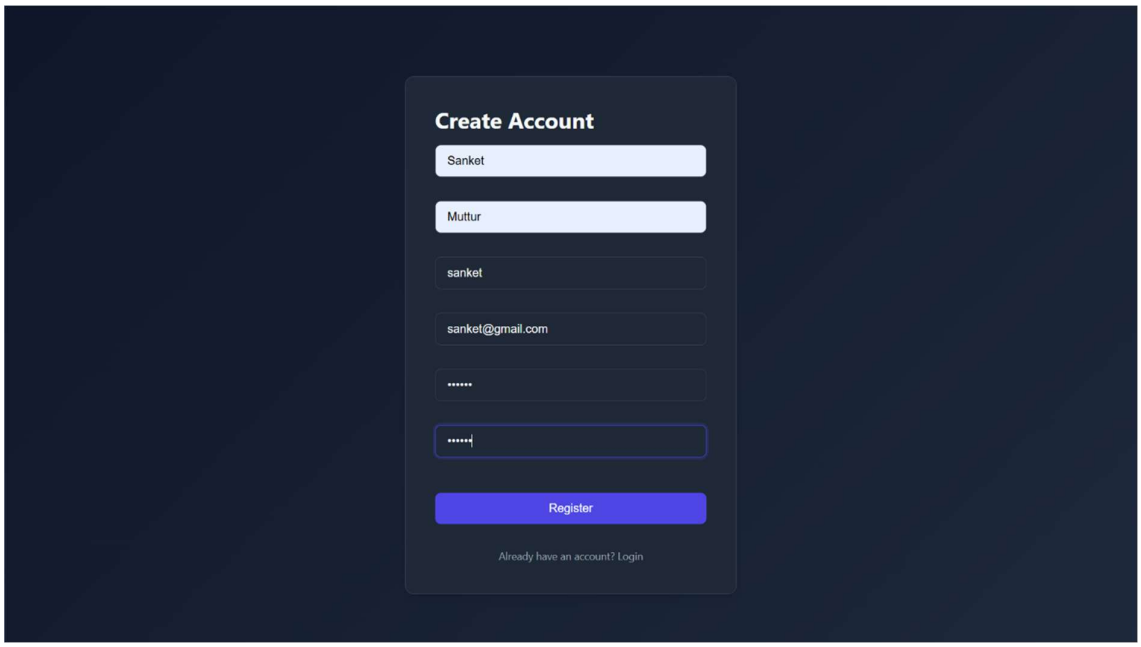
Authenticaion



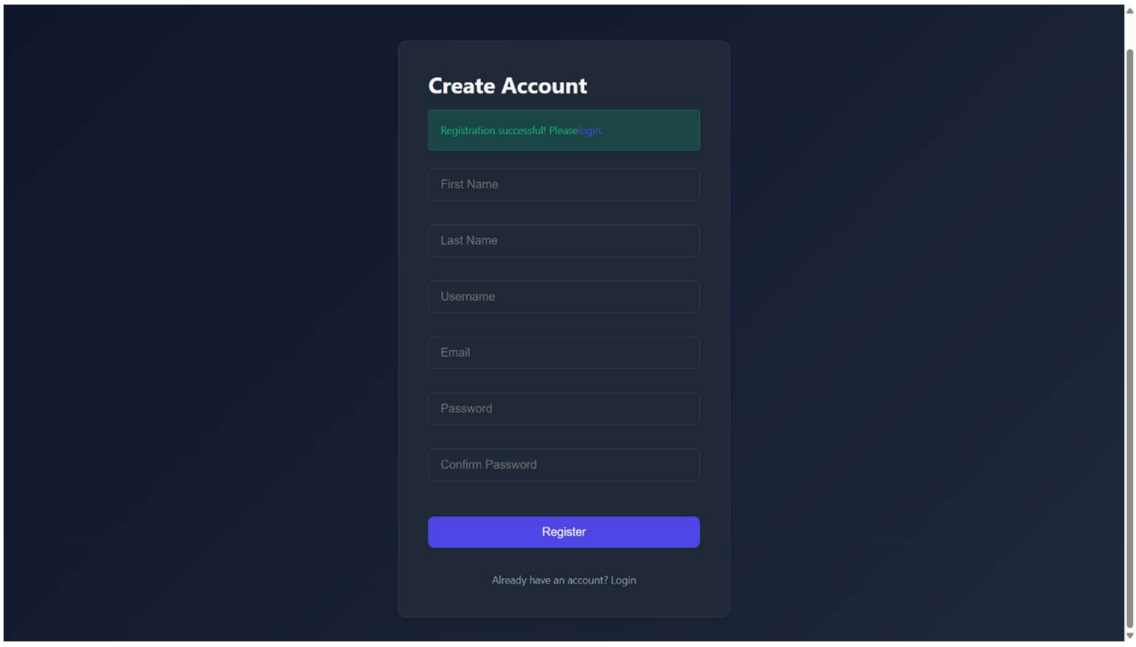
Invalid Registration



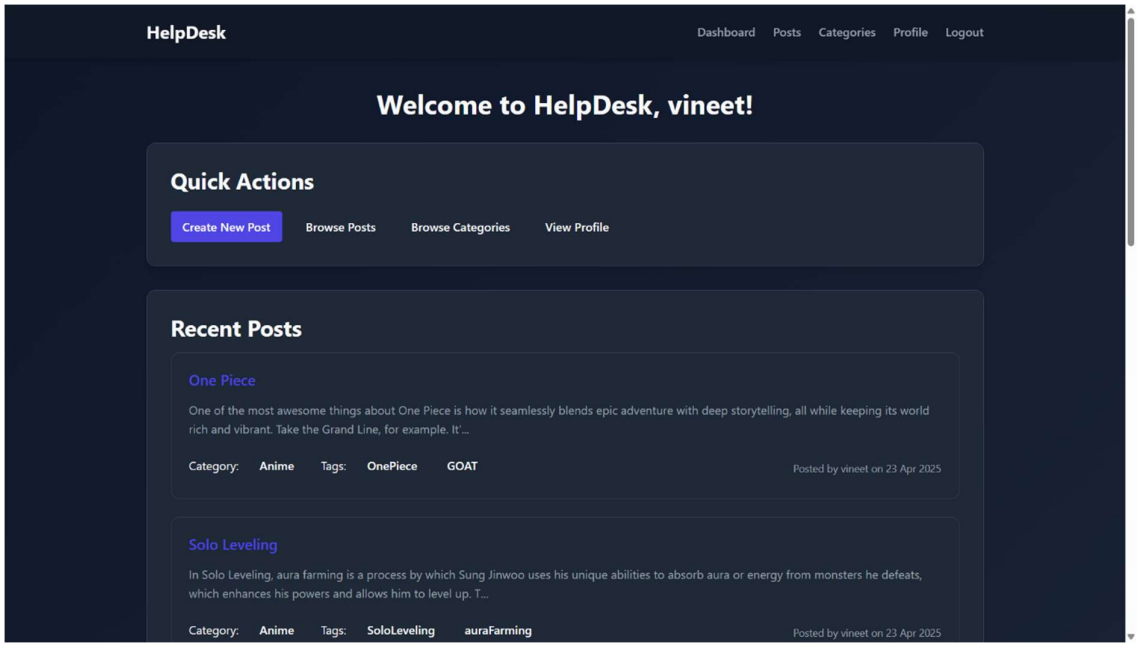
Registering



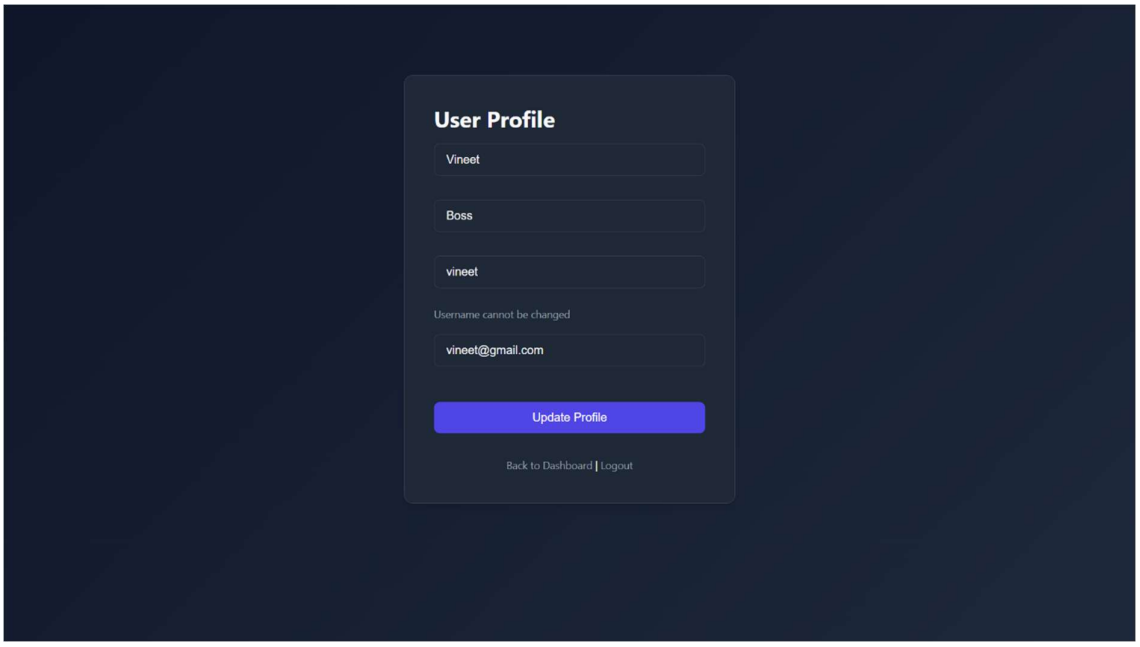
Successful Registration



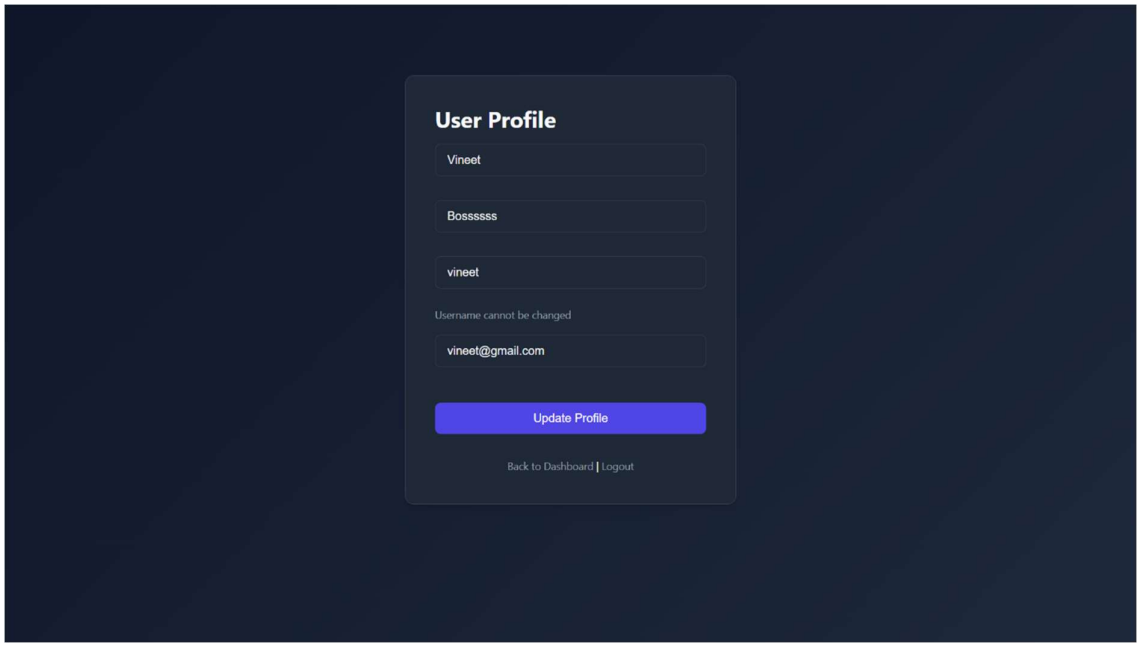
User Dashboard after Sign In



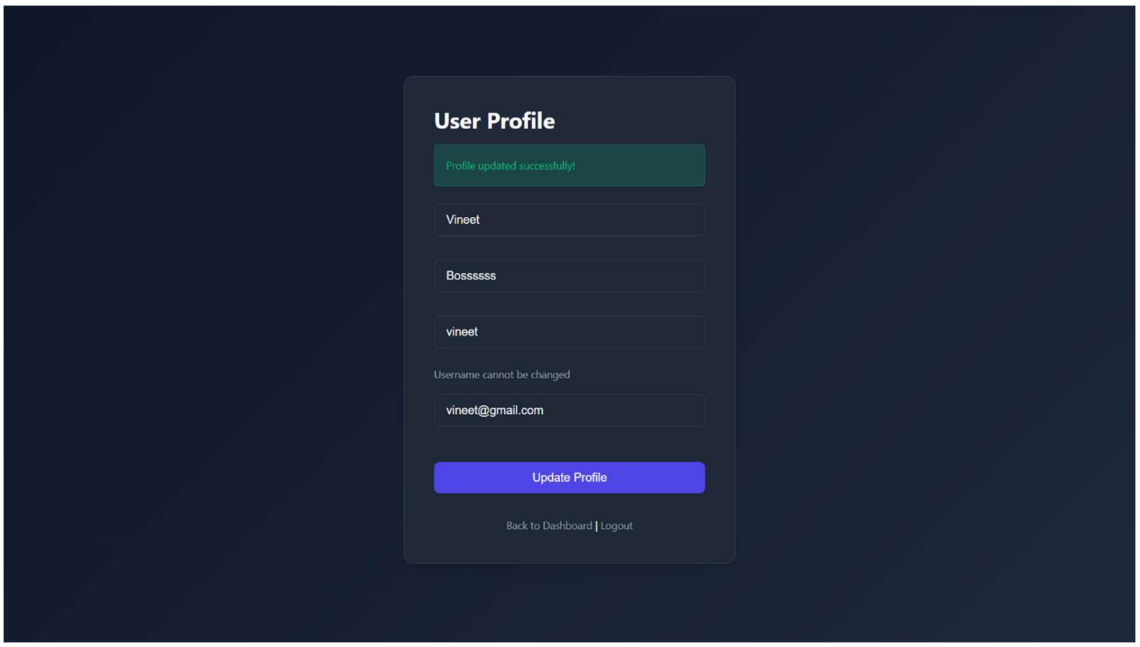
User Profile



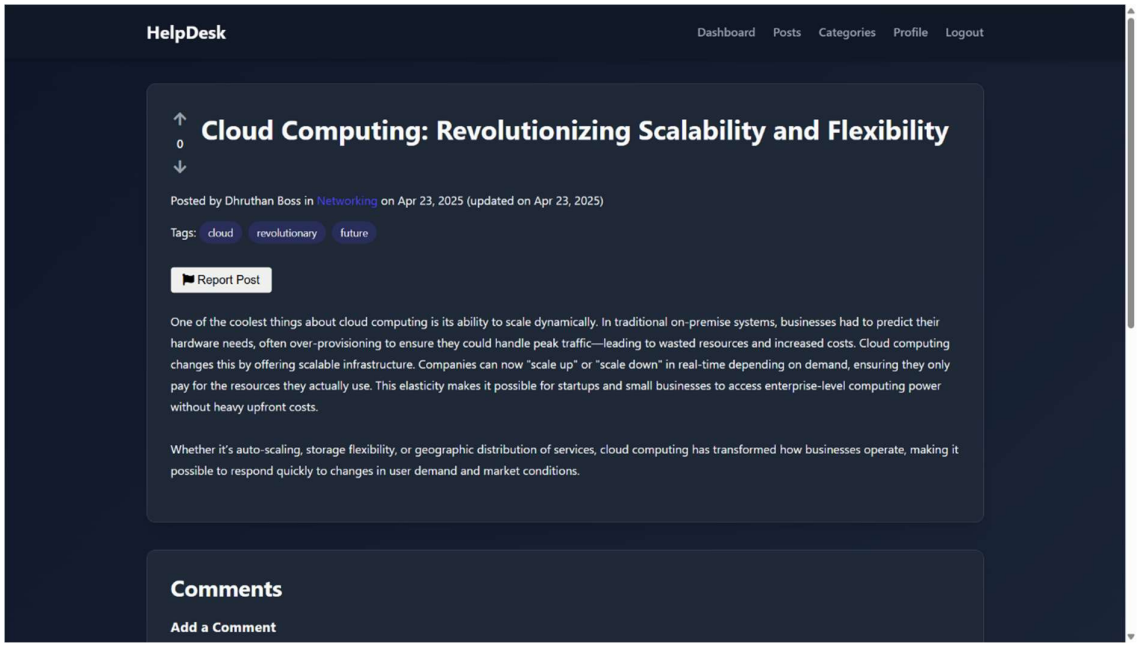
Updating User Profile



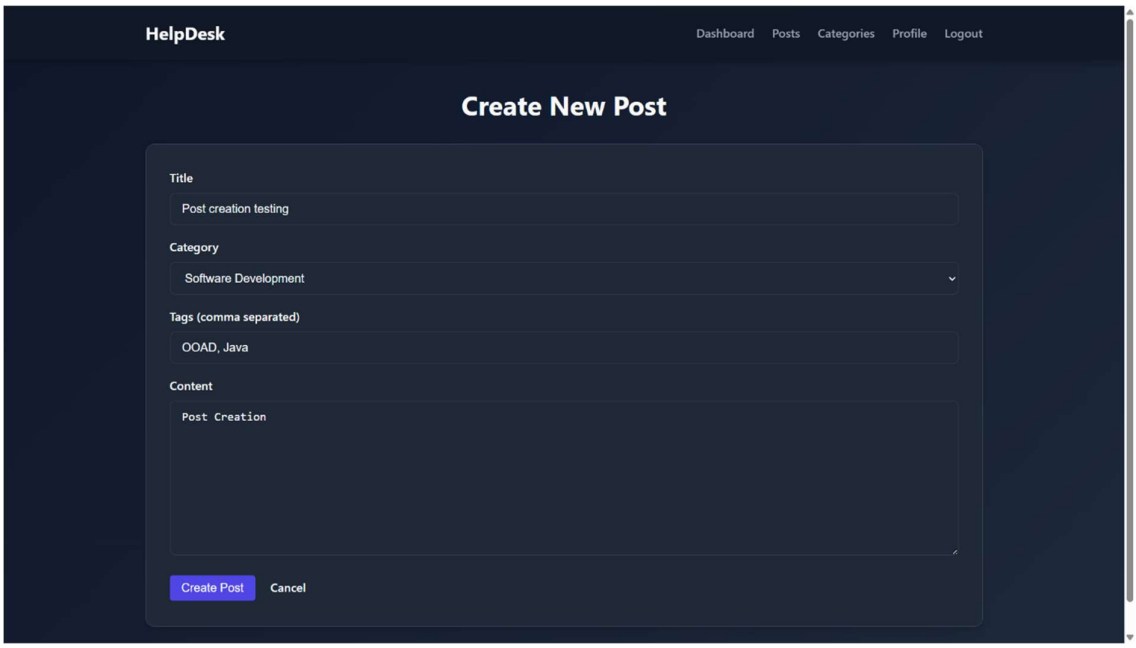
User Profile Update Successful



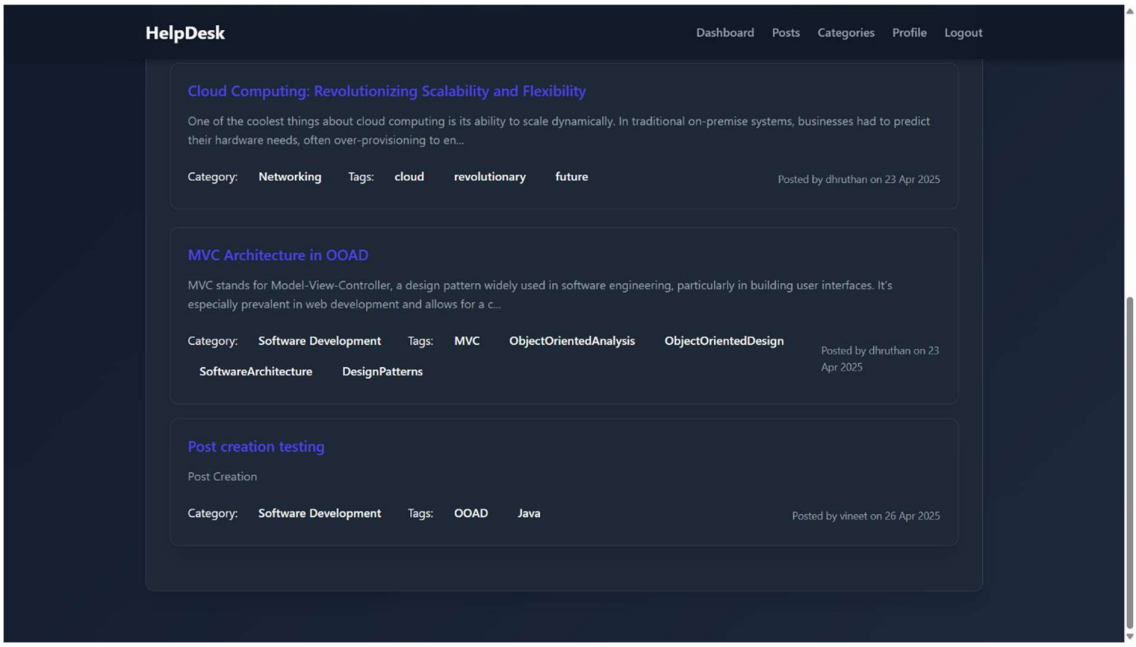
Viewing Posts



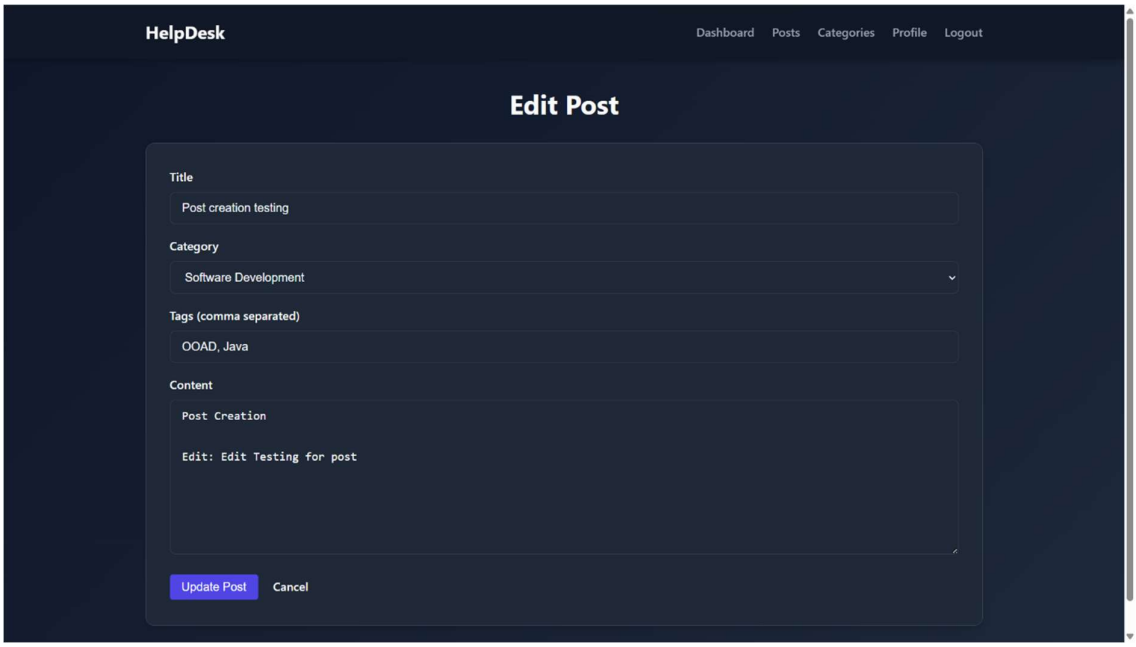
Post Creation



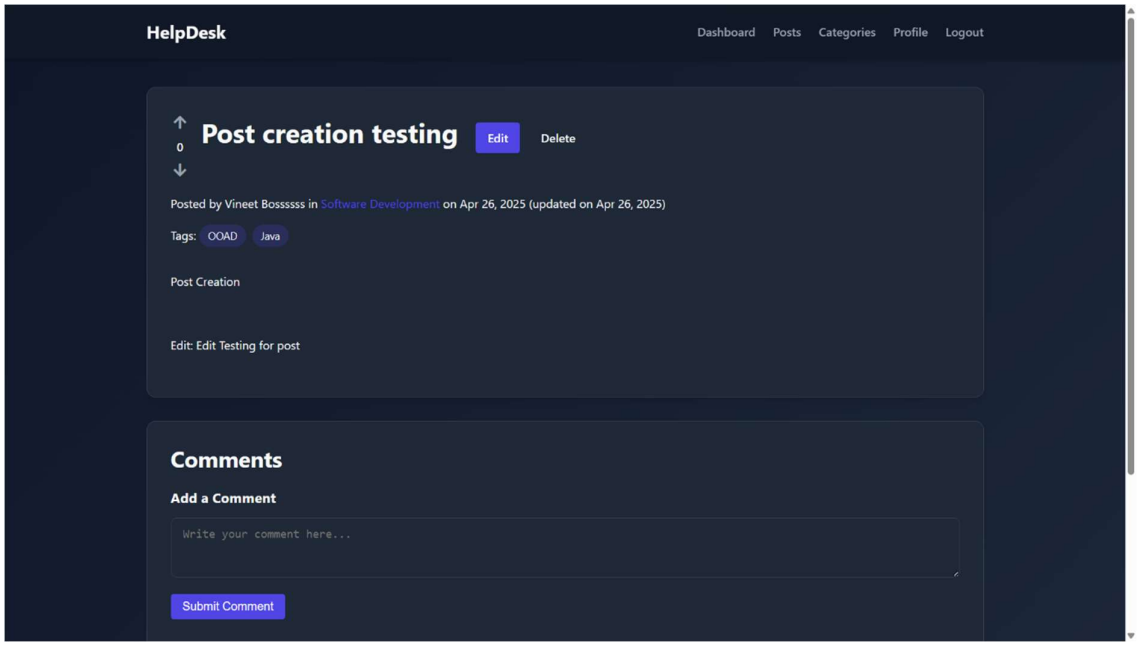
Post Created Successfully



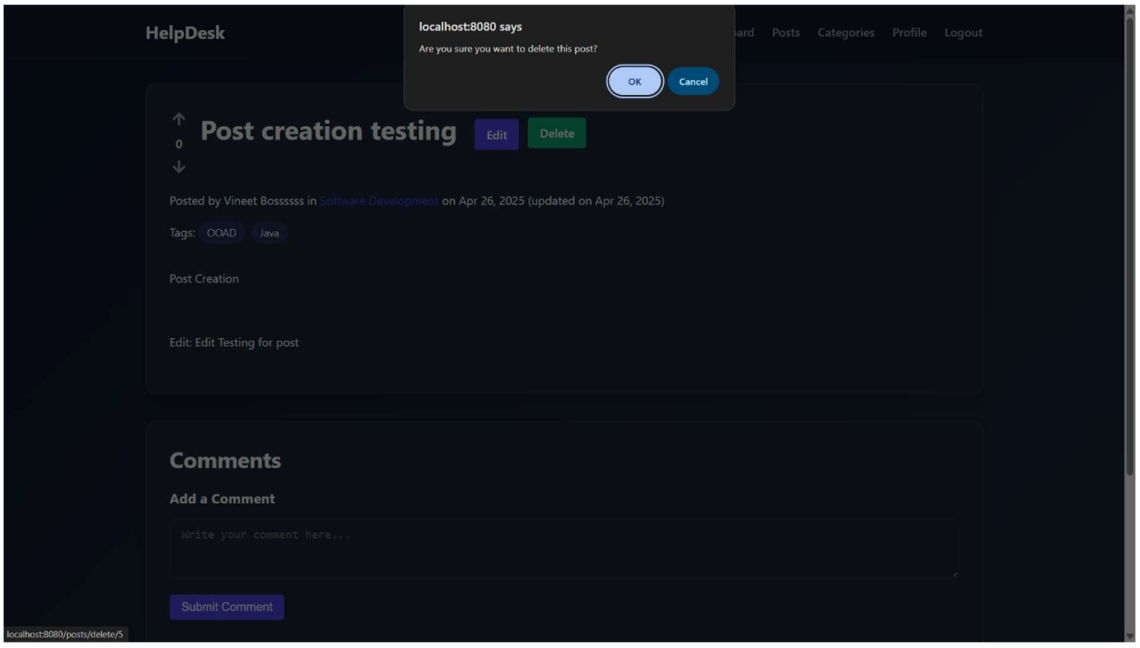
Editing the Post



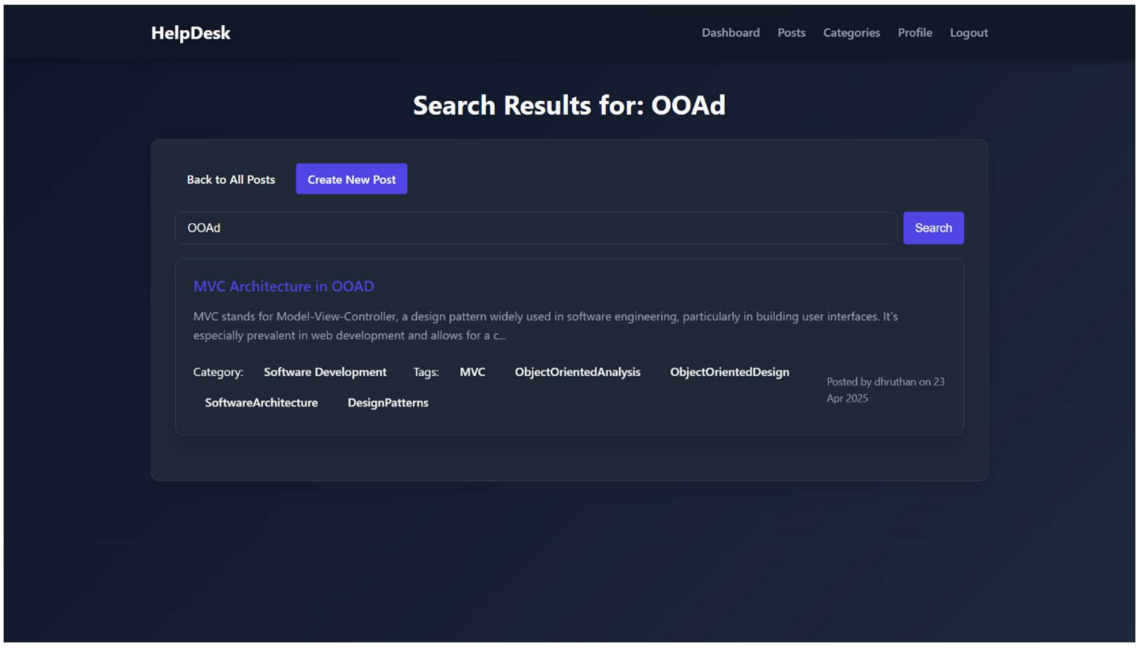
Successfully edited the post



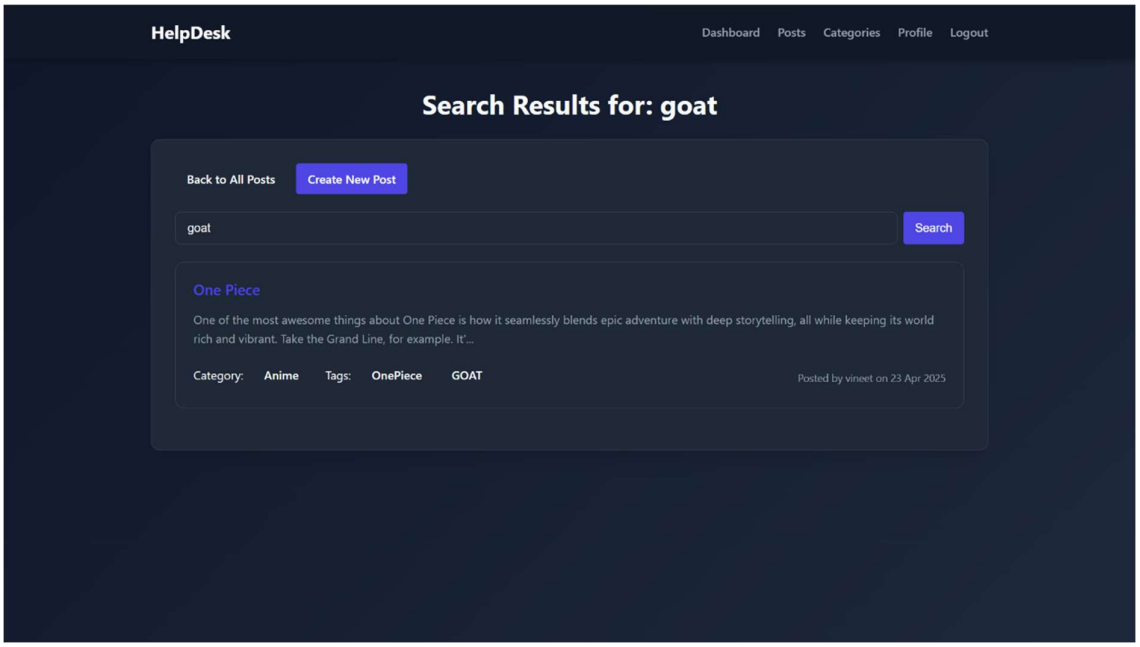
Deleting the Post



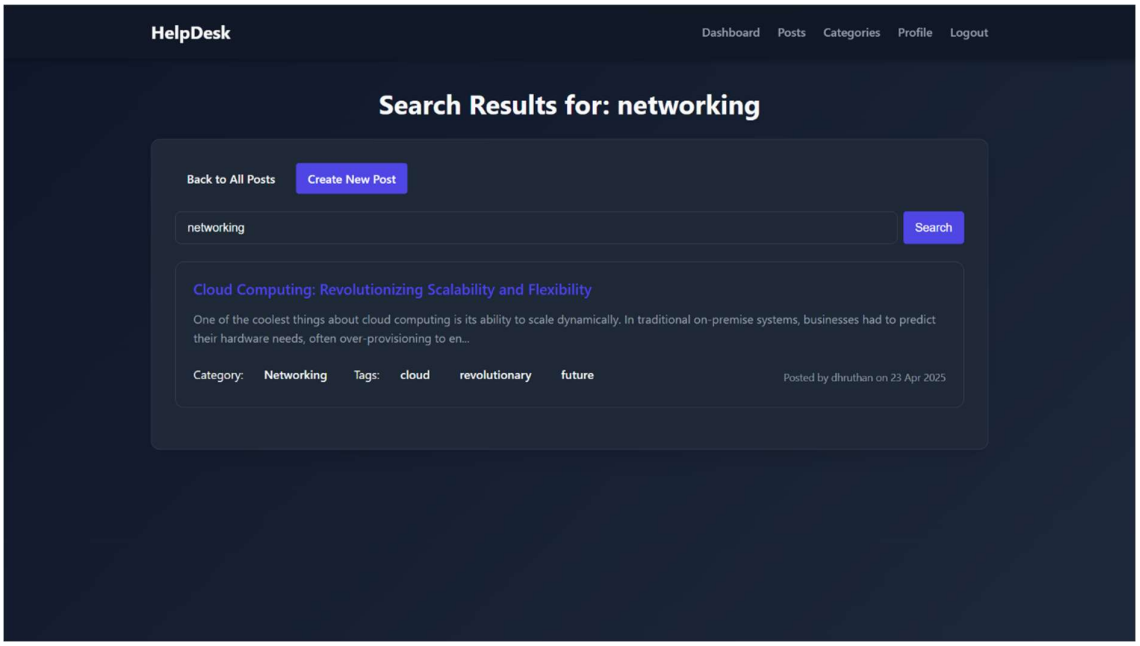
Search Post by Title



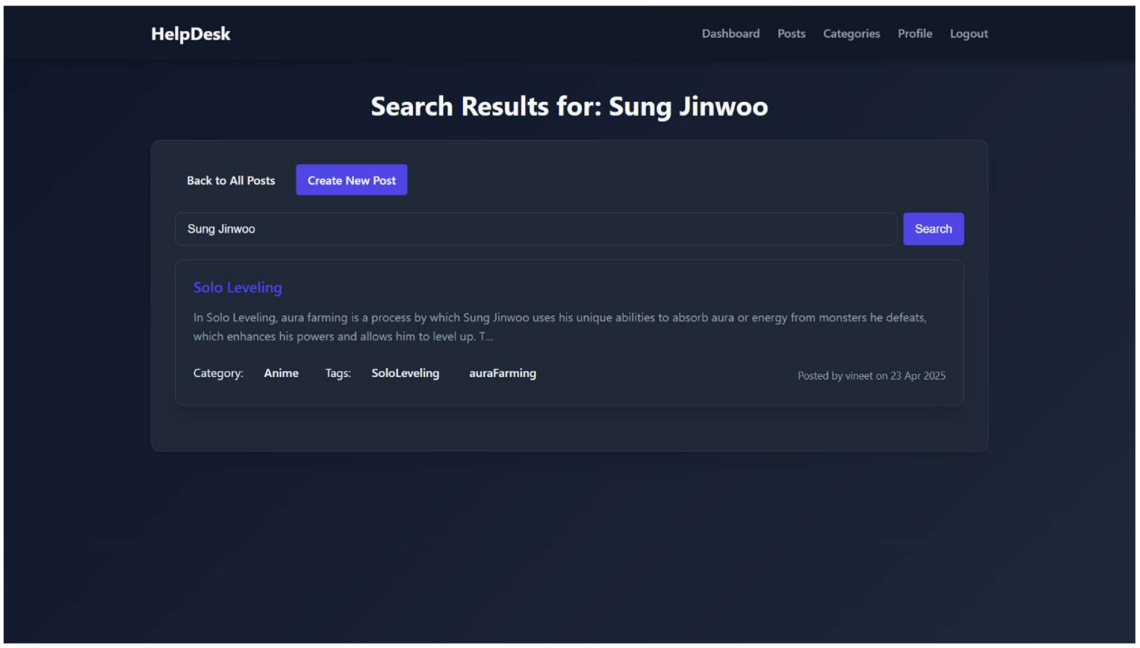
Search Post by Tag



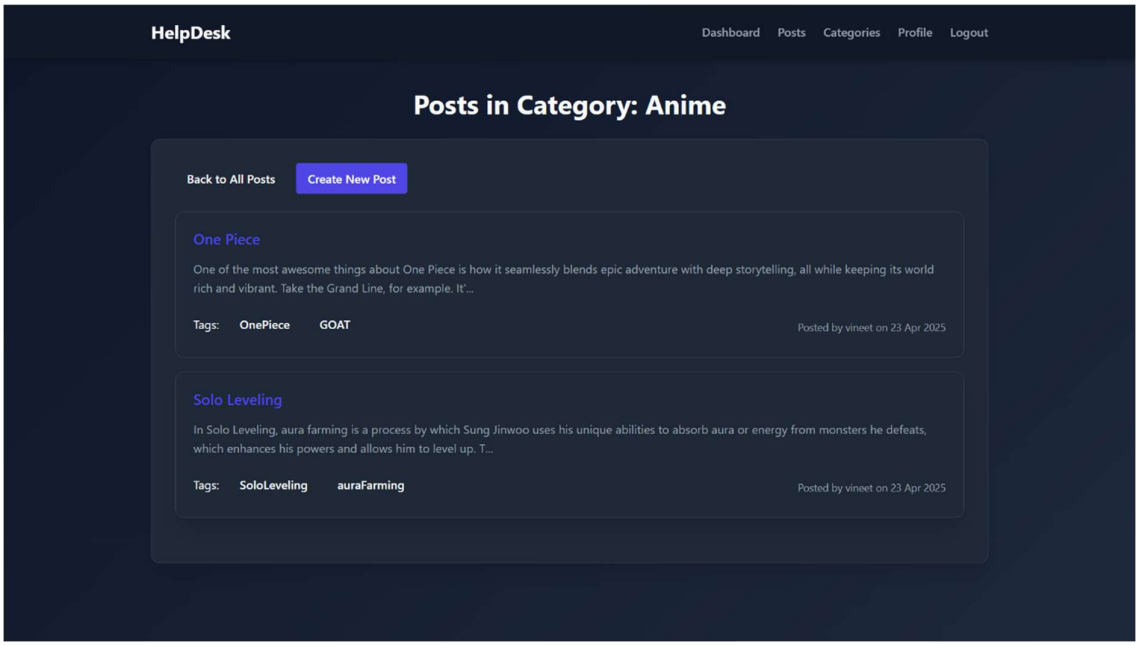
Search Post by Category



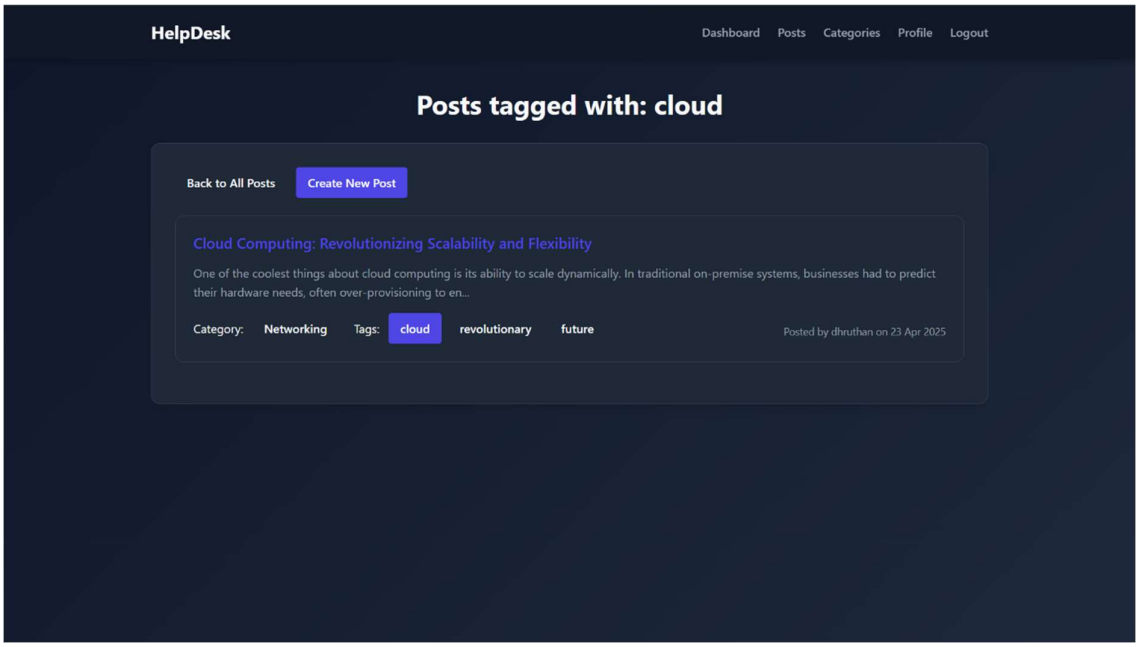
Search Post by Content



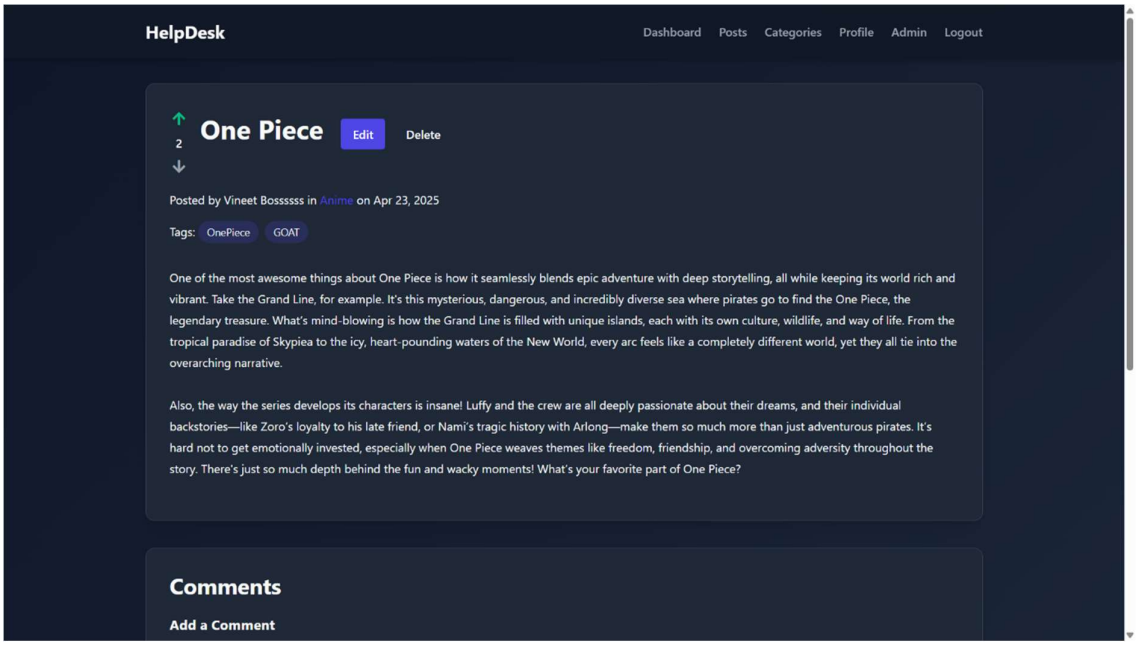
View Post by Category



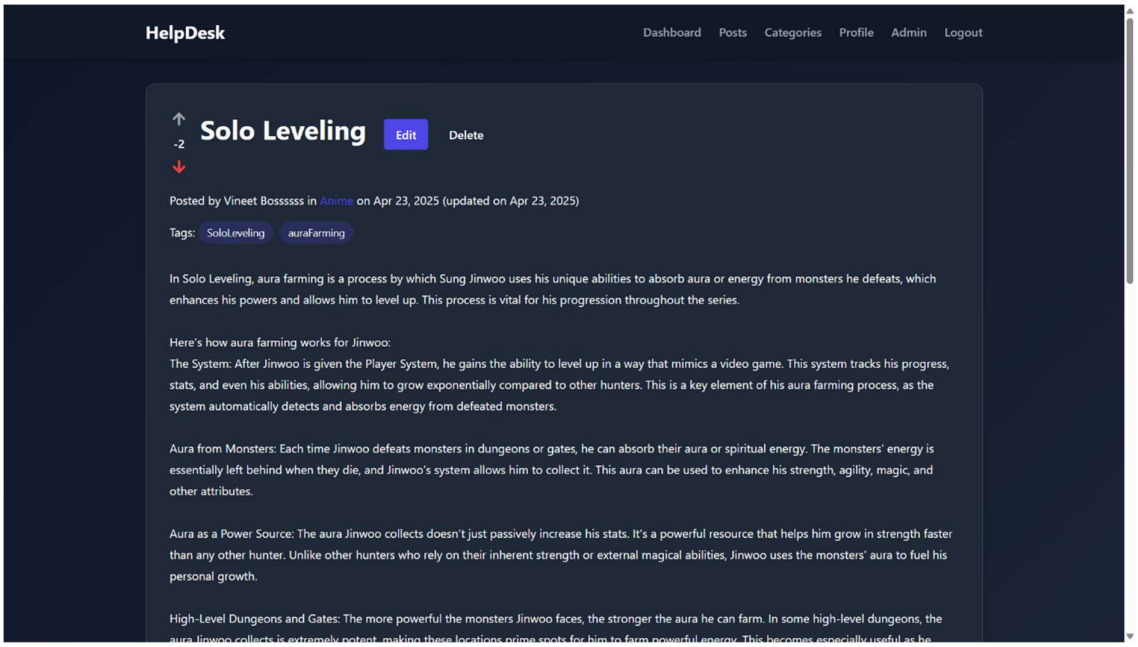
View Post by Tag



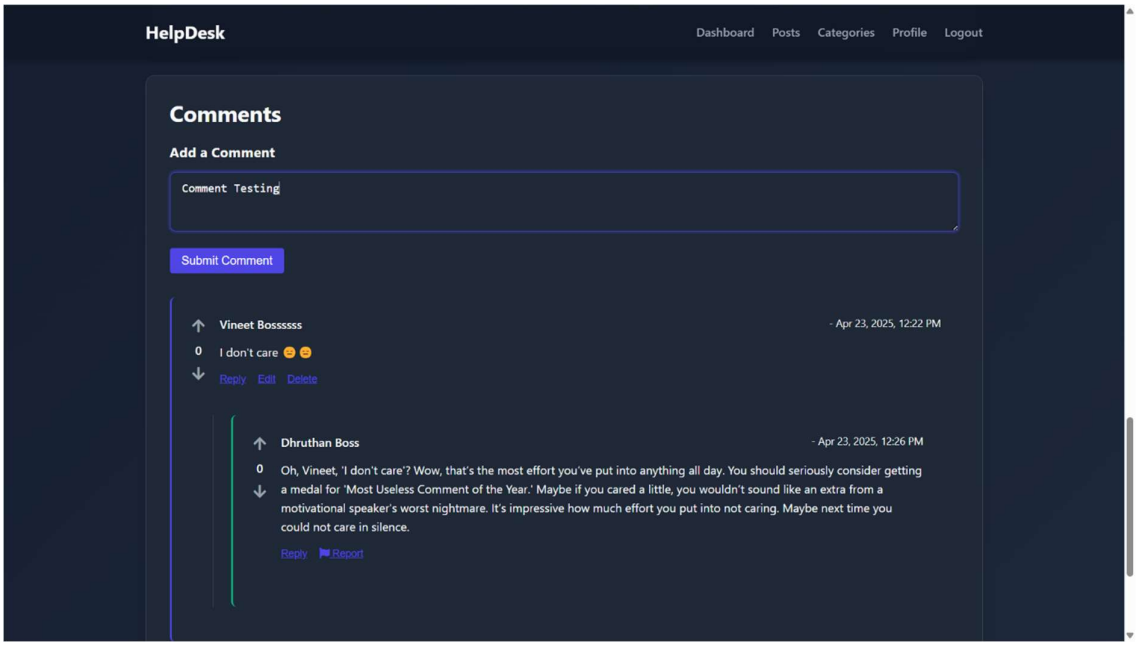
Upvoting the Post/Comment



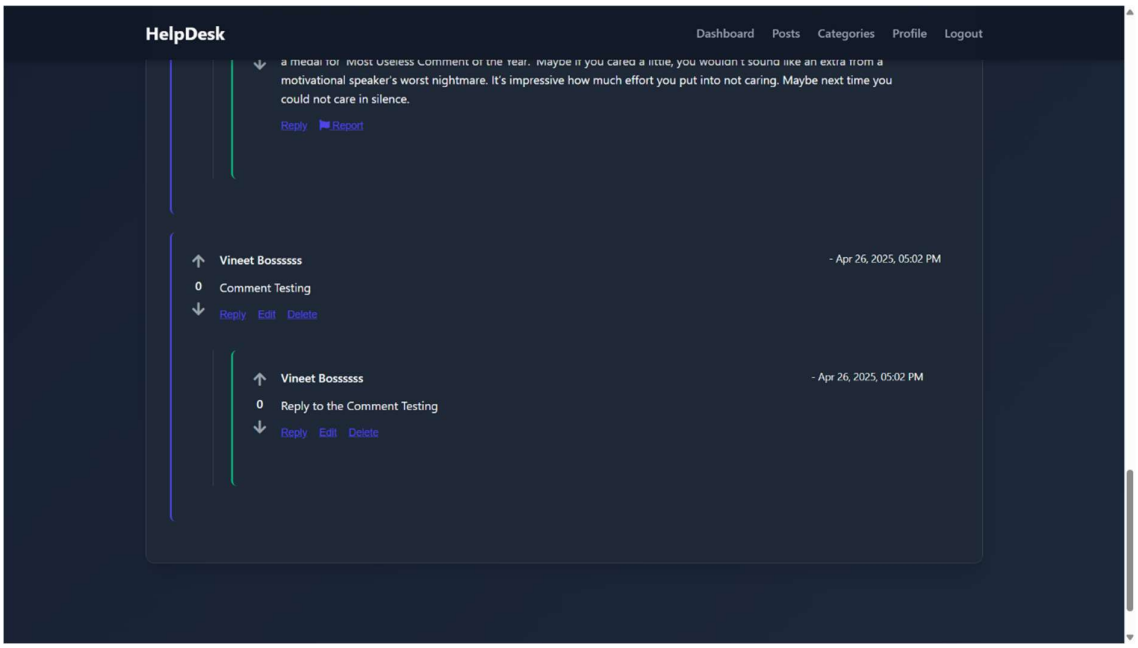
Downvoting the Post/Comment



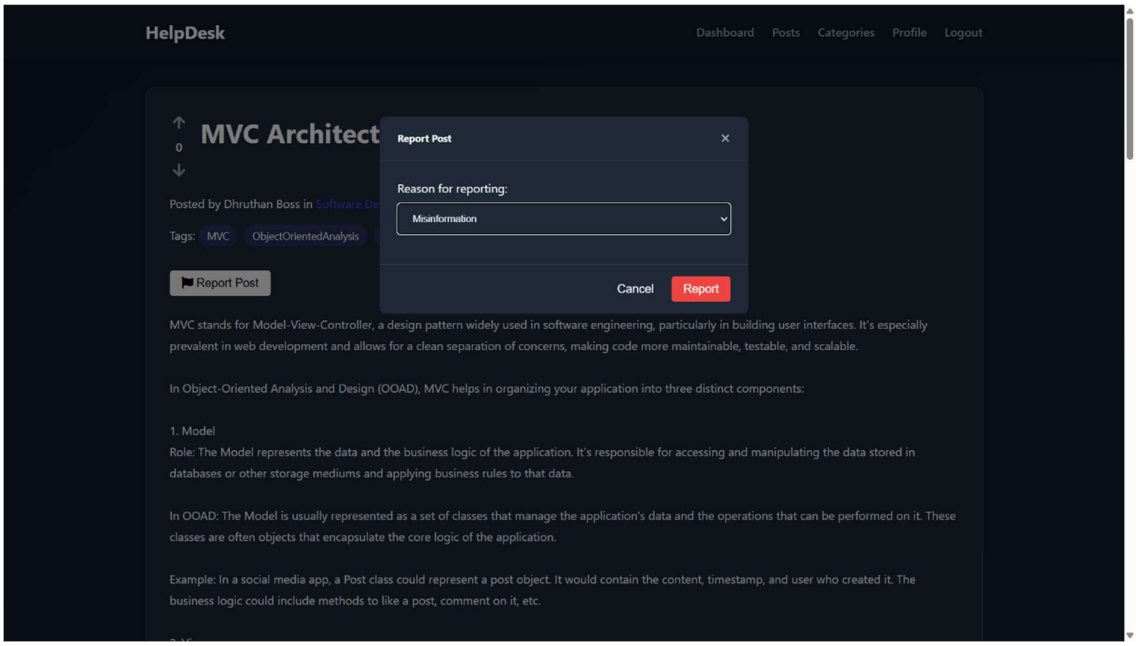
Commenting



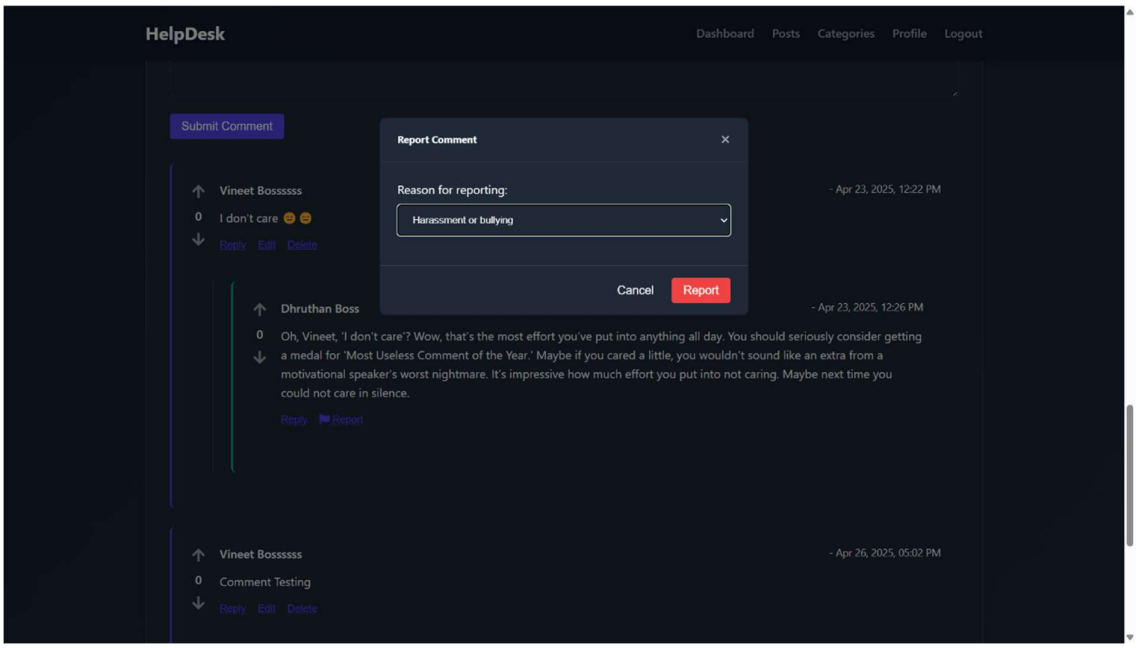
Replying to the Comment



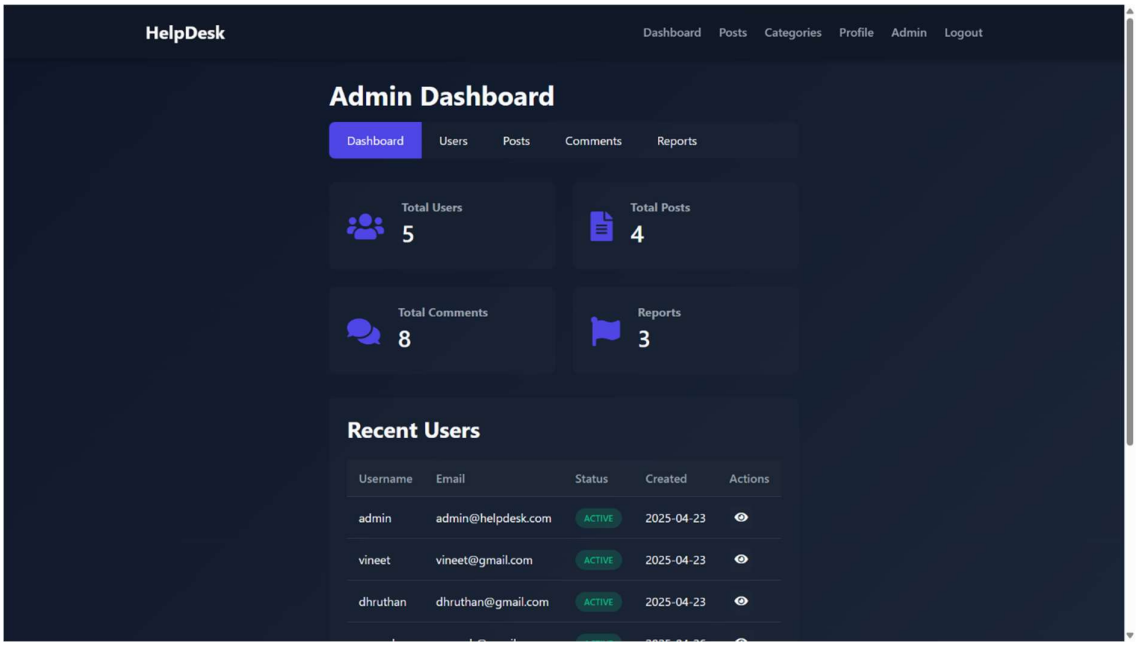
Reporting Post



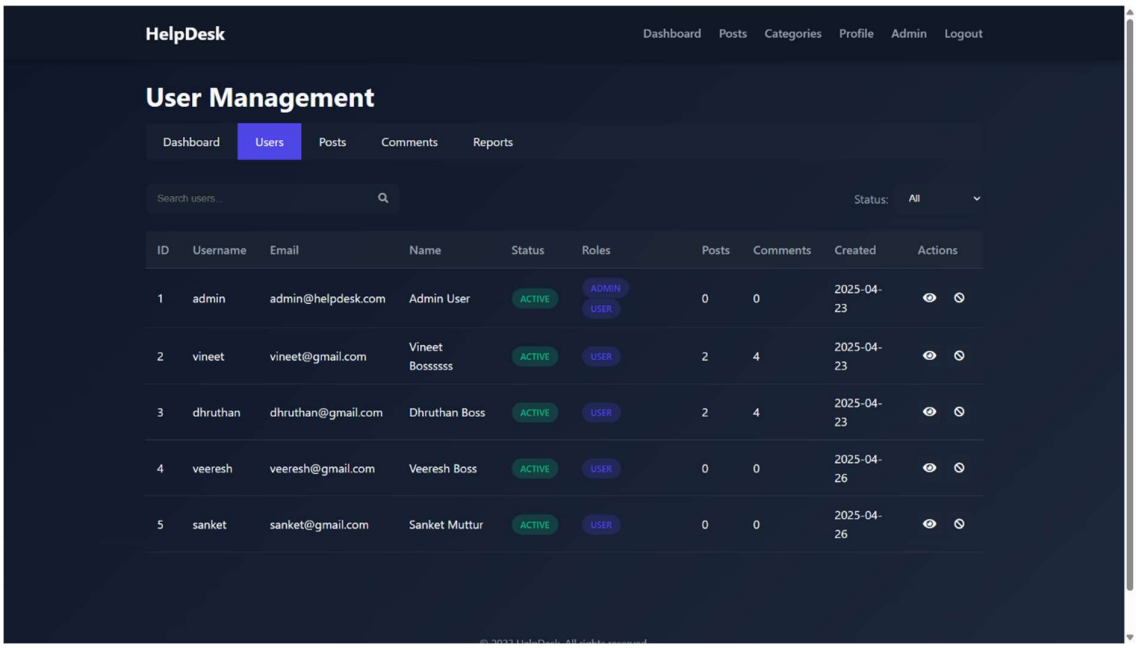
Reporting Comment



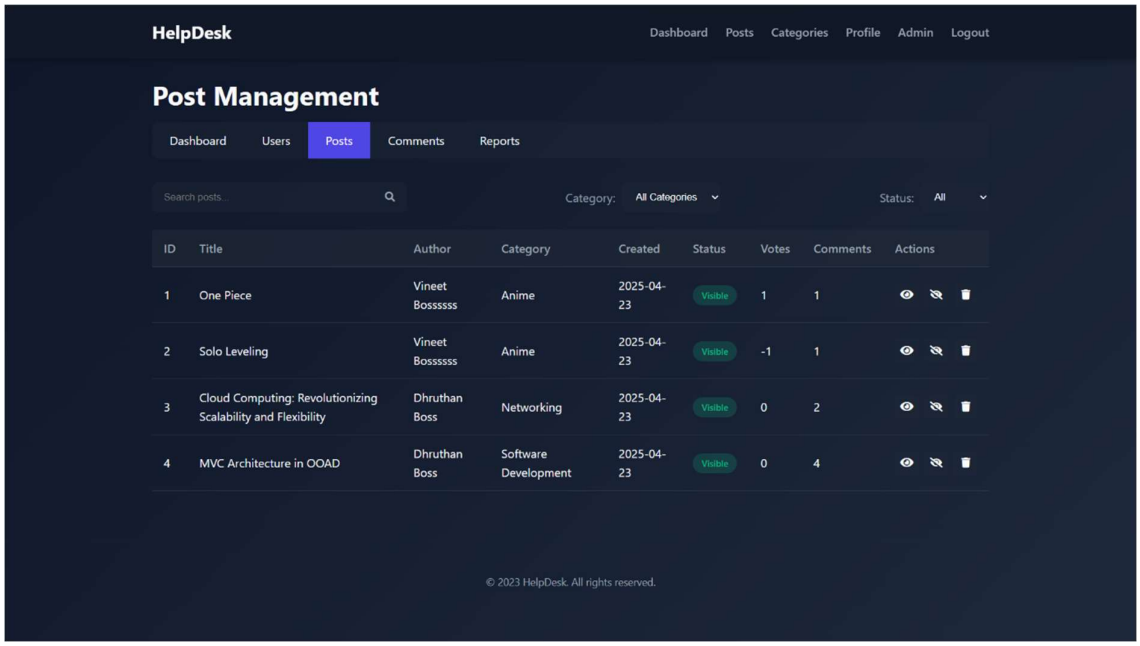
Admin Dashboard



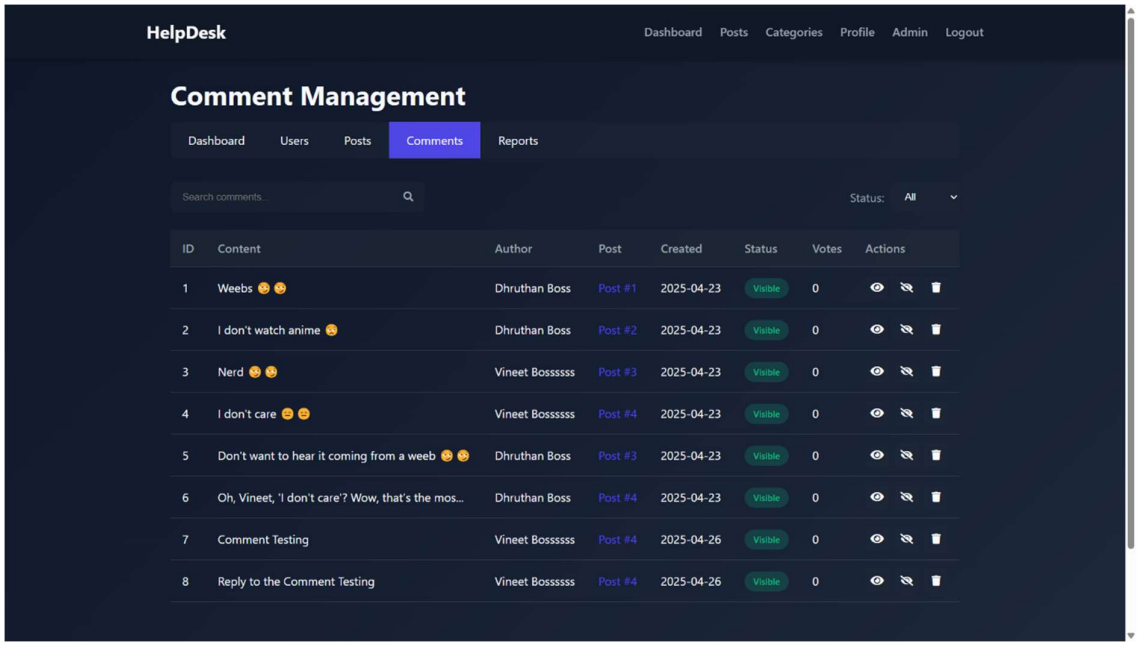
User Management (Can View individual User profile and Suspend the User)



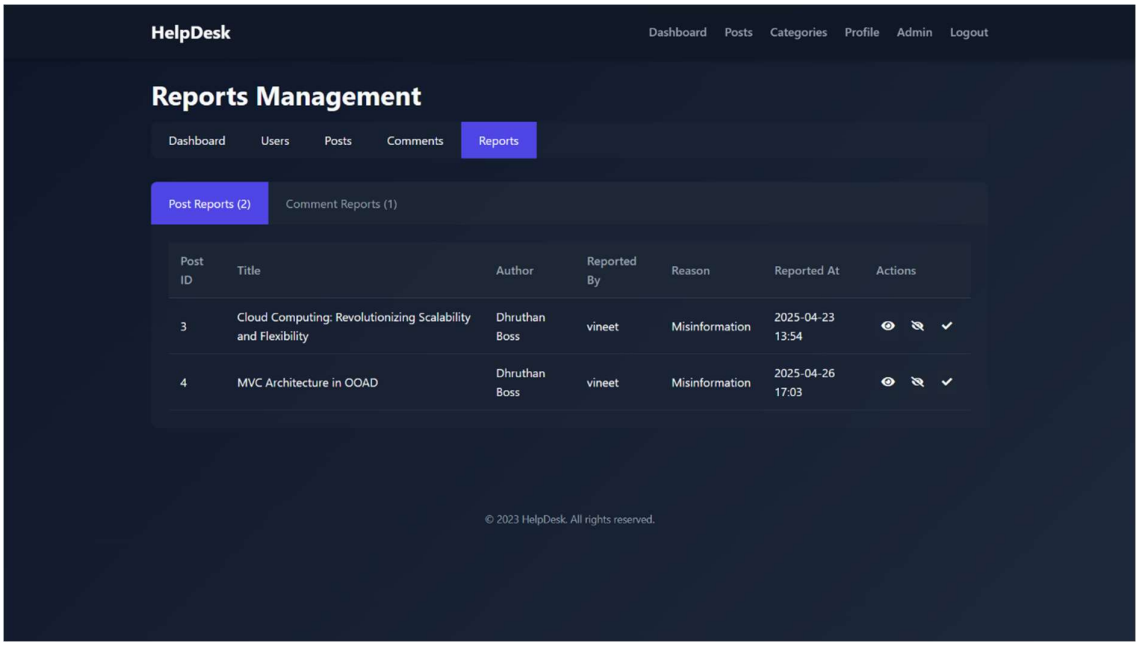
Post Management (Can View/Hide/Unhide/Delete the particular Post)



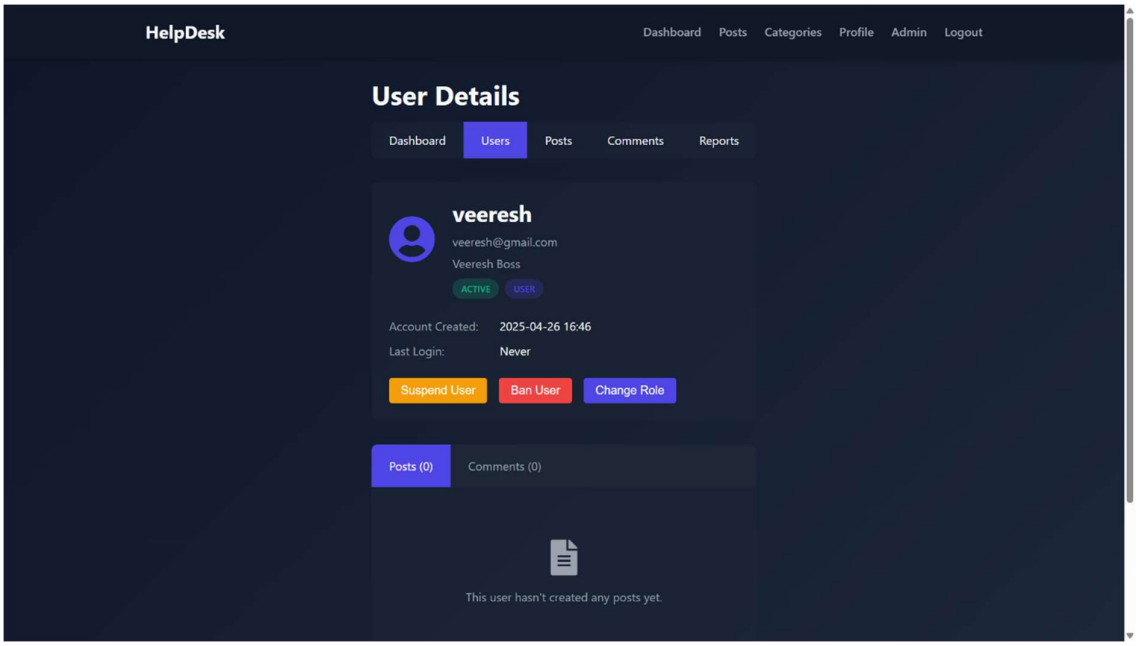
Comment Management (Can View/Hide/Unhide/Delete the particular Comment)



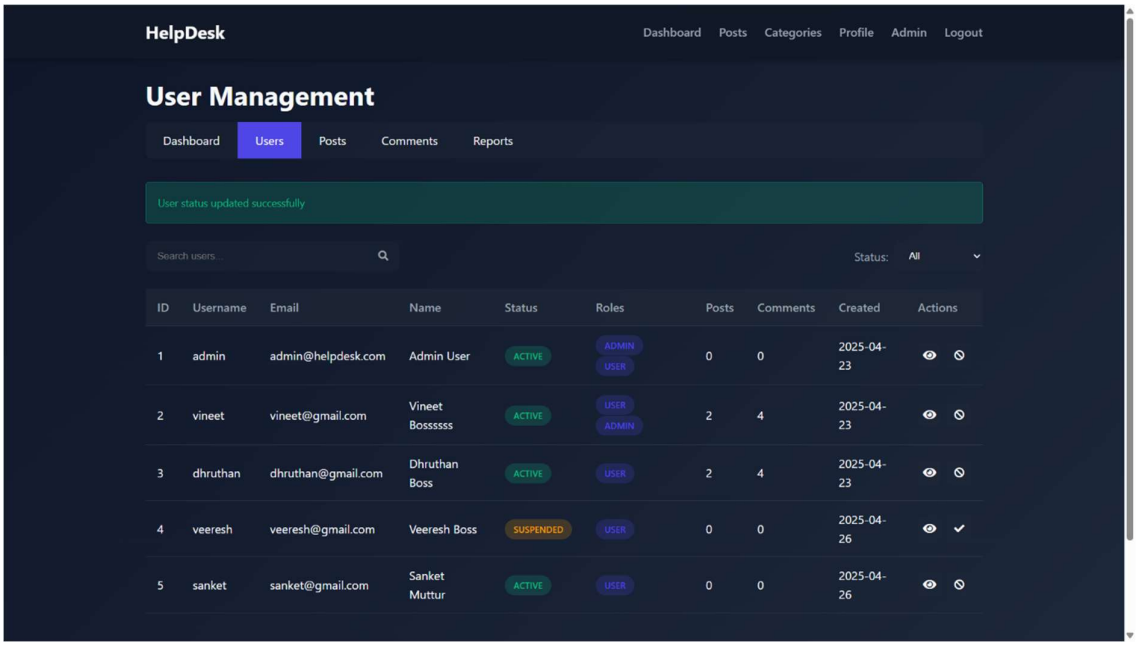
Report Management (Can View/Hide/Unhide/Mark as Resolved for individual Reported Post/Comment)



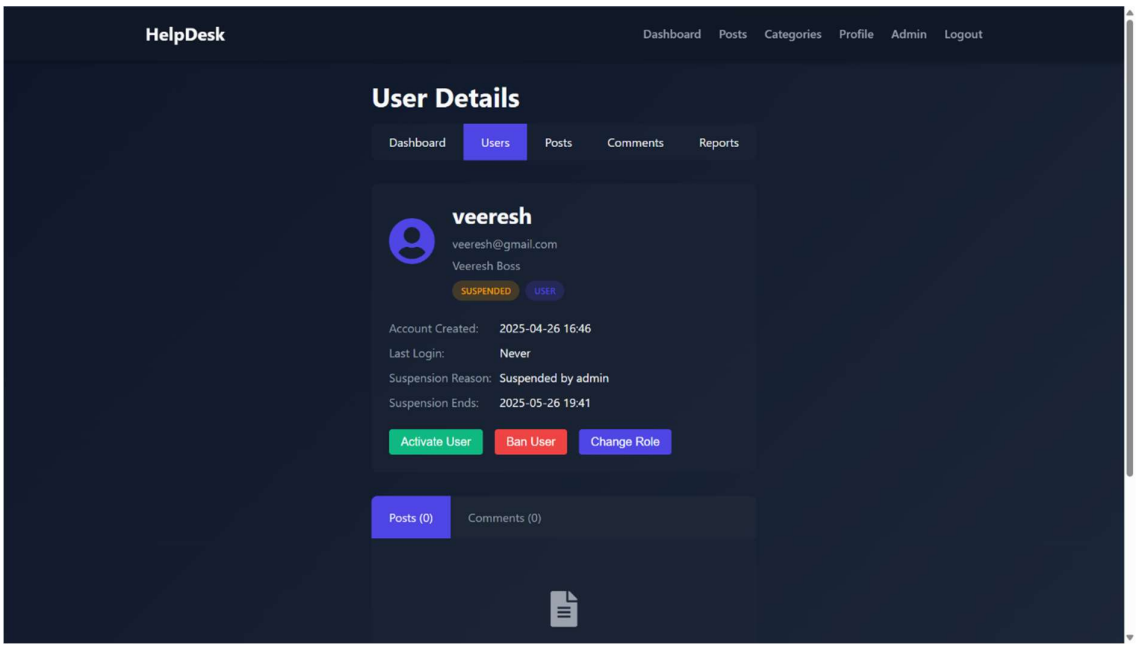
Viewing Individual User Profile (Can see every Post/Comment by the User)



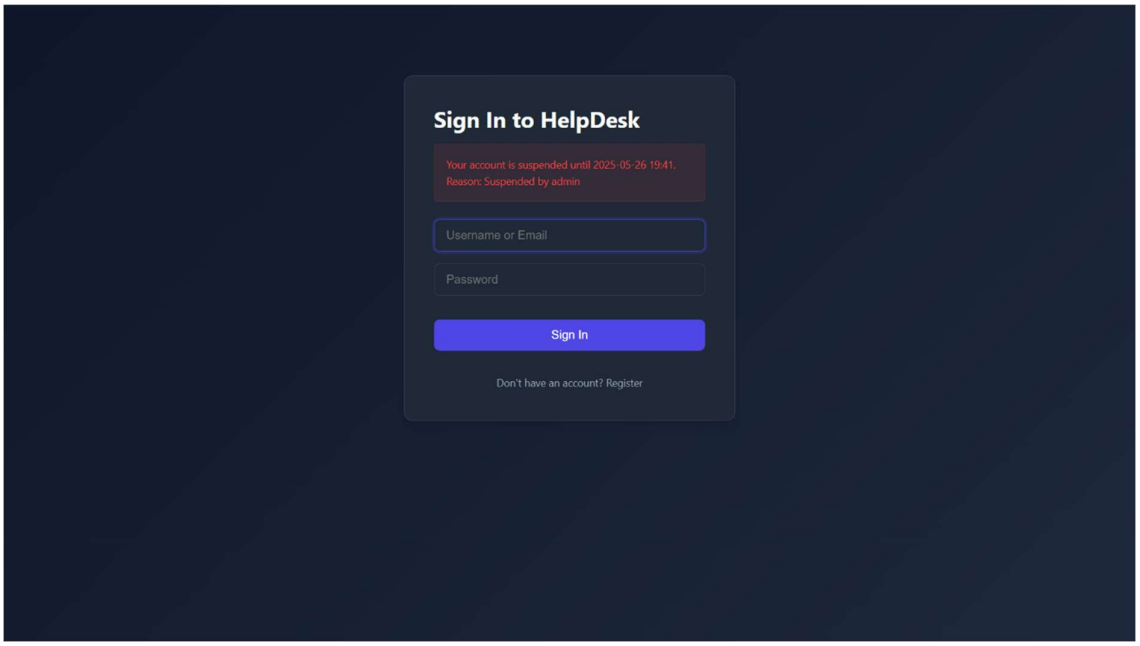
Suspended the User



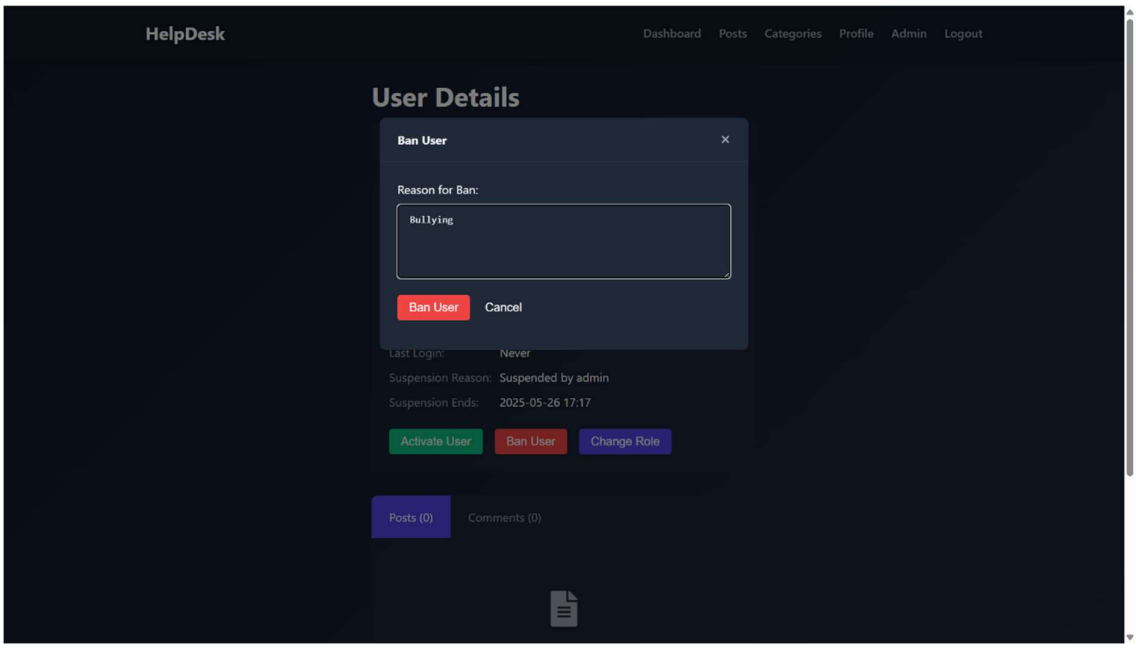
Suspended the User



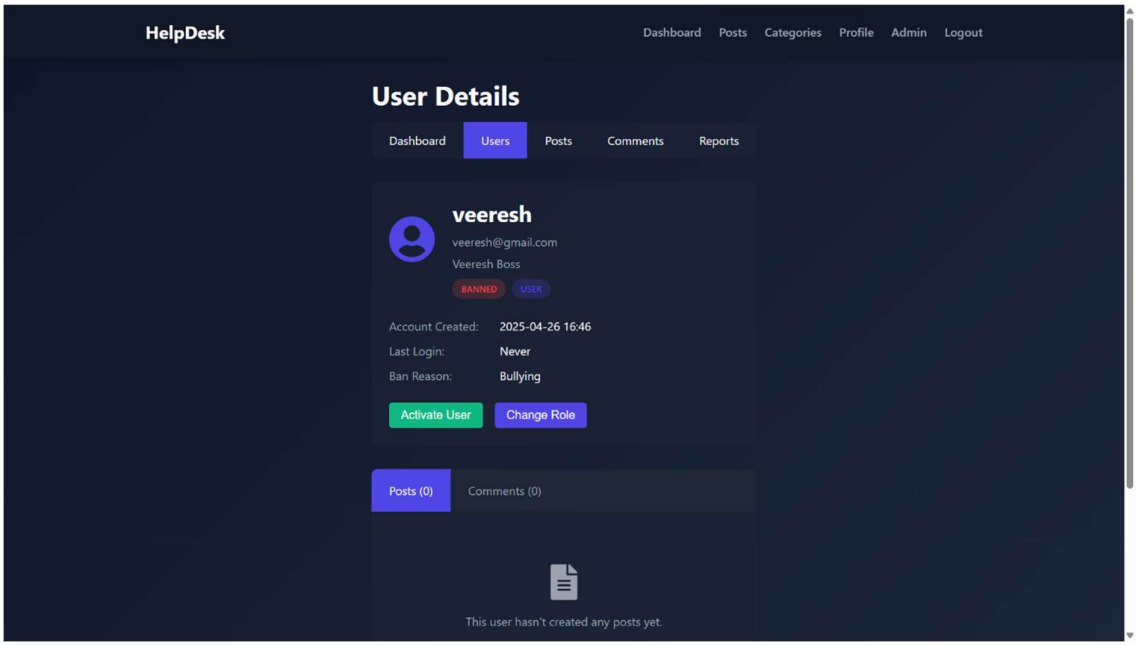
Suspended User can’t login or use the account for one month



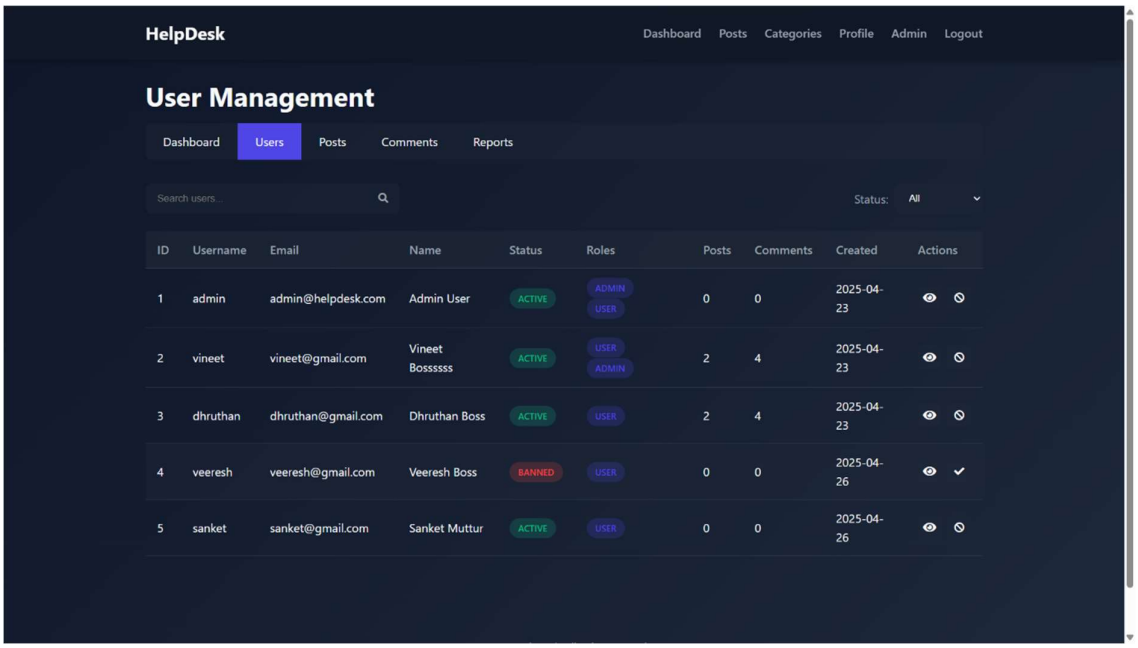
Banning the User



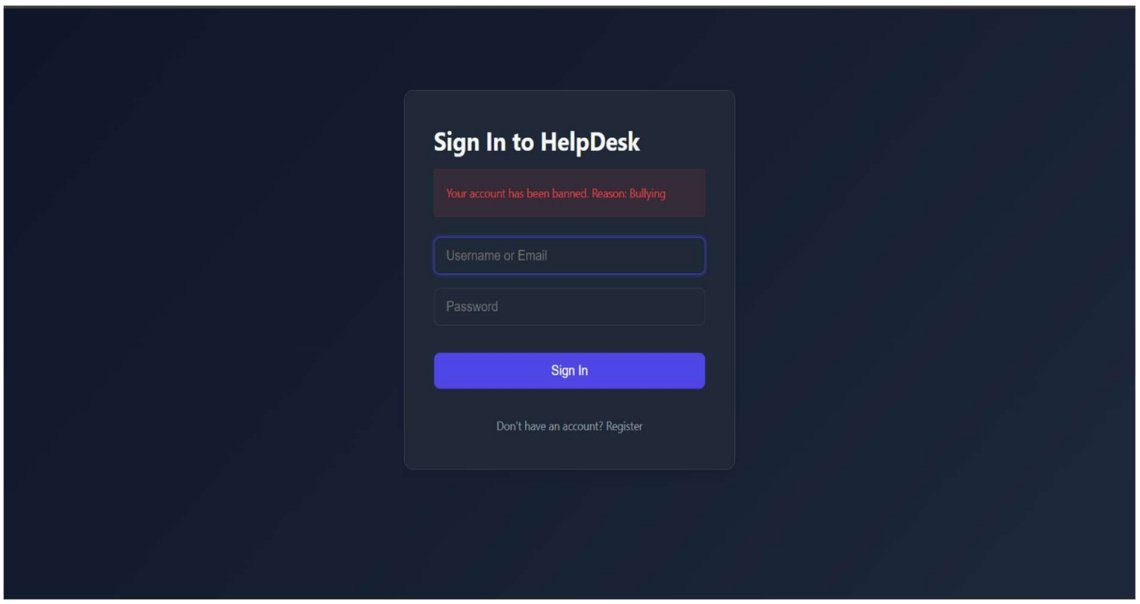
User Banned



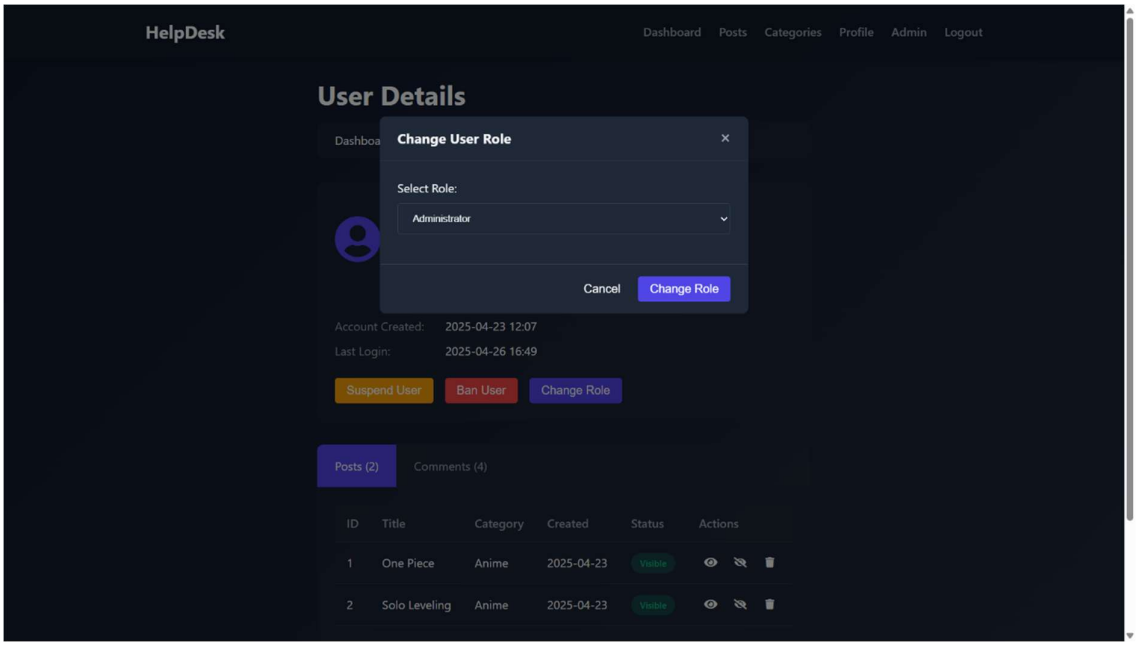
User Banned



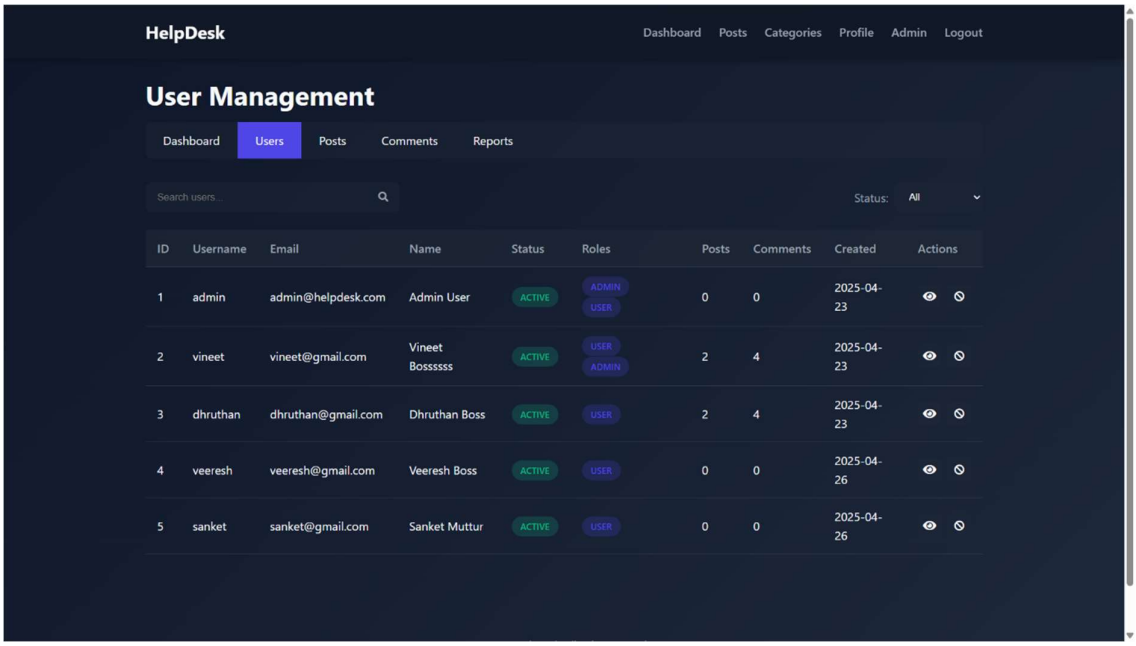
Banned User can’t login or use the account



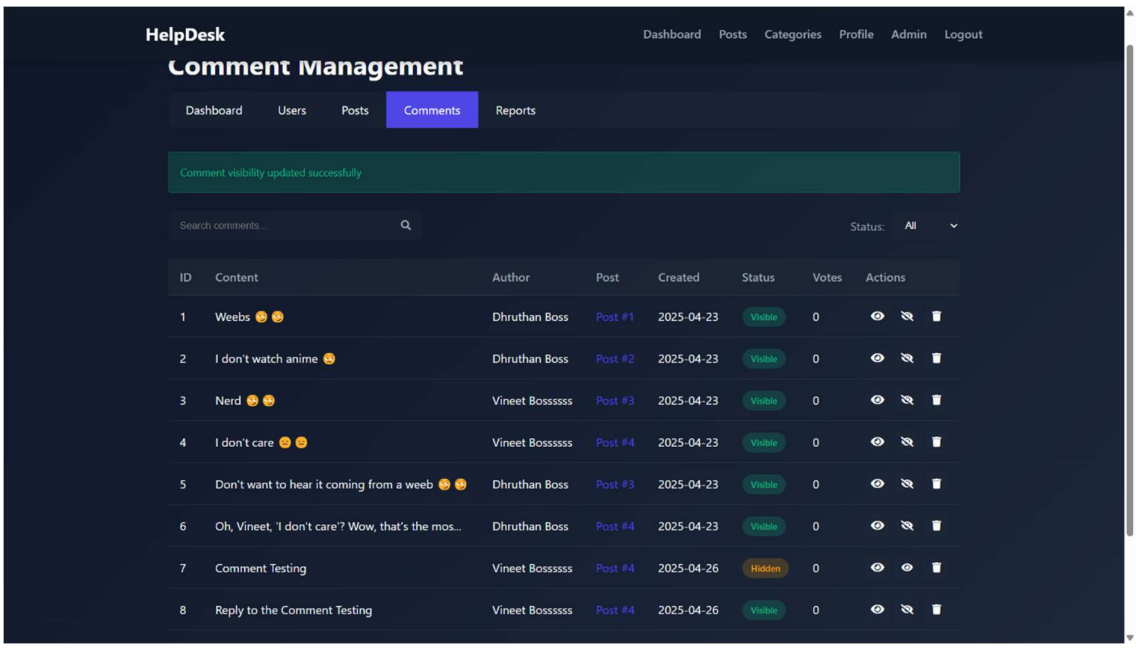
Changing the Role



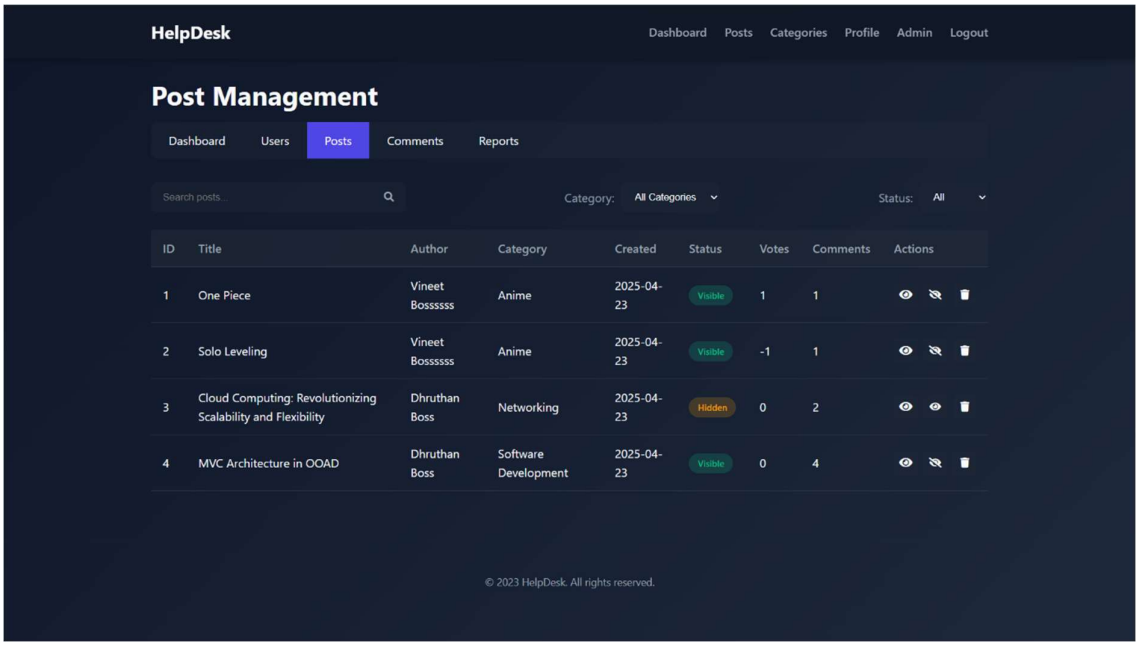
Role changed for the user “vineet” who now has same roles as “admin”



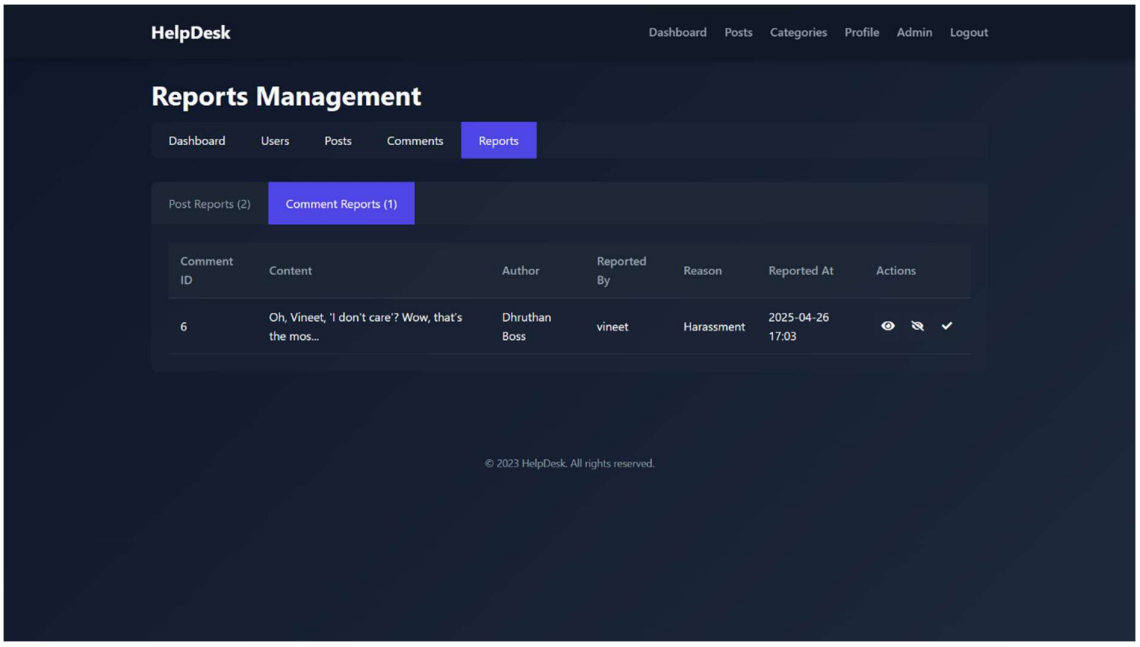
Hiding Comment



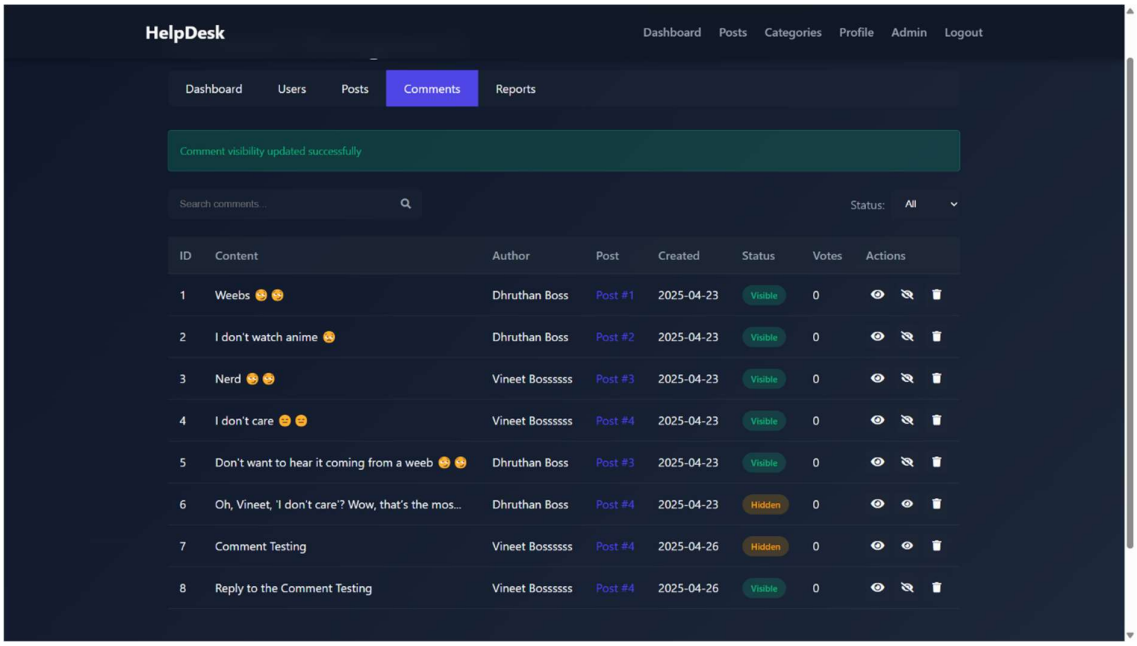
Hiding Post



Comment Hiding from Report



Comment Hidden



Individual contributions of the team members:

Name	Module Worked on
Dhruthan M N (PES2UG22CS181)	Major Use case 1
	Minor Use case 1
	Front Controller Pattern
	Model-View-Controller (MVC) Pattern
	Template Method Pattern
	Observer Pattern
Sanket B. Muttur (PES1UG23CS846)	Major Use case 4
	Minor Use case 4
	Repository Pattern
	Dependency Injection Pattern
	Data Transfer Object (DTO) Pattern
	Service Layer Pattern
Veeresh Amaragatti (PES1UG23CS847)	Major Use case 3
	Minor Use case 2
	Factory Method Pattern
	Builder Pattern
	Singleton Pattern
	Composite Pattern
Vineet Goel (PES1UG22CS697)	Major Use case 2
	Minor Use case 3
	Strategy Pattern
	Decorator Pattern
	Proxy Pattern