# Entity Relationship Model

## Entity: User

- Attributes:
  user_id (PK), hometown, age, bio, birthdate, brgy, contact_no, country, email, first_name, gender, image_data, image_name, image_type, last_name, middle_name, password, province

## Entity: Post

- Attributes:
  post_id (PK), caption, comment_id, comment_count, created_at, image_data, image_name, image_type, reaction_count, share_count, updated_at, user_id (FK)

## Entity: Share

- Attributes:
  share_id (PK), shared_at, post_id (FK), user_id (FK)

## Entity: Friendship

- Attributes:
  friendship_id (PK), created_at, updated_at, status, receiver_id (FK), requester_id (FK)

## Relationships:

1. User - Post:
   - Relationship: One-to-Many (User can create multiple posts)
   - Foreign Key: user_id in Post
2. User - Share:
   - Relationship: One-to-Many (User can share multiple posts)
   - Foreign Key: user_id in Share
3. Post - Share:
   - Relationship: One-to-Many (A post can be shared multiple times)
   - Foreign Key: post_id in Share
4. User - Friendship:
   - Relationship: Many-to-Many (Users can send and receive friendship requests)
   - Foreign Keys: requester_id, receiver_id in Friendship

## API User Controller

### 1. createUser

- URL: /api/register (POST)
- Purpose: Handles user registration by accepting user details and a profile image.
- Parameters:
    1. userJson: JSON string containing user details (e.g., name, email, etc.).
    2. imageFile: Multipart file containing the user's profile picture.
- Workflow:
    1. Parses the userJson into a User object using ObjectMapper.
    2. Calls the userService.createUser method to save the user details and profile picture in the database.
    3. Returns the created user details with HTTP status 201 CREATED if successful or an error message otherwise.

### 2. login

- URL: /api/login (POST)
- Purpose: Handles user login by validating credentials.
- Parameters:
    1. loginRequest: A User object containing the email and password.
- Workflow:
    1. Calls userService.loginUser to validate the credentials.
    2. If the credentials are valid, retrieves the user details using userService.findUserByEmail.
    3. Returns a success response with the user details if the login is successful.
    4. If invalid credentials are provided, returns an unauthorized response with status 401 UNAUTHORIZED.

### 3. getAllUsersDetailsAndProfiles

- URL: /api/users (GET)
- Purpose: Fetches the details and profile pictures of all users.
- Workflow:
    1. Calls userService.findAllUsers to retrieve all users from the database.
    2. Constructs a response for each user by including non-sensitive data and the profile picture (if available) as a Base64-encoded string.
    3. Returns a success response with a list of users if data is found or a 404 NOT FOUND response if no users exist.

## 4. getUserDetailsAndProfile

- URL: /api/user/{id} (GET)
- Purpose: Fetches the details and profile picture of a specific user by their ID.
- Parameters:
    1. id: The ID of the user to retrieve.
- Workflow:
    1. Calls userService.findUserByUserId to retrieve the user.
    2. If the user exists, constructs a response with non-sensitive details and the profile picture as a Base64-encoded string.
    3. Returns a success response if the user is found or a 404 NOT FOUND response if the user doesn't exist.

## 5. updateUserInfo

- URL: /api/user/{id} (PUT)
- Purpose: Updates a user's details.
- Parameters:
    1. id: The ID of the user to update.
    2. updatedUser: A User object containing updated user details.
- Workflow:
    1. Calls userService.updateUserInfo to update the user's information in the database.
    2. Returns the updated user object with a success response if successful.
    3. Returns an error response if the user is not found or if an exception occurs.

## 6. deleteUser

- URL: /api/user/{id} (DELETE)
- Purpose: Deletes a user by their ID.
- Parameters:
    1. id: The ID of the user to delete.
- Workflow:
    1. Calls userService.deleteUserById to delete the user.
    2. If the user is successfully deleted, returns a success response with status 200 OK.
    3. If the user does not exist, returns a 404 NOT FOUND response.
    4. If an exception occurs, returns a 500 INTERNAL SERVER ERROR response.

## API Post Controller

### 1. createPost Method

Endpoint: POST /api/post

Description:
This method handles the creation of a new post. It accepts two parts in a multipart request:

- A JSON string representing the Post object (post).
- An image file (imageFile).

Key Steps:

1. The userJson string is deserialized into a Post object using ObjectMapper.
2. The postService.createPost method is called to save the post and its associated image.
3. If successful, the created post is returned with a 201 Created status.
4. If any exception occurs, a response with error details is returned with a 500 Internal Server Error status.

Example Use Case:
Creating a new post with a caption, user association, and an optional image.

### 2. getAllPosts Method

Endpoint: GET /api/posts

Description:
This method retrieves all posts stored in the system.

Key Steps:

1. Calls postService.findAllPosts() to fetch all posts from the database.
2. If no posts are found, a 404 Not Found response is returned with a message: "No Posts found".
3. For each post:
    - Extracts key details like postId, caption, users, postedAt, shareCount, commentCount.
    - Encodes the post's image data as a Base64 string if the image exists.
4. Returns the list of posts in the response, with a 200 OK status.

Example Use Case:
Displaying a feed of posts on a social media platform.

### 3. updateReactionCounter Method

Endpoint: PUT /api/post/react/{id}

Description:
This method updates the reaction counter (e.g., likes, upvotes) for a specific post identified by its id.

Key Steps:

1. Reads the reactionCount value from the request body.
2. Validates the reactionCount to ensure it is not null.
3. Calls postService.updateReactionCounter(id, reactionCount) to update the post's reaction count.
4. Returns the updated post details if successful.
5. If any exception occurs (e.g., post not found), an error response is returned with a 500 Internal Server Error status.

Example Use Case:
Incrementing the number of likes on a post when a user reacts to it.

## Friendship Controller

### 1. sendFriendRequest

Endpoint: POST /api/friendship/send
Description: Sends a friend request from one user to another.

Key Steps:

- Validates the request body to ensure requesterId and receiverId are not null.
- Delegates the business logic to the friendshipService.sendFriendRequest method to create and save the friendship.
- Returns the created friendship as a FriendshipDTO with a 201 Created status.

Example Use Case:
A user sends a friend request to another user.

### 2. getFriendRequests

Endpoint: GET /api/friendship/requests/{receiverId}
Description: Retrieves all pending friend requests for a specific user (receiver).

Key Steps:

- Queries the friendshipRepository to find all friendships where the receiverId matches and the status is PENDING.
- Converts the retrieved Friendship objects into FriendshipDTO objects using the friendshipService.convertToDTO method.
- Returns the list of pending friend requests with a 200 OK status.

Example Use Case:
A user views the friend requests they have received but not yet accepted.

### 3. getFriendList

Endpoint: GET /api/friendship/friends/{receiverId}
Description: Retrieves a list of friends (accepted friendships) for a specific user.

Key Steps:

- Queries the friendshipRepository to find all friendships where the receiverId matches and the status is ACCEPTED.
- Converts the retrieved Friendship objects into FriendshipDTO objects using the friendshipService.convertToDTO method.
- Returns the list of friends with a 200 OK status.

Example Use Case:
A user views their current list of friends.

## 4. getFriendshipStatus

Endpoint: GET /api/friendship/status/full/{requesterId}/{receiverId}
Description: Checks the friendship status between two users.

Key Steps:

- Queries the friendshipRepository to find a friendship between the given requesterId and receiverId.
- If a friendship exists, returns the friendship's status (e.g., PENDING, ACCEPTED).
- If no friendship exists, returns a status of NONE.

Example Use Case:
Determining whether two users are friends, pending friends, or have no relationship.

## 5. cancelFriendRequest

Endpoint: DELETE /api/friendship/cancel/{requesterId}/{receiverId}
Description: Cancels a friend request sent by the requesterId to the receiverId.

Key Steps:

- Delegates the logic to the friendshipService.cancelFriendRequest method to delete the friend request.
- Returns a success message if the request is canceled.
- Returns a 404 Not Found response if the friend request does not exist.

Example Use Case:
A user decides to cancel a pending friend request they sent.

## 6. removeFriend

Endpoint: DELETE /api/friendship/remove/{friendshipId}
Description: Removes a friend relationship by deleting the corresponding friendship record.

Key Steps:

- Checks if a friendship record with the given friendshipId exists.
- If the record exists, delete it using friendshipRepository.deleteById(friendshipId).
- Returns a success message if the friendship is removed.
- Returns a 404 Not Found response if the friendship does not exist.

Example Use Case:
A user removes another user from their friend list.

### 7. updateUserInfo

Endpoint: PUT /api/friendship/accept/{friendshipId}
Description: Accepts a pending friend request by updating the friendship's status to ACCEPTED.

Key Steps:

- Delegates the logic to the friendshipService.acceptFriendRequest method to update the friendship's status.
- Returns the updated Friendship object if successful.
- Handles exceptions to return appropriate error responses, such as 404 Not Found if the friendship does not exist.

Example Use Case:
A user accepts a friend request they received.

## Key Features

### 1. Friend Request Management

- Sending Friend Requests:
  - Allows users to send friend requests to other users via the /friendship/send endpoint.
  - Validates the input to ensure both requesterId and receiverId are provided.
  - Provides a response with the created friendship request as confirmation.
- Canceling Friend Requests:
  - Allows users to cancel sent friend requests through the /friendship/cancel/{requesterId}/{receiverId} endpoint.
  - Ensures the request exists before canceling and handles cases where it doesn't exist.

Key Feature: Facilitates the initiation and cancellation of friendship requests between users.

### 2. Friendship Relationship Management

- Accepting Friend Requests:
  - Provides an endpoint to accept pending friend requests (/friendship/accept/{friendshipId}).
  - Updates the status of the friendship from PENDING to ACCEPTED.
- Removing Friends:
  - Enables users to remove existing friends using the /friendship/remove/{friendshipId} endpoint.
  - Deletes the friendship record if it exists.

Key Feature: Allows users to manage the lifecycle of their friendships, from accepting requests to removing friends.

### 3. Friendship Data Retrieval

- Fetching Pending Friend Requests:
    - Retrieves all pending friend requests for a specific user through the /friendship/requests/{receiverId} endpoint.
    - Converts raw data into FriendshipDTO objects for a clean, structured response.
- Fetching Friends List:
    - Provides a list of all accepted friendships (friends) for a specific user via the /friendship/friends/{receiverId} endpoint.
    - Helps users view their existing friends.
- Checking Friendship Status:
    - Returns the current friendship status (e.g., PENDING, ACCEPTED, or NONE) between two users using /friendship/status/full/{requesterId}/{receiverId}.
    - Useful for determining the state of a relationship between two users.

Key Feature: Offers a variety of endpoints to retrieve information about friendships, such as pending requests, current friends, and relationship statuses.

## 4. DTO-Based Responses

- Uses Data Transfer Objects (DTOs) to structure and standardize responses. This ensures:
    - Only relevant data is exposed to the client (e.g., avoiding sensitive fields like internal database IDs).
    - A clean and predictable response format for consumers of the API.

Key Feature: Promotes security and consistency by encapsulating only necessary data in the API responses.

## 5. Integration with Front-End Applications

- CORS Support:
    - Enables cross-origin requests from front-end applications hosted on http://localhost:3000 and http://localhost:3001.
    - Makes the API accessible for development purposes in front-end frameworks like React or Vue.js.

Key Feature: Ensures seamless integration with front-end applications.

## 6. Error Handling

- Includes robust error handling to provide meaningful feedback to API clients:
    - 404 Not Found: When a friendship request or friendship record is not found.
    - 400 Bad Request: When request parameters are invalid (e.g., missing required fields).
    - 500 Internal Server Error: For unexpected errors during API execution.
- Returns error messages in a structured format with additional details about the issue.

Key Feature: Improves the developer experience by providing clear and actionable error responses.

## 7. Status-Based Workflow

- Friendship relationships are managed through a well-defined status system:
    - PENDING: A request is sent but not yet accepted.
    - ACCEPTED: A friendship is established.
    - NONE: No relationship exists between the users.
- Ensures a clear and scalable workflow for handling friendships.

Key Feature: Provides a systematic way to handle different states in a friendship lifecycle.

## 9. CRUD Operations for Friendships

- Implements the following CRUD operations:
    - Create: Sending friend requests.
    - Read: Fetching friend requests, friends list, and relationship statuses.
    - Update: Accepting friend requests.
    - Delete: Canceling friend requests and removing friendships.

Key Feature: Covers all aspects of managing friendships in a user-friendly manner.