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.

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.

Implementation

```
@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.

```python
@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.

```python
@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.

```
@app.route("/forgot-password", methods=["GET", "POST"])
def forgot_password():
    if request.method == "POST":
        email = request.form["email"]

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

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

        if user:
            token = secrets.token_urlsafe(32)

            cur.execute("""
                UPDATE users
                SET reset_token=%s, reset_token_expiry=NOW() + INTERVAL '1 hour'
                WHERE email=%s
            """, (token, email))
            conn.commit()

            send_reset_email(email, token)

        cur.close()
        conn.close()

        return render_template("forgot_password.html", message="If the email exists, a reset link was sent.")

    return render_template("forgot_password.html")
```

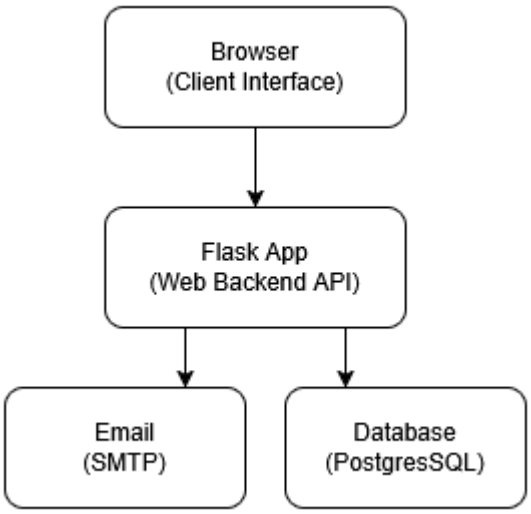
If the user wants to change the password they can request a password change token to be sent to their e-mail.

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.

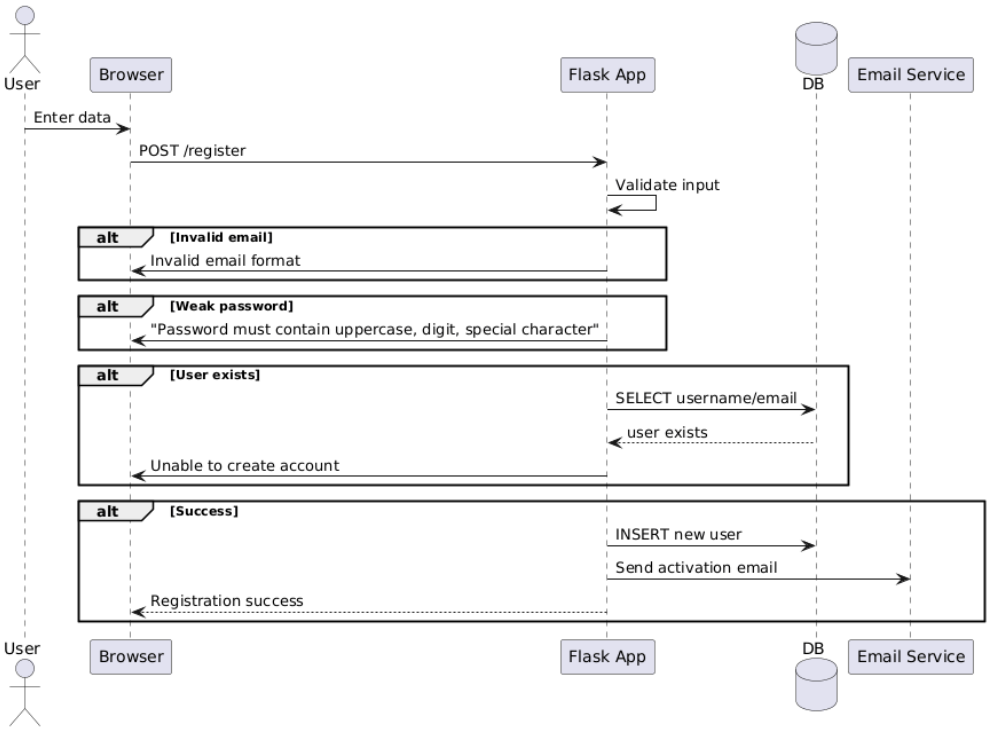
Architecture diagram



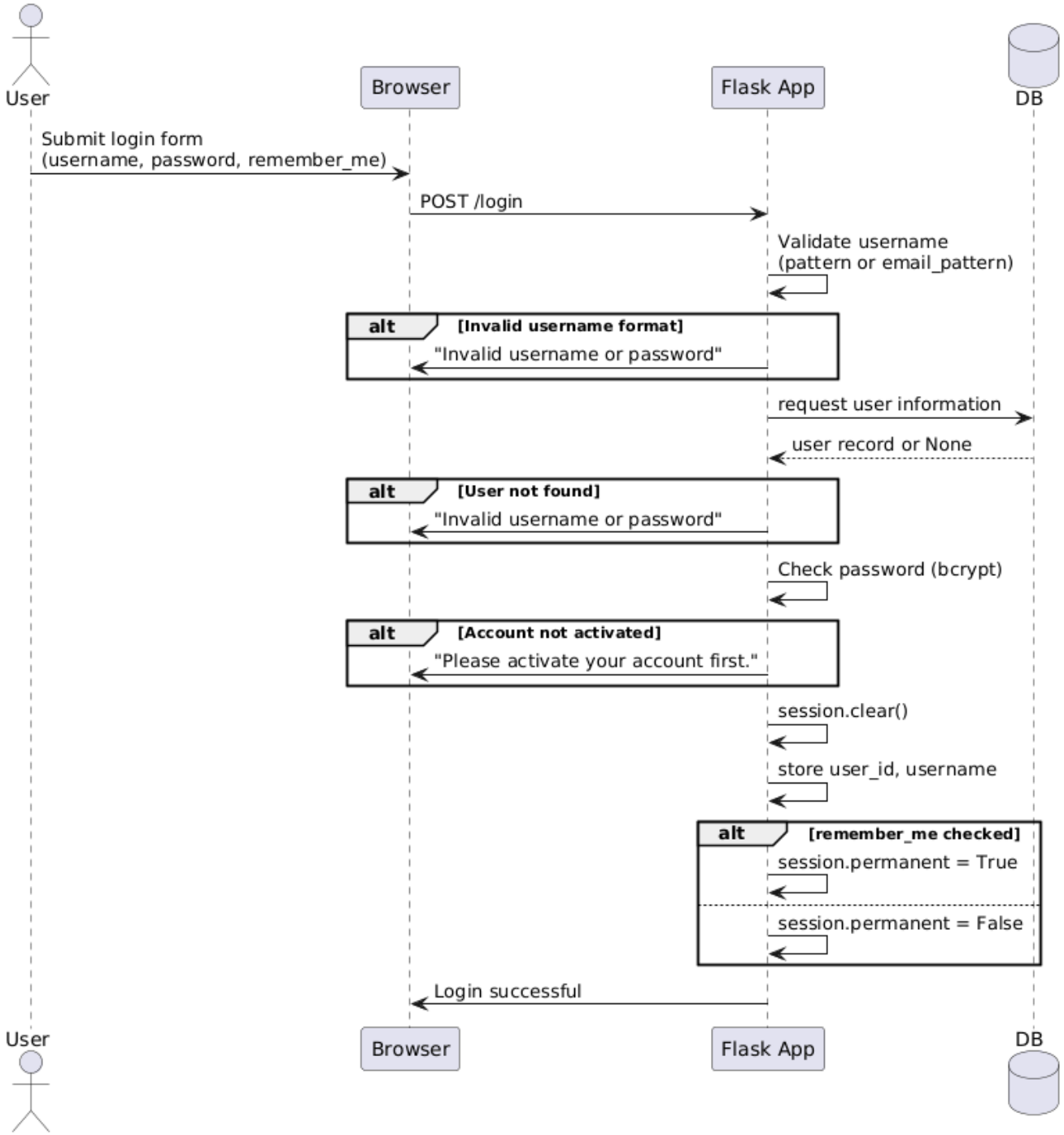
Database

Table	
PK	id
	username (UNIQUE, NOT NULL)
	email (UNIQUE, NOT NULL)
	pasword_hash (NOT NULL)
	is_activated (DEFAULT FALSE)
	activation_token
	activation_token_expiry
	created_at (DEFAULT NOW())
	reset_token
	reset_token_expiry

Registration flow



Login flow


User
Browser
Flask App
DB
Submit login form
(username, password, remember_me)
POST /login
Validate username
(pattern or email_pattern)
alt
[Invalid username format]
"Invalid username or password"
request user information
user record or None
alt
[User not found]
"Invalid username or password"
Check password (bcrypt)
alt
[Account not activated]
"Please activate your account first."
session.clear()
store user_id, username
alt
[remember_me checked]
session.permanent = True
session.permanent = False
Login successful
User
Browser
Flask App
DB

Resend confirmation email flow