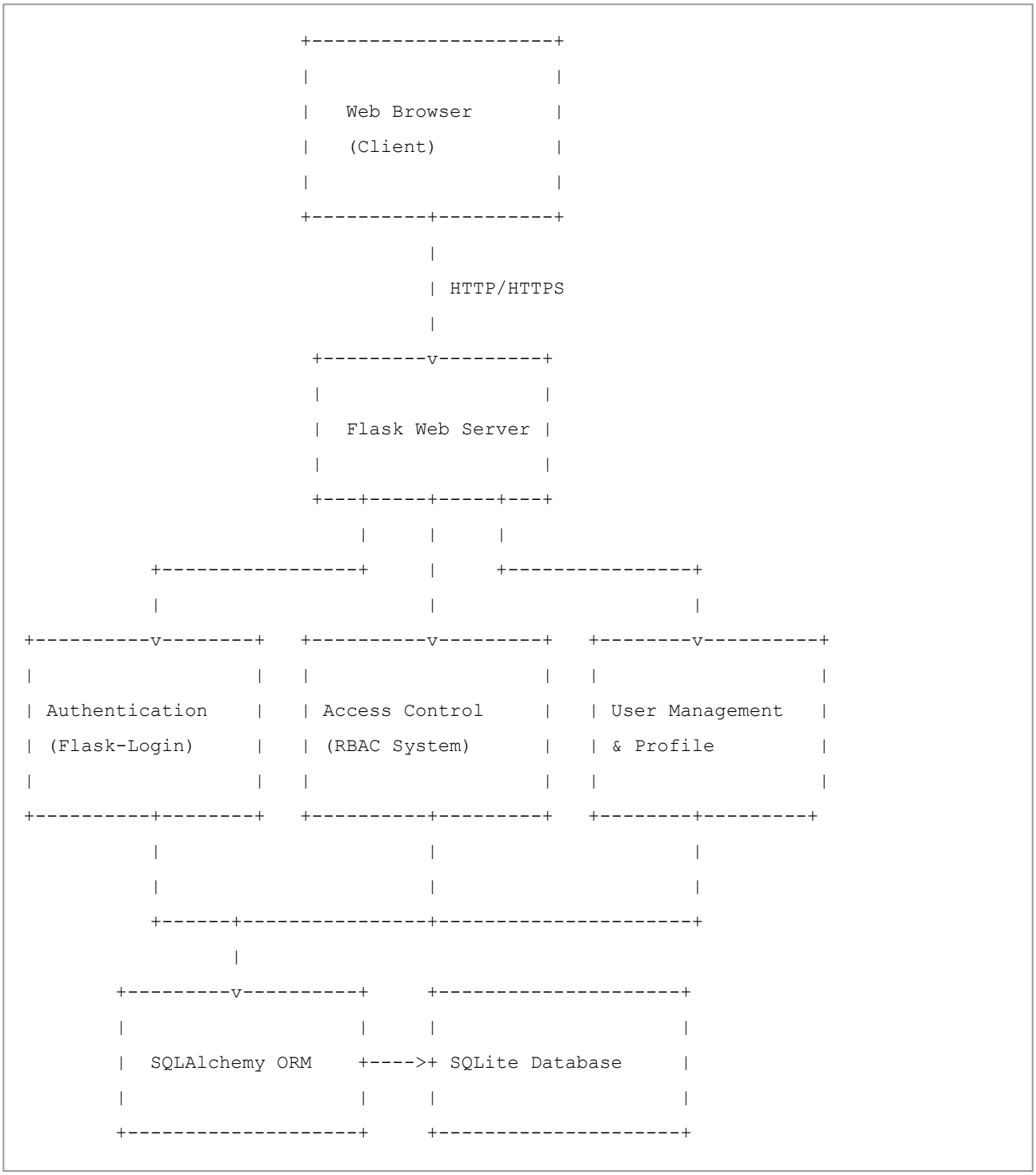
System Architecture for Access Control System

Overview

This document describes the architecture of the Access Control System, a Flask-based web application that implements secure authentication, role-based access control, user management, and profile management.

Architecture Diagram



Component Descriptions

Client Layer

- **Web Browser**: Users interact with the system through a standard web browser using HTML, CSS, and JavaScript.
- **Tailwind CSS**: Modern utility-first CSS framework for responsive design and UI components.
- **Alpine.js**: Lightweight JavaScript framework for reactive UI components.
- **Custom JavaScript**: Client-side validation, password strength meter, session timeout handling, table sorting.

Application Layer

- **Flask Framework**: Python web framework that provides the foundation for the application.
- **Blueprints**: Modular components for different application areas:
  - `auth_bp`: Authentication-related routes (login, logout, password reset)
  - `admin_bp`: Administration functions for user management
  - `profile_bp`: User profile management features

Security Layer

- **Flask-Login**: Handles user authentication and session management.
- **Flask-WTF**: Form handling with built-in CSRF protection.
- **Werkzeug Security**: Password hashing and verification.
- **Custom Security Utils**: Role-based access control, input validation, and security logging.

Data Access Layer

- **SQLAlchemy ORM**: Object-relational mapping for database interactions.
- **Models**: Data models representing application entities:
  - `User`: User account information and credentials
  - `Role`: User roles for access control
  - `AuditLog`: Security event logging
  - `PasswordResetToken`: Secure password reset functionality

Storage Layer

- **SQLite Database**: Lightweight embedded database for persistent storage.
- **File System**: Stores user profile images with secure naming conventions.

Key Design Patterns

Model-View-Controller (MVC)

- **Models**: SQLAlchemy database models in `app/models.py`
- **Views**: Jinja2 templates in `app/templates/`
- **Controllers**: Route functions in blueprint route files

Factory Pattern

- Application instance is created using a factory function (`create_app()`) to allow for testing and configuration flexibility.

Decorator Pattern

- Custom decorators for role-based access control (`admin_required`, `role_required`).

Repository Pattern

- Database models encapsulate data access methods for clean separation of concerns.

Security Architecture

Authentication Flow

1. User enters credentials on login page
2. Credentials are validated against hashed passwords in database
3. Failed login attempts are tracked and can trigger account lockout
4. Successful login creates a new session and logs the event

Authorization Flow

1. User requests access to a protected resource
2. Role-based middleware checks if the user has the required role
3. Access is granted or denied based on role verification
4. Authorization decisions are logged for audit purposes

Data Protection

- Passwords are hashed using bcrypt
- Form data is protected by CSRF tokens
- User inputs are validated and sanitized
- Security headers protect against common web vulnerabilities
- Session data is protected with secure cookies

Database Schema

The database schema includes the following main tables:

- `users`: User account information
- `roles`: Available roles in the system
- `user_roles`: Association table for user-role relationships
- `audit_logs`: Security event logging
- `password_reset_tokens`: Tokens for secure password reset

Module Dependencies

Backend Dependencies

- Flask core dependencies
- Authentication: flask-login
- Forms and validation: flask-wtf, email-validator
- Database: flask-sqlalchemy, sqlalchemy
- Security: bcrypt, werkzeug.security
- Configuration: python-dotenv
- (Note: Flask-Migrate was removed due to compatibility issues)

Frontend Dependencies

- Tailwind CSS: Modern utility-first CSS framework
- PostCSS: CSS transformations and processing
- Alpine.js: Lightweight JavaScript framework for reactivity
- Heroicons: SVG icon set

Error Handling

- Custom error handlers for common HTTP errors (404, 403, 500)
- Structured error logging
- User-friendly error messages without leaking sensitive information

Logging Architecture

- Authentication events: Success/failure logging
- Administrative actions: User management logging
- Security events: Access attempts, password changes
- Log rotation and storage management

Deployment Architecture

The application can be deployed using several approaches:

1. **Development**: Flask built-in server (not for production)
2. **Production**: WSGI server (Gunicorn/uWSGI) with reverse proxy (Nginx/Apache)
3. **Containerized**: Docker container with appropriate environment variables