

# **DESAFIO 3 DWF404 G03L**



**Facultad de Ingeniería  
Escuela de Computación (EIC)**

**Integrante:**

Nombre:	Nº Carnet
Bryan Steven Hernández Polio	HP240512

**Asignatura: Desarrollo de Aplicaciones con Web Frameworks**

**Fecha de entrega: 15 de octubre**

**Docente: Ingeniero Juan Carlos Menjivar**

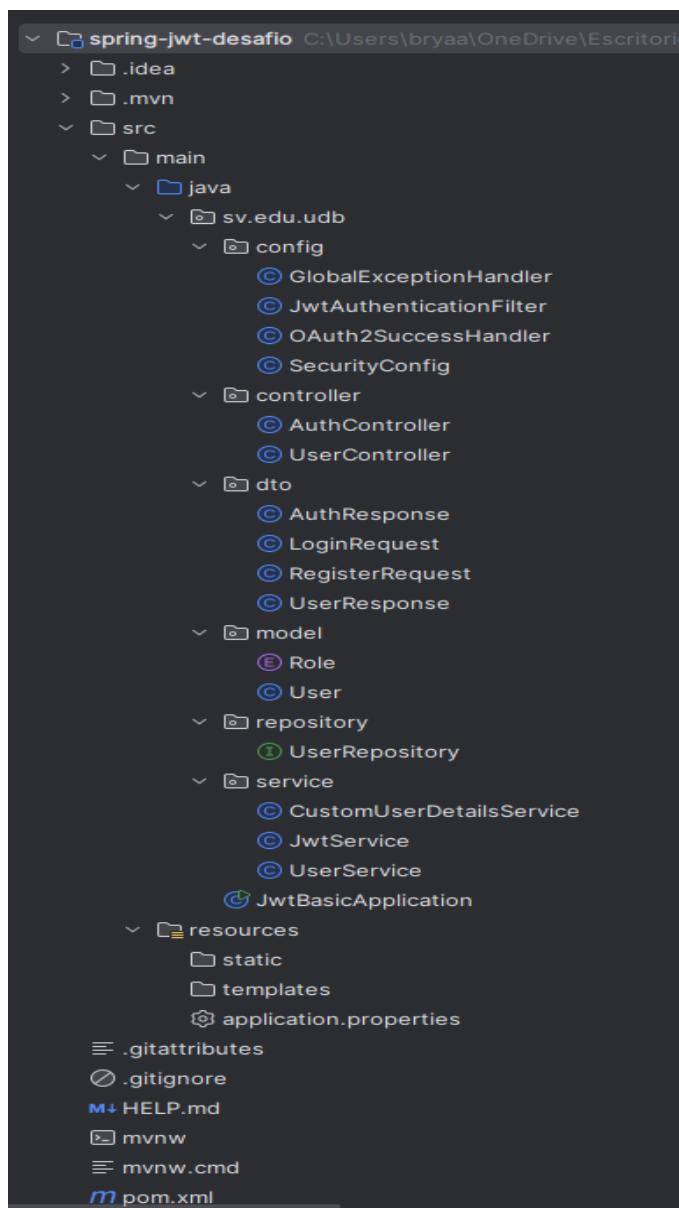
**Ramírez.**

# DOCUMENTACIÓN - API SEGURA CON SPRING BOOT, JWT Y OAUTH2

## 1. INTRODUCCIÓN AL SISTEMA

Este proyecto fue desarrollado para implementar una API REST segura desarrollada en Java con Spring Boot que proporciona un sistema completo de autenticación y autorización. El sistema permite a los usuarios registrarse y autenticarse mediante dos métodos: autenticación tradicional con JWT (JSON Web Tokens) y autenticación social mediante OAuth2 con GitHub.

## 2. ESTRUCTURA GENERAL DEL PROYECTO Y EXPLICACIÓN DE PAQUETES



### 3. EXPLICACIÓN DE LA ESTRUCTURA DEL PROYECTO

- **CONFIGURACIONES (Paquete config/)**

**SecurityConfig.java** - Define la configuración central de seguridad, estableciendo políticas de acceso para los diferentes endpoints, configurando la autenticación sin estado para trabajar con JWT, e integrando el sistema OAuth2 con GitHub como proveedor de identidad.

**JwtAuthenticationFilter.java** - Actúa como interceptor en cada petición HTTP, examinando la presencia y validez de los tokens JWT. Cuando encuentra un token válido, establece el contexto de autenticación para que los controladores puedan identificar al usuario.

**OAuth2SuccessHandler.java** - Gestiona el proceso posterior a una autenticación exitosa con GitHub, creando o actualizando el usuario en nuestra base de datos y generando un token JWT que permitirá al usuario acceder a los recursos protegidos.

**GlobalExceptionHandler.java** - Centraliza el manejo de errores en toda la aplicación, asegurando que cualquier excepción sea convertida a una respuesta JSON estructurada en lugar de páginas de error genéricas.

- **CONTROLADORES (Paquete controller/)**

**AuthController.java** - Expone los endpoints de autenticación, permitiendo el registro de nuevos usuarios y el inicio de sesión de usuarios existentes, devolviendo en ambos casos un token JWT válido.

**UserController.java** - Contiene endpoints que requieren autenticación, incluyendo funcionalidades para obtener el perfil del usuario actual y endpoints restringidos por roles de usuario.

- **OBJETOS DE TRANSFERENCIA (Paquete dto/)**

**LoginRequest.java** - Captura las credenciales de autenticación (email y contraseña) enviadas por el cliente.

**RegisterRequest.java** - Contiene la información necesaria para registrar un nuevo usuario en el sistema.

**AuthResponse.java** - Devuelve el token JWT junto con los datos básicos del usuario después de una autenticación exitosa.

**UserResponse.java** - Representa la información del usuario que se expone de forma segura al cliente, excluyendo datos sensibles.

- **MODELOS DE DATOS (Paquete model/)**

**User.java** - Define la entidad usuario que se persiste en la base de datos, implementando las interfaces necesarias para integrarse con Spring Security.

**Role.java** - Enumera los diferentes roles que puede tener un usuario dentro del sistema, determinando sus niveles de acceso.

- **ACCESO A DATOS (Paquete repository/)**

**UserRepository.java** - Proporciona operaciones de consulta y persistencia para la entidad usuario, incluyendo métodos personalizados para buscar usuarios por diferentes criterios.

- **SERVICIOS (Paquete service/)**

**UserService.java** - Encapsula la lógica relacionada con la gestión de usuarios, incluyendo la creación de usuarios tanto tradicionales como provenientes de OAuth2.

**JwtService.java** - Se encarga de todas las operaciones relacionadas con tokens JWT, desde su generación hasta su validación.

**CustomUserDetailsService.java** - Implementa la interfaz requerida por Spring Security para cargar los detalles del usuario durante el proceso de autenticación.

## 4. DIAGRAMA DE FLUJO DE AUTENTICACIÓN

### Listado de Flujo de JWT :

1. Cliente → POST /auth/register (email, password, name)
2. Servidor → Encripta password, crea usuario, genera JWT
3. Cliente → POST /auth/login (email, password)
4. Servidor → Valida credenciales, genera JWT
5. Cliente → Incluye JWT en header Authorization: Bearer {token}
6. Servidor → Valida JWT, permite acceso a endpoints protegidos

## **Listado de Flujo de OAuth2 con GitHub:**

1. Cliente → GET /oauth2/authorization/github
2. GitHub → Página de autorización (HTML)
3. Usuario → Autoriza aplicación en GitHub
4. GitHub → Redirige a /login/oauth2/code/github con code
5. Servidor → Intercambia code por access\_token, obtiene datos usuario
6. Servidor → Busca/crea usuario, genera JWT
7. Servidor → Retorna JSON con JWT y datos usuario

## **5. EXPLICACIÓN DEL USO DE SEGURIDAD CON ROLES Y ANOTACIONES**

### **@PreAuthorize:**

Sistema de Autorización con Roles y @PreAuthorize

El sistema implementa un control de acceso basado en roles que permite restringir el acceso a diferentes endpoints según los privilegios del usuario autenticado.

Definición de Roles:

Se establecieron dos roles principales: USER para usuarios regulares y ADMIN para administradores. Cada usuario al crearse en el sistema se le asigna uno de estos roles, que determina qué recursos puede acceder.

Mecanismo de Autorización:

La seguridad se implementa mediante la anotación @PreAuthorize, que evalúa las condiciones de acceso antes de ejecutar cada método. Por ejemplo, el endpoint /users/admin está protegido con la condición "hasRole('ADMIN')", lo que significa que solo usuarios con rol ADMIN pueden acceder a él.

Flujo de Verificación

Cuando un usuario autenticado intenta acceder a un endpoint protegido, el sistema:

1. Verifica la validez del token JWT
2. Extrae los roles del usuario desde el token
3. Evalúa si el usuario tiene el rol requerido
4. Permite o deniega el acceso según corresponda

Ventajas de la Implementación

Este enfoque proporciona un control granular sobre los recursos, es fácil de mantener y extender, y se integra naturalmente con el ecosistema de Spring Security. Los mensajes de error son claros: los usuarios sin los permisos adecuados reciben una respuesta HTTP 403 Forbidden con un mensaje explicativo.

## SEGURIDAD: (imagen)

SecurityConfig.java :

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .csrf(CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
        .sessionManagement(SessionManagementConfigurer<HttpSecurity> session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(AuthorizationManagerRequestMatcher auth -> auth
            .requestMatchers(patterns: "/auth/**", "/oauth2/**").permitAll()
            .requestMatchers(patterns: "/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
        )
}
```

## IMPLEMENTACION DE ROLES: (imágenes)

Role.Enum :

```
package sv.edu.udb.model; //BRYAN STEVEN HERNANDEZ POLIO HP240512

public enum Role { 14 usages
    USER, 1 usage
    ADMIN no usages
}
```

User.java :

```
//Implementacion de rol

@Enumerated(EnumType.STRING) 5 usages
private Role role;
}

// 
@Override 1 usage
public Collection<? extends GrantedAuthority> getAuthorities() {
    return Collections.singletonList(new SimpleGrantedAuthority(role: "ROLE_" + role.name()));
}
```

## USO DE @PreAuthorize: (imagen)

UserController.java :

```
💡 @GetMapping("/admin") no usages
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<String> adminEndpoint() {
    return ResponseEntity.ok( body: "Este es un endpoint solo para ADMIN");
}

@GetMapping("/profile") no usages
@PreAuthorize("hasRole('USER')")
public ResponseEntity<String> userProfile() { return ResponseEntity.ok( body: "Perfil de usuario"); }
}
```

## 6. EXPLICACIÓN DEL APPLICATION.PROPERTIES:

Este archivo guarda todos los ajustes importantes de la aplicación. Aquí se define cómo conectarse a la base de datos MySQL, con su dirección, usuario y contraseña. También se configura JPA para que maneje automáticamente las tablas de la base de datos.

Para la seguridad, establecemos la clave secreta que usan los tokens JWT y cuánto tiempo duran antes de expirar. Incluimos las credenciales de la aplicación de GitHub para que funcione el login con OAuth2.

El archivo también controla en qué puerto corre el servidor y tiene opciones especiales para desarrollo. Cada cambio aquí afecta directamente cómo funciona la aplicación.

## 7. EJEMPLOS DE PRUEBAS EN POSTMAN: ---- JWT ----- OAuth2 con GitHub -----

Registro de usuario tipo User en postman por post usando JWT:

Nos devolverá lo ingresado con su id y su respectivo token

The screenshot shows the Postman interface with a collection named "ring-jwt-desafio". A POST request is made to `http://localhost:8080/auth/register?role=USER`. The request body is a JSON object with fields: `"email": "usuario@test.com"`, `"password": "password123"`, and `"name": "Usuario Test"`. The response status is 200 OK, and the response body contains a JSON object with a token, type (Bearer), and user details (id: 1, email: "usuario@test.com", name: "Usuario Test", role: "USER").

Login con el usuario creado anteriormente con jwt exitoso y nos devuelve el token y datos:

The screenshot shows the Postman interface with the same collection. A POST request is made to `http://localhost:8080/auth/login`. The request body is a JSON object with fields: `"email": "usuario@test.com"` and `"password": "password123"`. The response status is 200 OK, and the response body contains a JSON object with a token, type (Bearer), and user details (id: 1, email: "usuario@test.com", name: "Usuario Test", role: "USER").

Realizando un get a los users normales No Admin, esto con el mismo token que nos dio funciona correctamente:

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8080/users/me`. The response body is a JSON object containing user information:

```
1 {  
2   "id": 1,  
3   "email": "usuario@test.com",  
4   "name": "Usuario Test",  
5   "role": "USER"  
6 }
```

Demostración de la base de datos y el usuario ingresado, su contraseña se encripta correctamente:

The screenshot shows the phpMyAdmin interface connected to the `jwt_basic_db` database. The `users` table has one row of data:

	<a href="#">id</a>	<a href="#">created_at</a>	<a href="#">email</a>	<a href="#">name</a>	<a href="#">oauth_id</a>	<a href="#">oauth_provider</a>	<a href="#">password</a>	<a href="#">role</a>
	1	2025-10-14 20:14:12.000000	usuario@test.com	Usuario Test	NULL	NULL	\$2a\$10\$Om8AKi9prLKj9TAaVVAquYSF84nUs9S9iicBQMpaQ...	USER

Prueba con User modo Admin se crea correctamente y nos da su token:

The screenshot shows a Postman collection named "jwt-desafio". A POST request is made to `http://localhost:8080/auth/register?role=ADMIN`. The request body is a JSON object with fields: email, password, and name. The response status is 200 OK, and the response body contains a token and user details.

```
1 {  
2   "email": "admin@test.com",  
3   "password": "admin123",  
4   "name": "Administrador"  
5 }
```

```
1 {  
2   "token": "eyJhbGciOiJIUzIzM4NCJ9.eyJzdkiOiJhZG1pbkB0ZXN0LmNvbSIsImIhdCI6MTc2MDQ5NDk4NCwiZXhwIjoxNzYwNTgxMzg0fQ.YbXkUoCSQi-F_UhriTqXwvHxDw9UzqKRA-Zq5ipLLYFTsK-FR9jfWkZutoNbCi-",  
3   "type": "Bearer",  
4   "user": {  
5     "id": 2,  
6     "email": "admin@test.com",  
7     "name": "Administrador",  
8     "role": "ADMIN"  
9   }  
10 }
```

Realizando un Get de admin con su respectivo token nos devuelve el mensaje de admin:

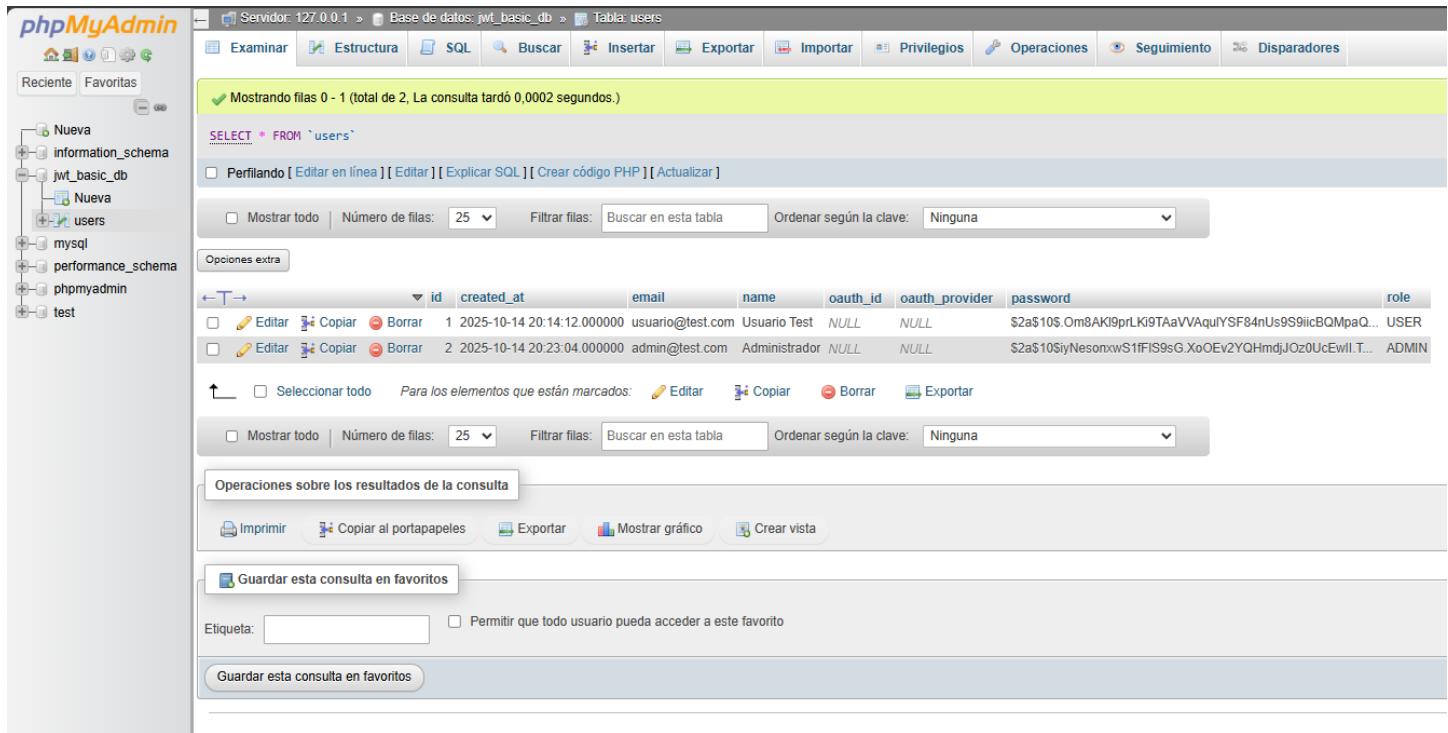
The screenshot shows a Postman collection named "wt-desafio". A GET request is made to `http://localhost:8080/users/admin`. The request includes an Authorization header set to "Bearer Token" with the value of the previously generated token. The response status is 200 OK, and the response body contains the message "Este es un endpoint solo para ADMIN".

Auth Type: Bearer Token

Token: RA-Zq5ipLLYFTsK-FR9jfWkZutoNbCi-

```
1 Este es un endpoint solo para ADMIN
```

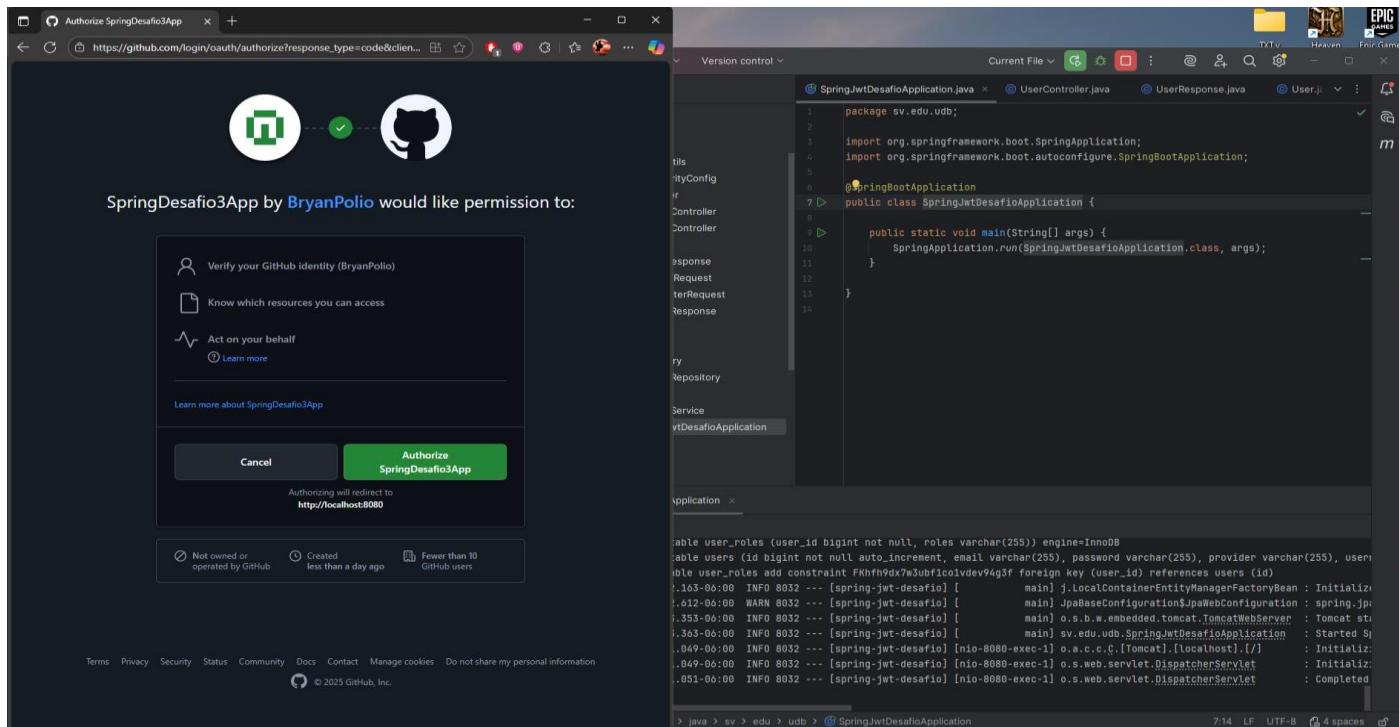
## Demostración de la base de datos, usuario admin ingresado:



The screenshot shows the phpMyAdmin interface for a database named 'jwt\_basic\_db'. The 'users' table is selected. The table has columns: id, created\_at, email, name, oauth\_id, oauth\_provider, password, and role. There are two rows:

	id	created_at	email	name	oauth_id	oauth_provider	password	role
1	1	2025-10-14 20:14:12.000000	usuario@test.com	Usuario Test	NULL	NULL	\$2a\$10\$Om8AKi9prLk9TaaVVAqUySF84nUs9S9icBQMpaQ...	USER
2	2	2025-10-14 20:23:04.000000	admin@test.com	Administrador	NULL	NULL	\$2a\$10\$iyNesonxwS1FIS9sG.XoOEv2YQHmdjOz0UcEwIl...	ADMIN

Prueba con OAuth2 con GitHub: <http://localhost:8080/oauth2/authorization/github>



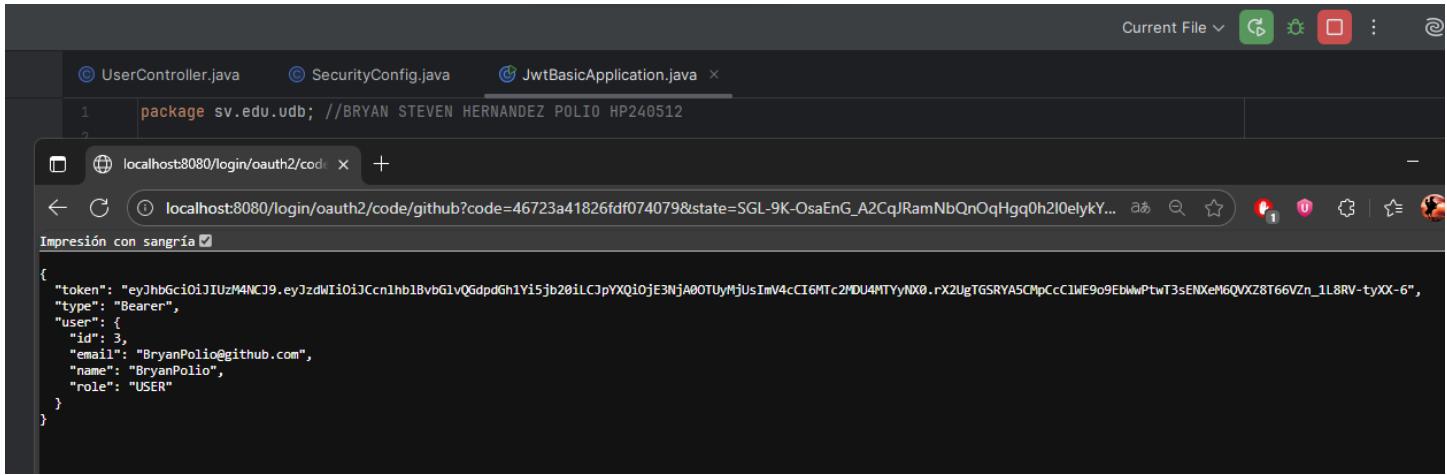
The screenshot shows a browser window with the URL [https://github.com/login/oauth/authorize?response\\_type=code&client\\_id=...](https://github.com/login/oauth/authorize?response_type=code&client_id=...). The GitHub OAuth consent screen is displayed, asking for permission to 'SpringDesafio3App by BryanPolio'. The permissions listed are:

- Verify your GitHub identity (BryanPolio)
- Know which resources you can access
- Act on your behalf

At the bottom, there are 'Cancel' and 'Authorize SpringDesafio3App' buttons. Below the buttons, it says 'Authorizing will redirect to <http://localhost:8080>'. At the very bottom, there are footer links for Terms, Privacy, Security, Status, Community, Docs, Contact, Manage cookies, and a note about cookie sharing.

On the right side of the screenshot, there is a code editor showing Java code for a Spring Boot application. The main method in `SpringJwtDesafioApplication.java` is:public static void main(String[] args) {  
 SpringApplication.run(SpringJwtDesafioApplication.class, args);  
}

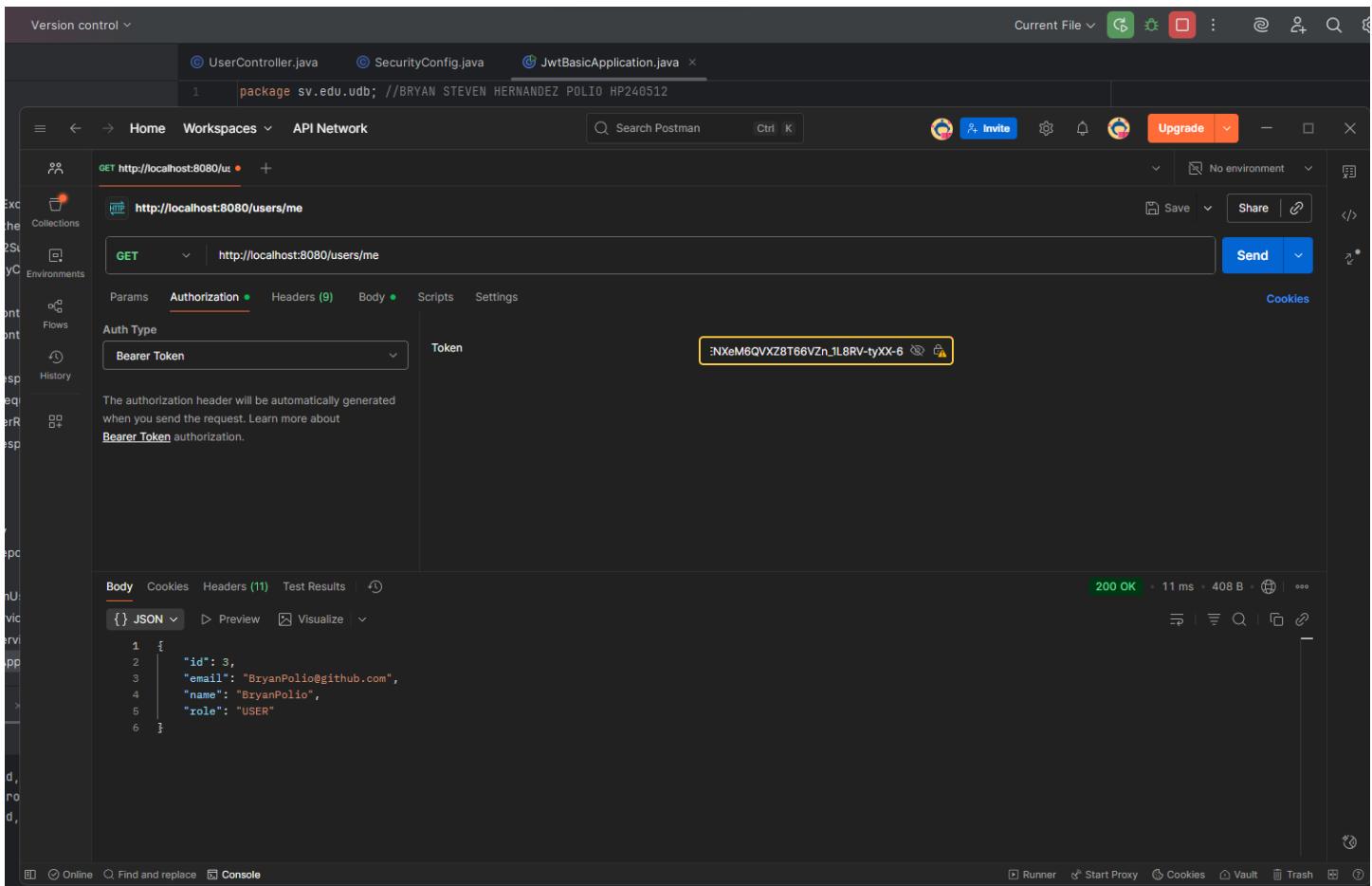
Una vez aceptado nos lleva a la pagina para seleccionar github y nos entrega los datos correctamente:



The screenshot shows a browser window with the URL `localhost:8080/login/oauth2/code/github?code=46723a41826fdf074079&state=SGL-9K-OsaEnG_A2CqJRamNbQnOqHgq0h2l0elykY...`. The page displays a JSON response from the application, which includes a token and user information.

```
{  
  "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJCcn1hb1Bvb61vQGdpdGh1Yi5jb201LCjpxXQiOjE3NjA0OTUyMjUsImV4cIT6MTc2MDU4MTYyNX9.rX2UgTGSRYA5CMpCcCIWE9o9EhkwPtwT3sENxeM6QVXZ8T66Vzn_1L8RV-tyXX-6",  
  "type": "Bearer",  
  "user": {  
    "id": 3,  
    "email": "BryanPolio@github.com",  
    "name": "BryanPolio",  
    "role": "USER"  
  }  
}
```

Get con el token de Github funciona correctamente:



The screenshot shows a Postman collection with a single API endpoint. The endpoint is `GET http://localhost:8080/users/me`. The Authorization tab is selected, showing a Bearer Token. The token value is `ENxeM6QVXZ8T66Vzn_1L8RV-tyXX-6`. The response body is displayed as JSON, matching the user information shown in the previous screenshot.

```
{  
  "id": 3,  
  "email": "BryanPolio@github.com",  
  "name": "BryanPolio",  
  "role": "USER"  
}
```

## Demostración de usuario Github en base de datos:

Mostrando filas 0 - 2 (total de 3, La consulta tardó 0,0003 segundos.)

	<a href="#">Editar</a>	<a href="#">Copiar</a>	<a href="#">Borrar</a>	<a href="#">1</a> 2025-10-14 20:14:12.000000	usuario@test.com	Usuario Test	NULL	NULL	\$2a\$10\$Om8AK9prLK9TAaVVAqulYSF84nUs9S9licBQMpaQ...	USER
	<a href="#">Editar</a>	<a href="#">Copiar</a>	<a href="#">Borrar</a>	<a href="#">2</a> 2025-10-14 20:23:04.000000	admin@test.com	Administrador	NULL	NULL	\$2a\$10\$iyNesonxwS1fFIS9sG.XoOEv2YQHmdjOz0UcEwlI.T...	ADMIN
	<a href="#">Editar</a>	<a href="#">Copiar</a>	<a href="#">Borrar</a>	<a href="#">3</a> 2025-10-14 20:27:05.000000	BryanPolio@github.com	BryanPolio	175163506	github	\$2a\$10\$JkkM9PkBoNxVbJnrgNiervhxQs5PBRRThOUMQx9w...	USER

## ALGUNAS PRUEBAS PARA OBTENER ERRORES:

Get sin token:

GET http://localhost:8080/users/me

401 Unauthorized

Body

```
{ "error": "Unauthorized", "message": "Authentication required" }
```

## Get con token invalido:

The screenshot shows the Postman interface with a request to `http://localhost:8080/users/me`. The `Authorization` field is set to `Bearer Token`, and the token value is `32b32jh3kj2h32kj3h23h2kh32kj32j3`. The response status is `401 Unauthorized` with a message: `{"error": "Unauthorized", "message": "Authentication required"}`.

## Get a Admin con un token de tipo usuario normal:

The screenshot shows the Postman interface with a request to `http://localhost:8080/users/admin`. The `Authorization` field is set to `Bearer Token`, and the token value is `eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ`. The response status is `403 Forbidden` with a message: `{"error": "Access Denied", "message": "No tienes permisos para acceder a este recurso"}`.

## **8. CONCLUSIONES PERSONALES SOBRE LA IMPLÉMENTACIÓN:**

Este pequeño proyecto/desafío me enseñó bastante sobre seguridad en APIs. Al principio fue un reto entender cómo integrar JWT con OAuth2, pero al final todo funcionó bien gracias a las guías y a la explicación de la teoría.

Lo más interesante fue ver cómo Spring Security maneja la autenticación automáticamente. Pude hacer que el login funcionara tanto con usuario/contraseña como con GitHub, y que los roles restringieran correctamente el acceso a los endpoints.

Aunque hubo momentos de frustración buscando soluciones en internet por errores no tan graves, el resultado final cumple con todo lo requerido. Ahora entiendo mejor cómo proteger APIs y puedo aplicar esto en futuros proyectos.

### **Comando creación base de datos phpMyAdmin:**

```
CREATE DATABASE IF NOT EXISTS jwt_basic_db;
```