Application Security – Laboratories – LAB03

User registration process (Part A)

Gaillard Théo
theo.gaillard@student.put.poznan.pl
Quivron Emile
emile.quivron@student.put.poznan.pl

Supervisor: Michał Apolinarski, Ph.D.
Politechnika Poznańska

December 2, 2025

Component Short Description

The user registration module is a secure software component responsible for creating new user accounts within the system. Its primary purpose is to allow new users to provide personal and authentication information, validate that information securely on the server side, and store the finalized account data in the application database.

Collected data

The component collects the following data from the user during registration:

- Email address – used as the unique identifier for the account and for sending activation tokens.
- Password – chosen by the user and processed using strong hashing before storage.
- Optional user metadata – such as username or full name, depending on system requirements.

Security assumptions and objectives

- The server is assumed to operate over a secure HTTPS channel to ensure confidentiality during transmission.
- All data submitted by users is considered untrusted and must be validated server-side to prevent injection attacks, malformed input, or exploitation attempts.
- Passwords must never be stored in plain text; they are protected using a secure one-way hashing function with salting.
- A unique activation token is generated for each new registration to confirm ownership of the provided email address.
- The module supports error handling and defensive measures throughout the process, including input validation, token verification, storage integrity constraints, and optional security hardening (rate limiting, advanced password policies, and event logging).

Component Requirements

Functional Requirements

FR-01 The system must provide a user registration form for entering required data.

FR-02 The server must validate all submitted fields for correctness, integrity, and constraints.

FR-03 The system must verify that the email address is not already registered.

FR-04 The system must hash passwords securely before storing them.

FR-05 The system must generate a unique account activation token after registration.

FR-06 The system must send an activation link containing the token to the user's email address.

FR-07 The system must activate the account by validating the token upon request.

FR-08 The system must handle invalid or expired tokens with clear error messages.

FR-09 The module must store user data (email, hashed password, token status, timestamps).

FR-10 The system should log validation or activation errors.

Optional Functional Requirements (Bonus Features)

OFR-01 Display a password strength meter during user input.

OFR-02 Enforce an advanced password policy (length, complexity, blacklist, etc.).

OFR-03 Perform email verification through DNS MX lookup or SMTP handshake.

OFR-04 Apply email domain restrictions via whitelist or blacklist.

OFR-05 Support registration using invitation tokens.

OFR-06 Implement CAPTCHA or rate limiting to prevent automated or abusive submissions.

OFR-07 Improve activation token security (short lifetime, association with device/IP, etc.).

OFR-08 Log successful and failed registration attempts as security events.

Non-Functional Requirements

Security Requirements

NFR-S01 Passwords must be stored using secure, modern cryptographic hashingitemize.

NFR-S02 Sensitive information (passwords, tokens) must not be logged.

NFR-S03 All communication between client and server should use HTTPS.

Usability Requirements

NFR-U01 Error and validation messages from the frontend must be clear and user-friendly.

NFR-U02 Registration should require no more than one user interaction step before activation.

Performance Requirements

NFR-P01 : Registration requests should be processed with low latency.

NFR-P02 : The system should scale to the required number of concurrent registrations.

Reliability Requirements

NFR-R01 : Duplicate account creation with the same email must be prevented.

NFR-R02 : Activation tokens must have a configurable expiration period.

NFR-R03 : Token validation must fail safely in cases of expiration or tampering.

Maintainability Requirements

NFR-M01 : The module must be structured with separate layers for validation, logic, and data access.

NFR-M02 : Configuration values must be stored in an accessible JSON configuration file.

Tech Stack

Frontend

- HTML5 and CSS3: For building the registration and activation pages.
- JavaScript: Handles client-side input validation (email format, password strength meter, etc.).
- Optional: Bootstrap or TailwindCSS for quick UI styling and responsive layouts.

Backend

- Python 3: Core application logic and registration workflows run here.
- Flask: Web framework for routing (e.g., `POST /register` and `GET /activate/<token>`), input validation, and returning HTML or JSON responses.

Security & Cryptography

- `bcrypt`: For secure password hashing with salt.
- `secrets` (Python standard library): For generating secure random activation tokens.

Database

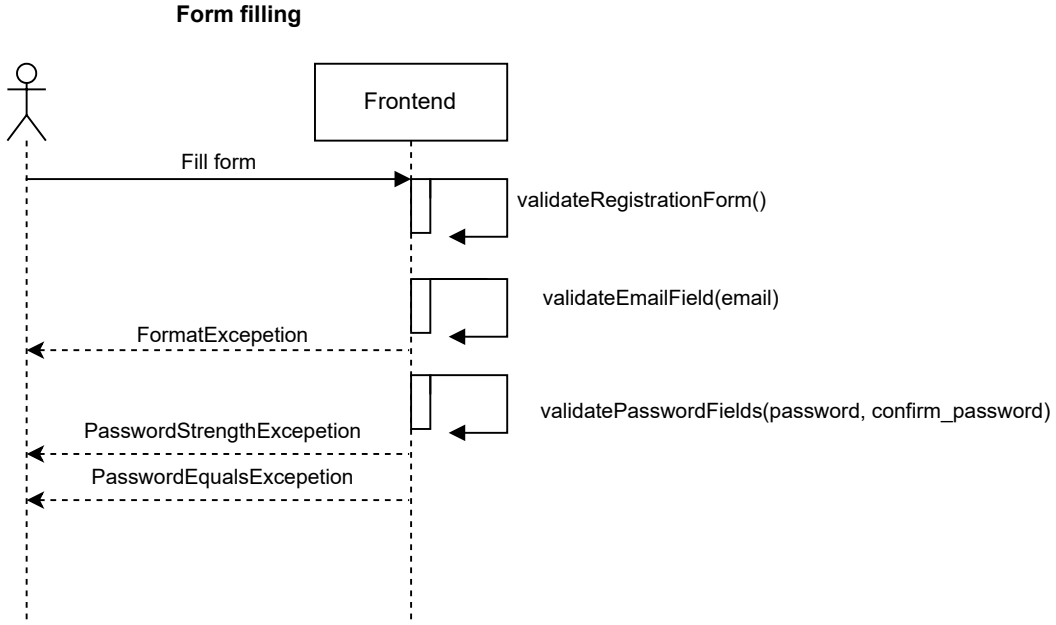
- SQLite3: local file-based relational database accessed using Python's built-in `sqlite3` module.
- Example tables and columns:
  - `users`: `id`, `email`, `password_hash`, `created_at`, `activated` (boolean)
  - `activation_tokens`: `token`, `user_id`, `expires_at`
- Constraints should ensure unique emails, token expiration rules, and referential integrity.

Web Server

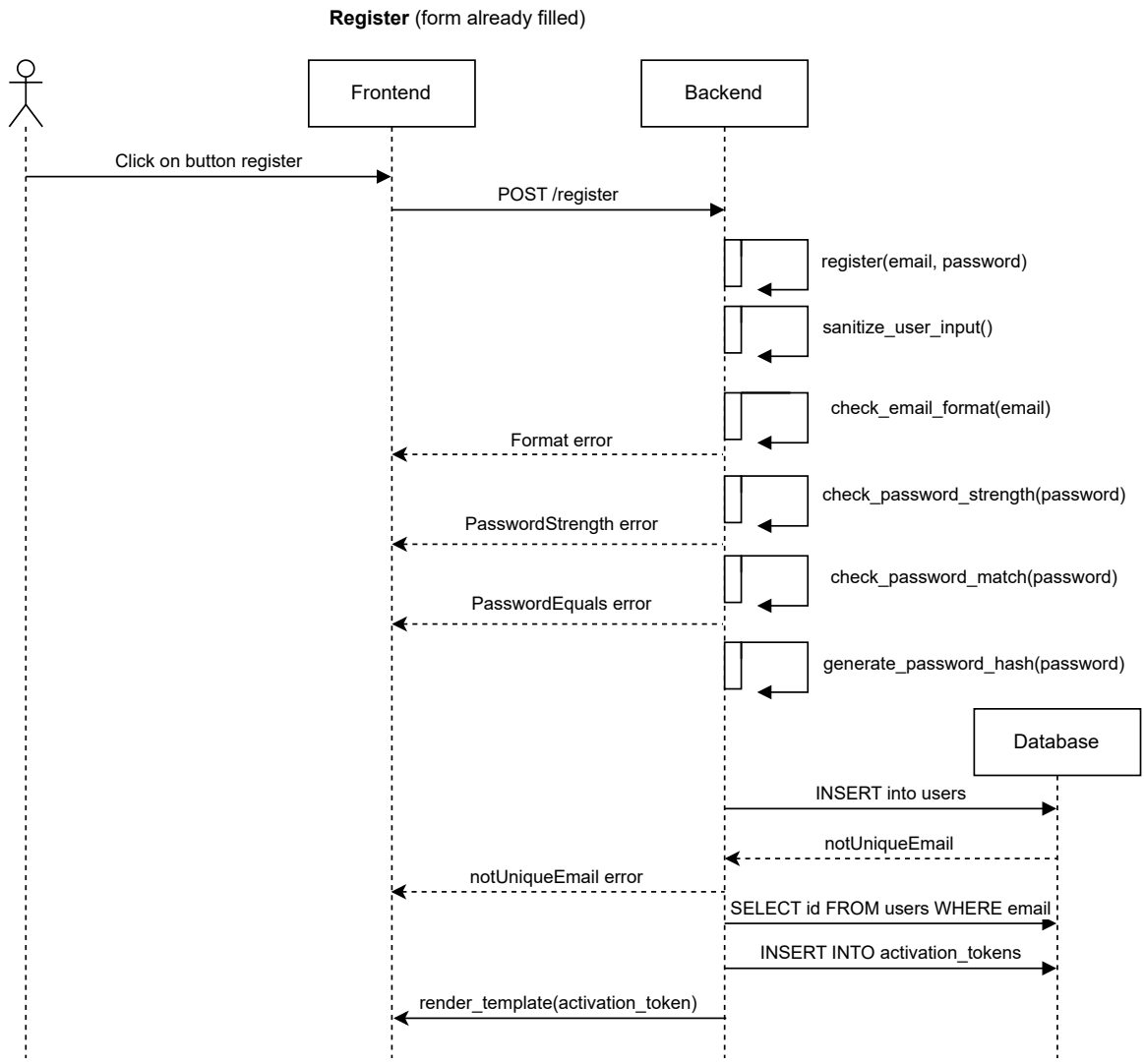
- `nginx` as reverse proxy to the Python application (via Gunicorn); serves static files and handles TLS termination.

UML Diagram

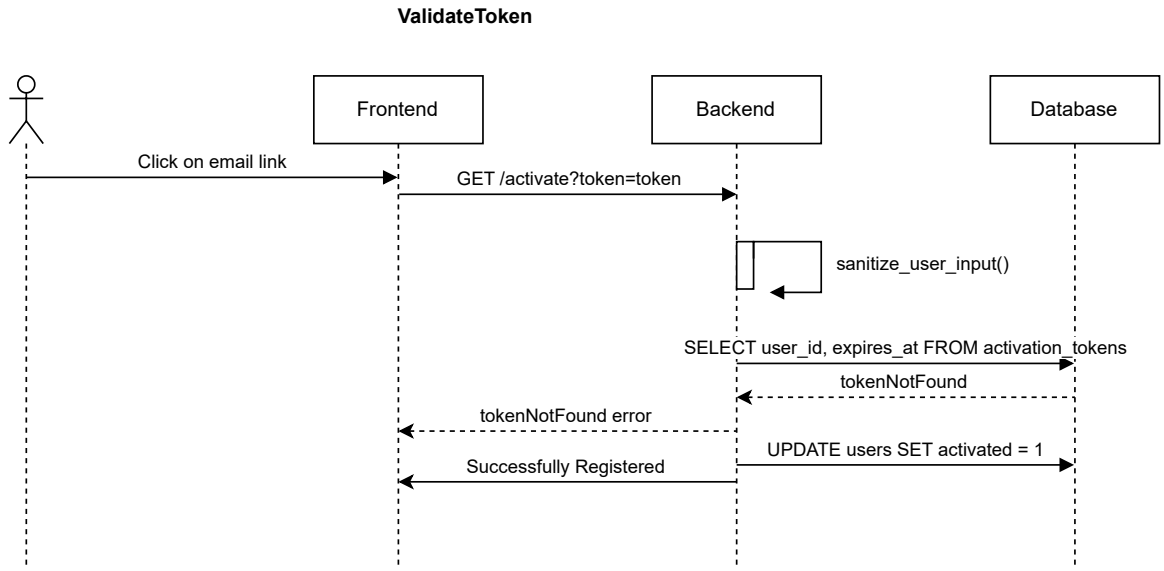
User Form Filling Process



User Registration Process



Token Validation Process



0.1 User registration flow presented

When a user wants to register on our platform, they will first access the registration page served by our Flask backend. The user fills out the registration form with their email and password. On the client side, JavaScript performs preliminary validation checks, such as ensuring the email format is correct and the password meets basic strength requirements. This immediate feedback enhances user experience by catching simple errors before submission.

Here's an example of the potential errors :

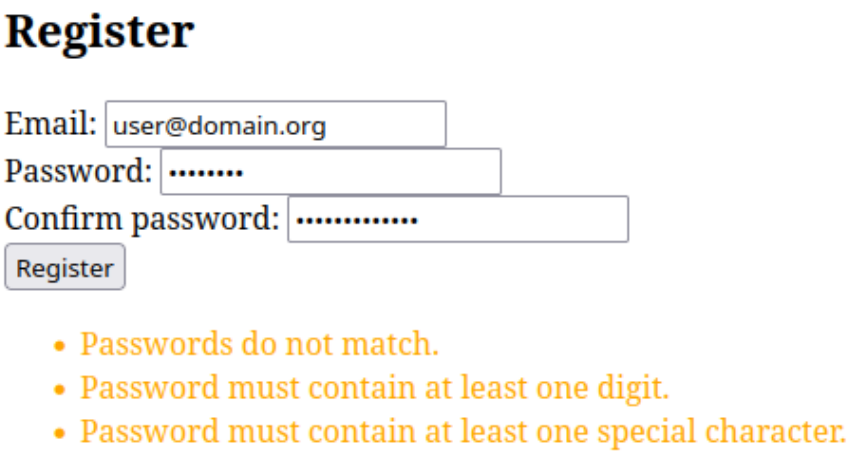


Figure 1: Example of validation error messages shown to the user during registration (email format, password strength, and required fields).

Once the user submits the form, the data is sent via a POST request to the Flask backend at the

`/register` endpoint. The backend performs comprehensive server-side validation to ensure all fields are correct. This includes checking that the email format is valid, it does not contain dangerous patterns (to prevent injection attacks), verifying that the password meets security policies (length, complexity). Then, the backend will try to save this new user in the database, ensuring the email is unique. The database also contains the user activation status. If any validation fails, the backend responds with appropriate error messages, which are then displayed to the user.

If the email is already registered here's an example of the error message shown to the user :

Figure 2: Example of error message shown to the user when trying to register with an already registered email.

Upon successful validation and user creation, the backend generates a unique activation token using Python's `secrets` module. This token is stored in the database along with an expiration timestamp and the user id associated with it. This token is then displayed in the frontend for demonstration purposes (in a future version, the token will be sent via email, according to the requirements).

Example of token being displayed to the user after successful registration :

Figure 3: Example of activation token displayed to the user after successful registration.

When the user clicks the activation link, it sends a GET request to the `/activate/<token>` endpoint on the backend. The backend retrieves the token from the URL, validates it against the database (checking for existence and expiration), and if valid, updates the user's activation status to true. The user is then informed of the successful activation.

Example of successful activation message shown to the user :

If the token is invalid or expired, the backend responds with an error message, which is displayed

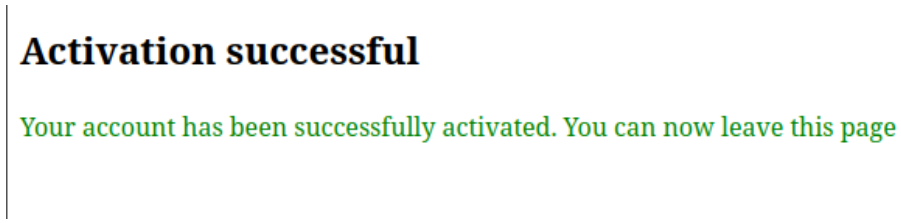


Figure 4: Example of successful activation message shown to the user.

to the user.

Example of error message shown to the user when trying to activate with an invalid or expired token :

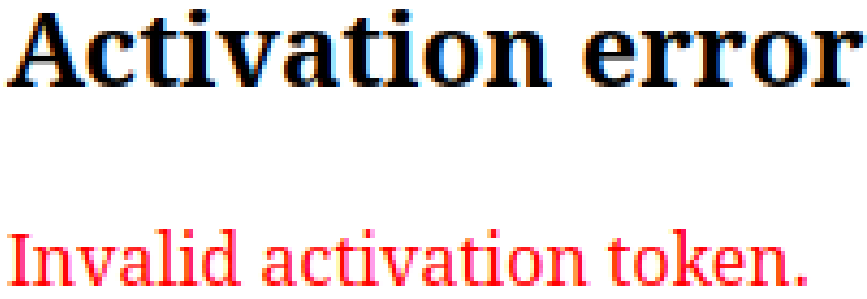


Figure 5: Example of error message shown to the user when trying to activate with an invalid or expired token.

Why Validate on Both Frontend and Backend

In the user registration process, we implement validation checks on both the frontend (client-side) and backend (server-side) for complementary reasons. Frontend validation provides immediate user feedback, improving UX by catching simple errors like password length or format issues before submission. It reduces unnecessary server requests and guides users to correct their input quickly.

However, frontend checks can be bypassed—attackers can disable JavaScript or manipulate requests directly. Backend validation is our security layer, defending against SQL injection, XSS, and malicious payloads. It enforces business rules and data integrity regardless of client behavior.

For example, our `validation.js` validates email format client-side:

```
const re = /^[^@\s]+@[^@\s]+\.[^@\s]+$/;
if (!re.test(email)) {
    errors.push("Email format is invalid.");
}
```

While our `app.py` backend sanitizes and re-validates server-side:

```
def sanitize_user_input(field_value: str, max_len: int = 255):
    if contains_dangerous_pattern(field_value):
        errors.append("User input error.")
```

The frontend enhances usability; the backend ensures security. Both working together create a robust system: fast feedback for legitimate users, strong protection against attackers. Never trust client-side validation alone—our backend is the final authority.