Here's a **detailed step-by-step configuration** to secure your **Spring Cloud API Gateway** with **Spring Security and JWT**, along with an **Authorization Service** that validates users from the database.

# **Architecture Overview**

1. **API Gateway (Spring Cloud Gateway)**
   * Routes requests to different microservices.
   * Checks JWT in incoming requests.
   * Rejects unauthorized requests.
2. **Authorization Service (Spring Boot + Spring Security + JWT)**
   * Authenticates users and issues JWT tokens.
   * Stores user details in **PostgreSQL/MySQL**.
   * Verifies token validity.
3. **Other Microservices**
   * Accept JWT-secured requests from the Gateway.
   * Extract user info from the JWT.

# **Step 1: Set Up API Gateway**

### **1.1 Add Dependencies**

**<dependency>**

**<groupId>**org.springframework.cloud**</groupId>**

**<artifactId>**spring-cloud-starter-gateway**</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-security**</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>**io.jsonwebtoken**</groupId>**

**<artifactId>**jjwt**</artifactId>**

**<version>**0.11.5**</version>**

**</dependency>**

### **1.2 Configure Routes in application.yml**

spring:

cloud:

gateway:

routes:

- id: first-microservice

uri: lb://SCMS-FIRST-MICROSERVICE

predicates:

- Path=/SCMS-first-microservice/\*\*

- id: second-microservice

uri: lb://SCMS-SECOND-MICROSERVICE

predicates:

- Path=/SCMS-second-microservice/\*\*

server:

port: 8080

* lb://SCMS-FIRST-MICROSERVICE assumes **service discovery (Eureka)** is used.
* Adjust if using static URLs.

### **1.3 Implement JWT Authentication in Gateway**

#### **JwtUtil.java (JWT Helper)**

import io.jsonwebtoken.\*;

import io.jsonwebtoken.security.Keys;

import org.springframework.stereotype.Component;

import java.security.Key;

import java.util.Date;

**@Component**

**public class JwtUtil {**

private static final String SECRET\_KEY = "your-256-bit-secret-your-256-bit-secret";

private static final long EXPIRATION\_TIME = 86400000; // 1 day

private final Key key = Keys.hmacShaKeyFor(SECRET\_KEY.getBytes());

**public String generateToken(String username) {**

**return** Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))

.signWith(key, SignatureAlgorithm.HS256)

.compact();

**}**

**public boolean validateToken(String token) {**

try {

Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token);

return true;

} catch (JwtException e) {

return false;

}

**}**

**public String extractUsername(String token) {**

return Jwts.parserBuilder()

.setSigningKey(key)

.build()

.parseClaimsJws(token)

.getBody()

.getSubject();

**}**

**}**

#### **JwtAuthenticationFilter.java (JWT Filter in API Gateway)**

import org.springframework.http.HttpHeaders;

import org.springframework.http.server.reactive.ServerHttpRequest;

import org.springframework.http.server.reactive.ServerHttpResponse;

import org.springframework.stereotype.Component;

import org.springframework.web.server.WebFilter;

import org.springframework.web.server.WebFilterChain;

import reactor.core.publisher.Mono;

**@Component**

**public class JwtAuthenticationFilter implements WebFilter {**

private final JwtUtil jwtUtil;

**public JwtAuthenticationFilter(JwtUtil jwtUtil) {**

**this.jwtUtil = jwtUtil;**

**}**

**@Override**

**public Mono<Void> filter(ServerHttpRequest request, WebFilterChain chain) {**

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

if (!jwtUtil.validateToken(token)) {

ServerHttpResponse response = request.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

}

} else {

ServerHttpResponse response = request.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

}

**return** chain.filter(request);

**}**

**}**

### **1.4 Apply Security in ScmsAPIGatewayConfig.java**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;

import org.springframework.security.config.web.server.ServerHttpSecurity;

import org.springframework.security.web.server.SecurityWebFilterChain;

**@Configuration**

**@EnableWebFluxSecurity**

**public class ScmsAPIGatewayConfig {**

private final JwtAuthenticationFilter jwtAuthenticationFilter;

public ScmsAPIGatewayConfig(JwtAuthenticationFilter jwtAuthenticationFilter) {

this.jwtAuthenticationFilter = jwtAuthenticationFilter;

}

**@Bean**

**public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {**

**return** http

.csrf(ServerHttpSecurity.CsrfSpec::disable)

.authorizeExchange(exchanges -> exchanges

.pathMatchers("/auth/\*\*").permitAll()

.pathMatchers("/SCMS-first-microservice/\*\*").authenticated()

.pathMatchers("/SCMS-second-microservice/\*\*").authenticated()

.anyExchange().permitAll()

)

.addFilterAt(jwtAuthenticationFilter, SecurityWebFiltersOrder.AUTHENTICATION)

.build();

**}**

**}**

# **Step 2: Create Authorization Service**

### **2.1 Add Dependencies**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

### **2.2 Define User Entity**

import jakarta.persistence.\*;

import lombok.Data;

**@Entity**

**@Data**

**@Table(name = "users")**

**public class User {**

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(unique = true)

private String username;

private String password;

**}**

### **2.3 Create UserDetailsService**

import org.springframework.security.core.userdetails.\*;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.stereotype.Service;

import java.util.Optional;

**@Service**

**public class CustomUserDetailsService implements UserDetailsService {**

private final UserRepository userRepository;

**public CustomUserDetailsService(UserRepository userRepository) {**

**this.userRepository = userRepository;**

**}**

**@Override**

**public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {**

Optional<User> user = userRepository.findByUsername(username);

return user.map(u -> User.withUsername(u.getUsername())

.password(u.getPassword())

.roles("USER").build())

.orElseThrow(() -> new UsernameNotFoundException("User not found"));

}

**}**

### **2.4 Expose Authentication Endpoint**

import org.springframework.web.bind.annotation.\*;

**@RestController**

**@RequestMapping("/auth")**

**public class AuthController {**

private final JwtUtil jwtUtil;

**public AuthController(JwtUtil jwtUtil) {**

**this.jwtUtil = jwtUtil;**

**}**

**@PostMapping("/login")**

**public String login(@RequestParam String username, @RequestParam String password) {**

return jwtUtil.generateToken(username);

**}**

**}**

# **Final Steps**

1. Start **Authorization Service** → Issue JWT.
2. Send JWT in **Authorization Header** (Bearer <token>) to API Gateway.
3. API Gateway **validates** JWT and routes requests.

This ensures **secure microservices communication** with JWT! 🚀 Let me know if you need enhancements.

**In JwtHelper I am getting errors at parserBuilder**  
  
  
It looks like you are using an outdated or incorrect version of the **io.jsonwebtoken (JJWT) library**, or the **JWT signing key** is not correctly set up. Let’s fix it.

## **Fix 1: Use Correct JJWT Version**

Ensure that you have **JJWT version 0.11.5** or later in your pom.xml:

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

🚀 **Why?** JJWT 0.11.x introduced **parserBuilder()** instead of Jwts.parser().

## **Fix 2: Correct JWT Signing Key**

Modify your JwtUtil class to ensure the **signing key** is properly formatted.

### **✅ Fixed JwtUtil.java**

import io.jsonwebtoken.\*;

import io.jsonwebtoken.security.Keys;

import org.springframework.stereotype.Component;

import java.security.Key;

import java.util.Date;

**@Component**

**public class JwtUtil {**

private static final String SECRET\_KEY = "your-256-bit-secret-your-256-bit-secret";

private static final long EXPIRATION\_TIME = 86400000; // 1 day

private final Key key = Keys.hmacShaKeyFor(SECRET\_KEY.getBytes());

**public String generateToken(String username) {**

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))

.signWith(key, SignatureAlgorithm.HS256)

.compact();

**}**

**public boolean validateToken(String token) {**

try {

Jwts.parserBuilder() // ✅ Use parserBuilder() instead of parser()

.setSigningKey(key)

.build()

.parseClaimsJws(token);

return true;

} catch (JwtException e) {

return false;

}

**}**

**public String extractUsername(String token) {**

return Jwts.parserBuilder()

.setSigningKey(key)

.build()

.parseClaimsJws(token)

.getBody()

.getSubject();

**}**

**}**

## **Fix 3: Ensure the Secret Key is at least 256 bits**

JJWT **requires a minimum 256-bit key for HS256**. If your secret key is too short, you will get an error.

### **✅ Fix: Use a Secure Key**

Modify SECRET\_KEY:

private static final String SECRET\_KEY = Base64.getEncoder().encodeToString("your-secure-secret-key-should-be-very-long".getBytes());

OR  
 Use java.security.SecureRandom:

private static final Key SECRET\_KEY = Keys.secretKeyFor(SignatureAlgorithm.HS256);

### **Final Steps to Fix Parser Issues**

1. **Ensure JJWT version is 0.11.5** in pom.xml.
2. **Use parserBuilder()**, not parser().
3. **Use a 256-bit secret key** (Keys.hmacShaKeyFor() method).

Now your JWT **should work without errors**! 🚀 Let me know if you need more help.

**To Clear Unstable JWT version dependencies**   
  
  
You are using **JJWT 0.12.5**, which is **a pre-release version** and may have breaking changes. The latest stable version is **0.11.5**.

### **Possible Issues in JJWT 0.12.5**

1. **parserBuilder() API may have changed** (since it's not a stable release).
2. **Dependency conflicts or missing transitive dependencies** might cause issues.
3. **Breaking changes from 0.11.x to 0.12.x** could affect how you parse or generate JWTs.

### **Solution 1: Downgrade to Latest Stable Version (0.11.5)**

Change your dependencies to:

<!-- Stable JJWT Dependencies (Recommended) -->

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

✅ This version is stable and works with parserBuilder().

### **Solution 2: Check for Changes in JJWT 0.12.5**

If you **must** use 0.12.5, check the official JJWT repository for API changes:  
 🔗 [JJWT GitHub](https://github.com/jwtk/jjwt)

Try replacing:

Jwts.parserBuilder()

with:

Jwts.builder().somethingNew() // Check official docs for correct method

But **downgrading to 0.11.5 is the safest solution**. Let me know if you need more help! 🚀

**To resolve issues When there is an issue with the WebFilter implementation**  
  
  
  
Your JwtAuthenticationFilter implementation has a few issues:

### **🔴 Issues in Your Code**

1. **request is undefined** → You need to extract ServerHttpRequest from exchange.
2. **request is being cast incorrectly** → ServerWebExchange is already available; no need to cast it from request.
3. **chain.filter(request) is incorrect** → It should be chain.filter(exchange).
4. **Does not set authentication in SecurityContext** → It only validates the token but doesn't set the authenticated user.

## **✅ Fixed JwtAuthenticationFilter Implementation**

This filter correctly **extracts, validates, and sets authentication** in the **Spring Security context**.

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.server.reactive.ServerHttpRequest;

import org.springframework.http.server.reactive.ServerHttpResponse;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.ReactiveSecurityContextHolder;

import org.springframework.security.core.context.SecurityContext;

import org.springframework.security.core.context.SecurityContextImpl;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.web.server.authentication.AuthenticationWebFilter;

import org.springframework.web.server.ServerWebExchange;

import org.springframework.web.server.WebFilter;

import org.springframework.web.server.WebFilterChain;

import reactor.core.publisher.Mono;

**public class JwtAuthenticationFilter implements WebFilter {**

private final JwtUtil jwtUtil;

private final UserDetailsService userDetailsService;

**public JwtAuthenticationFilter(JwtUtil jwtUtil, UserDetailsService userDetailsService) {**

**this.jwtUtil = jwtUtil;**

**this.userDetailsService = userDetailsService;**

**}**

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

ServerHttpRequest request = exchange.getRequest();

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

if (token == null || !token.startsWith("Bearer ")) {

return chain.filter(exchange); // Proceed if no token is present

}

token = token.substring(7); // Remove "Bearer " prefix

if (!jwtUtil.validateToken(token)) {

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

}

// Extract username and load user details

String username = jwtUtil.extractUsername(token);

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

UsernamePasswordAuthenticationToken authToken =

new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

SecurityContext securityContext = new SecurityContextImpl(authToken);

// Store authentication in the reactive security context

return chain.filter(exchange)

.contextWrite(ReactiveSecurityContextHolder.withSecurityContext(Mono.just(securityContext)));

**}**

**}**

## **Key Fixes**

✔ **Extracted ServerHttpRequest correctly** ✔ **Used chain.filter(exchange), not (ServerWebExchange) request** ✔ **Validated token before proceeding** ✔ **Set authentication in the SecurityContext** (so it works with Spring Security)

This filter will now correctly **extract JWT, validate it, and set user authentication** in the reactive context. 🚀 Let me know if you need further modifications!

**Clarification on Objective and Usage of Spring-web and Spring-web flux**  
  
  
  
**Cause of the Error**

The error indicates that **two conflicting Spring Security configurations** are present:

1. **WebSecurityConfiguration.class** (Used for Spring MVC Security)
2. **WebFluxSecurityConfiguration.class** (Used for Spring WebFlux Security)

Since **Spring Cloud Gateway is reactive**, you **must use WebFlux security** instead of MVC security.

## **✅ Solution: Ensure You Are Using Only WebFlux Security**

To fix the issue, follow these steps:

### **1️⃣ Remove Any Spring MVC Security Dependencies**

In your pom.xml, ensure **ONLY** spring-boot-starter-security and spring-boot-starter-webflux are present. **REMOVE** spring-boot-starter-web if it exists, as it causes conflicts.

<!-- Keep only WebFlux Security -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<!-- REMOVE spring-boot-starter-web to avoid conflicts -->

### **2️⃣ Set spring.main.web-application-type=reactive in application.yml**

Add this to application.yml:

spring:

main:

web-application-type: reactive

This ensures your application runs in **WebFlux mode**.

### **3️⃣ Use @EnableWebFluxSecurity Instead of @EnableWebSecurity**

Modify your **API Gateway Security Configuration** (ScmsAPIGatewayConfig.java):

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;

import org.springframework.security.config.web.server.ServerHttpSecurity;

import org.springframework.security.web.server.SecurityWebFilterChain;

**@Configuration**

**@EnableWebFluxSecurity // Ensure you use WebFlux security**

**public class ScmsAPIGatewayConfig {**

**@Bean**

**public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {**

return http

.csrf(ServerHttpSecurity.CsrfSpec::disable)

.authorizeExchange(exchange -> exchange

.pathMatchers("/SCMS-first-microservice/\*\*").authenticated()

.pathMatchers("/SCMS-second-microservice/\*\*").authenticated()

.anyExchange().permitAll())

.build();

**}**

**}**

### **4️⃣ Enable Bean Overriding (Optional)**

If you **still get the error**, add this in application.yml (**only if required**):

spring:

main:

allow-bean-definition-overriding: true

But this is **not recommended** unless necessary.

### **🚀 Final Steps**

1. **Ensure spring-boot-starter-web is removed**
2. **Use spring-boot-starter-webflux**
3. **Set spring.main.web-application-type=reactive**
4. **Use @EnableWebFluxSecurity instead of @EnableWebSecurity**
5. **Restart your application**

This will **resolve the conflict** and properly configure **Spring Security with WebFlux** for your API Gateway. 🚀 Let me know if you need further debugging!

**Leads to Clear-off authorization failure issues**   
  
Since you’ve verified all the key points, but you're still getting 401 Unauthorized, let’s **dig deeper** and perform a **step-by-step debugging** to pinpoint the issue.

## **🔍 Additional Debugging Steps**

### **✅ Step 1: Manually Test Directly Accessing the Authorization Microservice**

**Bypass API Gateway** and call your Authorization Microservice directly:  
 curl -X POST http://localhost:8080/auth/register -H "Content-Type: application/json" -d '{"username":"test","password":"123"}'

* + If this **fails with 401**, then the issue is inside the **Authorization Microservice’s security settings**.
  + If it **works fine**, the issue is at the **API Gateway level**.

### **✅ Step 2: Enable DEBUG Logs for Spring Security**

Add this property in both **API Gateway** and **Authorization Microservice**:

logging.level.org.springframework.security=DEBUG

Now restart your services and check the logs. You should see **detailed Spring Security logs** showing:

* Which filter is **blocking the request**?
* What authentication checks are **failing**?

### **✅ Step 3: Verify Security Configuration in Authorization Microservice**

Your **Authorization Microservice** should have:

**@Configuration**

**@EnableWebFluxSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityWebFilterChain securityFilterChain(ServerHttpSecurity httpSecurity) {**

return httpSecurity.csrf(ServerHttpSecurity.CsrfSpec::disable)

.authorizeExchange(exchange -> exchange

.pathMatchers("/auth/\*\*").permitAll() // Allow all /auth/\*\* requests

.anyExchange().authenticated())

.build();

**}**

**}**

### **🔹 Potential Issue: Spring Boot Auto-Security**

* If you **haven’t defined any security configuration**, **Spring Boot applies default security, blocking all endpoints.**
* You must explicitly allow /auth/\*\*.

### **✅ Step 4: Check API Gateway Routing**

Your **API Gateway routes should be correct**:

**@Bean**

**RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {**

return builder.routes()

.route(p -> p.path("/auth/\*\*")

.uri("lb://SCMS-authorisation-microservice")) // Ensure service name is correct

.route(p -> p.path("/SCMS-first-microservice/\*\*")

.uri("lb://SCMS-first-microservice"))

.route(p -> p.path("/SCMS-second-microservice/\*\*")

.uri("lb://SCMS-second-microservice"))

.build();

**}**

### **🔹 Check if API Gateway is properly forwarding requests**

Run the following test to check the actual routing:  
 curl -X POST http://localhost:8765/auth/register -H "Content-Type: application/json" -d '{"username":"test","password":"123"}' -v

* **Look at the response headers:**
  + If it contains SCMS-authorisation-microservice, the request **reached the Authorization Microservice**.
  + If **not**, the request might not be forwarded correctly.

### **✅ Step 5: Temporarily Disable API Gateway Security**

To isolate whether **API Gateway security is blocking the request**, try removing security temporarily:

**@Bean**

**public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {**

return http.csrf(ServerHttpSecurity.CsrfSpec::disable)

.authorizeExchange(exchange -> exchange.anyExchange().permitAll()) // Allow all

.build();

**}**

* **If /auth/register works after this**, then **API Gateway security is the issue**.
* **If it still fails**, then the issue is inside the **Authorization Microservice**.

### **✅ Step 6: Verify JWT Filter**

Your **JwtAuthenticationFilter** should \*\*skip filtering for /auth/\*\* routes:

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

String path = exchange.getRequest().getURI().getPath();

// Allow /auth/\*\* without JWT validation

if (path.startsWith("/auth/")) {

return chain.filter(exchange);

}

String token = exchange.getRequest().getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

if (token == null || !token.startsWith("Bearer ")) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);

return exchange.getResponse().setComplete();

}

return chain.filter(exchange);

**}**

## **🔹 Final Check**

### **🔥 Now, Test These Scenarios**

| **Test Case** | **Expected Response** |
| --- | --- |
| curl -X POST http://localhost:8080/auth/register | Should return **200 OK** (if microservice works) |
| curl -X POST http://localhost:8765/auth/register | Should return **200 OK** (if API Gateway works) |
| Check API Gateway logs | Should show **/auth/register is allowed** |
| Check Authorization Microservice logs | Should show **request reaching service** |

## **🚀 Next Steps**

1️⃣ **Perform all these debugging steps** 2️⃣ **Check logs carefully** 3️⃣ **Let me know where it still fails!**

I’ll help you fix it. 🚀🔥  
  
**Clarification on use of @EnableWebSecurity and @EnableWebfluxSecurity**  
  
You are using **Spring Security for Spring MVC (@EnableWebSecurity)**, but your API Gateway is using **Spring Security for WebFlux (@EnableWebFluxSecurity)**.

Here’s the **correct configuration to use** based on whether your **Authorization Microservice** is a **Spring Boot MVC (Servlet-based)** or **WebFlux (Reactive-based)** application.

### **🔥 Case 1: Your Authorization Microservice Uses Spring Boot MVC (Servlet-Based)**

If your Authorization Microservice is a **Spring Boot MVC** application, then **your configuration is correct** but needs **a few improvements**:

#### **✅ Use this Security Configuration for MVC-Based Authorization Microservice:**

**@Configuration**

**@EnableWebSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {**

return httpSecurity

.csrf(csrf -> csrf.disable()) // Disable CSRF for APIs

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll() // Allow /auth/\*\* without authentication

.anyRequest().authenticated()) // Secure other endpoints

.httpBasic(Customizer.withDefaults()) // Allow API testing via Postman

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // Use JWT, no sessions

.build();

**}**

**}**

### **Why This is Correct?**

✔ **@EnableWebSecurity**: Used for Servlet-based security (Spring MVC).  
 ✔ **CSRF Disabled**: Since it's a REST API.  
 ✔ **Permits /auth/\*\***: Ensures /auth/register and /auth/login are accessible.  
 ✔ **JWT Authentication (Stateless)**: Ensures API is stateless, avoiding session-based authentication.

### **🔥 Case 2: Your Authorization Microservice Uses WebFlux (Reactive-Based)**

If your **Authorization Microservice** is built using **Spring WebFlux**, you must use **@EnableWebFluxSecurity** instead.

#### **✅ Use this Security Configuration for WebFlux-Based Authorization Microservice:**

**@Configuration**

**@EnableWebFluxSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityWebFilterChain securityFilterChain(ServerHttpSecurity httpSecurity) {**

return httpSecurity

.csrf(ServerHttpSecurity.CsrfSpec::disable) // Disable CSRF for APIs

.authorizeExchange(auth -> auth

.pathMatchers("/auth/\*\*").permitAll() // Allow /auth/\*\* without authentication

.anyExchange().authenticated()) // Secure other endpoints

.httpBasic(Customizer.withDefaults()) // Allow API testing via Postman

.build();

**}**

**}**

### **Why This is Correct?**

✔ **@EnableWebFluxSecurity**: Required for WebFlux (Reactive-based applications).  
 ✔ **authorizeExchange() instead of authorizeHttpRequests()**: WebFlux uses authorizeExchange().  
 ✔ **Ensures /auth/\*\* endpoints are publicly accessible**.

## **🚀 Which One Should You Use?**

* If your **Authorization Microservice is built with Spring MVC**, use **Case 1** (@EnableWebSecurity).
* If your **Authorization Microservice is built with Spring WebFlux**, use **Case 2** (@EnableWebFluxSecurity).

### **🔹 How to Check If You Are Using WebFlux or MVC?**

If your pom.xml contains:  
  
 <dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

* + ✅ You are using **Spring MVC** → Use **Case 1 (@EnableWebSecurity)**.

If your pom.xml contains:  
  
 <dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

* + ✅ You are using **Spring WebFlux** → Use **Case 2 (@EnableWebFluxSecurity)**.

## **🚀 Final Steps to Fix Your 401 Unauthorized Issue**

### **✅ Make Sure You:**

1️⃣ Use the **correct security configuration (MVC or WebFlux)** for your Authorization Microservice.  
 2️⃣ Check logs (logging.level.org.springframework.security=DEBUG) to see if /auth/register is being blocked.  
 3️⃣ Ensure **Spring Security is not overriding API Gateway rules** (test by disabling security temporarily).  
 4️⃣ Use curl or Postman to test **API Gateway & Microservice separately** to identify where the issue is.

🚀 Try these steps and let me know what happens! 🔥  
  
  
**Clarification for Usage of Spring-web and Spring-web flux with respect to Current Industry standards**  
  
  
**Spring Web (MVC) vs. Spring WebFlux for Authorization Microservice – Industry Standards**

#### **✅ Industry Best Practice Recommendation:**

For an **Authorization Microservice**, **Spring Web (Spring MVC) is preferred** in most enterprise applications **unless** there is a specific requirement for reactive programming.

## **🚀 When to Use Spring Web (MVC) for the Authorization Microservice?**

**✔ Recommended for most industry-standard applications**

* **Easier to integrate with databases (JPA, Hibernate) and security mechanisms (Spring Security, JWT)**.
* **Better compatibility with existing enterprise systems** (works well with Spring Boot-based microservices).
* **More widely adopted and understood by developers**.
* **Ideal for request-response architectures** (e.g., authentication, user management).
* **Better debugging, monitoring, and observability** than reactive applications.

### **👉 Industry Standard Setup: Spring Web (MVC)**

* Use **Spring Boot Starter Web** (spring-boot-starter-web).
* Use **Spring Security for authentication & JWT**.
* Use **Spring Data JPA with PostgreSQL/MySQL** for user storage.
* Use **Spring MVC Controllers** (@RestController) for handling authentication.

✅ **Your pom.xml should have:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

✅ **Security Configuration (Spring MVC - Industry Standard)**

**@Configuration**

**@EnableWebSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {**

return httpSecurity

.csrf(csrf -> csrf.disable()) // Disable CSRF for APIs

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll() // Allow authentication endpoints

.anyRequest().authenticated()) // Secure all other endpoints

.httpBasic(Customizer.withDefaults()) // Enable authentication via Postman

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // JWT-based auth

.build();

**}**

**}**

## **🚀 When to Use Spring WebFlux for the Authorization Microservice?**

**Use WebFlux ONLY if:** 🔹 You are building a high-concurrency, non-blocking system with **thousands of concurrent users**.  
 🔹 Your microservice **interacts with other reactive services** (e.g., MongoDB, Cassandra, WebSockets).  
 🔹 Your system needs **real-time streaming (e.g., live authentication events, push notifications)**.  
 🔹 You are working in a **fully reactive ecosystem** (i.e., your other microservices are also using WebFlux).

### **👉 If Using WebFlux**

* Use **Spring Boot Starter WebFlux** (spring-boot-starter-webflux).
* Use **Reactive Spring Security (@EnableWebFluxSecurity)**.
* Use **Reactive MongoDB instead of JPA**.

✅ **Your pom.xml should have:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

✅ **Security Configuration (WebFlux)**

**@Configuration**

**@EnableWebFluxSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityWebFilterChain securityFilterChain(ServerHttpSecurity httpSecurity) {**

return httpSecurity

.csrf(ServerHttpSecurity.CsrfSpec::disable) // Disable CSRF for APIs

.authorizeExchange(auth -> auth

.pathMatchers("/auth/\*\*").permitAll() // Allow authentication endpoints

.anyExchange().authenticated()) // Secure all other endpoints

.httpBasic(Customizer.withDefaults()) // Enable authentication via Postman

.build();

**}**

**}**

## **🚀 Final Industry Recommendation**

### **🔹 Use Spring Web (MVC) for Authorization Microservice unless you have a strong reason for WebFlux.**

* ✅ **Better integration with databases (JPA, MySQL, PostgreSQL).**
* ✅ **Easier debugging, logging, and monitoring.**
* ✅ **More widely adopted in enterprise applications.**
* ✅ **Smoother integration with Spring Boot microservices (most services are non-reactive).**
* ✅ **Sufficient for handling authentication, authorization, and JWT validation.**

If you are not using **reactive databases (MongoDB, Cassandra)** or **high-concurrency streaming**, **stick to Spring Web (MVC).**

🚀 **Go with Spring Web (MVC) for your Authorization Microservice for industry-standard implementation!  
  
  
Clarification on Implementation of JWTAuthentication filter on Authorisation Microservice before UsernamePasswordAuthenticationFilter.**

### **In a Spring Web (MVC) Based Authorization Microservice, You Do Not Need a JWTAuthenticationFilter**

### **Here's Why:**

#### **1️⃣ Purpose of Authorization Microservice:**

* The **Authorization Microservice** is typically responsible for **authenticating users** and **issuing JWT tokens**—not validating them.
* JWT validation is **usually done by API Gateway** or **downstream microservices**.

#### **2️⃣ How Authentication Works in the Authorization Microservice:**

* Users send their **credentials** (username and password) to your /auth/login endpoint.
* Your service **authenticates** the user and **generates a JWT** using a utility class (like JwtUtil).
* You **send back the JWT** in the response for the client to use in future requests.

#### **3️⃣ JWTAuthenticationFilter Usage:**

* **JWTAuthenticationFilter** is mainly used to **validate incoming JWT tokens** in microservices or API Gateways, not in the service responsible for issuing them.

## **✅ Correct Approach for Authorization Microservice (Spring MVC)**

### **1. Security Configuration:**

**@Configuration**

**@EnableWebSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {**

return httpSecurity

.csrf(csrf -> csrf.disable()) // Disable CSRF protection for APIs

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll() // Allow access to /auth/\*\* endpoints

.anyRequest().authenticated()) // Secure other endpoints

.httpBasic(Customizer.withDefaults()) // Enable basic auth for testing via Postman

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // Stateless session

.build();

**}**

**}**

### **2. AuthController Implementation:**

**@RestController**

**@RequestMapping("/auth")**

**public class AuthController {**

@Autowired

private JwtUtil jwtUtil;

@Autowired

private AuthenticationManager authenticationManager;

**@PostMapping("/login")**

**public ResponseEntity<String> login(@RequestParam String username, @RequestParam String password) {**

try {

authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));

String token = jwtUtil.generateToken(username);

return ResponseEntity.ok(token);

} catch (Exception e) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");

**}**

**}**

**@PostMapping("/register")**

**public ResponseEntity<?> registerUser(@RequestBody RegisterUser registerUser) {**

// Registration logic

return ResponseEntity.ok("User registered successfully");

**}**

**}**

### **3. JWT Utility Class (JwtUtil.java):**

**@Component**

**public class JwtUtil {**

private static final String SECRET\_KEY = "your-256-bit-secret";

**public String generateToken(String username) {**

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

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

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

**}**

**}**

### **💡 Why You Don’t Need JWTAuthenticationFilter in Authorization Microservice:**

* **No Incoming JWT Tokens:**
  + Your /auth/login endpoint doesn’t receive JWT tokens to validate—it creates them.
* **Not Handling JWT Authentication:**
  + Your service's job is to **authenticate users** and **issue tokens**, not to verify tokens.
* **Stateless Sessions:**
  + Since the service uses JWT for stateless authentication, no session management is needed.

### **💡 Where to Use JWTAuthenticationFilter?**

* **API Gateway:**
  + Validate JWT tokens before routing requests to internal microservices.
* **Downstream Microservices:**
  + Verify JWT tokens to ensure the request is from an authenticated user.

## **🚀 Summary:**

* **Do Not Use JWTAuthenticationFilter** in your Authorization Microservice.
* Focus on **user authentication and JWT token generation**.
* Configure your **API Gateway** or **other microservices** to use JWTAuthenticationFilter for validating JWT tokens.

Let me know if you need more clarity or further guidance! 🚀

**401-Unauthorised issue fixing**   
  
  
Your configuration and authentication process seem correct at a high level, but the issue is likely due to one of the following problems:

### **🛠 Possible Issues and Fixes**

#### **1️⃣ Issue: UserDetailsService Not Loaded Correctly**

* The AuthenticationManager relies on **UserDetailsService** to load user details.
* If UserDetailsService is not correctly implemented or not returning a user, authentication will fail.

#### **✅ Fix: Ensure UserDetailsService Implementation is Correct**

Check if you have a correct implementation of UserDetailsService that loads user details from the database:

**@Service**

**public class CustomUserDetailsService implements UserDetailsService {**

@Autowired

private UserInfoRepository userInfoRepository;

**@Override**

**public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {**

RegisterUser user = userInfoRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException("User not found"));

return new User(user.getUsername(), user.getPassword(), new ArrayList<>());

**}**

**}**

* Make sure userInfoRepository.findByUsername(username) exists and correctly fetches users.

#### **2️⃣ Issue: Password Not Encoded in the Database**

* Spring Security **expects encoded passwords** (BCryptPasswordEncoder).
* If your passwords are stored in plain text, authentication **will fail**.

#### **✅ Fix: Encode Passwords When Storing Users**

Modify the registeringUser method to encode the password before saving:

**@Autowired**

**private BCryptPasswordEncoder passwordEncoder;**

**public RegisterUser registeringUser(RegisterUser registerUser) {**

registerUser.setPassword(passwordEncoder.encode(registerUser.getPassword()));

return userInfoRepository.save(registerUser);

**}**

🔹 **Ensure stored passwords are encoded** using BCrypt, or authentication will fail.

#### **3️⃣ Issue: No PasswordEncoder Bean Defined**

* You are using new BCryptPasswordEncoder() inside authenticationProvider(), but Spring Security **needs it as a bean**.

#### **✅ Fix: Define a Password Encoder Bean**

Add this in your SecurityConfig:

**@Bean**

**public BCryptPasswordEncoder passwordEncoder() {**

**return new BCryptPasswordEncoder();**

**}**

#### **4️⃣ Issue: Missing @Transactional for registeringUser**

* If registeringUser is not marked as @Transactional, it may not persist correctly.

#### **✅ Fix: Ensure @Transactional is Present**

**@Transactional**

**public RegisterUser registeringUser(RegisterUser registerUser) {**

registerUser.setPassword(passwordEncoder.encode(registerUser.getPassword()));

**return** userInfoRepository.save(registerUser);

**}**

### **🚀 Final Steps to Debug**

If you still get 401 Unauthorized:

1. **Check Logs**:  
   * Add log.info("Authenticating user: " + users.getUsername()); before authenticationManager.authenticate() to confirm that authentication is attempted.
2. **Verify Database Data**:  
   * Manually check the database to ensure **passwords are stored in encoded format**.
3. **Check Security Debugging Logs**:

Add this to application.properties to see detailed security logs:  
 logging.level.org.springframework.security=DEBUG

### **🎯 Final Expected Working Flow**

1. **User registers** → Password is encoded and saved.
2. **User logs in** → authenticationManager.authenticate() verifies username & password.
3. **If authentication is successful**, JWT is returned.

Let me know if you need further debugging help! 🚀

**Clarification on token handling at the API gateway level with the same type of secret key and algorithm**

It looks like your JWT validation method (jwtUtil.validateToken(token)) is returning false, causing the request to be unauthorized. Let's debug and fix the issue systematically.

## **🛠 Possible Issues and Fixes**

### **1️⃣ Issue: JWT Token is Not Properly Signed**

If the token generated during login is not signed using the correct secret key, the validation will fail.

### **✅ Fix: Ensure Consistent Secret Key**

Check if the secret key used to sign the JWT in Authorization Microservice is the same as the one used to validate it in API Gateway.

#### **🔹 Verify JWT Token Generation (Authorization Microservice)**

**@Component**

**public class JwtUtil {**

private static final String SECRET\_KEY = "yourSecretKey"; // SAME SECRET KEY

**public String generateToken(String username) {**

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 Hours Validity

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

**}**

**public boolean validateToken(String token) {**

try {

Jwts.parser().setSigningKey(SECRET\_KEY).parseClaimsJws(token);

return true;

} catch (Exception e) {

return false;

**}**

**}**

**}**

🔹 **Ensure that the same SECRET\_KEY is used in both microservices.**

### **2️⃣ Issue: Missing "Bearer " Prefix When Sending Token**

* Ensure that when calling other microservices, the Authorization header has Bearer <token>.
* Debug and print the received token in JwtAuthenticationFilter to check.

### **✅ Fix: Debug and Print the Token**

Modify your JwtAuthenticationFilter:

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

ServerHttpRequest request = exchange.getRequest();

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

String path = request.getURI().getPath();

if (path.startsWith("/auth/")) {

return chain.filter(exchange);

}

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

System.out.println("Extracted Token: " + token);

if (!jwtUtil.validateToken(token)) {

System.out.println("JWT Validation Failed");

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

}

} else {

System.out.println("Authorization Header Missing or Invalid");

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

}

return chain.filter(exchange);

**}**

🔹 **Check console logs to see if the token is being extracted properly.**

### **3️⃣ Issue: Expired Token**

* If you generated the token a while ago and used it later, it might have expired.
* Decode your token at [**jwt.io**](https://jwt.io/) and check the expiration (exp).

### **✅ Fix: Increase Token Expiry**

Modify generateToken:

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 24)) // 24 Hours Validity

### **4️⃣ Issue: JWT Token Algorithm Mismatch**

* Ensure that the JWT is signed with **HS256**, not RS256 or any other algorithm.

#### **✅ Fix: Use Correct Algorithm**

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

## **🛠 Final Steps for Debugging**

1. **Print the JWT Token** from AuthController **immediately after login**.
2. **Copy the Token** and **validate it at** [**jwt.io**](https://jwt.io/) to check if:
   * **Algorithm is HS256**
   * **Secret Key is correct**
   * **Expiration (exp) is valid**
3. **Try sending a request with Authorization: Bearer <your\_token> manually using Postman.**
4. **Check API Gateway Logs**:
   * Extracted Token: ...
   * JWT Validation Failed

Let me know what logs you get, and I can guide you further! 🚀

**Secret Key handling at API Gateway and Authorisation microservice level with same Secret Key**

Your JWT validation issue is likely caused by the fact that **the secret key is dynamically generated at runtime** in each microservice. Since the SECRETEKEY is generated in the constructor, the key will be different for the Authorization Microservice and API Gateway each time they restart.

### **💡 Issue: Secret Key is Not Persistent**

* The secret key should **remain the same** across all microservices and restarts.
* **Right now, every restart generates a new key**, meaning the tokens issued by the Authorization Microservice won't be valid in the API Gateway.

## **✅ Solution: Use a Fixed Secret Key**

Instead of dynamically generating the key in the constructor, **store a fixed key in application properties or an environment variable**.

### **1️⃣ Store a Fixed Secret Key**

Modify your JwtUtil class:

**@Component**

**public class JwtUtil {**

**@Value("${jwt.secret}") // Read from properties file or environment variable**

private String secretKey;

**private SecretKey getKey() {**

byte[] keyBytes = Decoders.BASE64.decode(secretKey);

return Keys.hmacShaKeyFor(keyBytes);

**}**

**public String generateToken(String username) {**

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 Hours

.signWith(getKey(), SignatureAlgorithm.HS256)

.compact();

**}**

**public boolean validateToken(String token) {**

try {

Jwts.parserBuilder().setSigningKey(getKey()).build().parseClaimsJws(token);

return true;

} catch (JwtException e) {

return false;

**}**

**}**

**}**

### **2️⃣ Add Secret Key in application.properties**

Store the same secret in **both** the Authorization Microservice and API Gateway.

jwt.secret=YourSuperSecretKeyBase64Encoded

**💡 Generate a secure base64 key** using this command:

openssl rand -base64 32

### **3️⃣ Ensure the Same Key in Both Microservices**

* The **Authorization Microservice** uses the key to **sign tokens**.
* The **API Gateway** uses the same key to **validate tokens**.
* Since it's stored in application.properties, it will remain the same after restarts.

## **✅ Final Steps for Debugging**

1. **Remove the dynamic key generation** from JwtUtil().
2. **Restart both services** after applying the fixed secret key.
3. **Verify JWT at** [**jwt.io**](https://jwt.io/) after login.
4. **Test API Gateway authentication again**.

🔹 This should resolve your issue! Try it and let me know if you need further debugging. 🚀

**Clarification on JwtAuthenticationFilter configuration after token validation**

### **🔍 Issues in Your JwtAuthenticationFilter Class**

**There are two main issues in your JwtAuthenticationFilter implementation.**

## **❌ Issue 1: Incorrect Response Handling After Token Validation**

### **What’s Wrong?**

**Currently, if the token is valid, you are returning HttpStatus.OK and completing the response:**

if (jwtUtil.validateToken(token)) {

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.OK);

return response.setComplete();

}

### **Why is this a problem?**

* **filter() should allow the request to proceed if the token is valid.**
* **Instead, you're sending an empty 200 OK response and stopping further request processing.**
* **This prevents the request from reaching the actual microservice endpoint.**

### **✅ Solution**

**If the token is valid, set authentication and continue the filter chain instead of stopping it.**

**Replace:**

if (jwtUtil.validateToken(token)) {

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.OK);

return response.setComplete();

}

**With:**

if (jwtUtil.validateToken(token)) {

return chain.filter(exchange); // Continue the request processing

}

## **❌ Issue 2: Missing Security Context Population**

### **What’s Wrong?**

**Even after validating the token, you are not setting the authenticated user in the security context.**

### **Why is this a problem?**

* **Spring Security still considers the user as unauthenticated because SecurityContext is not updated.**
* **Your downstream microservices won't know which user is making the request.**

### **✅ Solution: Set Authentication in SecurityContext**

**Modify the code inside the if (jwtUtil.validateToken(token)) block to extract user details and set authentication:**

if (jwtUtil.validateToken(token)) {

String username = jwtUtil.extractUsername(token);

Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());

SecurityContext context = SecurityContextHolder.createEmptyContext();

context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);

return chain.filter(exchange);

}

* **jwtUtil.extractUsername(token): Extracts the username from the JWT.**
* **UsernamePasswordAuthenticationToken: Represents an authenticated user.**
* **SecurityContextHolder.setContext(context): Stores authentication for the request.**

## **✅ Final Fixed Version**

**@Component**

**public class JwtAuthenticationFilter implements WebFilter {**

@Autowired

private JwtUtil jwtUtil;

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

ServerHttpRequest request = exchange.getRequest();

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

String path = request.getURI().getPath();

// Allow requests to /auth/\*\* without authentication

if (path.startsWith("/auth/")) {

return chain.filter(exchange);

}

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

if (jwtUtil.validateToken(token)) {

String username = jwtUtil.extractUsername(token);

Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());

SecurityContext context = SecurityContextHolder.createEmptyContext();

context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);

return chain.filter(exchange); // Continue request processing

}

}

// Unauthorized response if token is missing or invalid

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

return response.setComplete();

**}**

**}**

## **🔥 Summary of Fixes**

**✅ Fixed issue 1: Removed the incorrect HttpStatus.OK response and allowed the request to proceed.  
 ✅ Fixed issue 2: Set the authentication in SecurityContextHolder, so other microservices can access the authenticated user.**

**Now, your API Gateway should properly authenticate requests and allow valid JWTs to reach downstream microservices. 🚀**

**JwtAuthenticationFilter - Incorrect response handling  
  
  
🔍 Issues in Your JwtAuthenticationFilter Implementation**

**There are still two critical issues in your implementation that are causing the 401 Unauthorized error.**

## **❌ Issue 1: Incorrect Response Handling After Token Validation**

### **What’s Wrong?**

**You are still sending HttpStatus.OK manually after setting authentication:**

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.OK);

response.setComplete();

return chain.filter(exchange);

### **Why is this a problem?**

* **response.setComplete() stops the request from going further.**
* **The request never reaches the intended microservice because filter() is completed too early.**
* **The request needs to continue in the filter chain without returning an explicit response.**

### **✅ Solution**

* **Remove response.setStatusCode(HttpStatus.OK); and response.setComplete();.**
* **Just allow the request to continue down the filter chain.**

## **❌ Issue 2: SecurityContextHolder Does Not Work with Spring WebFlux**

### **What’s Wrong?**

* **SecurityContextHolder does not work in WebFlux like it does in traditional Spring Security.**
* **You cannot directly set authentication in SecurityContextHolder.**

### **✅ Solution**

**Instead of:**

**SecurityContext context = SecurityContextHolder.createEmptyContext();**

**context.setAuthentication(authentication);**

**SecurityContextHolder.setContext(context);**

* **Use SecurityContextRepository to set authentication in a reactive manner.**

## **✅ Final Fixed Version of JwtAuthenticationFilter**

**@Component**

**public class JwtAuthenticationFilter implements WebFilter {**

@Autowired

private JwtUtil jwtUtil;

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

ServerHttpRequest request = exchange.getRequest();

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

String path = request.getURI().getPath();

// Allow /auth/\*\* requests without authentication

if (path.startsWith("/auth/")) {

return chain.filter(exchange);

}

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

if (jwtUtil.validateToken(token)) {

String username = jwtUtil.extractUsername(token);

Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());

// Use Reactive Security Context

return chain.filter(exchange)

.contextWrite(ReactiveSecurityContextHolder.withAuthentication(authentication));

}

}

// If authentication fails, return 401 Unauthorized

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

**return** response.setComplete();

**}**

**}**

## **🔥 Summary of Fixes**

**✅ Fixed issue 1: Removed response.setStatusCode(HttpStatus.OK); and response.setComplete(); to allow request processing.  
 ✅ Fixed issue 2: Used ReactiveSecurityContextHolder.withAuthentication(authentication) instead of SecurityContextHolder.setContext(context).**

**Now, authenticated requests will correctly propagate through the security context, and unauthorized requests will get 401 Unauthorized. 🚀**

**Use Reactive Security Context when using Spring-webflux - for API Gateway**

**To properly use Reactive Security Context in a Spring WebFlux application, we need to leverage ReactiveSecurityContextHolder to store authentication details and ensure that it is accessible throughout the request lifecycle.**

### **✅ Updated JwtAuthenticationFilter with Proper Reactive Security Context**

**@Component**

**public class JwtAuthenticationFilter implements WebFilter {**

@Autowired

private JwtUtil jwtUtil;

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {**

ServerHttpRequest request = exchange.getRequest();

String token = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

String path = request.getURI().getPath();

// Allow /auth/\*\* requests without authentication

if (path.startsWith("/auth/")) {

return chain.filter(exchange);

}

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

if (jwtUtil.validateToken(token)) {

String username = jwtUtil.extractUsername(token);

Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());

// Create a Security Context

SecurityContext securityContext = new SecurityContextImpl(authentication);

// Store Security Context in ReactiveSecurityContextHolder

return chain.filter(exchange)

.contextWrite(ReactiveSecurityContextHolder.withSecurityContext(Mono.just(securityContext)));

}

}

// If authentication fails, return 401 Unauthorized

ServerHttpResponse response = exchange.getResponse();

response.setStatusCode(HttpStatus.UNAUTHORIZED);

**return** response.setComplete();

**}**

**}**

## **🔥 Key Fixes and Best Practices**

### **1️⃣ 🛠 Using ReactiveSecurityContextHolder Correctly**

**We create a Reactive Security Context using:** SecurityContext securityContext = new SecurityContextImpl(authentication);

**We store it in ReactiveSecurityContextHolder with:** contextWrite(ReactiveSecurityContextHolder.withSecurityContext(Mono.just(securityContext)))

### **2️⃣ 🚀 Ensuring the Request Proceeds Properly**

* **Removed response.setStatusCode(HttpStatus.OK); (it was blocking the request from reaching other microservices).**
* **The request now correctly flows through the filter chain.**

## **✅ Final Outcome**

**Now: 1️⃣ When a valid JWT is provided, authentication is stored in the reactive security context.  
 2️⃣ The authenticated request flows normally to the downstream microservices.  
 3️⃣ If the JWT is invalid, the filter returns a 401 Unauthorized response.**

**This is the correct way to implement authentication in a Spring WebFlux API Gateway using JWT. 🚀**