User Authentication & Authorization System

A production-ready system for secure user management with Python, Flask, SQLite, and JWT.

This system handles **user registration, login, role-based access control, and secure API endpoints**. It's lightweight, easy to set up, and comes with both automated and manual testing options.

Python 3.8+ Flask 3.0.0 JWT Enabled SQLite Database

Prepared by: Saba Mazin Al-lamea

Table of Contents

1.Features

1.1 Core Functionality

- **User Registration & Login** – Create accounts and authenticate securely.
- **JWT Authentication** – Token-based, stateless security.
- **Role-Based Access Control** – Admin vs User roles.
- **Permissions System** – Fine-grained control for read, write, delete, and admin actions.
- **Protected Routes** – Decorators enforce authentication and authorization.
- **SQLite Database** – Simple, zero-configuration storage.

1.2 Security Features

- **Bcrypt Password Hashing** – Industry-standard hashing with salt.
- **Password Strength Enforcement** – Ensures strong passwords.
- **Email Validation** – Rejects invalid email formats.
- **SQL Injection Protection** – Parameterized queries.
- **Token Expiry** – Configurable JWT expiration.

1.3 Developer-Friendly Features

- Modular, easy-to-read code structure.
- Clear, human-readable error messages.
- Automated test suite for endpoints.
- Ready for **Google Colab** or local setup.

2.3System Architecture

```
Client (Postman, Browser)
     │
     ▼
 Flask App
```

```
  ├── Routes (register, login, users/me, admin)
  ├── Auth Layer (JWT, decorators)
  └── Models (User, Role, Permission, DB operations)
      |
      ▼
 SQLite Database (auth_system.db)
```

Requests flow from client → Flask routes → authentication → database.

---

3. Project Structure

Google Colab:

```
Google-Colab-Notebook/
├── Cell 1: Dependencies Installation
├── Cell 2: config.py - Configuration Settings
├── Cell 3: models.py - Database Models
├── Cell 4: auth.py - Authentication Logic
├── Cell 5: routes.py - API Endpoints
├── Cell 6: main.py - Application Entry & Server Start
├── Cell 7: Automated Testing Suite
├── Cell 8: Manual Testing Helper Functions
└── auth_system.db (auto-generated SQLite database)
```

Standard File Structure (For Local/Production):

```
auth-system/
├── main.py              # Entry point of the application
├── config.py            # Configuration settings (database URL, secret
key)
├── models.py            # Data models (User, Role, Permission)
├── auth.py              # Authentication logic and JWT handling
├── routes.py            # Route definitions for all endpoints
├── requirements.txt     # Python dependencies
├── test_auth.py         # Unit tests with pytest
├── README.md            # Documentation
├── .env                 # Environment variables (optional)
└── auth_system.db       # SQLite database (auto-generated)
```

---

4. Prerequisites

- Python 3.8+
- pip (package manager)
- Git (optional)

Python packages:

Flask==3.0.0

PyJWT==2.8.0

Werkzeug==3.0.1

pytest==7.4.3

requests==2.31.0

pyngrok==7.0.0 # For Colab only

---

5. Installation Guide

Google Colab Setup

1. Open a new Google Colab notebook.
2. Install required Python packages (Flask, PyJWT, Werkzeug, requests, pytest, pyngrok).
3. Configure ngrok with your authentication token to expose the Flask app publicly.
4. Run the notebook cells in order to initialize the database, start the Flask app, and generate the public URL.
5. Test endpoints using automated tests or manual tools (cURL/Postman) via the public URL.

Local Machine Setup

1. Clone the repository or download the project ZIP.
2. Create a Python virtual environment and activate it.
3. Install dependencies using the requirements.txt file.
4. Optionally, create a .env file to store environment variables like SECRET_KEY, database path, JWT expiration, and Bcrypt configuration.

5. Start the Flask server locally; it will run at http://127.0.0.1:5000.
6. Run automated tests or access endpoints manually using cURL or Postman.

---

6. Configuration

The configuration of the system is handled through a central **config file** (config.py) and optional **environment variables**. This allows easy customization without changing the main code.

Main Settings

1. **Secret Key**
   - Used for signing JWT tokens.
   - Should be a **strong, unique string** in production.
   - Can be set via environment variable SECRET_KEY.
2. **JWT Settings**
   - Algorithm: HMAC-SHA256 (HS256)
   - Expiration: default **24 hours**, configurable via JWT_EXPIRATION_HOURS.
3. **Database**
   - SQLite database path can be customized with DATABASE_PATH.
   - Default: auth_system.db.
4. **Password Hashing**
   - Uses **Bcrypt** with a configurable number of hashing rounds (BCRYPT_LOG_ROUNDS).
   - Ensures secure password storage.
   -

Environment Variables Overview

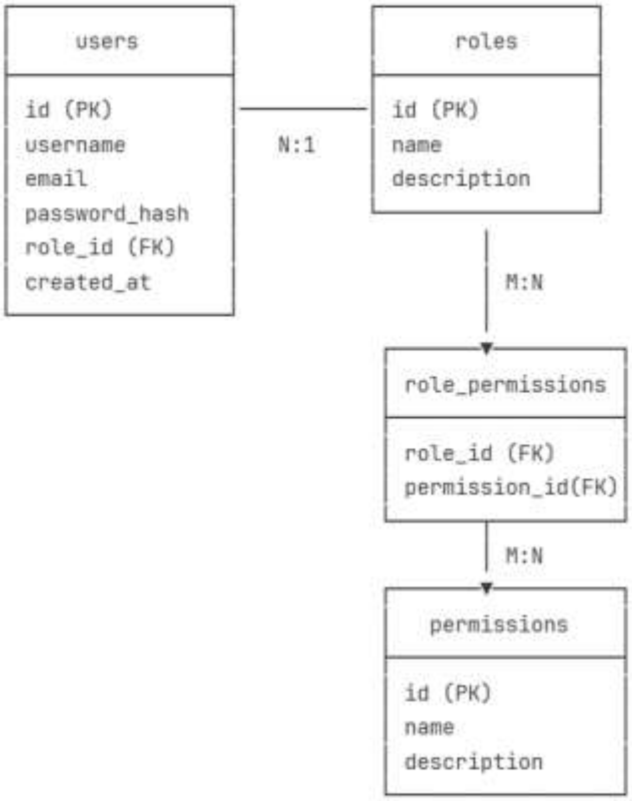
Variable	**Purpose**	**Default**	**Required**
SECRET_KEY	JWT signing key	Auto-generated	Yes (production)
DATABASE_PATH	SQLite database location	auth_system.db	No
JWT_EXPIRATION_HOURS	Token lifetime in hours	24	No

Variable	Purpose	Default	Required
BCRYPT_LOG_ROUNDS	Number of hashing rounds	12	No

Security Tip: Never expose the secret key publicly. Always store sensitive values in environment variables for production.

7.Database Schema

- **Tables**: users, roles, permissions, role_permissions
- **Default roles**: user (read/write), admin (read/write/delete/admin)
- **Password storage**: hashed with Bcryp



8.API Endpoints

Method	Endpoint	Auth	Role	Description
GET	/	No	None	API info
POST	/register	No	None	Create new user
POST	/login	No	None	Login user
GET	/users/me	Yes	Any	Get current user
GET	/admin	Yes	Admin	Admin-only route

9.Authentication & Authorization

□ JWT tokens include: user_id, email, role, exp.

□ **Decorators**:

- @login_required → Any logged-in user
- @admin_required → Admin only
- @permission_required('delete') → Specific permission

10.Testing Guide

Automated

pytest test_auth.py -v

Manual (cURL)

- Register, login, access /users/me, test admin routes.
- Postman collection recommended with environment variables for tokens.

H□ **Automated tests**: verify registration, login, protected routes, admin access.

□ **Manual testing**: use Postman or cURL with JWT tokens.

□ **Test coverage**: user registration, duplicate prevention, password strength, login validation, role-based access.

11.Usage Examples

- Register a user, obtain a JWT token, and use it to access protected endpoints.
- Admin flow: register admin, obtain token, access admin-only endpoints.
- Error handling: invalid credentials, missing token, weak password triggers proper HTTP error codes.

12.Security Highlights

- Strong password hashing with Bcrypt.
- JWT tokens with expiration; stateless authentication.
- Role-based and permission-based access control.
- Input validation and SQL injection prevention.
- No sensitive data stored in tokens.
- Ready for HTTPS and optional rate-limiting and CORS.

13.Troubleshooting

- **Missing modules** → install required packages.
- **Database locked** → close connections or restart runtime.
- **Invalid/expired token** → re-login to get a new token.
- **Ngrok errors (Colab)** → set correct auth token.
- **Port conflicts** → change Flask server port.
- **CORS issues** → install and configure Flask-CORS.

14.Deployment

Options:

- **Heroku**: push repo, configure environment variables, deploy.
- **Railway**: initialize project, set environment, deploy.
- **PythonAnywhere**: upload code, configure virtual environment, WSGI, and environment variables.