Registration Module Documentation (Final)

Kamil Grzymkowski (151908)

2026-01-11

Contents

1 Kamil Grzymkowski (151908), Application Security

- **Course**: Application Security
- **Level**: Graduate/Master's
- **Focus Areas**:
  - Secure user registration
  - Password security with Argon2 hashing
  - Email verification with secure tokens
  - Input validation (frontend and backend)
  - Cryptographic token generation and storage
  - Cross-platform validation via WebAssembly

1.1 Exercise Description

This exercise demonstrates a secure user registration module with email verification, implementing modern web security practices. The registration module is responsible for:

- Secure user registration with client-side and server-side validation
- Password strength validation with detailed scoring (0–7 scale)
- Cryptographically secure verification token generation
- Email verification workflow before account activation
- Password hashing using Argon2 algorithm
- Defense against common attacks (enumeration, injection, brute force)
- Shared validation logic between frontend (WASM) and backend (native Rust)

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 access to the application.

Its main goals are:

1. Ensure only valid, well-formed data is accepted through dual validation (frontend and backend).
2. Prevent user enumeration and information leakage through generic error responses.
3. Protect passwords using Argon2 hashing with random salts.
4. Store verification tokens as SHA256 hashes, never in plaintext.
5. Enforce email verification before the account can be used.
6. Automatically clean up expired tokens and unverified accounts.

2.2 Responsibilities

- Accept registration requests with username, email, and password.
- Validate input on both frontend (via WASM) and backend (native Rust).
- Hash passwords using Argon2 with random salts before storage.
- Generate 32-byte cryptographically secure verification tokens.
- Store only SHA256 hashes of tokens in the database.
- Send verification emails via SMTP (MailHog in development).
- Verify email tokens and mark users as verified.

- Clean up expired tokens and unverified users automatically.
- Provide translated error messages via the translator crate.

2.3 Security Assumptions

- All traffic is served over HTTPS in production.
- Database is encrypted using SQLCipher with a 32-byte key stored in system keyring or environment variable.
- Email delivery is handled via SMTP (MailHog for development, production SMTP for deployment).
- Verification tokens are sent only to the user's email address.
- Passwords are never logged, stored in plaintext, or returned to the client.
- Frontend validation is for user experience; backend validation is authoritative.
- Expired tokens and unverified users are cleaned up automatically every hour.

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:

- The frontend provides a registration form with real-time validation.
- Username must be 3–20 characters, printable UTF-8 only.
- Email must be valid format.
- Password must be 8–64 characters with uppercase, lowercase, digit, and special character.
- Returns specific error codes for duplicate username/email and validation failures.

3.1.2 FR-2: Email Verification

ID: FR-2

Description: The system must require email verification before the account is active.

Details:

- On successful registration, a secure verification token is generated.
- Token hash is stored in database (plaintext never stored).
- Verification email sent via SMTP with verification link.
- Token has configurable expiry.
- Verification endpoint validates token and marks user as verified.
- Token is deleted after successful verification.

3.1.3 FR-3: Input Validation

ID: FR-3

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

Details:

- Shared validation logic compiled to WASM for frontend use.
- Username: 3–20 characters, printable UTF-8.
- Email: Valid format.
- Password: 8–64 characters with complexity requirements.
- Password strength score calculated with visual indicator.
- Validation errors translated to user-friendly messages.

3.1.4 FR-4: Error Handling

ID: FR-4

Description: The system must provide clear, translated error messages.

Details:

- Error responses use typed error codes.
- Validation errors include field-specific error codes.
- All error codes translated with localization support.
- Server errors logged internally; clients receive generic error.

3.1.5 FR-5: Token Lifecycle

ID: FR-5

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

Details:

- Each user can have only one active verification token.
- Tokens expire after configurable duration.
- Expired tokens are automatically cleaned up.
- Unverified users with expired tokens are deleted.
- Token is deleted upon successful verification.

3.2 Non-Functional Requirements

3.2.1 NFR-1: Security

ID: NFR-1

Description: All security best practices must be followed.

Details:

- Passwords hashed with memory-hard algorithm.
- Verification tokens stored as hashes only.
- Database encrypted at rest.
- HTTPS enforced in production.
- Input validated on both frontend and backend.

3.2.2 NFR-2: Reliability

ID: NFR-2

Description: The module must be fault-tolerant.

Details:

- On email sending failure, user record is rolled back.
- Foreign keys ensure referential integrity.
- Automatic cleanup of expired tokens.

3.2.3 NFR-3: Usability

ID: NFR-3

Description: The module must provide clear user feedback.

Details:

- Real-time field validation on frontend.
- Password strength indicator.
- Translated error messages for all validation failures.
- Clear success/error states on all pages.

4 Component Architecture

4.1 Technology Stack

Frontend

- Vue 3 with TypeScript
- Vuetify for UI components
- Pinia for state management
- `@hey-api/openapi-ts` for automatic TypeScript client generation from OpenAPI spec
- `@hey-api/client-fetch` as the generated client's HTTP layer
- WASM modules for validation (`field-validator`) and translation (`translator`)

Backend

- Rust with Axum web framework
- Tokio async runtime
- `utoipa` for OpenAPI 3.1 specification generation
- `utoipa-axum` for automatic route documentation
- `utoipa-swagger-ui` for interactive API documentation (dev mode)
- SQLx for asynchronous database access
- SQLite with SQLCipher encryption
- Argon2 for password hashing (`argon2` crate)
- SHA256 for token hashing (`sha2` crate)
- `rand` for cryptographically secure random values
- `lettre` for SMTP email sending

Shared Components

- `api-types`: Request/response types with conditional `ToSchema` derivation for OpenAPI
- `field-validator`: Validation logic (compiled to native and WASM)

- `translator`: Error message translation with `rust-i18n`
- Serde for serialization/deserialization
- wasm-pack for WASM module compilation

Database

- SQLite database (`data.db` encrypted, `data_dev.db` unencrypted)
- SQLCipher encryption with 32-byte key
- Key stored in system keyring or `APPSEC_DB_KEY` environment variable
- Foreign keys enabled with CASCADE delete

4.2 OpenAPI and Client Generation

The project uses a type-safe API contract approach where the backend serves as the single source of truth for API definitions.

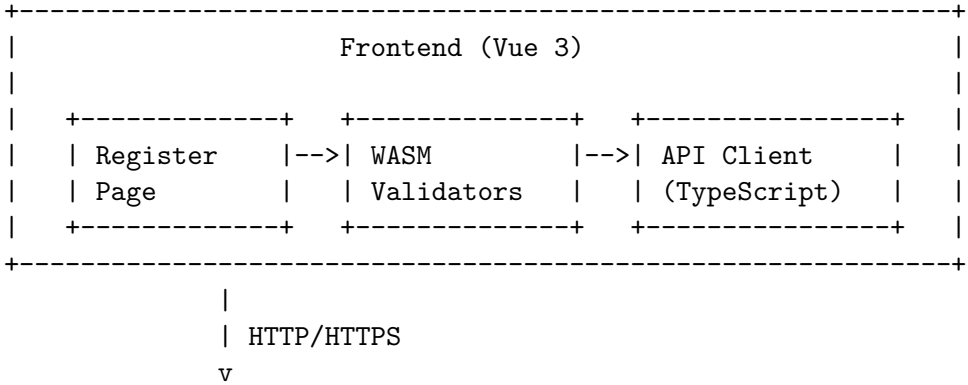
Backend OpenAPI Generation (utoipa)

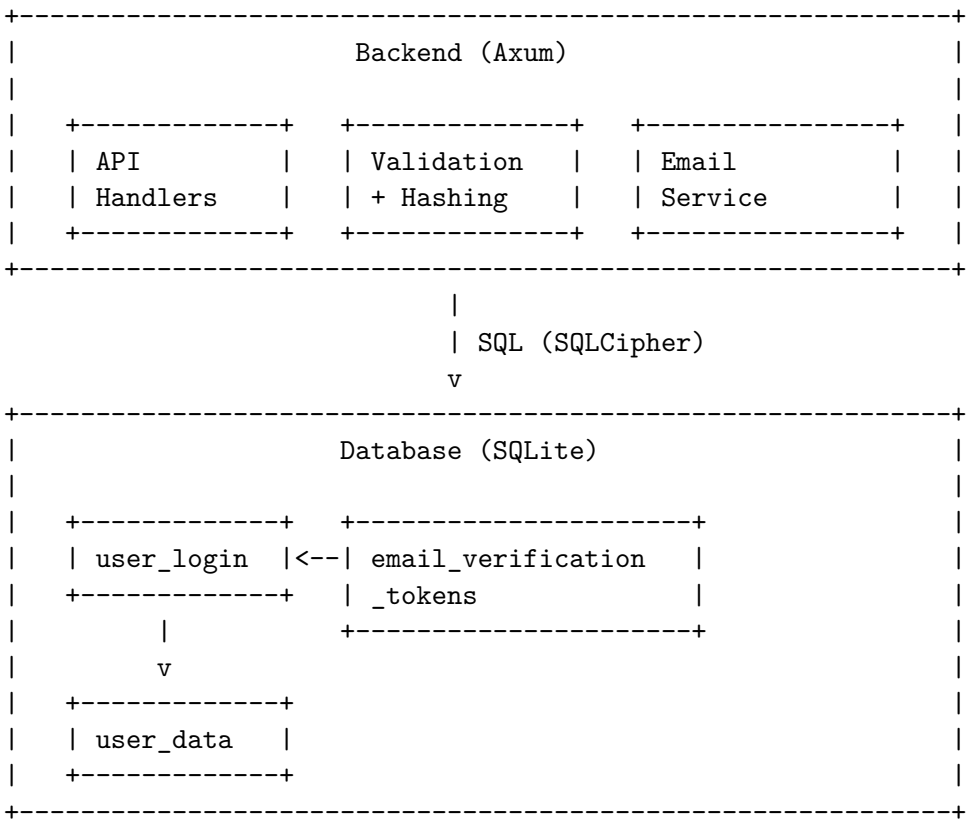
- All API endpoints are documented using `#[utoipa::path()]` proc macro.
- Request/response types derive `ToSchema` via feature flag: `#[cfg_attr(feature = "openapi", derive(ToSchema))]`.
- OpenAPI spec is generated at runtime using `OpenApiRouter::with_openapi()`.
- In development mode (`-dev` flag):
  - OpenAPI JSON available at `/api/openapi.json`
  - Swagger UI available at `/api/docs`
- Endpoints are organized into tags: `health`, `auth`, `counter`.

Benefits of This Approach

- **Single source of truth**: API contract defined once in Rust, consumed everywhere.
- **Type safety**: TypeScript types auto-generated, compile-time API contract validation.
- **No manual synchronization**: Changes to backend API automatically propagate to frontend types.
- **Interactive documentation**: Swagger UI for API exploration and testing.
- **Reduced boilerplate**: No manual API client code or type definitions needed.

4.3 High-Level Architecture Diagram





5 Database Structure

5.1 User Login Table (`user_login`)

Column	Type	Constraints	Notes
`user_id`	INTEGER	PRIMARY KEY AUTOINCREMENT	Unique user identifier
`username`	TEXT	UNIQUE, NOT NULL	Indexed for lookup
`email`	TEXT	UNIQUE, NOT NULL	Stored normalized (lowercase)
`password`	TEXT	NOT NULL	Argon2 hash in PHC format
`email_verified`	INTEGER	NOT NULL, DEFAULT 0	Boolean: 0=false, 1=true
`email_verified_at`	INTEGER	NULLABLE	Unix timestamp when verified

Constraints and notes:

- Unique constraints on `username` and `email` prevent duplicates.
- `email_verified` must be true before user can log in.

5.2 User Data Table (`user_data`)

Column	Type	Constraints	Notes
`user_id`	INTEGER	PRIMARY KEY, FOREIGN KEY → `user_login.user_id`	CASCADE on delete
`counter`	INTEGER	NOT NULL, DEFAULT 0	Application-specific data

Notes:

- One-to-one relationship with `user_login`.
- Created automatically during registration.
- Cascading delete ensures cleanup when user is removed.

5.3 Email Verification Tokens Table (`email_verification_tokens`)

Column	Type	Constraints	Notes
`user_id`	INTEGER	PRIMARY KEY, FOREIGN KEY → `user_login.user_id`	CASCADE on delete
`token_hash`	TEXT	NOT NULL	SHA256 hash of verification token
`expires_at`	INTEGER	NOT NULL	Unix timestamp for expiry
`created_at`	INTEGER	NOT NULL	Unix timestamp when created

Constraints and guarantees:

- Primary key on `user_id` ensures one token per user.
- Only SHA256 hash is stored; plaintext token is sent via email only.
- Token is valid only if `expires_at > current_time`.
- Token is deleted after successful verification.
- Cascading delete removes token when user is deleted.
- Hourly cleanup task deletes expired tokens and associated unverified users.

5.4 Database Configuration

- **Engine**: SQLite with SQLCipher encryption
- **Development**: Unencrypted `data_dev.db`
- **Production**: Encrypted `data.db`
- **Encryption Key**: 32-byte key (64 hex characters)
  - Priority 1: `APPSEC_DB_KEY` environment variable
  - Priority 2: System keyring (service: `APPSEC_DB_KEY`, user: `APPSEC`)
  - Auto-generated if not found
- **Foreign Keys**: Enabled with `PRAGMA foreign_keys = ON`
- **Cleanup**: Hourly task removes expired tokens and unverified users

6 UML Sequence Diagrams

6.1 Registration Flow Sequence Diagram

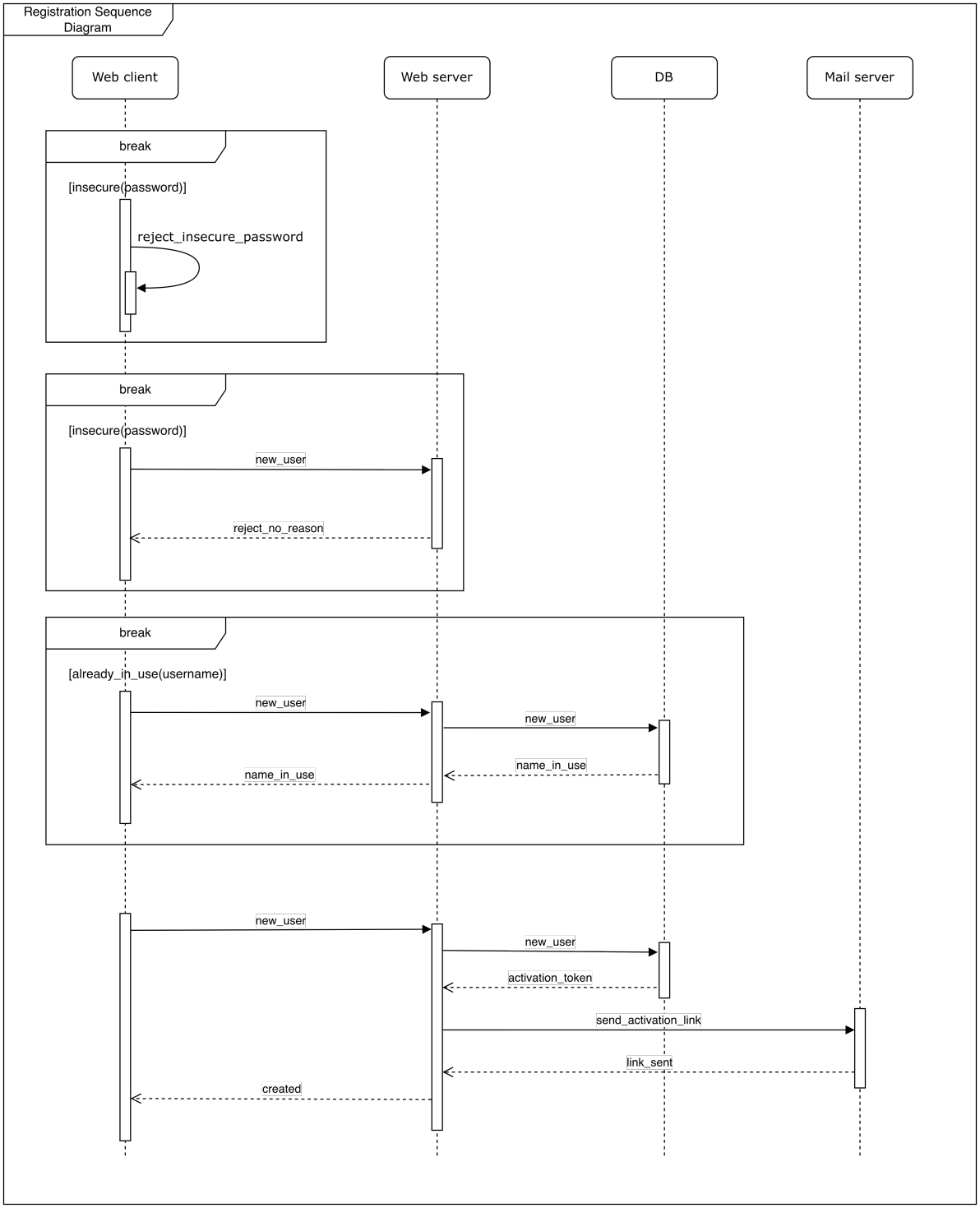


Figure 1: Registration Flow Sequence Diagram

6.2 Verification / Activation Flow Sequence Diagram

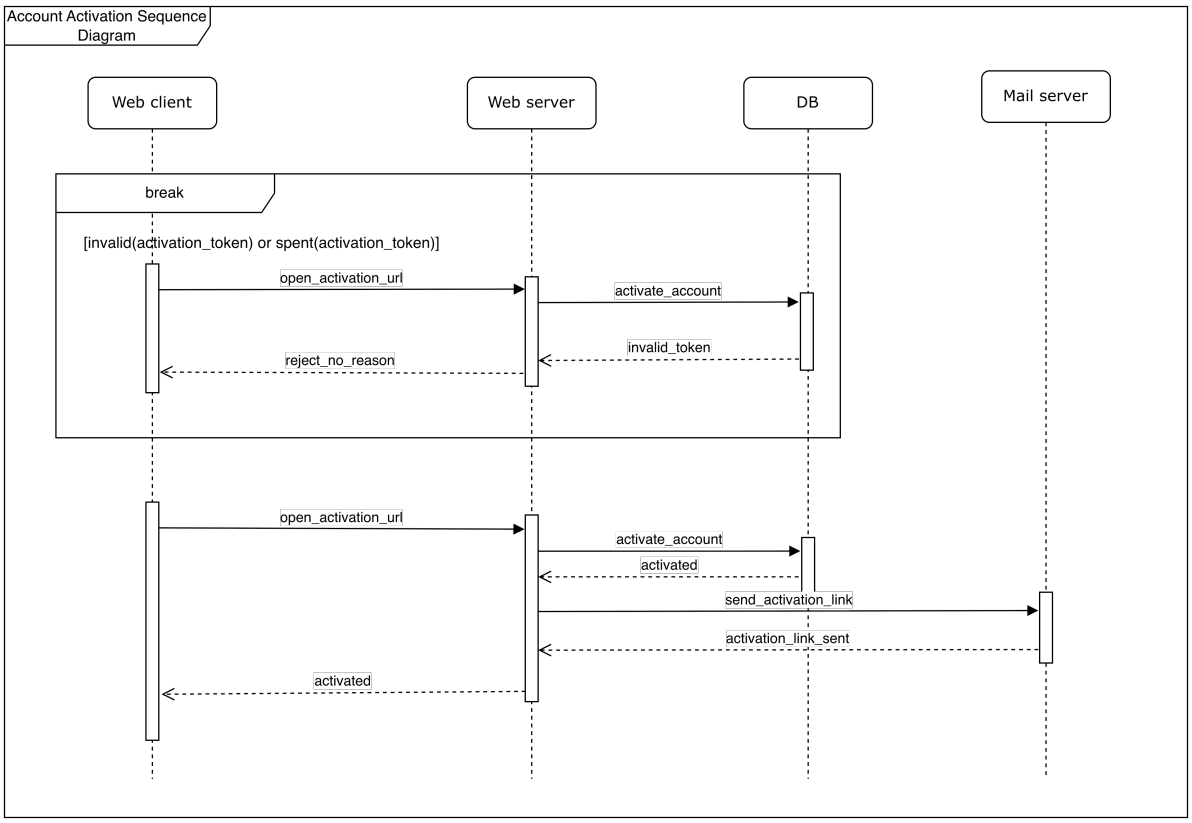


Figure 2: Verification Flow Sequence Diagram

7 Security Mechanisms

This section describes the security measures implemented in the registration module.

7.1 Password Security

- Argon2 algorithm (memory-hard, GPU-resistant).
- Random salt generated using `SaltString::generate(&mut OsRng)`.
- Password hash stored in PHC string format (includes algorithm, parameters, salt).
- Passwords are never logged, stored in plaintext, or returned to clients.

7.2 Token Security

- Verification tokens are 32 bytes of cryptographically secure random data.
- Tokens are hex-encoded for transmission (64 characters).
- Only SHA256 hashes of tokens are stored in the database.
- Plaintext tokens are sent only via email and never logged.
- Tokens have configurable expiry (`EMAIL_VERIFICATION_TOKEN_DURATION_HOURS`).
- One token per user (primary key constraint).

7.3 Error Handling

- Typed error codes from `api-types` crate.
- Specific codes for `USERNAME_TAKEN` and `EMAIL_TAKEN` (HTTP 409).
- Validation errors include field-specific details for user feedback.
- Internal errors return generic `INTERNAL` code (HTTP 500).
- All error codes translated to user-friendly messages via `translator` crate.

7.4 Automatic Cleanup

- Hourly cleanup task runs asynchronously.
- Deletes expired verification tokens.
- Deletes unverified users whose tokens have expired (via CASCADE).
- On email sending failure, user record is rolled back immediately.

7.5 Future Enhancements

- Rate limiting on registration endpoint.
- CAPTCHA integration for bot mitigation.
- Email domain whitelist/blacklist.
- Invite token system for controlled registration.

8 API Reference

This section documents the registration and email verification endpoints. All endpoints are documented using `utoipa` annotations and the OpenAPI specification is automatically generated. Types are defined in the `api-types` crate with conditional `ToSchema` derivation for OpenAPI support.

8.1 Endpoints Overview

- `POST /api/register` – Create a new user account and send verification email.
- `POST /api/verify-email` – Verify email address using token from email.
- `GET /api/health` – Health check endpoint.

In development mode, interactive documentation is available:

- `GET /api/openapi.json` – OpenAPI 3.1 specification (JSON)
- `GET /api/docs` – Swagger UI for interactive API exploration

8.2 `POST /api/register`

Registers a new user account and sends a verification email.

Request Body (JSON)

```
{
  "username": "string", // 3-20 chars, printable UTF-8
  "email": "string", // valid email format
  "password": "string" // 8-64 chars, complexity requirements
}
```

Success Response (HTTP 200)

Empty response body on success.

Error Responses

HTTP 400 Bad Request – Validation failed:

```
{
  "error": "VALIDATION",
  "validation": {
    "fieldErrors": [
      {
        "field": "USERNAME" | "EMAIL" | "PASSWORD",
        "errors": ["TOO_SHORT", "TOO_LONG", "INVALID_FORMAT", ...]
      }
    ]
  }
}
```

HTTP 409 Conflict – Username or email taken:

```
{
  "error": "USERNAME_TAKEN" | "EMAIL_TAKEN"
}
```

HTTP 500 Internal Server Error:

```
{
  "error": "INTERNAL"
}
```

Validation Error Codes

- `REQUIRED` – Field is empty
- `TOO_SHORT` – Below minimum length
- `TOO_LONG` – Exceeds maximum length
- `INVALID_CHARACTERS` – Contains invalid characters
- `INVALID_FORMAT` – Invalid format (email)
- `TOO_FEW_UPPERCASE_LETTERS` – Password missing uppercase
- `TOO_FEW_LOWERCASE_LETTERS` – Password missing lowercase
- `TOO_FEW_DIGITS` – Password missing digit
- `TOO_FEW_SPECIAL_CHARACTERS` – Password missing special char

8.3 `POST /api/verify-email`

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

Request Body (JSON)

```
{
  "token": "string" // 64-char hex token from email link
}
```

Success Response (HTTP 200)

Empty response body on success. User is now verified and can log in.

Error Responses

HTTP 400 Bad Request – Token expired or invalid:

```
{
  "error": "TOKEN_EXPIRED"
}
```

HTTP 500 Internal Server Error:

```
{
  "error": "INTERNAL"
}
```

Behavior

- Computes SHA256 hash of provided token.
- Looks up hash in `email_verification_tokens` table.
- Checks token has not expired (`expires_at > now`).
- Sets `email_verified = true` on user.
- Sets `email_verified_at` to current timestamp.
- Deletes token from database.
- Idempotent: returns success if user already verified.

8.4 `GET /api/health`

Simple health check endpoint.

Response (HTTP 200)

Returns basic health status indicating backend is running.

8.5 Type Definitions

Types are defined in `api-types/src/` with OpenAPI schema support:

- `requests.rs`: `RegistrationRequest`, `EmailVerificationRequest`
- `responses.rs`: `RegisterError`, `RegisterErrorResponse`, `VerifyEmailError`
- `validation.rs`: `ValidationFieldError`, `ValidationErrorData`
- `enums.rs`: `FieldType`, `ValidationErrorCode`, `PasswordStrength`

All types use conditional compilation for OpenAPI support:

```
#[derive(Debug, Clone, Serialize, Deserialize)]
#[cfg_attr(feature = "openapi", derive(ToSchema))]
#[serde(rename_all = "camelCase")]
pub struct RegistrationRequest {
    pub username: String,
    pub email: String,
    pub password: String,
}
```

The `openapi` feature is enabled in the backend but disabled for WASM builds, keeping the WASM binary size minimal while providing full OpenAPI documentation in the backend.

9 Implementation Details

This section provides implementation details including code examples and the client generation process.

9.1 User Interface

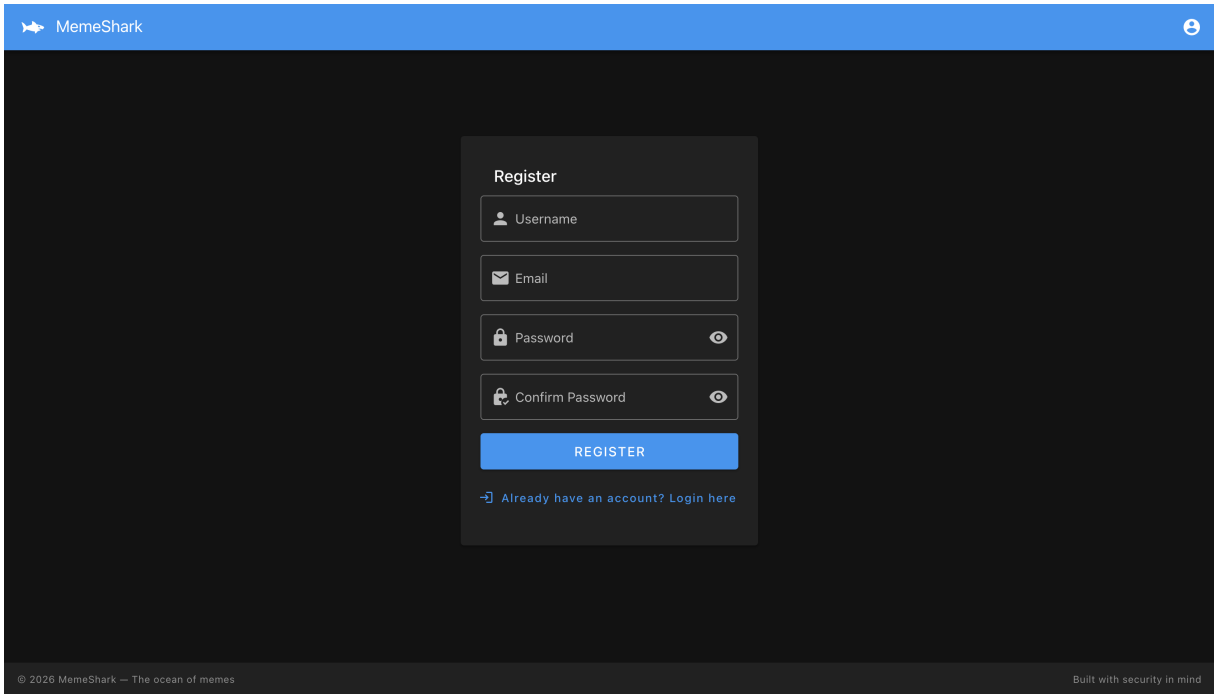


Figure 3: Registration form with real-time field validation. Shows username, email, and password fields with immediate feedback on input validity using WASM-based validation.

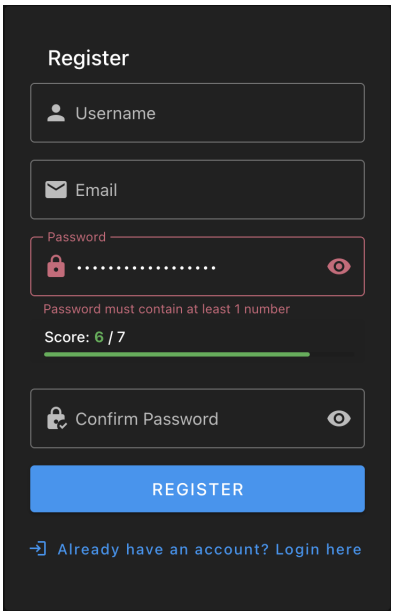


Figure 4: Password strength indicator showing the 0–7 score system. Visual feedback updates in real-time as the user types, displaying strength level (weak/medium/strong/cia) and specific requirements not yet met.

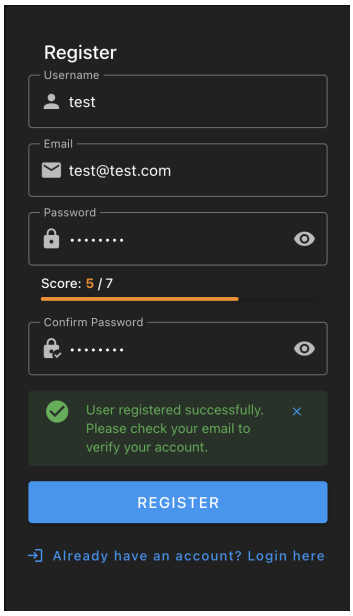


Figure 5: Successful registration confirmation message. Informs the user that a verification email has been sent and provides next steps.

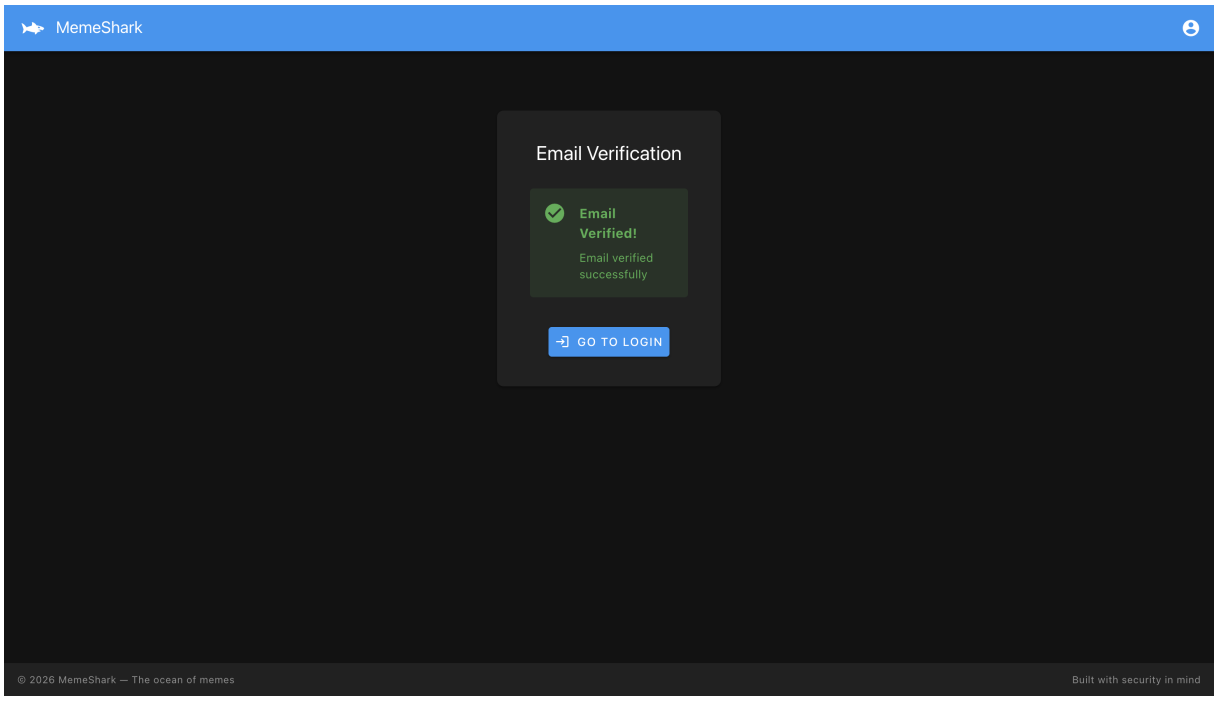


Figure 6: Email verification page showing successful account activation. Displayed after the user clicks the verification link from their email.

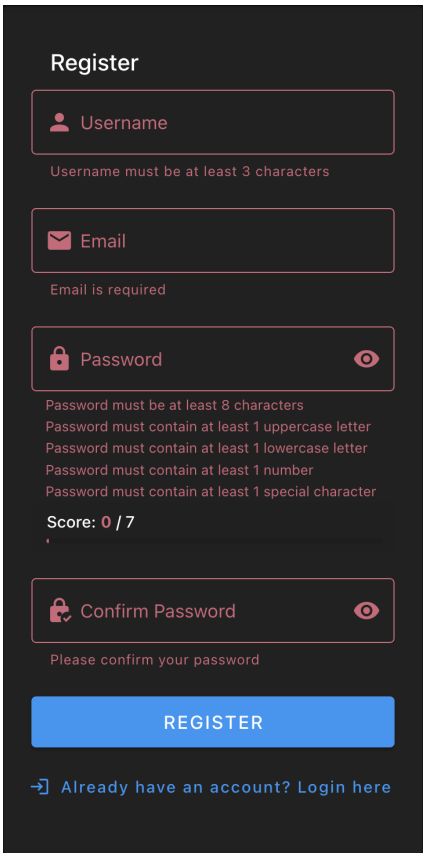


Figure 7: Form displaying validation errors with translated messages. Shows field-specific error messages generated by the WASM translator module.

9.2 Frontend Client Generation

The build script (`build.sh`) fetches the OpenAPI spec from the running backend:

```
curl -s http://localhost:4000/api/openapi.json \
  > frontend/src/generated/openapi.json
```

TypeScript client is generated using `@hey-api/openapi-ts`:

```
npx @hey-api/openapi-ts \
  --input src/generated/openapi.json \
  --output src/generated/api-client \
  --client @hey-api/client-fetch
```

Generated files in `frontend/src/generated/api-client/`:

- `types.gen.ts`: All TypeScript interfaces (`RegistrationRequest`, `LoginResponse`, etc.)
- `sdk.gen.ts`: Typed API functions (`registerUser()`, `verifyEmail()`, etc.)
- `index.ts`: Re-exports for convenient imports

Client configured in `frontend/src/api/client.ts` with base URL and credentials.

9.3 Registration Flow Implementation

Frontend registration call (`frontend/src/pages/register.vue`):

```
import { registerUser } from '@/generated/api-client';

const { data, error, response } = await registerUser({
  body: {
    username: formData.username,
    email: formData.email,
    password: formData.password,
  }
});

if (response.ok) {
  // Show translated success message
  statusMessage.value = translate('SUCCESS_REGISTERED', undefined);
} else if (error) {
  // Handle specific error codes
  if (error.error === 'USERNAME_TAKEN') {
    statusMessage.value = translate_error_code('USERNAME_TAKEN', undefined);
  } else if (error.error === 'VALIDATION') {
    // Display field-specific errors
    error.validation?.fieldErrors.forEach(fieldError => {
      fieldError.errors.forEach(code => {
        const msg = translate_field_validation_error(
          fieldError.field, code, undefined
        );
      });
    });
  }
}
```

9.4 Email Verification Implementation

Frontend verification (`frontend/src/pages/verify-email.vue`):

```
import { verifyEmail } from '@/generated/api-client';

// Extract token from URL query parameter
const route = useRoute();
const token = route.query.token as string;

if (!token) {
  // Show warning: no token provided
  return;
}

const { data, error, response } = await verifyEmail({
  body: { token }
});

if (response.ok) {
  // Show success, navigate to login
  statusMessage.value = translate('SUCCESS_EMAIL_VERIFIED', undefined);
} else {
  // Show error (TOKEN_EXPIRED or INTERNAL)
  statusMessage.value = translate_error_code(error.error, undefined);
}
```

9.5 Password Strength Validation

Frontend password field uses detailed validation (`frontend/src/components/auth/PasswordField.vue`):

```
import { validate_password_detailed } from '@/wasm/field-validator';

const result = JSON.parse(validate_password_detailed(password));

// result.score: 0-7 (based on length and character types)
// result.strength: "weak" | "medium" | "strong" | "cia"
// result.errors: ["TOO_SHORT", "TOO_FEW_UPPERCASE_LETTERS", ...]

// Score calculation:
// +1 for length >= 8
// +1 for length >= 12
// +1 for length >= 16
// +1 for uppercase letter
// +1 for lowercase letter
// +1 for digit
// +1 for special character
```

9.6 Implementation Files

9.6.1 Backend Files

- `backend/src/main.rs`
  - Application entry point.
  - Configures Axum router with API routes using `utoipa-axum`.
  - Builds OpenAPI spec at runtime with `OpenApiRouter::with_openapi()`.
  - Mounts Swagger UI at `/api/docs` in dev mode.
  - Initializes database connection and starts cleanup task.
- `backend/src/api/register.rs`
  - `POST /api/register` handler with `#[utoipa::path()]` annotation.
  - Validates input, checks uniqueness, creates user.
  - Generates verification token and sends email.
- `backend/src/api/verify_email.rs`
  - `POST /api/verify-email` handler with `#[utoipa::path()]` annotation.
  - Verifies token and marks user as verified.
- `backend/src/api/utils.rs`
  - Configuration constants from `build.rs`.
  - Token duration settings, base URLs.
- `backend/src/db/mod.rs`
  - `DBHandle` struct with connection pool.
  - Password hashing with Argon2.
  - Cleanup task for expired tokens.
- `backend/src/db/user_login.rs`
  - User table operations: create, lookup, verify.
- `backend/src/db/user_data.rs`
  - User data table operations.
- `backend/src/db/email_verification_tokens.rs`
  - Token table operations: create, lookup, delete.
- `backend/src/email.rs`
  - SMTP email sending via `lettre`.

  - Verification email template.

9.6.2 Frontend Files

- `frontend/src/pages/register.vue`
  - Registration page with form.
  - Uses generated `registerUser()` from API client.
  - Real-time validation and error display.
- `frontend/src/pages/verify-email.vue`
  - Email verification page.
  - Uses generated `verifyEmail()` from API client.
  - Extracts token from URL and calls API.
- `frontend/src/components/auth/UsernameField.vue`
  - Username input with WASM validation.
- `frontend/src/components/auth/EmailField.vue`
  - Email input with WASM validation.
- `frontend/src/components/auth/PasswordField.vue`
  - Password input with strength indicator (0–7 score).
  - WASM validation with detailed errors.
- `frontend/src/components/auth/ConfirmPasswordField.vue`
  - Password confirmation with match validation.
- `frontend/src/components/auth/AuthFormLayout.vue`
  - Common layout for auth forms.
- `frontend/src/components/auth/StatusMessage.vue`
  - Success/error message display.
- `frontend/src/api/client.ts`
  - API client configuration (base URL, credentials).
  - Re-exports generated API client for convenient imports.

9.6.3 Generated Files (Auto-Generated)

- `frontend/src/generated/openapi.json`
  - OpenAPI 3.1 specification fetched from backend.
  - Source of truth for client generation.
- `frontend/src/generated/api-client/types.gen.ts`
  - Auto-generated TypeScript interfaces from OpenAPI spec.
  - Includes `RegistrationRequest`, `RegisterErrorResponse`, etc.
- `frontend/src/generated/api-client/sdk.gen.ts`
  - Auto-generated typed API functions.
  - Includes `registerUser()`, `verifyEmail()`, `healthCheck()`.
- `frontend/src/generated/api-client/index.ts`
  - Re-exports all generated types and functions.

9.6.4 Shared Crates

- `api-types/src/`
  - `requests.rs`: Request types with `ToSchema` derivation.
  - `responses.rs`: Response and error types with `ToSchema` derivation.
  - `validation.rs`: Validation data structures.
  - `enums.rs`: `FieldType`, `ValidationErrorCode`, `PasswordStrength`.

  - `Cargo.toml`: `openapi` feature flag for conditional `utoipa` support.
- `field-validator/src/lib.rs`
  - `validate_username()`, `validate_email()`, `validate_password()`
  - `validate_field()` – WASM-exported wrapper
  - `validate_password_detailed()` – Returns strength score
- `translator/src/lib.rs`
  - `translate()`, `translate_error_code()`
  - `translate_field_validation_error()`
- `translator/locales/en.yml`
  - English translations for all error codes and messages.

9.6.5 Build and Configuration Files

- `build.sh` – Main build script (WASM, backend, OpenAPI, frontend)
- `dev.sh` – Development mode launcher
- `Dockerfile` – Container build configuration
- `backend/.env` – Environment variables (token durations, base URLs)
- `backend/build.rs` – Compile-time constant generation
- `Cargo.toml` – Workspace with utoipa dependencies
- `frontend/package.json` – NPM scripts including `generate:api`

10 Building and Running

10.1 Build Process

The project uses `build.sh` to orchestrate the entire build process:

```
# Full build (WASM, backend, OpenAPI client, frontend)
./build.sh

# Development mode with hot reload
./dev.sh
```

The build script handles:

- WASM module compilation (`field-validator`, `translator`)
- Backend build with OpenAPI generation
- TypeScript client generation from OpenAPI spec
- Frontend build

10.2 Development Mode

In development mode:

- Swagger UI available at `http://localhost:4000/api/docs`
- Emails sent to MailHog at `http://localhost:8025`
- Database is unencrypted (`data_dev.db`)

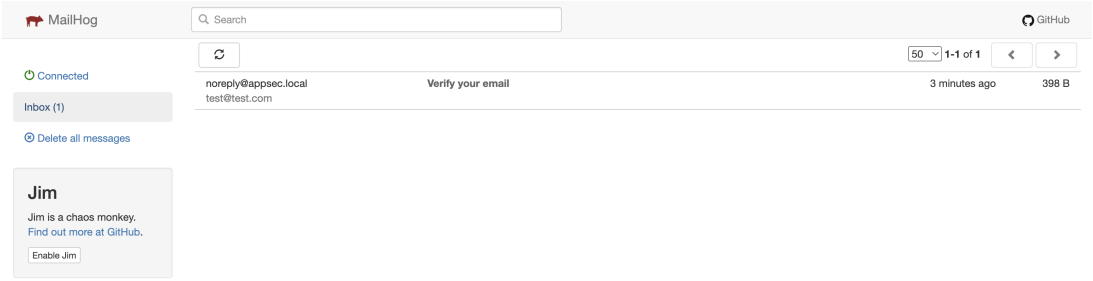


Figure 8: Swagger UI showing the API documentation interface available at `/api/docs` in development mode. Provides interactive testing of all API endpoints.

Figure 9: MailHog email testing interface showing captured verification emails. Allows developers to inspect email content and click verification links without sending real emails.

10.3 Production Mode

In production mode:

- Database encrypted with SQLCipher (`data.db`)
- OpenAPI endpoints disabled
- HTTPS required for all traffic

11 Version History

Version	Date	Changes
1.0	2025-12-23	Initial registration module documentation
1.1	2026-01-11	Aligned documentation with implementation (draft)
1.2	2026-01-11	Final report with full implementation details

Document Generated: 2025-12-23
Last Updated: 2026-01-11
Status: Final
Review Status: Complete