Software Design Document (SDD) document

CS673 Team-5 Project

TerrierConnect

Introduction

The Software Design Document (SDD) for TerrierConnect provides a comprehensive blueprint for developing and implementing a feature-rich, scalable, and user-centric web application. As a social networking platform tailored for seamless user engagement and interaction, TerrierConnect is designed with a robust client-server-database architecture, enhanced by strategic optimizations to maximize performance and usability. Leveraging Docker's containerization capabilities, the project aims to simplify deployment and maintenance, allowing for efficient use of server resources. This document outlines the architectural choices, technologies, and design principles, such as the three-tier architecture, that were strategically selected to meet the requirements while ensuring a high standard of security, scalability, and maintainability. Each section delves into specific layers of the system, including the frontend framework, backend development, database design, API functions, and reverse proxy configurations, forming the foundational structure of a reliable, modern web application.

Software Architecture

The project we delivered to the customer is a web application system based on the traditional client-server-database three-tier architecture. However, unlike the simple client-server-database architecture, we have made some optimizations to the software architecture based on the customer's usage scenario.

Docker virtualization technology

Since we hope that the computing resources of the customer's server can be utilized as much as possible, so that customers can run as many services as possible on their servers, we believe that using virtualization technology is a solution that can provide sufficient flexibility. In the use of virtualization technology, we chose to use Docker to run the backend software system. Since Docker can isolate containers in the virtual environment from each other and provide network communication so that each container can communicate with each other like multiple servers, we can configure the network connection and file system between containers by just editing the Dockerfile file.

Since Docker provides a convenient form of container deployment using the docker-compose configuration file, after writing a complete configuration file, customers can start running all services by just running a single line of command in the terminal. This can greatly reduce the difficulty of maintenance.

React frontend

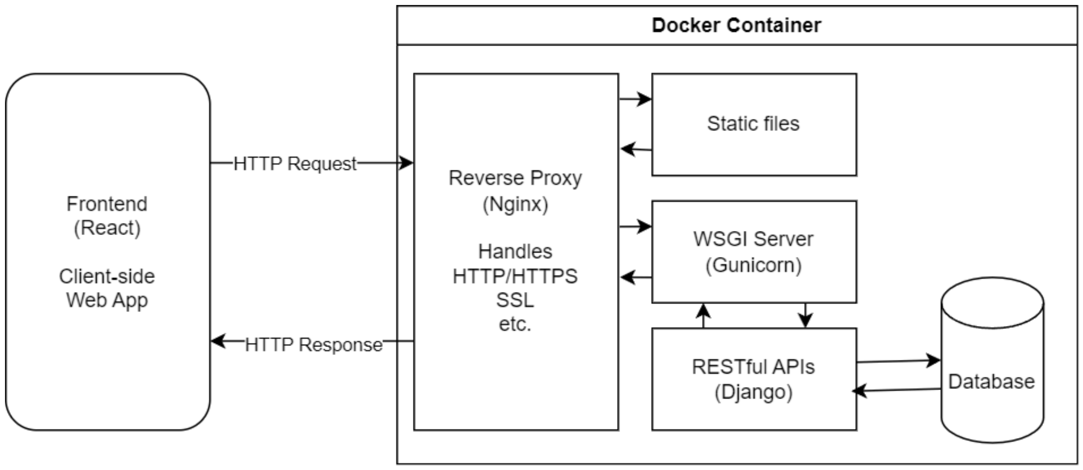
The front-end of the project uses the React framework, which is one of the most popular frontend frameworks at present. This means that its community will have stable security maintenance and functional updates for the entire framework. When using the client part, users only need to interact with components. React will communicate with the backend server through the corresponding REST API. After obtaining the data returned by the server, React will render the data returned by the server in the browser and present it to the user.

Django backend

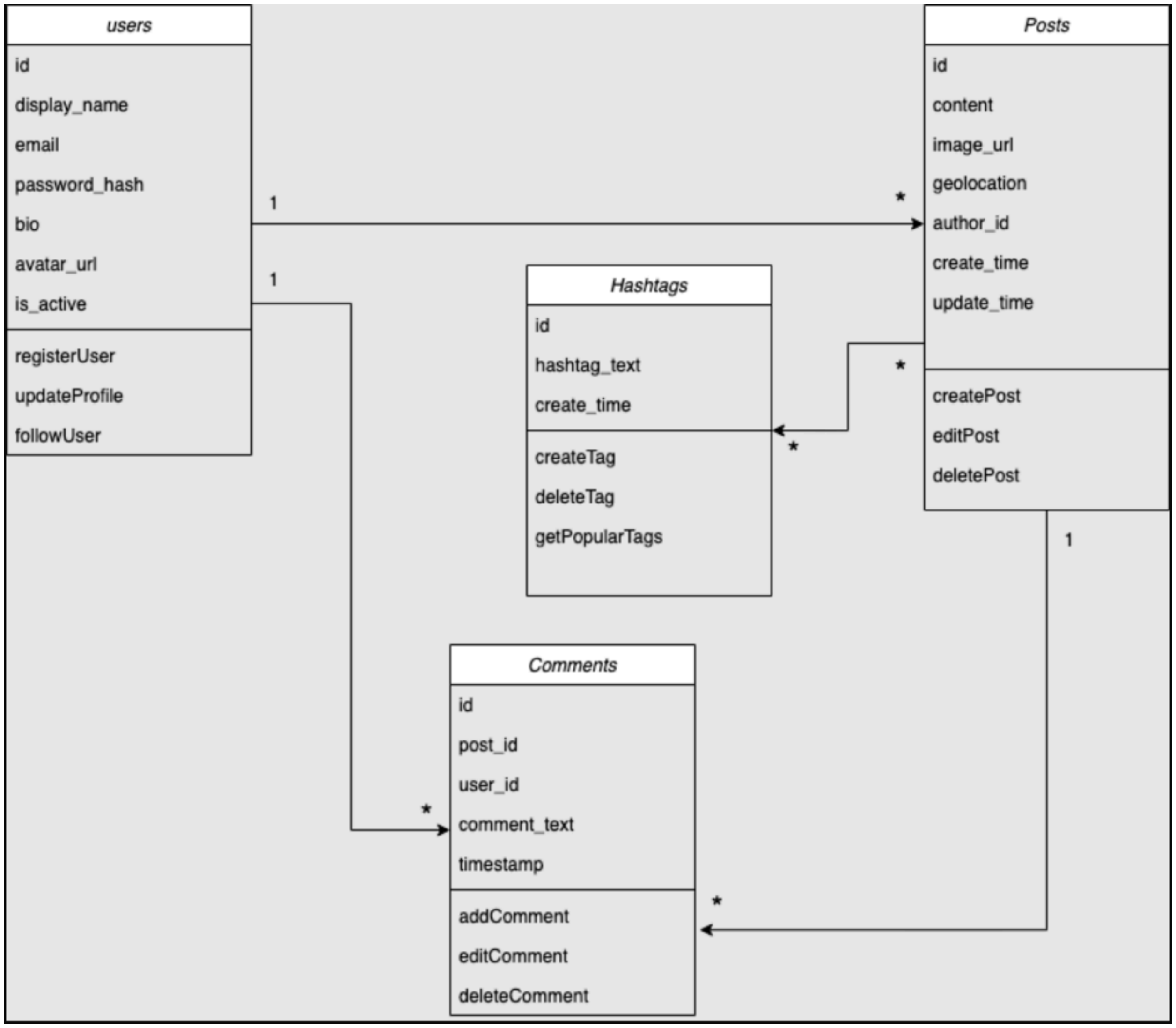
Django is the most commonly used web framework based on Python. It is very suitable for developers who need to quickly develop web projects that can be used in industrial environments. At the same time, it is also very scalable, making it suitable for most web application scenarios. In our project, the Django backend is connected to the PostgreSQL database.

Reverse Proxy

Due to the lightweight feature of the Django framework, it does not have a built-in ability for processing HTTP traffic. In Django, this part of the function is implemented by the reserved WSGI protocol. Django can process HTTP requests by connecting to a web server that implements the WSGI protocol. In the industry, the most commonly used WSGI server is Gunicorn. However, since Gunicorn only implements the functions of processing HTTP and forwarding HTTP requests to the connected web framework, in actual use, we usually deploy a reverse proxy in front of Gunicorn. The commonly used option is Nginx. In our project, Nginx can not only be used as a reverse proxy to process HTTP requests from clients, but also to serve static files. In some cases, it can also be used as a load balancer to help the stability of the service.



Class Diagram



Entities and Attributes

- **Users**
  - **Attributes**:
    - id: Primary key, unique identifier for each user.
    - display_name: The name displayed on the user's profile.
    - email: User's email address, used for registration and login.
    - password_hash: Hashed password for secure authentication.
    - bio: Short biography or description about the user.
    - avatar_url: URL of the user's profile picture.
    - is_active: Boolean flag indicating if the user account is active.

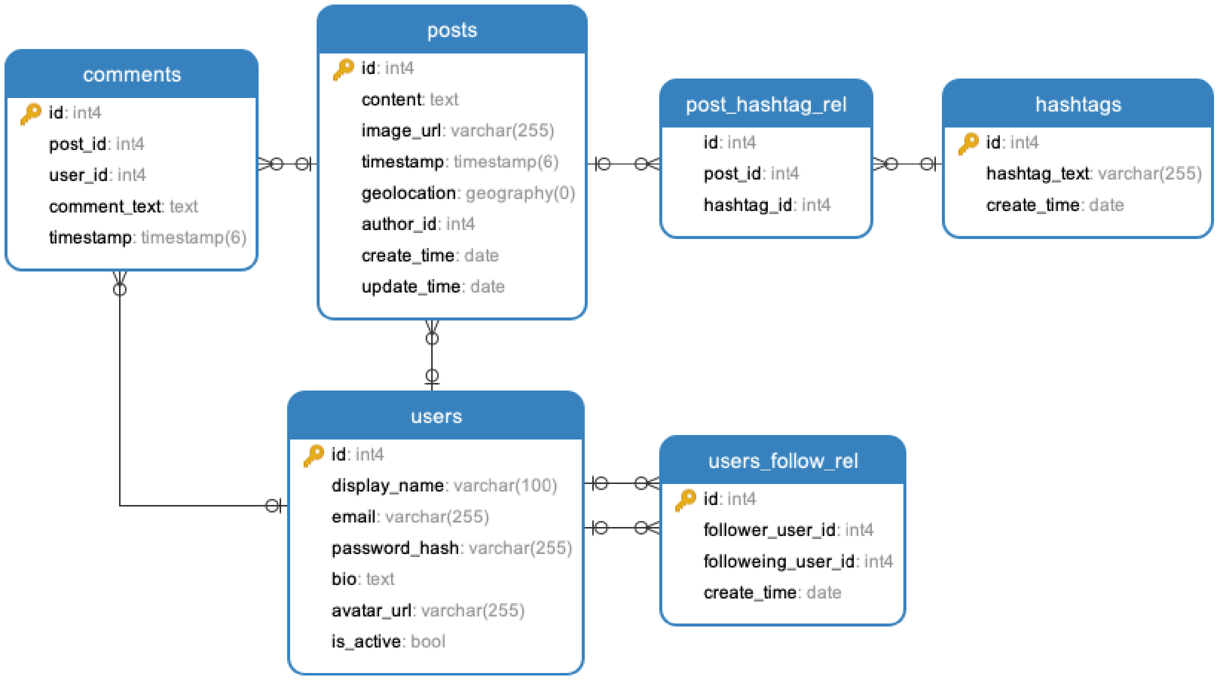
    - **Functions**:
        - registerUser(): Allows a new user to register on the platform.
        - updateProfile(): Updates the user's profile information.
        - followUser(): Allows a user to follow another user, establishing a following relationship.
- **Posts**
    - **Attributes**:
        - id: Primary key, unique identifier for each post.
        - content: Text content of the post.
        - image_url: URL of any image associated with the post.
        - geolocation: Geographical coordinates for location tagging.
        - author_id: Foreign key linking to the User who created the post.
        - create_time: Date of post creation.
        - update_time: Date of the last update to the post.
    - **Functions**:
        - createPost(): Creates a new post with optional content, image, and location.
        - editPost(): Edits the content or image of an existing post.
        - deletePost(): Deletes a specified post.
- **Comments**
    - **Attributes**:
        - id: Primary key, unique identifier for each comment.
        - post_id: Foreign key linking to the Post on which the comment is made.
        - user_id: Foreign key linking to the User who made the comment.
        - comment_text: The text content of the comment.
        - timestamp: Date and time when the comment was created.
    - **Functions**:
        - addComment(): Adds a new comment to a post.
        - editComment(): Edits the text of an existing comment.
        - deleteComment(): Deletes a specified comment.
- **Hashtags**
    - **Attributes**:
        - id: Primary key, unique identifier for each hashtag.
        - hashtag_text: Text content of the hashtag.
        - create_time: Date when the hashtag was created.
    - **Functions**:
        - createTag(): Creates a new hashtag.
        - deleteTag(): Deletes a specified hashtag if no longer in use.
        - getPopularTags(): Retrieves frequently used tags within a specified time frame.

Relationships

1. **User to Post** (1-to-Many):
    - **Description**: Each User can create multiple Posts, but each Post is authored by a single User.
    - **Notation**: Represented with 1 near User and * near Post.
2. **User to Comment** (1-to-Many):
    - **Description**: Each User can create multiple Comments, but each Comment is associated with only one User.
    - **Notation**: 1 on the User side and * on the Comment side.
3. **Post to Comment** (1-to-Many):
    - **Description**: Each Post can have multiple Comments, but each Comment is linked to a single Post.
    - **Notation**: 1 near Post and * near Comment.
4. **Post to Hashtag** (Many-to-Many):
    - **Description**: Each Post can have multiple Hashtags, and each Hashtag can be associated with multiple Posts.
    - **Notation**: * near both Post and Hashtag classes to represent the many-to-many relationship.

Database Design

ER diagram



Tables

comments:

- Stores information about comments made on posts.
- Attributes:
  - id: Primary key, unique identifier for each comment.
  - post_id: Foreign key, links to the post on which the comment is made.
  - user_id: Foreign key, links to the user who made the comment.
  - comment_text: The text content of the comment.
  - timestamp: Date and time when the comment was created.

posts:

- Stores information about individual posts made by users.
- Attributes:
  - id: Primary key, unique identifier for each post.
  - content: Text content of the post.
  - image_url: URL of any image associated with the post.
  - geolocation: Geographical coordinates for location tagging.
  - author_id: Foreign key, links to the user who created the post.
  - can_edit: Boolean flag indicating if the post is editable by the author.
  - create_time: Date of post creation.
  - update_time: Date of the last update to the post.

post_hashtag_rel:

- Manages the many-to-many relationship between posts and hashtags.
- Attributes:
  - id: Primary key, unique identifier for each record in the relation.
  - post_id: Foreign key, links to the associated post.
  - hashtag_id: Foreign key, links to the associated hashtag.

hashtags:

- Stores information about hashtags used in posts.
- Attributes:
  - id: Primary key, unique identifier for each hashtag.
  - hashtag_text: Text of the hashtag.
  - create_time: Date when the hashtag was created.

users:

- Stores information about registered users of the platform.

- Attributes:
  - id: Primary key, unique identifier for each user.
  - display_name: Name displayed on the user’s profile.
  - email: Email address of the user.
  - password_hash: Hashed password for secure authentication.
  - bio: Short biography or description provided by the user.
  - avatar_url: URL of the user's profile picture.
  - is_active: Boolean flag indicating if the user account is active.

users_follow_rel:

- Manages the follower-following relationships between users.
- Attributes:
  - id: Primary key, unique identifier for each record in the relation.
  - follower_user_id: Foreign key, links to the user who is following.
  - following_user_id: Foreign key, links to the user who is being followed.
  - create_time: Date when the following relationship was established.

Functions (APIs) Design

1. **Register**
   - Input:
     - Required: Username, BU email, password
     - Optional: Bio, avatar
   - Output:
     - Success or failure.
   - Acceptance Criteria:
     - The email address needs to be confirmed to be valid and whether it is a BU email address
     - Password should be encrypt
     - Required fields should be checked

2. **Login**
   - Input:
     - Required: BU email, password
   - Output:
     - Success or failure.
     - User token or session
     - User id
   - Acceptance Criteria:
     - Token or session should be returned to the Frontend for use on other requests

        - User id should be return to Frontend for getting user information

3. **GetUserInfoById**
    - Input:
        - Required: User id
    - Output:
        - Success or failure
        - User information except password
    - Acceptance Criteria:
        - Token or session authentication needed
        - Return all information of the user except password

4. **ListPosts (all posts or following)**
    - Input:
        - Required: Flag (0 or 1)
        - Optional: Hashtags
    - Output:
        - All posts order by time if Flag = 0
        - Following people's posts order by time if Flag = 1
    - Acceptance Criteria:
        - Token or session authentication needed
        - Return list of posts correctly based on Flag and Hashtag

5. **AddPost**
    - Input:
        - Required: Author id, content
        - Optional: Image, geolocation, hashtags
    - Output:
        - Success or failure
    - Acceptance Criteria:
        - Token or session authentication needed
        - Implement image uploading functionality
        - Able to include geolocation information
        - Able to create and link hashtags to the post
        - Record the creation time in db

6. **UpdatePost**
    - Input:
        - Required: Post id
        - Optional: Post information (content, image, location), hashtags
    - Output:
        - Success or failure

- Acceptance Criteria:
    - Token or session authentication needed
    - Only author can update their post
    - Able to create and link hashtags to the post
    - Record the update time in db

7. **DeletePost**
    - Input:
        - Required: Post id
    - Output:
        - Success or failure
    - Acceptance Criteria:
        - Token or session authentication needed
        - Only author can delete their post
        - Also need to delete images related to the post

8. **GetPostDetail**
    - Input:
        - Required: Post id
    - Output:
        - Success or failure
        - Post information
    - Acceptance Criteria:
        - Token or session authentication needed

9. **Follow**
    - Input:
        - Required: User id
    - Output:
        - Success or failure
    - Acceptance Criteria:
        - Token or session authentication needed

10. **Unfollow**
    - Input:
        - Required: User id
    - Output:
        - Success or failure
    - Acceptance Criteria:
        - Token or session authentication needed

11. **AddHashtags**

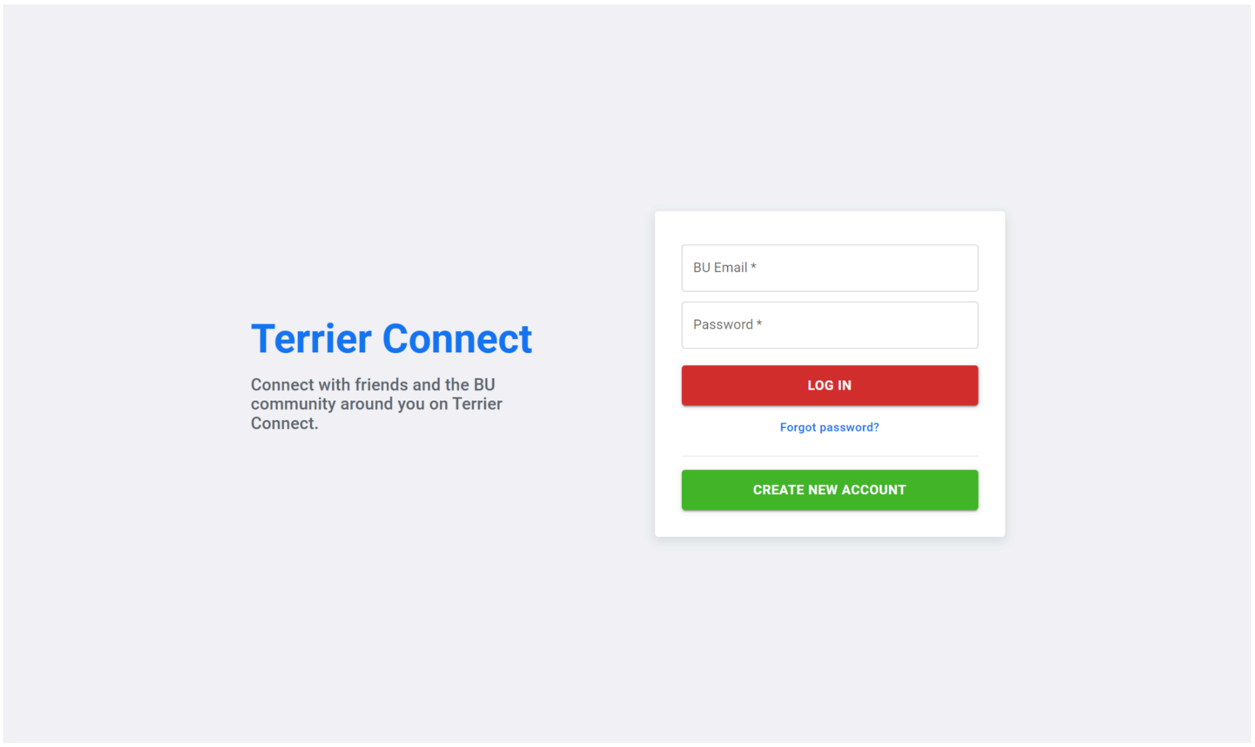
- Input:
  - Required: Hashtags
- Output:
  - Success or failure
- Acceptance Criteria:
  - Token or session authentication needed

12. FullTextSearch

- Input:
  - Required: Content text
- Output:
  - Success or failure
  - Posts list
- Acceptance Criteria:
  - Token or session authentication needed
  - Reasonable implementation in the backend and database
  - Scalable for other information searching in future

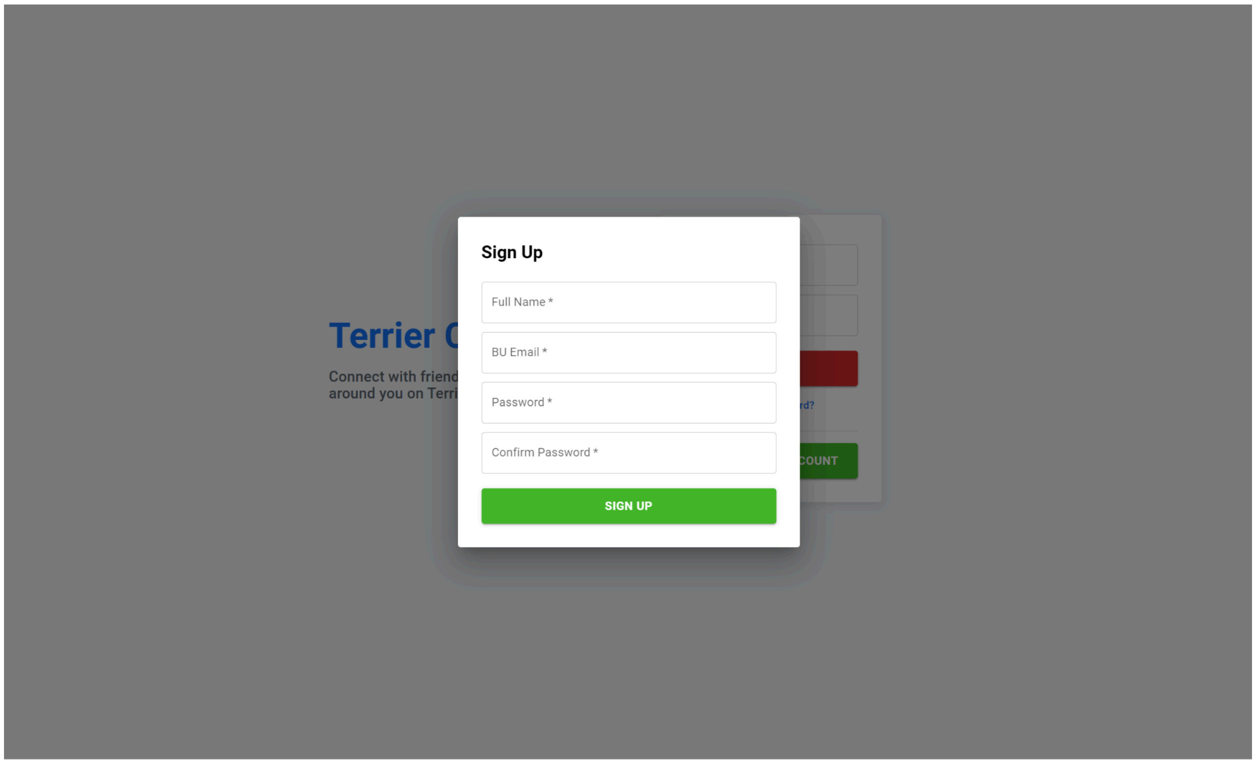
UI Design

Login Page

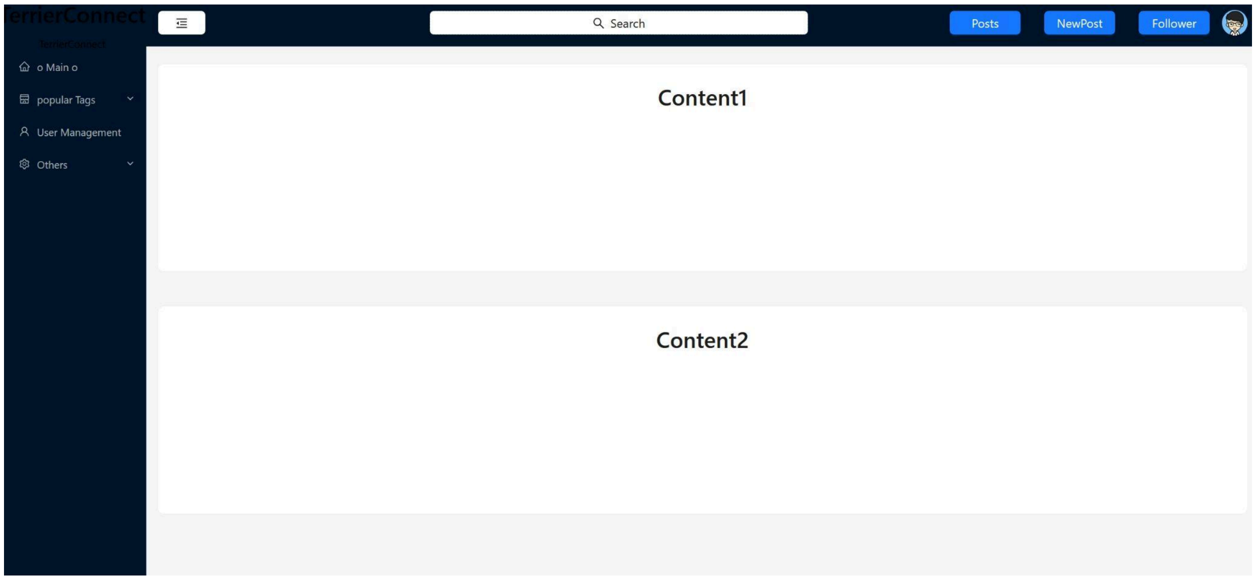

Terrier Connect

Connect with friends and the BU community around you on Terrier Connect.

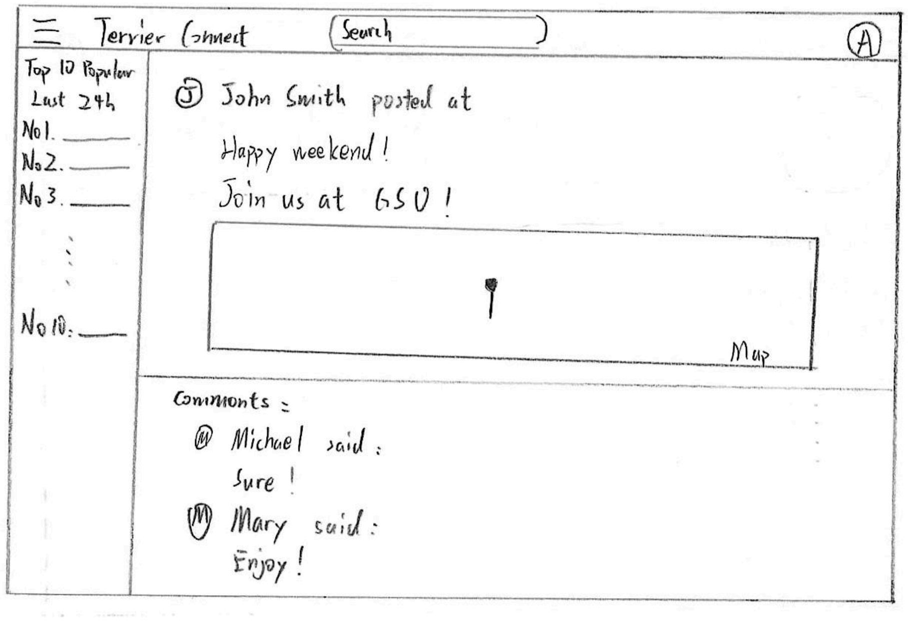
Registration Window



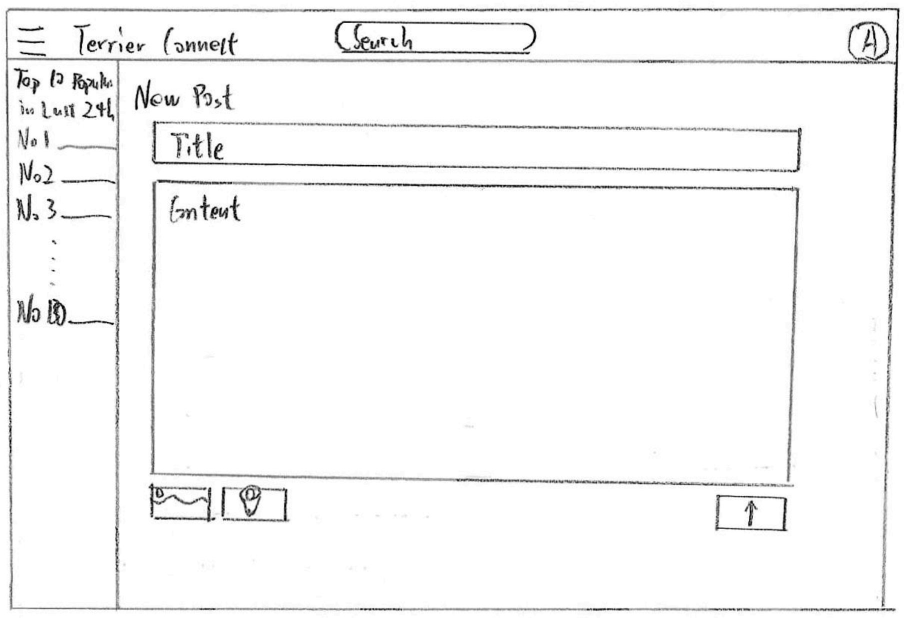
Home page(demo)



Post Detail page



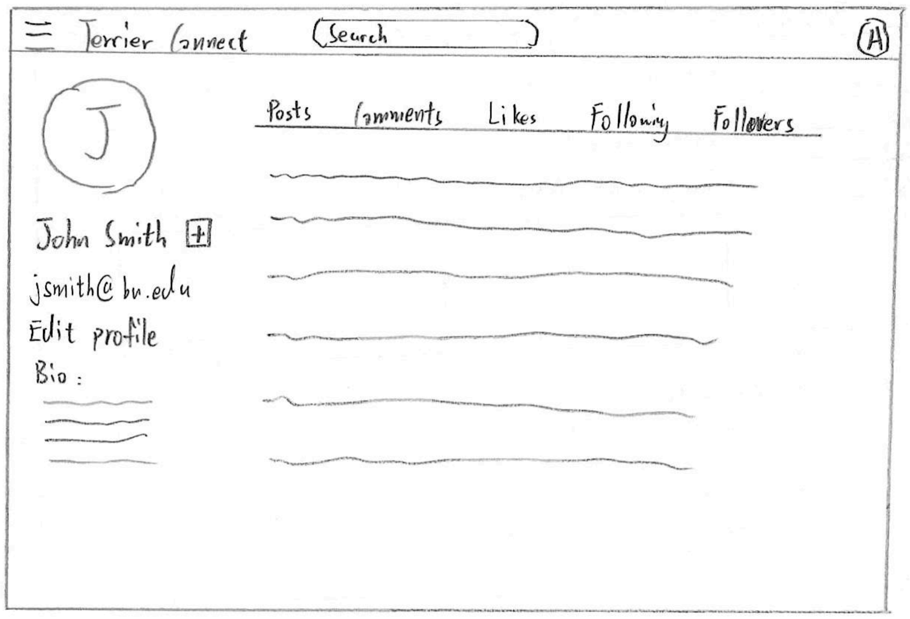
Create Post page



Update Post page

Same as the Create Post page, but modify some buttons.

User profile page



Design Patterns Overview

The design of TerrierConnect follows the **Model-View-Controller (MVC)** pattern, a widely adopted architecture that enhances the modularity, scalability, and maintainability of the application. The MVC framework divides the application into three main components:

1. **Model** - Represents the data layer of the application. It manages the core data logic, handles database interactions, and defines the relationships between different data entities. For example, in TerrierConnect, the Model layer handles user profiles, posts, comments, and hashtags, ensuring that the application has a reliable and consistent data structure.
2. **View** - Constitutes the presentation layer, which is responsible for displaying data and rendering the user interface. Built with the React framework, the View layer in TerrierConnect offers a responsive, user-friendly interface that allows users to interact seamlessly with various features, like viewing posts, following users, and navigating profiles.
3. **Controller** - Acts as the intermediary between the Model and View layers. It processes user inputs, calls relevant Model methods, and updates the View. In TerrierConnect, the Controller handles requests such as creating, updating, and deleting posts or managing user follow relationships. This enables TerrierConnect to handle complex user interactions and ensures efficient communication between the frontend and backend components.

By adopting the MVC pattern, TerrierConnect achieves a clean separation of concerns, where data management, user interface, and application logic remain distinct. This separation not only simplifies code maintenance and debugging but also enables independent development and testing of each layer, promoting a scalable and flexible development environment as the application evolves.