

## Especialización en Back End II

# Integrando Keycloak con Spring Boot

En este tutorial paso a paso, veremos cómo utilizar Keycloak, una poderosa herramienta de gestión de identidad y acceso, junto con Spring Boot, un marco de desarrollo rápido para aplicaciones Java, para implementar la seguridad en una aplicación web.

Exploraremos características de **Keycloak**, como ser la gestión de usuarios y grupos, la creación y configuración de una app de Spring Boot para poder ser integrada con Keycloak y luego una prueba de la correcta implementación mediante **Postman**.

## Índice

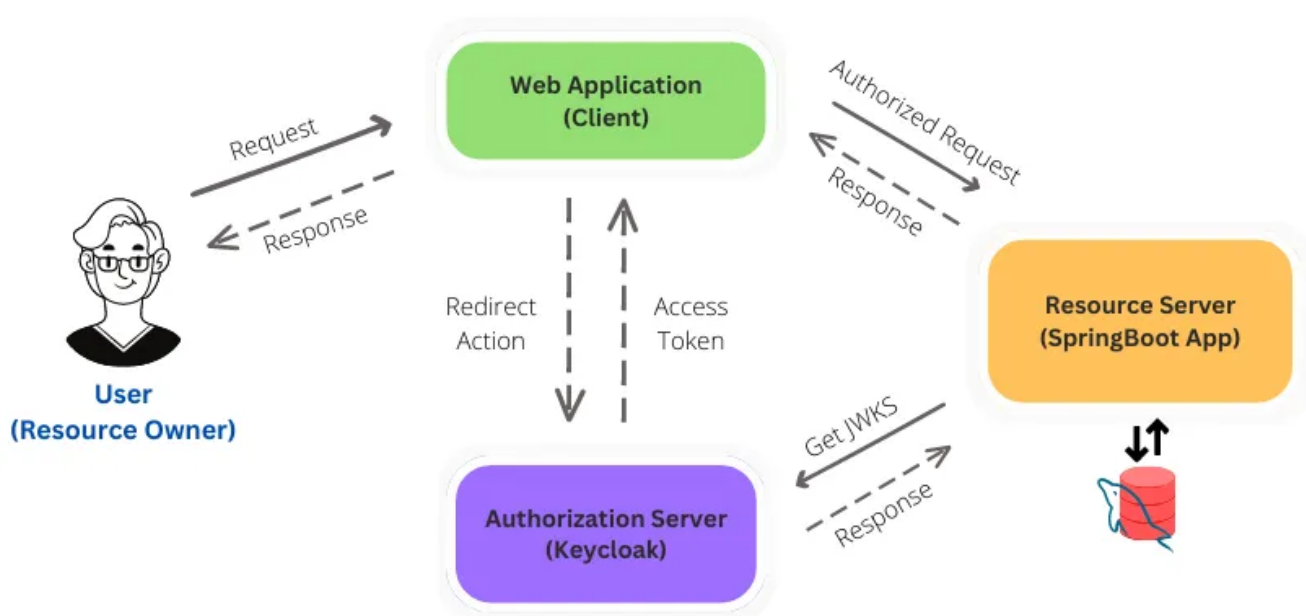
- [Repaso Flujo de autenticación de OAuth 2](#)
- [1era Parte - Configurando Keycloak](#)
- [2da Parte - Keycloak + Spring Boot](#)
- [3ra Parte - Probando con Postman](#)

# Repaso Flujo de autenticación de OAuth 2

Keycloak implementa casi todos los protocolos estándar de IAM, como OAuth 2.0, OpenID y SAML. Por lo tanto, podemos usar uno de estos protocolos para conectarnos con Keycloak.

La mejor forma es usar directamente una biblioteca OAuth para integrar Keycloak con nuestra aplicación. De esta manera, no se necesita ninguna biblioteca o adaptador de cliente específico de Keycloak para comunicarse con él.

Se puede entender el esquema estándar de autenticación **OAuth2** de tres pasos a través del siguiente diagrama:

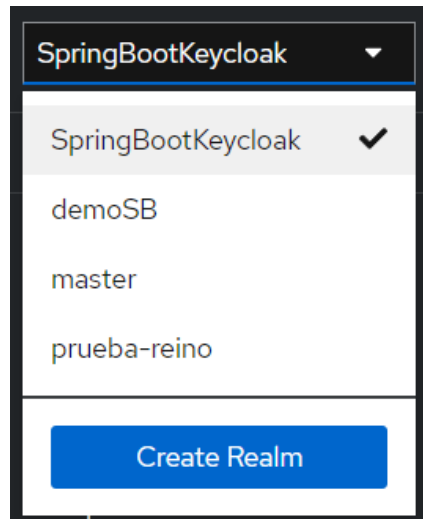


Habiendo repasado esto, veamos cómo podemos interconectar una aplicación desarrollada en Spring Boot con Keycloak.

¡Manos a la obra!

# 1era Parte - Configurando Keycloak

**Paso 1:** Vamos a crear un nuevo reino en nuestro panel de administración de Keycloak. Vamos a llamarlo **SpringBootKeycloak**. Una vez creado vamos a seleccionarlo como activo.



**Paso 2:** Vamos a crear un cliente. Recordemos que por cada aplicación que usemos debemos crear un nuevo cliente. Por lo que vamos al apartado **Clients** y vamos a crear uno nuevo con el nombre **springboot-keycloak-client** en el proceso N° 1. En los procesos 2 y 3 dejamos todas las configuraciones por defecto sin realizar cambios.

## Create client

Clients are applications and services that can request authentication of a user.

1 General Settings

2 Capability config

3 Login settings

Client type ?OpenID Connect

Client ID \* ?springboot-keycloak-client

Name ?

Description ?

Always display in UI ?☐ Off

Imagen: Paso 1 de creación de cliente

Acá es importante recordar que, al trabajar como localhost por defecto, **Keycloak** se va a estar ejecutando en nuestro **puerto 8080**, por lo que es recomendable que configuremos luego nuestra app de **Spring Boot** para que se levante en otro puerto, por ejemplo, el **8081**. Esto debemos especificarlo en el cliente

Para configurar la URI de nuestra app de Spring Boot, vamos a ir a nuestro cliente recién creado, y en el apartado: “**Valid redirect URIs**” colocaremos la URL <http://localhost:8081/> (8081 se puede reemplazar por el número de puerto que sea utilizado para la app de SpringBoot).

### General Settings

Client ID \* ⓘ

Name ⓘ

Description ⓘ

Always display in UI ⓘ ☐ Off

### Access settings

Root URL ⓘ

Home URL ⓘ

Valid redirect URIs ⓘ  ⓘ

[+ Add valid redirect URIs](#)

**Paso 3:** Ahora toca crear los roles. Recordemos que principalmente hay 2 tipos de roles en **Keycloak**.

- **Rol de Reino:** Es un rol global, perteneciente a ese reino específico. Este rol puede ser accedido desde cualquier cliente y asignarse a cualquier usuario.
- **Rol de Cliente:** Es un rol que pertenece solo a un cliente en específico. Estos roles no pueden ser accedidos desde otro cliente.

Un rol puede **combinarse con varios roles**. Entonces se convierte en un **Rol Compuesto**

Primero vamos a crear los roles exclusivos del cliente primero. Iremos al apartado **Cientes**, seleccionaremos nuestro cliente **springboot-keycloak-client** y en la pestaña Roles vamos a crear dos roles, uno user y otro admin, tal como en la siguiente imagen:

springboot-keycloak-client

OpenID Connect

Clients are applications and services that can request authentication of a user.

Settings

Roles

Client scopes

Sessions

Advanced

Search role by name

→

Create role

Role name	Composite
admin	False
user	False

Hecho esto, ahora tenemos que crear los roles a nivel de reino. Para ello vamos a ir al Menú **“Realm roles”**. Vamos a crear los mismos dos roles, un rol llamado **app\_user** (para los usuarios “comunes” de un sistema) y un usuario **app\_admin** (suponiendo un usuario o más que sean administradores).

SpringBootKeycloak

Manage

Clients

Client scopes

Realm roles

Users

Groups

Sessions

Events

Realm roles

Realm roles are the roles that you define for use in the current re

Search role by name

→

Create role

Role name
app_admin
app_user
default-roles-springbootkeycloak
offline_access
uma_authorization

A partir de esto, el rol **app\_user** del reino tiene que ser compuesto/estar relacionado con el rol **user** del cliente. De igual manera, el rol **app\_admin** del reino tiene que ser compuesto/estar relacionado con el rol **admin** del cliente.

Para hacer esta asociación, dentro del menú Realm Roles elegimos uno de los dos roles, hacemos click en él y arriba a la derecha, en el desplegable action hacemos click en **“Add associated roles”**

Realm roles > Role details

app\_admin

Composite

Details

Associated roles

Attributes

Users in role

Role name

app\_admin

Description

Save

Cancel

Action

Add associated roles

Delete this role

Una vez allí, seleccionamos el filtro por clientes y buscamos el rol de cliente que habíamos creado llamado admin y lo asociamos:

Assign roles to app\_admin

Filter by clients

admin

1 - 2

<input type="checkbox"/>	Name	Description
<input type="checkbox"/>	realm-management realm-admin	\${role_realm-admin}
<input type="checkbox"/>	springboot-keycloak-client admin	


1 - 2


Assign

Cancel

Una vez hecho esto, hacemos lo mismo para rol **app\_user**. Si todo sale bien, en el apartado Real Roles deberíamos ver ahora a ambos roles de reino con el valor **True** en la columna Composite de la tabla.

## Realm roles

Realm roles are the roles that you define for use in the current realm. [Learn more](#) 

<input type="text" value="Search role by name"/>	→	Create role
Role name	Composite	
<a href="#">app_admin</a>	True	
<a href="#">app_user</a>	True	
<a href="#">default-roles-springbootkeycloak</a> 	True	
<a href="#">offline_access</a>	False	
<a href="#">uma_authorization</a>	False	

**Paso 4:** Ahora crearemos una serie de usuarios que luego tendrán tanto roles de user como roles de admin.

- **globalAdmin** que tendrá el rol de reino **app\_admin**
- **user1** que tendrá el rol de reino **app\_user**
- **user2** que tendrá los roles de reino **app\_user** y **app\_admin**

Para simplificar esta prueba, coloquemos a los 3 usuarios la contraseña “password”.

En las imagenes a continuación veremos un ejemplo de creación y asignación para el usuario **globalAdmin**. Seguir los mismos pasos para **user1** y **user2**, asignando a dichos usuarios las características mencionadas anteriormente.

## Create user

Required user actions ?

Select action

Username \*

globalAdmin

Password \*

password

Email

New password confirmation \*

password

Email verified ?

☐ No

Temporary ?

☐ Off

First name

Save

Cancel

Last name

Groups ?

Join Groups

Create

Cancel

## globaladmin

Details

Attributes

Credentials

Role mapping

Groups

Consents

Identity provid

Search by name

→

☒ Hide inherited roles

Assign role

Unassign

<input type="checkbox"/>	Name	Inherited	Descripti
<input type="checkbox"/>	app_admin	False	—
<input type="checkbox"/>	default-roles-springbootkeycloak	False	\${role_de

Con esto terminamos la primera parte de nuestras configuraciones en Keycloak. Ahora, pasaremos a la 2da parte de este tutorial: A crear el proyecto de Spring Boot para luego integrarlo con Keycloak.

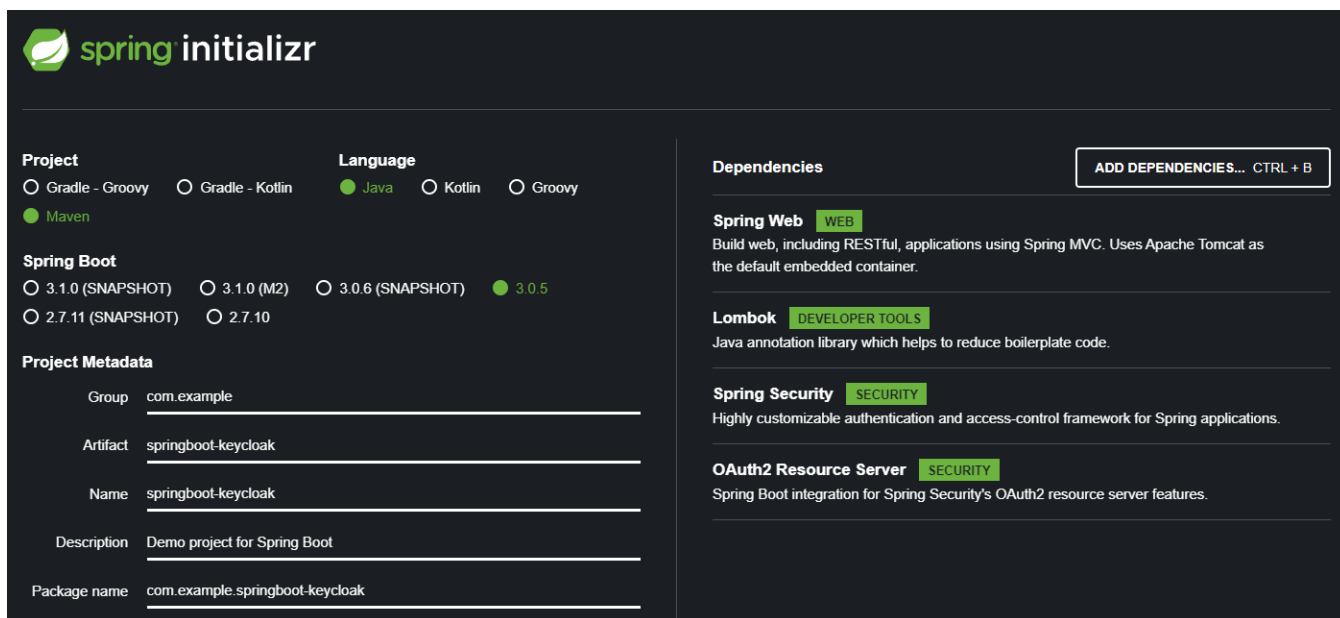


## 2da Parte - Keycloak + Spring Boot

**Paso 1:** Nos dirigimos a Initializr mediante <https://start.spring.io/> y vamos a crear un nuevo proyecto de Spring Boot teniendo en cuenta como dependencias a:

- Spring Web
- Lombok
- Spring Security
- OAuth2 Resource Server

Un detalle **MUY IMPORTANTE** a tener en cuenta son las versiones que seleccionemos. Recordá que si elegís como versión de **Spring Boot la 3** o cualquiera en adelante, tenés que tener instalado en tu pc y en tu IDE por lo menos el **JDK 17** o una versión superior para asegurarte compatibilidad.



The screenshot shows the Spring Initializr web application interface. It is divided into several sections:

- Project:** Includes radio buttons for **Gradle - Groovy**, **Gradle - Kotlin**, and **Maven** (selected).
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions **3.1.0 (SNAPSHOT)**, **3.1.0 (M2)**, **3.0.6 (SNAPSHOT)**, and **3.0.5** (selected).
- Project Metadata:** Includes input fields for **Group** (com.example), **Artifact** (springboot-keycloak), **Name** (springboot-keycloak), **Description** (Demo project for Spring Boot), and **Package name** (com.example.springboot-keycloak).
- Dependencies:** A list of selected dependencies with their categories in green boxes:
  - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
  - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
  - Spring Security** (SECURITY): Highly customizable authentication and access-control framework for Spring applications.
  - OAuth2 Resource Server** (SECURITY): Spring Boot integration for Spring Security's OAuth2 resource server features.

An **ADD DEPENDENCIES... CTRL + B** button is located at the top right of the Dependencies section.

**Podés descargar esta misma configuración desde este link:** [Link Configuración](#)

Una vez creado el proyecto, lo descargamos, lo descomprimos y lo abriremos en nuestro IDE.

**Paso 2:** Una vez creado levantado nuestro proyecto en el IDE, vamos a proceder a configurar el puerto 8081 para nuestra aplicación en el archivo `application.properties`. y vamos a agregar un `context-path` que sea `/api`

```
server.port:8081
server.servlet.context-path=/api
```

A partir de esto, vamos a crear un paquete controller y otro security dentro de nuestra app. En el paquete controller vamos a crear los siguientes endpoints:

```
@RestController
@RequestMapping("/test")
public class TestController {

    @GetMapping("/anonymous")
    public ResponseEntity<String> getAnonymous() {
        return ResponseEntity.ok("Hello Anonymous");
    }

    @GetMapping("/admin")
    public ResponseEntity<String> getAdmin() {
        return ResponseEntity.ok("Hello Admin");
    }

    @GetMapping("/user")
    public ResponseEntity<String> getUser() {
        return ResponseEntity.ok("Hello User");
    }
}
```

**Paso 3:** Una vez creado nuestra clase controller, toca agregar unas configuraciones extras a nuestro **application.properties** para lograr una correcta integración entre Keycloak y nuestra aplicación:

```
server.port=8081
server.servlet.context-path=/api

spring.application.name=springboot-keycloak

spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8080/
realms/SpringBootKeycloak
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=${spring.security.oau
th2.resourceserver.jwt.issuer-uri}/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
```

**Paso 4:** Crearemos, dentro del paquete security, una clase llamada **“JwtAuthConverterProperties”** para traer las configuraciones que hemos realizado a nuestra aplicación desarrollada con Spring Boot.

```
package com.example.springbootkeycloak.security;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.validation.annotation.Validated;

@Data
@Validated
@Configuration
@ConfigurationProperties(prefix = "jwt.auth.converter")
public class JwtAuthConverterProperties {

    private String resourceId;
    private String principalAttribute;
}
```

**Paso 5:** Una vez creada la clase del paso 4, procederemos a crear en el mismo paquete security, otra clase llamada **“JwtAuthConverter”**. En esta clase extraemos los principales datos y roles del access token del JWT.

```

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.JwtAuthen
ticationToken;
import
org.springframework.security.oauth2.server.resource.authentication.JwtGrant
edAuthoritiesConverter;
import org.springframework.stereotype.Component;

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,
AbstractAuthenticationToken> {

    private final JwtGrantedAuthoritiesConverter
jwtGrantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();

    private final JwtAuthConverterProperties properties;

    public JwtAuthConverter(JwtAuthConverterProperties properties) {
        this.properties = properties;
    }

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

    private String getPrincipalClaimName(Jwt jwt) {
        String claimName = JwtClaimNames.SUB;
        if (properties.getPrincipalAttribute() != null) {
            claimName = properties.getPrincipalAttribute();
        }
        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(properties.getResourceId())) == null
            || (resourceRoles = (Collection<String>)
resource.get("roles")) == null) {
            return Set.of();
        }
        return resourceRoles.stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
            .collect(Collectors.toSet());
    }
}

```

**Paso 6:** Por último, necesitamos agregar una clase de configuración para lo que respecta a OAuth2 y la autenticación con el JWT. Para ello, vamos a crear dentro del paquete security una clase llamada “**WebSecurityConfig**”.

```
package com.example.springbootkeycloak.security;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@RequiredArgsConstructor
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

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

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
            .requestMatchers(HttpMethod.GET, "/test/anonymous",
"/test/anonymous/**").permitAll()
            .requestMatchers(HttpMethod.GET, "/test/admin",
"/test/admin/**").hasRole(ADMIN)
            .requestMatchers(HttpMethod.GET, "/test/user").hasAnyRole(ADMIN,
USER)
            .anyRequest().authenticated();
        http.oauth2ResourceServer()
            .jwt()
            .jwtAuthenticationConverter(jwtAuthConverter);

        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        return http.build();
    }
}
```

Aquí estamos anulando la configuración de seguridad HTTP por defecto. Necesitamos especificar explícitamente que queremos que se comporte como un Servidor de Recursos. Esto se logra a través del uso del método SecurityFilterChain.

## 3ra Parte - Probando con Postman

## Prueba de un caso feliz (usuario con permisos)

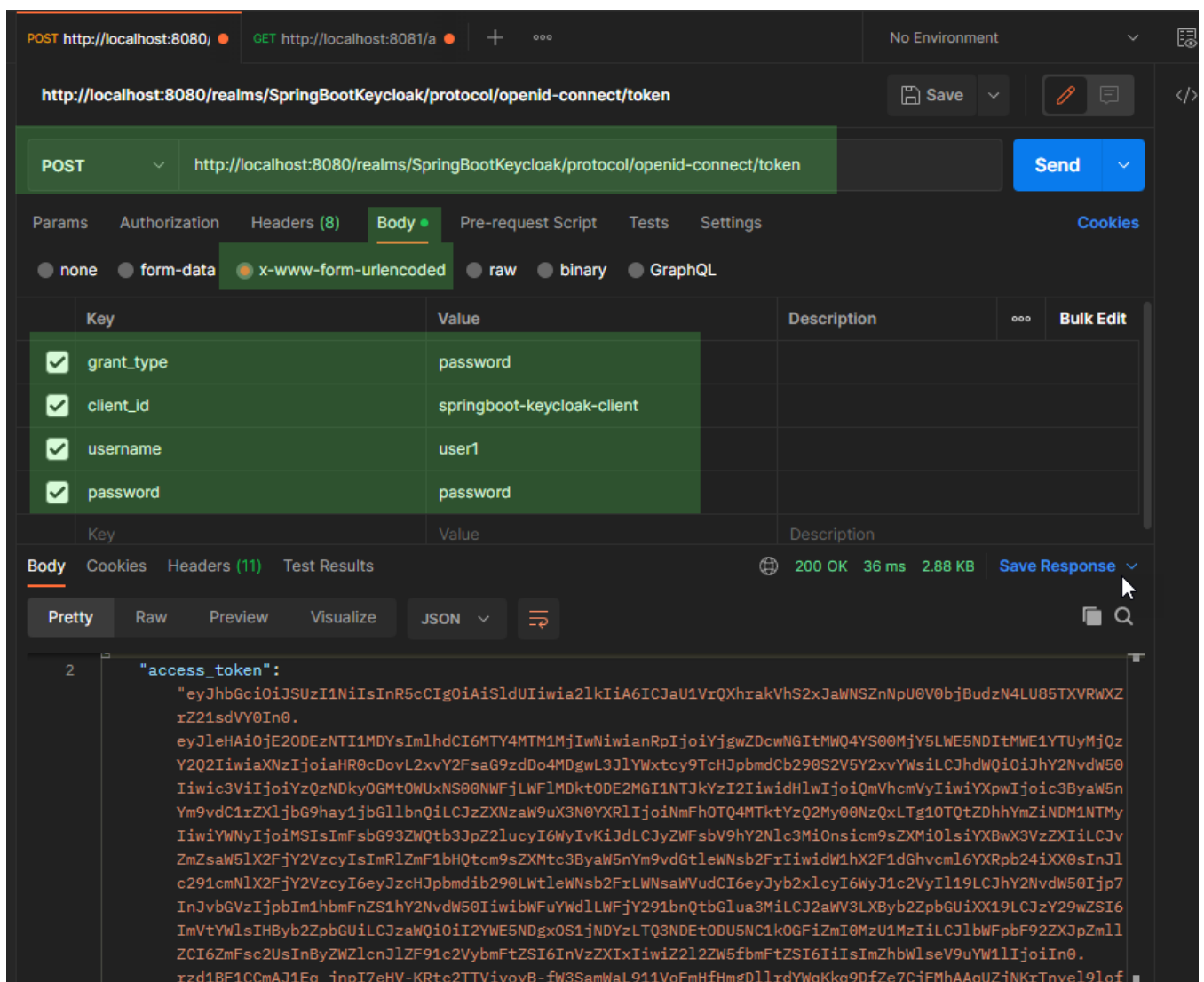
**Paso 1:** Vamos a dirigirnos a Postman y vamos a crear una nueva Request. Esta será de tipo POST y la haremos a la URI para obtención del token

(<http://localhost:8080/realms/SpringBootKeycloak/protocol/openid-connect/token>)

En el body de nuestra request , seleccionaremos x-www-form-urlencoded y vamos a tener en cuenta los siguientes parámetros:

- **grant\_type:** password
- **client\_id:** springboot-keycloak-client
- **username:** user1
- **password:** password

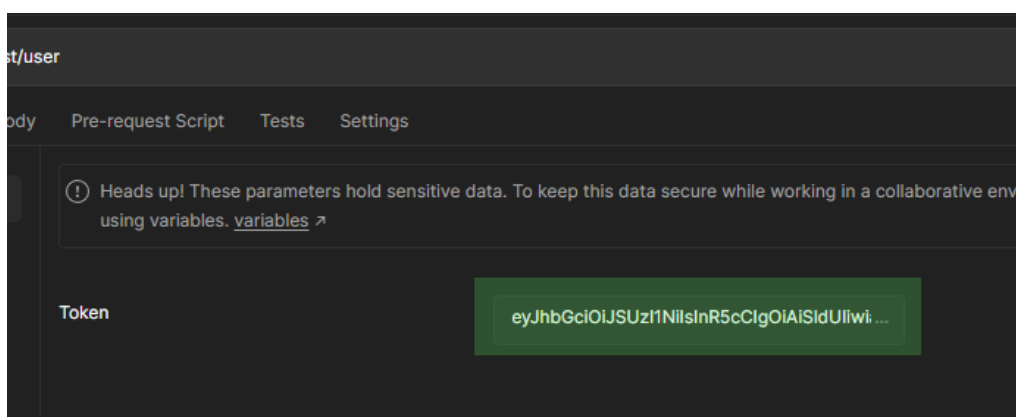
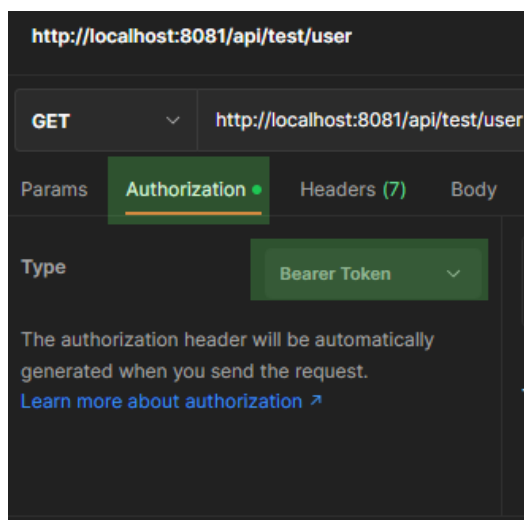
En `client_id` pusimos el nombre de nuestro cliente creado en Keycloak, en `username` vamos a probar con el `user1` y en `password` la contraseña que habíamos asignado al `user1`.



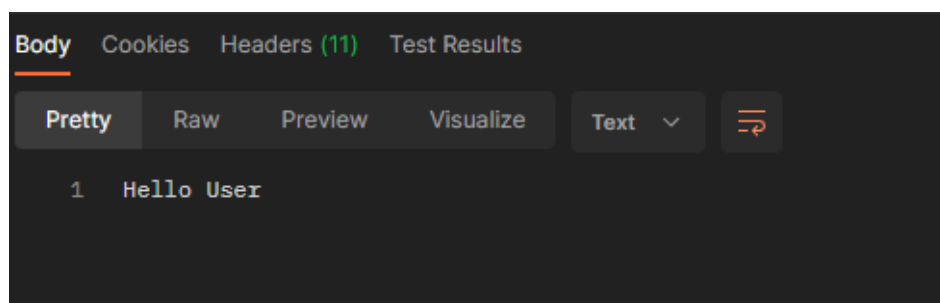
Si todo sale bien, vamos a obtener una response como la que se ve en la imagen con el **access token** y otros valores. Vamos a copiar todo el valor que se encuentra entre comillas en el apartado **access token**.

**Paso 2:** Con el token ya creado, vamos a crear una nueva request en **Postman**, de tipo **GET** que vaya a la URI del endpoint que creamos en nuestra app con **Spring Boot** que está destinada a los usuarios comunes (<http://localhost:8081/api/test/user>).

En la pestaña **Authorization** vamos a seleccionar el **Type: Bearer Token** y en el campo que dice **Token** a la derecha pegamos el **access token** que generamos en el paso 1.



Una vez completado todo esto, vamos a hacer click en Send y ver qué obtenemos de respuesta desde nuestra API. Si todo sale bien, tendremos un Status Code 200 y leeremos el mensaje que establecimos en nuestro end-point:





## Prueba de un caso no feliz (usuario sin permisos)

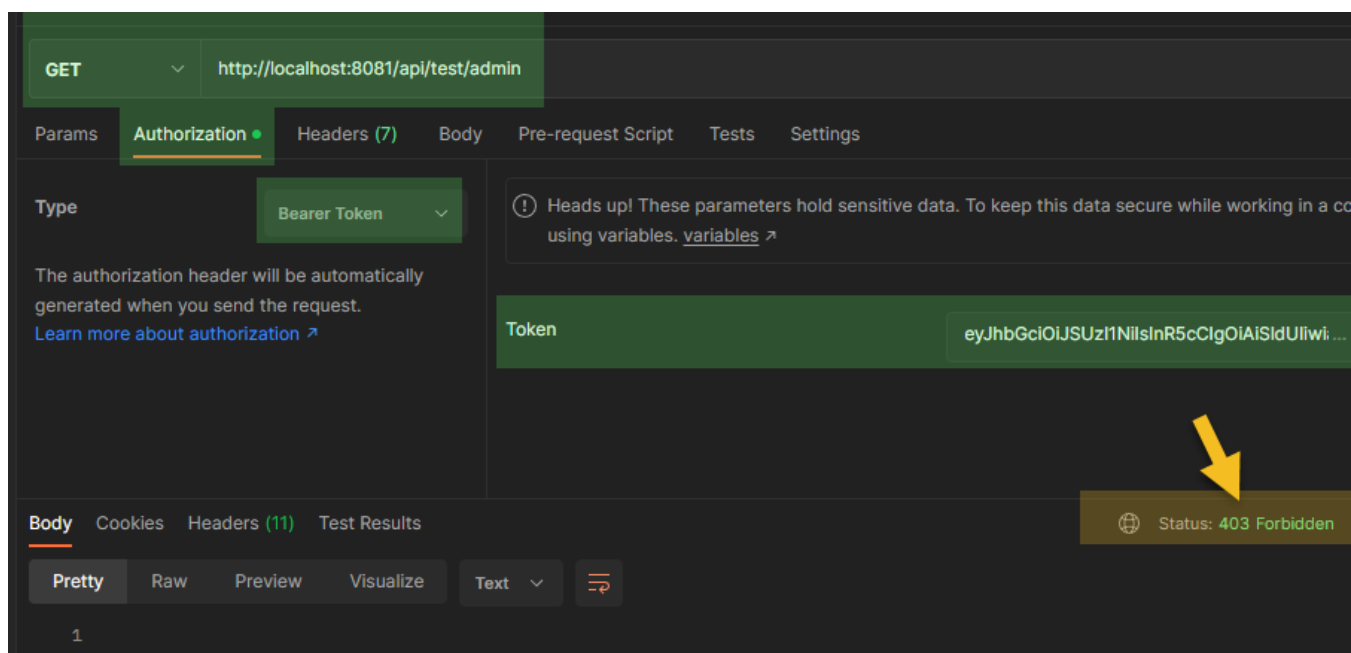
Tomemos el mismo caso anterior pero en lugar de querer acceder al **end-point** de **usuarios**, vamos a intentar acceder al **end-point** de **administradores**

(<http://localhost:8081/api/test/admin>) con el mismo **usuario** que configuramos (**user1**).

Claramente sabemos que **user1** no tiene permisos de administrador, así que como **respuesta** deberíamos de obtener que **no estamos autorizados**. Probemos esto.

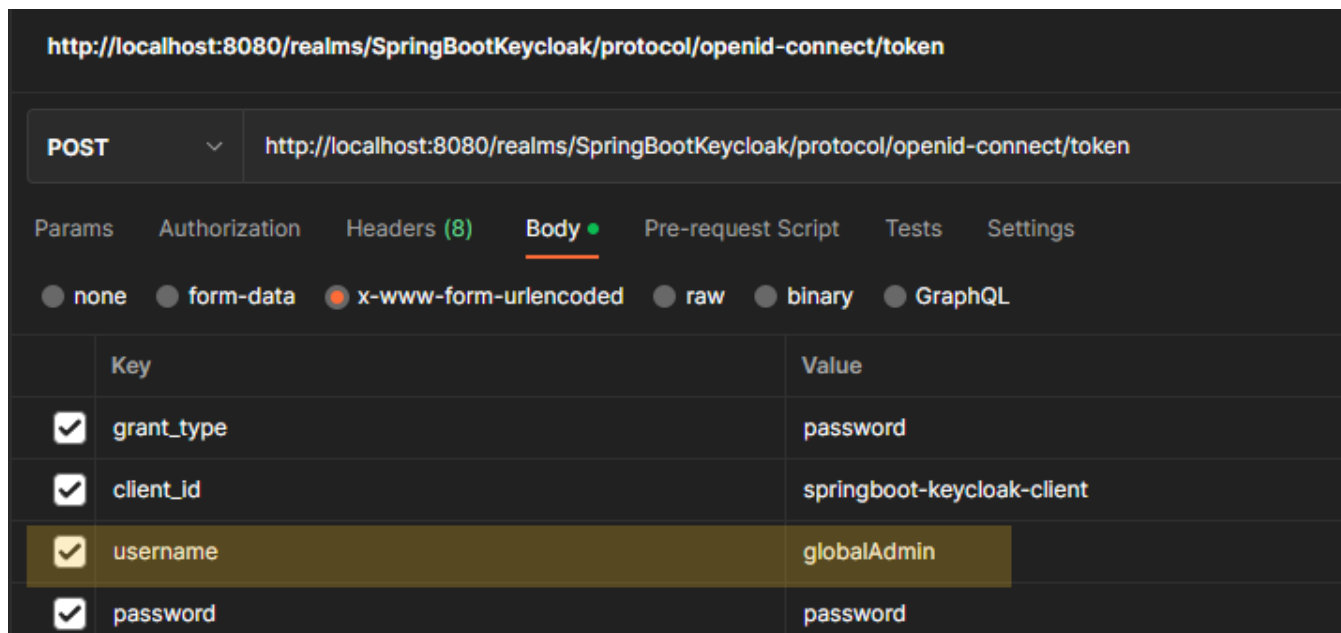
**Paso 1:** Vamos a generar nuevamente un token en Postman enviando una nueva solicitud desde la request que creamos en el paso 1 de la prueba anterior y copiamos el access token generado.

**Paso 2:** Creamos una nueva request de tipo **get** pero a la **URL de administradores que mencionamos anteriormente** (<http://localhost:8081/api/test/admin>). Vamos a la pestaña **Authorization**, seleccionamos de igual manera el **Type: Bearer Token** y en el campo que dice **Token** a la derecha pegamos el access token que generamos en el paso 1 y enviamos la solicitud de prueba mediante **Send**.



Como vemos, al no autenticarnos con un usuario con roles y permisos de administrador, no podremos acceder a dicho end-point.

Sin embargo, si generamos un nuevo token, cambiando los parámetros de usuario y contraseña por el de **admin** en nuestra primera request (que si tiene rol de administrador).



y colocamos el token obtenido en nuestra nueva solicitud al end-point de administradores:

Vamos a poder acceder sin problema alguno y a obtener el mensaje correspondiente al end-point en cuestión.