JWT Authentication

# Authentication Microservice

**Dependencies**

Add these dependencies with spring security

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.5</version>  
</dependency>

Core Library

Contains main interfaces and classes for JWT processing

Provides methods for creating, parsing, and validating JWTs

Required at compile time

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.5</version>  
 <scope>runtime</scope>  
</dependency>

Contains the actual implementation of the JWT APIs

Runtime only dependency since implementation details aren't needed at compile time

Handles cryptographic operations and token processing

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.11.5</version>  
 <scope>runtime</scope>  
</dependency>

Provides JSON processing capabilities using Jackson

Converts JWT claims to/from JSON format

Runtime only dependency as it's an implementation detail

When a dependency has <scope>runtime</scope>:

It's not needed during compilation (writing code)

It's only needed when actually running the application

Your IDE won't show it in code completion

Think of it this way:

jjwt-api: Contains interfaces like Claims, JwtParser that you use in your code

jjwt-impl: Contains classes that actually implement those interfaces

jjwt-jackson: Contains classes that handle JSON conversion

Jwts.parser() // This comes from jjwt-api  
 .setSigningKey(key)  
 .parseClaimsJws(token);

When you compile this code, you only need to know the method signatures (jjwt-api). When you run it, the actual implementation (jjwt-impl) does the work.

It's like a car interface:

API: "A car should have a drive() method"

Implementation: "Here's how the engine actually works"

You only need to know about drive(), not the engine details, when writing code

This separation:

Reduces compile-time dependencies

Makes the code more maintainable

Allows switching implementations without changing your code

### Main Components:

@Service  
public class AuthFacadeService {  
 // Facade pattern: provides simplified interface for auth operations  
 // Handles:  
 // - User registration  
 // - User login  
 // - Error handling  
 // - Logging  
}  
  
@Service  
public class AuthService {  
 // Core authentication logic:  
 // - User registration business logic  
 // - User authentication verification  
 // - Password handling  
}  
  
public class JwtUtil {  
 // JWT token operations:  
 // - Token creation  
 // - Token validation  
 // - Token parsing  
}

**Security Configuration:**

@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
 // Configures security rules:  
 // - URL patterns to protect  
 // - Authentication manager  
 // - Password encoder  
 // - CORS settings  
 // - JWT filter registration  
   
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) {  
 return http  
 .csrf().disable()  
 .authorizeHttpRequests()  
 .requestMatchers("/auth/\*\*").permitAll()  
 .anyRequest().authenticated()  
 .and()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.STATELESS)  
 .and()  
 .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)  
 .build();  
 }  
}

**Authentication Filter:**

@Component  
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
 private final JwtUtil jwtUtil;  
   
 @Override  
 protected void doFilterInternal(  
 HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain filterChain) {  
 // 1. Extract JWT from Authorization header  
 // 2. Validate token  
 // 3. Set authentication in SecurityContext  
 // 4. Continue filter chain  
 }  
}

**Data Transfer:**

public class UserDTO {  
 // Data Transfer Object  
 // Contains:  
 // - username  
 // - password  
 // Used for transferring data between layers  
}

**Model:**

public class User {  
 // Entity class  
 // Represents user in database  
 // Contains user details like:  
 // - username  
 // - password (hashed)  
 // - other user information  
}

**UserServiceDetails(Optional only if doing automatic authentication):**

UserDetailsService` It is a Spring Security interface used to load user-specific data during authentication. Its main job is to fetch user details (like username, password, roles) from a data source (e.g., database) when given a username. Filters (like your `JwtFilter`) use `UserDetailsService` to retrieve user information for authentication and authorization.

**(no `UserDetailsService`, manual authentication), here's the authentication flow:**

**\*\*1. User Registration/Login (Manual)\*\***

- User sends credentials to `/auth/register` or `/auth/login`

- `AuthService` manually validates credentials against database

- `AuthService` uses `PasswordEncoder` to hash/verify passwords

- If valid, `AuthService` generates JWT token using `JwtUtil`

- JWT token returned to client

**\*\*2. Subsequent Requests (JWT Validation)\*\***

- Client includes JWT in `Authorization: Bearer <token>` header

- `JwtFilter` intercepts all requests

- `JwtFilter` extracts and validates JWT using `JwtUtil`

- If JWT is valid, `JwtFilter` creates a simple `UsernamePasswordAuthenticationToken` with:

- Principal: username from JWT

- Credentials: null

- Authorities: empty list

- Sets authentication in `SecurityContext`

- Request proceeds to controller

**\*\*3. Authorization\*\***

- Spring Security checks if user is authenticated (non-null authentication in context)

- Since you have `.anyRequest().authenticated()`, any authenticated user can access protected endpoints

- No role-based authorization (authorities list is empty)

**## Key Points**

- \*\*No database lookup per request\*\* - JWT contains all needed info

- \*\*Stateless\*\* - no session storage

- \*\*Simple authorization\*\* - just authenticated/unauthenticated

- \*\*Manual credential validation\*\* - your `AuthService` handles login logic

- \*\*JWT-based\*\* - token carries authentication state

This is a lightweight approach suitable when you don't need complex role-based access control and want to avoid database hits on every request.

**Flow of Operations (Registration):**

1. Controller receives UserDTO  
2. AuthFacadeService.registerUser called  
3. AuthService.register processes registration  
4. User saved to database  
5. Response returned with created user

**Flow of Operations (Login):**

1. Controller receives UserDTO  
2. AuthFacadeService.loginUser called  
3. AuthService.authenticate verifies credentials  
4. JwtUtil creates JWT token  
5. Response returned with token

**Key Features:**

Error handling with try-catch

Structured logging with @Slf4j

JWT token-based authentication

Facade pattern for simplified interface

Constructor injection for dependencies

Response wrapping with ResponseEntity

**Complete Authentication Flow:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**A screen shot of a computer

AI-generated content may be incorrect.**

**Protected api call**

A screenshot of a computer

AI-generated content may be incorrect.

**What is servlet Container?**

A **Servlet Container** is the part of a web server (like **Tomcat**, Jetty, or Undertow) that is responsible for:

* Receiving HTTP requests from clients (browsers, Postman, etc.)
* Creating and managing **Servlet** instances (like the DispatcherServlet in Spring)
* Managing **Filters**, **Listeners**, and the **Servlet lifecycle**
* Sending HTTP responses back to the client

Here’s what happens under the hood:

* **Tomcat** starts and opens a port (usually 8080).
* It creates a **Servlet context** — a runtime environment for all servlets and filters.
* Your Spring Boot app registers:
  + **Filters** (like Spring Security filters)
  + **Servlets** (like DispatcherServlet)
* When a request comes in:
  + Tomcat passes it to the **Filter chain** (each filter can modify, block, or forward the request).
  + After filters, Tomcat calls the **target Servlet** (in Spring, that’s the DispatcherServlet).
  + The servlet handles the business logic (via controllers) and produces a response.
  + Tomcat sends that response back to the client.

# Product Microservice

**Steps:**

In the Product Microservice, add a JWT filter similar to the current setup.

Configure it to validate tokens using the public key or secret from the Authentication Microservice.

On each request to a protected endpoint, the Product Microservice checks the JWT in the Authorization header.

@RestController  
@RequestMapping("/products")  
public class ProductController {  
  
 @GetMapping  
 public List<Product> getAllProducts() {  
 // Fetch products from database  
 }  
  
 @PostMapping  
 public Product addProduct(@RequestBody Product product) {  
 // Save product to database  
 }  
}

**Security Configuration (Product Microservice):**

Use a similar SecurityConfig and JwtFilter as in your authentication service.

Permit only authenticated requests to /products/\*\*.

Database Interaction:

Use JPA or JDBC to interact with the products table.

**Flow:**

User registers/logins via Authentication Microservice, receives JWT.

User sends requests to Product Microservice with JWT in header.

Product Microservice validates JWT, then allows access to endpoints.

**JwtAuthenticationFilter**:

Intercepts every request

Extracts JWT from "Bearer" token

Validates token using JwtUtil

Sets authentication in SecurityContext

**Other Filters in the Filter chain:**

No, the `FilterChain` is not just the JWT filter. The `FilterChain` is a series of Spring Security filters that process HTTP requests in a specific order. Think of it as a pipeline of security checks.

Here's how the filter chain works:

1. When a request comes in, it goes through multiple filters in sequence

2. The `JwtAuthenticationFilter` is just one of these filters

3. The typical Spring Security filter chain includes:

Request →  
1. CorsFilter  
2. CsrfFilter (disabled in your config)  
3. JwtAuthenticationFilter (your custom filter)  
4. UsernamePasswordAuthenticationFilter  
5. SecurityContextHolderFilter  
→ Controller

JwtAuthenticationFilter:  
- Extracts JWT from request header  
- Validates token  
- Sets authentication in SecurityContext

CorsFilter:  
- Handles Cross-Origin Resource Sharing  
- Adds CORS headers to responses  
  
CsrfFilter (disabled in your config):  
- Prevents Cross-Site Request Forgery attacks  
- Validates CSRF tokens  
  
UsernamePasswordAuthenticationFilter:  
- Processes form-based authentication  
- Handles username/password login attempts  
  
SecurityContextHolderFilter:  
- Maintains the SecurityContext  
- Stores authentication between requests  
  
ExceptionTranslationFilter:  
- Handles security exceptions  
- Uses your configured authenticationEntryPoint

**In your SecurityConfig, you have:**

.csrf(AbstractHttpConfigurer::disable) // Disables CSRF filter  
.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // No session management  
.authorizeHttpRequests(...) // URL-based security rules  
.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class) // Places JWT filter early  
.exceptionHandling(...) // Custom 403 response

In your code, `filterChain.doFilter(request, response)` passes the request to the next filter in the chain. That's why it's important to:

if (authHeader == null || !authHeader.startsWith("Bearer ")) {  
 response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);  
 return; // Stops the chain if no valid token  
}  
// ...  
filterChain.doFilter(request, response); // Continues to next filter if token is valid

This is configured in your `SecurityConfig` using:

.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)

which places your JWT filter before the username/password authentication filter in the chain.

The `**UsernamePasswordAuthenticationFilter**` is specifically designed to handle form-based authentication. Here's what it does:

1. \*\*Intercepts POST Requests\*\*:

- By default, listens for POST requests to `/login`

- Expects form parameters `username` and `password`

2. \*\*Authentication Process\*\*:

```java

// Internal flow of UsernamePasswordAuthenticationFilter

- Extracts username and password from request

- Creates UsernamePasswordAuthenticationToken

- Passes token to AuthenticationManager

- AuthenticationManager validates credentials

```

3. \*\*Success/Failure Handling\*\*:

- On success: Sets authenticated user in SecurityContext

- On failure: Triggers authentication failure handlers

However, in your application:

1. You're not using this filter directly since you have JWT authentication

2. You're using it as a reference point to position your JWT filter:

```java

.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)

```

This means:

- Your JWT filter runs before form login would happen

- Form-based login is effectively bypassed

- All authentication is handled by your JWT mechanism instead

Your security flow is:

```plaintext

Request → JwtAuthenticationFilter → (UsernamePasswordAuthenticationFilter skipped) → Protected Resource

**why here we are using UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(username, null, Collections.emptyList()) in JWTFilter ?**

In this case, UsernamePasswordAuthenticationToken is being used as a general authentication token container, not for username/password authentication. Here's why:

Token Purpose:

// This creates an "authenticated" token with:  
// - Principal (username)  
// - Credentials (null since we don't need password)  
// - Authorities (empty list for now)  
UsernamePasswordAuthenticationToken authentication =   
 new UsernamePasswordAuthenticationToken(username, null, Collections.emptyList());

Context:

You've already validated the JWT token

You don't need the password anymore

You're just creating a token to store the authenticated user's info

Spring Security Usage:

UsernamePasswordAuthenticationToken implements Authentication interface

It's a convenient container for authenticated user details

Spring Security uses it to track who is logged in

**Importance of init method**

The `@PostConstruct` annotation in this case ensures that the `init()` method is called automatically after:

1. The bean (`JwtUtil`) is constructed

2. Dependencies are injected (like the `@Value("${jwt.secret}")` field)

3. Before the bean is put into service

Here's what happens in sequence:

1. Spring creates the `JwtUtil` bean

2. Spring injects the `secretString` value from properties

3. `@PostConstruct` triggers the `init()` method

4. The `init()` method creates the HMAC-SHA key from the secret string

This is important because:

- The `key` field needs the `secretString` to be already injected

- The `key` must be initialized before any JWT operations can happen

- Other methods like `validateToken` and `getUsername` depend on the `key` being properly initialized

Without `@PostConstruct`, you might get a `NullPointerException` if methods try to use the `key` before it's initialized.

**Roles vs Authentication Source:**

The JWT filter's job is to extract and provide the user's identity AND authorities from the token

SecurityConfig defines how these roles are used for authorization, not what they are

The roles/authorities should come from the JWT token itself, as it contains the user's claims

**Current Setup:**

// In JwtAuthenticationFilter  
UsernamePasswordAuthenticationToken authentication =  
 new UsernamePasswordAuthenticationToken(username, null, Collections.emptyList());

**SecurityConfig Role Example:**

.authorizeHttpRequests(auth -> auth  
 .requestMatchers("/admin/\*\*").hasRole("ADMIN") // This checks for roles  
 .requestMatchers("/products/\*\*").authenticated() // This only checks if user is authenticated  
)

The SecurityConfig defines what roles are required for which endpoints, but the actual roles must come from the authentication token created in the filter.

In your case, since you're only using .authenticated() checks, you don't need to set any roles in the filter. Your current implementation with Collections.emptyList() is correct.

**Key Points:**

If your SecurityConfig only uses .authenticated(), then Collections.emptyList() is correct

If SecurityConfig had .hasRole("ADMIN"), then the filter would need to include that role in the Authentication object

Your current implementation is correct because:

You're only checking for authentication, not specific roles

The empty authorities list is sufficient

The JWT filter is doing exactly what it needs to do