

Authentication & Authorisation in a Software



Secure, Scalable, and Professional

A comprehensive guide to implementing robust security in financial applications



jwt-fastapi.zip

Agenda

Core Concepts

Authentication vs Authorization

JWT Deep Dive

Why JSON Web Tokens?

Java Implementation

Spring Security & JWT

Python Implementation

FastAPI & JWT

Best Practices

Security & Pro Tips

Conclusion

Key Takeaways

Authentication vs Authorization

Authentication

Who are you?

Verifying the identity of a user or system.

Examples: Username/Password, Biometrics, OTP

Think of it as showing your ID to enter a building.

Authorization

What are you allowed to do?

Determining what resources a user has access to.

Examples: Admin access, Read-only access, Feature access

Think of it as the areas you're allowed to go inside the building.

Password vs. Passkey

Password		Passkey
Nature	User-created secret string	Device-bound cryptographic key pair
Storage	Showing an ID to enter a building	Stored on server (hashed or encrypted)
Security	Prone to phishing, brute-force, leaks	Resistant to phishing, device-bound, uses biometrics
User action	Must remember and type	No need to remember; can use face/fingerprint/etc.
Adoption	Universal	Growing, but not yet universal

Alternatives Gaining Ground

Opaque Reference Tokens: These random strings (used with OAuth2) require server validation for each request, combining much of the session model’s revocability and centralized control with the flexibility needed for APIs.

PASETO: This new token standard is designed to be “secure by default” and addresses many JWT weaknesses, potentially positioning itself as a more resilient alternative for stateless authentication in “next gen” systems

Session Cookies: These are also gaining grounds with enhanced security added. This is due to their simplicity and inherent scalability.

Why JSON Web Tokens (JWT)?

Digital Passport for Your web App 🛂

JWT Structure: `header.payload.signature`

Header

Metadata about the token (algorithm, type)

Payload

Claims about the user (ID, role, expiration)

Signature

Cryptographic verification of authenticity

Why is it Perfect for web APIs?

- **Stateless:** No server-side session storage needed
- **Self-Contained:** User info and permissions in the token
- **Secure:** Cryptographic signature ensures integrity
- **Scalable:** Excellent for distributed systems

JWT Structure - The Three Parts

A JWT is like a secure, encoded message with three parts, separated by dots (.)

Header

alg: Hashing algorithm
(HS256, RS256)

typ: Always "JWT"

```
{ "alg": "HS256",  
  "typ": "JWT" }
```

Payload

Contains "claims" about the
user

Standard: iss, exp, sub

Custom: userId, roles,
permissions

```
{ "userId": "123",  
  "name": "Alice",  
  "roles": ["user"],  
  "exp": 1678886400 }
```

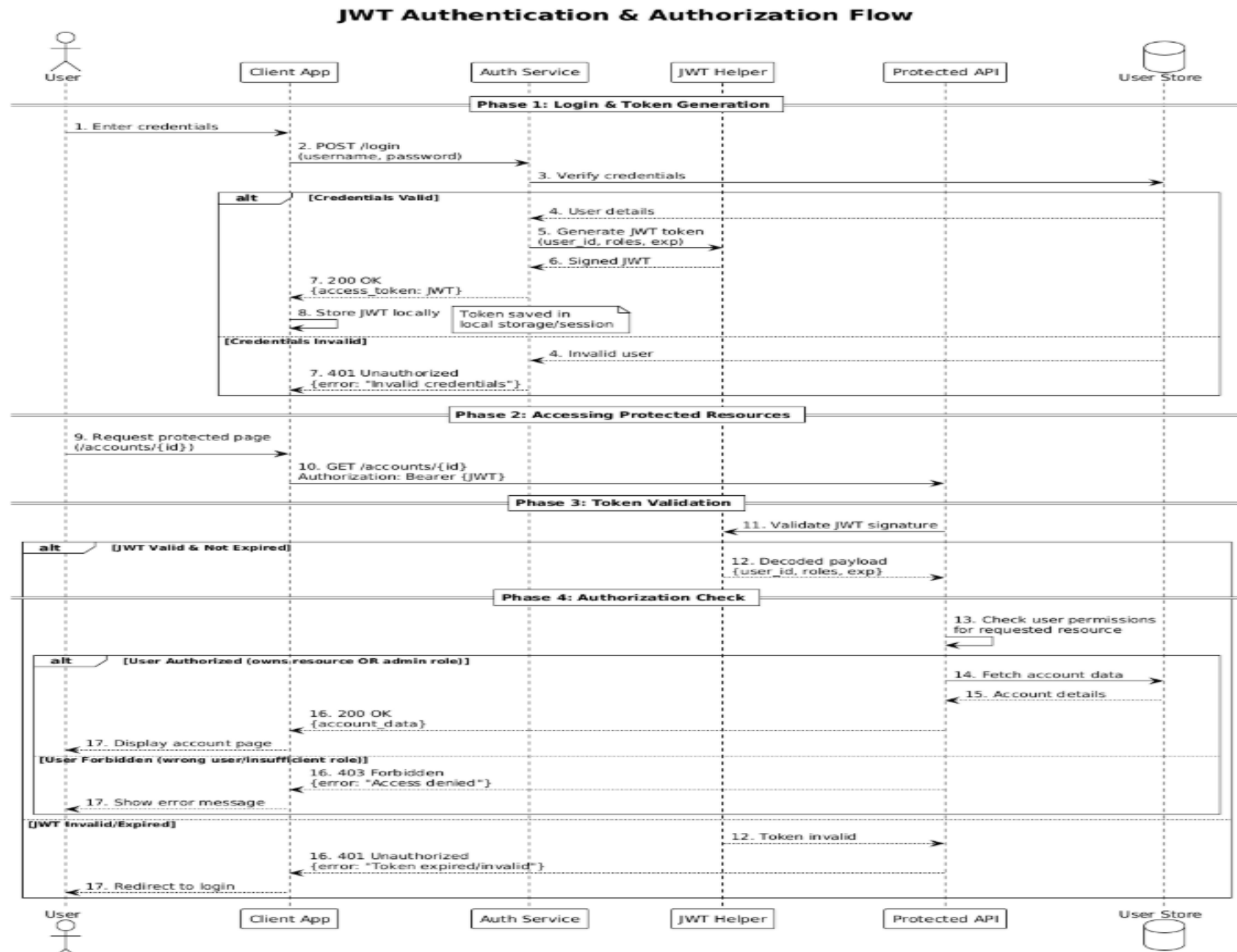
Signature

Created using encoded
Header + Payload + secret
key

Verifies token hasn't been
tampered with

Crucial for security!

JWT Lifecycle: Creation + Secured Endpoint Access



Phase 1: Login & Token Generation

Step 1-2: User Login

User enters credentials → Client App sends POST request to Auth Service

Step 3-4: Verify & Fetch

Auth Service verifies credentials → Fetches user details from User Store

Step 5-6: Generate JWT

JWT Helper creates token with user info → Signs with secret key

Step 7-8: Return Token

200 OK with JWT sent to client → Client stores JWT locally

Invalid Credentials: Auth Service returns 401 Unauthorized → Client shows error

Phase 2: Accessing Protected Resources

Step 9: Request Protected Page

User tries to access a page/resource that requires authentication and authorization

Step 10: GET Request with Authorization

Client App includes stored JWT in HTTP request header:

```
Authorization: Bearer <your_JWT_here>
```

Phase 3: Token Validation

Step 11: Validate JWT Signature

Protected API sends JWT to JWT Helper for signature validation

Step 12: JWT Valid

JWT Helper returns decoded payload (userId, roles, exp)

Step 12 (Invalid): JWT Invalid

JWT Helper determines token is invalid or expired

Step 17-18: Handle Invalid

401 Unauthorized returned → Client redirects to login

Phase 4: Authorization Check

Step 13: Check Permissions

Protected API checks if user's roles/permissions are sufficient for the resource

Step 14-16: Authorized Access

Fetch data → Return data → 200 OK with account_data

Step 17: Display Success

Client App displays the requested information

Insufficient Permissions

403 Forbidden → Client shows error message

Exercise 1: Spot the Security Flaw

A developer adds an account number to JWT payload:

```
{ "sub": "jane.doe", "account_number": "1234567890123456", "role": "CUSTOMER", "exp": 1672531199 }
```

🚨 Question: What's the primary security risk?

💡 Answer:

JWT payload is Base64Url encoded, NOT encrypted! Anyone with the token can decode and read the account number. Never store sensitive data in JWT payload - only non-sensitive claims like user ID and roles.

Exercise 2: Where is the JWT stored after a successful login (on the client-side)?

Solution?

The **Client App** (your frontend application running in the user's browser) needs to store the JWT in a secure location.

- **Local Storage:** A storage space in the browser that saves the data even after the browser window is closed. The Client App's JavaScript can read and write to it.
- **Session Storage:** Similar to Local Storage, but it's temporary. The data is cleared as soon as the user closes the browser tab. The Client App's JavaScript can also access this.
- **HTTP-Only Cookies:** A special type of cookie that is sent with every request to the server, but it **cannot** be accessed by the Client App's JavaScript. This makes it more secure against certain types of attacks (like XSS).

Exercise 3 - If a user tries to access a protected resource with an expired JWT, what HTTP status code would the Protected API likely return, and what would the Client App typically do?

Solution?

HTTP Status Code: 401 Unauthorized (often with a message like "Token expired").

Client App Action: The Client App would typically redirect the user to the login page to obtain a new, valid token.

Exercise 4 - Which component is responsible for actually creating and signing the JWT after a successful login?

Solution?

The **JWT Helper** (orchestrated by the Auth Service) is responsible for generating and signing the JWT.

Exercise 4 - Why is a signed JWT more secure than sending username/password with every request?

Solution –

- Reduced exposure - credentials sent only once,
- Tamper protection - signature prevents payload alteration,
- Efficiency - no database lookup for every request, only signature validation.

Authentication Schemes: Choose the Right Tool

JWT/OAuth2 (Recommended for APIs)

How: User logs in → gets JWT → presents in Authorization header

Perfect for: Mobile and web banking frontends

API Keys

How: Unique string sent with every request

Perfect for: Server-to-server communication

Session Cookies

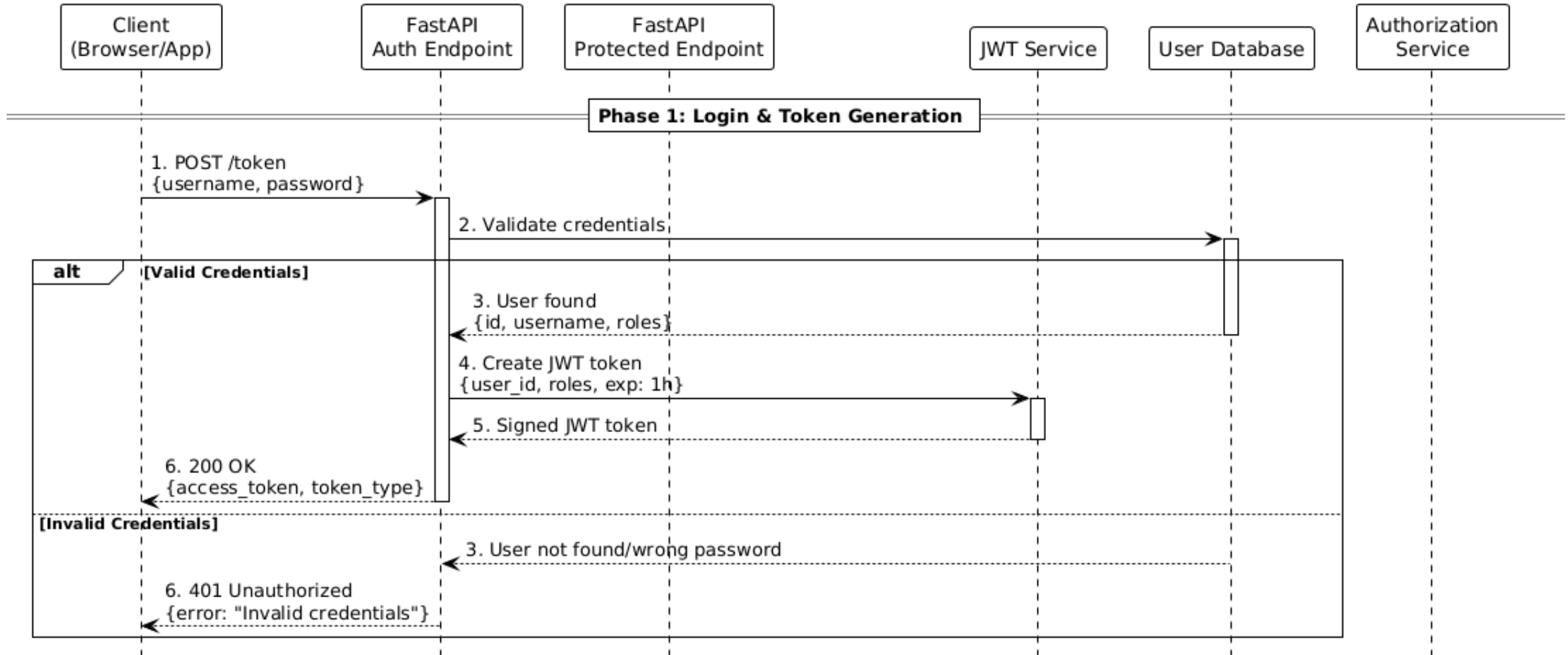
How: Server stores session, sends ID as cookie

Perfect for: Traditional server-rendered web apps

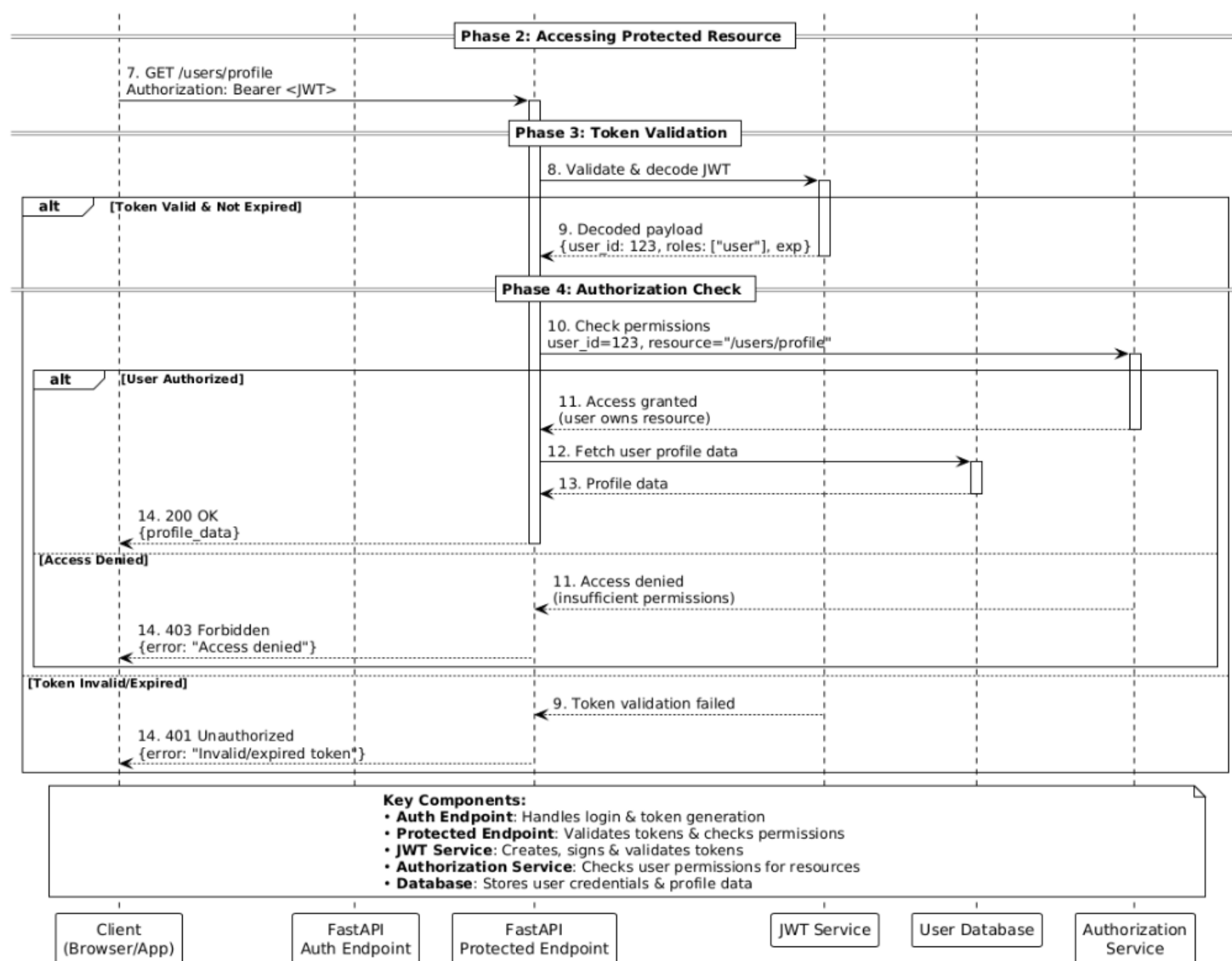
JWT Authentication & Authorisation Flow

FastAPI Implementation - Security Architecture Training

FastAPI JWT Authentication & Authorization Flow



Contd...



JWT Authentication & Authorization Flow

Phase 1: Login & Token Generation

- Client submits credentials to /token endpoint
- FastAPI validates credentials against database
- JWT Service creates signed token with user info & roles
- Token returned to client (expires in 1 hour)

Phase 2: Accessing Protected Resources

- Client includes JWT in Authorization header
- Protected endpoint receives Bearer token
- Request forwarded to token validation service

Phase 3: Token Validation

- JWT Service validates signature & expiration
- Decodes payload: user_id, roles, permissions
- Returns validation result to protected endpoint

Phase 4: Authorization Check

- Authorization Service checks user permissions
- Verifies resource access rights (owns resource?)
- Returns 200 OK with data OR 403 Forbidden

Key Components

Auth Endpoint

Handles login & token generation

Protected Endpoint

Validates tokens & serves resources

JWT Service

Creates, signs & validates tokens

Authorization Service

Checks user permissions for resources

User Database

Stores credentials & profile data

Security

Tokens are signed and expire after 1 hour. No credentials stored on client.

Error Handling

Clear HTTP status codes: 401 (unauthorized), 403 (forbidden), 200 (success).

Scalability

Stateless tokens enable horizontal scaling without session storage.

Best Practices

Role-based access control with proper separation of concerns.

Python Code Examples

Creating JWT Token

```
def create_access_token(data: dict) -> str:
    to_encode = data.copy()
    to_encode.update({"exp": datetime.now(timezone.utc) + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

Securing Endpoints with Dependencies

```
router = APIRouter(prefix="/accounts", tags=["accounts"])
```

[CodeMate](#) | [Qodo Gen](#): Options | [Test this function](#)

```
@router.get("/{account_id}")
```

```
async def get_account(account_id: str, current=Depends(role_required("customer"))):
    return {"account_id": account_id, "owner": current["username"], "role": current["role"]}
```

[CodeMate](#) | [Qodo Gen](#): Options | [Test this function](#)

```
@router.get("/admin/{account_id}")
```

```
async def admin_get_account(account_id: str, current=Depends(role_required("admin"))):
    return {"account_id": account_id, "requested_by": current["username"], "role": current["role"]}
```

You, 6 hours ago • adding project for JWT and Fast API

Code Deep Dive: Creating a JWT Access Token

This function creates a secure token that we can give to a user.

Python Code:

```
def create_access_token(data: dict) -> str:
    to_encode = data.copy()
    to_encode.update({"exp": datetime.now(timezone.utc) + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

Line-by-Line Breakdown:

to_encode = data.copy(): We copy the incoming user data (like username or user_id) to avoid accidentally changing the original dictionary. It's a good safety practice!

to_encode.update({...}): We add a crucial piece of information to the token: the expiration time (exp). This ensures tokens automatically become invalid after a set time.

return jwt.encode(...): This is where the magic happens! Creates a cryptographically signed token using our secret key and algorithm.

Code Deep Dive: Securing Endpoints with Dependencies

This code uses FastAPI to define and protect API endpoints.

Python Code:

```
router = APIRouter(prefix="/accounts", tags=["accounts"])

@router.get("/{account_id}")
async def get_account(account_id: str, current=Depends(role_required("customer"))):
    return {"account_id": account_id, "owner": current["username"], "role": current["role"]}



@router.get("/admin/{account_id}")
async def admin_get_account(account_id: str, current=Depends(role_required("admin"))):
    return {"account_id": account_id, "requested_by": current["username"], "role": current["role"]}
```

What's Happening Here? 🤔

Depends(role_required("customer")): This is the security guard! It performs JWT validation, expiry checks, and role verification before the main function runs.

Dependency Injection: FastAPI automatically runs the security check first. If it fails, the endpoint function never executes.

The Big Picture: A User's Journey

- 1 Login:** User sends username and password to a /login endpoint.
- 2 Token Creation:** The server verifies details and calls `create_access_token()` to generate a JWT.
- 3 Receive Token:** The server sends this JWT back to the user's browser/app.
- 4 Make a Request:** User tries to access `/accounts/123`. The browser includes the JWT in request headers.
- 5 Security Check:** The `Depends(role_required("customer"))` intercepts and validates the JWT.
- 6 Access Decision:**  **Success:** Valid token → Access granted
 **Failure:** Invalid/expired token → Error returned

Quick Understanding Check?

Question 1:

In the `create_access_token` function, what is the purpose of the `SECRET_KEY`? What could an attacker do if they discovered this key?

Solution 1:

The `SECRET_KEY` creates the token's digital signature, ensuring data integrity. If compromised, attackers could create fake tokens with any role (like admin access) and gain full system control. **Keeping this key secret is critical!**

Question 2:

What is the main purpose of `Depends(role_required("customer"))`?

Solution 2:

It acts as a reusable security gatekeeper that handles authentication and authorization before the endpoint function runs. This keeps endpoint code clean and focused on business logic.

Quick Understanding Check?

Question 3:

Can a user with {"role": "customer"} successfully access the URL /accounts/admin/456? Why or why not?

Solution 3:

No, they cannot. The /admin/{account_id} endpoint requires `role_required("admin")`. Since the user's role is "customer", the security check fails and returns a 403 Forbidden error before the function executes.

Key Principle: Each endpoint can have different role requirements, providing fine-grained access control.

Best Practices & Pro-Level Tips

Strong Secret Keys

Long, complex keys stored securely in environment variables

Short Expiration Times

15-60 minutes for access tokens minimizes theft damage

Refresh Token Flow

Long-lived refresh tokens for seamless user experience

Algorithm Specification

Always specify expected algorithm (HS256, RS256)

Secure Storage

HttpOnly cookies for web, secure storage for mobile

No Sensitive Data

NEVER put PII or financial data in JWT payload

Key Takeaways

AuthN vs AuthZ

Identity verification vs Permission checking - you need both!

Security First

Never expose sensitive data in JWT payload

JWT is the Standard

Stateless, secure, self-contained solution for modern APIs

Use Frameworks

Spring Security & FastAPI provide battle-tested tools

Master the Flow

Login → Token → Validation → Authorization

Layered Security

JWT + secure config + validation = robust system

Build Secure, Scale Confidently! 

Mini Project - Authentication & Authorization in Action



Building a Secure Personal Banking API



jwt-fastapi.zip

Audience

Java & Python Developers

Goal

Hands-on JWT Implementation

Duration

45 Minutes

The Core Concepts: A Quick Refresher

Authentication (AuthN)

Who are you?

The process of verifying a user's identity.

Analogy: Showing your ID card to a security guard to enter a building.

In our App: A user provides username and password to prove their identity.

Authorization (AuthZ)

What are you allowed to do?

The process of verifying if a user has permission to access a specific resource.

Analogy: Your keycard only grants access to specific floors or rooms.

In our App: A logged-in user can view their own account but not others'.

Hands-On Lab: Secure API

Your Mission

Build a secure REST API for a mini personal banking system. Implement a login endpoint that issues a JWT and use that token to protect other endpoints.

Core Features to Implement:

- ✓ **User Model:** Simple user with username, password, and role
- ✓ **Login Endpoint:** Authenticates user and returns JWT
- ✓ **Protected Endpoints:** Secure endpoints accessible only with a valid JWT

This exercise covers the entire Authentication and Authorisation flow!

The Auth Flow in Our App

1

Client Login

User sends username & password to POST /api/login

2

Server Validates & Issues JWT

API verifies credentials and generates a signed JWT containing user claims (like role)

3

Client Stores JWT

Client application receives and stores the JWT securely

The Auth Flow in Our App

4

Client Requests Protected Resource

Client makes a call to GET /api/account/details, including JWT in Authorization header

5

Server Validates JWT & Authorizes

API middleware validates JWT signature, expiration, and checks user role permissions. If valid, processes request; otherwise returns error.

FastAPI Security Implementation

Code Walkthrough for FIL Engineering Team

Authentication:

OAuth2 + JWT Tokens

Authorization:

Role-based Access Control

Framework:

FastAPI with Dependency Injection

Security:

Bcrypt Password Hashing

Professional implementation with security best practices

System Architecture Overview

Core Components

- ▶ **FastAPI** - Modern web framework
- ▶ **JWT** - Stateless token authentication
- ▶ **OAuth2** - Industry standard auth flow
- ▶ **Role-based** - Customer/Admin permissions
- ▶ **Dependency Injection** - Clean code architecture

Request Flow

1. Client Login

2. JWT Token Issue

3. Token Validation

4. Role Authorization

5. Resource Access

Clean File Organization

Application Structure

- ▶ `main.py` - FastAPI app & router setup
- ▶ `routers/` - Endpoint organization
- ▶ `deps.py` - Dependency injection logic
- ▶ `security.py` - JWT token management
- ▶ `users.py` - User authentication
- ▶ `models.py` - Pydantic data models

Separation of Concerns

- ▶ Auth logic separated from business logic
- ▶ Reusable security dependencies
- ▶ Modular router organization
- ▶ Environment-based configuration
- ▶ Type-safe models with Pydantic

Authentication Implementation

Login Process

```
@router.post("/token", response_model=Token) async def login(form_data: OAuth2PasswordRequestForm = Depends()):  
    user = authenticate(form_data.username, form_data.password) if not user: raise HTTPException(status_code=401,  
    detail="Invalid credentials") access_token = create_access_token({ "sub": user["username"], "role": user["role"]  
    }) return {"access_token": access_token, "token_type": "bearer"}
```

Security Features: Bcrypt password hashing, JWT expiration, OAuth2 standard compliance

Smart Authorization System

Role-Based Access Control

```
def role_required(required_role: str, allow_admin_on_get: bool = True):  
    async def _checker(request: Request,  
        current=Depends(get_current_user)): if current["role"].lower() == required_role.lower(): return current  
        if (allow_admin_on_get and request.method.upper() == "GET" and current["role"].lower() == "admin"): return current  
        raise HTTPException(status_code=403, detail="Operation not permitted") return _checker
```

- ▶ **Flexible permissions** - Admin can read customer data
- ▶ **Method-aware** - Different rules for GET vs POST
- ▶ **Reusable dependency** - Apply to any endpoint

Production-Ready Security

Implemented Features

- ▶ JWT token expiration
- ▶ Bcrypt password hashing
- ▶ Role-based authorization
- ▶ Token signature validation
- ▶ Secure secret key management

Production Recommendations

- ▶ Always use HTTPS in production
- ▶ Implement refresh token rotation
- ▶ Keep access tokens short-lived
- ▶ Validate additional JWT claims
- ▶ Regular secret key rotation

Key Insight: This implementation provides enterprise-grade security while maintaining code readability and maintainability.

Hands-on Learning Path

Progressive Skill Building

Beginner: Modify token expiration, test role boundaries

Intermediate: Add new roles, implement refresh tokens

Advanced: Database persistence, comprehensive testing

Expert: Threat modeling, security audit checklist

Each exercise builds real-world skills applicable to production systems

Upgrading Our Application

From In-Memory to Persistent & Secure

Goal: Walk through key improvements made to our application

Focus Areas:

- **Adding a Database** (Persistence)
- **Implementing Robust Authentication Flow** (Refresh Tokens)
- **Expanding Role-Based Access Control** (RBAC)
- **Ensuring Quality** with Automated Tests

What Problems Did We Solve?

Problem 1: Data Loss

Our old app kept all user information in memory.

Result: When the server restarted, all users were gone!

Problem 2: Short User Sessions

Access Tokens (JWTs) have a short lifespan (e.g., 15 minutes) for security.

Result: Once expired, users were forced to log in again, which is a bad user experience.

Our Solution:

A more professional, secure, and user-friendly application architecture.

The Foundation: Database & Dependencies

We introduced a lightweight but powerful database layer.

SQLite:

A simple, file-based database. No separate server needed! Great for getting started. The DB is stored in a file like `./data.db`.

SQLModel:

A modern library that combines SQLAlchemy (for DB communication) and Pydantic (for data validation). Makes defining database tables (User, RefreshToken) clean and easy.

Key Dependencies Added (requirements.txt):

- **sqlmodel:** For database models and interaction
- **python-dotenv:** To manage environment variables (like secret keys) in a `.env` file
- **pytest:** To write and run our automated tests

The Core Upgrade: Refresh Tokens

This is the most important security concept we added.

Access Token (The Movie Ticket):

- A short-lived JWT (e.g., expires in 15 minutes)
- Used to access protected API routes
- It's okay if this is stolen, as it becomes useless very quickly

Refresh Token (The Membership Card):

- A long-lived, random string (e.g., expires in 7 days)
- Its only job is to get a new access token
- It is never sent to access regular API routes, only to the `/token/refresh` endpoint

The Refresh Token Flow in Action

1. Login: User sends username and password to POST /token

2. Get Tokens: Server validates credentials and returns both access_token and refresh_token

3. Store Securely: Client stores these tokens. The refresh token's hash is saved in the database

4. Access API: Client uses the access_token to make API calls (e.g., GET /accounts/123)

The Refresh Token Flow in Action

5. Token Expires: After 15 minutes, the `access_token` expires. API returns 401 Unauthorized

6. Get New Token: Client sends its `refresh_token` to POST `/token/refresh`

7. Validation: Server finds the token's hash in DB, checks it's not expired/revoked, and issues a new `access_token`

The client can now continue making API calls. The user never had to log in again!

Storing Tokens: Security First!

We NEVER store the actual refresh token in our database.

Why?

If our database is ever compromised, attackers can't steal the tokens to impersonate users.

What we do: We use a one-way hash (SHA-256)

```
hash = SHA256(original_refresh_token)
```

You can't reverse a hash to get the original token.

Storing Tokens: Security First!

When a user sends a refresh token, we simply hash it and see if the hash exists in our database.

Our RefreshToken database model:

- **user_id:** Which user owns this token
- **token_hash:** The SHA-256 hash of the token
- **expires_at:** When the token is no longer valid
- **revoked:** A boolean flag to manually disable the token

Who Can Do What? Roles & Authorization

We introduced new roles to control access to different parts of the API.

New Roles: **admin, customer, teller, auditor**

Dependency: The `role_required()` function checks a user's role before allowing an action.

Who Can Do What? Roles & Authorization

API Endpoint	Required Role
GET /accounts/{id}	customer
POST /accounts/{id}/deposit	teller
GET /accounts/{id}/audit	auditor

Special Admin Rule:

For GET (read-only) requests: An admin can access routes meant for other roles. This is for easy oversight.

For POST, PUT, DELETE (write) requests: The role must be an exact match. An admin cannot deposit money unless explicitly given the teller role.

Trust, But Verify: Automated Testing

How do we know all this works? We wrote tests!

Framework: We use `pytest` with FastAPI's `TestClient`.

What We Test:

test_login_and_access_customer_route:

Can a customer log in and access their own data?

test_role_boundary_block_customer_from_admin:

Is a customer correctly blocked from an admin-only route?

Trust, But Verify: Automated Testing

test_refresh_token_flow:

Does the full refresh token flow work as expected? Can we use a new access token obtained from a refresh token?

Tests are our safety net. They ensure that future code changes don't break our critical security features.

Tech Stack & Key Libraries



Python Developers (FastAPI)

Framework & Dependencies:

- ✓ FastAPI
- ✓ uvicorn, python-jose, passlib
- ✓ python-jose for JWT handling

Key Concepts:

- ✓ **OAuth2PasswordBearer:** Extract token from header
- ✓ **Dependency Injection:** `get_current_user` function
- ✓ **Route Decorators:** Protect routes with dependencies



The Python Way: FastAPI

FastAPI leverages modern Python features and Dependency Injection. Security is "injected" directly into endpoints that need it.

Framework & Dependencies:

- **FastAPI:** High-performance web framework
- **uvicorn:** Server that runs FastAPI applications
- **python-jose & passlib:** JWT operations and password hashing

Key Concepts:

OAuth2PasswordBearer (Token Extractor 🔍)

Helper class that knows how to find and extract tokens from Authorization: Bearer <token> headers

Dependency Injection (Pre-Flight Check ✅)

Write validation functions (e.g., `get_current_user`) that run before endpoint code. FastAPI guarantees they succeed first

Route Decorators (Endpoint Guards 🛡️)

Standard `@app.get()` decorators. Protect routes by adding dependencies:

```
def get_data(user: User = Depends(get_current_user))
```



Understanding check for JWT?

Question 1:

How does a FastAPI endpoint ensure that a user is authenticated before its main logic is executed?

Solution 1:

Through Dependency Injection. The endpoint declares that it Depends on a function (like `get_current_user`), which contains token validation logic. FastAPI runs this dependency first.

Question 2:

What is the primary purpose of libraries like `python-jose` (Python)?

Solution 2:

Handle JWT operations. They manage the complex and crypto-sensitive operations of creating, signing, and parsing JSON Web Tokens, so we don't write that logic from scratch.


Conclusion & Key Learnings

What We Accomplished

- ✓ Built a functional secure API for banking
- ✓ Implemented complete Auth flow
- ✓ Hands-on JWT implementation
- ✓ Protected sensitive endpoints

Key Learnings

- ✓ **AuthN vs AuthZ:** Login vs Permission checking
- ✓ **JWTs:** Stateless, verifiable user information
- ✓ **Framework Power:** Spring Security & FastAPI
- ✓ **Security Best Practices:** Token validation & role-based access

 Congratulations!

You now have the fundamental skills to secure any modern web API!

One-Way Hash (SHA-256)

Security Improvement

Digital Fingerprints for Enhanced Security

Understanding the "Magic Blender" 🍹 Approach to Data Protection



"You can put ingredients in to make a smoothie, but you can never turn that smoothie back into the original ingredients."

What is a One-Way Hash?

A one-way hash function takes an input and scrambles it into a unique string of fixed length.

For SHA-256: Always produces a **64-character fingerprint**

Three Crucial Properties:

1. One-Way



Impossible to reverse. Can't figure out original password from hash.

2. Deterministic



Same input ALWAYS produces the exact same hash.

3. Unique




Tiny input change creates completely different hash.

Why Do We Use It?

Main Reason: Securely Store Sensitive Data

WITHOUT Hashing

```
User: "aman"  
Password: "MyPassword!@#"
```

If hacker steals database = DISASTER! 

They get every user's actual password

WITH Hashing

```
User: "aman"  
PasswordHash: "a4f2...d8e1"
```


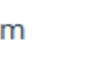


Hacker gets useless hashes! 

Can't reverse to find original passwords

How Do We Log Users In?

We DON'T un-hash the stored value!

Instead, we compare hashes:

- 1 **User enters password:** "MyPassword!@#" 
- 2 **System hashes the input:** Run through SHA-256 algorithm 
- 3 **Compare the two hashes:** Input hash vs Stored hash 
- ✓ **If they match:** Password is correct! User authenticated. 

We verify the user without ever knowing their actual password! 

How Does It Work? ⚡

You don't need to know the complex math. Just understand the **properties in action**:

Example 1: Deterministic Property

```
Input: HelloFIL  
SHA-256 Hash: 0229cb2f68d374a49c402179b8a33501f021e1273955ba4ea1f15158a55953b0
```

Result: Hash "HelloFIL" a million times → Always get this EXACT result! 🎯

How Does It Work? ⚡

Example 2: Avalanche Effect

Let's change just one letter: 'H' → 'h'

Input: HelloFIL

Hash:

0229cb2f68d374a49c402179b8a33501f021e1273955ba4ea1f15158a55953b0

Input: helloFIL

Hash:

9289bb28935c138924b89154a4f10118ac4934d404f3a4792d192f61a70f7e41

Completely different hashes! Even slightly different passwords have unique, unpredictable fingerprints



Key Takeaways

Security Benefits:

- **Password Protection:** Original passwords never stored
- **Data Breach Safety:** Stolen hashes are useless
- **Authentication:** Verify without knowing passwords
- **Integrity:** Detect any data tampering

SHA-256 Properties:

- **One-Way:** Cannot be reversed
- **Deterministic:** Same input = Same output
- **Avalanche Effect:** Small change = Big difference
- **Fixed Length:** Always 64 characters

Key Takeaways

Remember the Magic Blender!

Ingredients go in → Smoothie comes out → Can never get ingredients back

This simple analogy explains the core concept of one-way hashing perfectly!