# 1.Secure API Gateway:

## 1.1.Updating API-GW to Secure API-GW:

This update provides that only authenticated users can access customer-service and address-service via API Gateway. The flow is:

1. Clients send requests to API Gateway with a JWT token.
2. API Gateway validates the token using Keycloak
3. Only valid requests are forwarded to customer-service or address-service.
4. Unauthorized requests are rejected at the API Gateway level.

## 1.2.Adding the Dependencies to api-gateway:

|  |  |
| --- | --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>**spring-boot-starter-oauth2-resource-server**</artifactId>  </dependency> | It enables API Gateway to act as a resource server that validates JWT Tokens. |
| ~~<dependency>~~  ~~<groupId>org.springframework.cloud</groupId>~~  ~~<artifactId>~~**~~spring-cloud-starter-security~~**~~</artifactId>~~  ~~</dependency>~~ | ~~It enables Spring Security for API Gateway~~ |
| <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>**spring-security-oauth2-jose**</artifactId>  </dependency> | It handles JWT signature validation |

Since Security is now fully handled by “spring-boot-starter-oauth2-server” in Spring Boot 3.x and Spring Cloud 2023.x for resource servers (like API Gateway), spring-cloud-starter-security is an unnecessary dependency.

## 1.3.Updating application.properties:

Add below configuration to application.properties in API-GW:

|  |  |
| --- | --- |
| spring.security.oauth2.resourceserver.jwt.issuer-uri=**http://localhost:8080/realms/myrealm** | This is the URL of the Keycloak runs on localhost:8080 and the realm is “myrealm” API-GW uses this URL to validate the “iss” claim in JWT tokens. |
| spring.security.oauth2.resourceserver.jwt.jwk-set-uri=**http://localhost:8080/realms/myrealm/protocol/openid-connect/certs** | This is the enpoint Keycloak provides to verify JWT signatures. The gateway uses this endpoint to fetch public keys for validating JWTs. |

By these configurations, when a request with a JWT token reaches the API-GW:

1. API Gateway extracts the JWT from the **Authorization header**.
2. It sends the JWT token to **Keycloak’s issuer URI** to validate it.
3. If the JWT is valid, the request is **forwarded** to the correct microservice.
4. If it's invalid (expired, incorrect signature, etc.), API Gateway **blocks the request** with 401 Unauthorized.

|  |
| --- |
| NOTE: \*\* Related parameters are updated due to security vulnerabilities. There should be no sensitive data hardcoded. Check [SecurityInCWSMUP document](../../../SecurityInCWSMUP.docx). |

## 1.4.Adding SecurityConfig class:

Since SecurityConfig should have CORS settings, no need to have a class call “**CorsConfig**”.

Updated SecurityConfig class is:

|  |
| --- |
| package com.microservices.apigateway.config;  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 SecurityConfig {  @Bean  public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {  http  .csrf(ServerHttpSecurity.CsrfSpec::disable) // Disable CSRF for APIs  .authorizeExchange(exchanges -> exchanges  .pathMatchers("/customers/\*\*", "/addresses/\*\*").authenticated() // Secure microservices  .anyExchange().permitAll() // Allow all other requests  )  .oauth2ResourceServer(oauth2 -> oauth2.jwt()) // Enable JWT-based authentication  .cors(cors -> cors.configurationSource(request -> {  org.springframework.web.cors.CorsConfiguration config = new org.springframework.web.cors.CorsConfiguration();  config.addAllowedOrigin("\*"); // Allow all origins  config.addAllowedMethod("\*"); // Allow all HTTP methods  config.addAllowedHeader("\*"); // Allow all headers  config.addExposedHeader("Authorization"); // Expose Authorization header  return config;  }));  return http.build();  }  } |

# 2.Keycloak Configurations:

## 2.1.Creating a Realm:

**Realm:** customer-app-realm (since it is already created, no need to creat again)

If there is need for a new realm:

1. On the left menu, go to **Realm Settings**.
2. Click **Create Realm**.
3. **Enter the realm name**, e.g., *cs-new-realm*, and click **Save**.

## 2.2.Creating a client:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## 2.3.Configure Client Settings:

1. **Turn ON "Client authentication"** (if using confidential clients).
2. **Turn ON "Standard flow"** (for authorization code flow).
3. Set **Redirect URIs**: http://localhost:8082/\*
4. Click **Save**.

## 2.4.Get Client Credentials (For Backend Use):

1. In the **client settings page**, go to the **Credentials** tab.
2. Copy **Client ID** (customer🡺customer-service-client🡺Settings)
3. Copy **Client Secret**: IQMO1LDePpY3VtpA3dRBktD4UKNCXTqF (customer🡺customer-service-client🡺Credentials)

## 2.5.Create a User in Keycloak:

1. In the left menu, go to **Users** → Click **Add User**.
2. Fill in:
   * **Username**: testuser
   * **Email**: testuser@example.com
   * **First Name**: Test
   * **Last Name**: User
   * **Enable**: ✅ *(Keep enabled)*
3. Click **Save**.

A screenshot of a computer

Description automatically generated

## 2.6.Set Password for the User:

1. In the **Users** list, select testuser.
2. Go to the **Credentials** tab.
3. Click **Set Password**, enter a new password (e.g., testpassword), and click **Save**.

A screenshot of a computer

Description automatically generated

# 3.Keycloak Updates on API-GW:

|  |
| --- |
| spring.security.oauth2.resourceserver.jwt.issuer-uri=**http://localhost:8080/realms/customer-app-realm**  spring.security.oauth2.resourceserver.jwt.jwk-set-uri=**http://localhost:8080/realms/customer-app-realm/protocol/openid-connect/certs** |

In this configuration:

* + **issuer-uri:** Tells API-GW where to validate JWT tokens
  + **jwk-set-uri:** Provides the public key for token validation.

|  |
| --- |
| NOTE: \*\* Related parameters are stored as environment variables due to security vulnerabilities. There should be no sensitive data hardcoded. Check [SecurityInCWSMUP document](../../../SecurityInCWSMUP.docx). |

# 4.Refresh Token Mechanism:

To automate token refresh in the project, the authentication process should be needed to modify in a secure and scalable way.

## 4.1.Overview:

**Affected Microservices:** customer-service, address-service, api-gateway, logging-service(later)

Changes will be made in:

1. **SecurityConfig:** for token validation
2. **JwtAuthFilter:** for intercepting expired tokens
3. **AuthService:** for refreshing tokens
4. **RestTemplate/WebClient Configuration:** for secure API calls.

## 4.2.Storing Sensitive Data (Client Credentials):

Since all related parameters were hardcoded before, it’s better to store them as environment variables. This step is finished.

|  |
| --- |
| NOTE: \*\* Related parameters are stored as environment variables due to security vulnerabilities. There should be no sensitive data hardcoded. Check [SecurityInCWSMUP document](../../../SecurityInCWSMUP.docx). |

## 4.3.Adding a Customer JWT Filter (JwtAuthFilter):

JwtAuthFilter is a filter to intercepts requests, check token validity, and refresh it automatically if expired. In JwtAuthFilter class, it extracts the JWT token from request headers (“*String token = request.getHeader(HttpHeaders.****AUTHORIZATION****);*”). Later it checks if access\_token is expired (“***if*** *(****authService****.isTokenExpired(token)) {...}*”).

If it expired, it uses refresh\_token to get a new access\_token. Later, it updates request headers automatically. And:

|  |
| --- |
| **if** (**authService**.isTokenExpired(token)) {  String refreshToken = **authService**.getRefreshToken();**if** (refreshToken != **null**) {  String newAccessToken = **authService**.refreshAccessToken(refreshToken);  **if** (newAccessToken != **null**) {  response.setHeader(HttpHeaders.***AUTHORIZATION***, **"Bearer "** + newAccessToken);  token = newAccessToken; }  } } |

Finally, it passes authentication details to Spring Security:

|  |
| --- |
| SecurityContextHolder.*getContext*().setAuthentication(**authService**.getAuthentication(token)); filterChain.doFilter(request, response); |

## 4.4.Adding AuthService to Handle Token Refreshing:

AuthService centralizes the token handling, and ensures a single source of truth of authentication.

## 4.5.Adding SecurityConfig:

SecurityConfig intercepts every requests and applies JwtAuthFilter before authentication happens. It ensures role-based access control for admin routes. Automatically refreshes tokens when expired.

## 4.6.Enchancments of Implemented Refresh Token Mechanism:

|  |  |
| --- | --- |
| Security Concern | Solution Implemented |
| Hardcoded credentials | Uses application.properties + environment variables |
| Token leakage | Stores refresh tokens securely, not in code |
| Refresh Token Theft Risk | Only refreshes **if token is expired** |
| Unauthorized refresh attempts | Use @PreAuthorize("hasRole('ADMIN')") for admin endpoints |
| Unnecessary API calls | Prevents refresh if token is still valid |