# API Gateway – LLD Document

## 1.LLD Components Overview

| **Component** | **Responsibility** |
| --- | --- |
| **Spring Cloud Gateway** | API Gateway for routing requests to backend services |
| **Spring Security (OAuth2 Resource Server)** | Validates incoming JWTs |
| **Keycloak** | OAuth2 identity provider issuing access tokens |
| **Spring Cloud LoadBalancer / Eureka** | Enables service discovery and load-balanced calls |
| **Bucket4j** (or RedisRateLimiter) | Rate limiting filters for protecting backend services |

## 🗂️ 2. API Gateway Service LLD

### ✅ Component Diagram (UML)

A diagram of a computer

AI-generated content may be incorrect.

## ⚙️ 3. Configuration Files and Code

### 📍 application.yml

yaml

server:

port: 8080

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: user-service

uri: lb://user-service

predicates:

- Path=/users/\*\*

filters:

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 5

redis-rate-limiter.burstCapacity: 10

security:

oauth2:

resourceserver:

jwt:

jwk-set-uri: http://keycloak:8080/realms/demo/protocol/openid-connect/certs

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka

## 🔐 4. Authentication Logic

**SecurityConfig.java**

java

@Configuration

@EnableWebFluxSecurity

public class SecurityConfig {

@Bean

public SecurityWebFilterChain filterChain(ServerHttpSecurity http) {

return http

.authorizeExchange(exchange -> exchange

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

.anyExchange().authenticated())

.oauth2ResourceServer(oauth2 -> oauth2.jwt())

.build();

}

}

### 🔍 Explanation:

* Enables WebFlux security
* Validates incoming requests using JWTs
* Fetches public keys from Keycloak to validate token signatures

## 🔍 5. Service Discovery with Eureka

**application.yml (already covered)**

**Example: RouteLocator Bean (Dynamic Routing Optional)**

java

@Configuration

public class GatewayRoutes {

@Bean

public RouteLocator routeLocator(RouteLocatorBuilder builder) {

return builder.routes()

.route("user-service", r -> r.path("/users/\*\*")

.uri("lb://user-service"))

.build();

}

}

## 🔍 Explanation:

* The URI lb://user-service resolves to the actual IP of the user service using **Spring Cloud Load Balancer** and **Eureka Client**.

## 🚦 6. Custom Rate Limiter Filter (Optional)

java

@Component

public class RateLimiterFilter implements GatewayFilterFactory<RateLimiterFilter.Config> {

private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();

@Override

public GatewayFilter apply(Config config) {

return (exchange, chain) -> {

String ip = exchange.getRequest().getRemoteAddress().getAddress().getHostAddress();

Bucket bucket = buckets.computeIfAbsent(ip, k -> Bucket4j.builder()

.addLimit(Bandwidth.simple(5, Duration.ofSeconds(1)))

.build());

if (bucket.tryConsume(1)) {

return chain.filter(exchange);

}

exchange.getResponse().setStatusCode(HttpStatus.TOO\_MANY\_REQUESTS);

return exchange.getResponse().setComplete();

};

}

public static class Config {}

}

## **📡** 7. Discovery Registration

**build.gradle / pom.xml**

xml

CopyEdit

<dependency>

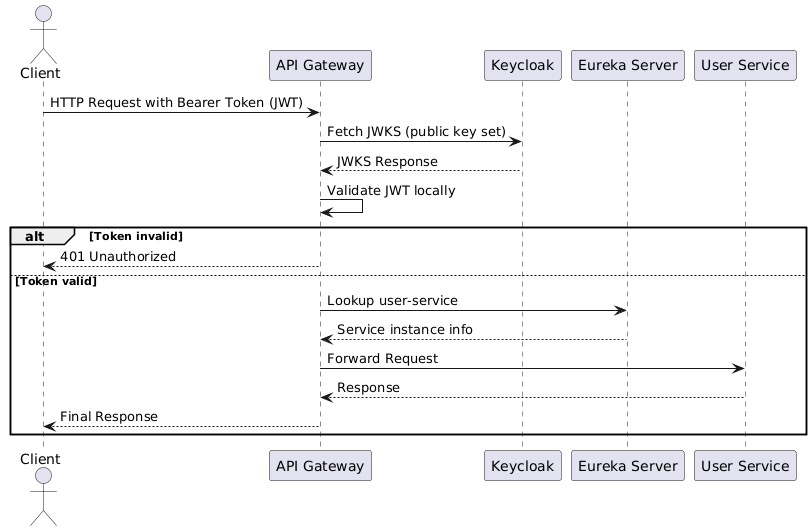
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

✅ Automatically registers this API Gateway with Eureka server.

**📶** 8. Call Flow Summary



### Explanation:

* The client sends a request with a JWT access token.
* The API Gateway fetches the **JWKS** once and caches it for local JWT validation.
* Upon successful validation:
  + It discovers the target microservice (e.g., user-service) via **Eureka**.
  + The request is routed and load-balanced to an instance of user-service.
* The response is forwarded back to the client.

**✅** LLD Deliverables

| **File/Module** | **Description** |
| --- | --- |
| SecurityConfig.java | Validates JWT from Keycloak |
| GatewayRoutes.java | Defines route rules |
| application.yml | Configuration for gateway, security, and discovery |
| RateLimiterFilter.java | Optional Bucket4j limiter per IP or user |
| pom.xml | Declares Spring Cloud Gateway, Security, Eureka, etc. |

## **9. Test Tools and Dependencies**

Add to pom.xml:

<dependency>

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

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

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-test</artifactId>

<scope>test</scope>

</dependency>

## 📁 10. Directory Structure

src/test/java/com/example/apigateway/

├── GatewayIntegrationTest.java

├── SecurityConfigTest.java

## ✅11. SecurityConfigTest.java

**Unit test for security configuration**

java

CopyEdit

package com.example.apigateway;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.context.annotation.Import;

import org.springframework.security.test.web.reactive.server.SecurityMockServerConfigurers;

import org.springframework.test.web.reactive.server.WebTestClient;

import com.example.apigateway.config.SecurityConfig;

@WebFluxTest

@Import(SecurityConfig.class)

public class SecurityConfigTest {

@Autowired

private WebTestClient webClient;

@Test

public void testUnauthenticatedAccessDenied() {

webClient.get().uri("/users")

.exchange()

.expectStatus().isUnauthorized();

}

@Test

public void testPublicPathAllowed() {

webClient.mutateWith(SecurityMockServerConfigurers.csrf())

.get().uri("/public/status")

.exchange()

.expectStatus().isOk(); // Assuming route is allowed

}

}

## 🔁 12. GatewayIntegrationTest.java

**Integration test for routing and token validation**

java

CopyEdit

package com.example.apigateway;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class GatewayIntegrationTest {

@Autowired

private WebTestClient webClient;

private final String fakeJwt = "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

+ "valid\_payload\_and\_signature";

@Test

public void testRouteWithToken() {

webClient.get().uri("/users")

.header("Authorization", fakeJwt)

.exchange()

.expectStatus().isOk(); // or .is5xxServerError() if downstream is not mocked

}

@Test

public void testMissingTokenFails() {

webClient.get().uri("/users")

.exchange()

.expectStatus().isUnauthorized();

}

}

🔧 For real token validation, you’ll want to mock Keycloak's /certs or use a real Keycloak instance with Testcontainers.

## 🧪 13. Other Optional Test Cases

| **Test Case** | **Description** |
| --- | --- |
| ✅ Token validation | Use Spring Security Test to simulate valid JWT |
| ✅ Rate limiting | Simulate bursts and verify 429 status |
| ✅ Routing | Ensure lb://user-service is hit for /users/\*\* |
| ✅ Public route | Whitelisted endpoints (e.g., /public/\*\*) |
| ✅ Downstream failure | Simulate user-service being unavailable |