

CONFIGURACIÓN DEL SERVIDOR DE AUTENTICACIÓN

Información general	
Duración estimada en minutos:	120
Docente:	Carlos Andrés Florez Villarraga
Guía no.	06

Información de la Guía

Un servidor de autenticación en microservicios es un componente que se encarga de validar la identidad de los usuarios que acceden a los servicios ofrecidos por una aplicación basada en microservicios.

La autenticación es un proceso crucial en cualquier sistema que maneje información confidencial o restringida, y es especialmente importante en un entorno de microservicios, donde múltiples servicios se comunican entre sí y comparten recursos.

Keycloak es una solución de servidor de autenticación y autorización de código abierto que puede utilizarse en un entorno de microservicios para proporcionar autenticación, autorización y gestión de identidades. Keycloak soporta varios protocolos estándar de autenticación y autorización, como OAuth2.0, OpenID Connect, SAML, y LDAP, lo que la hace una solución flexible y escalable.

Con Keycloak, se puede centralizar la gestión de identidades de los usuarios y permitir el acceso a diferentes aplicaciones y servicios a través de una única plataforma de autenticación. Además, Keycloak ofrece funcionalidades avanzadas de seguridad, como autenticación multifactor y gestión de sesiones, lo que la hace una solución robusta y segura para implementar en un entorno de microservicios.

El funcionamiento de keycloak puede ilustrarse en la siguiente arquitectura:

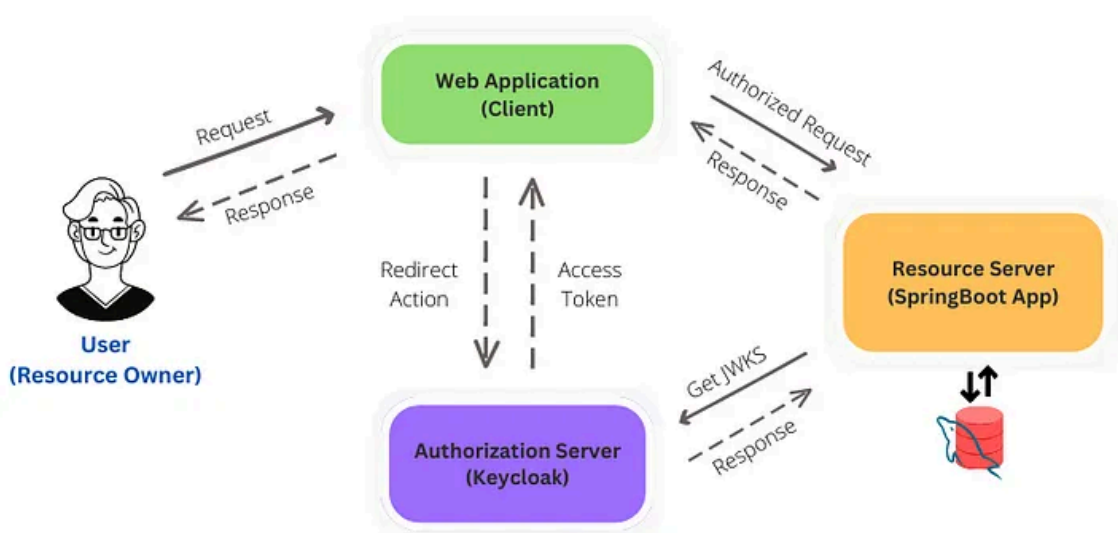


Imagen tomada de: <https://medium.com/geekculture/using-keycloak-with-spring-boot-3-0-376fa9f60e0b>

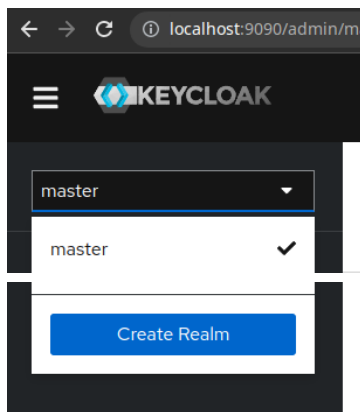
ACTIVIDAD

1. Crear un contenedor de docker que use la imagen `quay.io/keycloak/keycloak:21.0.2` así:

```
docker run -p 9090:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin -d --rm --volume keycloak_data:/opt/keycloak/data/ quay.io/keycloak/keycloak:21.0.2 start-dev
```

Se sugiere hacer uso de un volumen (keycloak_data) para que la configuración que vamos a hacer sobre el contenedor no se pierda cuando este se detenga. Además se está mapeando el puerto 8080 del contenedor al puerto 9090 del host (puede usar el de su preferencia).

2. Una vez se ejecute el contenedor, acceda a la consola de administración de Keycloak: <http://localhost:9090/admin> e inicie sesión con el usuario admin y contraseña admin.
3. Luego, cree un realm, en el menú izquierdo de click en master y luego vaya a "Create Realm".



En el formulario que aparece asígnele un nombre en "Realm name" y de click en "Create". Para esta guía el realm se llamará "tutorial-api". Un realm (reino) es una entidad de gestión que controla un conjunto de usuarios, sus credenciales, roles y grupos. Un usuario pertenece e inicia sesión en un reino. El realm es el punto central de configuración y gestión de nuestra aplicación.

4. Cree un nuevo cliente, en el menú de la izquierda elija la opción "Clients" y de click en el botón "Create client" y asigne los siguientes valores:

Client ID:	springboot-keycloak-client
Valid redirect URIs:	http://localhost:8080/ *
Web origins:	*
Authentication flow:	Habilitar la opción: Direct access grants

Los clientes son entidades que pueden solicitar a Keycloak la autenticación de usuarios. Un cliente representa un recurso al que pueden acceder determinados usuarios.

5. Una vez creado el cliente ahora vaya a la opción Roles y cree dos: "admin" y "user". Para crear el rol simplemente debe dar click en el botón "Create role" y asigne un nombre. Debe verse así:

The screenshot shows the Keycloak administration interface. On the left is a dark sidebar with a menu. The 'tutorial-api' realm is selected at the top. The 'Clients' menu item is highlighted. The main content area shows 'Clients > Client details' for 'springboot-keycloak-client'. Below this, there are tabs for 'Settings', 'Roles', 'Client scopes', 'Sessions', and 'Advanced'. The 'Roles' tab is active, displaying a search bar 'Search role by name' and a 'Create role' button. A table lists the roles for this client:

Role name	Composite
admin	False
user	False

Cada usuario debe tener un rol, ya que Keycloak usa el acceso basado en roles. Esto nos permite tener diferentes tipos de usuarios con diferentes permisos de usuario.

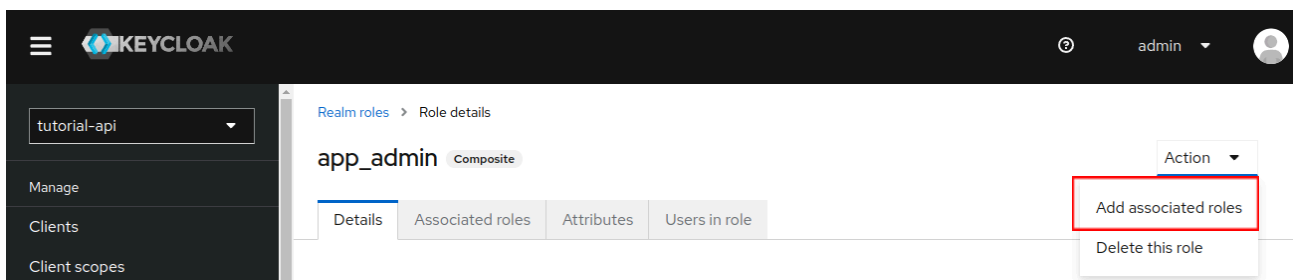
En Keycloak hay dos tipos de roles: Roles del cliente y roles del reino (realm roles), ambos se deben configurar para que todo funcione correctamente. En este punto primero creamos los roles del cliente.

6. Ingrese a la opción "Realm roles" (menú izquierdo) y cree dos roles: "app_admin" y "app_user". Una vez creados deben aparecer así en la tabla:

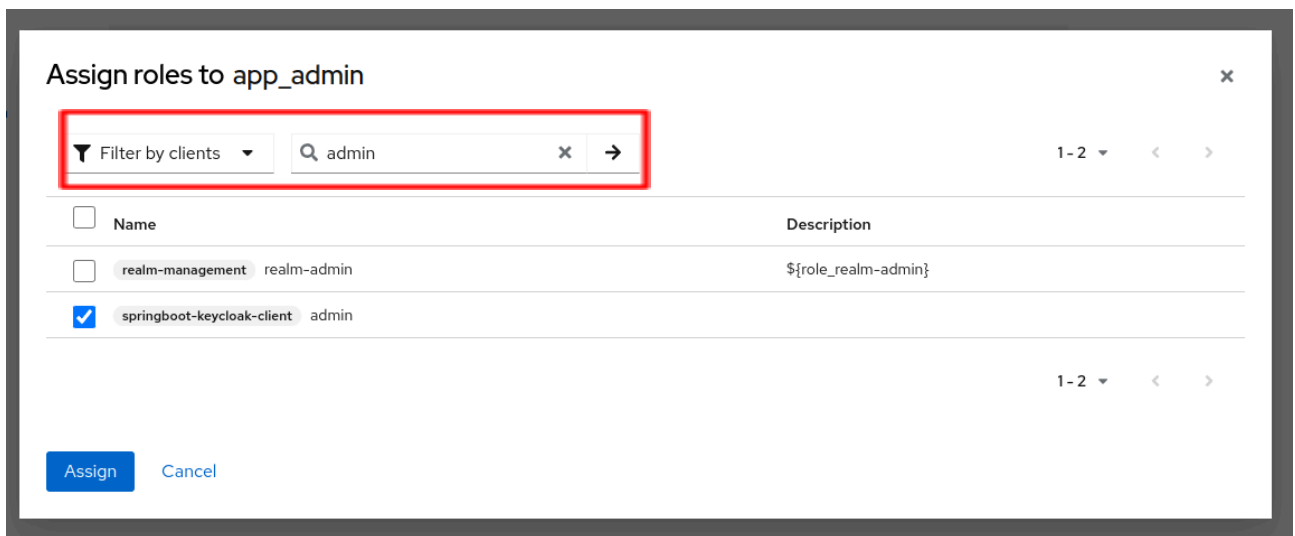
The screenshot shows the Keycloak administration interface with the 'Realm roles' tab selected in the sidebar. The main content area shows 'Realm roles' for 'tutorial-api'. Below this, there are tabs for 'Settings', 'Roles', 'Client scopes', 'Sessions', and 'Advanced'. The 'Roles' tab is active, displaying a search bar 'Search role by name' and a 'Create role' button. A table lists the realm roles:

Role name	Composite
app_admin	False
app_user	False
default-roles-tutorial-api	True
offline_access	False
uma_authorization	False

7. Ahora se debe asignar el rol adecuado del cliente al rol "app_admin" y "app_user" del reino.
8. De click en "app_admin" y dentro de dicha página, en la parte superior derecha hay una opción que dice Action:

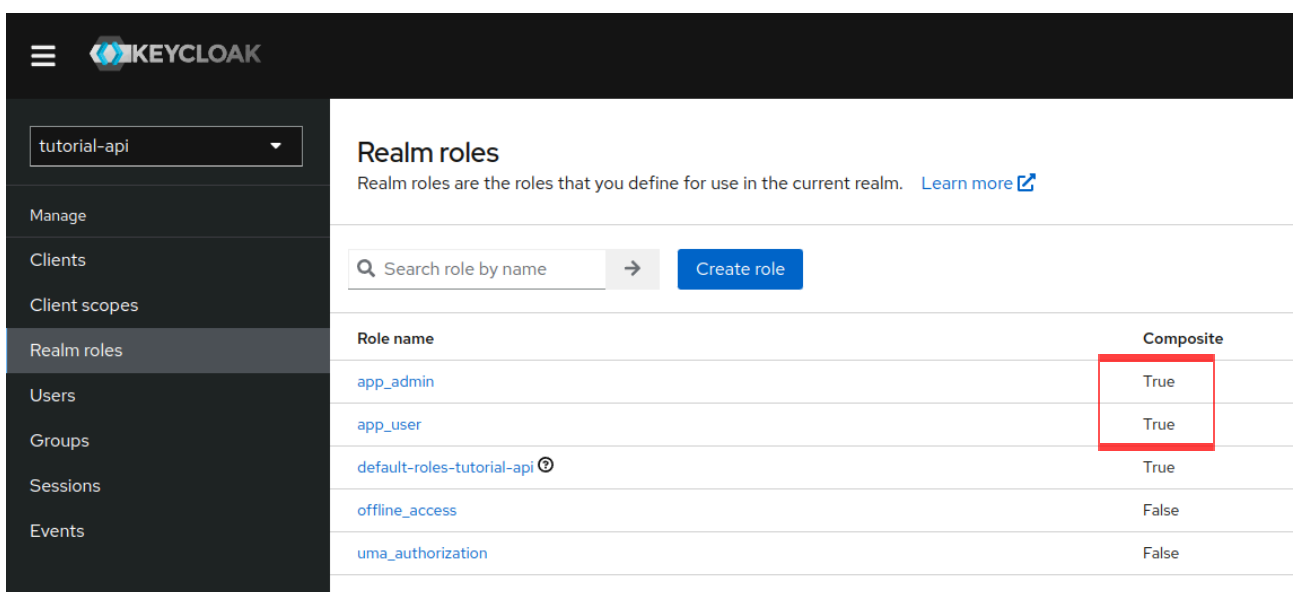


Debe dar click en la opción “Add associated roles”, en el modal que se muestra, debe seleccionar “Filter by clients” y en el cajón de al lado escriba “admin”. De la lista de roles que se muestran elija el que dice “springboot-keycloak-client admin” y de click en Assign.



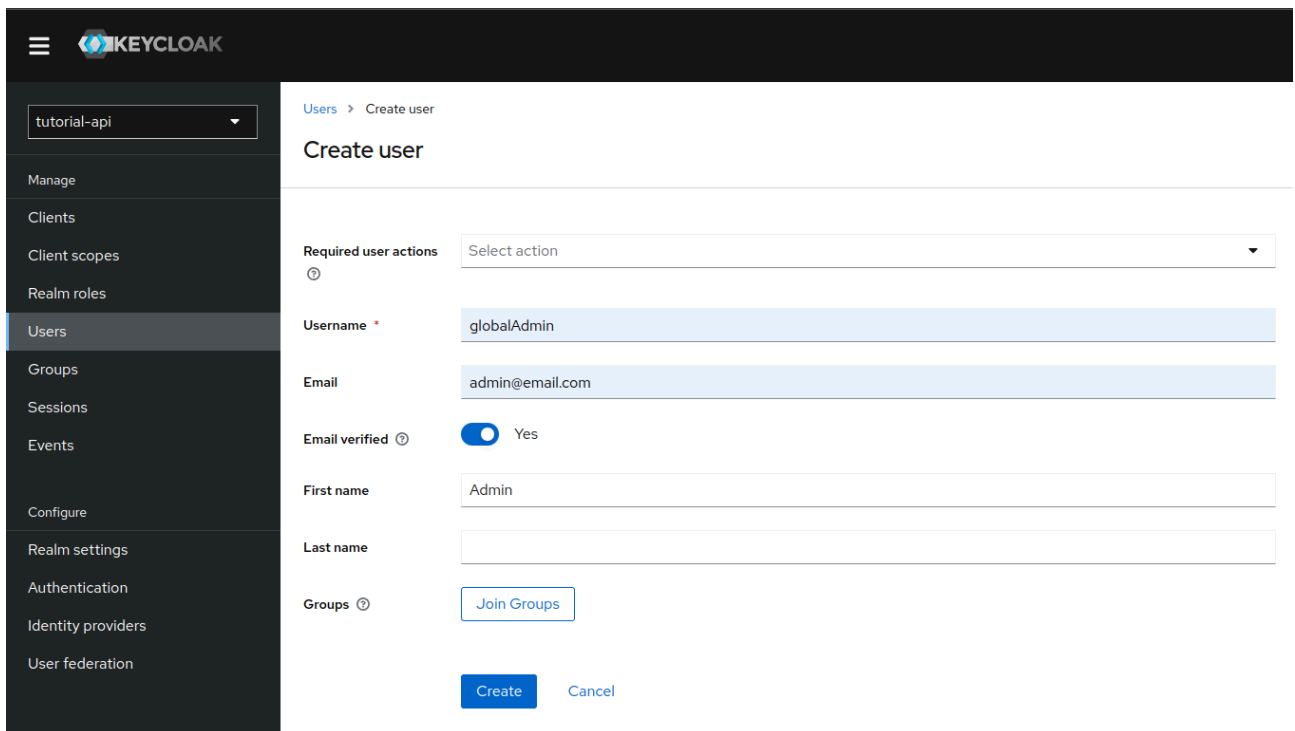
9. Haga lo mismo que en el punto anterior pero con el rol “app_user”, para este caso debe asignar el rol “springboot-keycloak-client user”.

10. Una vez asignados los roles, debe verse la lista así:



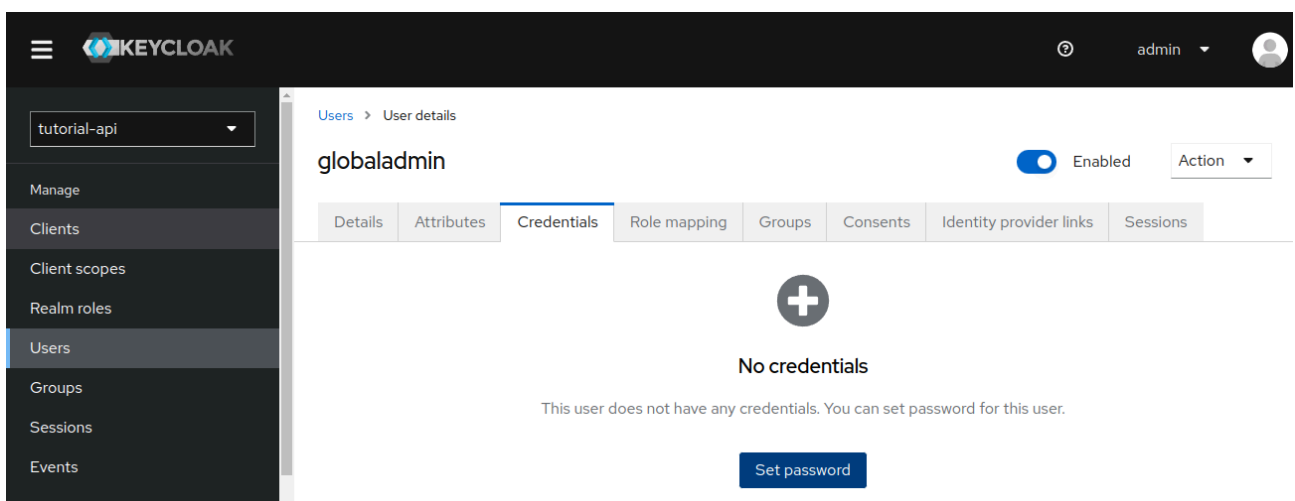
Fijese que en la columna donde dice “Composite” debe decir True para “app_admin” y “app_user”.

11. Finalmente, ahora vaya al menú Users (menú izquierdo) y cree un usuario administrador. Dé click en el botón “Add user” y asigne los siguientes valores:



The screenshot shows the Keycloak administration interface. On the left is a dark sidebar with a menu. The 'Users' option is highlighted. The main area is titled 'Create user'. It contains several form fields: 'Required user actions' (a dropdown menu), 'Username' (filled with 'globalAdmin'), 'Email' (filled with 'admin@email.com'), 'Email verified' (a toggle switch set to 'Yes'), 'First name' (filled with 'Admin'), 'Last name' (empty), and 'Groups' (a button labeled 'Join Groups'). At the bottom are 'Create' and 'Cancel' buttons.

De click en “Create” y en la siguiente pantalla vaya al tab “Credentials” y dé click en “Set password”.



The screenshot shows the 'User details' page for a user named 'globaladmin'. The 'Credentials' tab is selected. The page shows a large plus icon and the text 'No credentials'. Below this, it says 'This user does not have any credentials. You can set password for this user.' and there is a 'Set password' button. The top right of the page shows the user is logged in as 'admin'.

En el modal que aparece, ingresa como password la contraseña que desee (en esta guía se sugiere asignar password) y además verifique que donde dice “Temporary” esté en Off. Y seleccione “Save”.

Set password for globaladmin

Password *

Password confirmation *

Temporary ☐ Off

[Save](#) [Cancel](#)

12. Por último, debe asignar los roles al usuario "globaladmin", para este usuario particular, tendremos que asignar dos roles: "app_admin" y "manage-users". Vaya al tab "Role mapping" y debe ver algo así:

Users > User details

globaladmin Enabled Action

Details Attributes Credentials **Role mapping** Groups Consents Identity provider links Sessions

☒ Hide inherited roles [Assign role](#) [Unassign](#) 1-1

<input type="checkbox"/> Name	Inherited	Description
<input type="checkbox"/> default-roles-tutorial-api	False	\${role_default-roles}

1-1

Una vez en esta página, debe dar click en el botón "Assign role". En el modal que se muestra debe elegir el rol "app_admin" y dar click en Assign.

Assign roles to globaladmin

1-4

<input type="checkbox"/> Name	Description
<input checked="" type="checkbox"/> app_admin	
<input type="checkbox"/> app_user	
<input type="checkbox"/> offline_access	\${role_offline-access}
<input type="checkbox"/> uma_authorization	\${role_uma_authorization}

1-4

[Assign](#) [Cancel](#)

Para asignar el otro rol, volvemos a dar click en "Assign role", filtramos usando "Filter by clients" y en el cajón que está al lado escribimos: "manage-users". Elegimos "realm-management manage-users" y damos click en "Assign".

Assign roles to globaladmin
✕

Filter by clients
✕
→
1-1

<input checked="" type="checkbox"/> Name	Description
<input checked="" type="checkbox"/> realm-management manage-users	\${role_manage-users}

1-1

Assign
Cancel

Una vez elegidos ambos roles debe verse así:

Details
Attributes
Credentials
Role mapping
Groups
Consents
Identity provider links
Sessions

☒ Hide inherited roles
Assign role
Unassign

<input type="checkbox"/> Name	Inherited	Description
<input type="checkbox"/> app_admin	False	—
<input type="checkbox"/> default-roles-tutorial-api	False	\${role_default-roles}
<input type="checkbox"/> realm-management manage-users	False	\${role_manage-users}

- Cree un nuevo usuario siguiendo los pasos anteriores (paso 11 y 12), pero a este usuario sólo asígnele el rol “app_user”.
- Una vez hecho todos los pasos anteriores, ya tenemos un servidor de autenticación listo con dos roles (admin y user). Tenga en cuenta que se pueden crear tantos roles como sean necesarios.

Vaya a la página:

<http://localhost:9090/realms/tutorial-api/.well-known/openid-configuration>

Allí se lista la configuración de los endpoints del servidor.

- Para probar que los tokens sí se generan, pruebe (usando Postman) el endpoint token_endpoint de keycloak:

<http://localhost:9090/realms/tutorial-api/protocol/openid-connect/token>

Debe enviar una petición POST y en el body de dicha petición debe enviar los siguientes datos usando (x-www-form-urlencoded):

grant_type	password
client_id	springboot-keycloak-client

username	globaladmin
password	password

Si los datos son correctos se debe obtener una respuesta con los siguientes datos: access_token, expires_in, refresh_expires_in, refresh_token, token_type, not-before-policy, session_state y scope. De todos estos valores solo nos interesan dos: access_token y refresh_token.

16. Copie el access_token en la página <https://jwt.io/> y observe el contenido del payload.
17. Pruebe el endpoint anterior pero haciendo uso del usuario que creó en el punto 13. Observe la sección springboot-keycloak-client roles del json.
18. Keycloak alojará los usuarios que se registren en nuestro proyecto y se encargará de hacer el proceso de autenticación. Para esto, se debe revisar la API Rest de Keycloak ya que desde nuestro proyecto de spring boot será necesario crear un microservicio que se comunique con esa API y así poder registrar usuarios y validar sus credenciales de inicio de sesión

Para más información: <https://www.keycloak.org/docs-api/15.0/rest-api/index.html>

SERVIDOR DE RECURSOS

En una arquitectura OAuth2, un servidor de recursos es el servidor que contiene los recursos protegidos y requiere autenticación y autorización para acceder a ellos. Cuando una aplicación necesita acceder a un recurso protegido en un servidor de recursos, primero debe solicitar un token de acceso de un servidor de autorización OAuth2. Este token de acceso se utiliza para autenticar la solicitud de la aplicación y autorizar el acceso a los recursos protegidos. Luego, la aplicación puede enviar el token de acceso junto con la solicitud al servidor de recursos, que valida el token y, si es válido, devuelve el recurso solicitado.

Flujo de protocolo abstracto



Imagen tomada de: <https://www.digitalocean.com/community/tutorials/una-introduccion-a-oauth-2-es>

ACTIVIDAD

1. Agregue la siguiente dependencia al archivo `build.gradle` del proyecto Gateway:

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
```

2. En el archivo `application.properties` de dicho proyecto agregue la siguiente configuración:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:9090/realms/tutorial-api
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:9090/realms/tutorial-api/protocol/openid-connect/certs

jwt.auth.converter.resource-id=springboot-keycloak-client
jwt.auth.converter.principal-attribute=preferred_username

logging.level.org.springframework.security=DEBUG
```

Las dos primeras propiedades que observamos en el código anterior son propiedades de configuración que Spring Security utiliza para especificar la URI del emisor del token JWT en un servidor de recursos OAuth2. Esto se utiliza para garantizar que el token presentado sea auténtico y válido según el servidor de autenticación (en nuestro caso Keycloak).

Las otras dos propiedades son para acceder adecuadamente a algunos parámetros de Keycloak. Y la última no es necesaria, simplemente es para habilitar el log de Spring Security y observar en la consola lo que va ocurriendo mientras probamos.

3. Cree un nuevo paquete que se llame security (co.edu.eam.biblioteca.security), y agregue la siguiente clase:

```
package co.edu.eam.biblioteca.security;

import lombok.RequiredArgsConstructor;
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;

@RequiredArgsConstructor
@Configuration
@EnableWebFluxSecurity
public class WebSecurityConfig {

    public static final String ADMIN = "admin";
    public static final String USER = "user";
    private final JwtAuthConverter jwtAuthConverter;

    @Bean
    public SecurityWebFilterChain securityFilterChain(ServerHttpSecurity http) {

        http
            .authorizeExchange( e ->
                e.pathMatchers("/api/test/anonymous").permitAll()
                .pathMatchers("/api/test/admin").hasRole(ADMIN)
                .pathMatchers("/api/test/user").hasAnyRole(ADMIN, USER)
                .anyExchange().authenticated());

        http.oauth2ResourceServer()
            .jwt()
            .jwtAuthenticationConverter(jwtAuthConverter);
        http.csrf().disable();
        return http.build();
    }
}
```

Dado que Spring Cloud Gateway no usa un servidor HTTP convencional como Tomcat sino que usa un servidor HTTP reactivo (Netty), entonces algunas clases vienen de los paquetes reactive y reactor de Java, en Spring esto se logra con WebFlux. WebFlux es un módulo de Spring Framework que se utiliza para construir aplicaciones web reactivas y escalables en Java.

WebFlux se basa en dos tipos principales de objetos: Flux y Mono. Flux representa un flujo de datos que puede emitir cero o más elementos, mientras que Mono representa un flujo de datos que puede emitir cero o un solo elemento. Estos flujos de datos se utilizan para manejar operaciones asincrónicas, como la lectura o escritura de datos en una base de datos o la llamada a una API externa.

De todas maneras, Spring Security (y OAuth2 Resource Server) funcionan sin problemas en servidores HTTP reactivos. Por lo tanto, solo debemos preocuparnos de definir las rutas que

estarán protegidas y qué roles tienen la autorización para accederlas (estos roles ya fueron definidos previamente en Keycloak).

En este caso, tenemos una ruta que puede ser accedido por cualquier usuario (con o sin autenticación), otra que solo puede ser accedido por un usuario con el rol “admin” y otra ruta que puede ser accedida tanto por el rol “admin” como por el rol “user”.

4. En ese mismo paquete cree una nueva clase así:

```
package co.edu.eam.biblioteca.security;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.convert.converter.Converter;
import org.springframework.security.authentication.AbstractAuthenticationToken;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.oauth2.jwt.Jwt;
import org.springframework.security.oauth2.jwt.JwtClaimNames;
import org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationToken;
import org.springframework.security.oauth2.server.resource.authentication.JwtGrantedAuthoritiesConverter;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;

import java.util.Collection;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Component
public class JwtAuthConverter implements Converter<Jwt, Mono<AbstractAuthenticationToken>> {

    private final JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter = new
    JwtGrantedAuthoritiesConverter();

    @Value("${jwt.auth.converter.resource-id}")
    private String resourceId;

    @Value("${jwt.auth.converter.principal-attribute}")
    private String principalAttribute;

    @Override
    public Mono<AbstractAuthenticationToken> convert(Jwt jwt) {
        Collection<GrantedAuthority> authorities = Stream.concat(
            jwtGrantedAuthoritiesConverter.convert(jwt).stream(),
            extractResourceRoles(jwt).stream()).collect(Collectors.toSet());
        return Mono.just( new JwtAuthenticationToken(jwt, authorities,
        getPrincipalClaimName(jwt)));
    }

    private String getPrincipalClaimName(Jwt jwt) {
        String claimName = JwtClaimNames.SUB;
        if (principalAttribute != null) {
            claimName = principalAttribute;
        }
        return jwt.getClaim(claimName);
    }

    private Collection<? extends GrantedAuthority> extractResourceRoles(Jwt jwt) {
        Map<String, Object> resourceAccess = jwt.getClaim("resource_access");
        Map<String, Object> resource;
        Collection<String> resourceRoles;
```

```

        if (resourceAccess == null
            || (resource = (Map<String, Object>) resourceAccess.get(resourceId)) == null
            || (resourceRoles = (Collection<String>) resource.get("roles")) == null) {
            return Set.of();
        }
        return resourceRoles.stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
            .collect(Collectors.toSet());
    }
}

```

Este código es una implementación de un convertidor de autenticación en Spring Security que se utiliza para convertir un objeto JWT en un objeto de autenticación `AbstractAuthenticationToken`. Utiliza la biblioteca Reactor para manejar la conversión de forma asíncrona y utiliza algunas anotaciones de Spring para inyectar valores de configuración desde un archivo de propiedades.

La clase implementa la interfaz `Converter<Jwt, Mono<AbstractAuthenticationToken>>`, lo que significa que puede convertir un objeto `Jwt` en un objeto de autenticación `AbstractAuthenticationToken`, utilizando el método `convert()` de la interfaz.

Dentro del método `convert()`, el convertidor primero llama a un objeto `JwtGrantedAuthoritiesConverter` para obtener las autoridades del token JWT y luego extrae los roles de recursos del token utilizando el método `extractResourceRoles()`. Luego, se combinan las autoridades del token y los roles de recursos en un conjunto de autoridades y se utiliza para crear un objeto `JwtAuthenticationToken`, que es un subtipo de `AbstractAuthenticationToken`.

5. Cree un controlador de ejemplo en el proyecto Gateway para que retorne los datos del usuario que están codificados en el token. La clase puede verse así:

```

package co.edu.eam.biblioteca.controller;

import
org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationToken;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

import java.security.Principal;

@RestController
@RequestMapping("/api/test")
public class GatewayController {

    @GetMapping("/anonymous")
    public Mono<String> getAnonymous(){
        return Mono.just("Hola anónimo!");
    }

    @GetMapping("/admin")
    public Mono<String> getAdmin(Principal principal){
        JwtAuthenticationToken token = (JwtAuthenticationToken) principal;
        String username = (String) token.getTokenAttributes().get("name");
        String email = (String) token.getTokenAttributes().get("email");

        return Mono.just("Hola admin! "+username+" "+email);
    }
}

```

```

@GetMapping("/user")
public Mono<String> getUser(Principal principal){
    JwtAuthenticationToken token = (JwtAuthenticationToken) principal;
    String username = (String) token.getTokenAttributes().get("name");
    String email = (String) token.getTokenAttributes().get("email");

    return Mono.just("Hola usuario! "+username+" "+email);
}
}

```

Como se mencionó anteriormente, dado que Spring Cloud Gateway usa Netty como servidor HTTP entonces usa WebFlux, por lo tanto, lo ideal es que cada método del controlador retorne un `Mono` o un `Flux`. En este ejemplo solo se retorna un `String` con un mensaje según el tipo de rol. A partir de este objeto `JwtAuthenticationToken` (que funciona gracias al converter hecho en el punto anterior), se obtienen los atributos del token mediante el método `getTokenAttributes()` y se extraen el nombre y el correo electrónico del usuario.

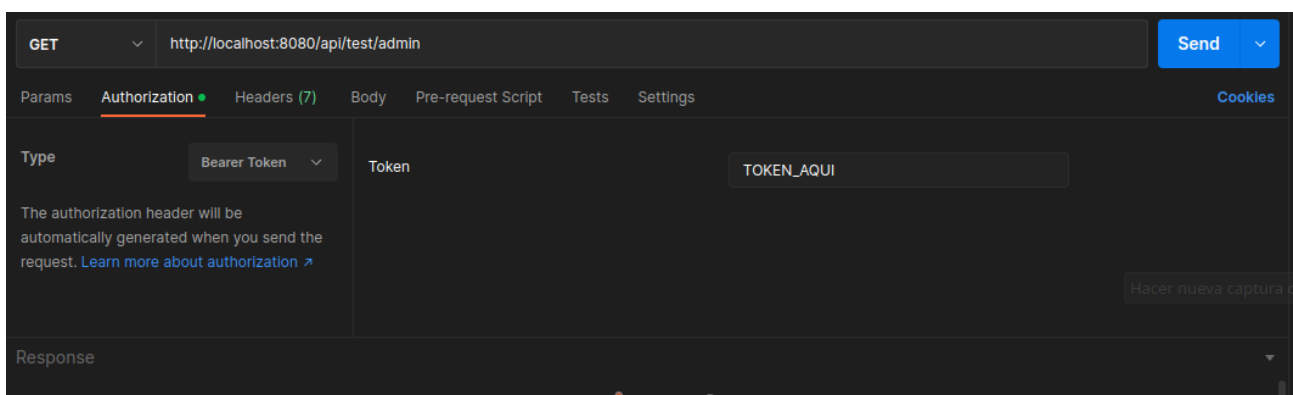
Tenga en cuenta que este controlador es solo de prueba, ya que los microservicios tienen sus propios endpoints (configurados en sus propios controladores), y Gateway simplemente hace la comprobación de autenticación y autorización, y hace el llamado al microservicio adecuado según la petición del cliente.

6. Pruebe cada endpoint hecho en el punto anterior (`/api/test`). Para este caso puede solo ejecutar el servidor de Eureka y el proyecto de Gateway (no es necesario que ejecute Clientes, Libros, Préstamos).

Para el anónimo simplemente envíe una petición GET en Postman al siguiente endpoint: <http://localhost:8080/api/test/anonymous>

Para los otros dos endpoints es necesario el token. Envíe una petición a keycloak en su endpoint <http://localhost:9090/realms/tutorial-api/protocol/openid-connect/token> como se explicó anteriormente. Use el access token devuelto para enviarlo dentro del encabezado de autenticación y así acceder a los endpoints: <http://localhost:8080/api/test/user> y <http://localhost:8080/api/test/admin>.

En Postman debe ir a la opción Authorization, elegir Bearer Token (en el cajón de selección de la izquierda) y pegar el token en el cajón de texto que se habilita.



Pruebe enviando un token de rol “user” para acceder a <http://localhost:8080/api/test/admin> y luego pruebe enviando un token de “admin” y compruebe qué pasa en cada caso.

NOTA. Si está ejecutando keycloak desde `docker-compose`, debe modificar el archivo `hosts` de su computador. Agregue lo siguiente: `127.0.0.1 keycloak`.

Para modificar el archivo leer esto: <https://www.hostinger.co/tutoriales/editar-archivo-hosts>

Y en postman cambie <http://localhost:9090/realms/tutorial-api/protocol/openid-connect/token> por <http://keycloak:9090/realms/tutorial-api/protocol/openid-connect/token>

De igual manera, en el archivo `docker-compose` debe agregar estas variables de entorno al servicio de gateway para que apunten correctamente al servicio de keycloak:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://keycloak:8080/realms/tutorial-api
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://keycloak:8080/realms/tutorial-api/
protocol/openid-connect/certs
```

Con estas modificaciones pruebe nuevamente los endpoints.

7. Modifique el método `securityFilterChain` de la clase `WebSecurityConfig` para que proteja el acceso a los endpoints de Libros, Clientes y Préstamos. Haga que para acceder a cualquier endpoint el usuario esté autenticado, en el caso de clientes y libros solo accedan los que tengan el rol "admin", en el caso de préstamos que sea un usuario con el rol "admin" o "user".
8. Cree un nuevo módulo (microservicio) que se llame `autenticacion` y agregue las siguientes dependencias:

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'co.edu.eam'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2022.0.1")
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.cloud:spring-cloud-starter-openfeign'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

```

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

tasks.named('test') {
    useJUnitPlatform()
}

```

9. El archivo `application.properties` de este nuevo microservicio debe quedar así:

```

spring.application.name=auth-service

server.port=0
eureka.instance.instance-id=${spring.application.name}:${random.value}
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

keycloak.admin-url: http://localhost:9090/admin/realms/tutorial-api
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:9090/realms/tutorial-api

```

10. La clase principal (donde está el `main`) debe tener la anotación: `@EnableFeignClients`. Feign es una biblioteca que simplifica la comunicación entre microservicios en una arquitectura basada en la nube.

11. Haga que este microservicio tenga tres endpoints en la ruta `"/api/auth"`:

```

@PostMapping("/signin")
public ResponseEntity<Respuesta<TokenDTO>> login(@RequestBody LoginDTO loginDTO) throws
Exception{
    return ResponseEntity.status(HttpStatus.OK).body(new Respuesta<>("Login correcto",
loginServicio.login(loginDTO)) );
}

@PostMapping("/refresh")
public ResponseEntity<Respuesta<TokenDTO>> refresh(@RequestBody TokenDTO token) throws
Exception{
    return ResponseEntity.status(HttpStatus.OK).body(new Respuesta<>("",
loginServicio.refresh(token)) );
}

@PostMapping("/signup")
public ResponseEntity<Respuesta<String>> createUser(@RequestBody NewUserDTO newUserDTO) throws
Exception{
    return ResponseEntity.status(HttpStatus.CREATED).body(new
Respuesta<>(loginServicio.createUser(newUserDTO) ? "Creado correctamente": "Error", ""));
}

```

La idea es que `LoginServicio` se encargue de hacer login (usando keycloak) y registrar usuarios (en Keycloak).

IMPORTANTE: Recuerde modificar el microservicio de `gateway` para incluir estas nuevas rutas. Haga que cualquier persona pueda acceder a `"/api/auth/**"` desde `WebSecurityConfig`.

12. Por ejemplo, para hacer el login (inicio de sesión) simplemente puede hacer una petición al endpoint <http://localhost:9090/realms/tutorial-api/protocol/openid-connect/token> (como hicimos previamente) con los datos necesarios. Para esto puede usar `FeignClient`.

Cree una clase que se llame `TokenFeignClient` en el paquete `co.edu.eam.biblioteca.feignclients`. Haga que quede así:

```
package co.edu.eam.biblioteca.feignclients;

import co.edu.eam.biblioteca.dto.TokenResponseDTO;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;

@FeignClient(name = "TokenKeycloak", url =
"${spring.security.oauth2.resourceserver.jwt.issuer-uri}")
public interface TokenFeignClient {

    @PostMapping(
        path = "/protocol/openid-connect/token",
        consumes = {MediaType.APPLICATION_FORM_URLENCODED_VALUE}
    )
    ResponseEntity<TokenResponseDTO> sendRequest(String ruta);
}
```

Observe que `TokenFeignClient` tiene un método `sendRequest`, el cual realiza la petición que hicimos en Postman para generar el token del usuario. Pero ahora es el microservicio de autenticación el encargado de hacerla por nosotros.

13. Cree el DTO `TokenResponseDTO` con los siguientes datos (estos datos son los que devuelve Keycloak al hacer la petición del token):

```
package co.edu.eam.biblioteca.dto;

public record TokenResponseDTO(
    String access_token,
    int expires_in,
    int refresh_expires_in,
    String refresh_token,
    String token_type,
    String session_state,
    String scope
) {
}
```

Cree el DTO `TokenDTO` y `LoginDTO` con los atributos que sean pertinentes.

14. Cree la clase `LoginServicio` en el paquete `co.edu.eam.biblioteca.service` y agregue lo siguiente:

```
package co.edu.eam.biblioteca.service;

@Service
@RequiredArgsConstructor
public class LoginServicio {

    private final TokenFeignClient tokenFeignClient;
```



```

public TokenDTO login(LoginDTO loginDTO){

    String clientID = "springboot-keycloak-client";
    String usernameAdmin = loginDTO.username();
    String passwordAdmin = loginDTO.password();
    String grantType = "password";

    String requestBody = String.format("client_id=%s&username=%s&password=%s&grant_type=%s",
clientID, usernameAdmin, passwordAdmin, grantType);

    ResponseEntity<TokenResponseDTO> responseEntity =
tokenFeignClient.sendRequest(requestBody);

    if(responseEntity.getStatusCode() == HttpStatus.OK) {
        TokenResponseDTO response = responseEntity.getBody();
        return new TokenDTO( response.access_token(), response.refresh_token() );
    }

    throw new RuntimeException("Error en la petición "+responseEntity.getStatusCode());

}

}

```

15. Dado que el microservicio de autenticación tiene la dependencia de spring-security entonces debe crear una clase `WebSecurityConfig` muy parecida a la que se hizo en el microservicio de gateway. Pero en el microservicio de autenticación haga que todas las peticiones sean permitidas.
16. Agregue el nuevo microservicio a docker-compose (también agregue keycloak), ejecute todo y pruebe el funcionamiento. Haga peticiones al microservicio de autenticación y pruebe que el login funciona correctamente.
17. Investigue cómo implementar el servicio de registro y de refresh token.

Para el registro: https://www.keycloak.org/docs-api/21.0.1/rest-api/index.html#_users_resource