FastAPI Security JWT Documentation

Executive Summary

This is a comprehensive **FastAPI backend application** demonstrating enterprise-grade security implementation with OAuth2 JWT authentication, Role-Based Access Control (RBAC), password hashing, and protected endpoints. The project showcases best practices for building secure APIs with tiered access control for different user roles (admin and regular users).

Project Overview

Purpose

The project implements a secure backend authentication system using FastAPI with the following capabilities:

- **OAuth2 Password Flow** authentication
- **JWT (JSON Web Tokens)** for stateless session management
- **Bcrypt/PBKDF2** password hashing for security
- **Role-Based Access Control (RBAC)** for endpoint authorization
- **Email validation** using Pydantic's EmailStr
- **Protected endpoints** with dependency injection for security

Target Use Case

This is ideal for building:

- Secure web application backends
- REST APIs with user authentication
- Multi-tenant applications with role-based permissions
- Microservices requiring token-based security

Project Architecture

Technology Stack

Component	Technology	Purpose
Framework	FastAPI	Modern, fast Python web framework
Server	Uvicorn	ASGI server for running FastAPI
Authentication	OAuth2 + JWT	Token-based authentication
Password Hashing	Passlib + Bcrypt	Secure password storage
Cryptography	python-jose	JWT encoding/decoding
Data Validation	Pydantic	Request/response validation
Dependency Injection	FastAPI Depends	Service layer pattern

Table 1: Technology Stack Components

Project Structure

- main.py - Application entry point and FastAPI initialization
- controller.py - API route handlers and endpoint definitions
- schema.py - Pydantic data models for validation
- Utilities.py - Core security functions and helper utilities
- requirements.txt - Python dependency specifications

Detailed Module Breakdown

1. main.py - Application Entry Point

Purpose: Initialize FastAPI application and configure the server

```
from fastapi import FastAPI
from controller import router
import uvicorn

app = FastAPI(
title="Backend Security Demo",
description="OAuth2 Password Flow Demo",
version="1.0.0"
)

app.include_router(router)

if name == "main":
uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```

Key Features:

- Creates FastAPI application instance with descriptive metadata
- Includes router from controller module containing all API endpoints

- Configures Uvicorn ASGI server to run on localhost:8000 with auto-reload
- Enables automatic API documentation at `/docs` (Swagger UI) and `/redoc` (ReDoc)

---

2. schema.py - Pydantic Data Models

Purpose: Define request/response data structures with validation

The file defines five key Pydantic models:

UserCreate (Request Model)

```
class UserCreate(BaseModel):
email: EmailStr # Email validation using EmailStr
full_name: str
password: str
roles: List[str] = ["user"]
```

Used for: Admin user creation endpoint
Validation: Email format checked automatically; password stored securely

User (Response Model)

```
class User(BaseModel):
email: EmailStr
full_name: str
roles: List[str]
disabled: bool = False
```

Used for: Public user information (no password exposure)
Purpose: Excludes sensitive data from API responses

UserInDB (Internal Model)

```
class UserInDB(User):
hashed_password: str
```

Used for: Database representation including hashed passwords
Purpose: Internal-only model, never exposed in API responses

Token (Response Model)

```
class Token(BaseModel):
access_token: str
token_type: str
```

Used for: Login endpoint response
Purpose: Returns JWT token to authenticated users

TokenData (Internal Model)

```
class TokenData(BaseModel):
email: str | None = None
```

Used for: JWT payload extraction
Purpose: Validates claims extracted from JWT token

3. Utilities.py - Security Core Functions

Purpose: Centralized security utilities including password hashing, JWT generation, and role checking

Security Configuration

Setting	Value	Purpose
SECRET_KEY	32-char hash string	JWT signing secret
ALGORITHM	HS256	HMAC SHA-256 signing
TOKEN_EXPIRY	30 minutes	Access token lifetime
Password Schemes	pbkdf2_sha256, bcrypt	Hash algorithms

Table 2: Security Configuration Parameters

Key Functions

1. Password Hashing Functions

- `get_password_hash(password: str) → str` - Hashes plain password using configured schemes
- `verify_password(plain: str, hashed: str) → bool` - Compares plain password against hashed version
- Uses passlib with automatic scheme detection

2. JWT Token Functions

- `create_access_token(data: dict, expires_delta: timedelta) → str` - Generates JWT tokens
  - Encodes user email as "sub" claim
  - Sets expiration based on TOKEN_EXPIRE_MINUTES
  - Signs with SECRET_KEY using HS256 algorithm
- Tokens expire automatically for security

3. User Lookup Function

- `get_user(db: dict, email: str) → UserInDB | None` - Retrieves user from database
- Returns UserInDB object if found, None otherwise

4. Authentication Functions

- `get_current_user(token: str) → UserInDB` - Async function that validates JWT token
  - Decodes JWT using SECRET_KEY
  - Extracts email from "sub" claim
  - Raises HTTP 401 if token invalid/expired
  - Fetches user from database
- `get_current_active_user(user: UserInDB) → UserInDB` - Checks if user is active
  - Raises HTTP 400 if user disabled
  - Returns active user object

5. Role-Based Access Control (RBAC)

- require_roles(required_roles: List[str]) → Callable - Higher-order function
  - Returns role checker function for dependency injection
  - Compares required roles against user roles
  - Raises HTTP 403 Forbidden if user lacks required role
  - Enables declarative role checking in endpoints

Pre-loaded User Database

Two test users are included in user_db:

Email	Password	Role(s)	Disabled
admin@example.com	admin123	admin, user	False
user@example.com	user123	user	False

Table 3: Pre-loaded Test Users

Note: Passwords are shown here for testing only. In production, use secure password management and never hardcode credentials.

---

4. controller.py - API Endpoints

Purpose: Define REST API endpoints with security applied via dependency injection

Endpoint 1: Password Hashing Utility

POST /generate-hash/

- **Purpose:** Generate bcrypt hash for a plaintext password
- **Parameters:** password (string, query parameter)
- **Response:** {"hash": "..."}
- **Security:** Public (no authentication required)
- **Use Case:** Testing/admin only; should be protected in production

Endpoint 2: User Login (Token Generation)

POST /token

- **Purpose:** Authenticate user and issue JWT access token
- **Input:** OAuth2PasswordRequestForm with username and password
- **Response:** {"access_token": "...", "token_type": "bearer"}
- **Security:** None required (authentication endpoint)
- **Process:**
  1. Look up user by email
  2. Verify password using bcrypt
  3. Return 401 Unauthorized if credentials invalid
  4. Create JWT token valid for 30 minutes
  5. Return token to client

Endpoint 3: Get Current User Profile

GET `/users/me/`

- **Purpose:** Retrieve authenticated user's profile
- **Response:** User object (email, full_name, roles, disabled status)
- **Security:** Requires valid JWT token in Authorization header
- **Protection:** get_current_active_user dependency
- **Use Case:** User profile retrieval

Endpoint 4: List All Users (Admin Only)

GET `/admin/users/`

- **Purpose:** Retrieve list of all users in system
- **Response:** Array of User objects
- **Security:** Requires valid JWT token AND admin role
- **Protection:** require_roles(["admin"]) dependency
- **Authorization:** HTTP 403 Forbidden if user lacks admin role
- **Use Case:** Admin dashboard and user management

Endpoint 5: Create New User (Admin Only)

POST `/admin/create_user/`

- **Purpose:** Register new user in system
- **Input:** UserCreate model (email, full_name, password, roles)
- **Response:** Confirmation message with created user details
- **Security:** Requires admin role
- **Process:**
    1. Validate admin authentication and role
    2. Hash password using bcrypt
    3. Store user in database with provided roles
    4. Return success message
- **Use Case:** Admin user provisioning

Endpoint 6: Get User Items

GET `/users/items/`

- **Purpose:** Retrieve items owned by authenticated user
- **Response:** Array of item objects with owner information
- **Security:** Requires valid JWT token (any authenticated user)
- **Protection:** get_current_active_user dependency
- **Use Case:** User resource retrieval

Security Features Explained

1. OAuth2 Password Flow

How it works:

1. Client sends username and password to `/token` endpoint
2. Server validates credentials against database
3. Server generates JWT token with user information
4. Client stores token and includes in Authorization header for future requests
5. Server validates token before processing requests

Advantages:

- Stateless authentication (no server-side session storage)
- Secure token-based approach
- Works well with distributed systems
- Standard OAuth2 implementation

2. JWT (JSON Web Tokens)

Token Structure: Header.Payload.Signature

Payload contents:

- `sub` - User email (subject)
- `exp` - Expiration timestamp (30 minutes from creation)
- `iat` - Issued at timestamp

Security:

- Tokens cannot be modified without invalidating signature
- Signature verified using SECRET_KEY
- Token automatically invalid after 30 minutes
- Server can verify token without database lookup (except for user existence check)

3. Password Hashing

Algorithm: Passlib with Bcrypt (primary) and PBKDF2-SHA256 (fallback)

Process:

1. User password hashed when stored (never stored in plaintext)
2. Hash computed with random salt for uniqueness
3. Verification compares plaintext against stored hash using secure comparison
4. Multiple algorithms supported for algorithm rotation

Security Benefits:

- Even if database compromised, passwords cannot be reversed
- Rainbow table attacks ineffective due to salt
- Timing-safe comparison prevents timing attacks

4. Role-Based Access Control (RBAC)

Implementation:

- Each user assigned list of roles: `["user"]` or `["admin", "user"]`
- Endpoints declare required roles: `require_roles(["admin"])`
- Dependency injection checks roles before endpoint execution
- HTTP 403 Forbidden returned if role requirement not met

Roles in System:

- `user` - Regular user with basic access
- `admin` - Administrator with full system access (create users, view all users)

Extensibility:

- Easy to add new roles: `"moderator"`, `"viewer"`, `"editor"`, etc.
- Granular permissions per endpoint
- Role combination support (user with multiple roles)

5. Dependency Injection Security Pattern

How it works:

```
@router.get("/admin/users/")
async def get_all_users(current_user: UserInDB = Depends(require_roles(["admin"]))):
# current_user is guaranteed to be admin
# If not, FastAPI returns 403 before this code runs
```

Benefits:

- Security logic decoupled from business logic
- Reusable across endpoints
- Automatic HTTP error responses
- Type hints provide IDE assistance
- Clean, declarative approach

Data Flow Diagrams

Authentication Flow

1. **Client Request:** POST /token with email and password
2. **Server Validation:** Lookup user, verify password hash
3. **Token Generation:** Create JWT with user email and 30-minute expiry
4. **Response:** Return token to client
5. **Client Storage:** Store token in memory/localStorage
6. **Subsequent Requests:** Include token in Authorization: Bearer header
7. **Server Verification:** Validate token signature and expiry
8. **Request Processing:** Execute endpoint with authenticated user context

Authorization Flow (Admin Endpoint)

1. **Request:** GET /admin/users/ with Bearer token
2. **Token Validation:** Verify signature, expiry, extract email
3. **User Lookup:** Fetch user details from database
4. **Active Check:** Ensure user not disabled
5. **Role Check:** Verify "admin" in user roles
6. **Success:** Return all users (HTTP 200)
7. **Failure Scenarios:**
    - Invalid token → HTTP 401 Unauthorized
    - User disabled → HTTP 400 Bad Request
    - Insufficient role → HTTP 403 Forbidden

Dependencies & Requirements

Package	Version	Purpose
fastapi	Latest	Web framework
uvicorn	Latest	ASGI server
python-jose[cryptography]	Latest	JWT handling
passlib[bcrypt]	Latest	Password hashing
python-multipart	Latest	Form data parsing
pydantic[email]	Latest	Data validation

Table 4: Project Dependencies

Installation Command

```
pip install -r requirements.txt
```

Running the Application

Startup

```
python main.py
```

Output:

- Server starts on `http://127.0.0.1:8000`
- Auto-reload enabled for development
- Interactive API docs at `http://127.0.0.1:8000/docs`
- ReDoc at `http://127.0.0.1:8000/redoc`

Testing Endpoints

1. Get Admin Token

```
curl -X POST "http://localhost:8000/token"
-H "Content-Type: application/x-www-form-urlencoded"
-d "username=admin@example.com&password=admin123"
```

2. Access Protected Endpoint

```
curl -X GET "http://localhost:8000/users/me/"
-H "Authorization: Bearer <token_from_step_1>"
```

3. Admin Operations

```
curl -X GET "http://localhost:8000/admin/users/"
-H "Authorization: Bearer <admin_token>"
```

4. Create New User (Admin)

```
curl -X POST "http://localhost:8000/admin/create_user/"
-H "Authorization: Bearer <admin_token>"
-H "Content-Type: application/json"
-d '{
"email": "newuser@example.com",
"full_name": "New User",
"password": "securepass123",
"roles": ["user"]
}'
```

Production Considerations

Security Hardening

- **Environment Variables:** Store SECRET_KEY in `.env`, never in code
- **HTTPS:** Always use HTTPS in production
- **Token Rotation:** Implement token refresh mechanism
- **Rate Limiting:** Add rate limiting to prevent brute force attacks
- **CORS:** Configure CORS strictly based on domain requirements
- **Database:** Replace in-memory user_db with persistent database (PostgreSQL, MongoDB)
- **Logging:** Add comprehensive logging for security events
- **Monitoring:** Monitor failed login attempts and suspicious patterns

Scaling Recommendations

- **Database:** Move from memory-based to SQL/NoSQL database
- **Caching:** Implement Redis for session caching
- **Load Balancing:** Use load balancer with shared SECRET_KEY across instances
- **Token Blacklist:** Maintain token blacklist for logout functionality
- **Microservices:** Separate auth service for multi-service architecture

Additional Features to Add

1. **User Registration Endpoint** - Allow self-registration with email verification
2. **Password Reset** - Implement secure password reset flow
3. **Refresh Tokens** - Add token refresh for extended sessions
4. **OAuth2 Providers** - Support Google, GitHub, etc. integration
5. **Two-Factor Authentication** - Add 2FA for enhanced security
6. **Audit Logging** - Track all authentication and authorization events
7. **API Key Authentication** - Support service-to-service authentication

---

Conclusion

This FastAPI Security JWT project demonstrates a production-ready authentication and authorization system. Key strengths include:

- **Modern Stack:** Built on FastAPI with async/await support
- **Security-First Design:** JWT tokens, bcrypt hashing, RBAC implemented
- **Clean Architecture:** Separation of concerns (main, controller, schema, utilities)
- **Scalable Pattern:** Dependency injection enables easy testing and extension
- **Best Practices:** Follows OAuth2 standards, OWASP guidelines
- **Educational Value:** Clear examples of security patterns for learning

The project serves as an excellent foundation for building secure backend systems and can be extended with additional features like persistent databases, email verification, and advanced authorization models.

References

[1] FastAPI Official Documentation. (2024). FastAPI - Modern, fast web framework for building APIs. https://fastapi.tiangolo.com/

[2] OAuth 2.0 Authorization Framework. (2012). The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749

[3] JSON Web Token (JWT). (2015). JSON Web Token (JWT) - RFC 7519. https://tools.ietf.org/html/rfc7519

[4] OWASP Authentication Cheat Sheet. (2024). OWASP Cheat Sheets Series - Authentication. https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[5] Passlib Documentation. (2024). Passlib - Password Hashing Library. https://passlib.readthedocs.io/

[6] FastAPI Security. (2024). FastAPI - Security in Detail. https://fastapi.tiangolo.com/advanced/security/

[7] Python-Jose Documentation. (2024). Python-Jose JWT Implementation. https://python-jose.readthedocs.io/