JWT FLOW:

**Execution Flow:**

1. User **logs in** (/generateToken) → JWT token is generated.
2. User **sends requests with the token** in the Authorization header.
3. **JwtFilter extracts and validates** the token.
4. If valid, **SecurityContextHolder stores authentication**.
5. Requests are then processed based on **roles and authorities**.

**Key Points:**

* JwtUtil manages token creation and validation.
* JwtFilter extracts and authenticates JWT from requests.
* SecurityConfig sets security rules and applies filters.
* UserController manages authentication.

**JWT questions**

**Why do we use a secret key in JWT?**

A **secret key** is used to **sign and verify** the JWT. Think of it like a **signature** on a contract—only the signer (your backend) knows how to create it, and anyone receiving the contract can verify its authenticity.

* When a token is **generated**, it is signed using the secret key.
* When a token is **received**, the backend verifies it using the same key.
* If someone **modifies** the token, the verification will fail because they don’t know the secret key.

**Example**

If your secret key is "MySuperSecretKey", it is used like this:

Key key = Keys.hmacShaKeyFor("MySuperSecretKey".getBytes());

String token = Jwts.builder()

.setSubject("user123")

.signWith(key, SignatureAlgorithm.HS256)

.compact();

* The token is **signed** using "MySuperSecretKey".
* When verifying the token later, it is **checked** using the same secret key.

**What is HS256 (HMAC-SHA256) and why do we use it?**

**HS256 (HMAC-SHA256)** is an algorithm used to **sign** the JWT securely.

* **HMAC (Hash-based Message Authentication Code)** ensures the signature is valid.
* **SHA256 (Secure Hash Algorithm 256-bit)** is a cryptographic function that makes it very difficult for attackers to fake a valid signature.

**Example**

If you send a JWT with the header:

{

"alg": "HS256",

"typ": "JWT"

}

It means that the signature at the end of the token was created using **HS256**.

**Why use HS256?**

* It is **fast** and secure.
* Only your **server** knows the secret key.
* It prevents token tampering.

**Who generates the token?**

The **backend server** generates the token when a user **logs in successfully**.

**How is it generated?**

1. The user sends their **username** and **password** to the backend.
2. The backend **authenticates** the user.
3. If valid, the backend **creates a JWT** and sends it to the user.

**Example Code**

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username) // The token contains the username

.setIssuedAt(new Date()) // When it was created

.setExpiration(new Date(System.currentTimeMillis() + 86400000)) // Expiry in 1 day

.signWith(getSigningKey(), SignatureAlgorithm.HS256) // Sign with secret key

.compact();

}

**Who receives the token?**

* The **frontend (React, Angular, etc.)** stores the token (usually in localStorage or sessionStorage).
* Every time the user makes a request, the frontend **sends the token** in the Authorization header:
* Authorization: Bearer eyJhbGciOiJIUzI1...

**Why is JWT stateless?**

JWT is **stateless** because **the server does not store any session data**.

**How does it work?**

* When a user logs in, the server **does NOT create a session** in memory.
* Instead, it **gives the user a token** that contains their identity.
* The server does not need to remember the user’s login state.
* Every request includes the token, so the server can **validate it without storing anything**.

**Why is this useful?**

✅ **Scalability** – Multiple servers can handle requests because no session is stored.  
✅ **Faster performance** – No need to check the session in a database.  
✅ **Stateless API** – Works well with REST APIs and microservices.

**Why is CSRF disabled?**

CSRF (Cross-Site Request Forgery) is an attack where a hacker tricks a user into sending **unauthorized requests**.

**When is CSRF a problem?**

* If a **website uses sessions or cookies**, the browser **automatically sends authentication data**.
* A hacker can make **requests on behalf of the user** without their consent.

**Why don’t we need CSRF protection in JWT?**

✅ **JWT is stored in localStorage or manually added to headers.**  
✅ **The browser does not automatically send JWTs like cookies.**  
✅ **An attacker cannot generate a valid JWT without knowing the secret key.**

**When should we enable CSRF?**

* If you are using **session-based authentication** (not JWT).
* If you use **cookies** for authentication.

**Example: Disabling CSRF in Spring Security**

http.csrf(csrf -> csrf.disable()); // Disables CSRF protection

* Since we use JWT, we disable CSRF.

**Summary**

| **Concept** | **Explanation** |
| --- | --- |
| **Secret Key** | Used to sign and verify JWTs, preventing tampering. |
| **HS256** | Secure algorithm for signing JWTs. |
| **Who generates the token?** | The backend generates a token when the user logs in. |
| **Stateless Authentication** | No session is stored; the token carries all necessary information. |
| **CSRF Disabled** | Since JWT is manually sent, attackers cannot force authentication. |

Let me know if you need any clarification! 🚀