PinToBeans: A Pinterest-Like System Database Design Report for CS6083, Spring 2025

Visweswar Sirish Parupudi - N14855438 - vsp7230@nyu.edu
Keshav Rajput -N14591076 - kr3412@nyu.edu

April 2025

Contents

1 Introduction

PinToBeans is an interactive platform that enables users to curate personalized pinboards, upload or collect images from external sources, and repin content from other users' collections. The system supports the creation of private follow streams, allowing users to aggregate posts from multiple boards into customized feeds. Users can engage socially by liking content, leaving comments (with permissions governed at the board level), and establishing friendships within the platform.

To ensure the stability and traceability of content, images are stored securely on the system, alongside metadata such as source URLs, descriptive tags, and timestamps. This metadata infrastructure supports robust search functionality and enhances the retrievability of resources across the platform.

This document presents a detailed database design for the PinToBeans platform, with emphasis on:

- Construction of the relational schema
- Entity-Relationship (ER) modeling
- SQL-based implementation of core platform functionalities

Key design objectives include efficient data storage, rigorous normalization practices, enforcement of data integrity via primary and foreign key constraints, and future-proofing the schema for scalability and extensibility.

The PinToBeans database architecture is centered around the following priorities:

- **Robust User Management:** Ensuring secure and flexible handling of user accounts and credentials.
- **Streamlined Pinning and Repinning Processes:** Supporting intuitive workflows for pin creation, pin reuse, and maintaining linkage to original sources.
- **Fine-Grained Access Control:** Implementing board-specific settings for comment permissions and content visibility.
- **Optimized Search and Retrieval:** Facilitating fast and accurate keyword-based searches through well-structured metadata.
- **Support for Social Features:** Enabling users to establish friendships, interact through likes and comments, and subscribe to follow streams aggregating multiple boards.

The overarching aim of this project is to establish a scalable and resilient database foundation for PinToBeans. This backend infrastructure is designed to deliver a seamless user experience while upholding the principles of consistency, reliability, and adaptability that are critical for sustaining an evolving social platform.

2 Assumptions and Justifications

Database Schema Edge Cases & Assumptions

Core Design Choices

- **Image Storage**: Web-pinned images store both `url` and `source_url`; uploaded images are stored as `stored_blob` with a system-generated URL.
- **Deletion Behavior**: `ON DELETE CASCADE` for user-content relationships (boards, pins); `ON DELETE RESTRICT` for images to prevent orphaned pins; `ON DELETE SET NULL` for `uploaded_by` to preserve images.

Pinning System

- **Original Pins**: Must have `is_original = TRUE` and `repin_from = NULL`; stores image metadata and tags.
- **Repins**: Reference the original via `root_pin_id` chain; inherit image/tags from the original; deletion of the original makes all repins inaccessible.
- **Edge Cases**: Circular repins are prevented via a trigger; duplicate pins on the same board are blocked by `UNIQUE(user_id, image_id, board_id)`; self-repins are prohibited.

Social Features

- **Friendships**: Enforce `user_id1 < user_id2` to prevent duplicates; self-friend requests are impossible; `status` must be either `PENDING` or `ACCEPTED`.
- **Follow Streams**: Cannot include own boards (silently filtered); auto-cleanup when empty.
- **Board Following**: Public boards can be followed by anyone; `friends_only_comments` restricts engagement.

Engagement Rules

- **Likes**: Always attributed to the original pin; deleted when the pin is removed.
- **Comments**: Tied to specific pin instances; friendship required if `friends_only_comments = TRUE`; validated via trigger before insertion.

Data Integrity

- **Constraints**: `UNIQUE` board names per user; valid URL format enforcement; non-empty text for tags/comments.
- **Orphan Prevention**: Periodic cleanup of unused tags; images are preserved even if uploader leaves.

General Assumptions

- All boards/pins are public unless `is_public = FALSE`.
- Image modifications require creating a new pin (no in-place updates).
- Timestamps use server time (`CURRENT_TIMESTAMP`).
- No browser extension is required for pinning (manual URL entry).

2.1 Additional Design Assumptions

- Friendships are symmetric and recorded only upon acceptance, with pending and rejected requests separately managed via the `FriendshipRequest` table.
- Repinning creates a new pin with its own timestamp but traces back to the original pin via `root_pin_id`.
- Likes on repins are automatically redirected to the original pin to centralize like counts.
- Comments may be restricted to friends only depending on board settings (`friends_only_comments`), with enforcement performed via database triggers.
- Timestamps are included for all critical user activities to support future analytics and chronological ordering.

- Image records store both external URLs and internal blobs to ensure resilience against external changes.
- Authentication and authorization are handled entirely at the application layer using sessions/cookies, assuming a single backend database connection for all users.
- Deletion actions are tightly controlled via cascading foreign keys to maintain database consistency without redundant records.
- Passwords are assumed to be hashed at the application level before storage in the database.
- Email format validation (ensuring valid email syntax) is handled at the application level.
- Usernames are not required to be unique across the platform.
- Length constraints on fields such as names, URLs, and comments are enforced at the application level to prevent excessively long entries.
- Emails are treated as case-sensitive unless normalized at the application layer.
- Duplicate likes or board follows by the same user are naturally prevented by primary key constraints.

2.2 Design Justifications

- **Separation of FriendshipRequest and Friendship**: Maintaining a separate `FriendshipRequest` table for pending/rejected friend requests ensures clean modeling of asymmetric requests without cluttering the accepted friendship relation. Symmetric friendships are only inserted into the `Friendship` table upon acceptance, reducing complexity.
- **Use of FollowStream and Follows Separately**: Private curation of boards (FollowStreams) is kept distinct from public board following (Follows), aligning with the requirement that follow streams are user-private while public follows are visible to everyone.
- **Stored Blobs Alongside URLs for Images**: To guard against image URL changes or deletions from external websites, images uploaded by users are stored both as `stored_blob` and their original URLs, ensuring long-term data stability.
- **Cascading Deletes for User Content**: `ON DELETE CASCADE` ensures that when a user deletes their account, all associated boards, pins, likes, and comments are automatically removed, preventing orphan records and preserving referential integrity without manual cleanup.
- **Repin Chain with Root Pin Tracking**: The `root_pin_id` enables fast aggregation of likes and analytics, while allowing unlimited repinning depth without causing complex recursive queries. It preserves the ”origin story” of every repin.
- **Redirected Likes to Root Pins**: By redirecting likes to the root pin, the platform centralizes popularity metrics around the original content, avoiding fragmentation of likes across repins and maintaining a coherent sense of trending images.
- **Trigger-based Enforcement for Permissions**: Comment insertion is validated via triggers to ensure that friendship rules are strictly enforced at the database level, eliminating any reliance on front-end security alone and preventing unauthorized writes.
- **ImageTag Many-to-Many Modeling**: Tags are normalized through a many-to-many `ImageTag` relationship, ensuring efficient querying, avoiding string searching in single fields, and making tagging extensible and manageable.
- **Pin Uniqueness per Board**: By enforcing `UNIQUE(user_id, image_id, board_id)` on `Pin`, users are prevented from pinning the same image multiple times to the same board, keeping boards clean and avoiding duplicate clutter.
- **Self-repin and Self-follow Prevention**: Triggers prevent users from repinning their own pins (inappropriately) or including their own boards in follow streams, protecting against logical inconsistencies and abuse of the platform’s engagement system.
- **Partial vs Total Participation Carefully Modeled**: Total participation (like User creating Pinboards, Pin needing User+Image+Board) ensures that no ”floating” pins or boards can exist. Partial participation is used only where optionality is appropriate (e.g., following boards, sending friend requests).

- **Symmetric Friendship Modeling**: Friendships are represented symmetrically, ensuring that users cannot have duplicate or conflicting friendships, and simplifying querying mutual relationships.
- **Use of Composite Primary Keys for Join Tables**: In `FriendshipRequest`, `ImageTag`, `Includes`, and `Likes`, composite primary keys enforce that duplicate relationships (e.g., liking the same pin twice) are impossible without needing extra surrogate IDs.
- **Timestamps for All Actions**: Storing timestamps on pinning, liking, commenting, friendship acceptance, and followstream creation allows for chronological sorting, trending analysis, and building timeline-based features in future extensions.
- **Application-Level Password Hashing**: Password hashing is assumed to be handled securely at the application layer before inserting into the database, following industry-standard practices of keeping authentication concerns separated from database logic.
- **Consistency with Future Analytics**: The schema is intentionally designed to support easy expansion into recommendation engines (based on likes/tags) and user behavior analysis (based on timestamps), without needing heavy restructuring later.

3 Entity-Relationship (ER) Model for PinToBeans

The entity-relationship (ER) model for PinToBeans captures the essential entities, relationships, and constraints that structure the database backend. Below, we describe the entities, their attributes, primary keys, and major relationships, along with participation constraints and cardinalities.

3.1 Entities

- **User** (user_id, name, email, password, created_at)
- **Pinboard** (board_id, name, category, friends_only_comments, created_at, user_id (FK))
- **Image** (image_id, url, source_url, uploaded_by (FK), stored_blob, created_at)
- **Tag** (tag_id, name)
- **FollowStream** (stream_id, user_id (FK), name, created_at)
- **Pin** (pin_id, user_id (FK), image_id (FK), board_id (FK), repin_from (FK self), root_pin_id (FK self), timestamp, is_original)

All entities are **strong entities**; no weak entities are needed.

3.2 Relationships and Cardinalities

- **Creates:** User → Pinboard (1:N, Total participation from Pinboard)
- **Uploads:** User → Image (1:N, Partial participation from Image)
- **Owns:** User → FollowStream (1:N, Total participation from FollowStream)
- **Pins Image onto Pinboard:** User → Pin → (Image and Pinboard) - User creates Pins (1:N, Total participation from Pin) - Each Pin is associated with exactly one Image and exactly one Pinboard (1:1, Total participation from Pin)
- **Repins:** Pin → Pin (Self-relationship via `repin_from`) - A Pin may optionally reference another Pin that it repins (0 or 1:1, Partial participation)
- **Tags:** Image ↔ Tag (M:N, Partial participation)
- **Likes:** User ↔ Pin (M:N, Partial participation)
- **Comments:** User ↔ Pin (M:N, Partial participation)
- **Requests Friendship:** User → User (M:N, Partial participation, via FriendshipRequest)
- **Are Friends:** User ↔ User (Symmetric M:N, Partial participation, via Friendship)
- **Follows Public Board:** User → Pinboard (M:N, Partial participation, via Follows)
- **Includes Board into Stream:** FollowStream → Pinboard (M:N, Total participation from Includes, Partial participation from FollowStream)

3.3 Participation Summary

- User: Total in creating Pinboards, partial in uploading Images, partial in sending Friend Requests
- Pinboard: Total in being created by User
- Image: Partial in being uploaded by User
- FollowStream: Total participation in Includes relationship
- Pin: Total participation from User, Image, Pinboard
- Like and Comment actions: Partial participation

3.4 Cardinality and Constraints Summary

- User to Pinboard: 1:N
- User to Image (upload): 1:N
- User to FriendshipRequest: M:N
- User to Friendship: Symmetric 1:1 for each accepted pair
- Image to Tag (ImageTag): M:N
- User to Likes: M:N
- User to Comments: M:N
- FollowStream to Includes: M:N

3.5 Weak Entity Sets

There are no classic weak entity sets in the design. All entities have their own primary keys and do not depend on identifying relationships. Composite primary keys (e.g., in FriendshipRequest, ImageTag) are used for associative (junction) tables but do not constitute weak entities.

3.6 ER Diagram

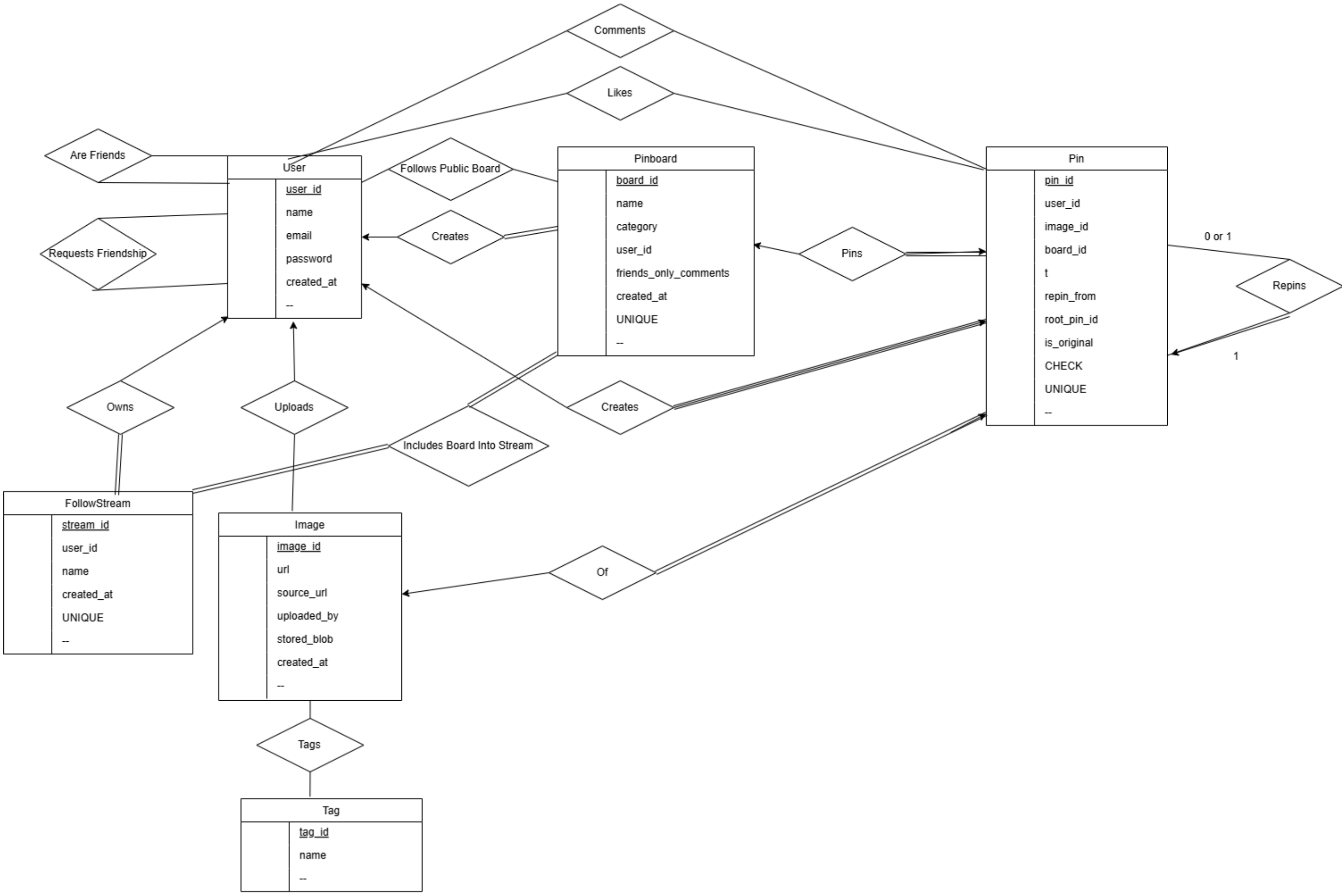


Figure 1: ER Diagram

4 Relational Schema

The database schema is designed to model a Pinterest-like platform, supporting users, pinboards, images, tags, pins, repins, likes, comments, friendships, and follow streams. The schema is structured to be space-efficient, normalized up to Boyce-Codd Normal Form (BCNF), and optimized for query performance. All major actions such as pinning, liking, and commenting are timestamped to allow for future data mining and chronological querying. Many-to-many relationships are handled through separate junction tables

4.1 Tables and Attributes

- **User**(user_id, name, email, password, created_at)
  Primary Key: user_id
  Unique Constraint: email
  Description: Stores registered users and their credentials.

- **FriendshipRequest**(requester_id, target_id, status, request_date)
  Primary Key: (requester_id, target_id)
  Description: Stores pending, accepted, or rejected friend requests.

- **Friendship**(user_id1, user_id2, since_date)
  Primary Key: (user_id1, user_id2)
  Description: Represents accepted friendships as symmetric relationships.

- **Pinboard**(board_id, name, category, user_id, friends_only_comments, created_at)
  Primary Key: board_id
  Unique Constraint: (user_id, name)
  Foreign Key: user_id references User(user_id)
  Description: Collections of pins created by users, with optional friend-only commenting settings.

- **Image**(image_id, url, source_url, uploaded_by, stored_blob, created_at)
  Primary Key: image_id
  Foreign Key: uploaded_by references User(user_id)
  Description: Stores image metadata, web source, and a binary backup.

- **Tag**(tag_id, name)
  Primary Key: tag_id
  Unique Constraint: name
  Description: Tags for categorizing images, enforcing uniqueness.

- **ImageTag**(image_id, tag_id)
  Primary Key: (image_id, tag_id)
  Foreign Keys: image_id references Image(image_id), tag_id references Tag(tag_id)
  Description: Many-to-many relationship between images and tags.

- **Pin**(pin_id, user_id, image_id, board_id, timestamp, repin_from, root_pin_id, is_original)
  Primary Key: pin_id
  Foreign Keys: user_id references User(user_id), image_id references Image(image_id), board_id references Pinboard(board_id), repin_from references Pin(pin_id) ON DELETE SET NULL, root_pin_id references Pin(pin_id) ON DELETE CASCADE
  Unique Constraint: (user_id, image_id, board_id)
  Description: Stores both original pins and repins by users.

- **Likes**(user_id, pin_id, timestamp)
  Primary Key: (user_id, pin_id)

Foreign Keys: user_id references User(user_id), pin_id references Pin(pin_id)
Description: Tracks user likes on pins, redirected automatically to root pins.

- **Comment**(comment_id, user_id, pin_id, text, timestamp)
  Primary Key: comment_id
  Foreign Keys: user_id references User(user_id), pin_id references Pin(pin_id)
  Description: Stores user comments on pins, subject to board permissions.

- **FollowStream**(stream_id, user_id, name, created_at)
  Primary Key: stream_id
  Unique Constraint: (user_id, name)
  Foreign Key: user_id references User(user_id)
  Description: Private collections of boards followed by users.

- **Includes**(stream_id, board_id)
  Primary Key: (stream_id, board_id)
  Foreign Keys: stream_id references FollowStream(stream_id), board_id references Pinboard(board_id)
  Description: Connects follow streams to their included boards.

- **Follows**(user_id, board_id, since_date)
  Primary Key: (user_id, board_id)
  Foreign Keys: user_id references User(user_id), board_id references Pinboard(board_id)
  Description: Publicly visible board-following behavior.

4.2 Foreign Key Constraints

The following foreign key constraints are enforced throughout the schema to maintain referential integrity. Each foreign key is accompanied by appropriate cascading behavior (either `ON DELETE CASCADE` or `ON DELETE SET NULL`) to ensure consistency when parent records are modified or deleted.

- `FriendshipRequest.requester_id` → `User.user_id`
- `FriendshipRequest.target_id` → `User.user_id`
- `Friendship.user_id1` → `User.user_id`
- `Friendship.user_id2` → `User.user_id`
- `Pinboard.user_id` → `User.user_id`
- `Image.uploaded_by` → `User.user_id` (nullable)
- `ImageTag.image_id` → `Image.image_id`
- `ImageTag.tag_id` → `Tag.tag_id`
- `Pin.user_id` → `User.user_id`
- `Pin.image_id` → `Image.image_id`
- `Pin.board_id` → `Pinboard.board_id`
- `Pin.repin_from` → `Pin.pin_id` (nullable)
- `Pin.root_pin_id` → `Pin.pin_id`
- `Likes.user_id` → `User.user_id`
- `Likes.pin_id` → `Pin.pin_id`
- `Comment.user_id` → `User.user_id`
- `Comment.pin_id` → `Pin.pin_id`
- `FollowStream.user_id` → `User.user_id`
- `Includes.stream_id` → `FollowStream.stream_id`
- `Includes.board_id` → `Pinboard.board_id`
- `Follows.user_id` → `User.user_id`
- `Follows.board_id` → `Pinboard.board_id`

4.3 Space Efficiency and Normalization

The schema for **PinToBeans** is carefully designed to minimize redundancy, ensure efficient storage, and guarantee data consistency through systematic application of relational database normalization principles:

- **Redundancy Elimination:** Attributes were factored into separate tables strictly based on functional dependencies. Each relation captures a single concept (e.g., Users, Pinboards, Pins), ensuring no duplication of data across tables. This minimizes storage space and reduces the risk of update anomalies.
- **BCNF Normalization:** All relations have been normalized up to Boyce-Codd Normal Form (BCNF). Every non-trivial functional dependency has a superkey on the left-hand side, ensuring that there are no partial, transitive, or redundant dependencies. As a result, update, delete, and insert anomalies are entirely avoided.
- **Lossless Decomposition:** During schema design, all decompositions were checked to be lossless. This guarantees that no information is lost when relations are decomposed into multiple tables and rejoined during query processing. Critical relationships between entities such as Pins, Users, Boards, and Images are preserved completely.
- **Dependency Preservation:** Where possible, decompositions were designed to preserve important functional dependencies. For example, dependencies such as user email uniquely identifying a user, or a board being uniquely identified by (user_id, board_name), are maintained directly in the schema to avoid expensive join operations during query execution.
- **Efficient Many-to-Many Modeling:** Many-to-many relationships, such as tagging an image with multiple tags, following multiple boards, or including multiple boards inside a follow stream, are modeled using dedicated associative entities (`ImageTag`, `Follows`, `Includes`). This separation avoids multi-valued dependencies and ensures efficient, scalable data retrieval.
- **Timestamping for Critical Events:** Timestamps are systematically included for all user-triggered actions such as pinning, liking, commenting, and creating follow streams. Rather than maintaining redundant history tables, a single timestamp per entity provides enough information for chronological sorting, data mining, and future analytics without bloating the database.
- **Resilience Against Data Anomalies:** Additional triggers were implemented to catch edge cases and enforce business rules, such as preventing self-friendship requests, disallowing self-inclusion of one's own boards into follow streams, redirecting likes to original pins, and validating comment permissions based on friendship status. This ensures that even application-level mistakes cannot compromise data integrity.

4.4 Keys, Constraints, and Referential Design

Primary keys and foreign key constraints ensure entity uniqueness, data integrity, and optimized join performance:

- **Primary Keys:** Surrogate keys (e.g., `user_id`, `image_id`, `pin_id`) are used for fast indexing and join operations.
- **Composite Keys:** Natural composite primary keys are employed for associative entities (e.g., `ImageTag`, `FriendshipRequest`) to reflect domain semantics without redundancy.
- **Unique Constraints:** Unique constraints on fields like `User.email`, `Pinboard.name` (per user), and `Tag.name` ensure domain-specific identity preservation.
- **Foreign Keys and Cascading Behavior:**
  - `ON DELETE CASCADE` for tightly coupled entities (e.g., delete a user → delete their boards, pins, friendships, likes).
  - `ON DELETE SET NULL` for loosely coupled entities where preservation is desirable (e.g., uploaded images remain if user is deleted).

5 Database Schema Implementation in PostgreSQL

The following SQL code defines the complete database schema for the PinToBeans system, including all tables, keys, foreign keys, check constraints, functions, and triggers to enforce data integrity.

5.1 Tables and Schema Definitions

Listing 1: PinToBeans Tables and Schema

```
-- Users
CREATE TABLE "User" (
    user_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL CHECK (length(name) > 0),
    email TEXT UNIQUE NOT NULL,
    password TEXT NOT NULL CHECK (length(password) >= 8),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Friend Requests
CREATE TABLE FriendshipRequest (
    requester_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE
       ,
    target_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    status TEXT NOT NULL CHECK (status IN ('PENDING', 'ACCEPTED', 'REJECTED'))
       ,
    request_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (requester_id, target_id),
    CHECK (requester_id <> target_id)
);

-- Accepted Friendships
CREATE TABLE Friendship (
    user_id1 INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    user_id2 INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    since_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id1, user_id2),
    CHECK (user_id1 < user_id2)
);

-- Pinboards
CREATE TABLE Pinboard (
    board_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL CHECK (length(name) > 0),
    category TEXT,
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    friends_only_comments BOOLEAN NOT NULL DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (user_id, name)
);

-- Images
CREATE TABLE Image (
    image_id SERIAL PRIMARY KEY,
    url TEXT NOT NULL,
    source_url TEXT,
    uploaded_by INTEGER REFERENCES "User"(user_id) ON DELETE SET NULL,
    stored_blob BYTEA NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tags
CREATE TABLE Tag (
    tag_id SERIAL PRIMARY KEY,
    name TEXT UNIQUE NOT NULL CHECK (length(name) > 0)
);

-- Image-Tag Mapping
CREATE TABLE ImageTag (
    image_id INTEGER NOT NULL REFERENCES Image(image_id) ON DELETE CASCADE,
    tag_id INTEGER NOT NULL REFERENCES Tag(tag_id) ON DELETE CASCADE,
    PRIMARY KEY (image_id, tag_id)
);

-- Pins and Repins
CREATE TABLE Pin (
    pin_id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    image_id INTEGER NOT NULL REFERENCES Image(image_id) ON DELETE RESTRICT,
    board_id INTEGER NOT NULL REFERENCES Pinboard(board_id) ON DELETE CASCADE,
    timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    repin_from INTEGER REFERENCES Pin(pin_id) ON DELETE SET NULL,
    root_pin_id INTEGER NOT NULL REFERENCES Pin(pin_id) ON DELETE CASCADE,
    is_original BOOLEAN NOT NULL DEFAULT TRUE,
    CHECK (NOT (is_original AND repin_from IS NOT NULL)),
    UNIQUE (user_id, image_id, board_id)
);

-- Likes
CREATE TABLE Likes (
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    pin_id INTEGER NOT NULL REFERENCES Pin(pin_id) ON DELETE CASCADE,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, pin_id)
);

-- Comments
CREATE TABLE Comment (
    comment_id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    pin_id INTEGER NOT NULL REFERENCES Pin(pin_id) ON DELETE CASCADE,
    text TEXT NOT NULL CHECK (length(text) > 0),
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Follow Streams
CREATE TABLE FollowStream (
    stream_id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    name TEXT NOT NULL CHECK (length(name) > 0),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (user_id, name)
);

-- Boards included inside Follow Streams
CREATE TABLE Includes (
    stream_id INTEGER NOT NULL REFERENCES FollowStream(stream_id) ON DELETE
       CASCADE,
    board_id INTEGER NOT NULL REFERENCES Pinboard(board_id) ON DELETE CASCADE,
    PRIMARY KEY (stream_id, board_id)
);

-- Public Board Following
CREATE TABLE Follows (
    user_id INTEGER NOT NULL REFERENCES "User"(user_id) ON DELETE CASCADE,
    board_id INTEGER NOT NULL REFERENCES Pinboard(board_id) ON DELETE CASCADE,
    since_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, board_id)
);
```

5.2 Triggers and Integrity Functions

Listing 2: PinToBeans Triggers and Functions

```
-- 1. Prevent Self-Friend Request
CREATE OR REPLACE FUNCTION prevent_self_friend_request()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.requester_id = NEW.target_id THEN
        RAISE EXCEPTION ’Cannot␣send␣friend␣request␣to␣yourself’;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_prevent_self_friend_request
BEFORE INSERT ON FriendshipRequest
FOR EACH ROW EXECUTE FUNCTION prevent_self_friend_request();
-- ============================================================================

-- 2. Prevent Including Own Board into Follow Stream
CREATE OR REPLACE FUNCTION prevent_self_followstream()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1 FROM Pinboard
        WHERE board_id = NEW.board_id
```

```
          AND user_id = (SELECT user_id FROM FollowStream WHERE stream_id =
             NEW.stream_id)
    ) THEN
        RAISE EXCEPTION ’Cannot␣include␣your␣own␣board␣in␣a␣follow␣stream.’;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_prevent_self_followstream
BEFORE INSERT ON Includes
FOR EACH ROW EXECUTE FUNCTION prevent_self_followstream();
-- ==============================================================================

-- 3. Redirect Likes to Original Pin
CREATE OR REPLACE FUNCTION redirect_like_to_original()
RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT repin_from FROM Pin WHERE pin_id = NEW.pin_id) IS NOT NULL THEN
        NEW.pin_id := (SELECT root_pin_id FROM Pin WHERE pin_id = NEW.pin_id);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_redirect_like
BEFORE INSERT ON Likes
FOR EACH ROW EXECUTE FUNCTION redirect_like_to_original();
-- ==============================================================================

-- 4. Set Root Pin ID Correctly
CREATE OR REPLACE FUNCTION set_root_pin_id()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.is_original THEN
        NEW.root_pin_id := NEW.pin_id;
    ELSE
        NEW.root_pin_id := (SELECT root_pin_id FROM Pin WHERE pin_id = NEW.
           repin_from);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_set_root_pin_id
BEFORE INSERT ON Pin
FOR EACH ROW EXECUTE FUNCTION set_root_pin_id();
-- ==============================================================================

-- 5. Validate Comment Permissions
CREATE OR REPLACE FUNCTION validate_comment_permissions()
```

```
RETURNS TRIGGER AS $$
DECLARE
    owner_user_id INTEGER;
BEGIN
    SELECT user_id INTO owner_user_id
    FROM Pinboard
    WHERE board_id = (SELECT board_id FROM Pin WHERE pin_id = NEW.pin_id);

    IF EXISTS (
        SELECT 1
        FROM Pinboard
        WHERE board_id = (SELECT board_id FROM Pin WHERE pin_id = NEW.pin_id)
          AND friends_only_comments = TRUE
          AND NOT EXISTS (
              SELECT 1 FROM Friendship
              WHERE (user_id1 = NEW.user_id AND user_id2 = owner_user_id)
                 OR (user_id1 = owner_user_id AND user_id2 = NEW.user_id)
          )
    ) THEN
        RAISE EXCEPTION 'Comment␣not␣allowed:␣Must␣be␣friends␣to␣comment␣on␣
           this␣board.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_comment_permissions
BEFORE INSERT ON Comment
FOR EACH ROW EXECUTE FUNCTION validate_comment_permissions();
-- ==============================================================================
```

6 Sample Data

Explain how you populated the database and why the data is meaningful.

6.1 Users

User ID	Name	Email
1	Naruto Uzumaki	naruto@leaf.com
2	Sasuke Uchiha	sasuke@leaf.com
3	Monkey D. Luffy	luffy@onepiece.com
4	Tony Stark	tony@starkindustries.com
5	Bruce Wayne	bruce@wayneenterprises.com
6	Goku Son	goku@dbz.com
7	Deku Midoriya	deku@ua.com
8	Thanos Titan	thanos@titan.com

6.2 Pinboards

Board ID	Name	Category	User ID	Friends Only?
1	Ramen Obsession	Food	1	No
2	Revenge Quotes	Motivation	2	Yes
3	Treasure Maps	Adventure	3	No
4	Tech Innovations	Technology	4	No
5	Dark Gotham	Cityscapes	5	Yes
6	Saiyan Battles	Anime	6	No
7	Hero Training	Fitness	7	Yes
8	Infinity Stones	Artifacts	8	No
9	Kame House	Relaxation	6	No
10	Hero Society	Inspiration	7	No

6.3 Images

Image ID	URL	Uploaded By (User ID)
1	ramen.jpg	1
2	sharingan.jpg	2
3	onepiece.jpg	3
4	arcreactor.jpg	4
5	batcave.jpg	5
6	ssjgoku.jpg	6
7	deku.jpg	7
8	infinitygauntlet.jpg	8
9	kamehouse.jpg	6
10	herosociety.jpg	7

6.4 Friendships and Likes

Friendships

User 1	User 2
1 (Naruto)	2 (Sasuke)
1 (Naruto)	7 (Deku)
4 (Tony)	5 (Bruce)

Likes

User ID	Liked Pin ID
2 (Sasuke)	1 (Naruto)
3 (Luffy)	2 (Sasuke)
4 (Tony)	3 (Luffy)
5 (Bruce)	5 (Bruce)
1 (Naruto)	3 (Luffy)
7 (Deku)	6 (Goku)

6.5 Pins (Originals and Repins)

Pin ID	User ID (Name)	Image ID (Name)	Board ID (Name)	Original?	Repin From
1	1 (Naruto)	1 (ramen.jpg)	1 (Ramen Obsession)	Yes	–
2	2 (Sasuke)	2 (sharingan.jpg)	2 (Revenge Quotes)	Yes	–
3	3 (Luffy)	3 (onepiece.jpg)	3 (Treasure Maps)	Yes	–
4	4 (Tony)	4 (arcreactor.jpg)	4 (Tech Innovations)	Yes	–
5	5 (Bruce)	5 (batcave.jpg)	5 (Dark Gotham)	Yes	–
6	6 (Goku)	6 (ssjgoku.jpg)	6 (Saiyan Battles)	Yes	–
7	7 (Deku)	7 (deku.jpg)	7 (Hero Training)	Yes	–
8	8 (Thanos)	8 (infinitygauntlet.jpg)	8 (Infinity Stones)	Yes	–
9	6 (Goku)	9 (kamehouse.jpg)	9 (Kame House)	Yes	–
10	7 (Deku)	10 (herosociety.jpg)	10 (Hero Society)	Yes	–
11	7 (Deku)	6 (ssjgoku.jpg)	7 (Hero Training)	No	6 (Goku)
12	1 (Naruto)	3 (onepiece.jpg)	1 (Ramen Obsession)	No	3 (Luffy)

6.6 Comments and Public Follows

Comments

User ID	Pin ID	Text
2 (Sasuke)	1 (Naruto)	I want ramen too!
1 (Naruto)	2 (Sasuke)	Cool sharingan!
7 (Deku)	6 (Goku)	Smash that fight!

Public Follows

User ID	Board ID
6 (Goku)	1 (Ramen Obsession)
2 (Sasuke)	3 (Treasure Maps)
7 (Deku)	5 (Dark Gotham)

6.7 FollowStreams and Includes

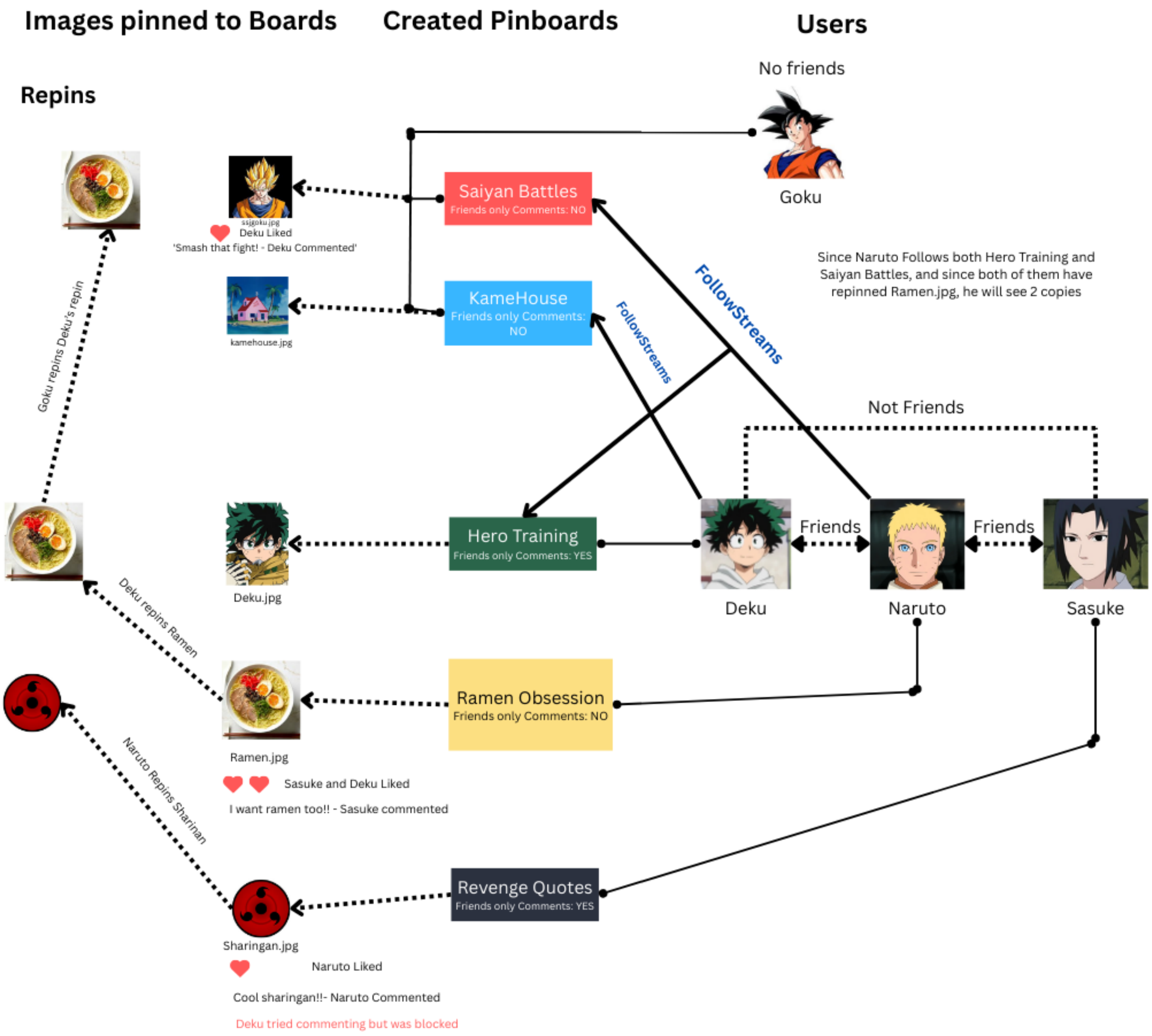
FollowStreams

Stream ID	Name	User
1	Anime Adventures	1 (Naruto)
2	Treasure Tech	3 (Luffy)
3	Relax with Goku	6 (Goku)
4	Hero World	7 (Deku)
5	Battle Boards	6 (Goku)

Boards Included

Stream ID	Board ID
1 (Naruto)	6, 7
2 (Luffy)	4
3 (Goku)	1
4 (Deku)	9
5 (Goku)	1

6.8 TEST DATA ILLUSTRATION



Images pinned to Boards
Created Pinboards
Users
No friends
Repins
Goku
ssjgoku.jpg
Deku Liked
'Smash that fight! - Deku Commented'
Saiyan Battles
Friends only Comments: NO
kamehouse.jpg
KameHouse
Friends only Comments: NO
FollowStreams
FollowStreams
Since Naruto Follows both Hero Training and Saiyan Battles, and since both of them have repinned Ramen.jpg, he will see 2 copies
Goku repins Deku's repin
Not Friends
Hero Training
Friends only Comments: YES
Deku.jpg
Friends
Friends
Deku
Naruto
Sasuke
Deku repins Ramen
Ramen Obsession
Friends only Comments: NO
Ramen.jpg
Sasuke and Deku Liked
I want ramen too!! - Sasuke commented
Naruto Repins Sharinan
Revenge Quotes
Friends only Comments: YES
Sharingan.jpg
Naruto Liked
Cool sharingan!!- Naruto Commented
Deku tried commenting but was blocked

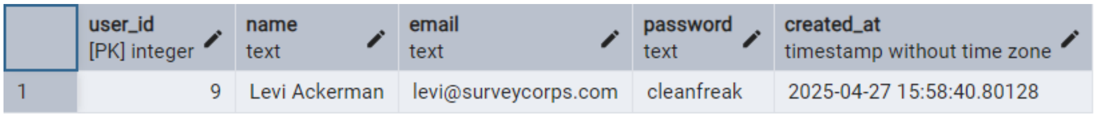
Figure 2: PinToBeans Architecture Diagram

7 SQL Queries for PinToBeans Functionality

7.1 Signing Up, Creating Boards, and Pinning

Sign Up a New User

```
INSERT INTO "User" (name, email, password)
VALUES (’Levi Ackerman’, ’levi@surveycorps.com’, ’cleanfreak’);


SELECT * FROM "User" WHERE email = ’levi@surveycorps.com’;
```

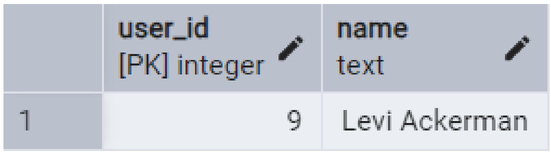


	user_id [PK] integer	name text	email text	password text	created_at timestamp without time zone
1	9	Levi Ackerman	levi@surveycorps.com	cleanfreak	2025-04-27 15:58:40.80128

Figure 3: Output after Signing Up Levi Ackerman

Login

```
SELECT user_id
FROM "User"
WHERE email = ’levi@surveycorps.com’
  AND password = ’cleanfreak’;
```

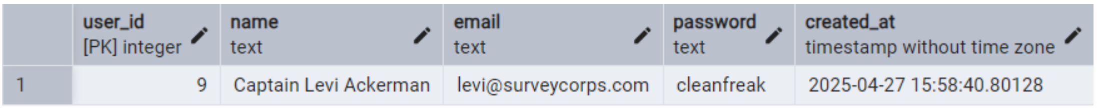


	user_id [PK] integer	name text
1	9	Levi Ackerman

Figure 4: Output after Login Levi Ackerman

Edit User Profile

```
-- Edit name
UPDATE "User"
SET name = ’Captain Levi Ackerman’
WHERE email = ’levi@surveycorps.com’;

-- Verify update
SELECT * FROM "User" WHERE email = ’levi@surveycorps.com’;
```

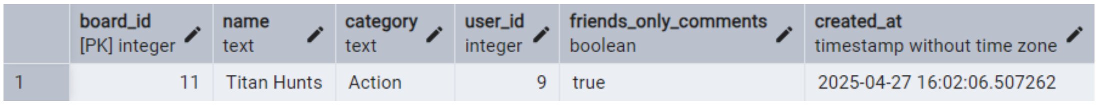


	user_id [PK] integer	name text	email text	password text	created_at timestamp without time zone
1	9	Captain Levi Ackerman	levi@surveycorps.com	cleanfreak	2025-04-27 15:58:40.80128

Figure 5: Output after Edit Profile

Create a New Pinboard

```
-- Create Levi’s Pinboard
INSERT INTO Pinboard (name, category, user_id, friends_only_comments)
VALUES (’Titan Hunts’, ’Action’,
(SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’), TRUE);

-- Check created pinboard
SELECT * FROM Pinboard WHERE name = ’Titan Hunts’;
```

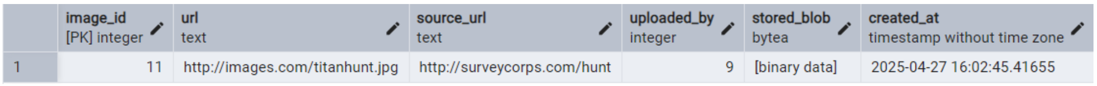


	board_id [PK] integer	name text	category text	user_id integer	friends_only_comments boolean	created_at timestamp without time zone
1	11	Titan Hunts	Action	9	true	2025-04-27 16:02:06.507262

Figure 6: Output after Levi makes a pinboard

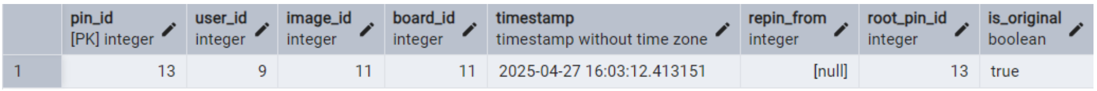
Pin a New Picture

```
-- Upload Image
INSERT INTO Image (url, source_url, uploaded_by, stored_blob)
VALUES (’http://images.com/titanhunt.jpg’, ’http://surveycorps.com/hunt’,
(SELECT user_id FROM "User" WHERE
email = ’levi@surveycorps.com’), decode(’abcd’, ’hex’));

-- Check image inserted
SELECT * FROM Image WHERE url = ’http://images.com/titanhunt.jpg’;

-- Pin it
INSERT INTO Pin (user_id, image_id, board_id, is_original, root_pin_id)
VALUES (
   (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’),
   (SELECT image_id FROM Image WHERE url = ’http://images.com/titanhunt.jpg’),
   (SELECT board_id FROM Pinboard WHERE name = ’Titan Hunts’),
   TRUE,
   0
);

-- Verify Pin
SELECT * FROM Pin WHERE user_id = (SELECT user_id FROM "User" WHERE email = ’levi@surv
```



	image_id [PK] integer	url text	source_url text	uploaded_by integer	stored_blob bytea	created_at timestamp without time zone
1	11	http://images.com/titanhunt.jpg	http://surveycorps.com/hunt	9	[binary data]	2025-04-27 16:02:45.41655

Figure 7: Output after Levi gets a new image

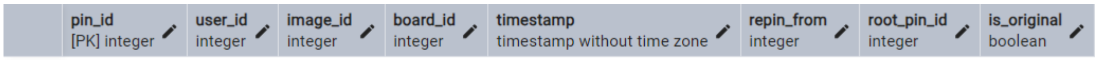


	pin_id [PK] integer	user_id integer	image_id integer	board_id integer	timestamp timestamp without time zone	repin_from integer	root_pin_id integer	is_original boolean
1	13	9	11	11	2025-04-27 16:03:12.413151	[null]	13	true

Figure 8: Output after Levi Pins it

Delete a Pinned Picture

```
DELETE FROM Pin
WHERE pin_id = (
  SELECT pin_id
  FROM Pin
  WHERE user_id = (SELECT user_id FROM "User"
  WHERE email = ’levi@surveycorps.com’)
  LIMIT 1
);

-- Check pins again
SELECT * FROM Pin WHERE user_id =
(SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’);
```



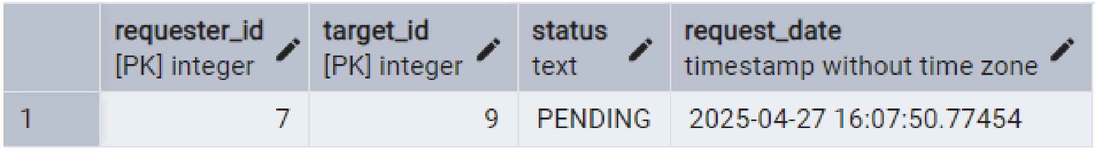
	pin_id [PK] integer	user_id integer	image_id integer	board_id integer	timestamp timestamp without time zone	repin_from integer	root_pin_id integer	is_original boolean

Figure 9: Output after Levi deletes a pin

7.2 Friends: Sending and Accepting Friend Requests

Send a Friend Request: Deku sends Friend Request to Levi

```
INSERT INTO FriendshipRequest (requester_id, target_id, status)
VALUES (
   (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’),
   (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’),
  ’PENDING’
);

-- Check friend request
SELECT * FROM FriendshipRequest
WHERE target_id = (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’);
```

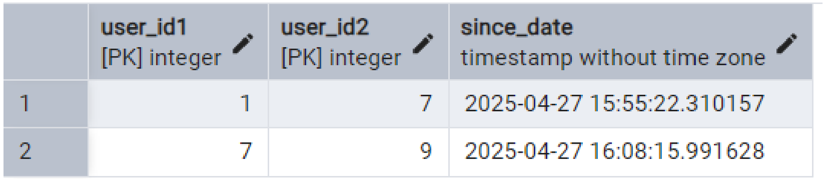


	requester_id [PK] integer	target_id [PK] integer	status text	request_date timestamp without time zone
1	7	9	PENDING	2025-04-27 16:07:50.77454

Figure 10: Output after Deku sends friend request to levi

Accept a Friend Request : Levi Accepts Friend Request

```
-- Accept Request
UPDATE FriendshipRequest
SET status = ’ACCEPTED’
WHERE requester_id = (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’)
AND target_id = (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’);

-- Insert into Friendship
INSERT INTO Friendship (user_id1, user_id2)
VALUES (
  LEAST(
    (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’),
    (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’)
  ),
  GREATEST(
    (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’),
    (SELECT user_id FROM "User" WHERE email = ’levi@surveycorps.com’)
  )
);

-- Check friendships
SELECT * FROM Friendship
WHERE user_id1 = (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’)
OR user_id2 = (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’);
```



	user_id1 [PK] integer	user_id2 [PK] integer	since_date timestamp without time zone
1	1	7	2025-04-27 15:55:22.310157
2	7	9	2025-04-27 16:08:15.991628

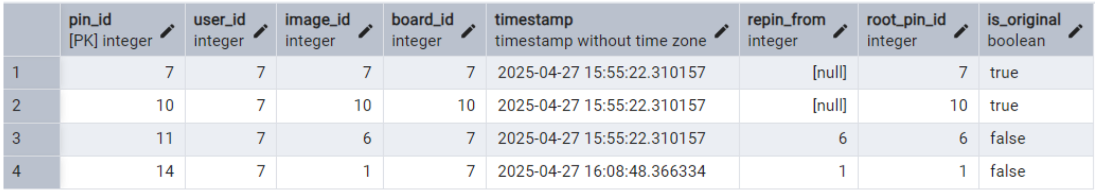
Figure 11: Output of Deku’s friends after Levi accepts friend request

7.3 Repinning and Following

Repin a Picture: Repin Naruto’s Ramen

```
-- Deku repins Ramen image
INSERT INTO Pin (user_id, image_id, board_id, repin_from, is_original, root_pin_id)
VALUES (
   (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’),
   (SELECT image_id FROM Image WHERE url = ’http://images.com/ramen.jpg’),
   (SELECT board_id FROM Pinboard WHERE name = ’Hero Training’),
   (SELECT pin_id FROM Pin WHERE image_id =
   (SELECT image_id FROM Image
  WHERE url = ’http://images.com/ramen.jpg’) LIMIT 1),
  FALSE,
  0
);
```

```
-- Check Deku’s pins
SELECT * FROM Pin
WHERE user_id = (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’);
```

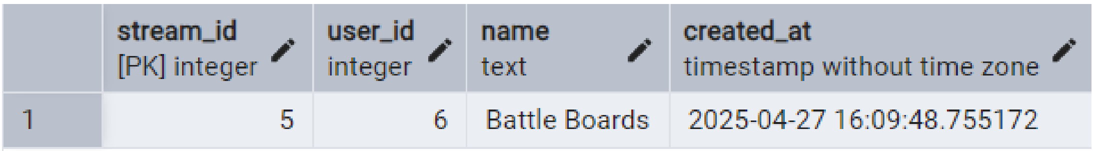


	pin_id [PK] integer	user_id integer	image_id integer	board_id integer	timestamp timestamp without time zone	repin_from integer	root_pin_id integer	is_original boolean
1	7	7	7	7	2025-04-27 15:55:22.310157	[null]	7	true
2	10	7	10	10	2025-04-27 15:55:22.310157	[null]	10	true
3	11	7	6	7	2025-04-27 15:55:22.310157	6	6	false
4	14	7	1	7	2025-04-27 16:08:48.366334	1	1	false

Figure 12: Output after repin

Create a Follow Stream

```
-- Goku creates a follow stream
INSERT INTO FollowStream (user_id, name)
VALUES (
  (SELECT user_id FROM "User" WHERE email = ’goku@dbz.com’),
  ’Battle Boards’
);

-- Verify stream
SELECT * FROM FollowStream WHERE name = ’Battle Boards’;
```

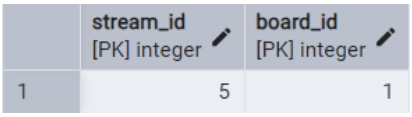


	stream_id [PK] integer	user_id integer	name text	created_at timestamp without time zone
1	5	6	Battle Boards	2025-04-27 16:09:48.755172

Figure 13: Output after Goku makes a followstream

Include Boards in a Follow Stream

```
-- Add Naruto’s board into Battle Boards
INSERT INTO Includes (stream_id, board_id)
VALUES (
  (SELECT stream_id FROM FollowStream WHERE name = ’Battle Boards’
  AND user_id = (SELECT user_id FROM "User" WHERE email = ’goku@dbz.com’)),
  (SELECT board_id FROM Pinboard WHERE name = ’Ramen Obsession’)
);

-- Verify includes
SELECT * FROM Includes
WHERE stream_id = (SELECT stream_id FROM FollowStream
WHERE name = ’Battle Boards’);
```

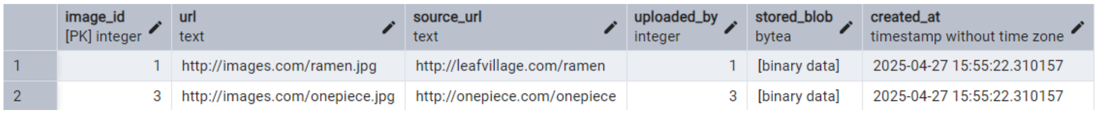


	stream_id [PK] integer	board_id [PK] integer
1	5	1

Figure 14: Output after addidng new board to followstream

Display Pictures from a Follow Stream : Newest First

```
-- Show images in Battle Boards
SELECT Image.*
FROM FollowStream FS
JOIN Includes I ON FS.stream_id = I.stream_id
JOIN Pinboard PB ON I.board_id = PB.board_id
JOIN Pin P ON P.board_id = PB.board_id
JOIN Image ON P.image_id = Image.image_id
WHERE FS.name = 'Battle Boards'
ORDER BY P.timestamp DESC;
```



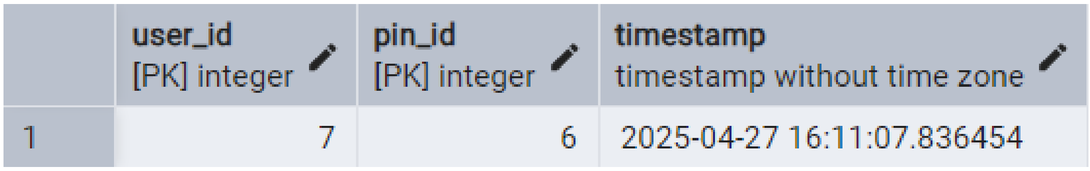
	image_id [PK] integer	url text	source_url text	uploaded_by integer	stored_blob bytea	created_at timestamp without time zone
1	1	http://images.com/ramen.jpg	http://leafvillage.com/ramen	1	[binary data]	2025-04-27 15:55:22.310157
2	3	http://images.com/onepiece.jpg	http://onepiece.com/onepiece	3	[binary data]	2025-04-27 15:55:22.310157

Figure 15: Output after follow stream images display

7.4 Liking and Commenting

Like a Picture

```
-- Deku likes Goku's SSJ battle pin
INSERT INTO Likes (user_id, pin_id)
VALUES (
  (SELECT user_id FROM "User" WHERE email = 'deku@ua.com'),
  (SELECT pin_id FROM Pin
  WHERE image_id = (SELECT image_id FROM Image WHERE
  url = 'http://images.com/ssjgoku.jpg') LIMIT 1)
);

-- Verify Likes
SELECT * FROM Likes
WHERE user_id = (SELECT user_id FROM "User" WHERE email = 'deku@ua.com');
```

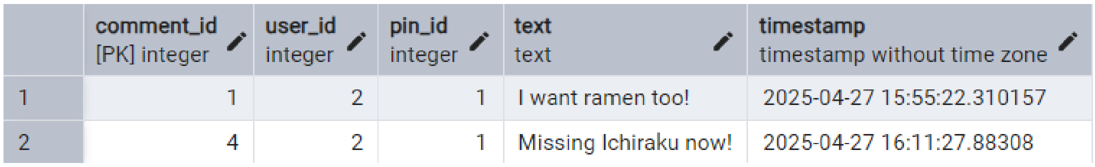


	user_id [PK] integer	pin_id [PK] integer	timestamp timestamp without time zone
1	7	6	2025-04-27 16:11:07.836454

Figure 16: Output after Liking a pin

Comment on a Picture (Valid Friendship)

```
-- Sasuke comments on Naruto's ramen pin
INSERT INTO Comment (user_id, pin_id, text)
VALUES (
  (SELECT user_id FROM "User" WHERE email = 'sasuke@leaf.com'),
  (SELECT pin_id FROM Pin WHERE image_id =
    (SELECT image_id FROM Image
        WHERE url = 'http://images.com/ramen.jpg') LIMIT 1),
  'Missing Ichiraku now!'
);

-- Verify Comments
SELECT * FROM Comment WHERE user_id =
(SELECT user_id FROM "User" WHERE email = 'sasuke@leaf.com');
```



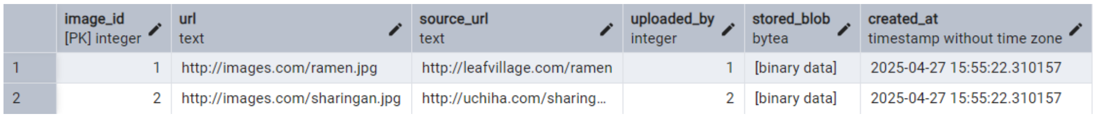
	comment_id [PK] integer	user_id integer	pin_id integer	text text	timestamp timestamp without time zone
1	1	2	1	I want ramen too!	2025-04-27 15:55:22.310157
2	4	2	1	Missing Ichiraku now!	2025-04-27 16:11:27.88308

Figure 17: Output after commenting on a pin with valid friendship

7.5 Keyword Search for Pictures

Search Pictures by Keyword

```
-- Search for 'ninja' tagged images
SELECT DISTINCT Image.*
FROM Image
JOIN ImageTag IT ON Image.image_id = IT.image_id
JOIN Tag T ON T.tag_id = IT.tag_id
WHERE T.name ILIKE '%ninja%'
ORDER BY Image.created_at DESC;
```



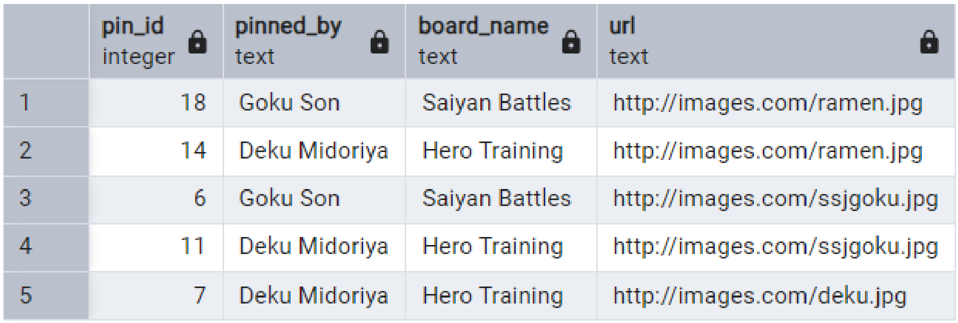
	image_id [PK] integer	url text	source_url text	uploaded_by integer	stored_blob bytea	created_at timestamp without time zone
1	1	http://images.com/ramen.jpg	http://leafvillage.com/ramen	1	[binary data]	2025-04-27 15:55:22.310157
2	2	http://images.com/sharingan.jpg	http://uchiha.com/sharing...	2	[binary data]	2025-04-27 15:55:22.310157

Figure 18: Output after searching images with keywords

7.6 Queries and Testing for robustness and edge cases

FollowStream and Reins and Re-repins

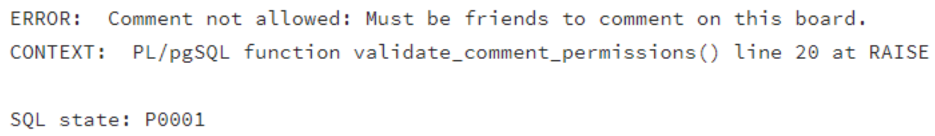
```
-- Goku now repins ramen.jpg from Deku's repin (NOT Naruto's original)
INSERT INTO Pin (user_id, image_id, board_id, repin_from, is_original, root_pin_id)
VALUES (
  (SELECT user_id FROM "User" WHERE email = 'goku@dbz.com'),
  (SELECT image_id FROM Image WHERE url = 'http://images.com/ramen.jpg'),
  (SELECT board_id FROM Pinboard WHERE name = 'Saiyan Battles'),
  (SELECT pin_id FROM Pin
    WHERE user_id = (SELECT user_id FROM "User" WHERE email = 'deku@ua.com')
    AND image_id = (SELECT image_id FROM Image WHERE url = 'http://images.com/ramen.jp
    LIMIT 1),
  FALSE,
  0
);


-- Naruto's Anime Adventures stream check
SELECT P.pin_id, U.name AS pinned_by, PB.name AS board_name, IMG.url
FROM FollowStream FS
JOIN Includes I ON FS.stream_id = I.stream_id
JOIN Pinboard PB ON PB.board_id = I.board_id
JOIN Pin P ON P.board_id = PB.board_id
JOIN Image IMG ON P.image_id = IMG.image_id
JOIN "User" U ON U.user_id = P.user_id
WHERE FS.name = 'Anime Adventures'
AND FS.user_id = (SELECT user_id FROM "User" WHERE email = 'naruto@leaf.com')
ORDER BY P.timestamp DESC;
```



	pin_id integer	pinned_by text	board_name text	url text
1	18	Goku Son	Saiyan Battles	http://images.com/ramen.jpg
2	14	Deku Midoriya	Hero Training	http://images.com/ramen.jpg
3	6	Goku Son	Saiyan Battles	http://images.com/ssjgoku.jpg
4	11	Deku Midoriya	Hero Training	http://images.com/ssjgoku.jpg
5	7	Deku Midoriya	Hero Training	http://images.com/deku.jpg

Non-Friend Comment Block

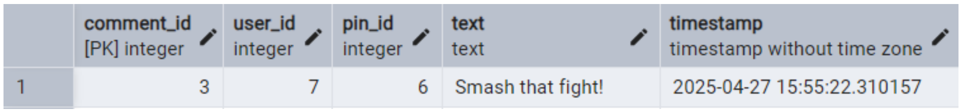
```
-- Thanos tries to comment on Revenge Quotes (should fail)
INSERT INTO Comment (user_id, pin_id, text)
VALUES (
  (SELECT user_id FROM "User" WHERE email = 'thanos@titan.com'),
  (SELECT pin_id FROM Pin WHERE board_id = (SELECT board_id FROM Pinboard WHERE name =
  'Balance in all things...'
);
```



```
ERROR:  Comment not allowed: Must be friends to comment on this board.
CONTEXT:  PL/pgSQL function validate_comment_permissions() line 20 at RAISE

SQL state: P0001
```

Non-Friend comment but pinboard allows non friend comments

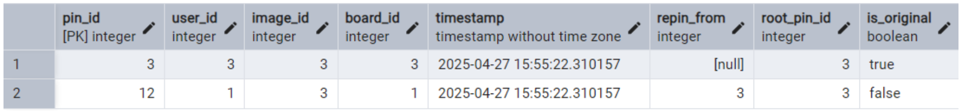
```
-- Deku comments on Goku’s SSJ battle pin
INSERT INTO Comment (user_id, pin_id, text)
VALUES (
   (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’),
   (SELECT pin_id FROM Pin
    WHERE image_id = (SELECT image_id FROM Image WHERE url = ’http://images.com/ssjgoku
    LIMIT 1),
   ’Smash that fight!’
);

-- Verify Comment inserted
SELECT * FROM Comment
WHERE user_id = (SELECT user_id FROM "User" WHERE email = ’deku@ua.com’);
```



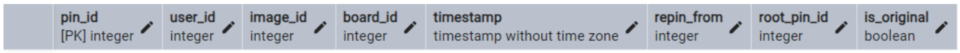
	comment_id [PK] integer	user_id integer	pin_id integer	text text	timestamp timestamp without time zone
1	3	7	6	Smash that fight!	2025-04-27 15:55:22.310157

Delete Root Pin (Cascade Repins)

```
-- Show Luffy’s pin and any repins of his One Piece map
SELECT * FROM Pin
WHERE image_id = (SELECT image_id FROM Image WHERE url = ’http://images.com/onepiece.


-- Delete Luffy’s original One Piece pin
DELETE FROM Pin
WHERE pin_id = (
     SELECT pin_id FROM Pin
     WHERE user_id = 3 -- Luffy
     AND image_id = (SELECT image_id FROM Image WHERE url = ’http://images.com/onepiece
);


-- Confirm both original and repins are gone
SELECT * FROM Pin
WHERE image_id = (SELECT image_id FROM Image WHERE url = ’http://images.com/onepiece.
```



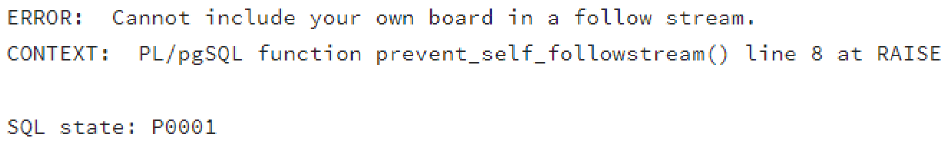
	pin_id [PK] integer	user_id integer	image_id integer	board_id integer	timestamp timestamp without time zone	repin_from integer	root_pin_id integer	is_original boolean
1	3	3	3	3	2025-04-27 15:55:22.310157	[null]	3	true
2	12	1	3	1	2025-04-27 15:55:22.310157	3	3	false



	pin_id [PK] integer	user_id integer	image_id integer	board_id integer	timestamp timestamp without time zone	repin_from integer	root_pin_id integer	is_original boolean

Prevent Self-FollowStream (Include Own Board)

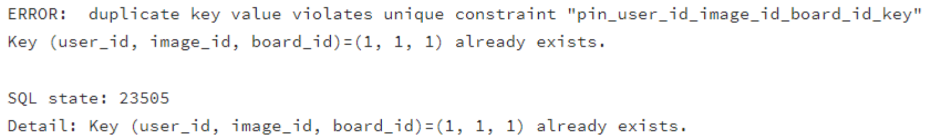
```
-- Naruto tries to add his own board to his own follow stream
INSERT INTO Includes (stream_id, board_id)
VALUES (
  (SELECT stream_id FROM FollowStream WHERE name = ’Anime Adventures’
   AND user_id = (SELECT user_id FROM "User" WHERE email = ’naruto@leaf.com’)),
  (SELECT board_id FROM Pinboard WHERE name = ’Ramen Obsession’)
);
```



```
ERROR:  Cannot include your own board in a follow stream.
CONTEXT:  PL/pgSQL function prevent_self_followstream() line 8 at RAISE

SQL state: P0001
```

Prevent Pinning Same Image Twice to Same Board

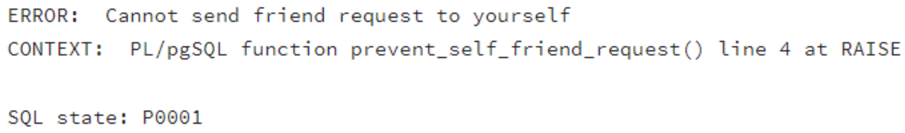
```
-- Naruto tries to pin ramen.jpg AGAIN into Ramen Obsession
INSERT INTO Pin (user_id, image_id, board_id, is_original, root_pin_id)
VALUES (
  (SELECT user_id FROM "User" WHERE email = ’naruto@leaf.com’),
  (SELECT image_id FROM Image WHERE url = ’http://images.com/ramen.jpg’),
  (SELECT board_id FROM Pinboard WHERE name = ’Ramen Obsession’),
  TRUE,
  0
);
```



```
ERROR:  duplicate key value violates unique constraint "pin_user_id_image_id_board_id_key"
Key (user_id, image_id, board_id)=(1, 1, 1) already exists.

SQL state: 23505
Detail: Key (user_id, image_id, board_id)=(1, 1, 1) already exists.
```

Prevent Self-Friending

```
-- Naruto tries to send friend request to himself
INSERT INTO FriendshipRequest (requester_id, target_id, status)
VALUES (
  (SELECT user_id FROM "User" WHERE email = ’naruto@leaf.com’),
  (SELECT user_id FROM "User" WHERE email = ’naruto@leaf.com’),
  ’PENDING’
);
```



```
ERROR:  Cannot send friend request to yourself
CONTEXT:  PL/pgSQL function prevent_self_friend_request() line 4 at RAISE

SQL state: P0001
```

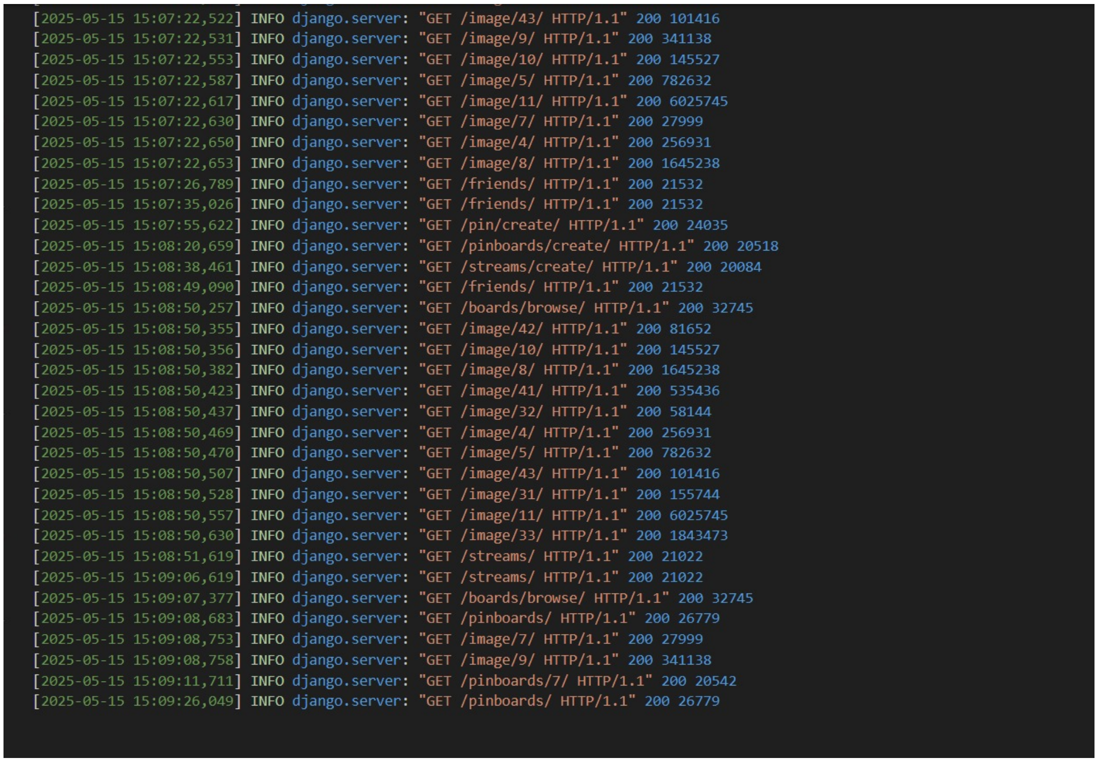
8 Front-end Application and Full stack site

8.1 TechStack used

The programming languages used were Python and HTML and CSS. Django was used for creating the backend, managing the routing and interacting with the database using ORM. HTML templates and CSS were used in tandem to make a better frontend, i.e user interface. The database management system used was PostgreSQL. Logging was performed as well to see the kind of requests a user might send.

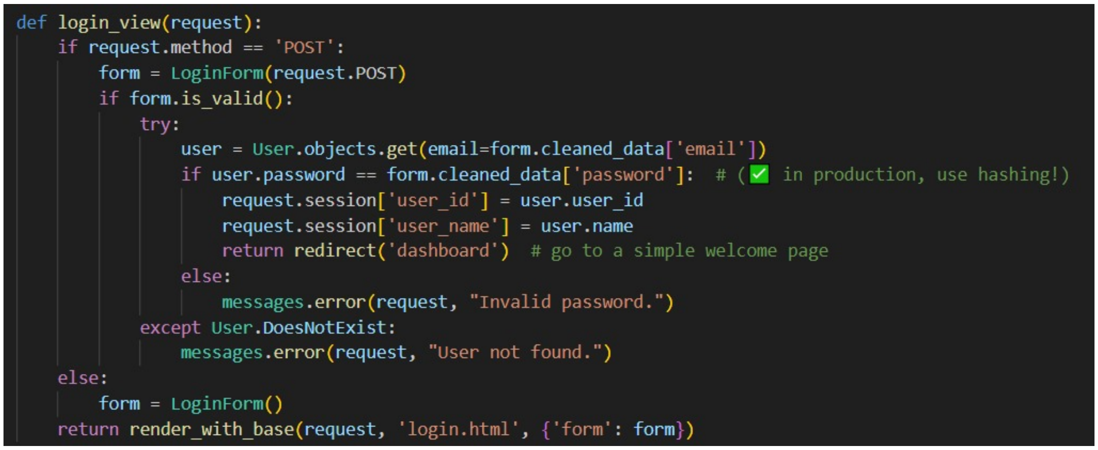
8.2 Code based details

In our code to implement certain functionality, such as login authentication,register user or making a pinboard, we have made use of views, the code for which is under views.py. This is a significant addition on top of what we decided in part 1 of our project. We didn't think we would need views but using views makes our life easier for properties that might be static for the rest of the user session. Django internally handles all the transactions for the calls that it makes to the database, so ACID properties were enforced trivially. Triggers were made use of trivially since they were handled by PostgreSQL. Other than this, our implementation is standard dev code.



```
[2025-05-15 15:07:22,522] INFO django.server: "GET /image/43/ HTTP/1.1" 200 101416
[2025-05-15 15:07:22,531] INFO django.server: "GET /image/9/ HTTP/1.1" 200 341138
[2025-05-15 15:07:22,553] INFO django.server: "GET /image/10/ HTTP/1.1" 200 145527
[2025-05-15 15:07:22,587] INFO django.server: "GET /image/5/ HTTP/1.1" 200 782632
[2025-05-15 15:07:22,617] INFO django.server: "GET /image/11/ HTTP/1.1" 200 6025745
[2025-05-15 15:07:22,630] INFO django.server: "GET /image/7/ HTTP/1.1" 200 27999
[2025-05-15 15:07:22,650] INFO django.server: "GET /image/4/ HTTP/1.1" 200 256931
[2025-05-15 15:07:22,653] INFO django.server: "GET /image/8/ HTTP/1.1" 200 1645238
[2025-05-15 15:07:26,789] INFO django.server: "GET /friends/ HTTP/1.1" 200 21532
[2025-05-15 15:07:35,026] INFO django.server: "GET /friends/ HTTP/1.1" 200 21532
[2025-05-15 15:07:55,622] INFO django.server: "GET /pin/create/ HTTP/1.1" 200 24035
[2025-05-15 15:08:20,659] INFO django.server: "GET /pinboards/create/ HTTP/1.1" 200 20518
[2025-05-15 15:08:38,461] INFO django.server: "GET /streams/create/ HTTP/1.1" 200 20084
[2025-05-15 15:08:49,090] INFO django.server: "GET /friends/ HTTP/1.1" 200 21532
[2025-05-15 15:08:50,257] INFO django.server: "GET /boards/browse/ HTTP/1.1" 200 32745
[2025-05-15 15:08:50,355] INFO django.server: "GET /image/42/ HTTP/1.1" 200 81652
[2025-05-15 15:08:50,356] INFO django.server: "GET /image/10/ HTTP/1.1" 200 145527
[2025-05-15 15:08:50,382] INFO django.server: "GET /image/8/ HTTP/1.1" 200 1645238
[2025-05-15 15:08:50,423] INFO django.server: "GET /image/41/ HTTP/1.1" 200 535436
[2025-05-15 15:08:50,437] INFO django.server: "GET /image/32/ HTTP/1.1" 200 58144
[2025-05-15 15:08:50,469] INFO django.server: "GET /image/4/ HTTP/1.1" 200 256931
[2025-05-15 15:08:50,470] INFO django.server: "GET /image/5/ HTTP/1.1" 200 782632
[2025-05-15 15:08:50,507] INFO django.server: "GET /image/43/ HTTP/1.1" 200 101416
[2025-05-15 15:08:50,528] INFO django.server: "GET /image/31/ HTTP/1.1" 200 155744
[2025-05-15 15:08:50,557] INFO django.server: "GET /image/11/ HTTP/1.1" 200 6025745
[2025-05-15 15:08:50,630] INFO django.server: "GET /image/33/ HTTP/1.1" 200 1843473
[2025-05-15 15:08:51,619] INFO django.server: "GET /streams/ HTTP/1.1" 200 21022
[2025-05-15 15:09:06,619] INFO django.server: "GET /streams/ HTTP/1.1" 200 21022
[2025-05-15 15:09:07,377] INFO django.server: "GET /boards/browse/ HTTP/1.1" 200 32745
[2025-05-15 15:09:08,683] INFO django.server: "GET /pinboards/ HTTP/1.1" 200 26779
[2025-05-15 15:09:08,753] INFO django.server: "GET /image/7/ HTTP/1.1" 200 27999
[2025-05-15 15:09:08,758] INFO django.server: "GET /image/9/ HTTP/1.1" 200 341138
[2025-05-15 15:09:11,711] INFO django.server: "GET /pinboards/7/ HTTP/1.1" 200 20542
[2025-05-15 15:09:26,049] INFO django.server: "GET /pinboards/ HTTP/1.1" 200 26779
```

Figure 19: Logging Data, screenshot of our logs



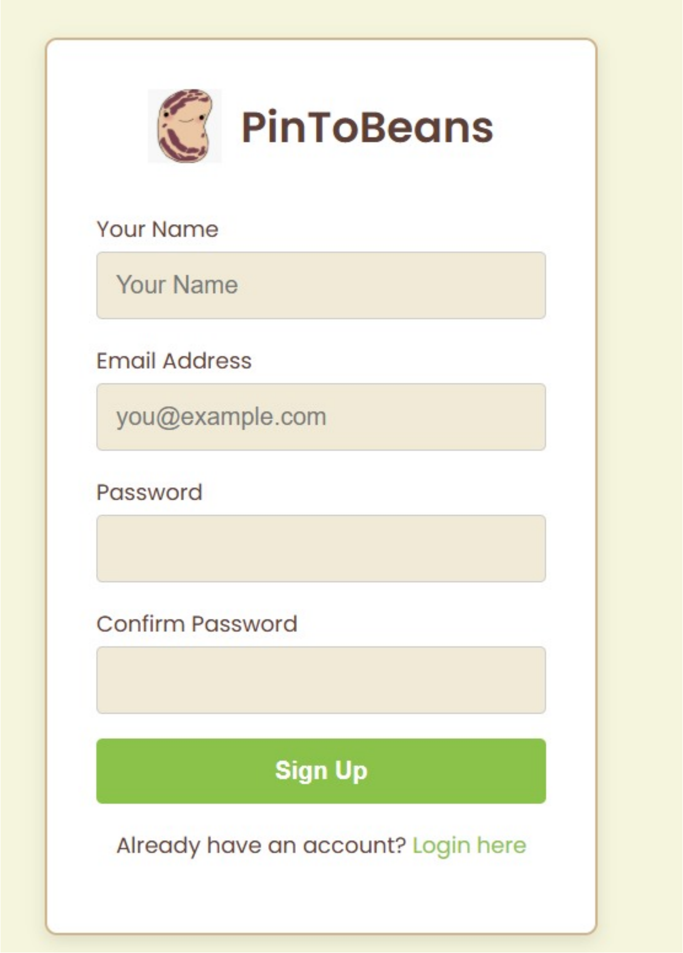
```
def login_view(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            try:
                user = User.objects.get(email=form.cleaned_data['email'])
                if user.password == form.cleaned_data['password']:  # (✅ in production, use hashing!)
                    request.session['user_id'] = user.user_id
                    request.session['user_name'] = user.name
                    return redirect('dashboard')  # go to a simple welcome page
                else:
                    messages.error(request, "Invalid password.")
            except User.DoesNotExist:
                messages.error(request, "User not found.")
    else:
        form = LoginForm()
    return render_with_base(request, 'login.html', {'form': form})
```

Figure 20: One particular view that was implemented.

8.3 User Experience/Flow of the application

The user first sees a login page, the user can login normally or register if it's their first time. Once logged in, they see the global homepage. On this page, they can see their profile picture on the top right. If they click it, they can access their profile information. On the left side is a toolbar for navigation. The tabs on top are different streams for a user. All pins are showed on the homepage.

All the different pages that one can access and see are shown below, they are all simple, standard pages that one would expect a mock website like this to have.



PinToBeans

Your Name

Your Name

Email Address

you@example.com

Password

Confirm Password

Sign Up

Already have an account? Login here

Figure 21: Register Page

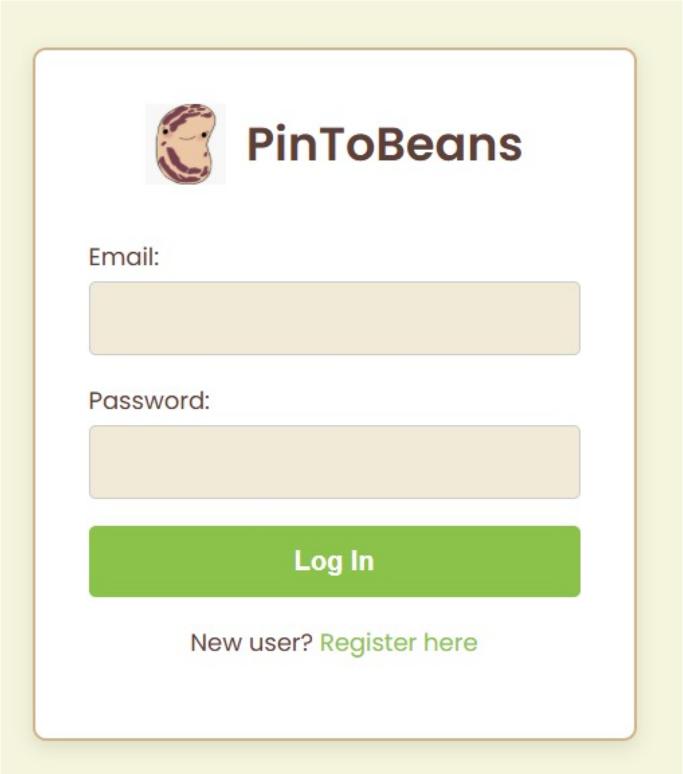


Figure 22: Login Page

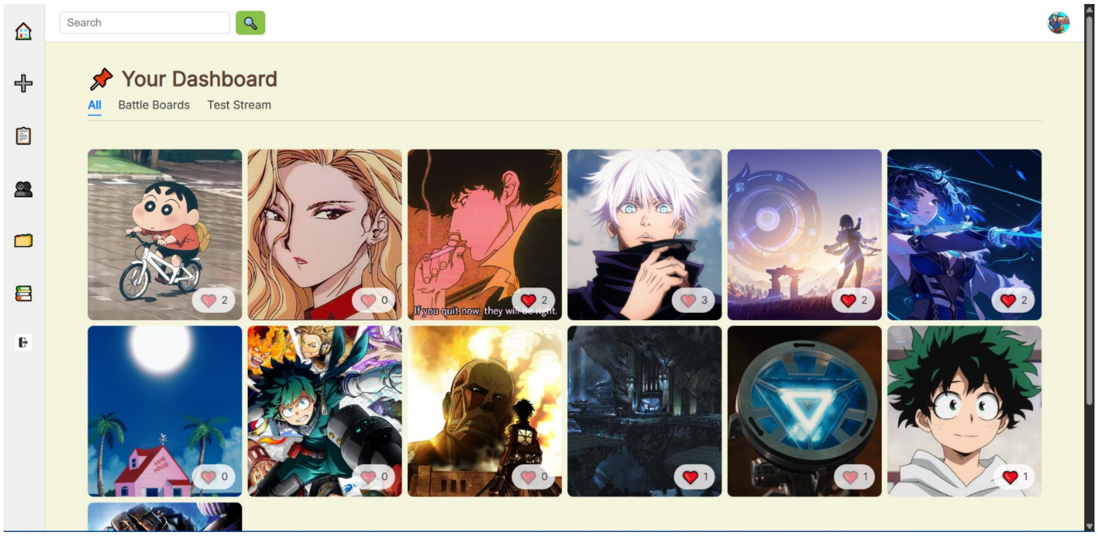
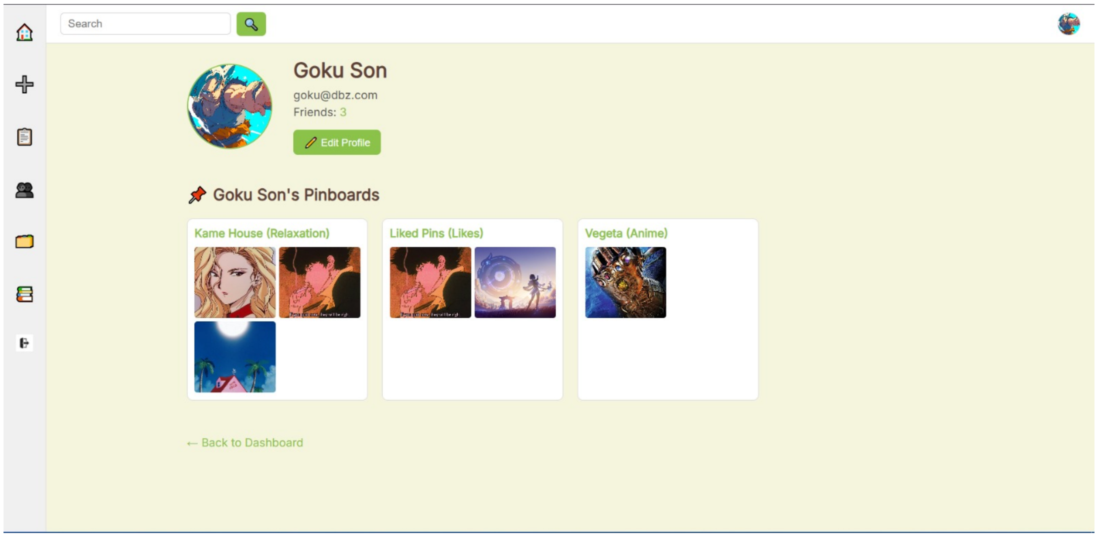


Figure 23: Dashboard



Goku Son

goku@dbz.com

Friends: 3

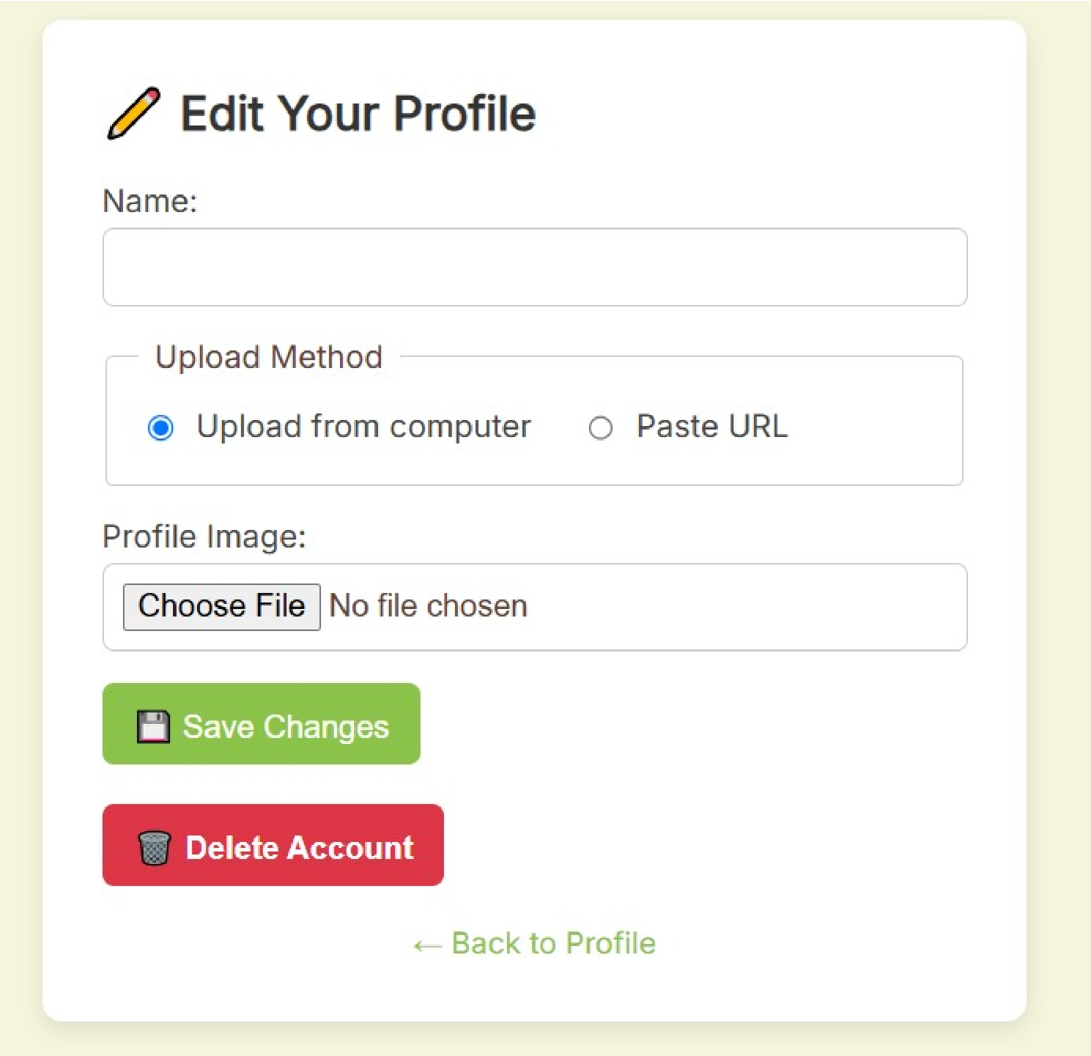
Goku Son's Pinboards

Kame House (Relaxation)

Liked Pins (Likes)

Vegeta (Anime)

Figure 24: Profile Page



Edit Your Profile

Name:

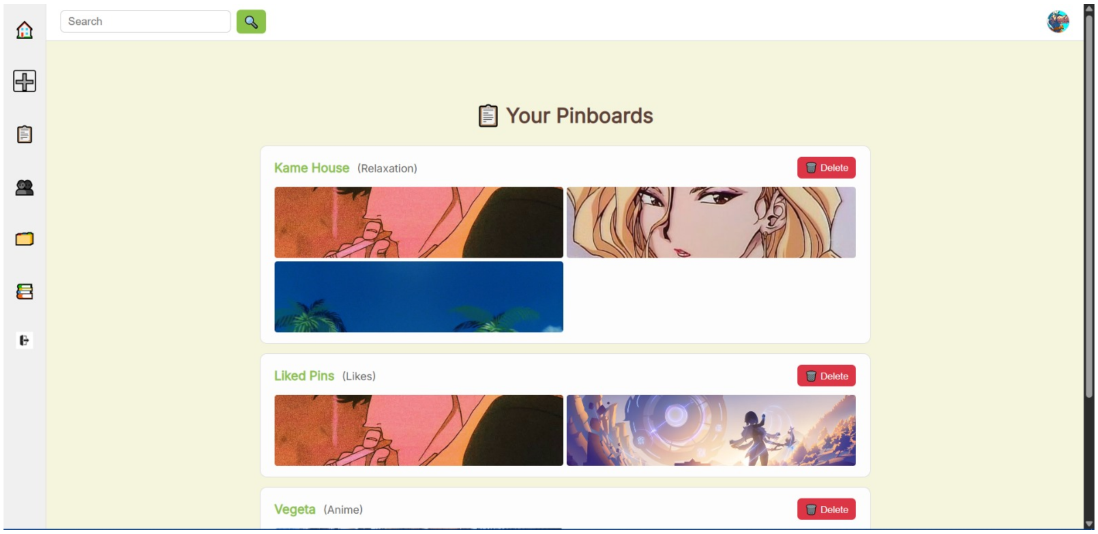
Upload Method

● Upload from computer ○ Paste URL

Profile Image:

Choose File No file chosen

Figure 25: Profile Edit Page



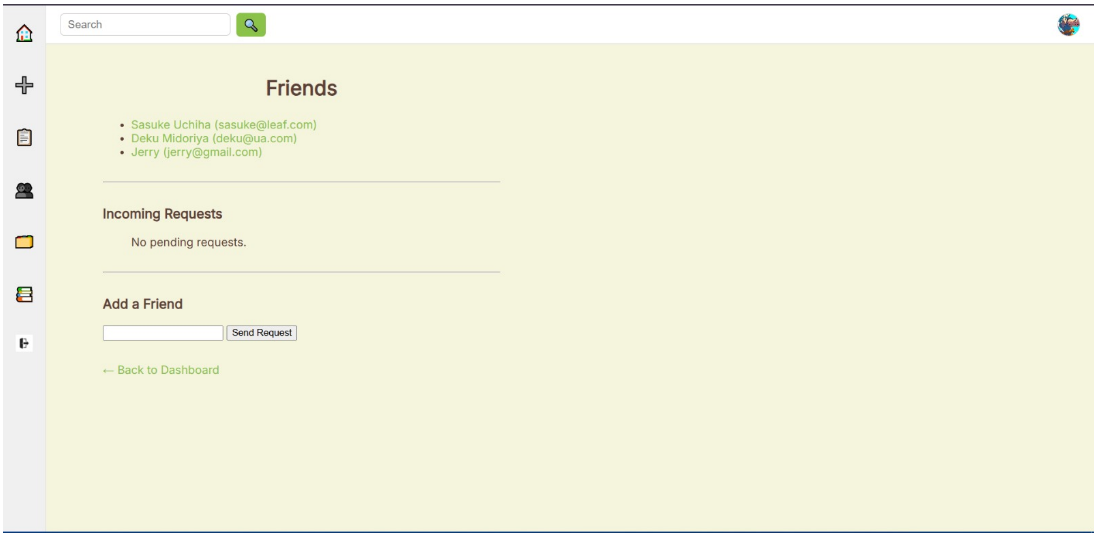
Your Pinboards

Kame House (Relaxation)

Liked Pins (Likes)

Vegeta (Anime)

Figure 26: User’s Pinboard Page



Friends

- Sasuke Uchiha (sasuke@leaf.com)
- Deku Midoriya (deku@ua.com)
- Jerry (jerry@gmail.com)

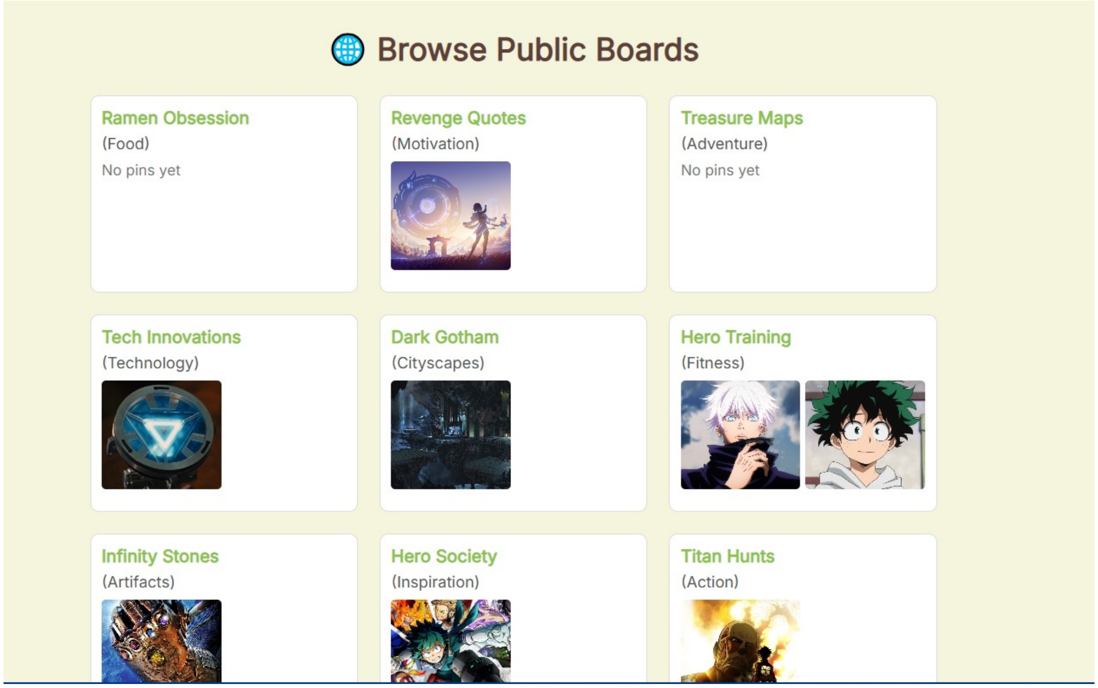
Incoming Requests

No pending requests.

Add a Friend

← Back to Dashboard

Figure 27: Friends Page



Browse Public Boards

Ramen Obsession
(Food)
No pins yet

Revenge Quotes
(Motivation)

Treasure Maps
(Adventure)
No pins yet

Tech Innovations
(Technology)

Dark Gotham
(Cityscapes)

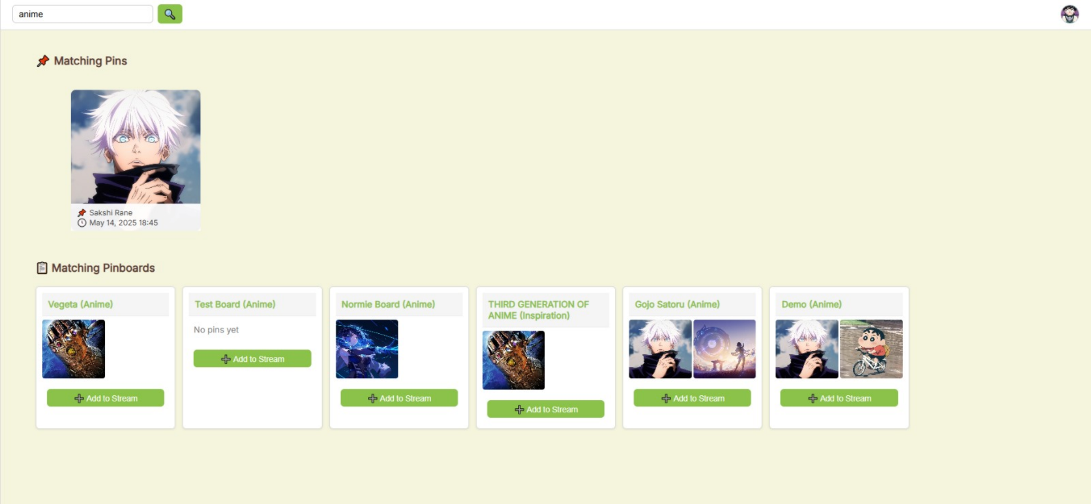
Hero Training
(Fitness)

Infinity Stones
(Artifacts)

Hero Society
(Inspiration)

Titan Hunts
(Action)

Figure 28: View All Public Boards



anime

Matching Pins

Sakshi Rane
May 14, 2025 18:45

Matching Pinboards

Vegeta (Anime)

Test Board (Anime)
No pins yet

Normie Board (Anime)

THIRD GENERATION OF ANIME (Inspiration)

Gojo Satoru (Anime)

Demo (Anime)

Figure 29: Results for searching tag ”anime”

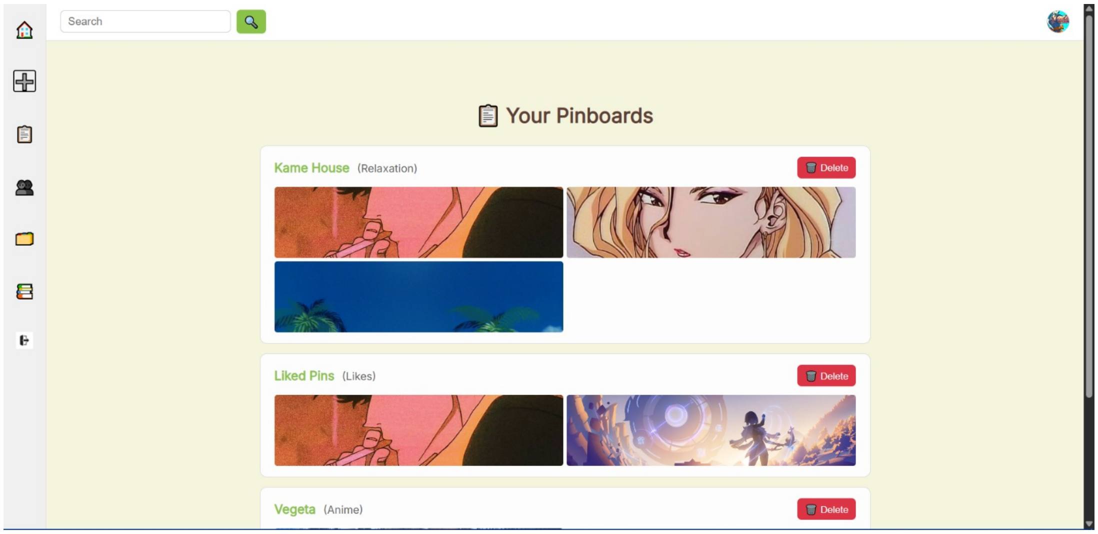
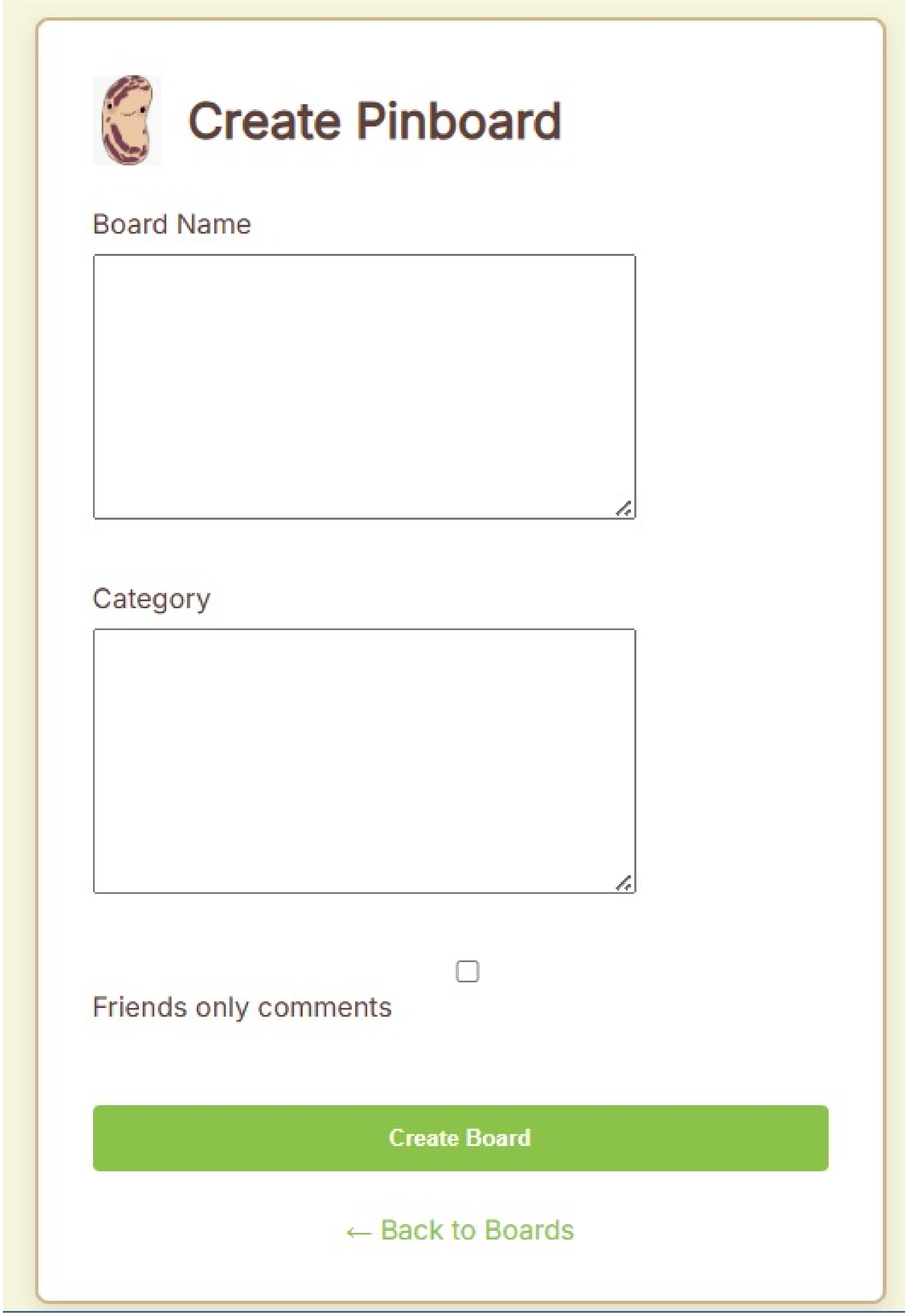


Figure 30: Uploading an image(pin)



Create Pinboard

Board Name

Category

□ Friends only comments

Figure 31: Page For Creating Pinboard

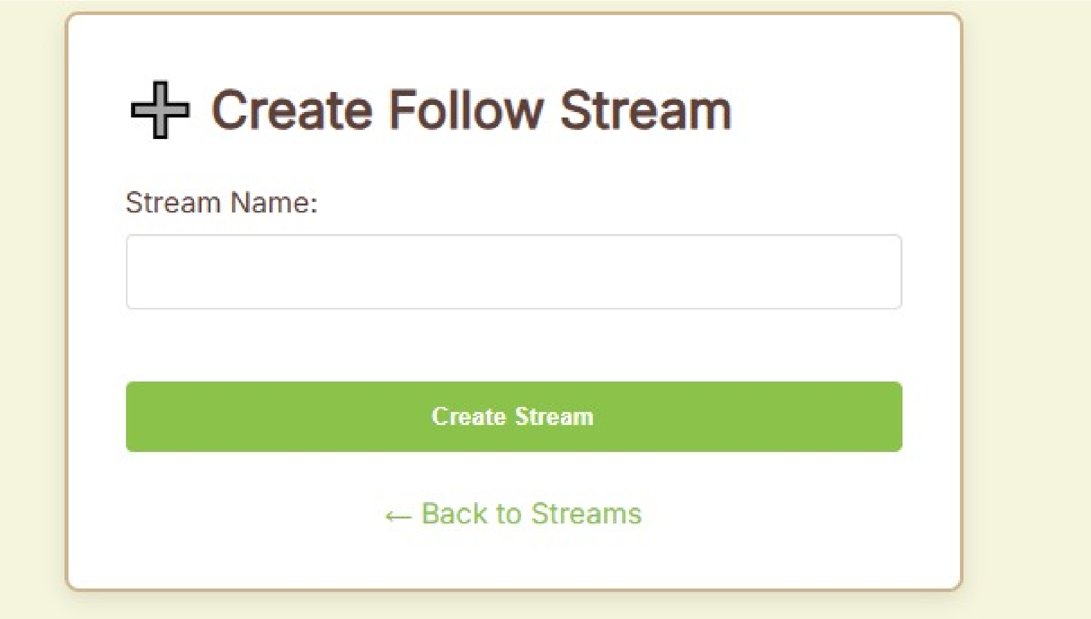


Figure 32: Page For Creating Followstream

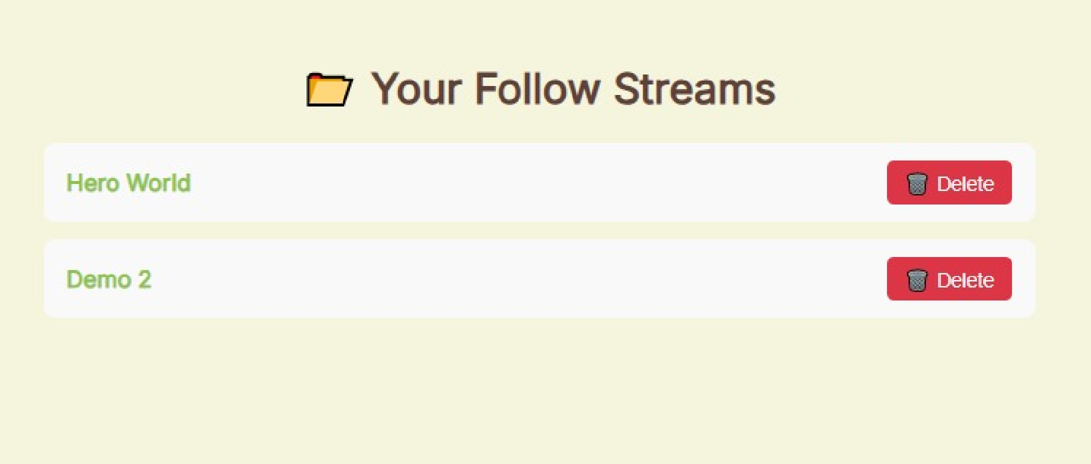


Figure 33: Page For Managing Followstreams

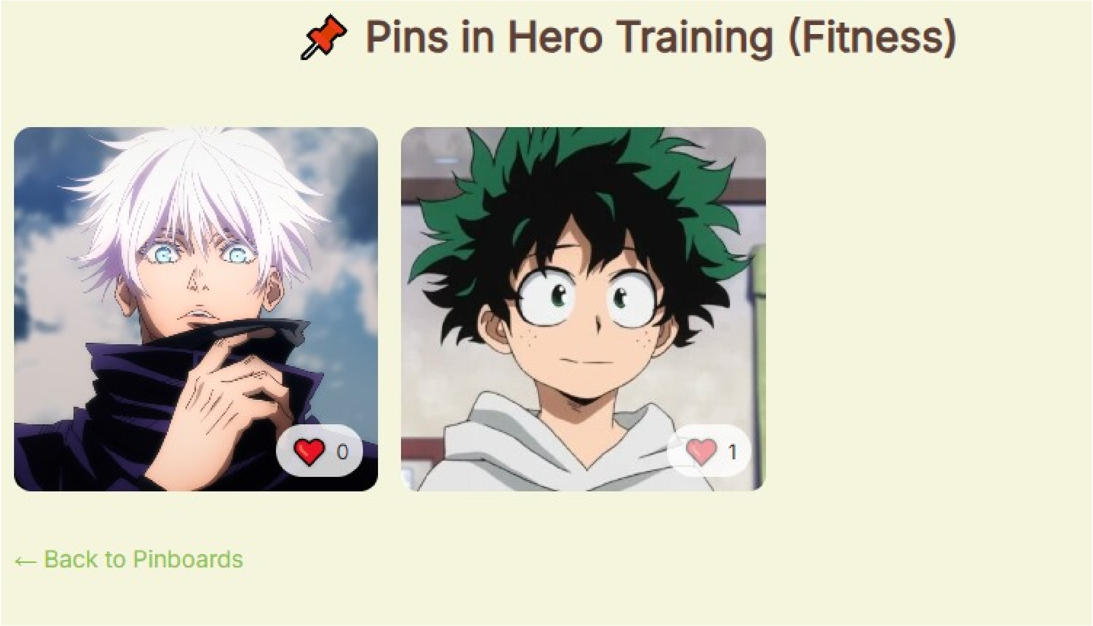

Pins in Hero Training (Fitness)

Figure 34: A particular Pinboard

8.4 Features Implemented

Everything was implemented as discussed in the earlier sections of this report. This includes the login, pinning/repinning, pinboard management, followstreams, and friends functionality. Logging was also implemented for resolving future server end issues(recoverability).
New Features: Every user by default has a pinboard called ”Liked Pins”, this is to ensure that the user can get started with managing their pins fast. This is similar to the concept of liked music on Spotify. Usually a named pinboard is a curation of the user, however the ”Liked Pins” pinboard represents the general tastes of the user.

8.5 Minor Bugs

The following three bugs were found, which could be part of our future work that we ought to fix.
1. The initial uploader of a pin cannot comment on their own pin.
2. When adding two particular follow streams of size 2 and 3, we get a resultant with less number of pins than expected(4 instead of 5).
3. The global like counter gets refreshed between users who are on a different session instance.
Other than this, some UI elements can be improved for consistency and quality of life features, such as ”reset password” and ”forgot password,” which could be added.

9 Conclusion

In this project, we have successfully designed and implemented **PinToBeans**, a Pinterest-inspired web service, culminating in a platform we are quite satisfied with. A core achievement was the development of a comprehensive relational database backend. This schema meticulously captures all essential entities, including users, pinboards, pins, images, tags, friendships, likes, comments, and follow streams, with careful attention given to data integrity, normalization, edge case handling, trigger-based enforcement of rules.

Through a systematic approach, we ensured that all functional requirements were met. This includes features such as friend-restricted commenting, repinning behavior with root pin tracking, and private/public board following. The backend was thoroughly tested with extensive sample data, validating its correctness, robustness, and user-centric behavior. This provides a reliable and scalable foundation to support the dynamic interactions expected in a modern social media platform.

Complementing the robust backend, the website now presents all desired functionality to its full extent, featuring a clean user interface. We also incorporated custom features beyond the initial project scope, such as a dedicated ”Liked Pins” pinboard and comprehensive profile management. Overall, this project has resulted in a well-rounded application, and we are pleased with the final outcome, successfully concluding this phase of our work.