Tutorial: Securing Microservices with Spring Cloud Gateway

Repository: TudorFernea/gateway-security-tutorial

1. Introduction

In modern microservices architectures, security cannot be managed individually by every single service. If a system has 50 microservices, updating security libraries in 50 places is unmaintainable.

The industry-standard solution is the **API Gateway Pattern**. The Gateway acts as the single entry point ("The Front Door") for all traffic. It handles cross-cutting concerns like Authentication, Authorization, SSL Termination, and Logging before the request ever reaches the internal services.[1]

Goal of this Tutorial

We will build a secured microservices system using **Spring Cloud Gateway**. We will implement:

1. **Centralized Security:** A custom filter to block unauthorized requests.
2. **Role-Based Access Control :** Distinguishing between ADMIN and GUEST users.
3. **Identity Propagation:** Injecting user roles into HTTP headers for downstream services.
4. **Observability:** A global audit logger to track all traffic.

---

2. System Architecture

The system consists of two Dockerized containers:

1. **Gateway Service (Port 8080):** The public-facing edge server. It inspects X-API-KEY headers.
2. **Dummy Service (Port 8081):** A private internal service. It trusts the Gateway and alters its response based on the propagated X-USER-ROLE header.

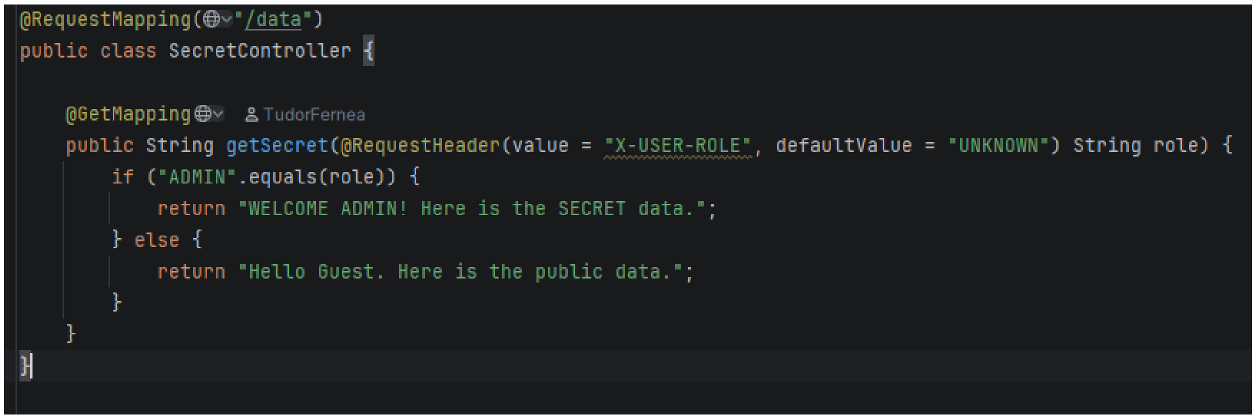
---

3. Implementation Guide

Step 1: The Internal Service

First, we create a simple Spring Boot service (dummy-service). This service does not handle authentication itself. Instead, it relies on the Gateway to tell it who the user is via the X-USER-ROLE header.

Key Code: SecretController.java



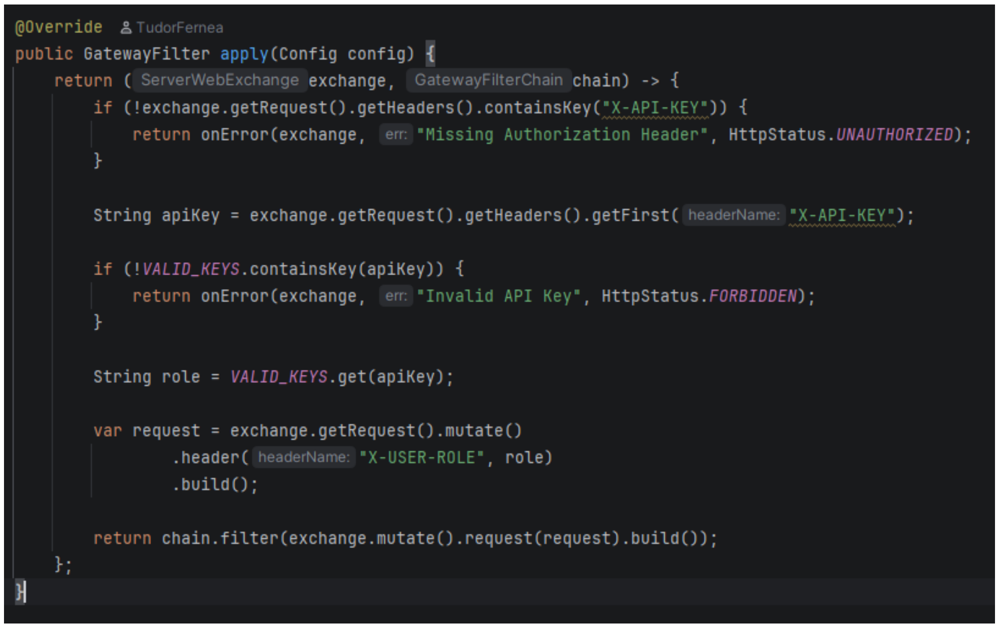
```
@RequestMapping("/data")
public class SecretController {

    @GetMapping    TudorFernea
    public String getSecret(@RequestHeader(value = "X-USER-ROLE", defaultValue = "UNKNOWN") String role) {
        if ("ADMIN".equals(role)) {
            return "WELCOME ADMIN! Here is the SECRET data.";
        } else {
            return "Hello Guest. Here is the public data.";
        }
    }
}
```

Explanation: This demonstrates **decoupling**, the service focuses on business logic rather than security logic.

Step 2: The Gateway Security Filter

We implement a custom GatewayFilter factory. This class intercepts requests and validates the API Key against an in-memory whitelist.

Key Code: AuthFilter.java



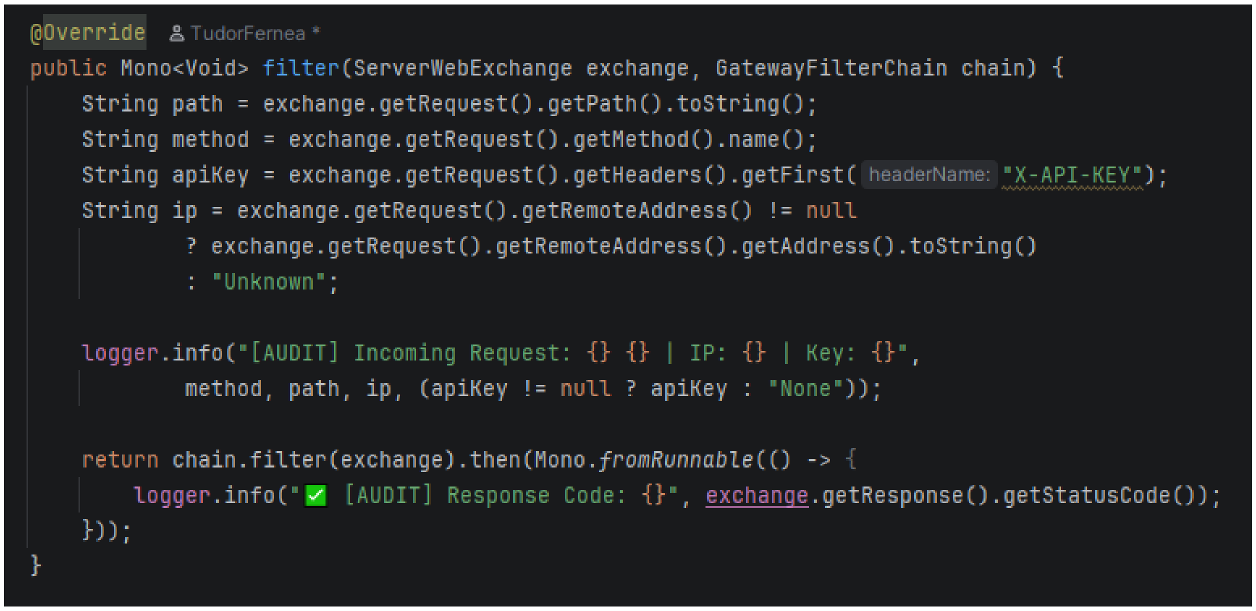
```
@Override  TudorFernea
public GatewayFilter apply(Config config) {
    return ( ServerWebExchange exchange,  GatewayFilterChain chain) -> {
        if (!exchange.getRequest().getHeaders().containsKey("X-API-KEY")) {
            return onError(exchange,  err: "Missing Authorization Header", HttpStatus.UNAUTHORIZED);
        }

        String apiKey = exchange.getRequest().getHeaders().getFirst( headerName: "X-API-KEY");

        if (!VALID_KEYS.containsKey(apiKey)) {
            return onError(exchange,  err: "Invalid API Key", HttpStatus.FORBIDDEN);
        }

        String role = VALID_KEYS.get(apiKey);

        var request = exchange.getRequest().mutate()
                .header( headerName: "X-USER-ROLE", role)
                .build();

        return chain.filter(exchange.mutate().request(request).build());
    };
}
```

Explanation: If the key is valid, we mutate the request to add the X-USER-ROLE header. This safely passes the user's identity to the internal network.

Step 3: Global Auditing (The Monitor)

To ensure observability, we add a GlobalFilter that logs every incoming request. Unlike the security filter, this requires no configuration and runs automatically for all routes.

Key Code: GlobalLoggingFilter.java



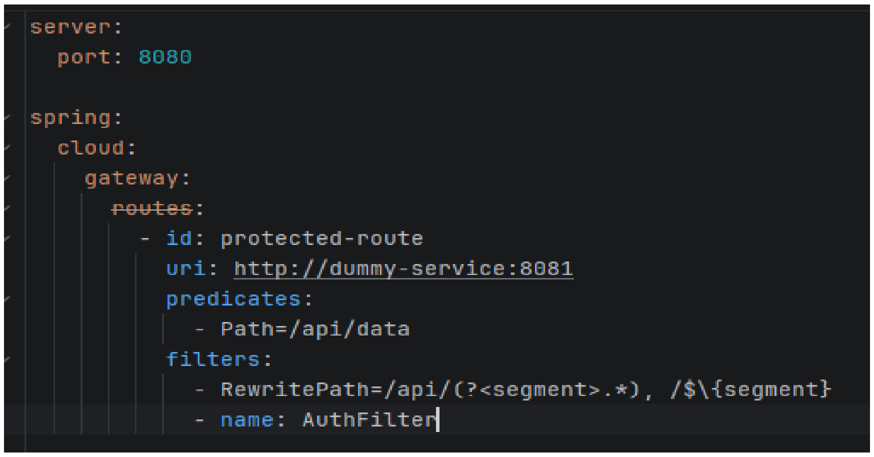
```java
@Override  TudorFernea *
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    String path = exchange.getRequest().getPath().toString();
    String method = exchange.getRequest().getMethod().name();
    String apiKey = exchange.getRequest().getHeaders().getFirst( headerName: "X-API-KEY");
    String ip = exchange.getRequest().getRemoteAddress() != null
            ? exchange.getRequest().getRemoteAddress().getAddress().toString()
            : "Unknown";

    logger.info("[AUDIT] Incoming Request: {} {} | IP: {} | Key: {}",
            method, path, ip, (apiKey != null ? apiKey : "None"));

    return chain.filter(exchange).then(Mono.fromRunnable(() -> {
        logger.info("✅ [AUDIT] Response Code: {}", exchange.getResponse().getStatusCode());
    }));
}
```

---

4. Configuration & Deployment

The Routing Configuration

These are brought together through the application.yml in the Gateway service. This configuration defines the "predicates" (matching rules) and "filters" (logic).



```yaml
server:
  port: 8080

spring:
  cloud:
    gateway:
      ~~routes~~:
        - id: protected-route
          uri: http://dummy-service:8081
          predicates:
            - Path=/api/data
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}
            - name: AuthFilter
```

Explanation: Any request to /api/data is stripped of the prefix, passed through the AuthFilter, and forwarded to the Dummy Service.

Deployment with Docker

We use docker-compose to spin up both services simultaneously in a private network.

- **Gateway** is exposed on port 8080.
- **Dummy Service** is running on 8081 but is accessible *through* the Gateway.