PROJECT 2 DOCUMENTATION

Ayush Pandey (ap6178) , Eric Asare (ea2525)

0 - SUBMITTED FILES

This explains the locations of various files

- Database folder contains
    - ER Diagram
    - tables.sql - Updated SQL queries for creating the table
    - checks.sql - sample queries to test table creation and insertions

- Server folder contains
    - Backend Folder
        - **friends.py** - contains the query for adding friends, listing friends, searching for friends and friend recommendations
        - **likes_comment.py** - contains query for liking a photo, fetching number of photo likes, fetching photo comments, comment search**,** and add_comment
        - **login.py** - contains queries for signing up and logging in a user
        - **photos.py** - contains queries for adding and deleting a photo, listing all photos, searching photos by tag, conjunctive search, 10 most popular tags**,** creating an album, listing an album, deleting an album, top contributors, you may also like features
    - Frontend Folder
        - frontend.py - for rendering and routing the html pages
    - Static Folder
        - contains the css files for styling and js file for interactivity
    - Template Folder
        - contains the html files.

1- ABOUT PHOTOBOOK

Photobook is a web based photo sharing application designed and implemented using PostgreSQL engine and Python Flask.

Data stored in the system includes : Users, Albums, Friends, Photos, Tags, Comments and Likes.

The system support the following use cases:

1. **User management:** becoming a registered user, adding and listing friends, tracking top 10 users.
2. **Album and Photo management:** browse photos as a registered user or visitor, registered users can upload photos and create albums, and delete albums.
3. **Tag Management:** Viewing your/all Photos by Tag Name, Viewing the Most Popular Tags, conjunctive tag queries photo search
4. **Comments:** post a comment, like a photo, search for users who made a comment
5. **Recommendations:** recommend friends of friends, suggest photos users may also like based on common tags.

2- SCHEMA CHANGES AFTER APRIL 17TH

Updated schema is found in database/tables.sql

1. **Users**
    a. Password length changed to 72 since hashed value is longer | `password VARCHAR(72) NULL`

2. **Albums**
    a. Datetime is set to the default timestamp. Users are no longer required to insert time. `datetime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP`

3. **Photos**
    a. Caption is increased from 100 characters to 250 `caption VARCHAR(250) NULL`
    b. Datetime is set to the default timestamp. Users are no longer required to insert time. `datetime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP`
    c. Using path URL instead of BYTEA for storing photos. `path TEXT NOT NULL,`

4. **Friends Table** - Check legit friendship trigger
    a. A user cannot add himself as a friend (previously a user could - we fixed that)

```
CREATE OR REPLACE FUNCTION check_legit_friend_relationship()
RETURNS TRIGGER AS $$
```

```
BEGIN
  IF NOT EXISTS (SELECT 1 FROM users WHERE user_id = NEW.user_id
AND is_visitor = FALSE) OR
    NOT EXISTS (SELECT 1 FROM users WHERE user_id = NEW.friend_id
AND is_visitor = FALSE) OR
    NOT EXISTS (SELECT 1 WHERE NEW.user_id <> NEW.friend_id)
THEN
     RAISE EXCEPTION 'Both users must be non-visitors and not the same
user.';
  END IF;
  RETURN NEW;
END;
```

5. **Comments Table**
    a. Datetime is set to the default timestamp. Users are no longer required to insert time. datetime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
6. **Likes Table**
    a. Datetime is set to the default timestamp. Users are no longer required to insert time. datetime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP

3 - EXPLANATION OF IMPORTANT QUERIES

A. Friend recommendations

This is done in two steps

Step 1: Find all friends of user A

```
SELECT friend_id
      FROM friends
      WHERE user_id = %s,
 (user_id,))
```

Step 2: Recommend friends of friends

```
SELECT u.user_id, u.fname, u.lname, u.email, COUNT(*) as
appearance_count
            FROM friends f
            JOIN users u ON f.friend_id = u.user_id
            WHERE f.user_id IN %s AND f.friend_id != %s AND
f.friend_id NOT IN (SELECT friend_id FROM friends WHERE user_id = %s)
            GROUP BY u.user_id, u.lname, u.fname, u.email
        , (tuple(user_friends), user_id, user_id))

friends_of_friends = cursor.fetchall()
friends_of_friends = sorted(friends_of_friends, key=lambda x: x[4],
reverse=True)
```

Here's a breakdown of the steps:

1. **Find all friends of user A**:
   - The SQL query retrieves all friend IDs from the friends table where the user_id matches a given user_id.
   - The results are fetched and stored in a set called user_friends.
2. **Recommend friends of friends**:
   - If the set user_friends is not empty (meaning user A has friends):
     - Another SQL query is executed to find all friends of friends of user A and count how many times they appear.
     - The query joins the friends table with the users table to get information about the friends of friends.
     - It selects users who are friends with any user in the user_friends set (excluding user A) and who are not already friends with user A.
     - The results are grouped by user ID, last name, first name, and email, and the count of appearances is calculated.
     - The results are sorted based on the appearance count in descending order.
     - The sorted results are converted into a list of dictionaries, each containing information about a recommended friend of a friend, including user ID, first name, last name, email, and appearance count.

B. You may also like

1. Fetch five most commonly used tags among the user's photos

```
            SELECT tag_name, COUNT(tag_name) as tag_count
            FROM tags
            WHERE photo_id IN (
                  SELECT photo_id FROM photos WHERE album_id IN (
                  SELECT album_id FROM albums WHERE user_id = %s
                  )
            )
            GROUP BY tag_name
            ORDER BY tag_count DESC
            LIMIT 5, (user_id,))
```

2. Find photos with similar tags

```
       SELECT P.photo_id, P.caption, P.path,
                        A.album_name, U.fname, U.lname,
                        COUNT(DISTINCT T.tag_name) / NULLIF((SELECT
      COUNT(*) FROM tags WHERE photo_id = P.photo_id), 0) AS
      relevance_ratio
                        FROM photos P
                        INNER JOIN tags T ON P.photo_id = T.photo_id
                        INNER JOIN albums A ON P.album_id = A.album_id
                        INNER JOIN users U ON A.user_id = U.user_id
                        WHERE T.tag_name IN %s
                        AND U.user_id != %s
                        GROUP BY P.photo_id, P.caption, P.path,
      A.album_name, U.fname, U.lname
                        ORDER BY relevance_ratio DESC
                        LIMIT 10
                  """, (tuple(user_tags),) + (user_id,))
```

Breakdown:

1. INNER JOIN clauses: Connects the photos table with tags, albums, and users tables to gather relevant data.
3. COUNT() function and NULLIF: Calculates the relevance ratio by counting the number of distinct tag names associated with each photo and dividing it by the total count of tags for that photo. The NULLIF function prevents division by zero errors.
4. WHERE clause: Filters photos based on specific tags (tag_name) and excludes the specified user (user_id != %s).
5. GROUP BY clause: Groups the results by photo details to aggregate data.
6. ORDER BY clause: Sorts the results by relevance ratio in descending order.
7. LIMIT clause: Limits the results to the top 10 photos with the highest relevance ratios

C. Comment Search

```
SELECT U.user_id, U.fname, U.lname, COUNT(C.text) as comment_count
FROM comments C
INNER JOIN users U ON U.user_id = C.user_id
```

```
WHERE C.text = %s AND U.is_visitor = FALSE
GROUP BY U.user_id
ORDER BY comment_count DESC", (comment,))
```

The SQL query retrieves user information and counts the number of comments made by each user where the comment text matches a given parameter, while also ensuring that only non-visitor users are considered. The results are then sorted based on the comment count in descending order.

- INNER JOIN:
  - INNER JOIN users U ON U.user_id = C.user_id: Joins the users table (U) with the comments table (C) based on the user_id column. This ensures that we retrieve user information for each comment.
- WHERE clause:
  - C.text = %s: Specifies a condition that the text of the comment (C.text) should match a given parameter (%s).
  - U.is_visitor = FALSE: Adds a condition that ensures we only consider users who are not visitors (is_visitor is set to FALSE).
- GROUP BY clause:
  - GROUP BY U.user_id: Groups the results by the user ID. This means that the count of comments will be aggregated per user.
- ORDER BY clause:
  - ORDER BY comment_count DESC: Orders the results by the comment count (comment_count) in descending order. This means users with the highest comment counts will appear first in the result set.

D. conjunctive tag search

```
SELECT P.photo_id, P.caption, P.path, A.album_name, U.fname, U.lname
                  FROM photos P
                  INNER JOIN albums A ON P.album_id = A.album_id
```

```
                   INNER JOIN users U ON A.user_id = U.user_id
                   WHERE P.photo_id IN (
                   SELECT photo_id
                   FROM tags
                   WHERE tag_name IN %s
                   GROUP BY photo_id
                   HAVING COUNT(DISTINCT tag_name) = %s
                   )
                   LIMIT 100""", (tuple(tags), len(set(tags))))
```

1. JOIN clauses: These clauses are used to combine rows from two or more tables based on a related column between them. In this query, two INNER JOIN operations are performed:
   - INNER JOIN albums A ON P.album_id = A.album_id: This joins the photos table with the albums table based on the album_id column.
   - INNER JOIN users U ON A.user_id = U.user_id: This joins the albums table with the users table based on the user_id column.

2. WHERE clause: This filters the rows returned by the query based on specified conditions. In this query, it filters photos based on the tags associated with them. It selects photos where the photo_id is in the result set of a subquery that retrieves photo_id from the tags table for tags that match the ones provided (%s) and where the count of distinct tag names associated with each photo is equal to the total number of tags provided.

3. Subquery: This is the query within the WHERE clause. It selects photo_id from the tags table where the tag_name is in the provided list of tags (%s). It groups the result by photo_id and then filters the groups to only include those where the count of distinct tag_name values is equal to the total number of tags provided.
4. LIMIT clause: This limits the number of rows returned by the query to 100.

E. 10 Most Popular Tags

```
SELECT tag_name, COUNT(photo_id) FROM tags GROUP BY tag_name ORDER BY
COUNT(photo_id) DESC LIMIT 10
```

1. GROUP BY clause: Groups the rows of the table based on the values in the tag_name column. This means that all rows with the same tag name are grouped together.
2. COUNT() function: This function calculates the number of occurrences of each tag by counting the photo_ids associated with each tag.
3. ORDER BY clause: Orders the results based on the count of photo_ids in descending order (from highest count to lowest count). This means the most frequently occurring tags will appear first.
4. DESC keyword: Specifies that the ordering should be in descending order.
5. LIMIT clause: Limits the number of rows returned by the query to 10. So, only the top 10 most frequently used tags are retrieved.

F. Deleting An Album

1. Get the user_id
2. Delete the album , constraint on DB ensures cascading delete of all photos associated with album

```
SELECT user_id FROM albums WHERE album_id = %s", (album_id,))
DELETE FROM albums WHERE album_id = %s AND user_id = %s", (album_id,
user_id))
```

Constraint on DB:

```
-- Albums Table
```

```
CREATE TABLE albums (
      album_id SERIAL PRIMARY KEY,
      album_name VARCHAR(50) NOT NULL,
      user_id INT NOT NULL,
      datetime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

      CONSTRAINT fk_user_id FOREIGN KEY (user_id) REFERENCES users(user_id)
ON DELETE CASCADE
);
```

G. Top Contributors

```
SELECT U.user_id, U.fname, U.lname, COUNT(DISTINCT P.photo_id) +
COUNT(DISTINCT C.comment_id) AS contribution
FROM users U LEFT JOIN albums A ON U.user_id = A.user_id
LEFT JOIN photos P ON A.album_id = P.album_id
LEFT JOIN comments C ON U.user_id = C.user_id
WHERE U.is_visitor = FALSE
GROUP BY U.user_id
HAVING COUNT(DISTINCT P.photo_id) + COUNT(DISTINCT C.comment_id) > 0
ORDER BY contribution DESC LIMIT 10
```

1. SELECT statement: Specifies the columns to retrieve: user_id, fname, lname, and the combined count of unique photo uploads and comments as contribution.
2. FROM clause: Indicates the primary table (users) and other tables (albums, photos, comments) linked through LEFT JOIN operations.
3. LEFT JOIN clauses: Connects the users table with albums, photos, and comments tables to gather data on user contributions.

4. WHERE clause: Filters out users who are not active contributors (is_visitor = FALSE).
5. GROUP BY clause: Groups the results by user_id to aggregate contributions for each user.
6. HAVING clause: Filters out users with no contributions (contribution > 0).
7. ORDER BY clause: Sorts the results by contribution in descending order.
8. LIMIT clause: Limits the results to the top 10 contributors.

4 - DEVELOPMENT PROCESS

1. Designing Database Schema
2. Implementation of Database schema in Flask
3. Building APIs
4. Building Frontend pages
5. Connecting Frontend to Backend APIs

❖ Design

The details in designing the database including the ER diagram, Schema, SQLs and report 1 can be found here: https://github.com/ayushpandeynp/photobook/tree/master/database

❖ Code Structure

Server

- Backend - api calls to DB
- Frontend - routing for pages
- Static - javascript files for interactivity and css files for styling
- Templates - all html that displays webpage

❖ Libraries Used

a. **psycopg2** as PostgreSQL database adapter for the Python
b. **Bcrypt** for modern password hashing
c. **Axios** for asynchronous calls. Axios is a promise based HTTP client.
d. **fontawesome** for styling icons.
e. **JWT** for authentication.
f. **UUID** for unique identifiers generation.
g. **render_template** for displaying the html pages.

❖ Building APIs

Public – doesn’t require token authentication

User scope – requires token authentication

The SQL queries were wrapped under api call

Users

1. Login (/login)
2. Register (/signup , /visitor_signup)

   visitor signup allows most of the data fields to be null. This is needed when a visitor wants to add a comment or like.

Friend

1. Add a friend -/add-friend
2. List all friends of a user - /list-friends'
3. Search for a friend by name - /search-users
4. Friend Recommendation - '/friend-recommendation'

Albums & Photos & Tags

1. Create an album (user scope) - '/create-album'
2. List all albums (user scope)) - /list-albums'
3. List all albums (public scope) - /list-albums-public'
4. List all photos, by album (public) - '/list-photos-by-album'
5. Delete album (user scope) - '/delete-album',
6. Delete photo from an album (user scope) -
7. Photo search by tags (public) - '/photos-with-tags', /photos-with-tags-user
8. You may also like - /you-may-also-like'
9. Top 10 users who make the largest contribution (comments + photos count) - '/top-contributors',
10. Most popular tags (public) '/popular-tags'

Comments & Likes

1. Add comment to a photo (user or visitor – should exist on the users table) '/add-comment',
2. Like a photo (user or visitor) '/like-photo',
3. Total likes of a photo, along with their associated users who liked (public) /photo-likes'
4. All comments of a photo, along with their associated users who commented (public) '/photo-comments'
5. Comment search, returns names of users, and photos with matching comments (public) '/comment-search'

❖ Constraints Implemented

1. In signing up , DOB information can be optional.
2. If the user already exists in the database with the same email address an error message is produced.
3. All photos and albums are made public and registered and non-registered users can browse it.
4. Only registered users can upload photos
5. Users can search between their photos or all photos by a toggle.
6. Top contributors are just registered users - we filter out the visitors.
7. Deleting photos and albums. Deleting the album should delete all photos in it.
8. Users can only modify or delete their photos
9. Users search through the photos by specifying conjunctive tag queries
10. Both registered and unregistered users can leave comments
11. Users cannot leave comments on their own photos
12. Search for users whose comments match exactly the entered comment. If multiple users, return in order of the number of comments matched.

13. There is a limit of 10 on popular tags. Clicking on a tag opens the photos associated with it.
14. Any user should be able to see how many likes a photo has and the names of the users who liked this photo.
15. Extra - Visitors or registered users cannot like a photo more than once

❖ Assumptions

1. Friend relationship is undirected which means that if User 1 add User 2 as a friend, then User 2's friend is User 1 and vise versa
2. Deleting an album will delete all data associated with it, photos and even comments.
3. In the You May Also Like functionality:
    a. We order based on conciseness = (the total matched of top popular tags/ number of tags belonging to the photo), the higher the value the higher it is ranked.