Enterprise Software Platform Project Report

Team SOAP

Github URL - https://github.com/AtharvaJadhav/scalable-microblogging-platform

1. Introduction:

Project Overview

- Purpose and Functionality: The software platform is a scalable microblogging platform, designed as a clone of Reddit. It allows users to create posts, comment on them, and engage in community discussions, similar to the popular social media and news aggregation site. This platform provides a user-friendly interface for content sharing and discussion, fostering a vibrant online community.

Objectives

- Community Engagement: To create a dynamic online space where users can freely share, discuss, and engage with content.
- Scalability and Performance: Ensuring the platform can handle a growing number of users and increased data flow without compromising on performance.
- User Experience: Deliver a seamless and intuitive user interface, making content creation, sharing, and interaction straightforward and enjoyable.
- Security and Privacy: Implement robust security measures to protect user data and ensure privacy.

Scope

- Capabilities: The platform supports post creation, commenting, and user interaction in a structured and moderated environment. It includes features that allow for easy navigation, content discovery, and user participation.
- Limitations: While inspired by Reddit, this platform may not initially include the full range of features found on the larger site, such as advanced content filtering, comprehensive moderation tools, or a highly personalized recommendation system.

List of Technologies/Packages

Client

- UI: Next.js, Chakra UI, Formik
  Data Handling: GraphQL, Urql, GraphQL codegen

Server

- API: GraphQL API, Node.js, Express, Type-GraphQL, Apollo-Server-Express

Database

- Main Storage: PostgreSQL, Type-ORM

Cache and Session Store

- Session Management: Express-Session, Redis, Connect-Redis

Security

- Encryption and Hashing: Argon2

2. Design Patterns Used

Singleton Pattern

Singleton was used for managing database connections with PostgreSQL, ensuring there was only one connection pool instance throughout the application. This pattern helps in efficiently managing resources, such as database connections, and provides a consistent access point throughout the application. It reduced overhead by preventing multiple instances of the same connection pool.

Factory Pattern

The Factory pattern was applied in creating objects for various types of posts or user profiles. This pattern allows the creation of objects without exposing the creation logic to the client and refers to the newly created object using a common interface. It offers flexibility in adding new types of posts or user entities without altering existing code. This makes the platform more extensible and maintainable, especially important as the platform scales and evolves.

Observer Pattern

This pattern can be implemented for real-time features like emailing. When a user is registered or when he forgets a password, the user can be notified. The Observer pattern is ideal for creating a loosely coupled design where the state of one object (subject) is monitored by other objects (observers). It's particularly useful for event handling systems, which are crucial in interactive platforms like a microblogging site.

Strategy Pattern

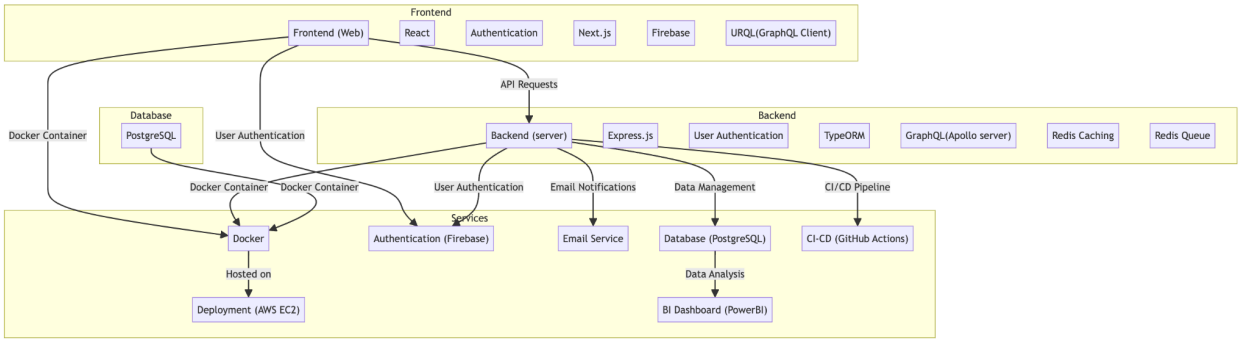
This pattern was used for implementing different algorithms for content recommendations or sorting posts based on various criteria (e.g., popularity, newness). This pattern lets the algorithm vary independently from the clients that use it, thereby promoting algorithmic flexibility and extensibility.

Decorator Pattern

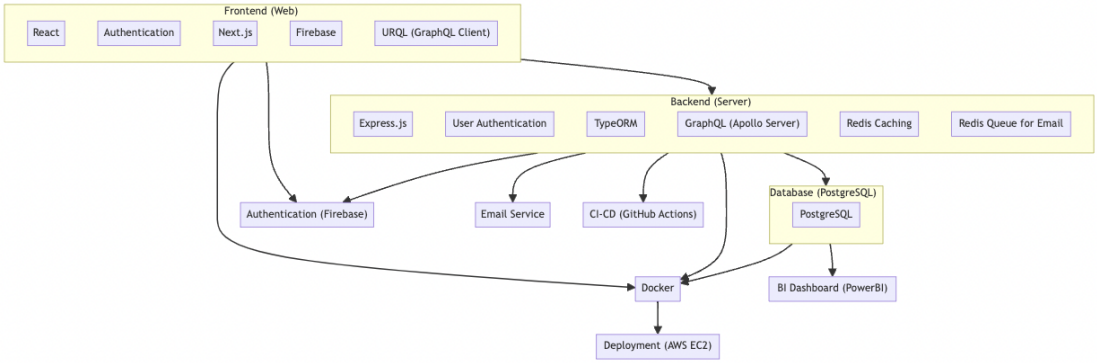
The Decorator pattern was used for adding additional features to objects dynamically. This is particularly useful in UI components using libraries like Chakra UI, where we dynamically altered the appearance or behavior of UI elements. It provides a flexible alternative to subclassing for extending functionality. This is crucial in a UI-rich application where user experience is paramount.

3. Architecture Diagram(s)

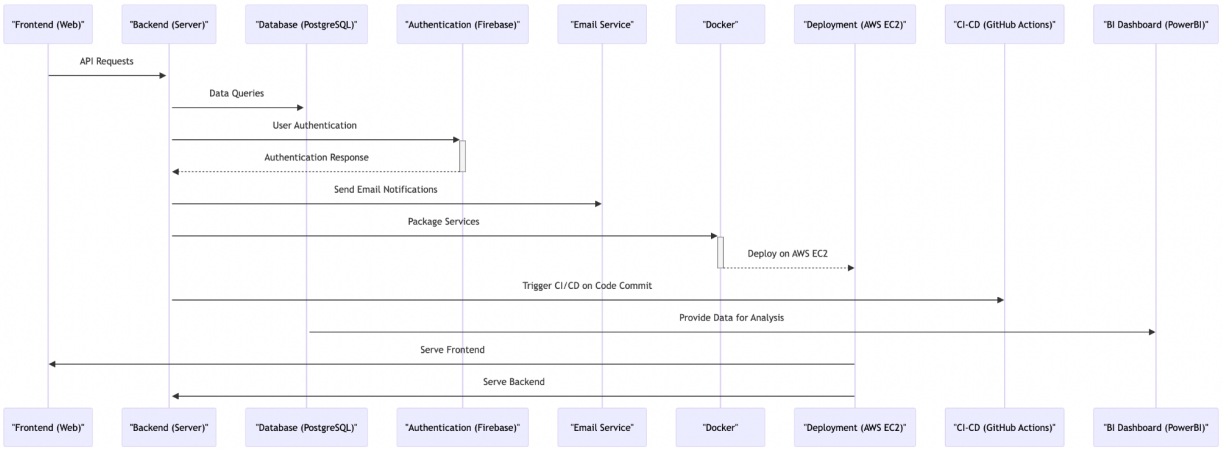
High Level Architecture: Provide a high level diagram showing the overall system architecture.



Component Diagram: Detail the key components/modules of the software and their interactions.



Data Flow Diagram: Illustrate how data moves through the system.



4. Lessons Learned

Challenges and Solutions:

We intentionally sought out new learning experiences by selecting tools we had not used before, including Next.js, URQL (a GraphQL Client), Firebase, TypeORM, PostgreSQL, and Redis (utilized for managing sessions and email queuing). We also explored GraphQL with Apollo Server. To build our proficiency in these technologies, we immersed ourselves in a range of tutorials and effectively applied these tools in our project.

Successes:

The project achieved numerous successes, including the implementation of a microservices architecture with OpenAPI spec-based REST APIs. It features Single Sign-On (SSO) via Firebase, supporting federated sign-ups like Google, and robust security with Two-Factor Authentication using systems like Google Authenticator. The project employs comprehensive CI/CD workflows for efficient development, utilizes a NoSQL database, and enhances performance through Redis caching. Efficient data handling is ensured with integrated queues/streams, and observability is maintained for logs and traces. Additionally, it includes a BI Dashboard using PowerBI for key metrics and data visualization, with all components deployable via Docker.

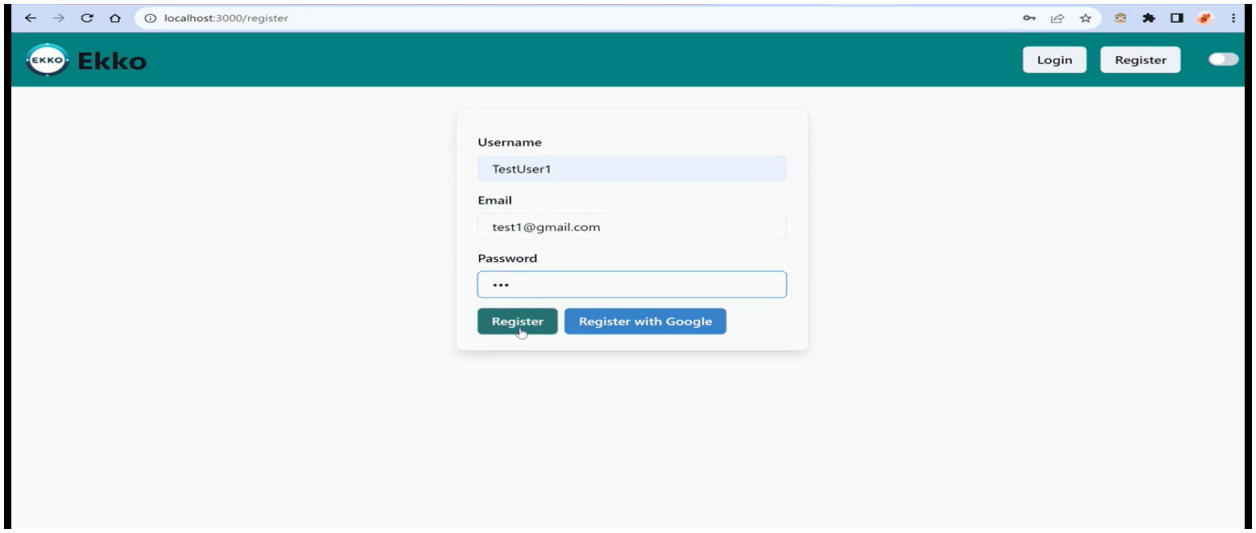
Areas for Improvement:

Limited time was allocated to UI design, with a primary focus on essential project components and deliverables. There is a commitment to enhancing the UI in future iterations.
Additionally, efforts will be directed towards exploring optional deliverables, including automated infrastructure setup and code deployment.

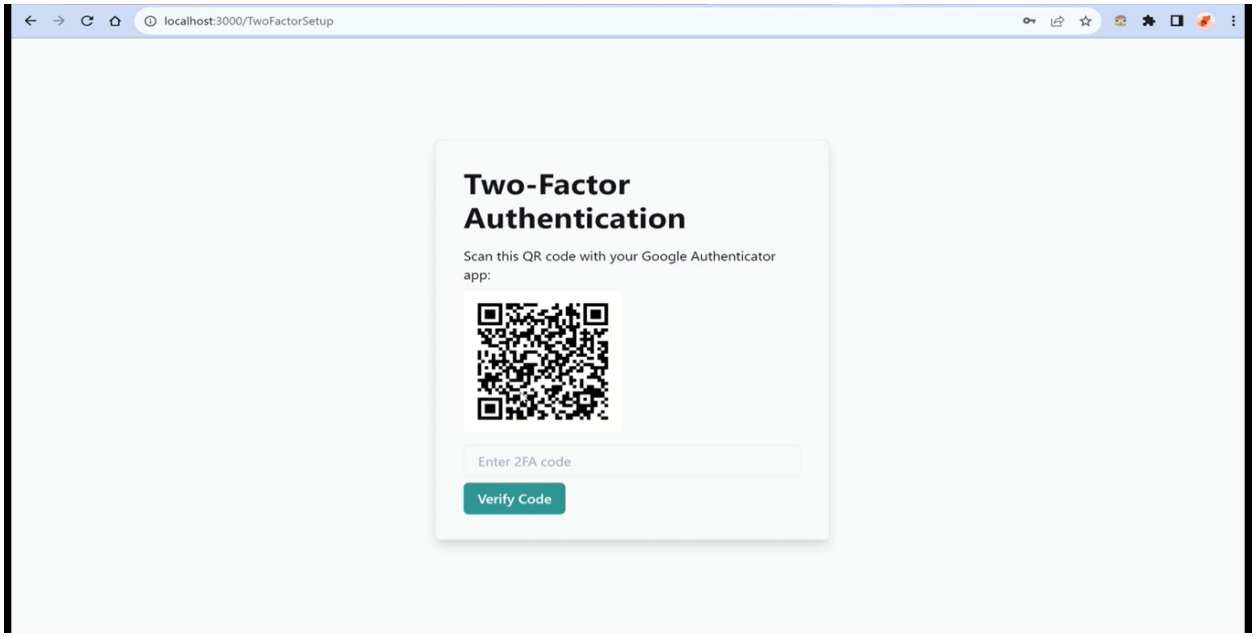
5. Screenshots

Login and Registration demo:
https://drive.google.com/file/d/1e_sPpDh_vD6cu3-OEKuUMspk-f-cWyGW/view?usp=sharing

Register with basic details (email, password, etc.)



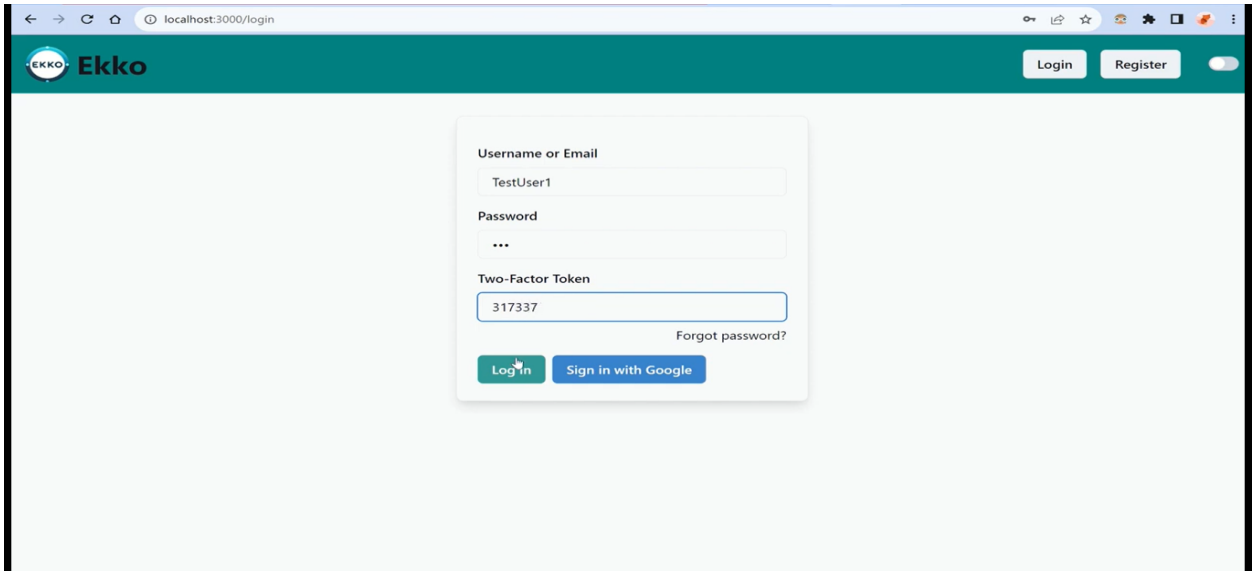
Undergo 2-factor authentication using Google Authenticator



Sign In:

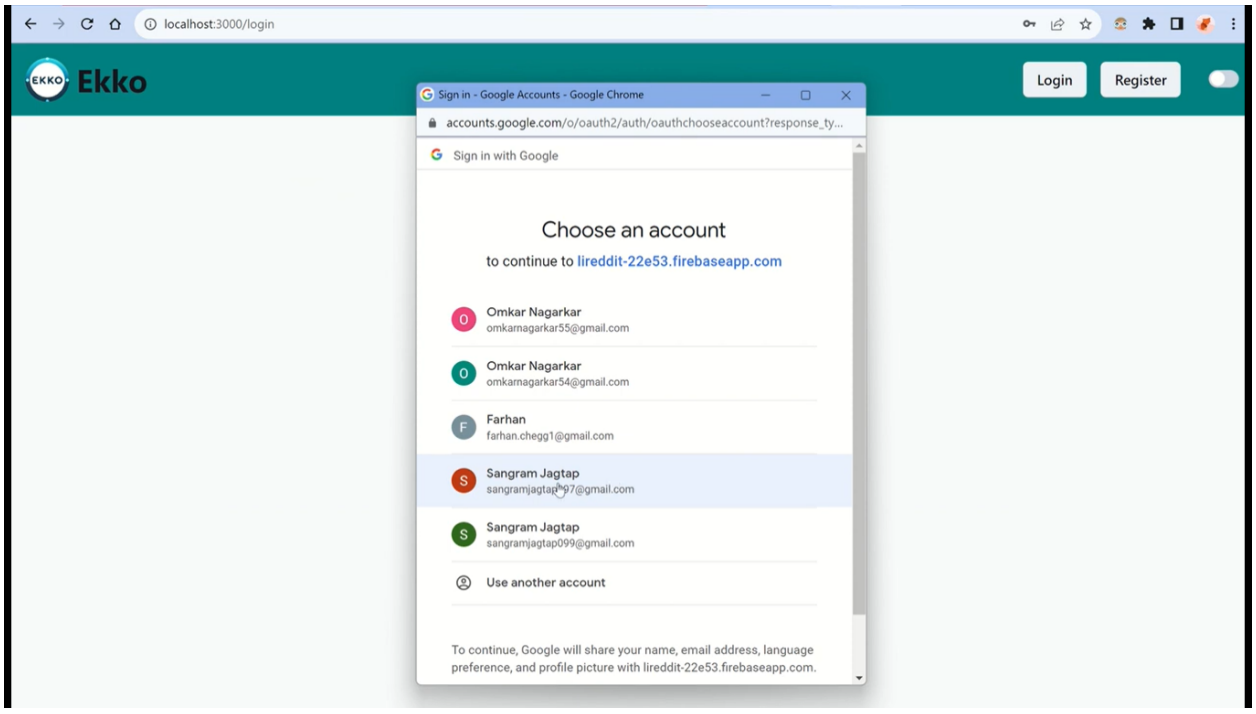
- Enter credentials (username/password)
- Provide token (secret key) from Google Authenticator

- Access granted to home page upon correct token and credentials

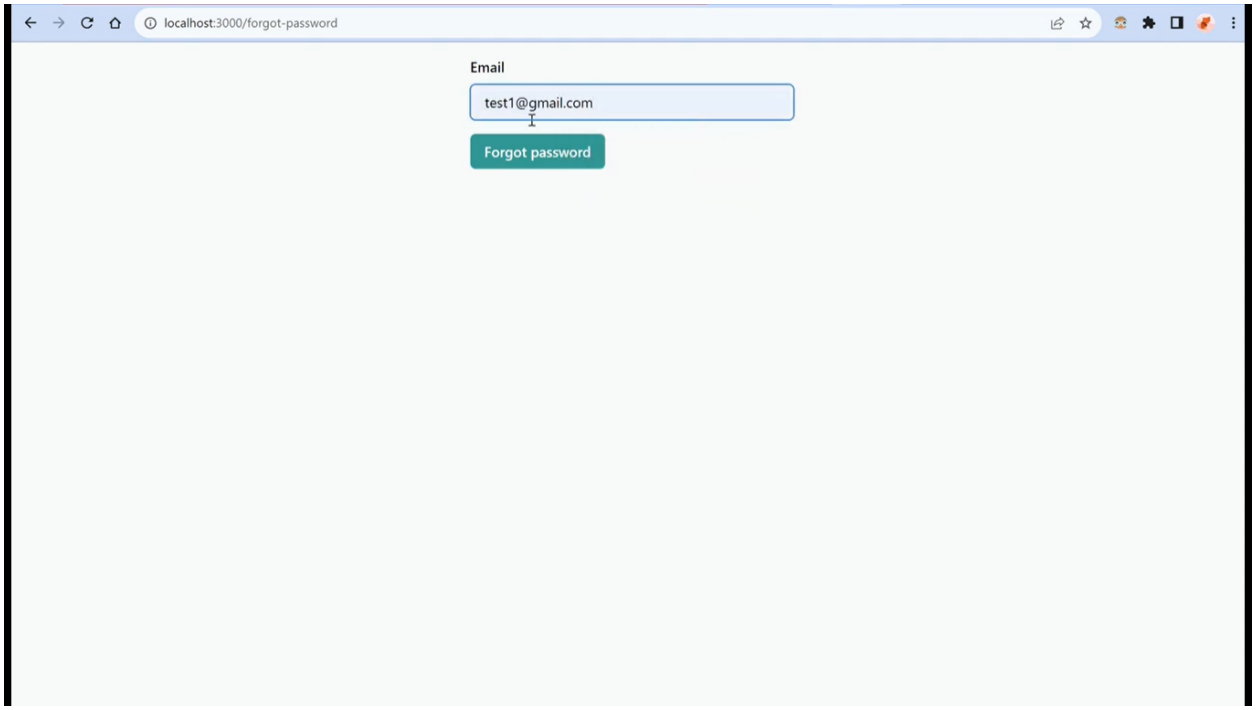


Sign In with Google (using Firebase):

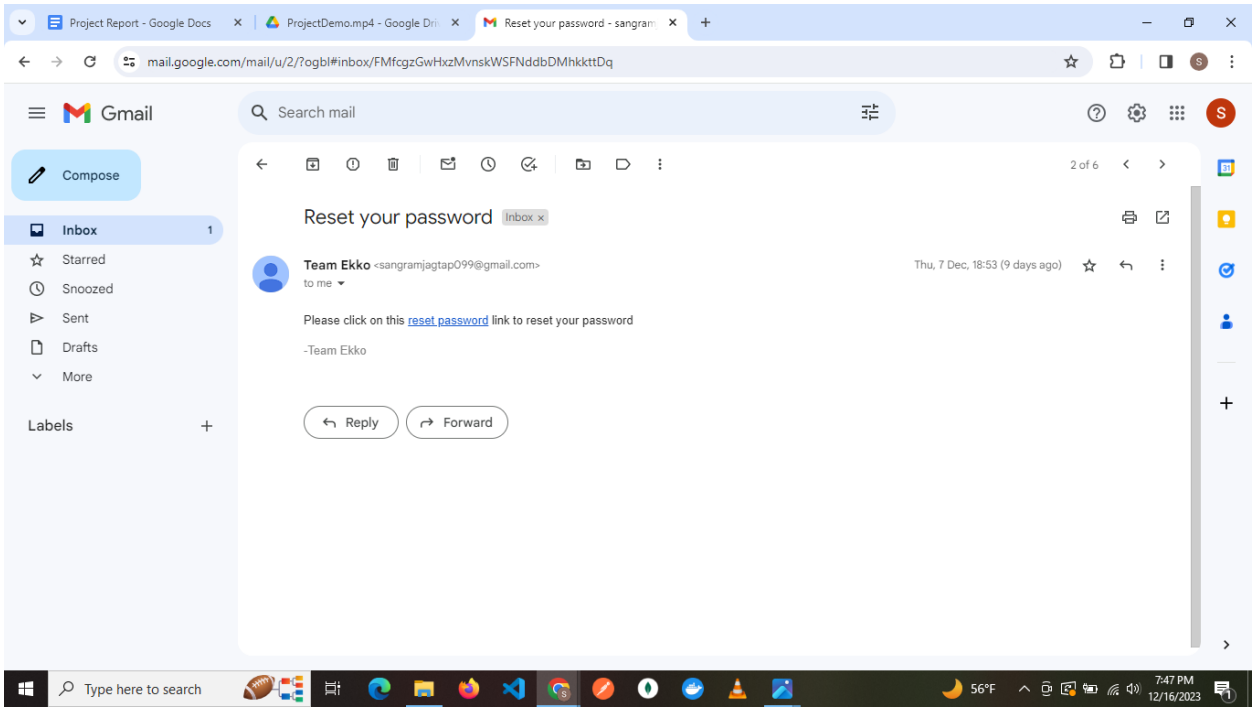
- Log in using Google account
- No additional token required



Forgot Password

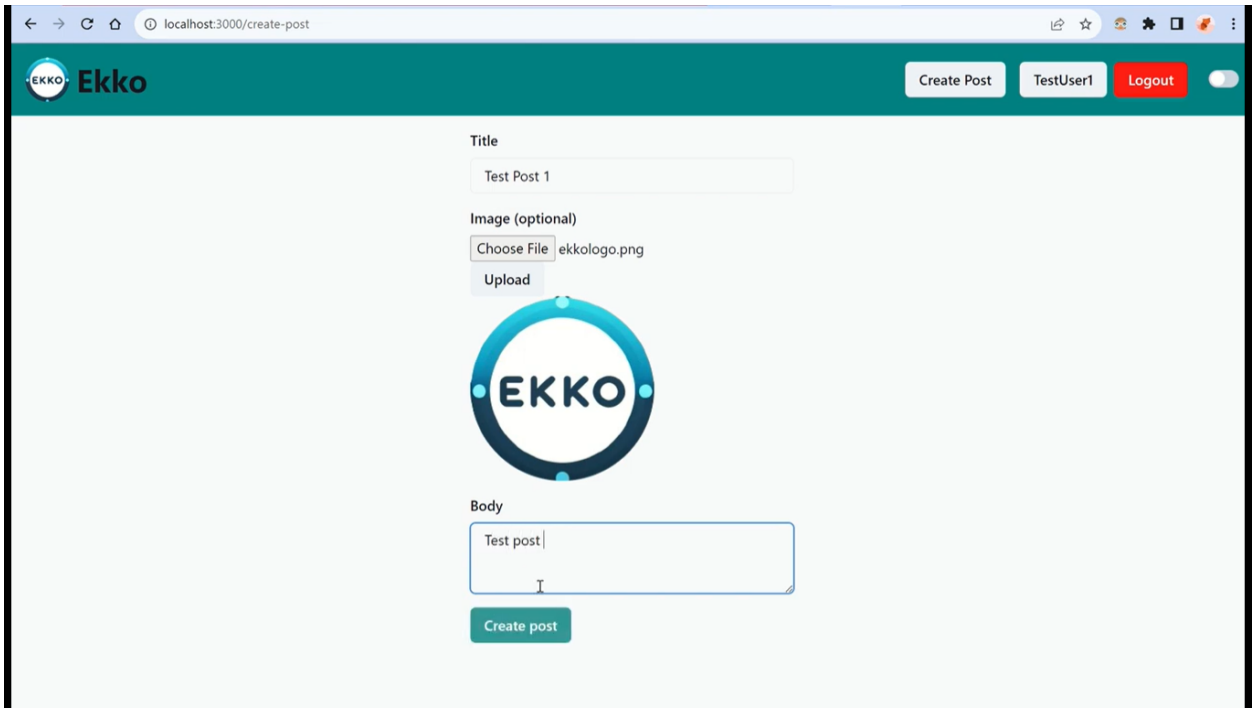


Email confirmation for resetting password

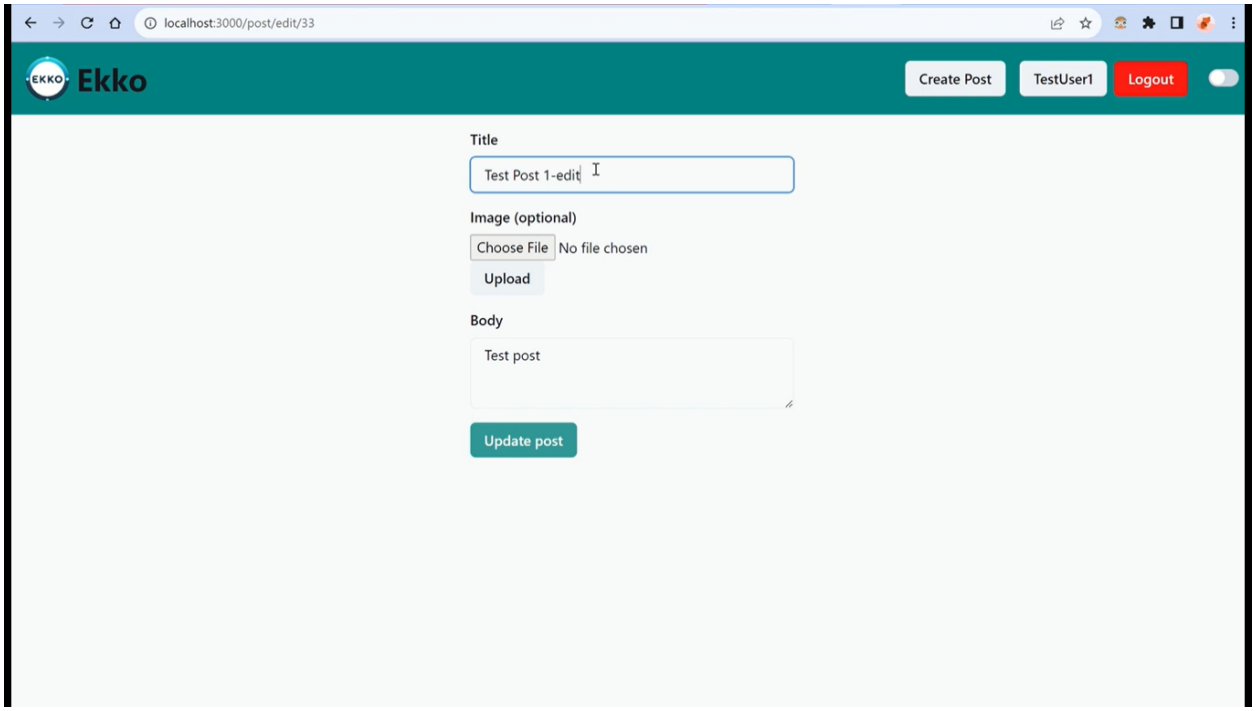


Blogging app functionality demo:
https://drive.google.com/file/d/1e_sPpDh_vD6cu3-OEKuUMspk-f-cWyGW/view?usp=sharing

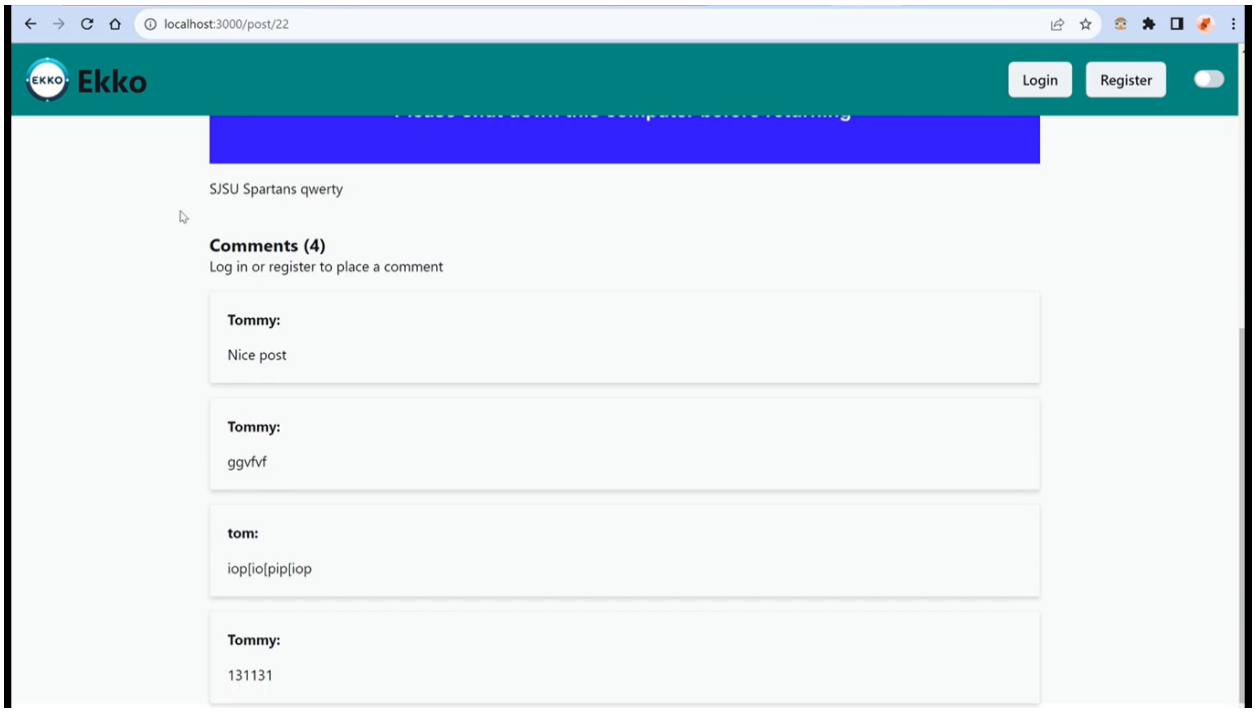
Create Post



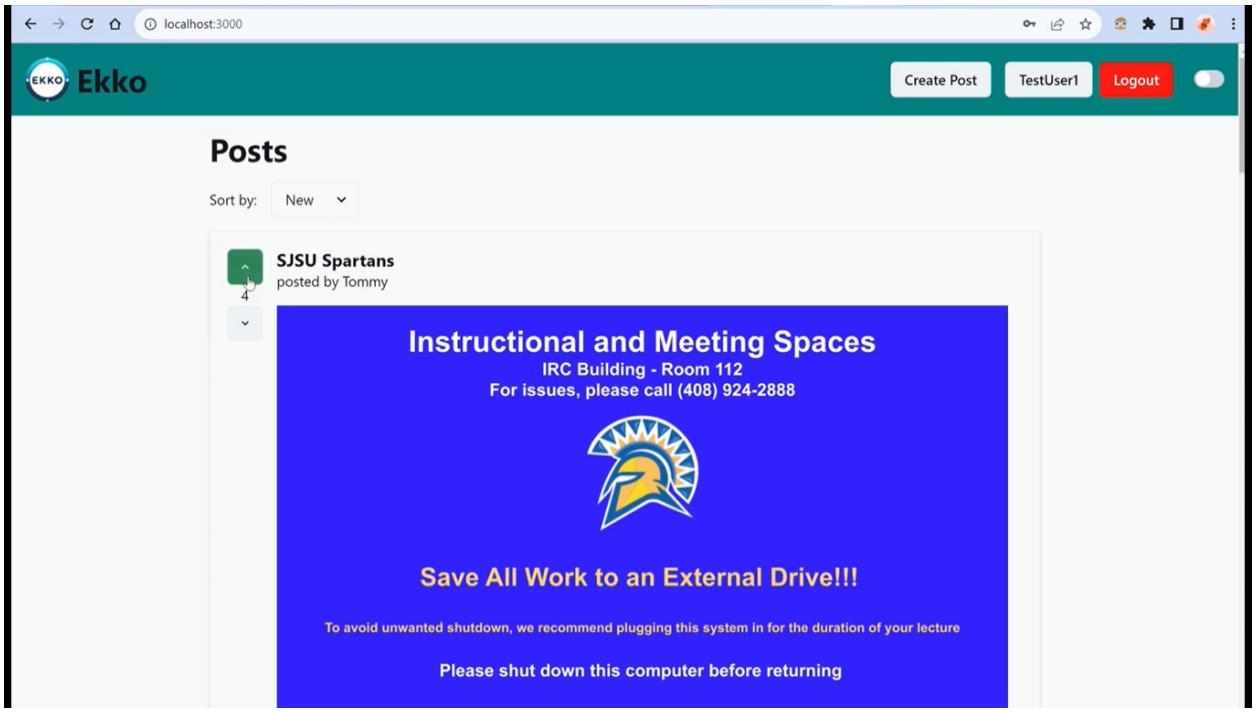
Update Post



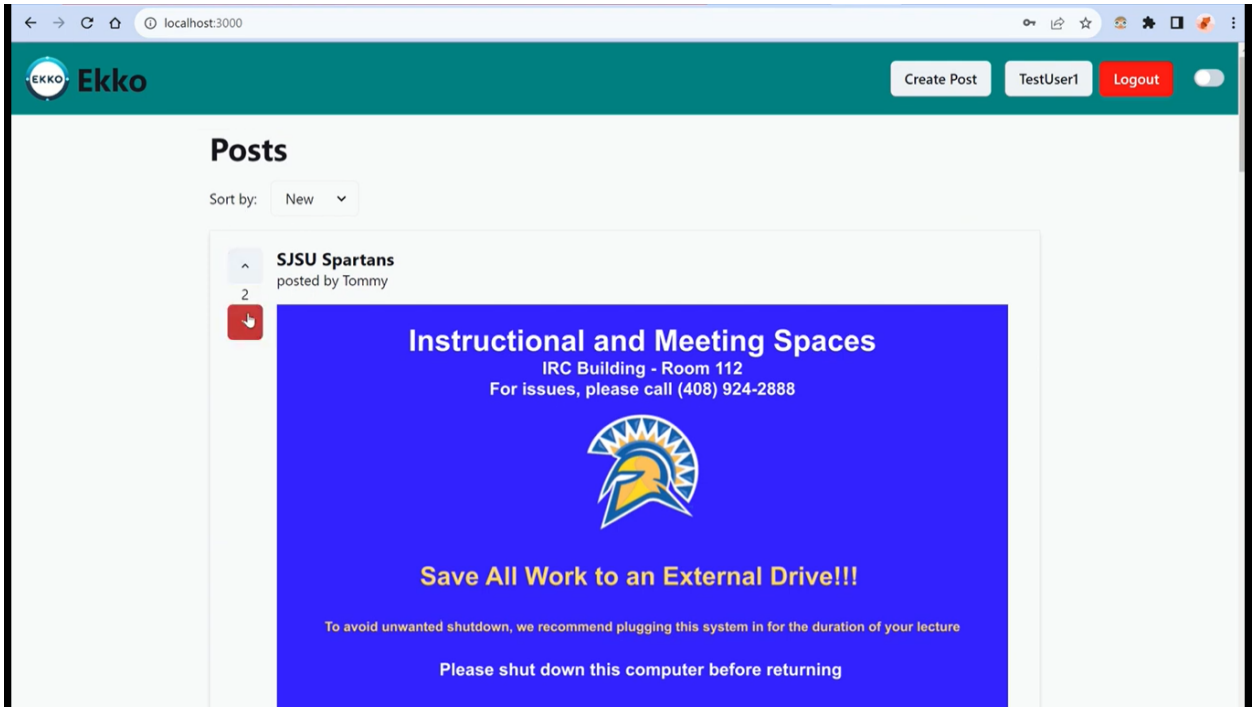
Comments made on the post



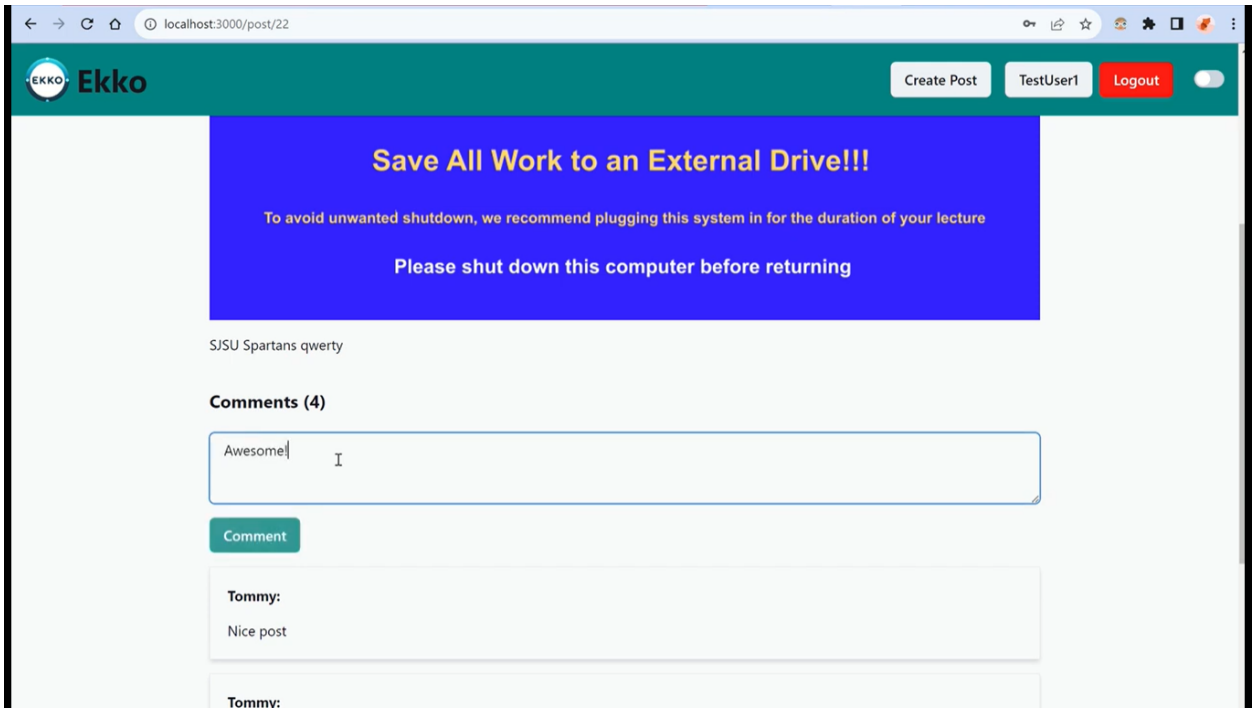
Upvote the post



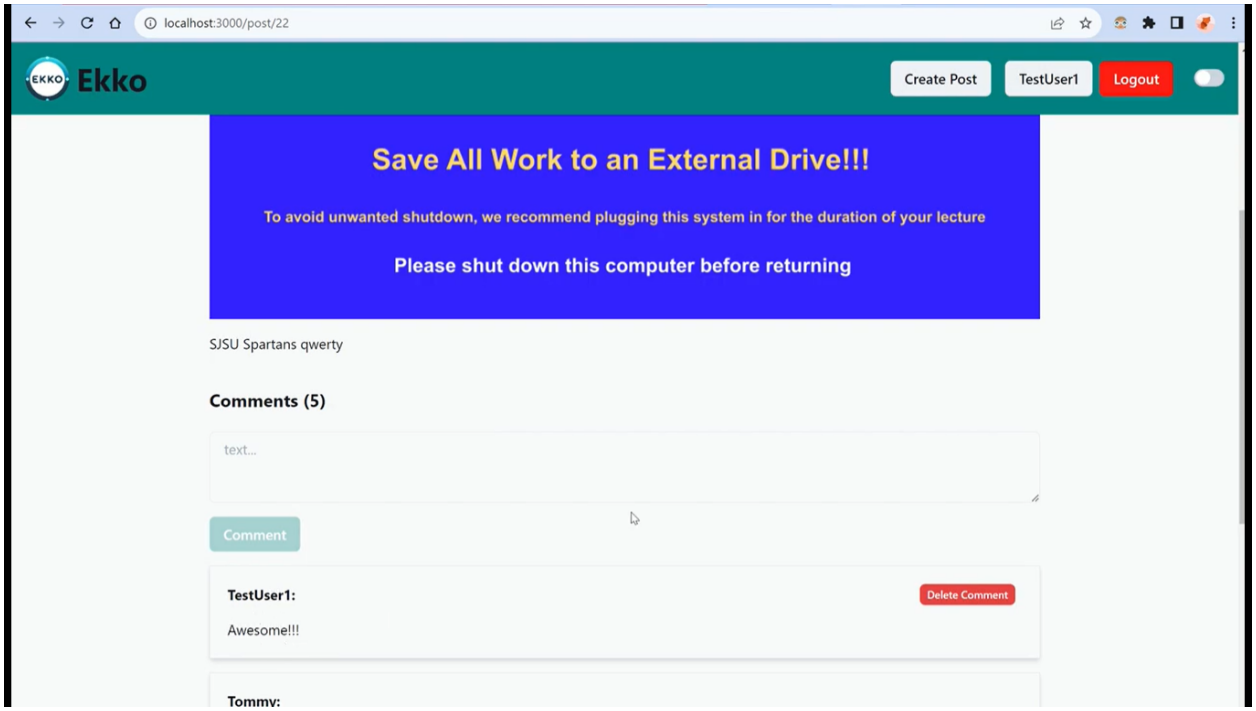
Downvote the post



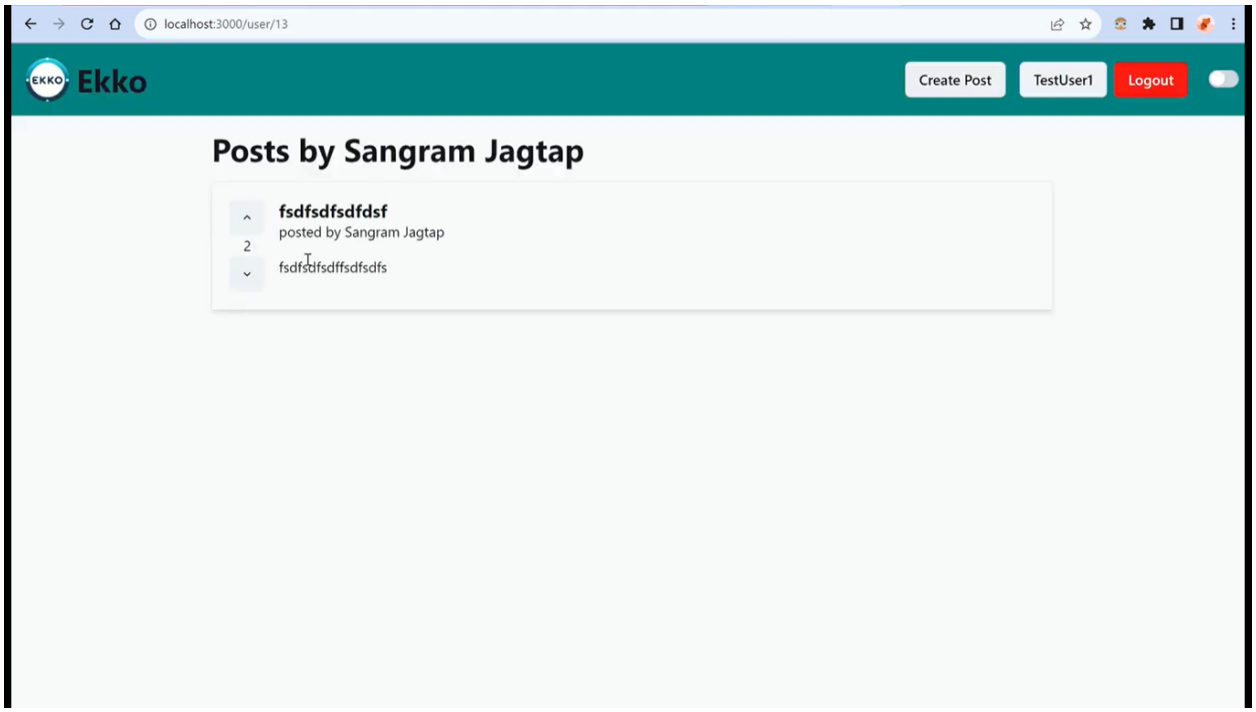
To comment on user's post



Delete the comment on the post

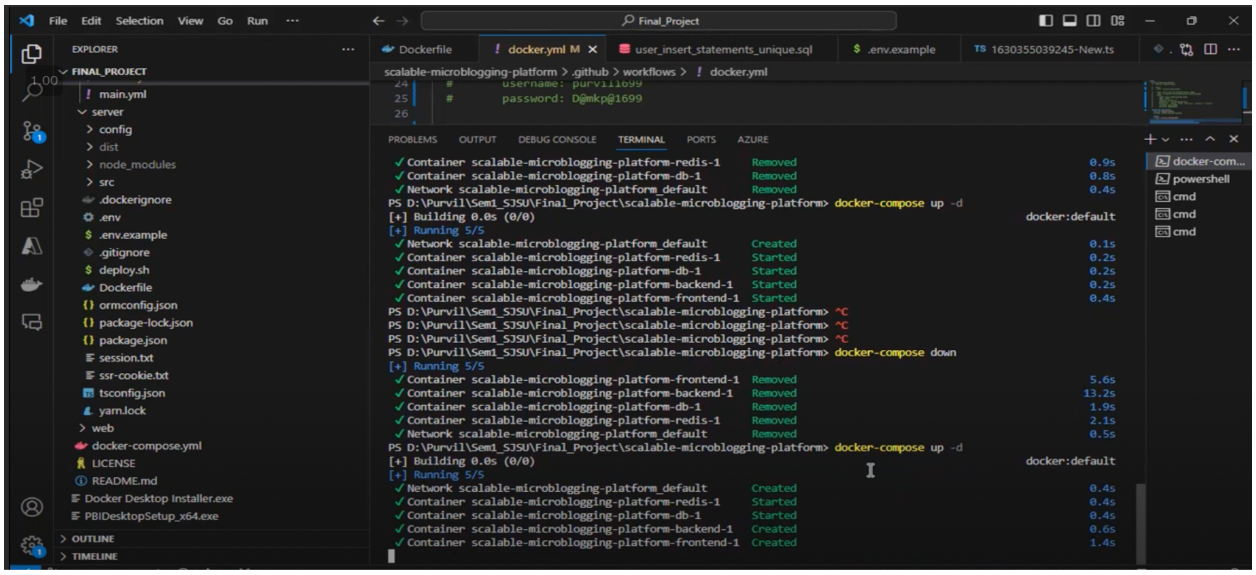


Check posts of other user

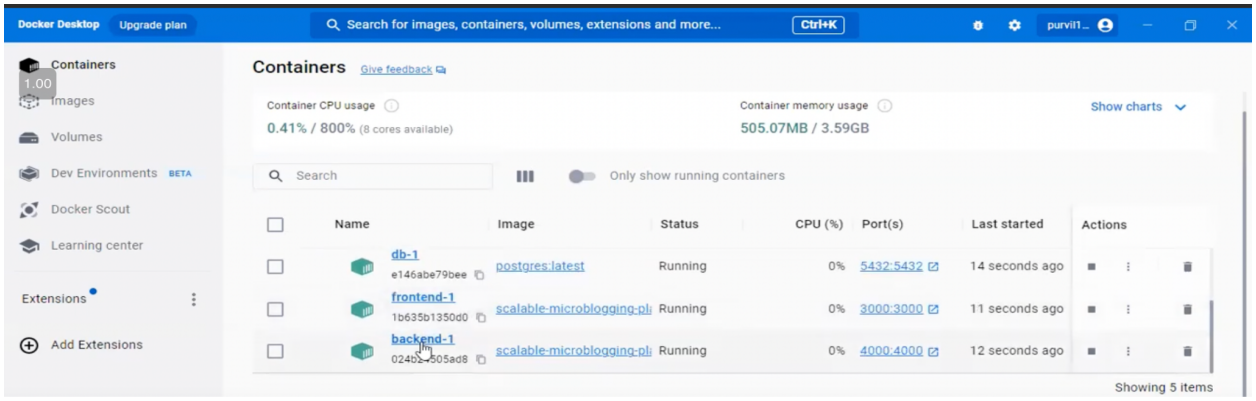


Dockerize the application demo:
https://drive.google.com/file/d/1mHwbFJL-Khp7xYWp4gXpowvCnhNni737/view?usp=drive_link

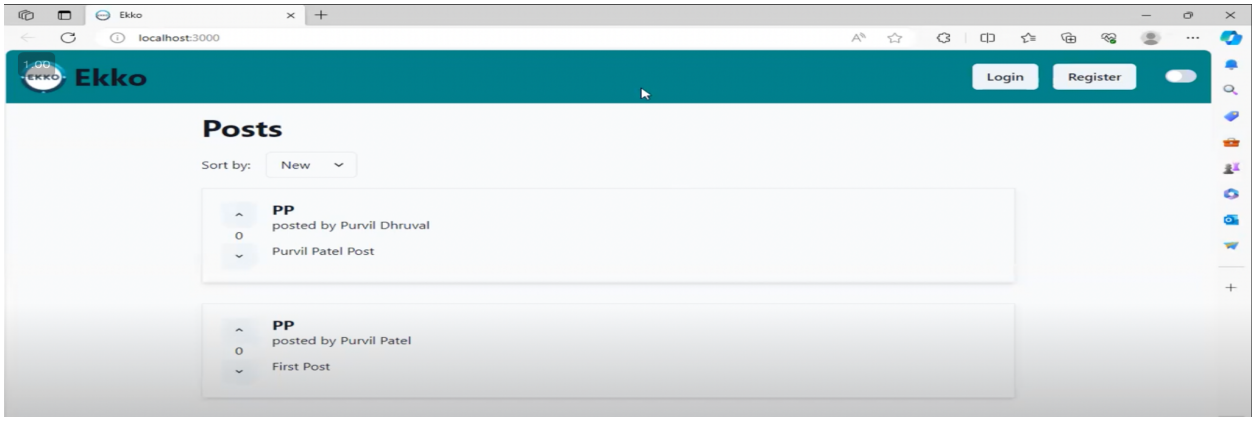
Created the docker containers



Container is created



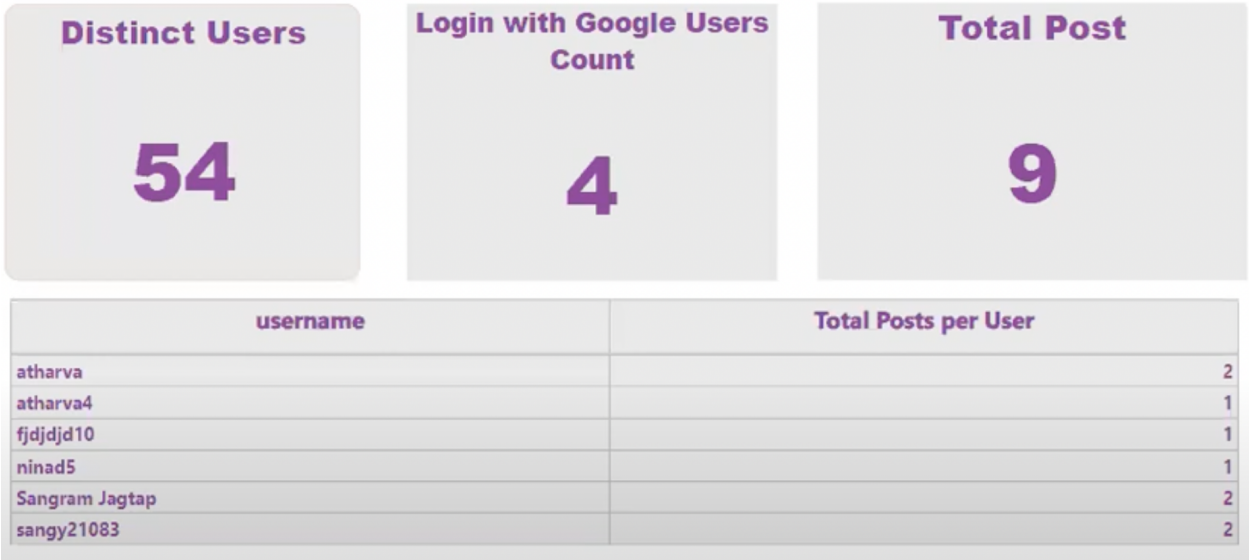
Application running on the docker



BI Dashboard demo:

https://drive.google.com/file/d/1-B4TtropiXxPQflE8GcDjBY7nqyqMf_j/view?usp=drive_link

BI Dashboard integrated with the postgresql



username	Total Posts per User
atharva	2
atharva4	1
fjdjdjd10	1
ninad5	1
Sangram Jagtap	2
sangy21083	2

Scaling Plan

Overview

Current State Analysis:

- Performance: Evaluates how the application currently manages user traffic and data processing, focusing on response times and stability under load.
- Infrastructure: Reviews the existing setup for scalability potential, examining if the current architecture can support growth.
- Resource Efficiency: Analyzes current resource use, considering both the effectiveness in handling the load and the cost implications.

Scaling for Different User Bases

Here's a more detailed breakdown for each user base level:

100k Users:

- Infrastructure: Utilize moderate cloud services (e.g., AWS, Google Cloud) with a couple of medium-sized instances for handling web traffic and a relational database like PostgreSQL. Implement basic load balancing to distribute traffic and reduce server load.
- Estimated Cost: Approximately $300 - $800 per month, considering moderate server usage, database operations, and some data storage and transfer. Costs can vary based on the specific services and resources chosen.
- Challenges: Ensuring efficient database operations and managing increased traffic. Solutions include optimizing database queries, implementing caching strategies (like Redis), and carefully monitoring resource usage.

1M Users:

- Infrastructure: Upgrade to more robust cloud services with auto-scaling features to handle traffic spikes, and a distributed database system for improved data management. Advanced load balancing techniques will be necessary to efficiently distribute the increased traffic.
- Estimated Cost: Ranges from $2,000 to $5,000 per month, factoring in higher costs for enhanced server capabilities, more complex database management, and potential use of additional services like CDN for content delivery.
- Challenges: Managing more significant traffic volumes and maintaining database performance. Strategies include more aggressive caching, database sharding, and possibly introducing a microservices architecture for better scalability.

100M Users:

- Infrastructure: High-capacity cloud solutions with global CDN integration for faster content delivery worldwide, microservices architecture for better modular scaling, and extensive database scaling to handle large volumes of data.

- Estimated Cost: Expected to be around $20,000 to $50,000 per month, accounting for a substantial increase in server resources, sophisticated database solutions, and the extensive use of CDNs and other advanced technologies.
- Challenges: Dealing with massive traffic and data volume, requiring highly scalable and reliable infrastructure. Solutions involve employing advanced data storage and retrieval techniques, extensive use of CDNs, and potentially leveraging AI for predictive scaling and resource optimization.

1B Users:

- Infrastructure: Enterprise-level cloud infrastructure capable of handling an enormous scale, utilizing massive parallel processing databases for quick data handling, and AI-driven resource management to optimize resource allocation in real-time.
- Estimated Cost: Likely in the range of $100,000 to $500,000 per month, given the need for extremely high-capacity and resilient infrastructure, sophisticated technology solutions, and possibly custom developments.
- Challenges: Ensuring system stability and scalability at an unprecedented scale. This involves utilizing cutting-edge technology, possibly exploring options like quantum computing for data processing, and continually innovating to stay ahead of scalability needs.

Long-Term Scalability

FutureProofing Strategies:

- Modular Architecture: Adopt a modular architecture like microservices, which allows for easier scaling and updating of individual components without affecting the entire system.
- Elasticity and Auto-Scaling: Implement elastic cloud solutions that automatically scale resources up or down based on real-time demand, ensuring cost-effectiveness and performance.
- Continuous Monitoring and Optimization: Regularly monitor performance metrics and optimize the system accordingly to handle growth, including refining database structures, updating caching strategies, and enhancing load balancing techniques.

Technology Considerations:

- Emerging Technologies: Stay abreast of emerging technologies like serverless architectures, edge computing, and potential advancements in AI and machine learning for predictive scaling.
- Data Management Innovations: Prepare for advancements in database technologies, such as new NoSQL solutions or improvements in distributed database systems, to handle large-scale data efficiently.
- Sustainability and Security: Ensure that scaling strategies consider not only performance but also sustainability and security aspects, incorporating green computing practices and robust security protocols.

Conclusion

In summary, scaling an application to support user bases from 100k to 1 billion requires a dynamic and forward-thinking approach. Adopting scalable and modular architectures, leveraging cloud-based solutions with elasticity and auto-scaling, and continuously monitoring and optimizing the system is key. Future-proofing strategies are crucial, involving staying updated with technological advancements, focusing on sustainable and secure scaling practices, and being prepared to incorporate emerging technologies and data management innovations. As the application grows, it's essential to balance scalability, cost, and performance, ensuring that the platform remains robust, efficient, and adaptable to future challenges and opportunities.