**Authentication and Authorization**

* For every request, except the login or signup page request,authentication is done.
* Authentication means verifying the credentials.
* The token is generated for the user.

This creates a JWT token containing:

* The username (setSubject)
* The issue date (setIssuedAt)
* The expiration date (setExpiration)
* A signature for security (signWith)

| **Authentication Type** | **Use Case** |
| --- | --- |
| UsernamePasswordAuthenticationToken | Standard username/password login. |
| BearerTokenAuthentication | JWT token-based authentication. |
| OAuth2AuthenticationToken | OAuth2 authentication (Google, Facebook, etc.). |
| AnonymousAuthenticationToken | For unauthenticated users (guests). |
| RememberMeAuthenticationToken | For "Remember Me" login functionality. |

* For every request, the token must be validated and the username must be extracted from the token. Then the username(email) that is unique, is authenticated and the data is fetched from the database,which is running inside the docker container. this is the jwt authentication filter.
* The authentication token is stored in the security context.this is the copy of the current token for reusability.

**Why store authentication in SecurityContext?**

* + Once set, Spring Security knows the user is authenticated.
  + The user does not need to log in again for every request.
  + Subsequent requests can simply check SecurityContext instead of re-validating the JWT.
* After this the normal request and response is carried out using dispatcher servlet.

## **Key Differences: Access Token vs. Refresh Token**

| **Feature** | **Access Token** | **Refresh Token** |
| --- | --- | --- |
| Purpose | Authenticate API requests | Generate a new access token |
| Lifetime | Short (e.g., 5-30 mins) | Long (e.g., 7-30 days) |
| Storage | In-memory / Local storage | Secure storage (Database) |
| Sent in Requests | Yes (Authorization: Bearer <token>) | No (used only when refreshing) |
| Contains User Info | Yes (username, roles, expiration) | Yes (but only for refreshing tokens) |
| Security Risk | High (exposed to client-side apps) | Low (used only when needed) |
| Requires Database Storage?  **Tokens Types** | No | Yes (to track revoked tokens) |

| **Token Type** | **Purpose** | **Lifespan** | **Storage** | **Example Use Case** |
| --- | --- | --- | --- | --- |
| **Access Token** | API authentication | Short-lived (5-30 min) | Client-side (Memory/Local Storage) | Used in JWT-based authentication |
| **Refresh Token** | Generates new access token(API) | Long-lived (7-30 days) | Secure storage (DB/HTTP-only cookie) | Allows session continuation |
| **ID Token** | Stores user profile info | Same as access token | Client-side | Used in OAuth & OpenID Connect |
| **Session Token** | Maintains user session | Session-based | Server-side | Traditional login systems |
| **CSRF Token** | Prevents CSRF attacks | Per session/request | HTTP-only cookie | Secure web form submissions |
| **OAuth Token** | Authorization in OAuth 2.0 | Short-lived or long-lived | Third-party apps | Logging in with Google or Facebook |
| **API Key** | Identifies applications | Long-lived (until revoked) | Server-side | External API authentication |

**User and Token Entity**

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "user\_id")

public User user;

@OneToMany(mappedBy = "user")

private List<Token> tokens;

**@RequiredArgsConstructor**

* This **Lombok annotation** generates a constructor that **injects all final fields** automatically.
* It helps in **constructor-based dependency injection**.

**Application Configuration**

| Component | Purpose |
| --- | --- |
| @Configuration | Defines a Spring configuration class. |
| @RequiredArgsConstructor | Injects dependencies using Lombok. |
| UserDetailsService | Fetches user details from the database for authentication. |
| AuthenticationProvider | Validates user credentials and loads user details. |
| AuthenticationManager | Manages authentication using AuthenticationProvider. |
| PasswordEncoder | Hashes passwords using BCryptPasswordEncoder. |
| AuditorAware | Automatically tracks user actions in JPA entities.(insert or updated by which userid) |

**Authentication Controller**

### **User Registration (POST /register)**

1. Receive user details.
2. Hash the password.
3. Save user in the database.
4. Generate JWT access & refresh tokens.
5. Store token & return response.

### **User Login (POST /authenticate)**

1. Authenticate user credentials.
2. Generate new JWT tokens.
3. Revoke old tokens.
4. Store new token & return response.

### **Refresh Token (POST /refresh-token)**

1. Extract refresh token.
2. Validate & verify token.
3. Generate new access token.
4. Revoke old tokens.
5. Store new token & return response.

**Working**

Each API call verifies the Access Token.

If the Access Token is valid, the request is processed.

If the Access Token is expired, the Refresh Token is used to get a new Access Token.

The refresh token is stored for each login user.it is terminated when the user is loggedout.

### **Reasons for Generating Both Tokens During Registration**

* When a user registers, they should be able to log in immediately without needing another authentication request.
* To provide a seamless experience, we issue both tokens so they can start interacting with the system right away.
* This reduces API calls since the refresh token can be used later to get new access tokens.

### **Reasons for Generating a New Token for an Authenticated User**

1. **Security**
   * If an attacker somehow steals an old access token, it will no longer be valid.
   * We revoke all previously issued tokens to prevent unauthorized access.
2. **Token Expiry Handling**
   * The old token may have expired, so we issue a fresh one.
   * This ensures the user always has a valid access token.
3. **Ensuring One Active Session at a Time**
   * If a user logs in from another device, their old tokens will be revoked.
   * This prevents multiple concurrent sessions, improving security.

**Security Configuration**

They give or define the url permissions either it is public(white url) or accessed by admin.

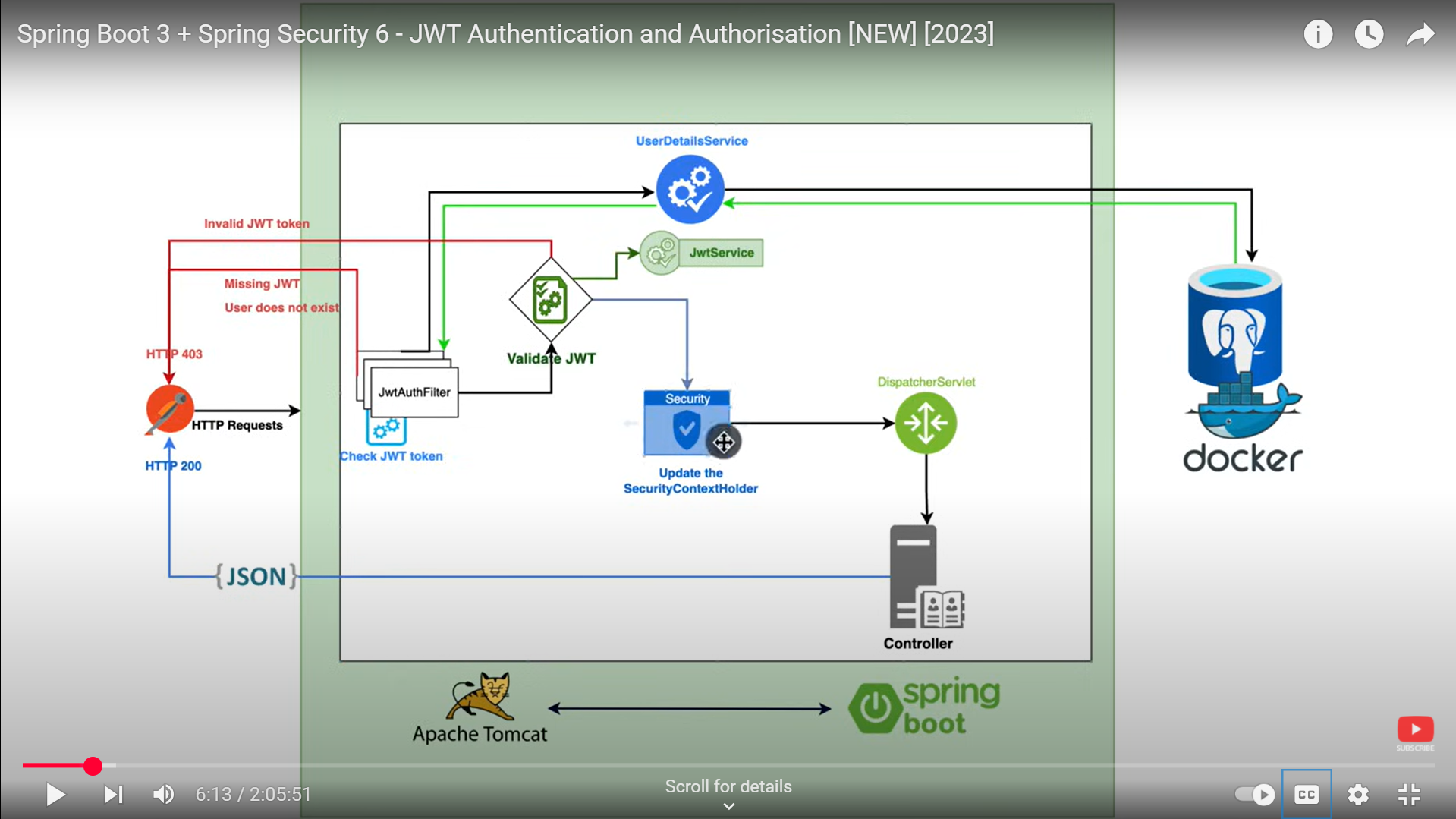
It contains the authentication manager that manges the authentication throught te authentication provider

**@AuthenticationPrincipal**

@AuthenticationPrincipal is a Spring Security annotation used in **Spring Boot applications** to retrieve the **currently authenticated user’s details** from the **SecurityContext**.

**Note**

Authentication is the validation of the credentials and the authorisation is the permission to use the functionalities based on the role(user or admin.



Reference:

<https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbU92RFJyRnllMTlXWmpuSUVJSTZnVTZfY1RBUXxBQ3Jtc0tscWZ3eU90azNCTkRocTdBekpqQy1Ba3ViTy1wUTlNOFRrdzJJcW1jU2N4WFJQZ3NyLU8tZkdQU2lQb1B5UXhTdW5ORmxBZ1hXR1o5aVo4aldPMG5WZHdodncxb29GZ29IdGNRSWJTckVQWFBhemNFaw&q=https%3A%2F%2Fgithub.com%2Fali-bouali%2Fspring-boot-3-jwt-security&v=KxqlJblhzfI>

**Execution or Control Flow**

**Running the Spring Boot Application**

* Loads Spring Boot components (via @SpringBootApplication)
* Scans and registers beans (using @ComponentScan)
* Loads Spring Security Configuration (SecurityConfig.class)
* Loads the database configuration (application.yml or application.properties)
* Starts the embedded Tomcat server

**Dependency Injection & Bean Initialization**

| **Bean Name** | **Purpose** |
| --- | --- |
| **AuthenticationManager** | **Manages authentication process** |
| **DaoAuthenticationProvider** | **Validates username & password** |
| **UserDetailsService** | **Fetches user from DB** |
| **PasswordEncoder** | **Hashes passwords using BCrypt** |
| **JwtUtil** | **Handles JWT creation/validation** |
| **JwtAuthenticationFilter** | **Intercepts requests to check JWT** |

**Security Configuration Loads**

* Disables CSRF-Cross-Site Request Forgery

**CSRF attacks require:**

* A logged-in session stored in a browser cookie.
* An attacker’s request using the victim’s session.

**But in JWT authentication:**

* No session → Only token-based authentication.
* Attacker cannot access the JWT token from the victim's browser.

* Configures API Access Rules
* Registers JWT Authentication Filter
* Loads AuthenticationProvider

**Steps**

1️⃣ Start Spring Boot → Beans & Security Configuration Load

2️⃣ User Sends Login Request → AuthenticationManager Authenticates

3️⃣ AuthenticationProvider Verifies User → Generates JWT(json web Token)

4️⃣ User Sends Request with JWT

5️⃣ JwtAuthenticationFilter Intercepts → Validates Token

6️⃣ SecurityContextHolder Stores User Authentication

7️⃣ Role-Based Authorization Checks if Access is Allowed

8️⃣ If Authorized → Controller Processes Request

9️⃣ Response is Sent to Client

🔟 Single Session Handling Ensures One Login Per User using redis

**Redis**

Single-session login can be enforced using Redis by storing active JWT tokens.

🔹 Super Fast: Data is stored in RAM, so it is faster than databases like MySQL.

🔹 Used for Caching: Reduces database load by storing frequently accessed data.

🔹 Session Management: Helps implement single-session login by tracking active users.

🔹 Supports Key-Value Data Storage: Like a Map<String, Object>.

🔹 Persistent Option: Can save data to disk to avoid loss on restart.

Redis is used to **manage user login sessions** by keeping track of session data (such as authentication tokens, user preferences, and temporary data) **in memory** instead of storing it in a database.