Registration Module Documentation

Kamil Grzymkowski (151908)

2025-12-23

Contents

1 Kamil Grzymkowski (151908), Application Security,

- **Course**: Application Security
- **Level**: Graduate/Master's
- **Focus Areas**:
  - Secure authentication and authorization
  - Password security and hashing
  - Email verification systems
  - Input validation and sanitization
  - Cryptographic token generation
  - Secure web application architecture

1.1 Exercise Description

This exercise demonstrates a complete registration and email verification system that implements modern web security practices. The registration module is responsible for:

- Secure user registration with validated input
- Cryptographically secure token generation
- Email verification workflow
- Password security with Argon2 hashing
- Defense against common attacks (replay, enumeration, injection, credential stuffing)
- Clear separation between frontend, backend, and database concerns

2 Component Description

2.1 Purpose

The registration module provides a secure mechanism for creating user accounts and verifying ownership of email addresses before granting full access to the application.

Its main goals are:

1. Ensure only valid, well-formed data is accepted.
2. Prevent user enumeration and information leakage.
3. Protect passwords and verification tokens using modern cryptography.
4. Enforce email verification before login succeeds.
5. Provide a structure that can be audited for security.

2.2 Responsibilities

- Accept registration requests from the frontend.
- Validate and normalize user input (username, email, password).
- Hash passwords using Argon2.
- Generate and store cryptographically secure verification tokens.
- Expose verification endpoints for email link handling.
- Protect against replay and token brute forcing.
- Provide clear, generic responses to the frontend (no leaking of internal state).

2.3 Data Collected

The registration module stores only the minimal data required for account creation and email verification.

User Table

- `id`: UUID or integer, primary key.
- `username`
- `email`
- `password_hash`
- `is_verified` (boolean)
- `created_at`, `updated_at`

Email Verification Token Table

- `id`: primary key.
- `user_id`: foreign key.
- `token_hash`: hash of the token, never the raw token.
- `expires_at`
- `consumed_at` (nullable)
- `created_at`

Standard technical metadata may be logged for operational and security purposes:

- Timestamp of registration attempts.
- Source IP (for rate limiting and incident analysis).
- Generic success/failure markers (without sensitive details).

2.4 Security Assumptions

- All traffic is served over HTTPS in production.
- Secrets such as database credentials and key material are not stored in source code.
- The database is not directly exposed to the internet (only the backend service can access it).
- Email delivery is handled by a trusted provider or simulated in development (e.g., logs instead of real emails).
- Clients do not bypass the frontend and always call public API endpoints (the backend still validates all input).
- User passwords are never logged, stored in plaintext, or returned to the client.

3 Component Requirements

3.1 Functional Requirements

3.1.1 FR-1: User Registration

ID: FR-1

Description: The system must allow a new user to register with username, email, and password.

Details:

- Registration is exposed as an HTTP endpoint on the backend.
- The frontend provides a dedicated registration form.
- A success response does not reveal whether the email or username already exists.

3.1.2 FR-2: Email Verification

ID: FR-2

Description: The system must require email verification before enabling a full login.

Details:

- On successful registration, a verification token is generated and stored in hashed form.
- A verification link containing the token is presented (or logged in dev mode).
- Clicking the link (or calling the verification endpoint) marks the user as verified if the token is valid and unexpired.

3.1.3 FR-3: Input Validation

ID: FR-3

Description: All registration inputs must be strictly validated on backend and frontend.

Details:

- Username: reasonable length, allowed characters, normalized format.
- Email: checked against a robust email format and normalized (e.g., lowercase).
- Password: minimum length and complexity requirements enforced.

3.1.4 FR-4: Error Handling

ID: FR-4

Description: The system must provide generic and safe error messages.

Details:

- No distinction between "email already registered" and "user not found" in responses.
- Server errors are logged on the backend but not exposed to users.
- Client receives short, user-friendly messages.

3.1.5 FR-5: Token Lifecycle

ID: FR-5

Description: Verification tokens must be single-use and time-limited.

Details:

- Each token can be used only once.
- Tokens have an explicit expiration time.
- Expired or consumed tokens are rejected with a generic error.

3.1.6 FR-6: Health Check

ID: FR-6

Description: The module must expose a simple health endpoint for monitoring.

Details:

- Allows external systems to check that the backend is running.
- Does not expose internal or user data.

3.2 Non-Functional Requirements

3.2.1 NFR-1: Security

ID: NFR-1

Description: All security best practices must be followed.

Details:

- No plaintext passwords or tokens stored.
- Verification tokens stored only as hashes.
- Generic error messages (prevent user enumeration).
- HTTPS enforced in production.
- No PII in URLs beyond what is unavoidable (e.g., query token is opaque).
- Planned rate limiting to soften brute force and abuse.
- Input validated both on frontend and backend.

3.2.2 NFR-2: Performance

ID: NFR-2

Description: The module must handle typical concurrent registration loads.

Details:

- Database operations complete in under 100 ms under normal load.
- Token generation in under 50 ms.
- API responses in under 200 ms for typical requests.
- Argon2 parameters chosen to balance security and latency.

3.2.3 NFR-3: Reliability

ID: NFR-3

Description: The module must be fault-tolerant.

Details:

- Database transactions roll back on any failure.
- Partial registrations are cleaned up if processing fails mid-flow.
- Errors are logged and can be correlated across services.
- Verification tokens cannot be reused after consumption.

3.2.4 NFR-4: Usability

ID: NFR-4

Description: The module must provide clear user feedback.

Details:

- Frontend shows field-level validation errors.
- Password strength indicators and hints are provided where appropriate.
- After registration, user receives clear instructions to check email.
- After verification, user is guided to login.

4 Component Architecture

4.1 Technology Stack

Frontend

- Vue 3 with TypeScript
- Vuetify for UI components and consistent styling
- Axios or fetch-based API service layer
- Protobuf-generated TypeScript client (`frontend/src/generated/api.ts`)

Backend

- Rust
- Axum for HTTP routing
- SQLx for asynchronous database access
- Argon2 for password hashing
- SHA2 for token hashing
- `rand_core` / `rand` for cryptographically secure random values
- Protobuf-generated Rust types (`backend/src/generated/api.v1.rs`)

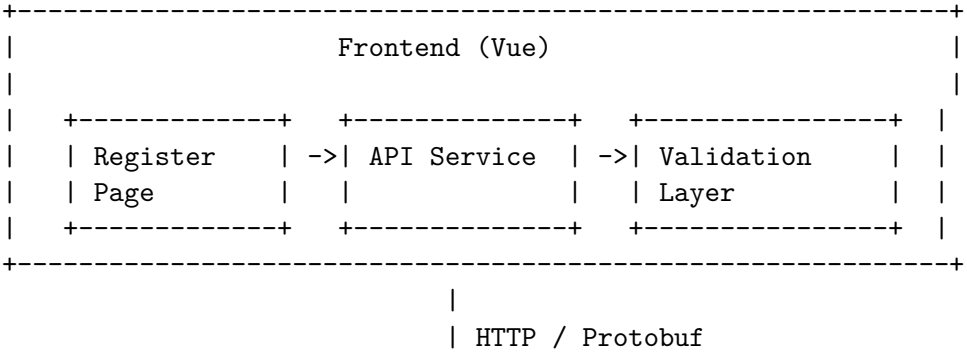
Database

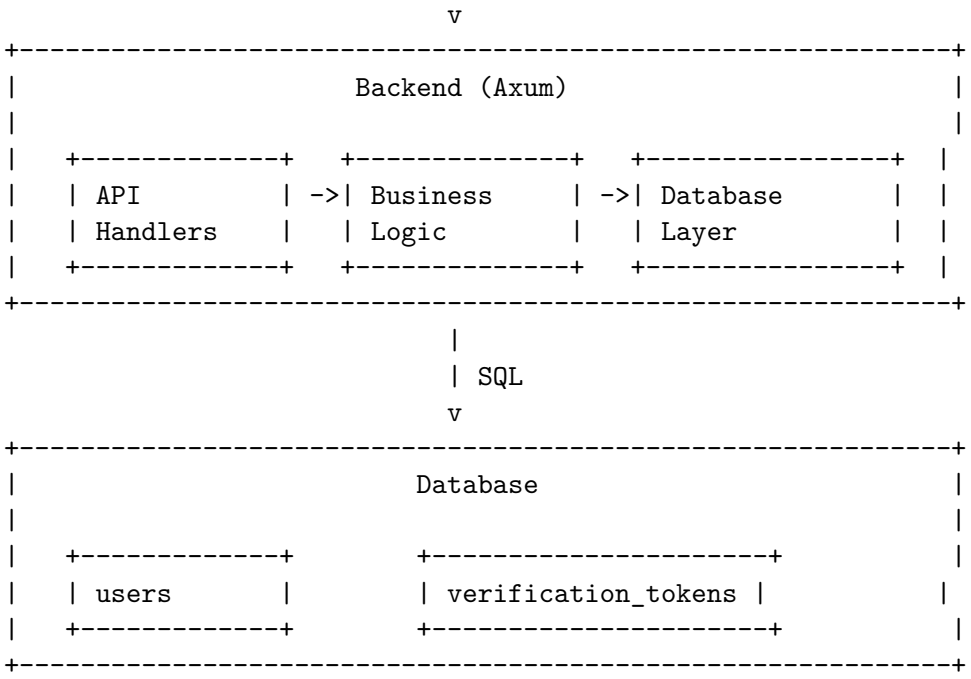
- Relational database (e.g., PostgreSQL or SQLite, configured per environment)
- Tables for users and verification tokens with strong constraints

API Format

- Protocol Buffers for message definitions (`proto/api.proto`)
- HTTP endpoints exchanging Protobuf-backed JSON or binary messages (depending on configuration)

4.2 High-Level Architecture Diagram





4.3 Data Flow

1. User opens the registration page on the frontend.
2. User fills in username, email, and password.
3. Frontend performs client-side validation and sends a registration request to the backend.
4. Backend validates the input again and:
   - Hashes the password with Argon2.
   - Creates a user record in the database.
   - Generates a random verification token and stores its hash in `verification_tokens`.
5. Backend returns a success response.
6. In development mode, an email-like message (with a verification link containing the raw token) is logged or displayed to simulate delivery.
7. User follows the verification link.
8. Frontend sends the token to the verification endpoint.
9. Backend:
   - Locates the token by its hash.
   - Verifies expiry and that it is unused.
   - Marks the token as consumed.
   - Marks the associated user as verified.
10. Backend returns a success response; frontend shows a confirmation message.

5 Database Structure

5.1 Users Table

Column	Type	Constraints	Notes
`id`	UUID / INTEGER	Primary key	Unique user identifier
`username`	TEXT	Unique, NOT NULL	Indexed for lookup

Column	Type	Constraints	Notes
email	TEXT	Unique, NOT NULL	Indexed, stored normalized
password_hash	TEXT	NOT NULL	Argon2 hash, never plaintext
is_verified	BOOLEAN	NOT NULL, default false	Becomes true after verification
created_at	TIMESTAMP	NOT NULL, default now()	
updated_at	TIMESTAMP	NOT NULL, default now()	Updated on changes

Typical constraints and checks:

- Unique constraint on (username) and (email).
- Email stored in lowercase to avoid duplicates by case difference.
- Application logic prevents updates that would violate uniqueness.

5.2 Verification Tokens Table

Column	Type	Constraints	Notes
id	UUID / INTEGER	Primary key	Internal identifier
user_id	UUID / INTEGER	NOT NULL, FK → users.id	Indexed for lookups
token_hash	TEXT	NOT NULL	Hash of random token, unique
expires_at	TIMESTAMP	NOT NULL	Tokens invalid after this timestamp
consumed_at	TIMESTAMP	NULLABLE	Set on successful verification
created_at	TIMESTAMP	NOT NULL, default now()	

Constraints and guarantees:

- token_hash is unique to prevent collisions.
- A token is considered valid only if:
  - now() <= expires_at
  - consumed_at IS NULL
- Foreign key user_id enforces referential integrity.

6 UML Sequence Diagrams

6.1 Registration Flow Sequence Diagram

6.2 Verification / Activation Flow Sequence Diagram

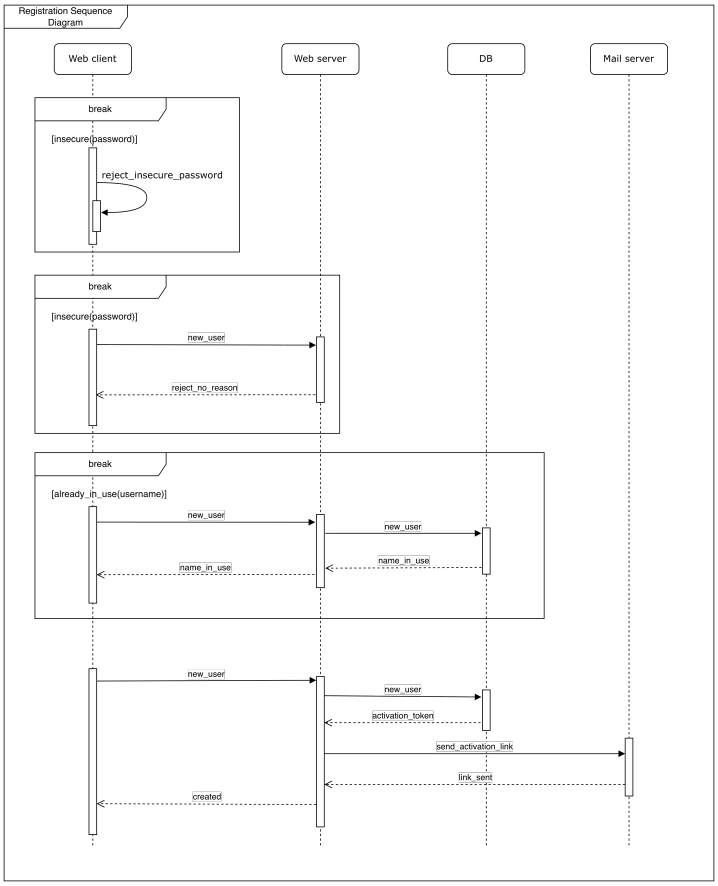


Figure 1: Registration Flow Sequence Diagram

7 Security Features Summary

7.1 Implemented Security Measures

Password Hashing

- Argon2 is used for password hashing.
- Parameters tuned for modern hardware to resist brute force.
- Passwords are never stored or logged in plaintext.

Token Security

- Verification tokens are random, high-entropy values.
- Only token hashes are stored in the database.
-
- Tokens are not predictable and cannot be derived from user data.

Input Validation and Normalization

- Backend validates all fields regardless of frontend checks.
- Email is normalized (e.g., lowercased).
- Usernames are restricted to safe characters and reasonable lengths.
- Passwords must meet minimum length and complexity requirements.

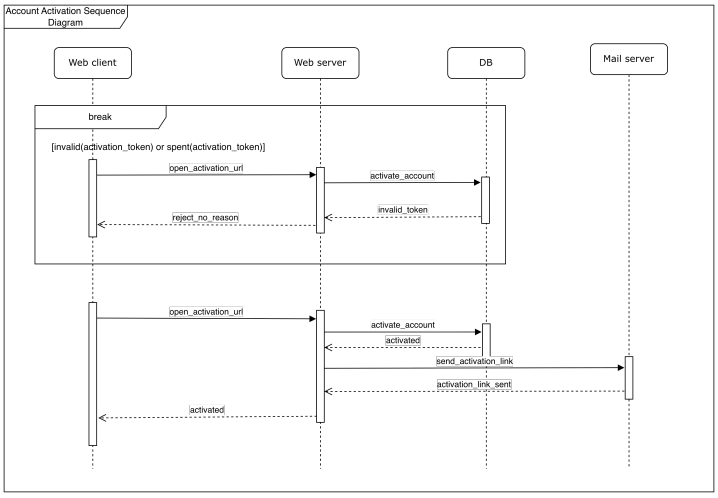


Figure 2: Verification Flow Sequence Diagram

Error Handling and Enumeration Resistance

- Error responses are generic: no difference between invalid credentials and unverified accounts, for example.
- Registration responses do not reveal if an email or username already exists.
- Detailed errors are logged server-side only.

Transport Security

- HTTPS enforced in production to protect credentials and tokens in transit.
- No sensitive data is transmitted unencrypted over the network.

Database Security

- Prepared statements are used (via SQLx) to prevent SQL injection.
- Separate database credentials for environments.
- Password and token hashes are not reversible.
- Access to the database is limited to the backend service.

Replay and Abuse Protection

- Tokens can be used only once and are marked as consumed.
- Expired tokens are rejected.
- Future enhancements include rate limiting and CAPTCHA.

7.2 Future Security Enhancements

Planned or potential enhancements:

- Rate limiting on registration and verification endpoints.
- CAPTCHA integration for bot and script mitigation.
- Production email service integration (SMTP or API-based).
- Scheduled cleanup of expired and consumed tokens.

- Session management using JWT or secure server-side sessions.
- Optional two-factor or multi-factor authentication.
- Password reset flow with secure tokens.
- Account lockout or step-up verification after repeated failures.

8 API Reference

This section documents the main endpoints implemented by the registration module. Protobuf definitions are provided in `proto/api.proto` and are compiled into both backend and frontend code.

8.1 Endpoints Overview

- `POST /api/register` – Create a new user account and issue a verification token.
- `POST /api/verify-email` – Verify an email using a token.
- `GET /api/health` – Lightweight health check.

8.2 `POST /api/register`

Registers a new user account and issues an email verification token.

Request (Protobuf)

```
message RegistrationRequest {
  string username = 1;
  string email = 2;
  string password = 3;
}
```

Response (Protobuf)

```
message ApiResponse {
  bool success = 1;
  string message = 2;
}
```

Behavior

- Performs full backend validation of all fields.
- If validation fails, returns `success = false` with a generic message.
- On success:
  - Creates a new user with `is_verified = false`.
  - Hashes and stores the user's password.
  - Generates a verification token and stores its hash with expiry.
  - In development, logs or displays a mock email including the verification link.

Status Codes

- **200 OK** – Request processed;
  - `success = true` on successful registration.
  - `success = false` if validation failed.
- **400 Bad Request** – Malformed payload or missing fields.
- **500 Internal Server Error** – Unexpected server error.

8.3 `POST /api/verify-email`

Verifies an email address using a token from the verification link.

Request (Protobuf)

```
message VerificationRequest {
  string token = 1;
}
```

Response (Protobuf)

```
message ApiResponse {
  bool success = 1;
  string message = 2;
}
```

Behavior

- Hashes the incoming `token` and attempts to find a matching record.
- Checks token expiry and whether it has already been consumed.
- If valid:
  - Marks the associated user as verified.
  - Marks the token record as consumed.
- Responds with `success = true` on successful verification, or `success = false` with a generic message otherwise.

Status Codes

- **200 OK** – Verification processed; `success` indicates outcome.
- **400 Bad Request** – Malformed payload.
- **404 Not Found** (optional, may still return 200 with `success = false`) – No valid token found.
- **500 Internal Server Error** – Unexpected server error.

8.4 `GET /api/health`

Simple health check endpoint.

Behavior

- Returns a basic success payload indicating the backend is up.
- Does not access user data or reveal internal state beyond health.

Status Codes

- **200 OK** – Backend is healthy.

9 Implementation Files

This section lists the core implementation files that make up the registration module.

9.1 Backend Files

- `backend/src/main.rs`
  - Application entry point.
  - Configures Axum router and mounts registration and verification routes.
  - Initializes database connection and shared state.
- `backend/src/api.rs`
  - HTTP handlers for:
    - `POST /api/register`
    - `POST /api/verify-email`
    - `GET /api/health`
  - Parses Protobuf/JSON payloads and returns `ApiResponse` messages.
- `backend/src/db/mod.rs`
  - Database access layer.
  - Functions for:
    - Creating users.
    - Storing and looking up verification tokens.
    - Marking users as verified.
  - Encapsulates SQLx queries and transactions.
- `backend/src/generated/api.v1.rs`
  - Generated Rust types from `proto/api.proto`.
  - Shared message definitions for requests and responses.

9.2 Frontend Files

- `frontend/src/pages/register.vue`
  - Registration page.
  - Presents username/email/password form.
  - Displays validation errors and success messages.
- `frontend/src/pages/verify-email.vue`
  - Verification landing page.
  - Reads token (for example, from query parameters).
  - Calls verification API and shows user feedback.
- `frontend/src/components/UserRegistration.vue`
  - Reusable registration form component.
  - Handles field-level validation, user interaction, and emits submission events.

- `frontend/src/services/api.ts`
  - Frontend API service layer.
  - Functions such as `registerUser` and `verifyEmail` wrapping HTTP calls.
  - Uses generated Protobuf TypeScript definitions.
- `frontend/src/generated/api.ts`
  - Generated TypeScript client from `proto/api.proto`.
  - Type-safe request and response interfaces.

9.3 Proto Files

- `proto/api.proto`
  - Protobuf definitions for:
    - `RegistrationRequest`
    - `VerificationRequest`
    - `ApiResponse`
    - Any shared error or metadata types.
  - Single source of truth for backend and frontend API types.

9.4 Documentation Files

- `documentation/registration-module.md`
  - This document; primary reference for the registration module.
- `documentation/email-verify-feat.md`
  - Focused description of the email verification feature and its behavior.
- `documentation/email-verify-implementation.md`
  - Implementation details of the verification functionality and integration.

10 Usage Examples

10.1 Development Mode

Run backend and frontend in development:

```
# Backend
cd backend
cargo run -- --dev

# Frontend (separate terminal)
cd frontend
npm install
npm run dev
```

In development mode, the email verification link is typically logged to the console or terminal instead of sending a real email. This allows manual testing of the full verification flow without configuring an SMTP server.

10.2 Registration Flow Example

```
import { registerUser } from '@/services/api';

async function handleRegister(): Promise<void> {
  const response = await registerUser({
    username: 'john_doe',
    email: 'john@example.com',
    password: 'SecureP@ssw0rd123',
  });

  if (response.success) {
    // "Registration successful. Please check your email to verify your account."
  } else {
    // "Registration failed. Please check your details and try again."
  }
}
```

On the backend:

- A user record is created.
- The password is stored as an Argon2 hash.
- A verification token is generated and stored as a hash.
- A mock email with a link like `https://example.com/verify-email?token=<token>` is logged.

10.3 Verification Flow Example

```
import { verifyEmail } from '@/services/api';

async function handleVerify(): Promise<void> {
  const params = new URLSearchParams(window.location.search);
  const token = params.get('token');

  if (!token) {
    return;
  }

  const response = await verifyEmail(token);

  if (response.success) {
    // "Email verified successfully. You can now log in."
  } else {
    // "Verification failed or token has expired. Please request a new verification
        email."
  }
}
```

10.4 Libraries and Tools

- Axum – Rust web framework
- SQLx – Asynchronous SQL toolkit
- Argon2 – Password hashing algorithm
- SHA2 – Cryptographic hash functions
- `rand_core` / `rand` – Cryptographically secure RNG
- Protocol Buffers – API schema and type generation
- Vue 3 – Frontend framework

- Vuetify – Material Design component library

11 Version History

Version	Date	Changes
1.0	2025-12-23	Initial single-file registration module documentation

Document Generated: 2025-12-23
Last Updated: 2025-12-23
Status: Complete
Review Status: Ready for review