Login Module Documentation (Final)

Kamil Grzymkowski (151908)

2026-01-20

Contents

1 Kamil Grzymkowski (151908), Application Security

- **Course**: Application Security
- **Level**: Graduate/Master's
- **Focus Areas**:
  - Secure user registration and email verification
  - User login with cookie-based session management
  - Secure password reset workflow
  - Password security with Argon2 hashing
  - Cryptographic token generation and storage
  - Input validation (frontend and backend)
  - Cross-platform validation via WebAssembly

1.1 Document Version Notice

Version 2.1 (Final): This document includes the complete authentication module with registration, login, session management, and password reset functionality. This is the final version with implementation screenshots and conclusions.

1.2 Exercise Description

This exercise demonstrates a complete secure authentication module implementing modern web security practices. The module provides:

- **User Registration**: Secure registration with client-side and server-side validation, password strength scoring (0–7 scale), and email verification workflow
- **User Login**: Authentication with username/password, cookie-based session tokens, and email verification enforcement
- **Session Management**: Secure HTTP-only cookies, configurable session duration, automatic session refresh, and server-side session storage
- **Password Reset**: Secure token-based password reset via email with automatic token expiry
- **Security Features**: Argon2 password hashing, SHA256 token hashing, defense against enumeration attacks, and automatic cleanup of expired data
- **Shared Validation**: Cross-platform validation logic compiled to WebAssembly for frontend and native Rust for backend

2 Component Description

2.1 Purpose

The authentication module provides a complete secure mechanism for user registration, login, session management, and password reset. It ensures secure access to the application while protecting user credentials and session data.

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 all tokens (verification, session, password reset) as SHA256 hashes, never in plaintext.

5. Enforce email verification before the account can be used.
6. Manage user sessions securely using HTTP-only cookies with server-side session storage.
7. Provide secure password reset workflow via email tokens.
8. Automatically clean up expired tokens, sessions, and unverified accounts.

2.2 Responsibilities

2.2.1 Registration

- 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.

2.2.2 Login and Session Management

- Authenticate users with username and password.
- Verify email is confirmed before allowing login.
- Generate 32-byte cryptographically secure session tokens.
- Store session token hashes in the database with expiry timestamps.
- Set HTTP-only, Secure (in production), SameSite=Strict cookies.
- Provide session validation endpoint for frontend auth state.
- Support session refresh to extend session lifetime.
- Invalidate sessions on logout by removing from database and clearing cookie.

2.2.3 Password Reset

- Accept password reset requests by email address.
- Always return success response to prevent email enumeration.
- Generate secure reset tokens and store hashes in database.
- Set password reset flag on user account.
- Send password reset email with secure link.
- Validate reset tokens and update password on completion.
- Clear reset token and flag after successful password change.

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 and reset tokens are sent only to the user's email address.
- Passwords are never logged, stored in plaintext, or returned to the client.
- Session tokens are stored in HTTP-only cookies, inaccessible to JavaScript.
- Frontend validation is for user experience; backend validation is authoritative.
- Expired tokens, sessions, and unverified users are cleaned up automatically every hour.

- Password reset always returns success to prevent email enumeration attacks.

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.

3.1.6 FR-6: User Login

ID: FR-6

Description: The system must allow verified users to log in with username and password.

Details:

- Login form accepts username and password.
- System validates username format before database lookup.
- Password is verified against Argon2 hash in database.
- Login is rejected if email is not verified.
- On success, a session token is generated and stored as HTTP-only cookie.
- Login response includes session expiry and user info.

3.1.7 FR-7: Session Management

ID: FR-7

Description: The system must manage user sessions securely.

Details:

- Sessions are stored server-side with token hash as primary key.
- Session tokens are 32 bytes of cryptographically secure random data.
- Cookies are HTTP-only, Secure (in production), SameSite=Strict.
- Session duration is configurable (default: 7 days).
- Frontend can check session validity via `GET /api/auth/check`.
- Frontend can refresh session via `POST /api/auth/refresh`.
- Expired sessions are automatically cleaned up hourly.

3.1.8 FR-8: User Logout

ID: FR-8

Description: The system must allow users to log out and destroy their session.

Details:

- Logout endpoint deletes session from database.
- Session cookie is cleared by setting expired cookie.
- Logout always returns success (idempotent).

3.1.9 FR-9: Password Reset Request

ID: FR-9

Description: The system must allow users to request a password reset via email.

Details:

- User provides email address to request password reset.
- System always returns success to prevent email enumeration.
- If email exists, a reset token is generated and emailed.
- Reset token hash is stored in database.
- Password reset flag is set on user account.
- Reset tokens have configurable expiry.

3.1.10 FR-10: Password Reset Completion

ID: FR-10

Description: The system must allow users to set a new password using a valid reset token.

Details:

- User provides reset token and new password.
- System validates token exists and is not expired.
- New password is validated against password policy.
- Password is hashed and stored.
- Reset token is deleted after successful reset.
- Password reset flag is cleared on user account.

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 Argon2 algorithm.
- All tokens (verification, session, reset) stored as SHA256 hashes only.
- Database encrypted at rest with SQLCipher.
- HTTPS enforced in production.
- Session cookies are HTTP-only and Secure.
- Input validated on both frontend and backend.
- Password reset prevents email enumeration.

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 and sessions.
- Session refresh extends validity without requiring re-login.

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.
- Automatic session refresh in background.

3.2.4 NFR-4: Performance

ID: NFR-4

Description: The module must perform efficiently.

Details:

- Session lookup by token hash is O(1) via primary key.
- Cleanup tasks run hourly to prevent table bloat.
- WASM validation runs client-side to reduce server load.
- Session refresh uses existing token (no new token generation).

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

```
+-------------------------------------------------------------+
|                     Frontend (Vue 3)                        |
|                                                             |
|   +-------------+   +--------------+   +----------------+   |
|   | Register    |-->| WASM         |-->| API Client     |   |
|   | Page        |   | Validators   |   | (TypeScript)   |   |
|   +-------------+   +--------------+   +----------------+   |
+-------------------------------------------------------------+
              |
              | HTTP/HTTPS
              v
+-------------------------------------------------------------+
|                      Backend (Axum)                         |
|                                                             |
|   +-------------+   +--------------+   +----------------+   |
|   | API         |   | Validation   |   | Email          |   |
|   | Handlers    |   | + Hashing    |   | Service        |   |
|   +-------------+   +--------------+   +----------------+   |
+-------------------------------------------------------------+
                             |
                             | SQL (SQLCipher)
                             v
+-------------------------------------------------------------+
|                     Database (SQLite)                       |
|                                                             |
|   +-------------+   +----------------------+                |
|   | user_login  |<--| email_verification   |                |
|   +-------------+   | _tokens              |                |
|         |           +----------------------+                |
|         v                                                   |
|   +-------------+                                           |
|   | user_data   |                                           |
|   +-------------+                                           |
+-------------------------------------------------------------+
```

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	NULLABLE	Argon2 hash in PHC format, NULL during reset
`email_verified`	INTEGER	NOT NULL, DEFAULT 0	Boolean: 0=false, 1=true
`email_verified_at`	INTEGER	NULLABLE	Unix timestamp when verified
`password_reset`	INTEGER	NOT NULL, DEFAULT 0	Boolean: password reset in progress
`role`	INTEGER	NOT NULL, DEFAULT 0	0=User, 1=Admin
`deleted_at`	INTEGER	NULLABLE	Unix timestamp when soft-deleted

Constraints and notes:

- Unique constraints on `username` and `email` prevent duplicates.
- `email_verified` must be true before user can log in.
- `password` can be NULL when a password reset is in progress.
- `password_reset` flag indicates a reset token has been issued.
- `role` determines access level (0=User, 1=Admin).
- `deleted_at` enables soft delete; non-NULL means user is deleted.
- Authentication queries filter by `deleted_at IS NULL` to exclude deleted users.

5.2 User Sessions Table (`user_sessions`)

Column	Type	Constraints	Notes
`user_id`	INTEGER	FOREIGN KEY → `user_login.user_id`	CASCADE on delete
`session_id`	TEXT	NOT NULL	Random identifier for logging
`session_hash`	TEXT	PRIMARY KEY, NOT NULL	SHA256 hash of session token
`session_expiry`	INTEGER	NOT NULL	Unix timestamp for session expiry
`session_created_at`	INTEGER	NOT NULL	Unix timestamp when created

Constraints and notes:

- Primary key on `session_hash` enables O(1) lookup by token.
- Users can have multiple active sessions (multi-device support).
- Sessions are validated by checking `session_expiry > now`.
- Cascading delete removes sessions when user is deleted.
- Expired sessions are cleaned up hourly.

5.3 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.

5.4 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.

5.5 Password Reset Tokens Table (`password_reset_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 reset 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 reset token per user.
- New reset request overwrites existing token (UPSERT).
- 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 password reset.
- Cascading delete removes token when user is deleted.
- Hourly cleanup task deletes expired tokens.

5.6 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 sessions

6 UML Sequence Diagrams

6.1 Registration Flow Sequence Diagram

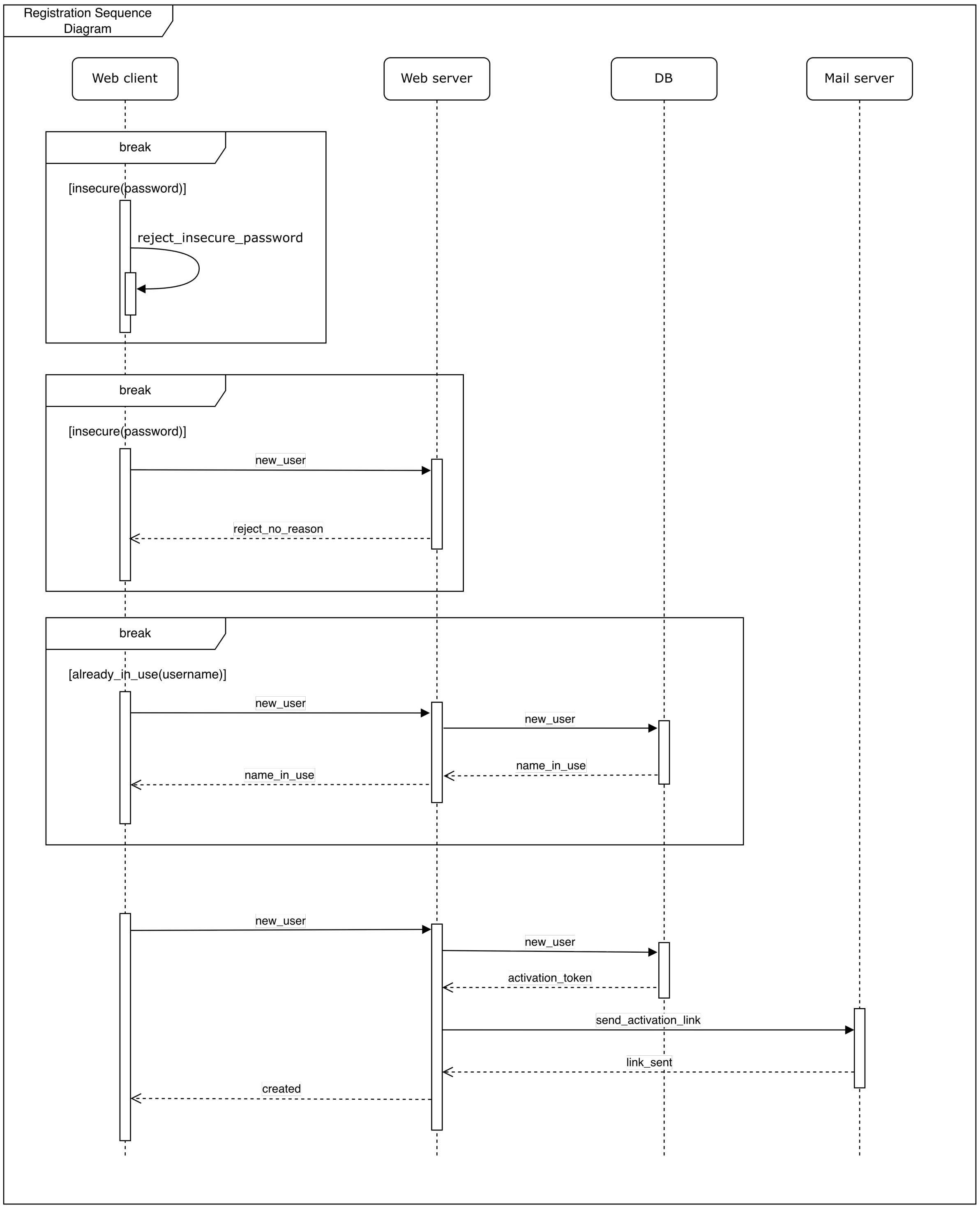


Figure 1: Registration Flow Sequence Diagram

6.2 Verification / Activation Flow Sequence Diagram

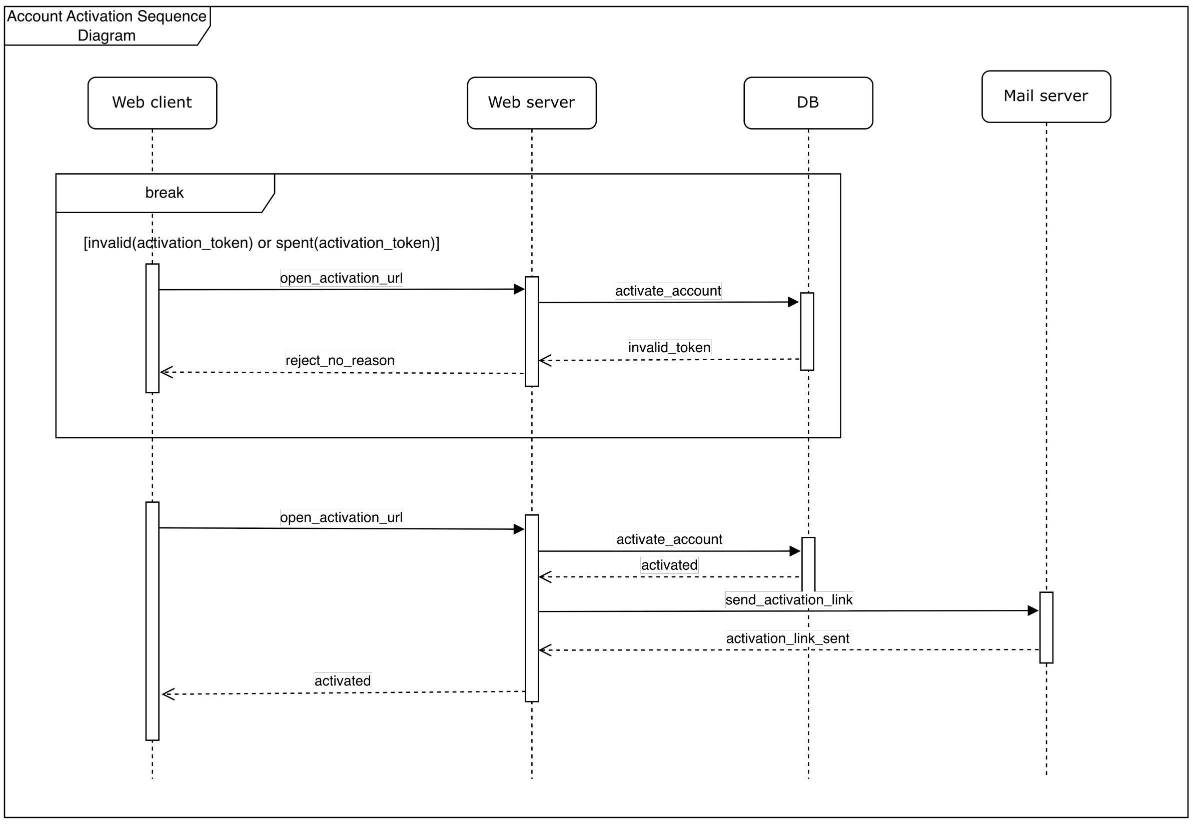


Figure 2: Verification Flow Sequence Diagram

6.3 Login Flow Sequence Diagram

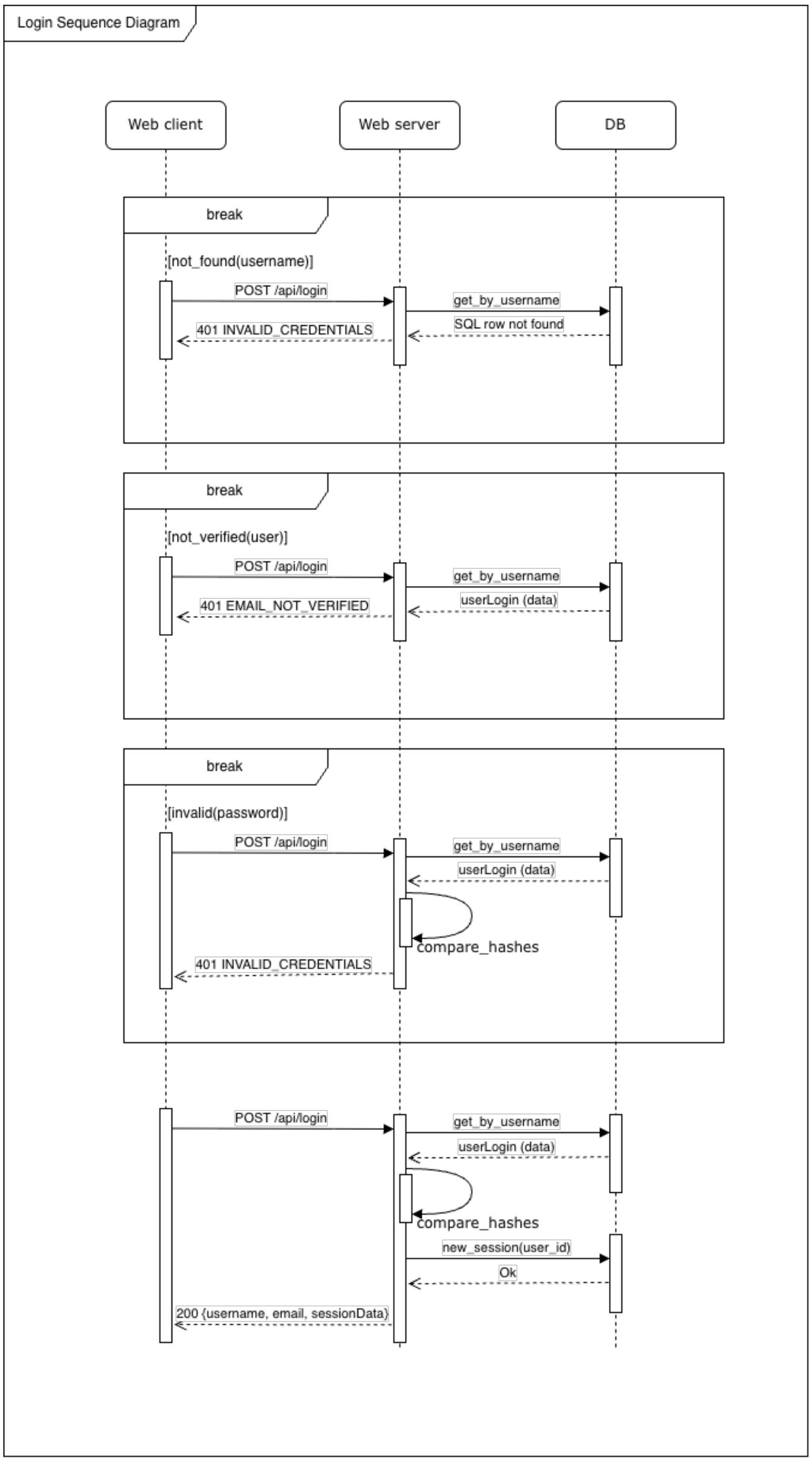


Figure 3: Login Flow Sequence Diagram showing authentication, session creation, and cookie setting.

7 Security Mechanisms

This section describes the security measures implemented in the authentication 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.
- Password complexity requirements: 8–64 characters, uppercase, lowercase, digit, special character.
- Password strength scoring (0–7) provides user feedback.

7.2 Token Security

7.2.1 Email Verification Tokens

- 32 bytes of cryptographically secure random data (`OsRng`).
- Hex-encoded for transmission (64 characters).
- Only SHA256 hashes stored in database.
- Plaintext tokens sent only via email, never logged.
- Configurable expiry (`EMAIL_VERIFICATION_TOKEN_DURATION_HOURS`).
- One token per user (primary key constraint).
- Deleted after successful verification.

7.2.2 Session Tokens

- 32 bytes of cryptographically secure random data.
- Stored in HTTP-only cookies (inaccessible to JavaScript).
- Only SHA256 hashes stored in database.
- Cookies set with `Secure` flag in production (HTTPS only).
- `SameSite=Strict` prevents CSRF attacks by blocking cross-site cookie transmission.
- Configurable duration (`SESSION_DURATION_DAYS`, default 7).
- Users can have multiple active sessions (multi-device).
- Session refresh extends expiry without generating new token.

7.2.3 Password Reset Tokens

- 32 bytes of cryptographically secure random data.
- Only SHA256 hashes stored in database.
- Configurable expiry (`PASSWORD_RESET_TOKEN_DURATION_HOURS`).
- One token per user (new request overwrites existing via UPSERT).
- Deleted after successful password reset.
- Password reset flag tracks active reset process.

7.3 Session Management Security

- Server-side session storage (no sensitive data in cookie).

- Session lookup by token hash is O(1) via primary key.
- Cookie attributes:
  - `HttpOnly`: Prevents XSS attacks from accessing token.
  - `Secure`: Cookie only sent over HTTPS (production).
  - `SameSite=Strict`: Prevents CSRF by blocking cross-site requests.
  - `Path=/`: Cookie sent with all requests to origin.
- Logout invalidates session server-side and clears cookie.
- Expired sessions cannot be used (server validates expiry).

7.4 Transport Layer Security (TLS)

The application supports HTTPS with TLS 1.3 for secure communication:

- TLS termination handled by `axum-server` with `rustls` backend.
- Uses the `ring` cryptographic provider for TLS operations.
- Certificate and key paths configurable via `config.toml`:
  - `tls.enabled`: Toggle TLS on/off.
  - `tls.cert_path`: Path to PEM certificate file.
  - `tls.key_path`: Path to PEM private key file.
- Self-signed certificates can be generated for development using the included `generate-certs.sh` script.
- Production deployments should use certificates from a trusted CA (e.g., Let's Encrypt).

7.5 HTTP Strict Transport Security (HSTS)

HSTS ensures browsers always use HTTPS for the application:

- Implemented as Axum middleware that adds the `Strict-Transport-Security` header.
- Configurable via `config.toml`:
  - `security.hsts_enabled`: Toggle HSTS on/off.
  - `security.hsts_max_age_seconds`: Cache duration (default: 31536000 = 1 year).
  - `security.hsts_include_subdomains`: Apply to all subdomains.
  - `security.hsts_preload`: Enable for HSTS preload list submission.
- Example header: `Strict-Transport-Security: max-age=31536000; includeSubDomains`
- Only meaningful when TLS is enabled; browsers ignore HSTS over HTTP.

7.6 Cross-Origin Resource Sharing (CORS)

CORS policy restricts which origins can access the API:

- Development mode: Allows `http://localhost:*` origins for local development.
- Production mode: Only allows origins specified in `security.cors_allowed_origins`.
- Credentials (`allow_credentials: true`) enabled for cookie-based authentication.
- Allowed methods: GET, POST, PUT, DELETE.
- Allowed headers: Content-Type, Authorization.
- Exposed headers: Set-Cookie (for session management).
- Implementation uses `tower-http::cors::CorsLayer`.

7.7 Role-Based Access Control

The application implements a simple role-based access control system:

- Two roles: `User` (default) and `Admin`.
- First registered user automatically becomes Admin.
- Admin capabilities:
  - View all users.
  - Change user roles.
  - Delete users (soft delete).
  - View and restore deleted posts.
- Role stored in `user_login.role` column.
- Protected endpoints use `AdminUser` extractor for authorization.

7.8 User Soft Delete

When an admin deletes a user, the system uses soft delete for security and audit purposes:

- Users are not removed from the database; instead, `deleted_at` timestamp is set.
- Soft-deleted users cannot log in (authentication queries filter by `deleted_at IS NULL`).
- Existing sessions of deleted users are invalidated (session validation checks user status).
- Posts and comments by deleted users display "Deleted User" instead of the username.
- This protects against inappropriate usernames remaining visible after account deletion.
- Username and email of deleted users remain reserved (prevents re-registration).
- Admin user list shows deleted users with visual indication for audit purposes.
- API responses include `isUserDeleted` boolean flag for frontend styling.

7.9 Database Encryption

Production database uses SQLCipher for at-rest encryption:

- Encryption key sourced from environment variable (`APPSEC_DB_KEY`).
- Key must be 64 hex characters (32 bytes).
- Falls back to system keyring if environment variable not set.
- Development mode uses unencrypted database for easier debugging.
- Key format: `"x'<HEX_KEY>'"` passed as PRAGMA.

7.10 Anti-Enumeration Measures

- Password reset always returns 200 OK regardless of email existence.
- Login returns generic "invalid credentials" for both wrong username and wrong password.
- Registration returns specific errors for taken username/email (trade-off for UX).

7.11 Error Handling

- Typed error codes from `api-types` crate.
- Specific codes for authentication failures.
- Validation errors include field-specific details for user feedback.
- Internal errors return generic code (HTTP 500).
- All error codes translated to user-friendly messages via `translator` crate.
- Stack traces and internal details never exposed to clients.

7.12 Automatic Cleanup

- Cleanup task runs asynchronously every hour.

- Deletes expired email verification tokens.
- Deletes expired password reset tokens.
- Deletes expired sessions.
- Unverified users with expired tokens deleted via CASCADE.
- On email sending failure, user record is rolled back immediately.

7.13 Input Validation

- Dual validation: frontend (WASM) and backend (native Rust).
- Same validation logic compiled for both platforms.
- Frontend validation for UX; backend is authoritative.
- Username: 3–20 characters, printable UTF-8.
- Email: Valid format validation.
- Password: 8–64 characters, complexity requirements.

7.14 Future Enhancements

- Rate limiting on login endpoint.
- Account lockout after failed attempts.
- CAPTCHA integration for bot mitigation.
- Device/session management (view and revoke sessions).
- Security event logging (failed logins, password resets).
- Multi-factor authentication (MFA/2FA).

8 API Reference

This section documents all authentication 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
- `POST /api/verify-email` – Verify email address
- `POST /api/login` – Authenticate and create session
- `POST /api/logout` – Destroy session
- `GET /api/auth/check` – Validate current session
- `POST /api/auth/refresh` – Extend session lifetime
- `POST /api/request-password-reset` – Request password reset
- `POST /api/complete-password-reset` – Complete password reset
- `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"
}
```

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.

Error Responses

HTTP 400 Bad Request – Token expired or invalid:

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

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 `POST /api/login`

Authenticates a user and creates a session.

Request Body (JSON)

```
{
  "username": "string",
  "password": "string"
}
```

Success Response (HTTP 200)

```
{
  "username": "string",
  "email": "string",
  "sessionExpiresAt": 1234567890, // Unix timestamp
  "sessionCreatedAt": 1234567890 // Unix timestamp
}
```

Also sets `session_token` HTTP-only cookie.

Error Responses

HTTP 400 Bad Request – Validation failed:

```
{
  "error": "VALIDATION",
  "validation": { ... }
}
```

HTTP 401 Unauthorized:

```
{
  "error": "INVALID_CREDENTIALS" | "EMAIL_NOT_VERIFIED"
}
```

8.5 `POST /api/logout`

Logs out the user by destroying their session.

Request

No body required. Uses `session_token` cookie for identification.

Success Response (HTTP 200)

Empty response body. Clears `session_token` cookie.

8.6 `GET /api/auth/check`

Validates the current session and returns user info.

Request

No body required. Uses `session_token` cookie.

Success Response (HTTP 200)

```
{
  "username": "string",
  "email": "string",
  "sessionExpiresAt": 1234567890,
  "sessionCreatedAt": 1234567890
}
```

Error Response

HTTP 401 Unauthorized:

```
{
  "error": "INVALID_CREDENTIALS"
}
```

8.7 `POST /api/auth/refresh`

Extends the current session's lifetime.

Request

No body required. Uses `session_token` cookie.

Success Response (HTTP 200)

```
{
  "username": "string",
  "email": "string",
  "sessionExpiresAt": 1234567890, // Updated expiry
  "sessionCreatedAt": 1234567890
}
```

Also updates `session_token` cookie expiry.

Error Response

HTTP 401 Unauthorized:

```
{
  "error": "INVALID_CREDENTIALS"
}
```

8.8 `POST /api/request-password-reset`

Requests a password reset email. Always returns success for security.

Request Body (JSON)

```
{
  "email": "string"
}
```

Success Response (HTTP 200)

Empty response body. If email exists, reset email is sent.

Note

Always returns 200 OK to prevent email enumeration attacks.

8.9 `POST /api/complete-password-reset`

Completes a password reset using the token from email.

Request Body (JSON)

```
{
  "token": "string", // 64-char hex token from email link
  "newPassword": "string" // Must meet password requirements
}
```

Success Response (HTTP 200)

Empty response body.

Error Responses

HTTP 400 Bad Request:

```
{
  "error": "INVALID_TOKEN" | "VALIDATION",
  "validation": { ... } // Only present for VALIDATION
}
```

8.10 `GET /api/health`

Simple health check endpoint.

Response (HTTP 200)

Returns basic health status indicating backend is running.

8.11 Common Error Codes

- `VALIDATION` – Input validation failed
- `INTERNAL` – Internal server error
- `INVALID_CREDENTIALS` – Authentication failed
- `EMAIL_NOT_VERIFIED` – Email verification required
- `TOKEN_EXPIRED` – Verification token expired
- `INVALID_TOKEN` – Reset token invalid or expired
- `USERNAME_TAKEN` – Username already registered
- `EMAIL_TAKEN` – Email already registered

8.12 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

8.13 Type Definitions

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

- `requests.rs`: `RegistrationRequest`, `EmailVerificationRequest`, `LoginRequest`, `PasswordResetReques`
  `PasswordResetCompleteRequest`
- `responses.rs`: `RegisterError`, `RegisterErrorResponse`, `VerifyEmailError`, `LoginResponse`, `LoginError`, `AuthSessionResponse`
- `validation.rs`: `ValidationFieldError`, `ValidationErrorData`
- `enums.rs`: `FieldType`, `ValidationErrorCode`, `PasswordStrength`

All types use conditional compilation for OpenAPI support:

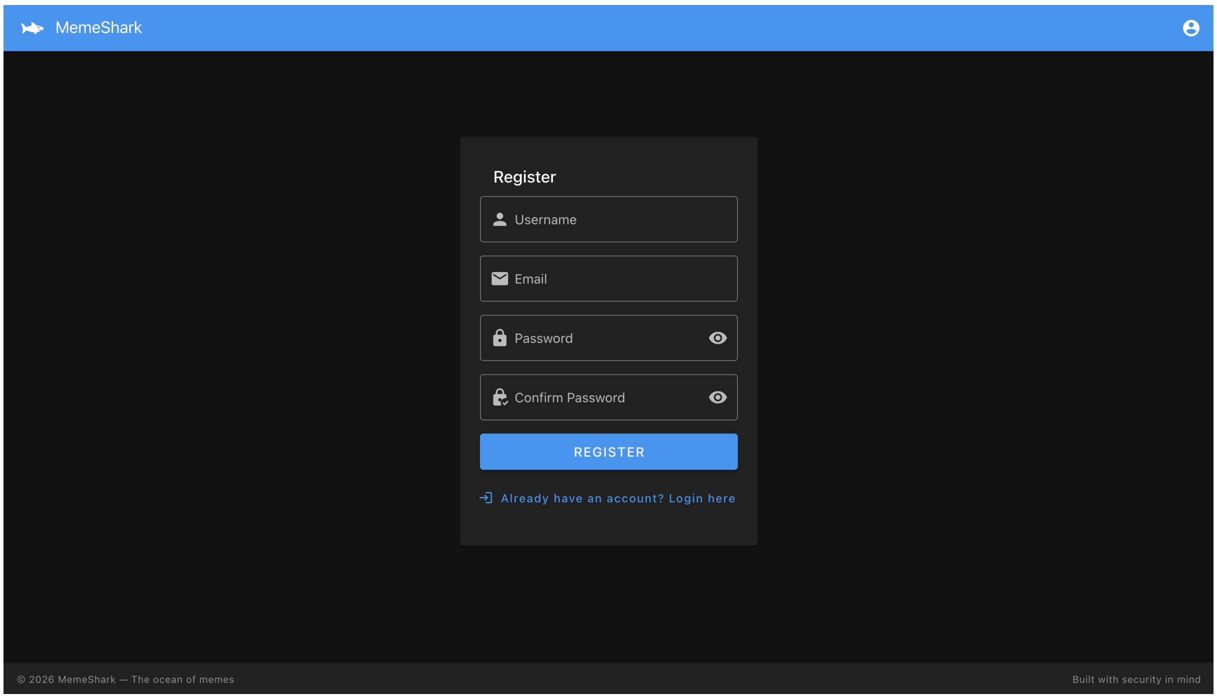
```
#[derive(Debug, Clone, Serialize, Deserialize)]
#[cfg_attr(feature = "openapi", derive(ToSchema))]
#[serde(rename_all = "camelCase")]
pub struct RegistrationRequest {
    pub username: String,
    pub email: String,
    pub password: String,
}
```

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

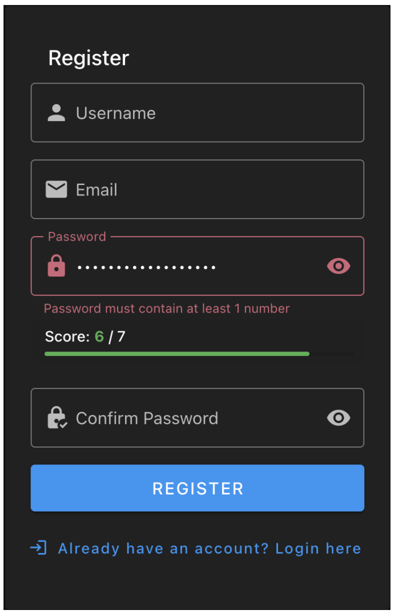
This section provides implementation details including code examples, screenshots, and the client generation process.

9.1 User Interface - Registration



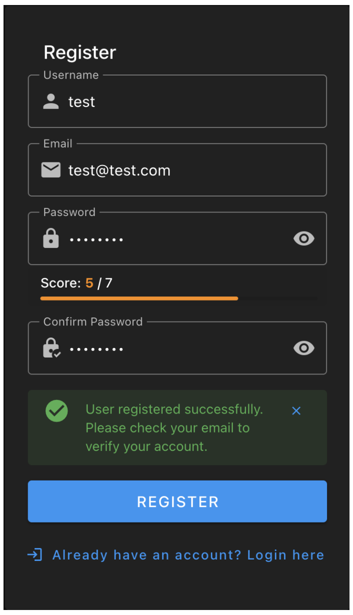
MemeShark
Register
Username
Email
Password
Confirm Password
REGISTER
Already have an account? Login here
© 2026 MemeShark — The ocean of memes
Built with security in mind

Figure 4: Registration form with real-time field validation. Shows username, email, and password fields with immediate feedback on input validity using WASM-based validation.



Register
Username
Email
Password
Password must contain at least 1 number
Score: 6 / 7
Confirm Password
REGISTER
Already have an account? Login here

Figure 5: 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.


Register

Username: test

Email: test@test.com

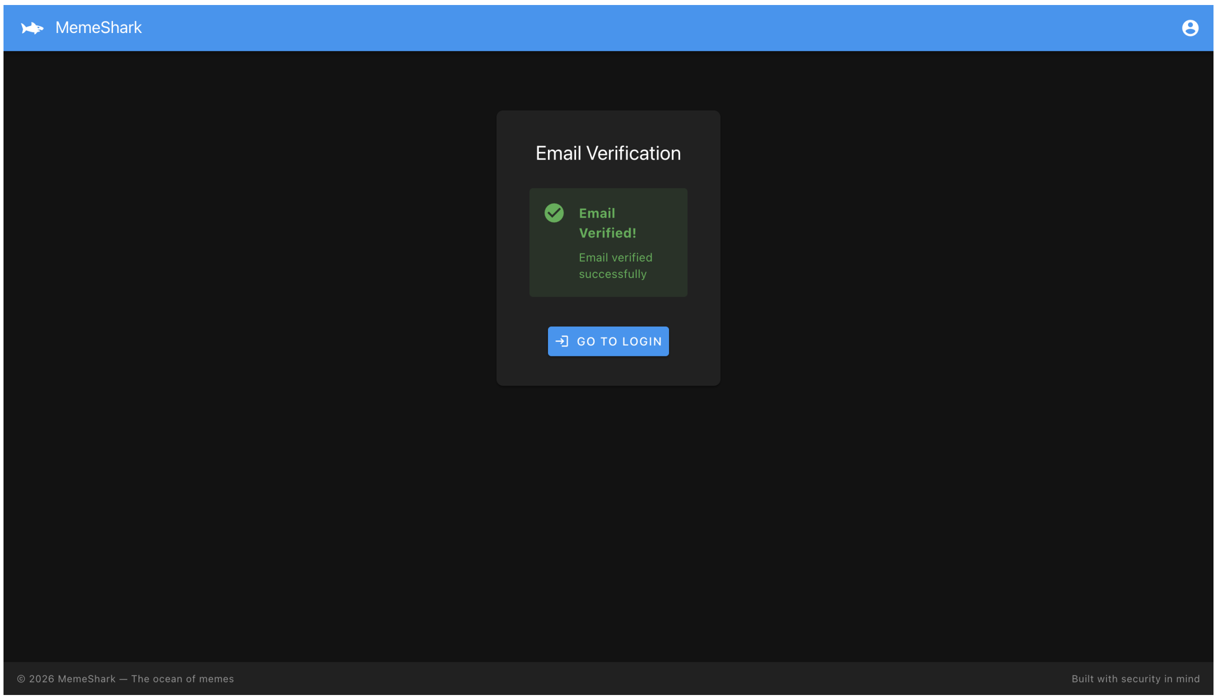
Password: ••••••••

Score: 5 / 7

Confirm Password: ••••••••

User registered successfully. Please check your email to verify your account.

Figure 6: Successful registration confirmation message. Informs the user that a verification email has been sent and provides next steps.


MemeShark

Email Verification

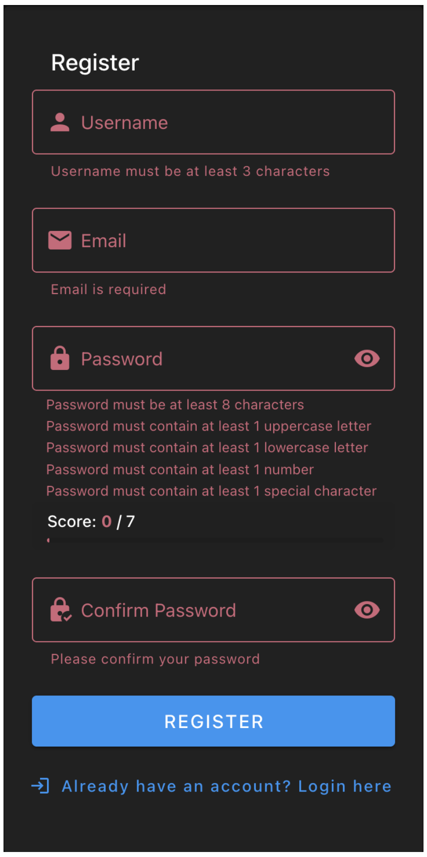
Email Verified!

Email verified successfully

© 2026 MemeShark — The ocean of memes

Built with security in mind

Figure 7: Email verification page showing successful account activation. Displayed after the user clicks the verification link from their email.



Username must be at least 3 characters

Email is required

Password must be at least 8 characters
Password must contain at least 1 uppercase letter
Password must contain at least 1 lowercase letter
Password must contain at least 1 number
Password must contain at least 1 special character
Score: 0 / 7

Please confirm your password

Figure 8: Form displaying validation errors with translated messages. Shows field-specific error messages generated by the WASM translator module.

9.2 User Interface - Login

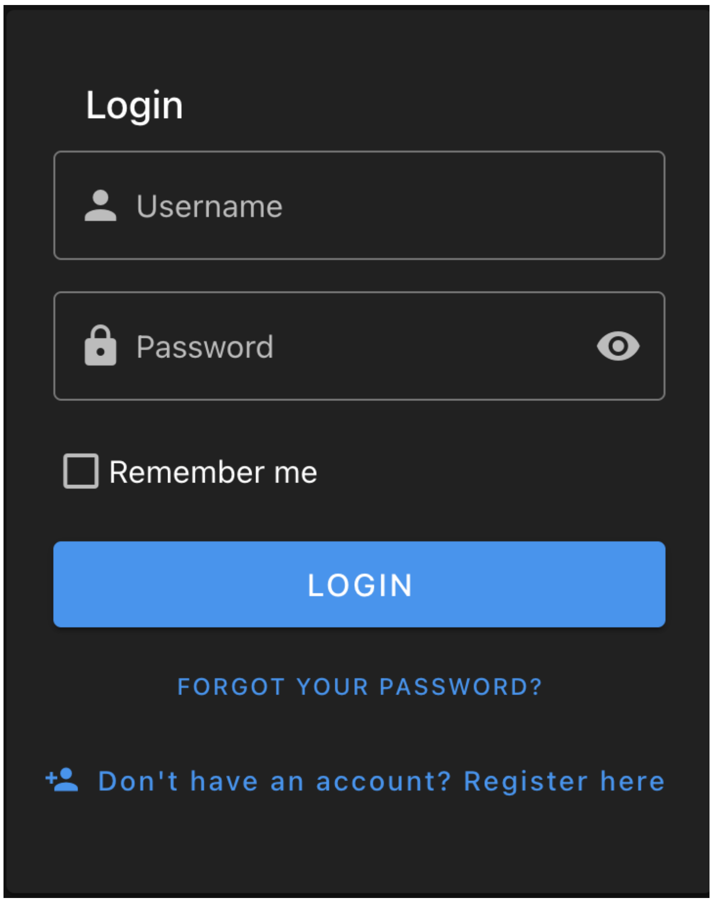
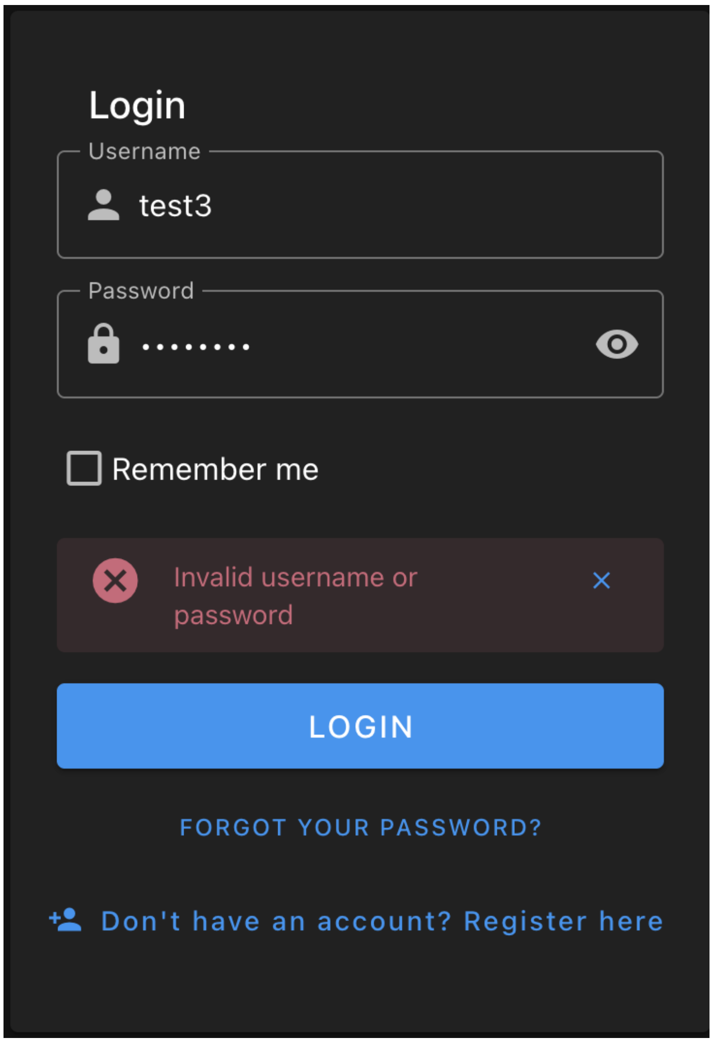


Figure 9: Login form with username and password fields. Includes “Remember me” checkbox and link to forgot password page.



Invalid username or password

Figure 10: Login error displaying translated error message for invalid credentials or unverified email.

9.3 User Interface - Password Reset

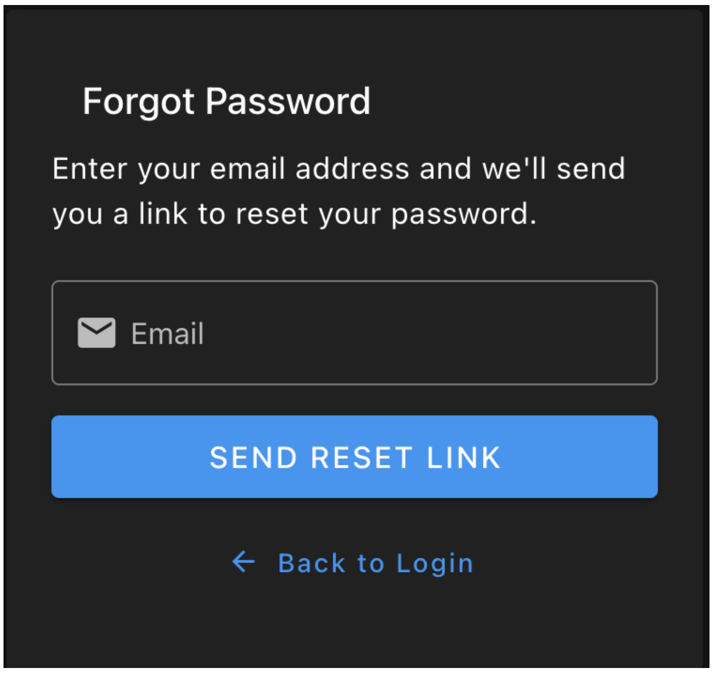

Forgot Password
Enter your email address and we'll send you a link to reset your password.
Email
SEND RESET LINK
Back to Login

Figure 11: Forgot password page where users enter their email address to request a password reset link.

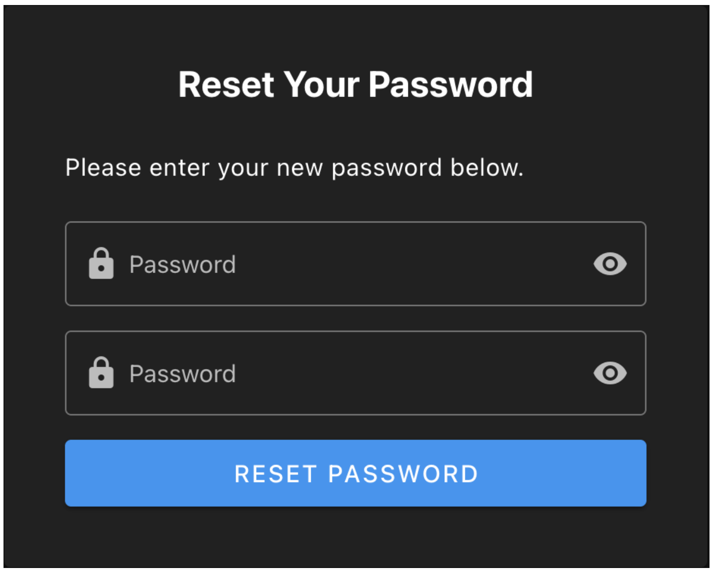
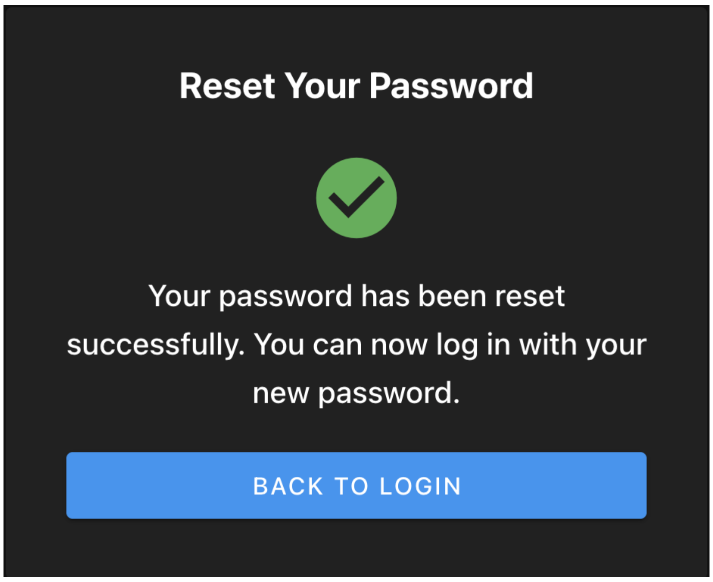


Figure 12: Password reset form where users enter their new password after clicking the reset link from email.


Reset Your Password

Your password has been reset successfully. You can now log in with your new password.

Figure 13: Successful password reset confirmation with link to login page.

9.4 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()`, `login()`, etc.)
- `index.ts`: Re-exports for convenient imports

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

9.5 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
        );
      });
    });
  }
}
```

9.6 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);
}
```

9.7 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
```

9.8 Session Management Implementation

Frontend session handling (`frontend/src/stores/auth.ts`):

```
// On login success
authStore.setUser(data) // Stores user in state and localStorage

// Session refresh scheduling
const sessionLifetime = sessionExpiresAt - sessionCreatedAt
const refreshAt = sessionCreatedAt + (sessionLifetime / 2)
// Timer calls POST /api/auth/refresh at 50% of session lifetime

// On logout
authStore.clearUser() // Clears state and localStorage
```

9.9 Backend Implementation Examples

9.9.1 Password Strength Calculation (Rust)

The `validate_password_detailed` function in `field-validator/src/lib.rs` calculates password strength:

```
pub fn validate_password_detailed(password: &str) -> String {
    let validation_errors = validate_password(password);

    // Calculate score based on various factors
    let mut score = 0u32;
    let len = password.len();

    let mut has_upper = false;
    let mut has_lower = false;
    let mut has_digit = false;
    let mut has_special = false;

    for c in password.chars() {
        has_upper |= c.is_uppercase();
        has_lower |= c.is_lowercase();
        has_digit |= c.is_numeric();
        has_special |= !c.is_alphanumeric();
    }

    score += (len >= 8) as u32; // +1 for length >= 8
    score += (len >= 12) as u32; // +1 for length >= 12
    score += (len >= 16) as u32; // +1 for length >= 16
    score += has_upper as u32; // +1 for uppercase
    score += has_lower as u32; // +1 for lowercase
    score += has_digit as u32; // +1 for digit
    score += has_special as u32; // +1 for special char

    let strength = match score {
        ..=PASSWORD_SCORE_WEAK_MAX => PasswordStrength::Weak,
        ..=PASSWORD_SCORE_MEDIUM_MAX => PasswordStrength::Medium,
        ..=PASSWORD_SCORE_STRONG_MAX => PasswordStrength::Strong,
        _ => PasswordStrength::Cia,
    };
    // Return JSON with errors, strength, and score
    serde_json::to_string(&ValidationDetailedPasswordData::new(
        validation_errors, strength, score
    )).unwrap_or_default()
}
```

9.9.2 Login Handler with Session Creation (Rust)

Key parts of the login handler in `backend/src/api/login.rs` showing session token generation and cookie setting:

```
// After successful password verification...
let session_token = generate_session_token(config.session.token_bytes);
let session_hash = hash_token(&session_token)?; // SHA256 hash for storage

let now = OffsetDateTime::now_utc();
let expiry = now + time::Duration::days(config.session.duration_days);
let session = UserSession {
    user_id: user.user_id,
    session_id: generate_session_id(),
    session_hash, // Only hash stored in database
    session_expiry: expiry,
    session_created_at: now,
};

db.user_sessions_table.insert(&session).await?;

// Set HTTP-only cookie with the actual token (not hash)
let cookie = create_session_cookie(session_token, Some(expiry), db.is_dev);
cookies.add(cookie);

// Return session metadata (timestamps for client-side refresh scheduling)
LoginResponse {
    username: user.username,
    email: user.email,
    role: user.role,
    session_expires_at: expiry.unix_timestamp(),
    session_created_at: now.unix_timestamp(),
}
```

9.9.3 Session Database Operations (Rust)

The `UserSession` struct and key operations from `backend/src/db/user_sessions.rs`:

```
#[derive(FromRow, Clone, Debug)]
```

```
pub struct UserSession {
    pub user_id: i64,
    pub session_id: String,
    pub session_hash: String, // SHA256 hash of session token
    pub session_expiry: OffsetDateTime,
    pub session_created_at: OffsetDateTime,
}

impl UserSessionsTable {
    pub async fn insert(&self, row: &UserSession) -> Result<()> {
        sqlx::query("INSERT INTO user_sessions (...) VALUES ($1, $2, $3, $4, $5)")
            .bind(row.user_id)
            .bind(&row.session_id)
            .bind(&row.session_hash)
            .bind(row.session_expiry)
            .bind(row.session_created_at)
            .execute(&self.conn_pool).await?;
        Ok(())
    }

    pub async fn get_by_hash(&self, session_hash: &str)
        -> Result<UserSession, sqlx::Error>
    {
        sqlx::query_as::<_, UserSession>(
            "SELECT * FROM user_sessions WHERE session_hash = $1"
        ).bind(session_hash).fetch_one(&self.conn_pool).await
    }

    pub async fn cleanup_expired_sessions(&self) -> Result<()> {
        sqlx::query("DELETE FROM user_sessions WHERE session_expiry < $1")
            .bind(OffsetDateTime::now_utc())
            .execute(&self.conn_pool).await?;
        Ok(())
    }
}
```

9.10 Implementation Files

9.10.1 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/login.rs`
  - `POST /api/login` handler.
  - Validates credentials, checks email verification.

  - Creates session and sets HTTP-only cookie.
- `backend/src/api/logout.rs`
  - `POST /api/logout` handler.
  - Deletes session from database.
  - Clears session cookie.
- `backend/src/api/auth.rs`
  - `GET /api/auth/check` – Validates session and returns user info.
  - `POST /api/auth/refresh` – Extends session lifetime.
- `backend/src/api/auth_extractor.rs`
  - Axum extractor for session validation.
  - Reads session token from cookie.
  - Validates against database.
- `backend/src/api/password_reset.rs`
  - `POST /api/request-password-reset` – Generates reset token.
  - `POST /api/complete-password-reset` – Validates token and updates password.
- `backend/src/db/user_sessions.rs`
  - Session table operations: create, lookup, delete, refresh.
- `backend/src/db/password_reset_tokens.rs`
  - Reset token table operations: create (UPSERT), lookup, delete.
- `backend/src/api/utils.rs`
  - Configuration constants from `build.rs`.
  - Token duration settings, base URLs.
  - Cookie creation helper functions.
- `backend/src/db/mod.rs`
  - `DBHandle` struct with connection pool.
  - Password hashing with Argon2.
  - Cleanup task for expired tokens and sessions.
- `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 and password reset email templates.

9.10.2 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/pages/login.vue`
  - Login page with username/password form.
  - Handles session storage on success.

> – Displays login errors with translation.
> `frontend/src/pages/forgot-password.vue`
> – Password reset request page.
> – Email input with validation.
> – Always shows success message (security).
> `frontend/src/pages/reset-password.vue`
> – Password reset completion page.
> – Extracts token from URL.
> – New password input with strength validation.
> `frontend/src/stores/auth.ts`
> – Pinia store for authentication state.
> – Manages user session data.
> – Handles session refresh scheduling.

- `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.

9.10.3 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`, `LoginResponse`, etc.
- `frontend/src/generated/api-client/sdk.gen.ts`
  – Auto-generated typed API functions.
  – Includes `registerUser()`, `verifyEmail()`, `login()`, `logout()`, etc.
- `frontend/src/generated/api-client/index.ts`
  – Re-exports all generated types and functions.

9.10.4 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.

9.10.5 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
```

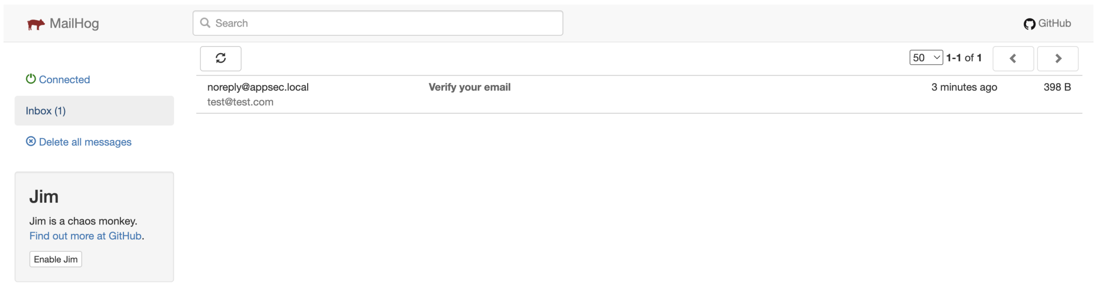
The build script handles:

- 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:

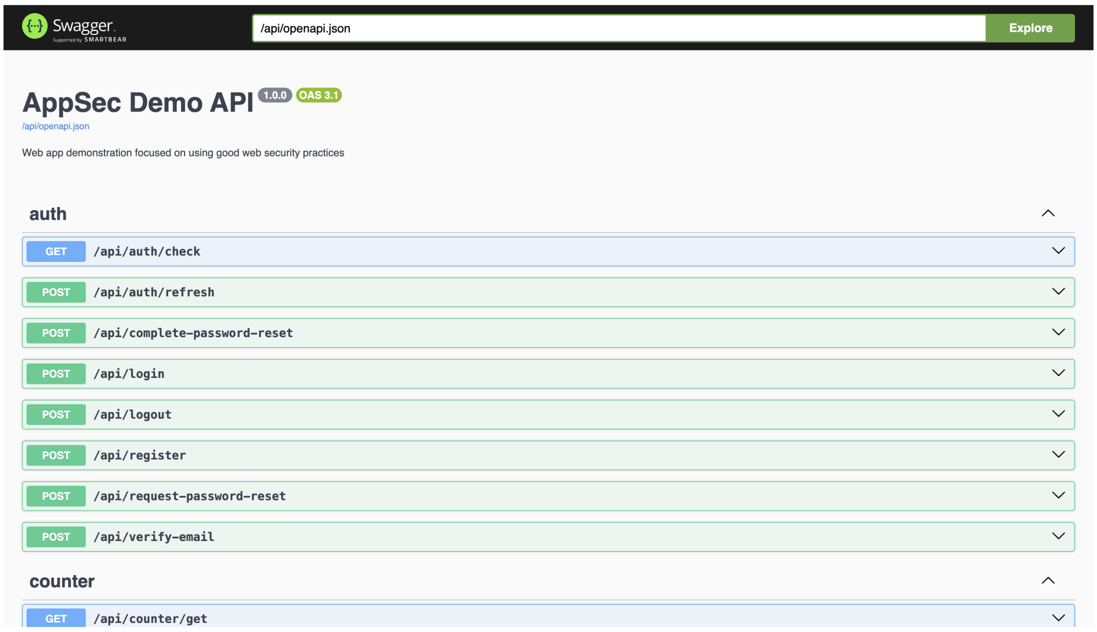
- Swagger UI available at `http://localhost:4000/api/docs`
- Emails sent to MailHog at `http://localhost:8025`
- Database is unencrypted (`data_dev.db`)



Jim is a chaos monkey.
Find out more at GitHub.

```
noreply@appsec.local    Verify your email    3 minutes ago    398 B
test@test.com
```

Figure 14: Swagger UI showing the API documentation interface available at `/api/docs` in development mode. Provides interactive testing of all API endpoints.



Web app demonstration focused on using good web security practices

```
GET  /api/auth/check
POST /api/auth/refresh
POST /api/complete-password-reset
POST /api/login
POST /api/logout
POST /api/register
POST /api/request-password-reset
POST /api/verify-email
GET  /api/counter/get
```

Figure 15: 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

10.4 Configuration

The application is configured via `config.toml` in the working directory. Key sections:

```
[server]
bind_addr = "0.0.0.0"      # Listen on all interfaces
port = 4000
dev_mode = false           # Production mode
max_body_size = 10485760   # 10MB

[database]
prod_path = "/app/data/data.db"
db_key_env_var_name = "APPSEC_DB_KEY"

[tls]
enabled = true
cert_path = "/app/certs/cert.pem"
key_path = "/app/certs/key.pem"

[security]
hsts_enabled = true
hsts_max_age_seconds = 31536000  # 1 year
hsts_include_subdomains = true
cors_allowed_origins = "https://example.com"

[mail]
smtp_host = "mailhog"
smtp_port = 1025
from_email = "noreply@example.com"
```

10.5 Docker Deployment

The application includes Docker support for containerized deployment.

10.5.1 Prerequisites

- Docker and Docker Compose installed
- TLS certificates (self-signed for testing or CA-signed for production)

10.5.2 Generate TLS Certificates

For local testing, generate self-signed certificates:

```
# Using the provided script
./generate-certs.sh

# Or manually with OpenSSL
openssl req -x509 -newkey rsa:4096 \
    -keyout certs/key.pem \
    -out certs/cert.pem \
    -days 365 -nodes \
    -subj "/CN=localhost/O=AppSec/C=US" \
    -addext "subjectAltName=DNS:localhost,IP:127.0.0.1"
```

10.5.3 Set Environment Variables

Create a `.env` file with the database encryption key:

```
# Generate a 64-character hex key (32 bytes)
echo "APPSEC_DB_KEY=$(openssl rand -hex 32)" > .env
```

10.5.4 Docker Compose Configuration

The `docker-compose.yml` defines the application stack:

```
services:
  app:
    build: .
    ports:
      - "4000:4000"
    environment:
      - APPSEC_DB_KEY=${APPSEC_DB_KEY}
    volumes:
      - app-data:/app/data
      - ./certs:/app/certs:ro
    depends_on:
      - mailhog

  mailhog:
    image: mailhog/mailhog:latest
    ports:
      - "8025:8025"  # Web UI
      - "1025:1025"  # SMTP

volumes:
  app-data:
```

10.5.5 Build and Run

```
# Build the Docker image
docker-compose build

# Start the containers
docker-compose up -d

# View logs
docker-compose logs -f app
```

```
# Stop the containers
docker-compose down

# Stop and remove volumes (clears database)
docker-compose down -v
```

10.5.6 Verify Deployment

Test the deployment with curl:

```
# Test HTTPS (use -k to accept self-signed cert)
curl -k https://localhost:4000/api/health

# Check HSTS header
curl -k -I https://localhost:4000/api/health

# Expected header:
# strict-transport-security: max-age=31536000; includeSubDomains
```

10.5.7 Accessing the Application

- Application: `https://localhost:4000`
- MailHog UI: `http://localhost:8025` (for viewing test emails)

10.6 Production Deployment Checklist

Before deploying to production:

- Use CA-signed TLS certificates (e.g., Let's Encrypt).
- Set a strong, randomly generated `APPSEC_DB_KEY`.
- Configure `cors_allowed_origins` with your domain.
- Update `base_url_prod` in config to your domain.
- Configure a real SMTP server for email delivery.
- Enable `hsts_preload` only if submitting to the HSTS preload list.
- Set up persistent storage for `/app/data` volume.
- Configure reverse proxy (nginx, Caddy) if needed.
- Set up monitoring and logging.

11 Conclusions

This project successfully implements a complete authentication module with secure user registration, login, session management, and password reset functionality. The implementation follows modern web security best practices and demonstrates a practical approach to building secure authentication systems.

11.1 Key Implementation Choices

11.1.1 Password Security

Argon2 was chosen as the password hashing algorithm due to its memory-hard properties, which provide strong resistance against GPU-based and ASIC-based attacks. The algorithm parameters are configured to balance security with acceptable response times. Password complexity requirements enforce a minimum of 8 characters with uppercase, lowercase, digit, and special character, while the strength scoring system (0–7) provides immediate user feedback.

11.1.2 Token Management

All tokens (email verification, session, password reset) use 32 bytes of cryptographically secure random data. Only SHA256 hashes are stored in the database, ensuring that even database breaches do not expose usable tokens. This approach follows the principle of storing only what is necessary for verification.

11.1.3 Session Security

Cookie-based sessions with HTTP-only and Secure flags prevent common attacks like XSS token theft. Server-side session storage means sensitive data never leaves the server, and the SameSite=Strict attribute provides strong CSRF protection by blocking cross-site cookie transmission. The automatic session refresh mechanism maintains user experience while ensuring sessions remain valid.

11.1.4 Cross-Platform Validation

The shared validation logic compiled to both WebAssembly (frontend) and native Rust (backend) ensures consistency and reduces code duplication. This approach allows real-time client-side validation for better user experience while maintaining authoritative server-side validation for security.

11.1.5 Anti-Enumeration Measures

The password reset endpoint always returns success regardless of whether the email exists, preventing attackers from enumerating valid accounts. Login failures use a generic "invalid credentials" message for both wrong username and wrong password cases.

11.2 Security Mechanisms Summary

The authentication module implements the following security mechanisms:

- **Defense in Depth**: Multiple layers of validation (frontend WASM, backend Rust) and security controls
- **Secure Storage**: Argon2 password hashing, SHA256 token hashing, SQLCipher database encryption
- **Transport Security**: HTTPS requirement in production, Secure cookie flags
- **Session Protection**: HTTP-only cookies, server-side storage, automatic expiry and cleanup

- **Input Validation**: Strict validation rules for all user inputs on both client and server
- **Error Handling**: Typed error codes with translation support, no internal details exposed
- **Automatic Cleanup**: Hourly removal of expired tokens, sessions, and unverified users

11.3 Future Improvements

While the current implementation provides a solid security foundation, several enhancements could further improve the system:

- **Rate Limiting**: Implement rate limiting on authentication endpoints to prevent brute-force attacks
- **Account Lockout**: Temporary account lockout after multiple failed login attempts
- **CAPTCHA**: Integration of CAPTCHA for bot mitigation on registration and login
- **Multi-Factor Authentication**: Add TOTP-based or email-based second factor
- **Device Management**: Allow users to view and revoke active sessions
- **Security Logging**: Comprehensive logging of security events for audit and monitoring

11.4 Lessons Learned

The development process highlighted several important considerations:

1. **Type Safety**: Using Rust's type system and shared crates across frontend (WASM) and backend significantly reduced bugs and ensured consistency.
2. **OpenAPI Integration**: Generating TypeScript clients from the OpenAPI specification eliminated manual API synchronization and reduced integration errors.
3. **Security vs. Usability**: Balancing security requirements with user experience required careful consideration, such as the trade-off between enumeration protection and helpful error messages during registration.
4. **Token Lifecycle**: Proper token management including generation, storage, validation, and cleanup is crucial for maintaining system security over time.

The authentication module demonstrates that modern web security practices can be implemented effectively without sacrificing developer experience or user usability, provided careful attention is paid to security requirements throughout the design and implementation process.

12 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
2.0	2026-01-12	Added login, session management, and password reset (draft)
2.1	2026-01-20	Final report with screenshots, implementation details, and conclusions
3.0	2026-01-20	Added TLS, HSTS, CORS, Docker deployment, first-user admin
3.1	2026-01-20	Added user soft delete functionality

Version 3.1 Changes:

- User soft delete instead of hard delete
- Deleted users cannot log in (sessions invalidated)

- Posts and comments by deleted users show “Deleted User”
- Admin user list shows deleted status for audit
- API responses include `isUserDeleted` flag for frontend styling
- Database schema updated with `deleted_at` column

Version 3.0 Changes:

- TLS/HTTPS support with configurable certificates
- HSTS (HTTP Strict Transport Security) middleware
- Production-safe CORS configuration
- First registered user automatically becomes admin
- Docker deployment with docker-compose
- Configuration via config.toml file
- Database encryption key from environment variable
- Updated SameSite cookie attribute to Strict

Document Generated: 2025-12-23
Last Updated: 2026-01-20
Status: Final
Review Status: Complete