



## ***UE21CS352B - Object-Oriented Analysis & Design using Java***

### **Mini Project Report Blog Engine**

*Submitted by:*

<b>Rishik Makam</b>	<b>PES1UG21CS317</b>
<b>M Paramesh Kumar</b>	<b>PES1UG21CS335</b>
<b>Moksha Pradhan</b>	<b>PES1UG21CS342</b>
<b>N Manoj Kumar</b>	<b>PES1UG21CS369</b>

*6<sup>th</sup> Semester F Section*

**Prof. Bhargavi Mokashi**

**January - May 2024**

**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: Blog Engine**

The Blog Engine project offers a comprehensive set of features to create a dynamic and engaging blogging platform. By offering personalized feeds, search capabilities, and interactivity features, the platform aims to provide an engaging and user-friendly experience for both content creators and consumers. Additionally, the inclusion of admin privileges ensures the platform's integrity and compliance with community guidelines. Overall, the project demonstrates effective utilization of web development technologies to create a functional and engaging blogging platform.

### **Key Features:**

#### **1. Account Creation:**

Users can create accounts to access the platform. Account creation involves providing basic information such as username, email, and password.

#### **2. Personalized Feed:**

Upon logging in, users are presented with a personalized feed that displays blog posts only from the accounts they follow. This feature enhances user experience by delivering content tailored to their preferences.

#### **3. Search Functionality:**

The platform offers a search feature allowing users to search for specific posts or accounts. Users can search by keywords to find relevant posts, and also search for specific user accounts.

#### **4. Anonymous Access:**

Recent blog posts are accessible to all users without requiring login. However, users are restricted from commenting on posts or performing other interactive actions without authentication.

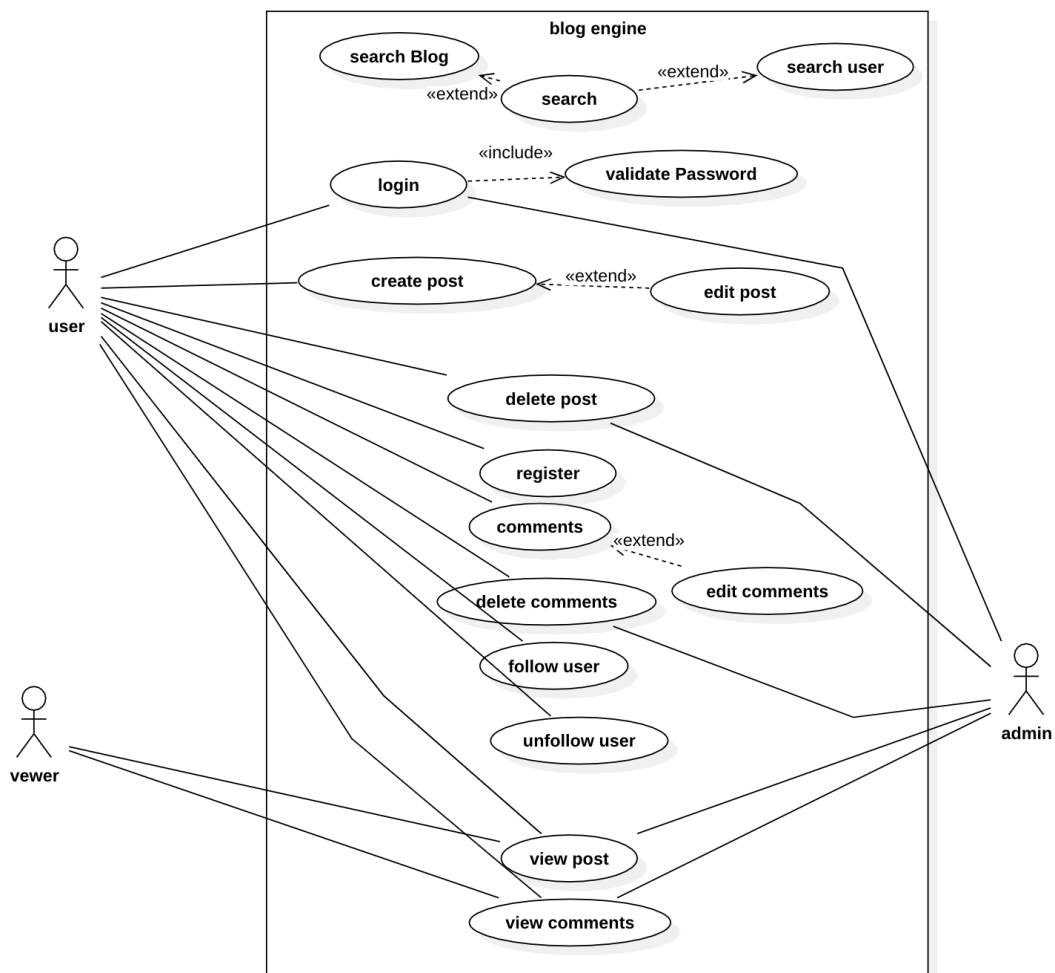
#### **5. Interactivity Features:**

- Following: Users can follow other accounts to receive updates from them in their feed.
- Commenting: Authenticated users can leave comments on blog posts, fostering engagement and discussion.

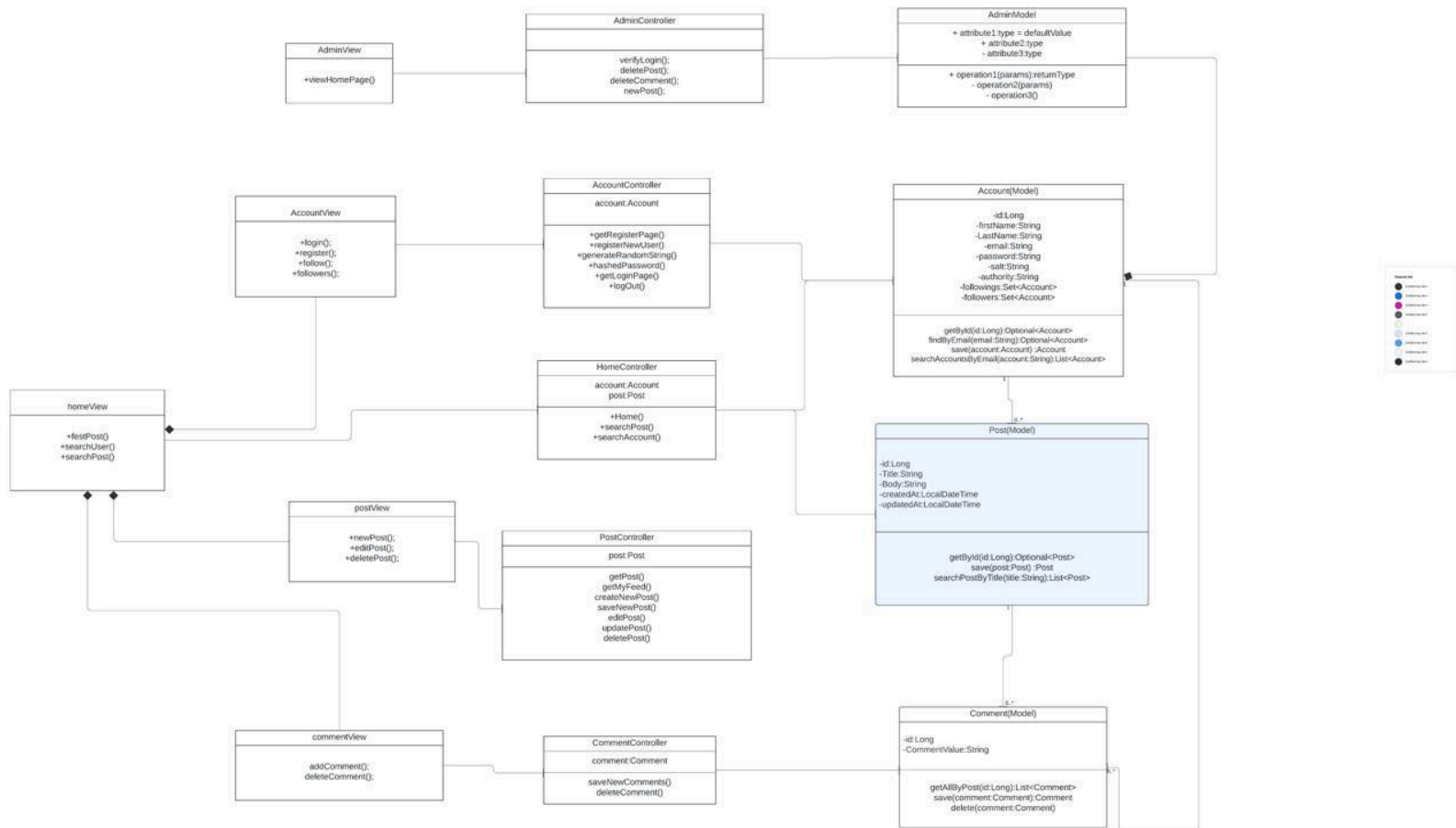
**6. Admin Privileges:**An admin role is designated with special privileges to manage the platform. Admins have the ability to delete inappropriate blog posts or comments to maintain the integrity of the platform.

## Models:

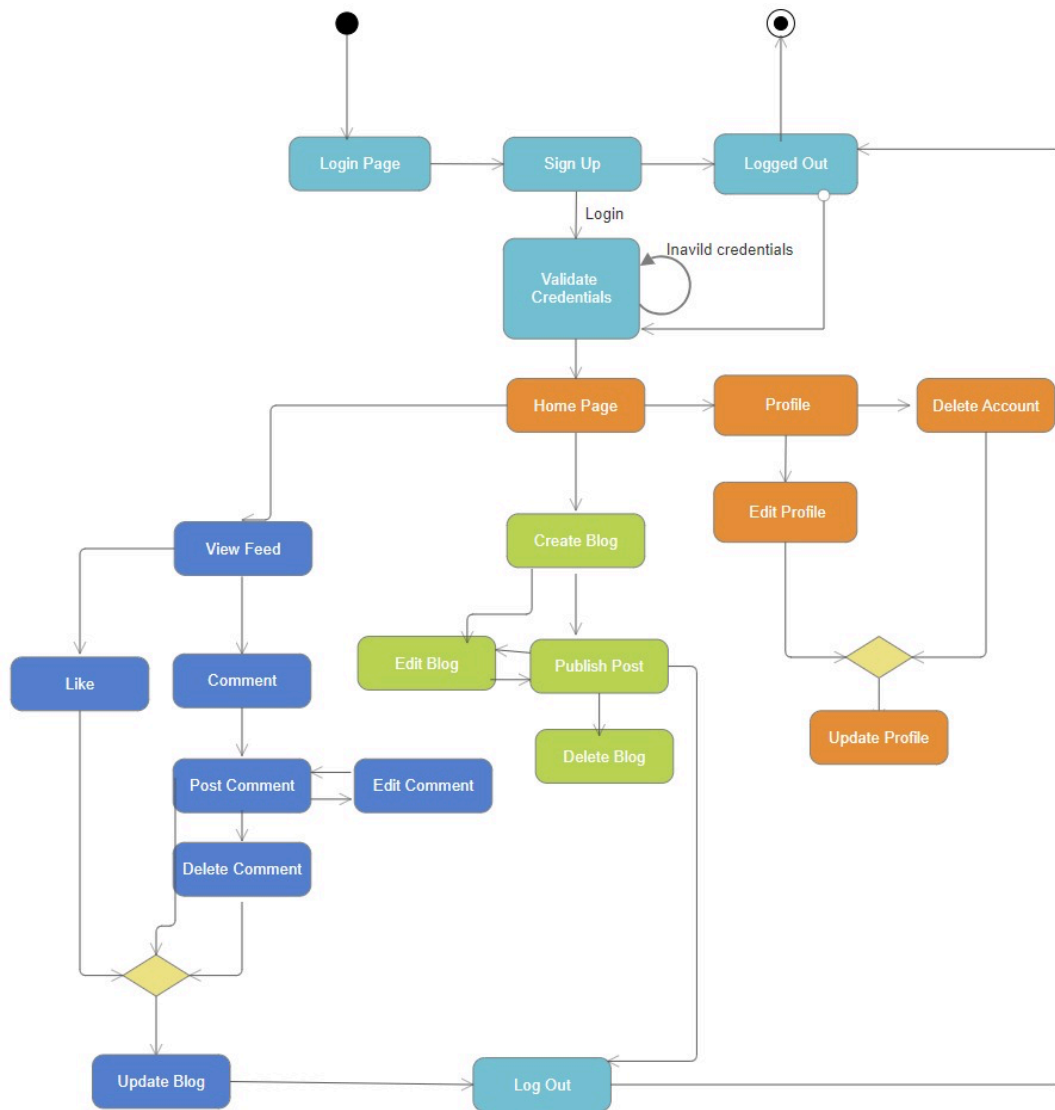
### Use Case Diagram:



## Class Diagram:



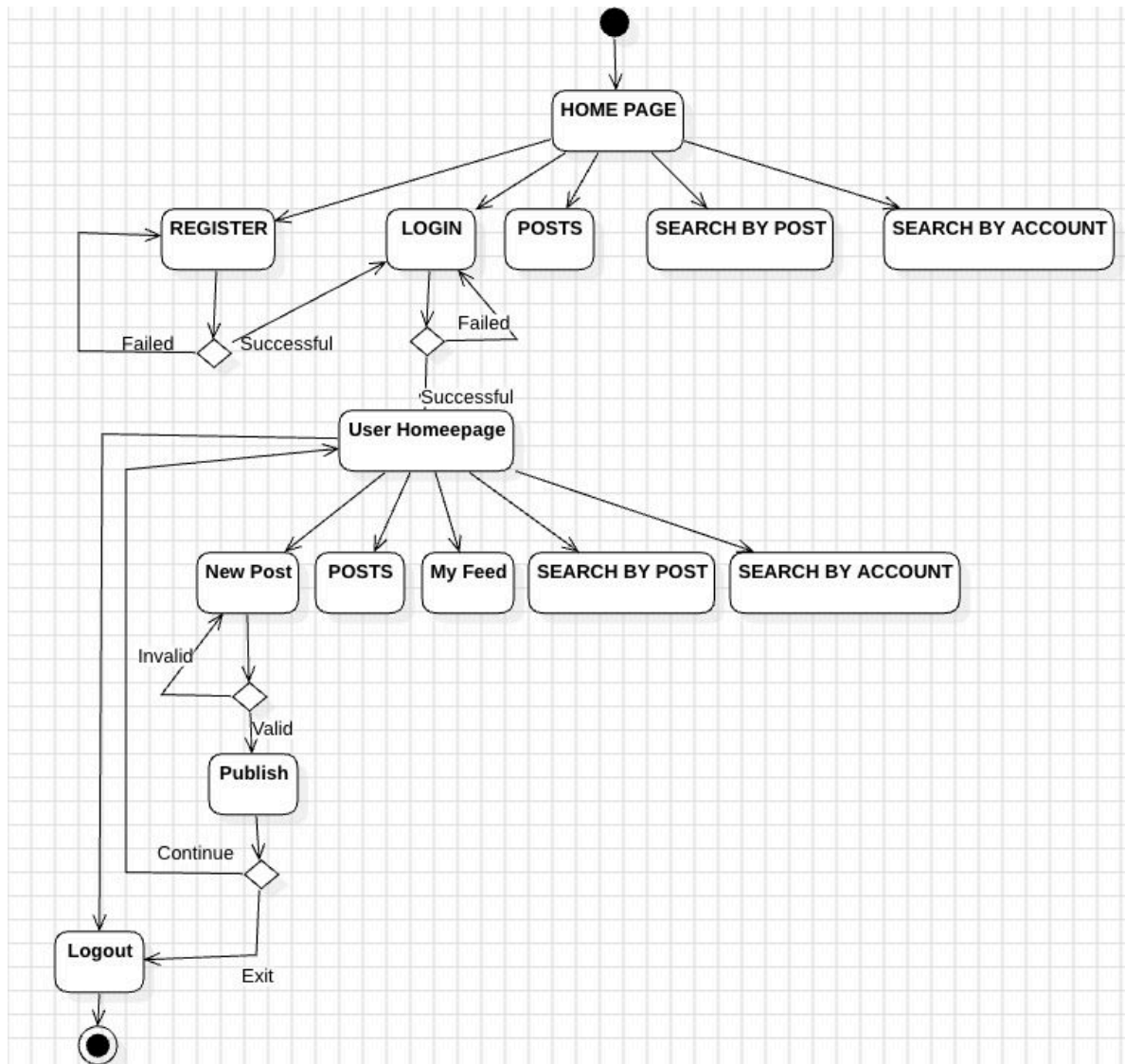
## State Diagram:



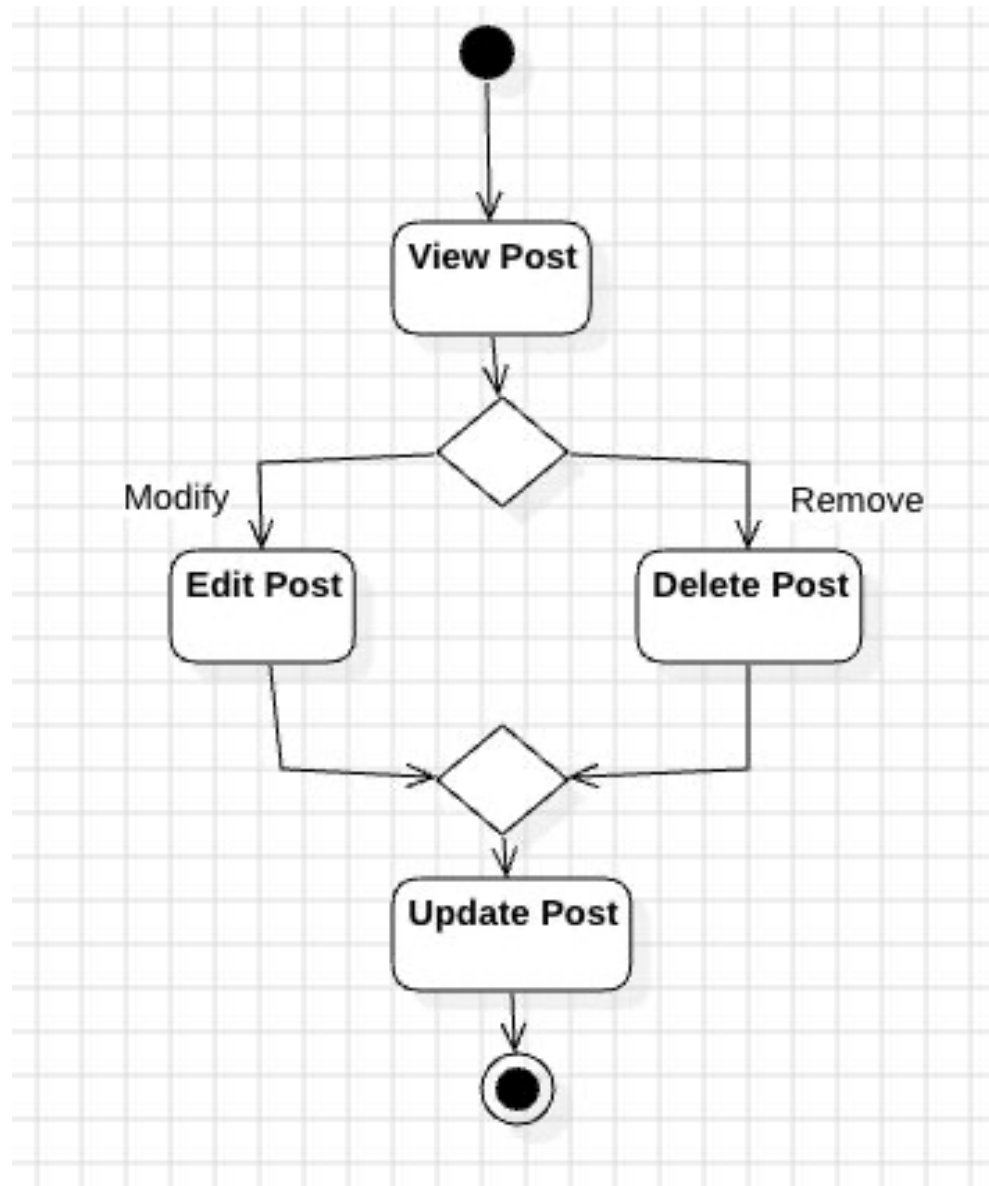
BLOG ENGINE : STATE DIAGRAM

## Activity Diagrams:

### 1. Major Use Case



## 2. Minor Use case



# Architecture Patterns, Design Principles, and Design Patterns:

## Architecture Patterns

### 1. Facade Design Pattern for Login Functionality

The Facade pattern is a **structural design pattern** that provides a simplified interface to a complex system of classes, hiding its internal implementation details. It promotes loose coupling between clients and subsystems, improving maintainability and ease of use.

In our BlogEngine, we've implemented the Facade design pattern to simplify the login process and encapsulate the complexities involved in authenticating users. The Facade acts as a unified interface to a set of interfaces in the subsystem, providing a higher-level interface that makes it easier to use.

The intent of the Facade pattern in our system is to provide a simple and unified interface to handle the various types of logins (user login and admin login), abstracting away the details of the authentication process and providing a seamless experience for users.

#### Implementation:

Our Facade class, named LoginFacade, exposes a set of high-level methods corresponding to the different types of logins supported by the system:

userLogin: Handles user authentication.

adminLogin: Handles admin authentication.

#### How Facade Helps:

- Simplified Interface: Clients interact with a single, easy-to-use interface, hiding the complexities of the underlying authentication subsystem.



- Encapsulation: The Facade encapsulates the login process, promoting information hiding and reducing dependency on internal subsystem components.
- Flexibility: Allows for changes in the authentication mechanism without affecting client code, as long as the Facade interface remains unchanged.

## 2. Factory Method

In our BlogEninge, we have implemented the **Factory** design pattern to manage the service objects effectively. The factory pattern ensures that the services are initialised whatever is required.

### Importance of Singleton Pattern:

Encapsulation of Object Creation: The Factory Method pattern encapsulates the creation of objects, allowing the system to delegate the responsibility of instantiation to specialized factory classes. This promotes code organisation and maintains a clear separation of concerns.

Centralized Configuration: Factory methods provide a centralized location for configuring and managing the instantiation of service objects. This centralization simplifies maintenance and promotes consistency in object creation throughout the system.

### How Singleton Helps:

Code Reusability: Factory methods promote code reuse by providing a reusable mechanism for creating objects across different parts of the system. This reduces duplication and promotes maintainability by ensuring consistency in object creation logic.

Enhanced Separation of Concerns: Separating the responsibility of object creation from the client code enhances the separation of concerns within the system. Clients focus on utilising services, while factories handle the complexities of object instantiation.

### 3. Builder Design Pattern for Post-Creation

In our Blog Engine, we have employed the Builder design pattern to facilitate the creation of complex post object with varying attributes. The Builder pattern separates the construction of a complex object from its representation, allowing for flexible and fluent object creation.

#### Importance of Builder Pattern:

- Simplified Object Creation: The Builder pattern simplifies the process of creating Post objects by providing a step-by-step approach to set individual attributes. This promotes readability and maintainability, especially when dealing with objects with many optional parameters.
- Flexible Configuration: With the Builder pattern, clients can configure Post objects with different combinations of attributes without the need for multiple constructors or setter methods. This flexibility accommodates diverse use cases and enhances the reusability of the Post class.
- Encapsulation of Construction Logic: The Builder encapsulates the construction logic of Post objects within the Builder class, abstracting away the complexity from the client code. This promotes encapsulation and reduces code duplication, resulting in cleaner and more modular codebase.

#### Drawbacks of Direct Object Instantiation:

- Constructor Overloading: In the absence of the Builder pattern, constructors with multiple parameters may lead to constructor overloading, complicating the class interface and making it challenging to maintain.

- Inflexible Configuration: Direct object instantiation with setters may result in an inflexible configuration process, especially when dealing with immutable objects or objects with complex initialization requirements.

#### How Builder Helps:

- Clear and Fluent API: The Builder pattern provides a clear and fluent API for constructing Post objects, allowing clients to specify attributes in a natural and readable manner.
- Complex Object Construction: With the Builder pattern, clients can construct Post objects with complex initialization requirements, such as default values, optional parameters, and conditional logic, in a systematic and organized manner.
- Immutable Post Objects: The Builder pattern facilitates the creation of immutable Post objects by enforcing parameter validation and ensuring that the object state remains consistent throughout the construction process.

The implementation of this pattern consists of the following components:

- Post Class: The post class represents the main entity in our system, encapsulating information about post details such as post body, author, created time and updated time. The post class has a private constructor to enforce immutability and a public static inner class named PostBuilder for constructing Post objects.
- PostBuilder Class: The ProductBuilder class serves as the builder for constructing Post objects step by step. It provides setter methods for configuring individual attributes of the Post, such as name, price, category, and quantity. Each setter method returns the builder instance itself, allowing for method chaining to set multiple attributes fluently.

- build() Method: The build() method in the PostBuilder class finalizes the construction process by instantiating a new Post object with the configured attributes. This method creates a new instance of the Post class using the values set by the client through the setter methods.

## 4. Model – View – Controller Pattern (MVC)

The Model-View-Controller (MVC) architecture pattern is widely used in software development to separate the concerns of an application into three interconnected components: the Model, View, and Controller. Each component has a specific role and responsibility within the system.

### Components of MVC:

#### 1. Model:

- The Model represents the application's data and business logic. It encapsulates the data access, manipulation, and validation operations, independent of the user interface or presentation logic.
- In our BlogEngine, the Model includes entities such as Post, Comments, Account and Likes and corresponding services for data retrieval, manipulation, and persistence.

#### 2. View:

- The View represents the presentation layer of the application, responsible for displaying data to the user and handling user input. It presents the information retrieved from the Model in a user-friendly format and communicates user actions back to the Controller.
- In our system, the View comprises the user interface components such as web pages, forms, and dashboards, which have been designed using **Thymeleaf**, that enable interaction with the users and provide access to the functionalities offered by the system.

#### 3. Controller:

- The Controller acts as an intermediary between the Model and the View, handling user input, processing requests, and updating the Model accordingly. It orchestrates the flow of data and controls the application's behavior based on user interactions.

- In our system , the Controller contains the logic to interpret user requests, invoke appropriate services from the Model layer, and render the corresponding views to the user based on the requested actions.

#### Benefits of MVC:

- **Separation of Concerns:** MVC promotes a clear separation of concerns, allowing each component to focus on its specific responsibilities. This enhances code maintainability, reusability, and testability by minimizing dependencies and facilitating modular design.
- **Parallel Development:** The modular structure of MVC enables parallel development of different components by different teams or developers, promoting collaboration and accelerating the development process.
- **Flexibility and Scalability:** MVC facilitates flexibility and scalability by allowing modifications or extensions to one component without affecting the others. This enables the system to adapt to changing requirements and scale efficiently as the system evolves.

## Architecture Principles

### 1. Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is one of the SOLID principles of object-oriented design, proposed by Robert C. Martin. SRP states that a class should have only one reason to change, meaning that it should have only one responsibility or job within the system.

#### Importance of SRP:

- **Improved Reusability:** Classes with well-defined responsibilities are more likely to be reusable in other parts of the system or in different projects. They encapsulate specific functionalities, making them versatile and adaptable to various scenarios.
- **Facilitates Testing:** When each class has a single responsibility, it becomes simpler to write unit tests for individual components. Testing

becomes more focused and comprehensive, leading to better test coverage and more robust code.

### Implementation of SRP in the BlogEngine:

In our BlogEngine, we have adhered to the Single Responsibility Principle (SRP) to enhance the maintainability, readability, and extensibility of our codebase. SRP dictates that a class should have only one reason to change, meaning it should have only one responsibility or purpose.

#### 1. Account Class:

- The Account class is responsible for representing user entities within the system.
- Its responsibilities include managing user information such as username, email, password, and address.
- The Account class adheres to SRP by solely focusing on managing user-related data and behaviour. It does not contain logic unrelated to user management.

#### 2. Post Class:

- The Post class encapsulates product entities in the system.
- Its responsibilities include storing product information such as title, body, createdAt and UpdateAt.
- The Post class conforms to SRP by concentrating solely on product-related functionalities, such as retrieving product details, editing post information.

#### 3. Comment Class:

- The Comment class represents the comment to the post functionality in the system.
- The Comment class follows SRP by handling comment-related operations exclusively, ensuring separation of concerns and modularity.

## **2. Interface Segregation Principle (ISP)**

The Interface Segregation Principle (ISP) is one of the five SOLID principles of object-oriented design. It states that a client should not be forced to depend on interfaces it does not use. In other words, interfaces should be specific to the needs of the clients that use them, rather than being overly general and including methods that are not relevant to all clients.

#### Importance of Interface Segregation Principle (ISP):

- Reduced Dependency: By segregating interfaces based on client-specific functionalities, ISP reduces the dependency of clients on irrelevant methods. This promotes loose coupling between components and facilitates easier maintenance and evolution of the codebase.
- Improved Cohesion: ISP leads to interfaces that are more focused and cohesive, containing only the methods that are relevant to a specific client or group of clients. This improves the readability and understandability of the code, as well as making it easier to reason about and maintain.
- Enhanced Testability: Segregating interfaces based on client-specific requirements allows for more targeted and efficient testing. Each client can be tested independently, leading to more robust and reliable testing processes.
- Flexible Evolution: ISP makes the system more flexible and adaptable to change. When new clients or functionalities are introduced, it's easier to extend existing interfaces or create new ones tailored to their specific needs without impacting existing clients or interfaces.

#### Implementation of ISP in theBlogEngine:

In our blogEngine, we have applied the Interface Segregation Principle (ISP) to ensure that client-specific interfaces are tailored to the needs of the clients, thereby preventing clients from being forced to depend on interfaces they do not use.

##### 1. AccountService Interface:

- The AccountService interface defines methods related to user management, such as `createAccount()` and `deleteAccount()`.

- By segregating account-specific operations into the AccountService interface, we adhere to ISP, ensuring that clients requiring Account management functionalities depend only on this interface.

## 2. PostService Interface:

- The PostService interface declares methods for product management, including savePost(), editPost(), and deletePost().
- By segregating postt-related operations into a PostService interface, we follow ISP, allowing clients needing postt management capabilities to rely solely on this interface.

## 3. CommentService Interface:

- The CommentService interface specifies methods for cart management, such as createComment(),deleteComment().
- Adhering to ISP, we segregate cart-specific functionalities into a CartService interface, enabling clients requiring cart management features to interact solely with this interface.

## **Dependency Inversion Principle (DIP):**

The Dependency Inversion Principle (DIP) is a fundamental tenet of object-oriented design, forming one of the core principles of the SOLID principles. It states that high-level modules should not depend on low-level modules, but both should depend on abstractions. Additionally, it emphasizes that abstractions should not depend on details, but details should depend on abstractions.

### **Importance of Dependency Inversion Principle (DIP):**

- Decoupling Components: DIP promotes decoupling between high-level and low-level modules by introducing abstractions. This reduces the direct



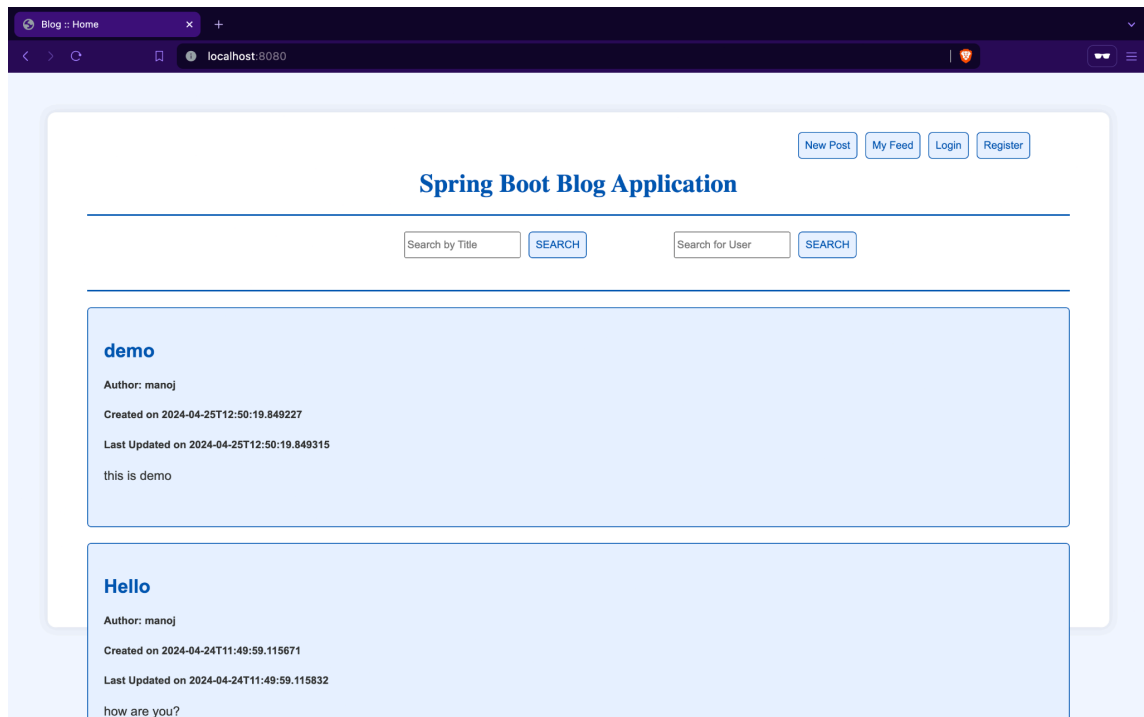
dependencies between modules, facilitating easier maintenance, testing, and evolution of the system.

- Flexibility and Extensibility: By relying on abstractions rather than concrete implementations, DIP makes the system more flexible and extensible. New implementations can be introduced without requiring changes to existing code, fostering adaptability to changing requirements.
- Encouraging Good Design Practices: DIP encourages the use of interfaces or abstract classes to define contracts between components, promoting good design practices such as separation of concerns and modularization.

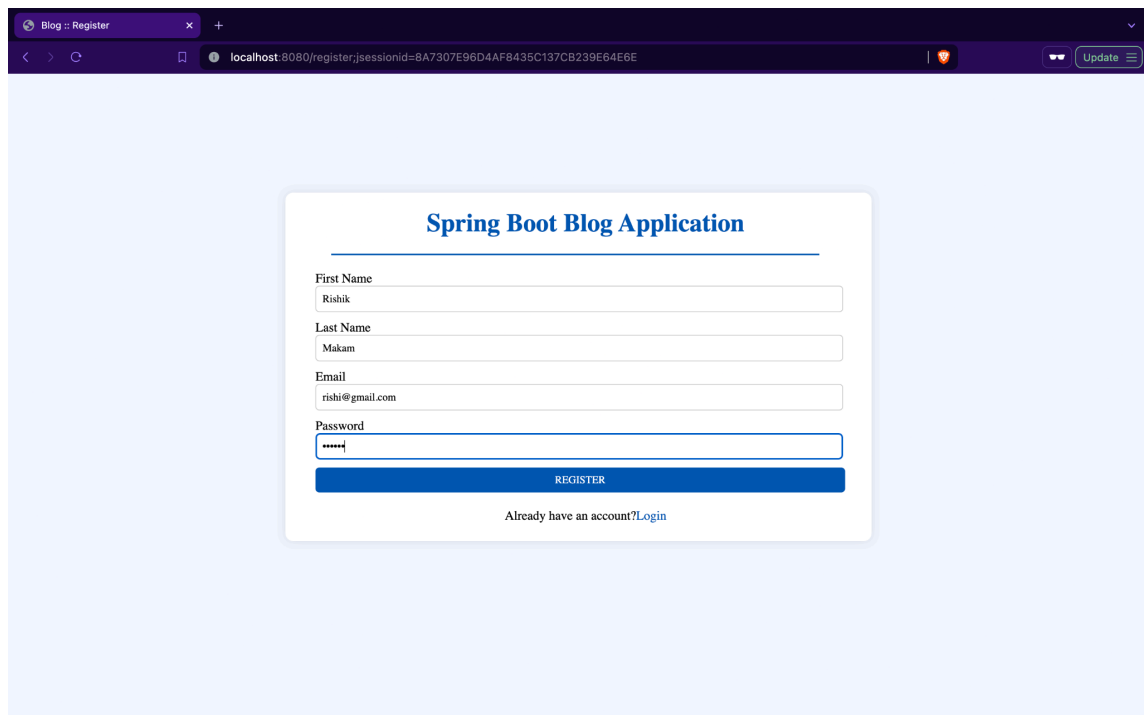
#### Implementation of DIP in theBlogEninge:

The CommentController follows the Dependency Inversion Principle by relying on interfaces (CommentService, AccountService, and PostService) for its dependencies instead of concrete implementations. This allows for flexibility and easier maintenance. Each method in the controller has a single responsibility related to comment management, promoting interface segregation. Overall, the controller adheres to DIP through constructor injection and abstraction over concretion.

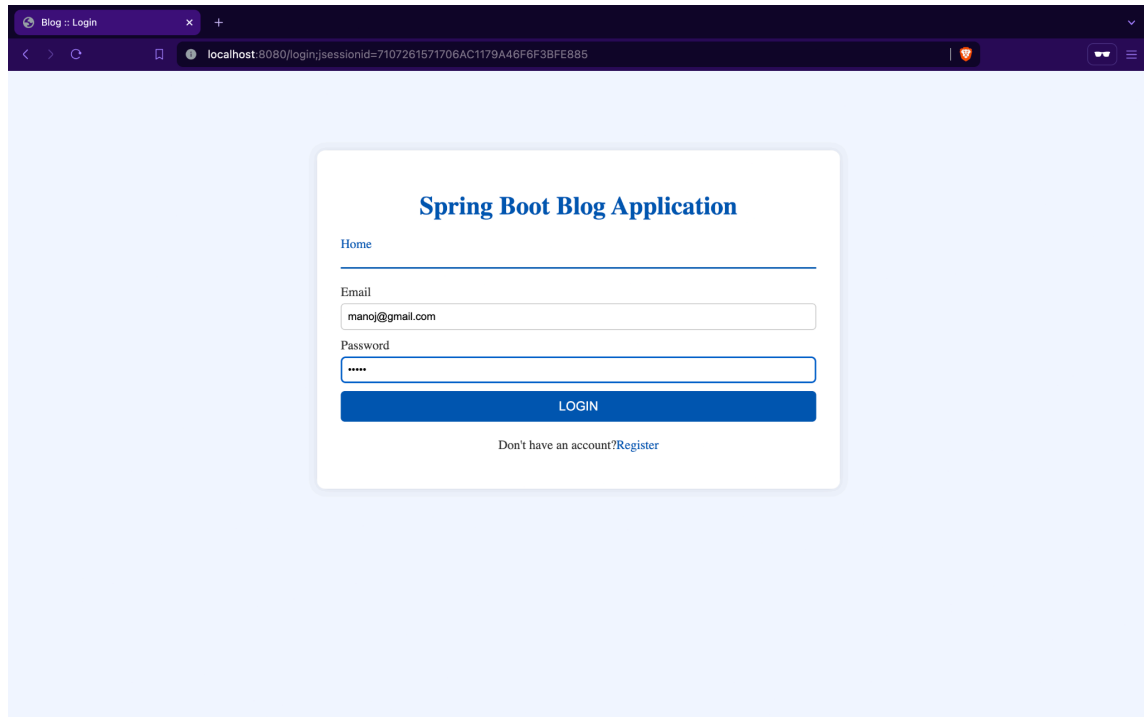
HOME PAGE :



## REGISTER PAGE :



## LOGIN PAGE:



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/login;jsessionid=7107261571706AC1179A46F6F3BFE885'. The page title is 'Blog :: Login'. The main content area features a white card with the heading 'Spring Boot Blog Application'. Below the heading is a 'Home' link. The login form includes an 'Email' field with 'manoj@gmail.com', a 'Password' field with masked characters, and a blue 'LOGIN' button. A link for users without an account is provided at the bottom.

Spring Boot Blog Application

[Home](#)

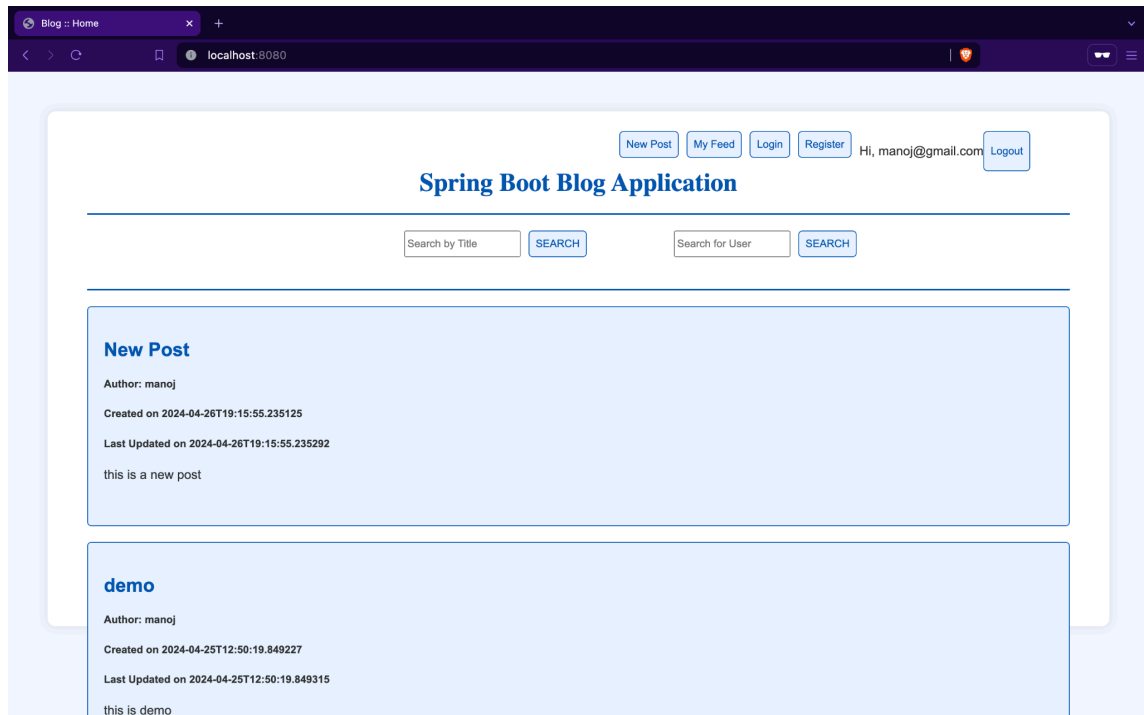
Email  
manoj@gmail.com

Password  
\*\*\*\*\*

[LOGIN](#)

Don't have an account? [Register](#)

## USER HOME PAGE (AFTER LOGGING IN):



The screenshot shows the user home page after logging in. The browser window title is 'Blog :: Home' and the address bar shows 'localhost:8080'. The page features a navigation bar with links for 'New Post', 'My Feed', 'Login', 'Register', and a 'Logout' button next to the user's name 'Hi, manoj@gmail.com'. The main content area has two search bars: 'Search by Title' and 'Search for User', both with 'SEARCH' buttons. Below the search bars are two post cards. The first card is titled 'New Post' and shows details for a post by 'manoj' created on 2024-04-26T19:15:55.235125, last updated on 2024-04-26T19:15:55.235292, with the content 'this is a new post'. The second card is titled 'demo' and shows details for a post by 'manoj' created on 2024-04-25T12:50:19.849227, last updated on 2024-04-25T12:50:19.849315, with the content 'this is demo'.

[New Post](#) [My Feed](#) [Login](#) [Register](#) Hi, manoj@gmail.com [Logout](#)

Spring Boot Blog Application

[SEARCH](#)  [SEARCH](#)

**New Post**

Author: manoj

Created on 2024-04-26T19:15:55.235125

Last Updated on 2024-04-26T19:15:55.235292

this is a new post

**demo**

Author: manoj

Created on 2024-04-25T12:50:19.849227

Last Updated on 2024-04-25T12:50:19.849315

this is demo

POST:

Blog :: Post

localhost:8080/posts/12

Update

[Home](#)

**demo**

Author: manoj

Created on: 2024-04-25T12:50:19.849227

Last Updated on: 2024-04-25T12:50:19.849315

this is demo

Comment

Post

**Comments**

hey

Author: Paramesh

[Delete](#)

NEW POST:

Blog :: New Post

localhost:8080/posts/new

Home

**Write New Post**

Title

New Post

Body

this is a new post

PUBLISH POST

Published by manoj

EDIT POST:

Blog :: Post

localhost:8080/posts/9

Update

[Home](#)

Windows OS

Author: Paramesh

Created on: null

Last Updated on: 2024-04-23T22:57:56.905411

This is the worst OS

Edit

Delete

Comment

Post

Comments

great content

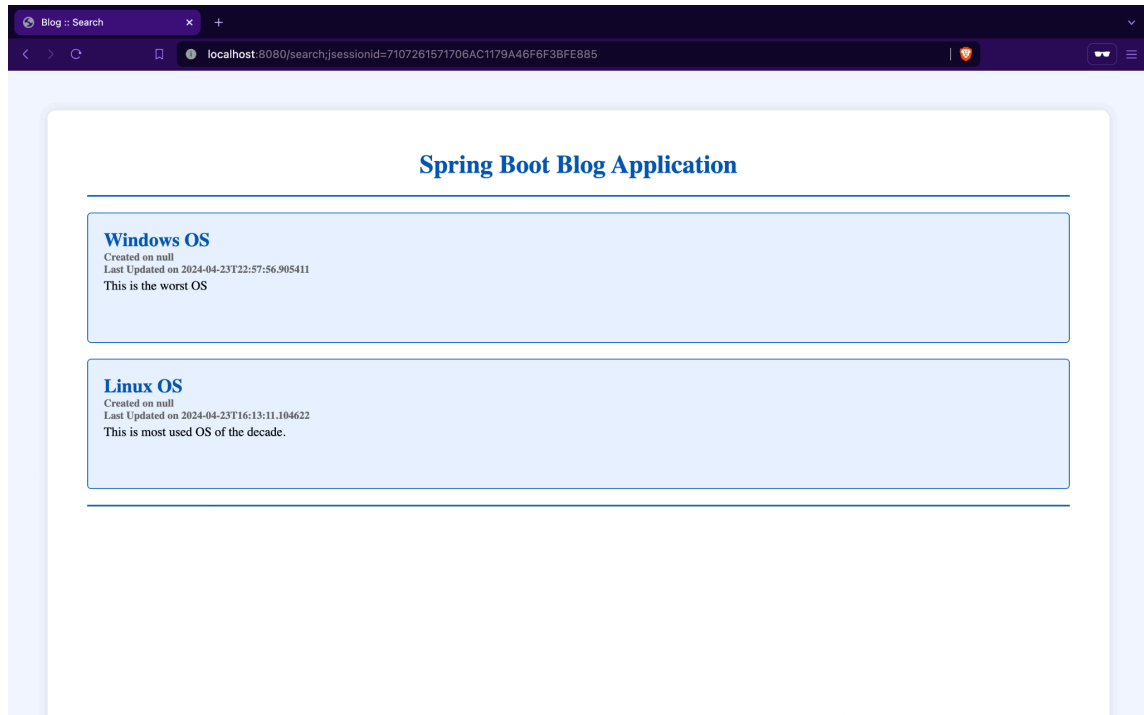
Author: admin

how are you?

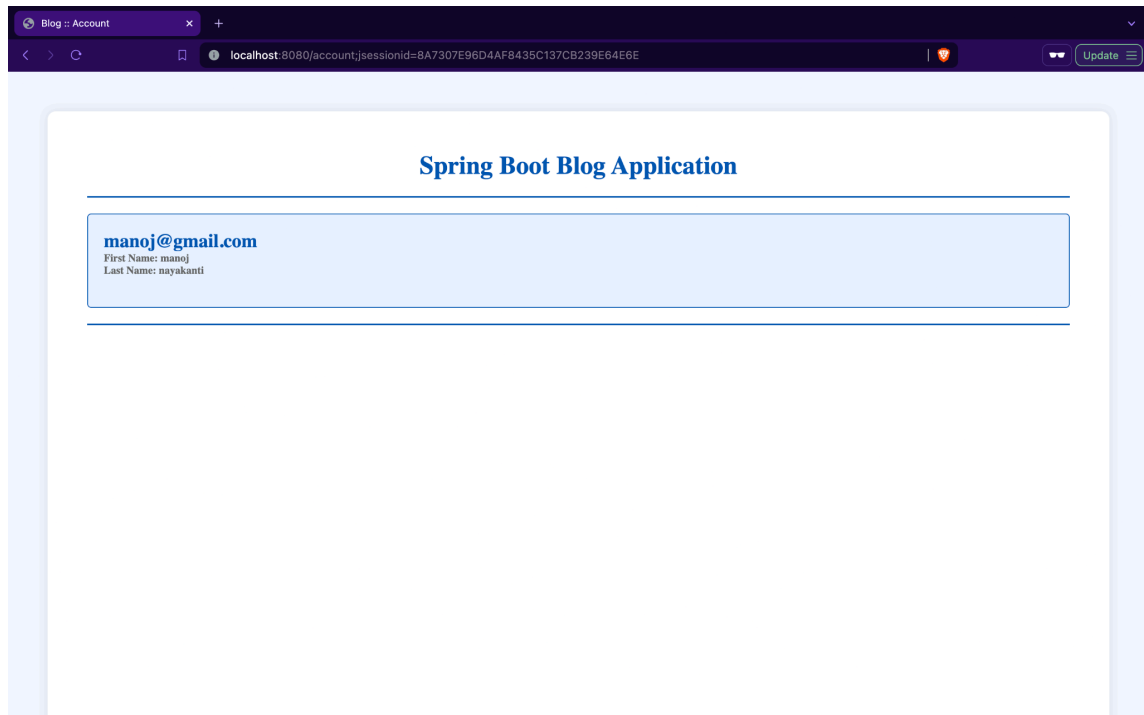
Author: Paramesh

Delete

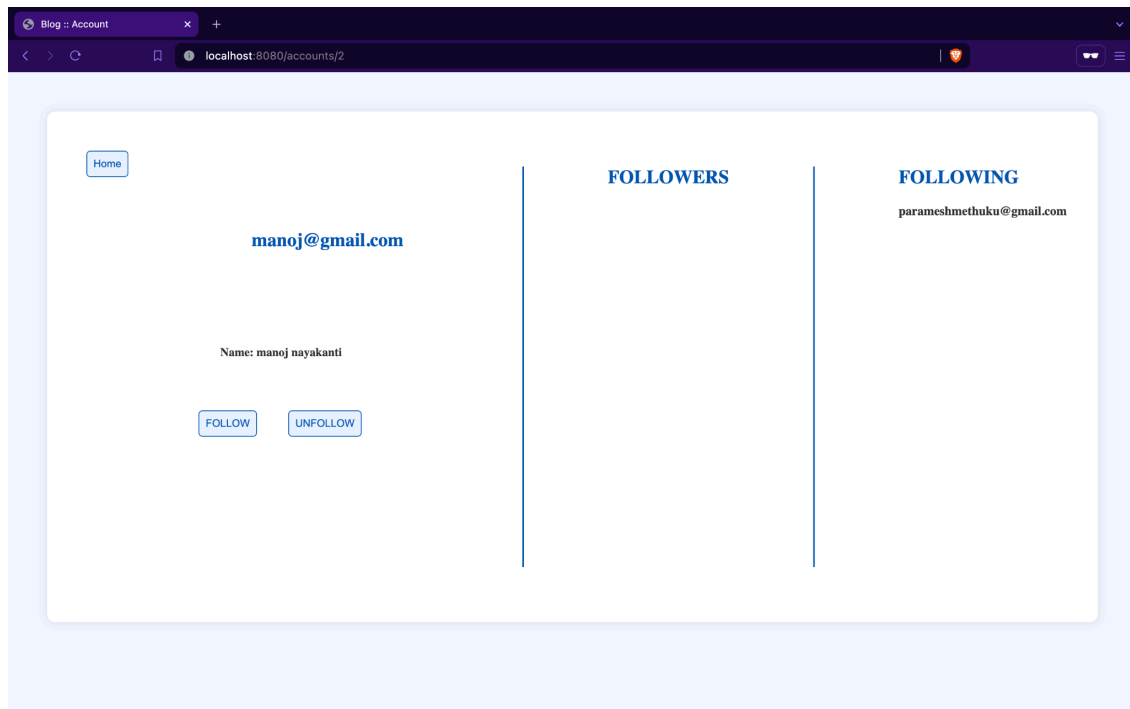
## SEARCH BY POST :



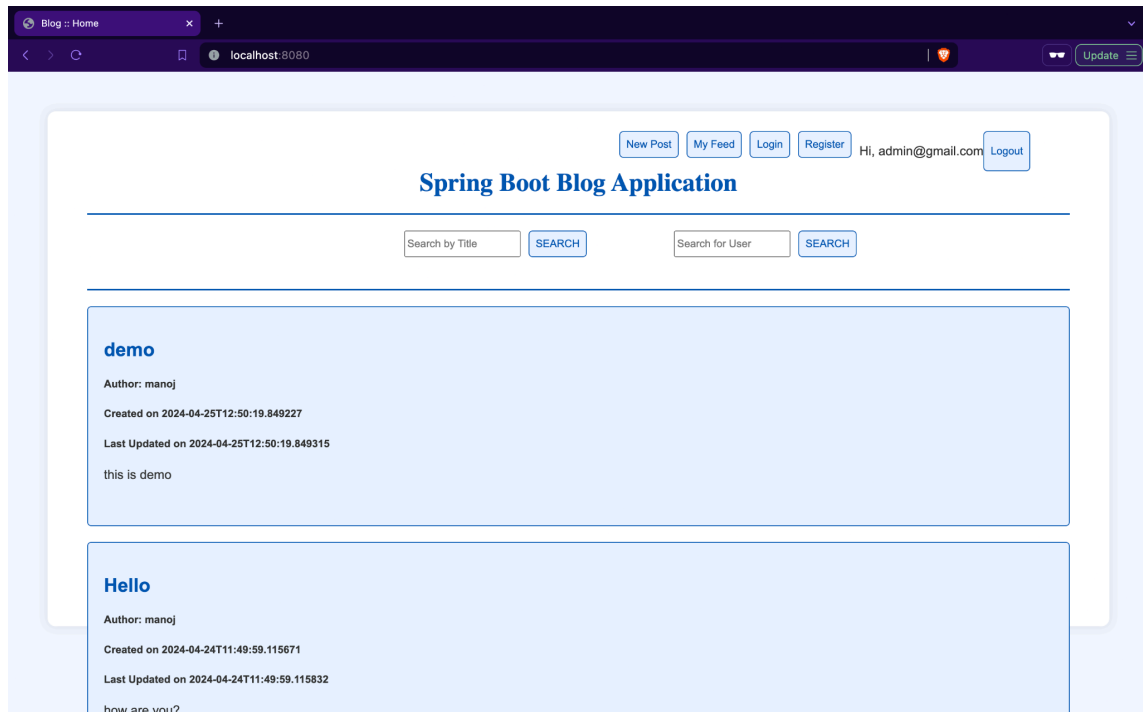
## SEARCH BY ACCOUNT:



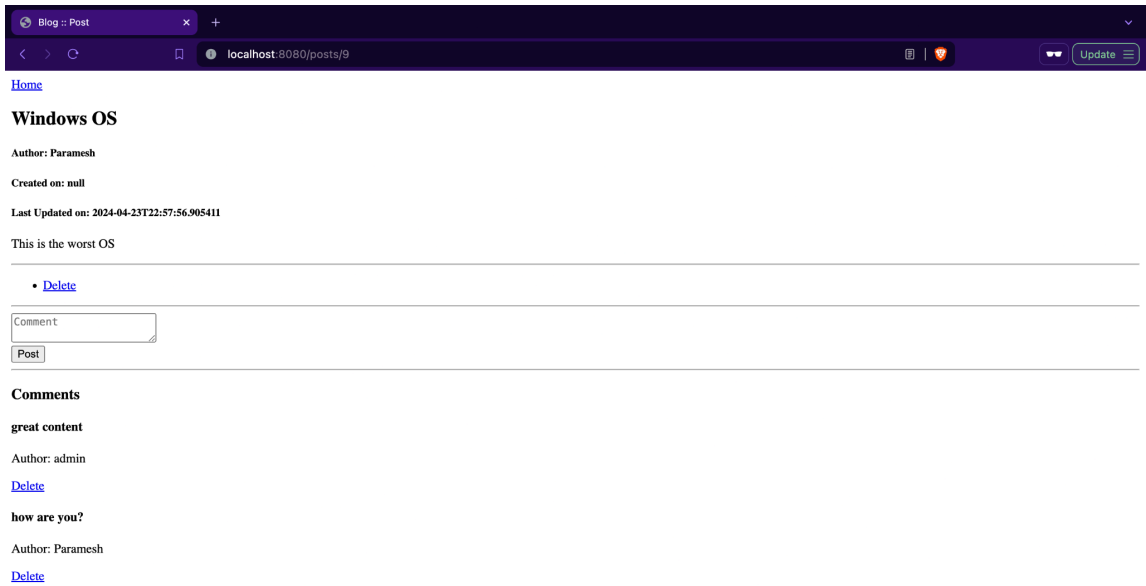
ACCOUNT PAGE (with user blogs , followers and following):



ADMIN HOME PAGE :



ADMIN OVERVIEW OF A POST:



GITHUB: <https://github.com/Paramesh555/BlogEngineMVC>

Individual contributions of the team members:

Name	Module worked on
Rishik Makam	Post Implementation
M Paramesh Kumar	Account Implementation
Moksha Pradhan	Follower Implementation
N Manoj Kumar	Comments Implementation



