Especialización en Back End III

Integrando Go con Keycloak en una arquitectura Spring Cloud

Vamos a crear un middleware para validar el JWT. Para el ejemplo, la función del middleware recibe el rol que debe contener el JWT para poder consumir el endpoint.

```
server := gin.Default()
server.Use(middleware.IsAuthorizedJWT("USER"))

usersRoute := server.Group("/users")
usersRoute.GET(":id", handler.FindById())
```

La función **IsAuthorizedJWT("USER")** validará con Keycloak el token y luego permitirá el acceso a la API en caso de recibir un token válido. Para conectarnos con Keycloak y validar el token, vamos a utilizar la librería **go-oidc**.

Podemos descargarla mediante:

go get github.com/coreos/go-oidc

Veamos más en detalle:

```
var RealmConfigURL string =
"http://localhost:8082/realms/digital-house"

// el clientId creado en Keycloak

var clientID string = "gateway"
```

```
//Structs necesarios para mapear el contenido del JWT

type Claims struct {
   ResourceAccess

client `json:"resource_access,omitempty"`
   JTI

          string `json:"jti,omitempty"`
}
type client struct {
   Gateway

clientRoles `json:"Gateway,omitempty"`
}
type clientRoles struct {
   Roles

[]string `json:"roles,omitempty"`
}
func IsAuthorizedJWT(role string) gin.HandlerFunc {
   return func(c

*gin.Context) {
      //Tomamos del header el token, recordemos que el formato
es “Bearer  {token}”

       rawAccessToken :=
strings.Replace(c.GetHeader("Authorization"),

"Bearer ", "", 1)

       tr :=

&http.Transport{
           TLSClientConfig:

&tls.Config{InsecureSkipVerify: true},
       }
       client :=

&http.Client{
           Timeout:

  time.Duration(6000) * time.Second,
```

```
            Transport:

tr,
        }
        ctx := oidc.ClientContext(context.Background(), client)




      //OpenId se conecta con el proveedor (Keycloak)

        provider, err := oidc.NewProvider(ctx,

RealmConfigURL)
        if err != nil {
            authorizationFailed("authorization failed while
getting the provider: "+err.Error(), c)
            return
        }

        oidcConfig :=

&oidc.Config{
            ClientID:

clientID,
        }


        //OpenId utiliza las claves públicas para validar
la firma del JWT

        verifier := provider.Verifier(oidcConfig)
        idToken, err := verifier.Verify(ctx,

rawAccessToken)
        if err != nil {
            authorizationFailed("authorization failed while
verifying the token: "+err.Error(), c)
            return
        }
      //Si el token es válido, mapeamos su contenido
        var IDTokenClaims

Claims
```

```
        if err := idToken.Claims(&IDTokenClaims);

err != nil {
            authorizationFailed("claims : "+err.Error(), c)
            return
        }
       //Obtenemos los roles que están asociados al cliente en
nuestro caso los roles están ubicados en "resource_access":
{"gateway":

{"roles": ["EDITOR","USER"]}}
        user_access_roles :=

IDTokenClaims.ResourceAccess.Gateway.Roles
        for _, b := range user_access_roles

{
        //si el token contiene el rol indicado, se le permite
acceder.

      if b == role {
                c.Next()
                return
            }
        }
        authorizationFailed("user not allowed to access this api",
c)
    }
}




 // Este método se encarga de retornar el error
func authorizationFailed(message string, c *gin.Context) {
    data :=

Res401Struct{
        Status:

   "FAILED",
        HTTPCode:

http.StatusUnauthorized,
```

```
       Message:
message,
   }

   c.AbortWithStatusJSON(200, gin.H{"response": data})

}
```

Configuración del API Gateway

Así será el **application.yml**:

```
# Configuraciones del servidor
server:
 port: 8090
# Configuraciones de Eureka
eureka:
 instance:
   hostname: localhost
   prefer-ip-address: true
 client:
   register-with-eureka: true
   fetch-registry: true
   serviceUrl:
     defaultZone: http://localhost:8761/eureka
# Configuraciones de Spring Cloud
```

```yaml
spring:
 application:
   name: ms-gateway
 security:
   oauth2:
     client:
       provider:
         api-gateway-service:
           issuer-uri: http://localhost:8082/realms/digital-house
       registration:
         api-gateway-service:
           provider: api-gateway-service
           client-id: gateway
           client-secret: a5946L5jX0YKydqNxay68LKKU2BxOnof
           authorization-grant-type: authorization_code
           redirect-uri:
'http://localhost:8090/login/oauth2/code/keycloak'
 cloud:
   gateway:
     default-filters:
       - TokenRelay
```

```
    routes:
      - id: users-service
        uri: lb://users-service
        predicates:
          - Path=/api/v1/users/**
        filters:
          - StripPrefix=2
```

Al microservicio creado en Go, lo registramos como **users-service** en Eureka.

Restringiendo todas las peticiones y activando el login de Keycloak

```
@Configuration
public class SecurityConfiguration {


 @Bean
 public SecurityWebFilterChain
springSecurityFilterChain(ServerHttpSecurity http) {


   return http.authorizeExchange().anyExchange().authenticated()
       .and()
```

```
        .oauth2Login()
        .and()
        .csrf().disable().build();


 }


 @Bean
 public JwtDecoder jwtDecoder() {
   return
NimbusJwtDecoder.withJwkSetUri("http://localhost:8082/realms/digi
tal-house/protocol/openid-connect/certs").build();
 }
}
```