Spring Security with JWT — Deep Dive Notes

> 📋 **Overview This document explains how Spring Security and JWT work at an internal, mechanical level — not just what to write, but what Spring is actually doing behind the scenes at each step.**

🗺️ The Big Picture — Two Separate Problems

Spring Security + JWT is solving two distinct problems simultaneously:

Problem	Solved By
Who is this user? (Identity)	JWT token + `JwtAuthenticationFilter`
What can they do? (Authorization)	`SecurityFilterChain` rules + `SecurityContext`

These two concerns are cleanly separated and run at different moments in the request lifecycle.

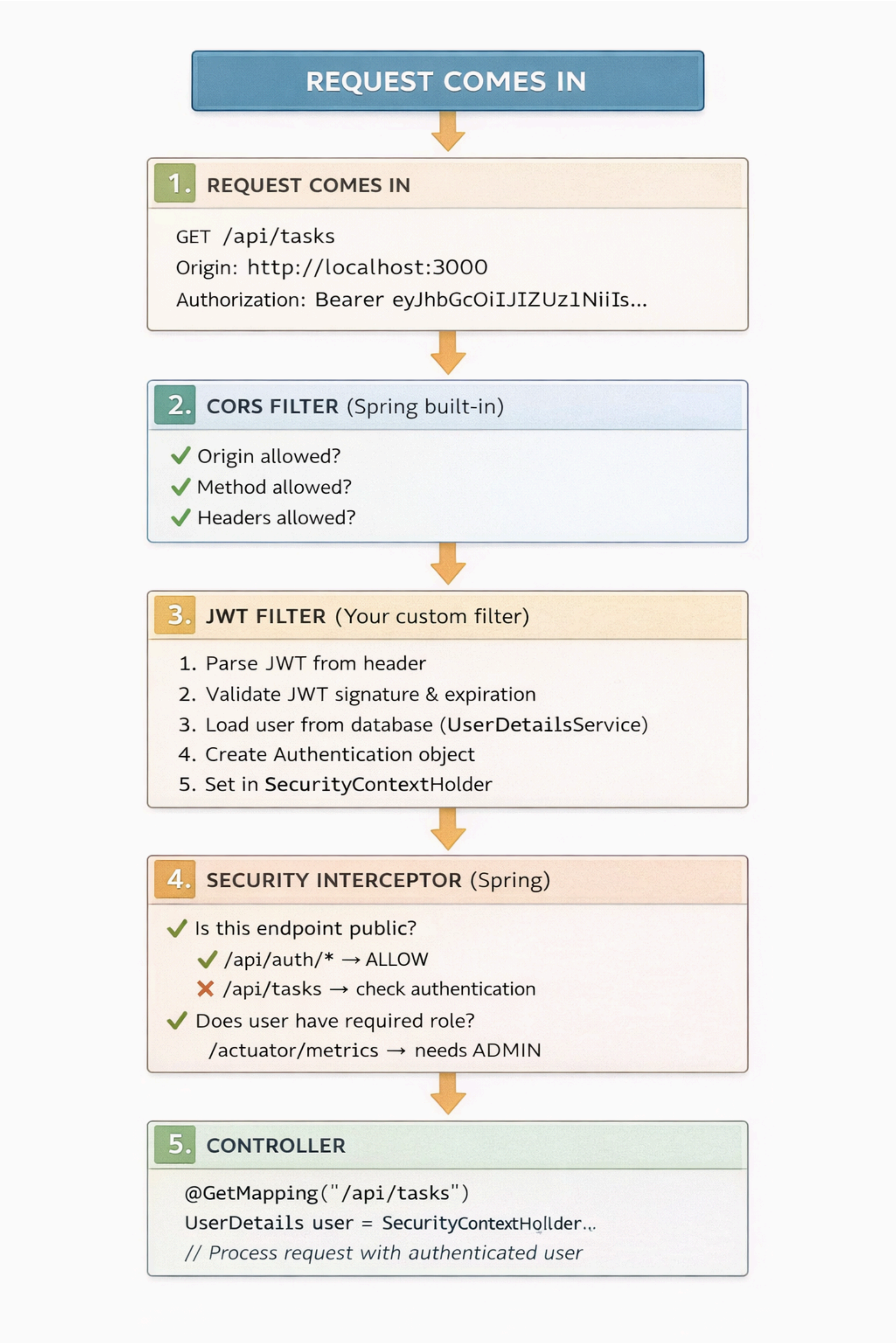
🔗 The Filter Chain — The Core Mechanism

What Is It?

Every HTTP request in a Spring Boot app passes through a **chain of servlet filters** *before* it ever reaches your `@RestController`. Spring Security works by injecting its own filters into this chain.

```
Incoming Request
     ↓
[Filter 1: CorsFilter]
     ↓
[Filter 2: JwtAuthenticationFilter]  ← YOUR custom filter
     ↓
[Filter 3: UsernamePasswordAuthenticationFilter]  ← Spring's default (bypassed with JWT)
     ↓
[Filter N: ...]
     ↓
DispatcherServlet → @RestController
```

> **Key Insight Your `JwtAuthenticationFilter extends OncePerRequestFilter` ensures it runs exactly once per request, regardless of how many times the filter chain is invoked internally (e.g., with forwards/includes). The "once" guarantee is important.**



REQUEST COMES IN

1. REQUEST COMES IN

```
GET /api/tasks
Origin: http://localhost:3000
Authorization: Bearer eyJhbGcOiIJIZUz1NiiIs...
```

2. CORS FILTER (Spring built-in)

- ✓ Origin allowed?
- ✓ Method allowed?
- ✓ Headers allowed?

3. JWT FILTER (Your custom filter)

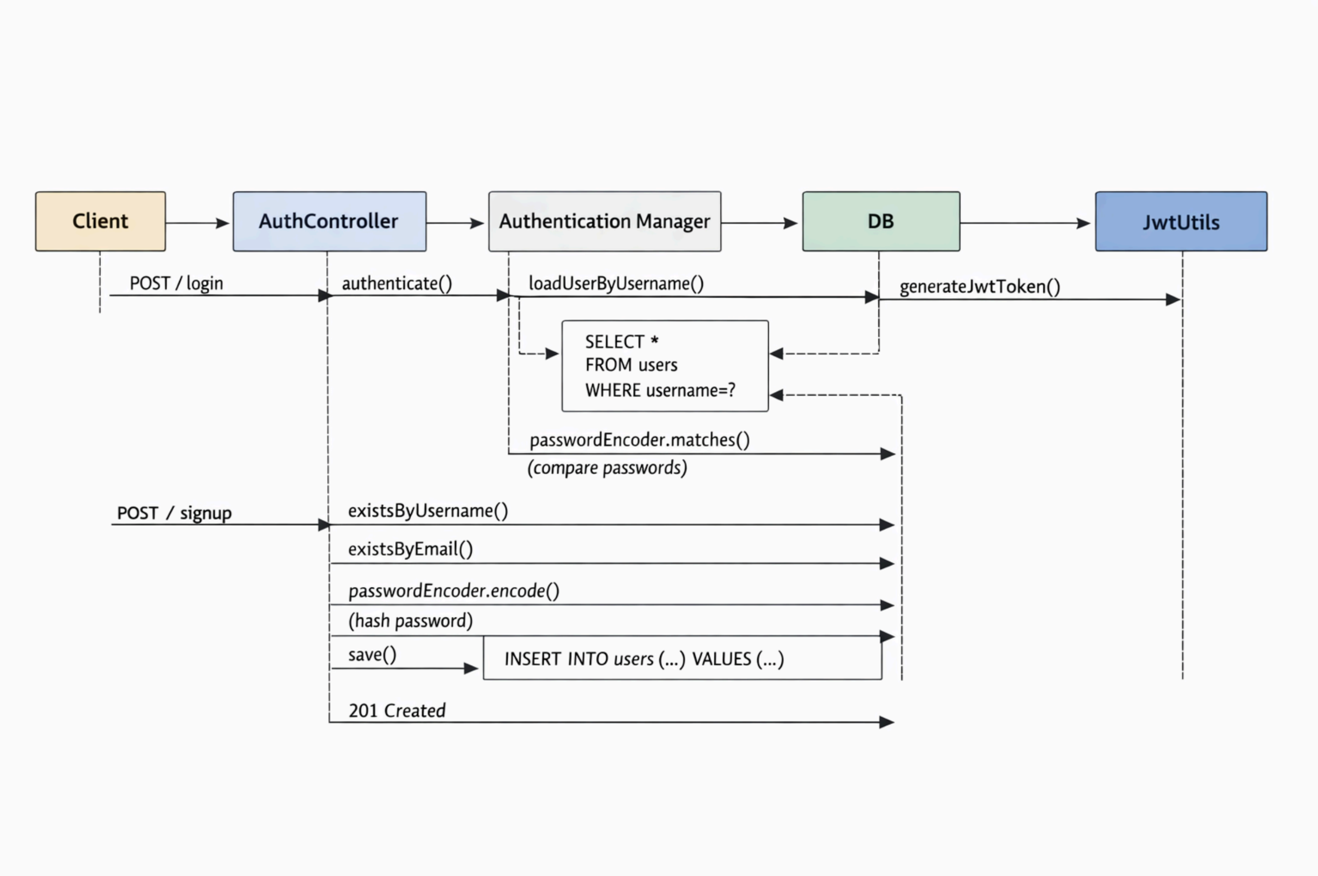
1. Parse JWT from header
2. Validate JWT signature & expiration
3. Load user from database (UserDetailsService)
4. Create Authentication object
5. Set in SecurityContextHolder

4. SECURITY INTERCEPTOR (Spring)

- ✓ Is this endpoint public?
  - ✓ /api/auth/* → ALLOW
  - ✗ /api/tasks → check authentication
- ✓ Does user have required role?
  - /actuator/metrics → needs ADMIN

5. CONTROLLER

```
@GetMapping("/api/tasks")
UserDetails user = SecurityContextHollder...
// Process request with authenticated user
```



What Happens If No Token?

The filter chain **still continues**. The filter doesn't stop the request — it simply *doesn't populate* the `SecurityContext`. Later, when the request reaches the authorization check, Spring finds an empty context and returns `401 Unauthorized`.

This is deliberate: the filter's only job is to **populate identity if a valid token exists**. The *decision* to block or allow is made further down the chain.

🏗️ SecurityContext — Spring's Identity Bus

The Mechanism

`SecurityContextHolder` is essentially a **thread-local variable**. For each incoming request (which runs on its own thread), Spring maintains an isolated `SecurityContext` that holds the currently authenticated `Authentication` object.

```
Thread A (Request 1: user "alice")        Thread B (Request 2: user "bob")
SecurityContextHolder → Authentication    SecurityContextHolder → Authentication
    Principal: "alice"                        Principal: "bob"
    Authorities: [ROLE_USER]                  Authorities: [ROLE_ADMIN]
```

> ⚠ **Why This Matters** Because it's thread-local, there's no shared state between requests. This is what makes JWT stateless — you're not looking up a session in a database or cache. You're reconstructing identity from the token on every single request.

What Gets Put In There?

`JwtAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken` and puts it in the `SecurityContextHolder`. This token has three parts:

- **Principal** — the `UserDetails` object (who they are)
- **Credentials** — `null` (already verified via JWT; we don't need the password again)

- **Authorities** — the `GrantedAuthority` list (what roles they have)

Setting `Credentials` to `null` is an intentional signal: *this authentication is pre-verified, no password check needed here*.

🔑 JWT — What It Actually Is

The Structure at the Byte Level

A JWT is three Base64URL-encoded JSON objects joined by dots:

```
[Base64URL(Header)].[Base64URL(Payload)].[HMAC-SHA256 Signature]
```

Header JSON:

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

Payload JSON (Claims):

```
{ "sub": "john.doe", "iat": 1700000000, "exp": 1700086400 }
```

Signature:

```
HMAC-SHA256(
    base64url(header) + "." + base64url(payload),
    secretKey
)
```

> 🔥 **What Makes JWT "Secure"? The payload is not encrypted — it's only Base64-encoded, meaning anyone can decode and read it. The security comes from the signature. If an attacker modifies even one character of the payload, the signature won't match when the server re-computes it. That mismatch is what `JwtUtils.validateJwtToken()` catches.**

Why Stateless?

Traditional sessions work like a coat check: the server keeps the coat (session data) and gives you a ticket (session ID). On every request, you hand in the ticket and the server looks up your coat.

JWT is different: the server gives you the coat. The token **carries all necessary identity info** (username, roles, expiry). The server doesn't need to "look up" anything in a database — it just verifies the signature and reads the claims.

This is why `SessionCreationPolicy.STATELESS` in your `SecurityConfig` is correct with JWT — there is no session to create or look up.

🔐 The Login Flow — What Spring Does Internally

When `AuthController.authenticateUser()` calls `authenticationManager.authenticate(...)`, a multi-step process begins inside Spring Security:

Step 1: AuthenticationManager Delegates

`AuthenticationManager` is an interface with one method: `authenticate()`. The actual implementation Spring uses is `ProviderManager`, which holds a list of `AuthenticationProvider` objects. It iterates through them, asking each: *"Can you handle this authentication type?"*

Step 2: DaoAuthenticationProvider Takes Over

`DaoAuthenticationProvider` says: *"Yes, I handle `UsernamePasswordAuthenticationToken`."* It:

1. Calls `UserDetailsService.loadUserByUsername(username)` — this is **your** `UserDetailsServiceImpl`, which hits the database via `UserRepository`
2. Gets back a `UserDetails` object with the stored **hashed** password
3. Calls `PasswordEncoder.matches(rawPassword, hashedPassword)` — BCrypt re-hashes the input with the stored salt and compares

Step 3: BCrypt's One-Way Hashing

BCrypt doesn't decrypt the stored hash. It takes the raw input, extracts the salt **from the stored hash** (the salt is embedded in the `$2a$10$...` string), runs the same hashing algorithm, and checks if the result matches. This is why you can't reverse it.

Step 4: Successful Authentication

If the passwords match, `DaoAuthenticationProvider` returns a fully authenticated `Authentication` object (with `isAuthenticated() == true`). This bubbles back up to your `AuthController`, which then calls `JwtUtils.generateJwtToken()`.

🎫 JWT Generation — What the Library Does

When `JwtUtils.generateJwtToken()` is called, the JJWT library:

1. Serializes the header JSON to bytes, then Base64URL-encodes it
2. Serializes your claims (subject = username, issuedAt, expiration) to bytes, then Base64URL-encodes it
3. Computes `HMAC-SHA256(encodedHeader + "." + encodedPayload, secretKey)`
4. Base64URL-encodes that signature
5. Concatenates all three with dots → the final JWT string

The `jwtSecret` from your `application.properties` is the key used to sign. In production this must be:

- At least 256 bits (32 characters) for HS256
- Stored as an environment variable, never in source control

🔍 The Filter — What Happens On Every Request

After login, the client sends the JWT in every request header:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Inside `JwtAuthenticationFilter.doFilterInternal()`:

1. Header Extraction The filter reads the `Authorization` header. If it starts with `"Bearer "`, it slices off the prefix (7 characters) to get the raw token string.

2. Validation `JwtUtils.validateJwtToken()` calls JJWT's parser, which:

- Decodes the header and payload
- Recomputes the HMAC signature using your secret key
- Compares it to the token's signature
- Checks the `exp` (expiration) claim against the current time

If any of these fail, JJWT throws an exception (`ExpiredJwtException`, `SignatureException`, `MalformedJwtException`, etc.). The `try-catch` catches all of them and returns `false`.

3. Username Extraction If valid, the `sub` claim is read directly from the payload — no database call needed here.

4. UserDetails Load `userDetailsService.loadUserByUsername(username)` **does** hit the database here. This is intentional and necessary — you need current role/authority info, and you need to check if the account is still enabled/locked. The token only proves identity; current authorization state lives in the DB.

5. SecurityContext Population A `UsernamePasswordAuthenticationToken` is constructed and placed in the `SecurityContextHolder`. From this point on, any code in the request — your controller, any `@Service`, anything — can call `SecurityContextHolder.getContext().getAuthentication()` and get the current user.

6. filterChain.doFilter() — Always Called This is critical: the filter *always* calls `filterChain.doFilter()` to pass the request to the next filter, regardless of whether authentication succeeded. The filter is **not** the gatekeeper — the authorization layer further down the chain is.

🛡️ Authorization — How Access Rules Are Enforced

After the filter chain completes its work, Spring Security's `FilterSecurityInterceptor` (or in modern Spring, `AuthorizationFilter`) checks the populated `SecurityContext` against the rules you defined in `SecurityFilterChain`:

```
.authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/auth/**").permitAll()
    .requestMatchers("/actuator/**").hasRole("ADMIN")
    .anyRequest().authenticated()
)
```

How Rule Matching Works

Rules are evaluated **top to bottom, first match wins**. Spring checks the request URL against each `requestMatcher` pattern:

- `permitAll()` — the `SecurityContext` is ignored entirely; anyone passes
- `hasRole("ADMIN")` — Spring checks if the `Authentication` has a `GrantedAuthority` equal to `"ROLE_ADMIN"` (Spring automatically prepends `"ROLE_"` when you use `hasRole()`)
- `authenticated()` — Spring checks that `authentication != null && authentication.isAuthenticated() == true`

If the check fails, Spring throws `AccessDeniedException` (→ `403`) or `AuthenticationException` (→ `401`).

👤 UserDetails — The Abstraction Bridge

`UserDetails` is Spring Security's own interface for representing a user. Your `User` JPA entity knows nothing about Spring Security, and Spring Security knows nothing about your database schema. `UserDetailsServiceImpl` is the **adapter** between them.

The returned `UserDetails` object carries:

- `getUsername()` → used as the authentication principal
- `getPassword()` → used by `DaoAuthenticationProvider` for comparison
- `getAuthorities()` → `GrantedAuthority` collection → drives authorization decisions
- `isEnabled()`, `isAccountNonLocked()`, etc. → Spring checks these automatically before authentication succeeds

> ✏️ **Roles vs Authorities A `GrantedAuthority` is just a string. By convention, roles are prefixed with `ROLE_` (e.g., `ROLE_ADMIN`). `hasRole("ADMIN")` in your config checks for `ROLE_ADMIN`. `hasAuthority("ROLE_ADMIN")` also works but is more explicit. The difference: `hasRole` auto-adds the prefix, `hasAuthority` does not.**

🔒 Password Encoding — BCrypt Internals

`BCryptPasswordEncoder` is used because:

1. **It's slow by design** — each hash takes ~100ms at cost factor 10. This makes brute-force attacks expensive.

2. **It includes a salt** — every hash is unique even for the same password. Rainbow tables don't work.
3. **One-way** — you can never recover the original password from the hash.

When you call `passwordEncoder.encode("mypassword")`, BCrypt:

1. Generates a random 128-bit salt
2. Runs the Blowfish cipher iteratively (2^10 = 1024 rounds at strength 10)
3. Returns a 60-character string: `$2a$10$[22 char salt][31 char hash]`

When you call `passwordEncoder.matches("mypassword", "$2a$10$...")`:

1. Extracts the salt from the stored string
2. Runs the same process with that salt
3. Compares the output — **never compares the raw inputs**

🔄 CORS — Why It's Configured in Spring, Not Just Nginx

CORS (Cross-Origin Resource Sharing) is a *browser* security feature. The browser, before sending your `fetch()` call, asks the server: *"Do you allow requests from `http://localhost:3000`?"* The server responds with headers like `Access-Control-Allow-Origin`.

Spring Security's CORS configuration generates these response headers. Without it, even if your Nginx passes the request through, the browser will reject the response.

`config.setAllowedOrigins(List.of("*"))` means any origin is allowed. In production, restrict this to your actual frontend domain.

> **⚠ CORS ≠ Security CORS only restricts browsers. Curl, Postman, and server-to-server calls ignore CORS entirely. JWT authentication is your actual security layer.**

📋 DTO Pattern — Why Not Return JPA Entities Directly

Returning your `User` JPA entity from a controller would:

- Expose the `password` field (even hashed, this is a leak)
- Couple your API contract to your database schema (changing a column name breaks clients)
- Potentially trigger lazy-loading of Hibernate associations, causing `LazyInitializationException` outside a transaction

DTOs (`JwtResponse`, `LoginRequest`, etc.) define an explicit **API contract** that is independent of your internal data model. You control exactly what goes over the wire.

🏃 DataInitializer — CommandLineRunner Hook

`CommandLineRunner` is a Spring Boot interface. Any bean implementing it has its `run()` method called **after the application context is fully loaded, just before the application is ready to serve requests**. This makes it ideal for seeding initial data.

Spring Boot executes all `CommandLineRunner` beans in order (you can use `@Order` to control sequence if you have multiple).

🧵 Thread Safety Considerations

`SecurityContextHolder` uses a `ThreadLocal` by default. This is correct for standard synchronous request handling where one thread processes one request.

If you use `@Async` methods or reactive (WebFlux) programming, the `SecurityContext` does **not** automatically propagate to new threads. You would need `SecurityContextHolder.setStrategyName(SecurityContextHolder.MODE_INHERITABLETHREADLOCAL)` or explicit context propagation.

📊 Summary — Component Responsibility Map

Component	Responsibility	Spring Abstraction Used
`JwtUtils`	Create & validate JWT tokens	JJWT library
`JwtAuthenticationFilter`	Intercept every request, populate `SecurityContext`	`OncePerRequestFilter`
`UserDetailsServiceImpl`	Load user from DB into Spring's format	`UserDetailsService`
`DaoAuthenticationProvider`	Orchestrate login: load user + check password	`AuthenticationProvider`
`BCryptPasswordEncoder`	Hash and verify passwords	`PasswordEncoder`
`SecurityFilterChain`	Define URL access rules	Spring Security DSL
`AuthController`	Expose login/register HTTP endpoints	`@RestController`
`DataInitializer`	Seed initial data on startup	`CommandLineRunner`
DTOs	Define API request/response shapes	Plain Java objects

🔗 Related Topics

- Spring Security Architecture — FilterChainProxy, SecurityFilterChain hierarchy
- JWT RFC 7519 — The specification for JSON Web Tokens
- OAuth2 vs JWT — When to use what
- Spring Boot Auto-configuration — How `@EnableWebSecurity` wires everything
- BCrypt Algorithm — Password hashing deep dive