Students: Orsat Puljizević ER2058
Zvonimir Pipić ER2059
Course: Application security
Excersise: User login, session management and password reset process

The component for user registration and log in is made using flask for python and postgresql. The purpose of the component is to register new users into the database and validate them on log in. It requests users username and e-mail. The passwords are hashed and salted with bcrypt library. We are using Flasks session manager and we implemented a password recovery system.

Functional requirements

FR1	Account creation by entering username, email and password.
FR2	System should validate input formats and prevent duplicate emails and usernames.
FR3	Activation token should be generated for every new user and sent via email.
FR4	Account is activated after valid token is used, before that there is no access.
FR5	User can request new activation token.
FR6	User should be able to login using either username or email and password.
FR7	Information is stored in a PostgresSQL database.
FR8	User should be able to reset their password.
FR9	User should be able to log out.
FR10	Authenticated users can create new posts with a title and one media file (image, GIF, video).
FR11	Users can add keywords to posts for categorization and search.
FR12	Users can view posts and associated comments.
FR13	Authenticated users can comment on posts and reply to other comments.
FR14	Users can like or dislike posts; each user may vote once per post.
FR15	Post owners may delete their own posts.
FR16	Administrators or users with appropriate permissions may delete or recover any post.
FR17	Deleted posts are hidden from regular users but visible to authorized roles.

Nonfunctional requirements

NFR1	Passwords should be hashed using bcrypt and stored as hashes.
NFR2	Activation token should be generated using cryptographically secure random values.
NFR3	Response time from the forms should be minimal.
NFR4	Activation tokens expire after 24 hours.
NFR5	Error message should not reveal sensitive information.
NFR6	Emails and usernames should be unique, enforced by the database.
NFR7	New sessions should be created on log in and destroyed on log out.
NFR8	Reset password tokens expire after 1 hour.
NFR9	User input must not allow cross-site scripting (XSS).
NFR10	Database operations must be protected against SQL injection.
NFR11	Uploaded files must be validated and sanitized before storage.
NFR12	Only explicitly allowed file formats can be uploaded.
NFR13	Authorization checks must be enforced server-side.

Implementation
AUTHETICATION

```
@app.route("/register", methods=["POST"])
def register_post():
    username = request.form["username"]
    email = request.form["email"]
    password = request.form["password"]
    password_r = request.form["password_r"]

    if not re.match(email_pattern, email):
        return render_template("register.html", error="Invalid email format.")

    if re.match(username_pattern, username):

        conn = get_db_connection()
        cur = conn.cursor()

        cur.execute("SELECT * FROM users WHERE username = %s OR email = %s", (username, email))
        user = cur.fetchone()

        cur.close()
        conn.close()
    else:
        return render_template("register.html", error="Passwords must contain at least 1 upper case letter,"
                                                      "1 special character and be at least 8 characters long."
                                                      "Usernames can only contain letters, numbers and underscore.")
```

During registration we first check if email and username is valid, then, if it is, we check if username or e-mail is already taken.

```
if not user:
    if password == password_r:

        if re.match(password_pattern, password):

            hashed = hash_password(password)

            conn = get_db_connection()
            cur = conn.cursor()

            token, expiry = send_token(email)

            cur.execute(
                """INSERT INTO users (username, email, password_hash,
                activation_token, activation_token_expiry) VALUES (%s, %s, %s, %s, %s)""",
                (username, email, hashed, token, expiry)
            )
            conn.commit()
            cur.close()
            conn.close()
            return "Register successful!"
        else:
            return render_template("register.html", error="Passwords must contain at least 1 upper case letter,"
                                                          "1 special character and be at least 8 characters long."
                                                          "Usernames can only contain letters, numbers and underscore.")
    else:
        return render_template("register.html", error="Passwords are not matching.")
else:
    return render_template("register.html", error="Unable to create account. Try again with different data.")
```

If the username and e-mail are not taken we procede. Password and repeated password must match. Finally if the password matches our regex for requirements the user is added to the database, but the account is not yet activated.

```
password_pattern = r'^(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*(),.?":{}|<>]).{8,}$'
username_pattern = r'^[A-Za-z0-9_]{3,}$'
email_pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
```

Theese regexes define username, e-mail and password requirements.

```
@app.route("/activate")
def activate():
    token = request.args.get("token")

    conn = get_db_connection()
    cur = conn.cursor()

    cur.execute("""
        UPDATE users
        SET is_activated = TRUE, activation_token = NULL, activation_token_expiry = NULL
        WHERE activation_token = %s
          AND activation_token_expiry > NOW()
        RETURNING id
    """, (token,))

    result = cur.fetchone()
    conn.commit()
    cur.close()
    conn.close()

    return "Activation successful!" if result else "Invalid or expired token."
```

Users will get an e-mail with the activation link. If the link is clicked within the next 24 hours the account will be activated.

```
@app.route("/login", methods=["POST"])
def login_post():
    username = request.form["username"]
    password = request.form["password"]

    password_pattern = r'^(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*(),.?":{}|<>]).{8,}$'
    username_pattern = r'^[A-Za-z0-9_]{3,}$'

    if re.match(username_pattern, username) or re.match(email_pattern, username):
        conn = get_db_connection()
        cur = conn.cursor()

        cur.execute("""
                SELECT password_hash, is_activated, email, activation_token, id, username
                FROM users
                WHERE username = %s OR email = %s
            """, (username, username))
        user = cur.fetchone()

        cur.close()
        conn.close()
    else:
        return render_template("login.html", error="Invalid username or password")
```

During log in procedure again we first check the validity of the username or e-mail before we insert it in a SQL queuery.

```
    if user and check_password(password, user[0]):
        if not user[1]:  # is_activated is FALSE
            return render_template("login.html",
                                   error="Please activate your account first.",
                                   resend_token=user[3],
                                   email=user[2])

        session.clear()
        session["user_id"] = user[4]
        session["username"] = user[5]

        if request.form.get("remember_me"):
            session.permanent = True
        else:
            session.permanent = False

        return redirect(url_for("index"))
    else:
        return render_template("login.html", error="Invalid username or password")
```

If the given password has the same hash as the hash stored in a database the password is correct. We also check if the account is activated. We create a user session, then redirect the user.

If the user wants to change the password they can request a password change token to be sent to their e-mail.

USERS

There are 3 types of users: admin, moderators and regular users. There is only one admin and he has the absoulte control over all other users and posts. Moderators have grained control, for example some can delete posts, some can ban other users, some can modify other users' permissions and some can do multiple things. Regular users can only post, delete their own posts, comment and vote on other posts. If user is banned, they cannot log in anymore, and their email cannot be reused.

POSTS

Posts consist of text title, keywrods and a media file which can be an image, gif or a video. Title and keywords are stored as a plain text in the database. XSS is prevented with Jinja2's automatic output escaping (text inside {{ }}) which ensures that during template rendering, everything inside those brackets is displayed as plain text. All database interactions are performed using paremterized SQL queries preventing SQL injection.

Uploaded media files are validated in multiple steps. First the file size is checked so it doesn't eceed the maximum allowed file size. Then the file name is converted to a secure version using Python's utils module. From that file name, the extension is extracted and checked. After that, the magic byte sniffing is used. Magic byte sniffing checks the first 4 KB of the file to ensure that the beginning of the file actually mathces the extension it says it is supposed to be. In the next step, the file is saved in a fixed directory with a randomly generated suffix so it prevents the overwrites.

```python
def _save_reencoded_image(file_storage, abs_path: str, kind: str, original: str):
    # prevent decompression bomb
    Image.MAX_IMAGE_PIXELS = 20_000_000  # tune (e.g. 20MP)

    img = Image.open(file_storage.stream)
    img.verify()  # validates structure
    file_storage.stream.seek(0)

    img = Image.open(file_storage.stream)

    # normalize orientation then drop metadata by re-encoding
    img = ImageOps.exif_transpose(img)

    # convert to a safe mode
    if img.mode not in ("RGB", "RGBA"):
        img = img.convert("RGB")

    # max dimensions
    max_side = 4000
    if max(img.size) > max_side:
        img.thumbnail((max_side, max_side))

    # save without EXIF/metadata
    save_kwargs = {}
    if kind == "jpg":
        if img.mode == "RGBA":
            img = img.convert("RGB")
        save_kwargs = {"quality": 85, "optimize": True}
        img.save(abs_path, format="JPEG", **save_kwargs)
    elif kind == "png":
        img.save(abs_path, format="PNG", optimize=True)
    elif kind == "webp":
        img.save(abs_path, format="WEBP", quality=85, method=6)
    else:
        return None

    size = os.path.getsize(abs_path)
    return {
        "file_path": abs_path,
        "kind": kind,
        "media_type": "image",
        "original_filename": original,
        "file_size_bytes": size,
    }
```

Images are additionaly sanitized by limiting the max pixells whichi prevents file bombs. After that the metadata is cleared so it doesn't leak any privacy info and correct exif orientation is applied. Then if the image is not encoded in RGB or RGBA it is converted to RGB. Finally, image is re-encoded and saved using Python's Pillow enconder which strips hidden chunks and malicious embedded content. GIF validation only checks the amount of rames confirming that the file is indeed a GIF.

Videos are not sanitozed specifically because the best way of doing so would require using ffmpeg and transcoding every video in the safe format.

```python
@posts_bp.route("/media/<path:filename>", methods=["GET"])
def media(filename):
    if "/" in filename or ".." in filename:
        abort(404)

    abs_path = os.path.join(current_app.config["UPLOAD_FOLDER"], filename)
    if not os.path.exists(abs_path):
        abort(404)

    resp = send_file(abs_path, conditional=True)
    resp.headers["X-Content-Type-Options"] = "nosniff"
    resp.headers["Cache-Control"] = "public, max-age=86400"
    return resp
```

The media files are not exposed directly as static files, instead they are served as flask endpoints as seen in the screenshot above. First, the file name is checked and it is rejected if it contains '/' or '..' so path traversal attacks are prevented. The nosniff header is added which instructs browser not to guess MIME type so it doesn't interpret the files as executable content. Caching is allowed because the file will not change once it is uploaded.
Post deletion is handled in a way that the post still stays visible for moderators and admin while being hidden from users and feed with a flag is_deleted in the database. Admin and moderators with permission can also recover deleted posts.
For all the operations requiring special permissions, the checks are done on the server side as well so even if someone manages to modify the fronted to appear as if they have the permission, the check is done again and it won't be bypassed.

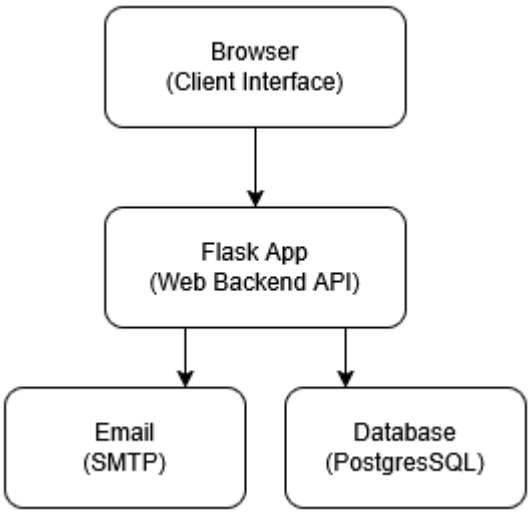
Tech stack

Frontend uses HTML5 and Jinja2 templates (form Flask).
Backend uses following Python libraries:

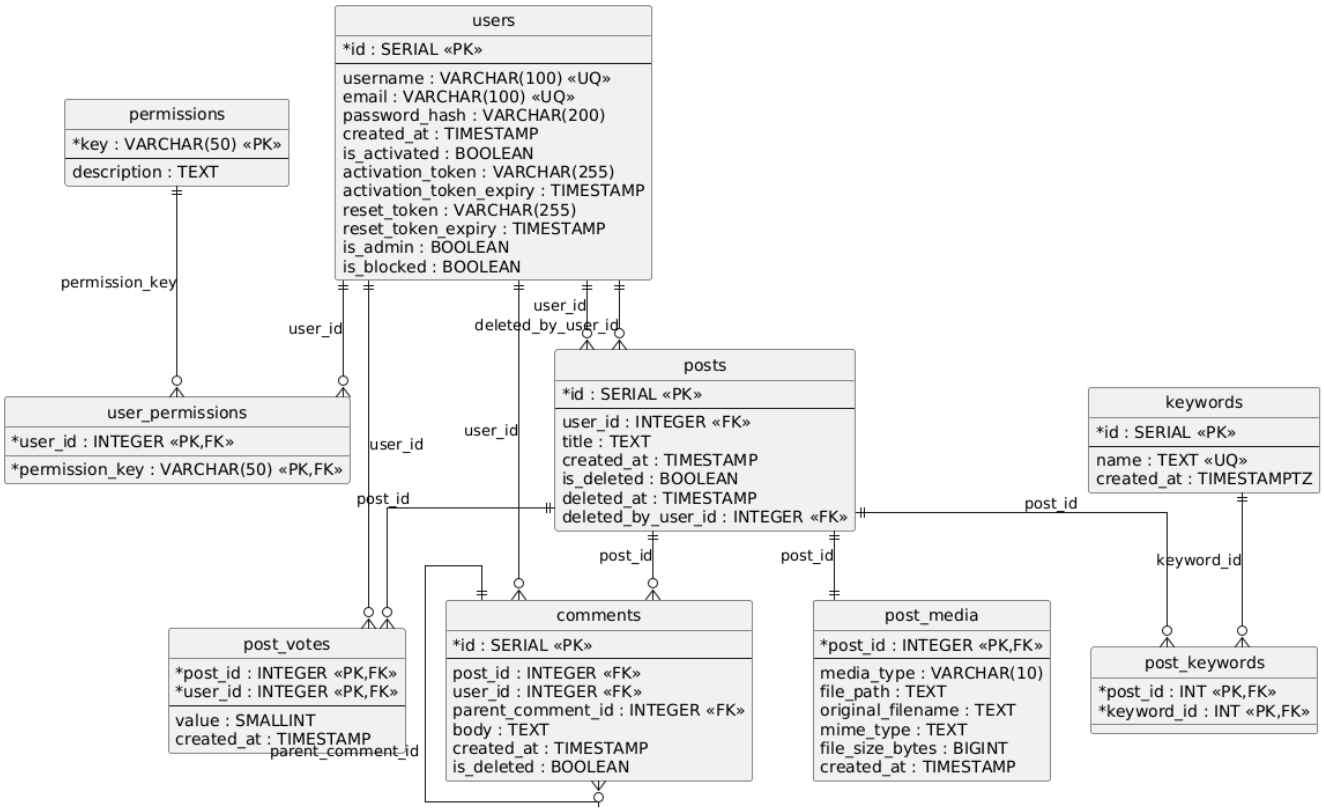
- Flask: used for routing, request handling, input validation, password hashing, token generation, sending emails, template rendering, session management
- bcrypt: password hashing
- secrets: random token generation
- re: regex validation
- smtplib: email sending
- dotenv: storing sensitive passwords (e.g. email)

Database layer consists of PostgreSQL and psycopg2 which is used for communication with the database.

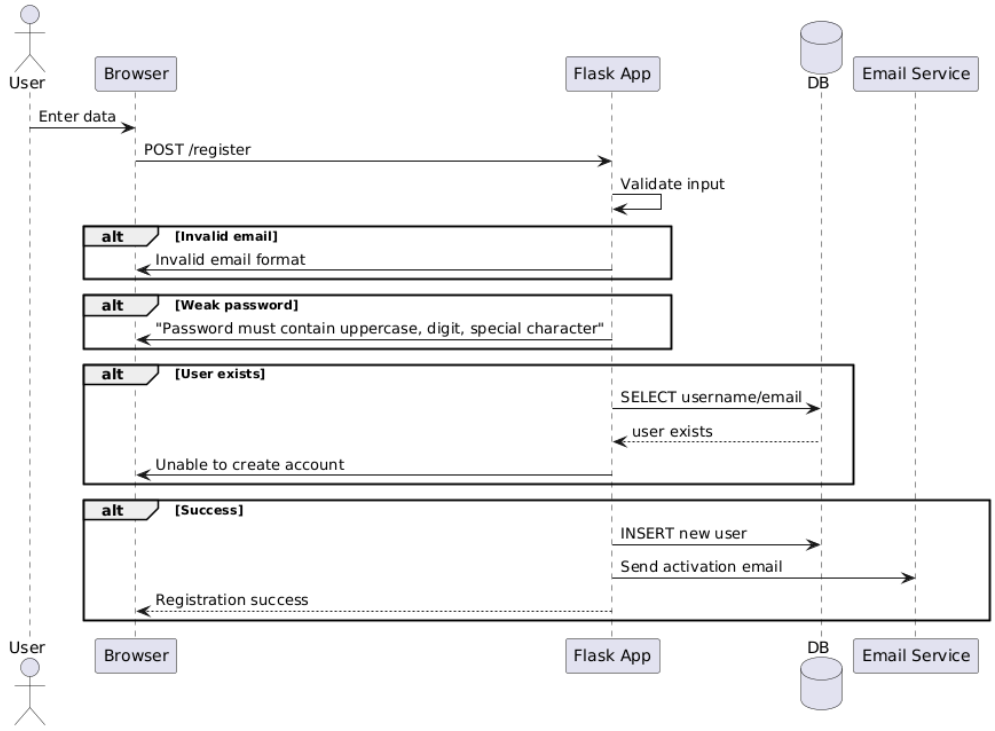
Architecture diagram


Browser
(Client Interface)
Flask App
(Web Backend API)
Email
(SMTP)
Database
(PostgresSQL)

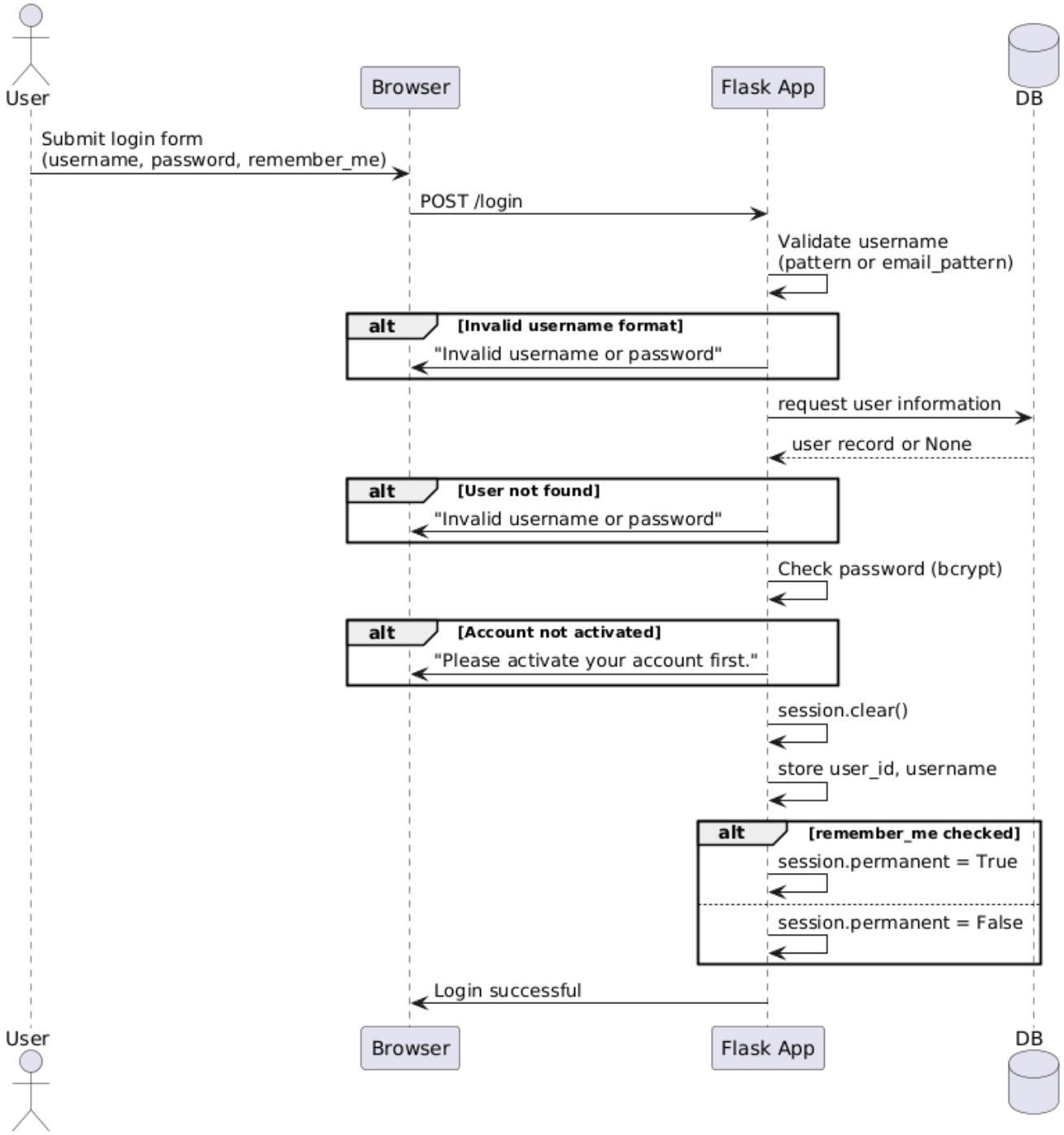
Database



Registration flow

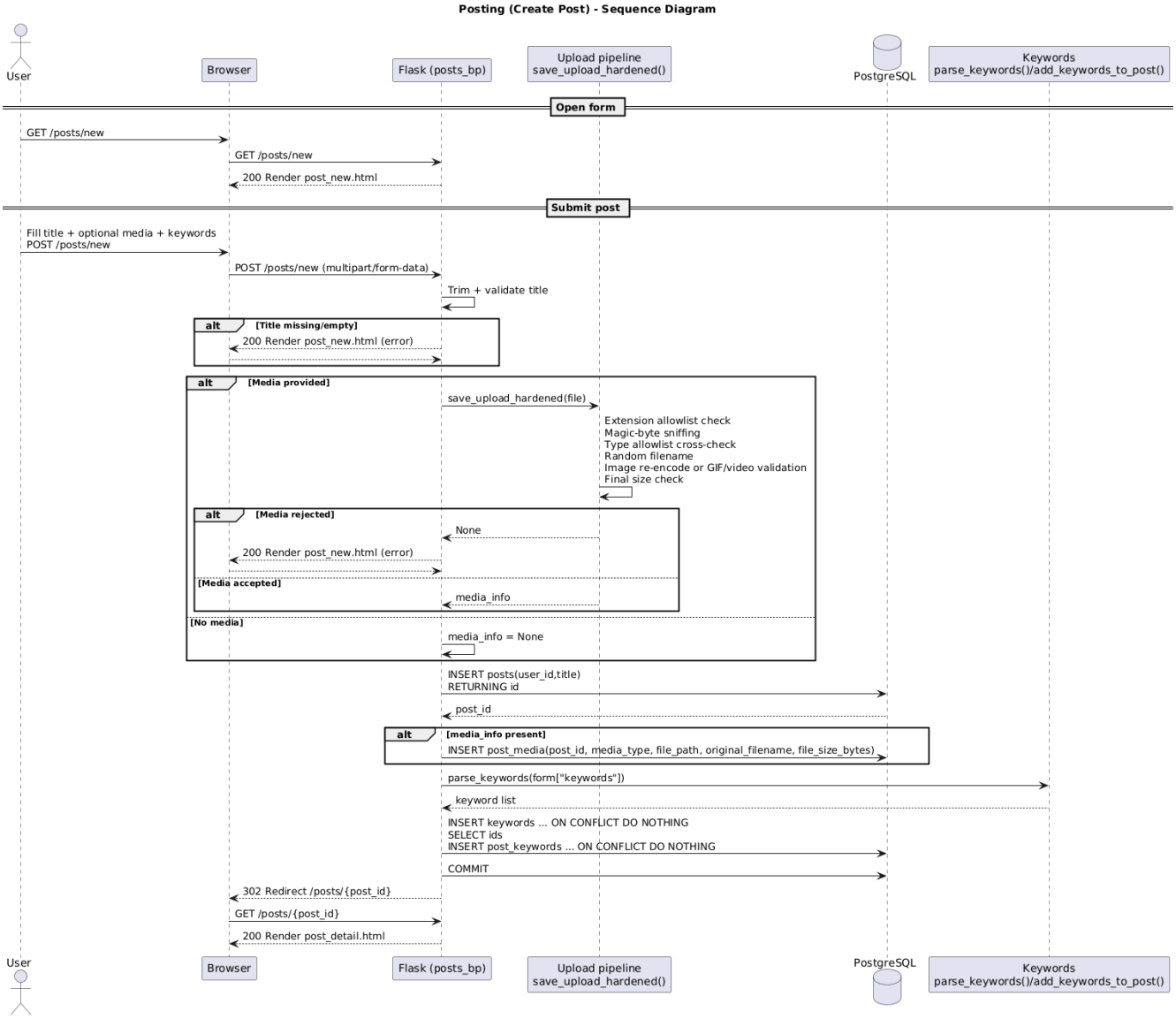

User
Browser
Flask App
DB
Email Service
Enter data
POST /register
Validate input
alt [Invalid email]
Invalid email format
alt [Weak password]
"Password must contain uppercase, digit, special character"
alt [User exists]
SELECT username/email
user exists
Unable to create account
alt [Success]
INSERT new user
Send activation email
Registration success
User
Browser
Flask App
DB
Email Service

Login flow

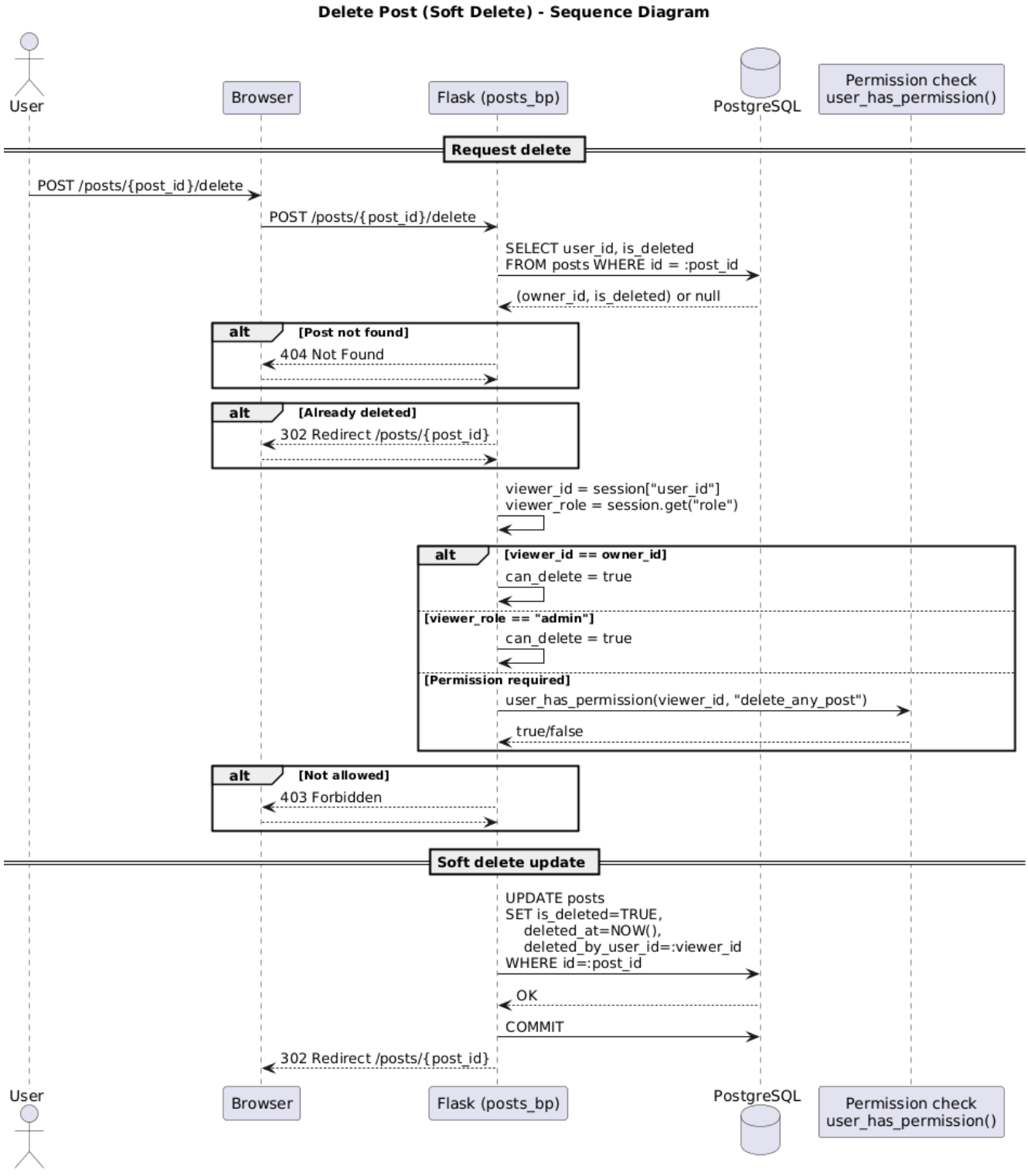


Resend confirmation email flow

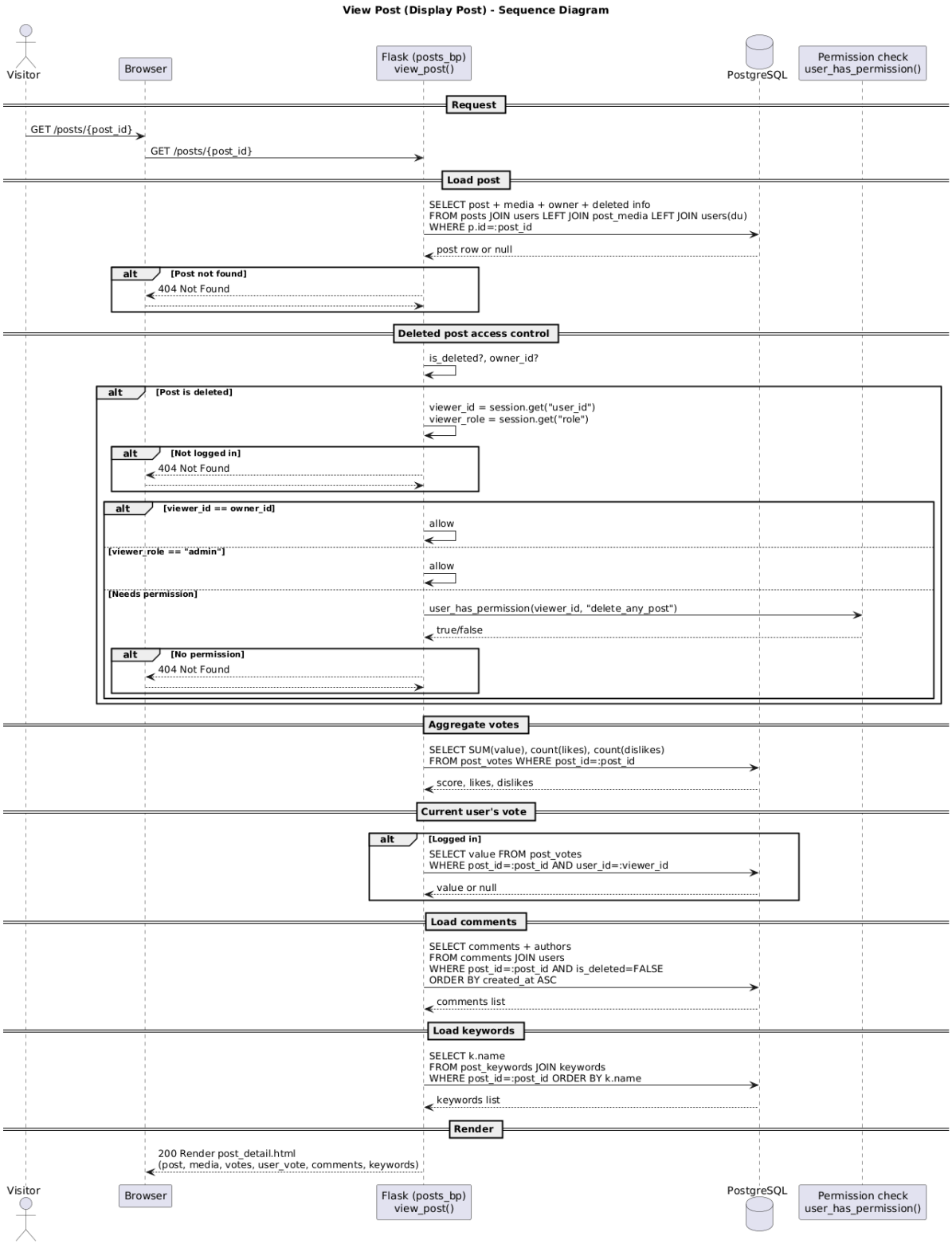
New post flow



Post deletion flow


Delete Post (Soft Delete) - Sequence Diagram
User
Browser
Flask (posts_bp)
PostgreSQL
Permission check user_has_permission()
Request delete
POST /posts/{post_id}/delete
POST /posts/{post_id}/delete
SELECT user_id, is_deleted FROM posts WHERE id = :post_id
(owner_id, is_deleted) or null
alt [Post not found]
404 Not Found
alt [Already deleted]
302 Redirect /posts/{post_id}
viewer_id = session["user_id"]
viewer_role = session.get("role")
alt [viewer_id == owner_id]
can_delete = true
[viewer_role == "admin"]
can_delete = true
[Permission required]
user_has_permission(viewer_id, "delete_any_post")
true/false
alt [Not allowed]
403 Forbidden
Soft delete update
UPDATE posts SET is_deleted=TRUE, deleted_at=NOW(), deleted_by_user_id=:viewer_id WHERE id=:post_id
OK
COMMIT
302 Redirect /posts/{post_id}

Displaying a post flow



Loading media flow