

**Estudiante:** Juan Diego Roa Porras - 2210086

## **Explicación Implementación JWT**

### **Preprocesamiento del proyecto**

Antes de comenzar con la implementación de JWT, se llevó a cabo una fase de preprocesamiento en la que se realizó una copia del proyecto base que se ha estado manejando en el curso. Este proceso incluyó:

- **Estructura de Proveedores y Tipos de Documento:** Se replicaron el modelo, repositorio, servicio y controlador correspondientes a estas entidades.
- **Gestión de Excepciones y Validaciones:** Se implementaron las clases encargadas del manejo de errores y validaciones.
- **Módulo de Usuarios:** Se clonó gran parte del modelo, repositorio, servicio y controlador relacionados con la gestión de usuarios.

Este paso permitió contar con una base sólida antes de integrar JWT para la autenticación y autorización del sistema.

### **Front-End**

El desarrollo del front-end se realizó utilizando **HTML, CSS y JavaScript clásico**, siguiendo las directrices del taller. Para la comunicación con el backend, se emplearon exclusivamente solicitudes **Fetch**, sin el uso de frameworks adicionales.

En términos de funcionalidad, el front es bastante simple, incluyendo un **Login y un SignUp** basados en un diseño de *Universe.io*. Además, se reutilizó la interfaz de gestión de **proveedores** trabajada en clases anteriores, con el propósito de validar que el **token JWT** se genere y funcione correctamente tras el inicio de sesión.

## Pantallazos Resultado Final




### Log In

Usuario

Contraseña

Iniciar Sesión

También puedes iniciar sesión con:

No tienes una cuenta? Únete Aquí

### Sign Up

Nombre

Correo Electrónico

Cedula de Ciudadania

Usuario

Contraseña

Repetir Contraseña

Registrarse

Ya tienes una cuenta? Inicia Sesión Aquí

Administración de Proveedores

Nuevo Proveedor

Cerrar Sesión

1	8901231224	Colgate	Bogotá	3124561238	Carrera 123 # 12 - 80	ver
2	8901235647	Carvajal S.A	Bogotá	2346781122	Avenida 7 # 12 - 80	ver
4	0124578963	test ingreso	Bogota	9876543210	test ingreso	ver

## Back-End

### UsuarioDTO

Se implementó una clase Auxiliar denominada de esta manera, con el objetivo de esconder el ID y el rol de los usuarios, manejó las mismas validaciones que la clase Usuario .

En general no afectó en sobremedida la lógica de implementación simplemente, cuando se le devolvía al cliente una instancia de tipo Usuario, se le devolvería en cambio una instancia de UsuarioDTO.

## LoginDTO

Clase auxiliar para el endpoint de login en el UsuarioController, simplemente estaba compuesto de dos atributos de tipo String, el userName y el Password.

## Usuario Repositorio

Al igual que en el ejemplo visto en clase, se empleó una consulta para localizar al usuario mediante su **userName**. Es importante destacar que, en la base de datos, este campo está definido como **único (Unique)**, garantizando que no existan duplicados.

```
/**
 *
 * @author roa
 */
public interface UsuarioRepositorio extends JpaRepository<Usuario, Long> {

    @Query("SELECT u FROM Usuario as u where u.userName=:userName")
    Usuario findByUsername(@Param("userName") String userName);
}
```

## Configuración de Seguridad con Spring Security y JWT

La clase SecurityConfig define las reglas de seguridad de la aplicación mediante **Spring Security** y **JWT** para la autenticación. Su principal objetivo es **asegurar los endpoints, controlar el acceso de los usuarios y gestionar sesiones sin estado**.

El método `securityFilterChain` establece las reglas de acceso a los endpoints. Se **habilita CORS** para permitir peticiones desde distintos orígenes y se **desactiva CSRF**, ya que la autenticación se maneja con tokens. Los endpoints `/api/usuarios/signUp` y `/api/usuarios/login` son públicos, mientras que el resto requiere autenticación. Además, se configura la gestión de sesiones en modo **STATELESS**, lo que significa que la información de sesión no se almacenará en el servidor. Para validar los tokens en cada solicitud, se incorpora el `JwtFilter` antes del `UsernamePasswordAuthenticationFilter`.

La autenticación de usuarios es gestionada por `authenticationProvider`, que utiliza un `DaoAuthenticationProvider`. Para proteger las credenciales, se emplea **BCryptPasswordEncoder** con un factor de fuerza de **12**, asegurando que las contraseñas sean almacenadas de manera segura. Este proveedor se integra con `UsuarioDetailsService`, el cual **carga los datos del usuario desde la base de datos**.

La configuración de **CORS** permite solicitudes desde cualquier origen (\*), aunque en un entorno de producción debería restringirse a dominios específicos. Se habilitan los métodos HTTP esenciales (**GET, POST, PUT, DELETE y OPTIONS**) y se permite cualquier encabezado en las solicitudes. También se autoriza el uso de credenciales en las peticiones.

Finalmente, `authenticationManager` gestiona la autenticación utilizando la configuración proporcionada por `AuthenticationConfiguration`, asegurando que los usuarios puedan autenticarse correctamente mediante JWT.

## Validación de Tokens con JwtFilter

`JwtFilter` extiende `OncePerRequestFilter` y **actúa como un filtro de seguridad** que intercepta cada solicitud para validar la presencia y autenticidad del **token JWT**. Su función principal es **extraer al usuario autenticado** y asignarlo al contexto de seguridad de Spring Security.

En cada petición, el método `doFilterInternal` obtiene el header de autorización y verifica si contiene un token válido con el prefijo **Bearer**. Si el token está presente, se extrae y se utiliza `JWTService` para recuperar el nombre de usuario asociado.

Si el usuario aún no está autenticado en `SecurityContextHolder`, se carga su información desde la base de datos mediante `UsuarioDetailsService`. Luego, el filtro **valida el token** con `jwtService.validateToken()`, y si es correcto, **crea un UsernamePasswordAuthenticationToken** con los detalles del usuario y sus permisos, asignándolo al contexto de seguridad.

Finalmente, la solicitud sigue su flujo normal con `filterChain.doFilter(request, response)`, garantizando que **todas las solicitudes protegidas sean validadas automáticamente** antes de acceder a los recursos de la aplicación.

## Gestión de Autenticación y Seguridad en la Aplicación

### UsuarioDetails: Representación de Usuarios en Spring Security

La clase `UsuarioDetails` implementa `UserDetails` de Spring Security, permitiendo que la entidad `Usuario` se utilice en los procesos de autenticación y autorización. Su objetivo es **adaptar la información del usuario** a un formato compatible con Spring Security.

En su constructor, recibe un objeto `Usuario`, del cual **extrae el nombre de usuario y la contraseña**. Además, el método `getAuthorities()` asigna los roles correspondientes: **ADMIN** si el rol es `0`, o **USER** en cualquier otro caso. Otras verificaciones, como `isAccountNonExpired()`, `isAccountNonLocked()`, `isEnabled()` e `isCredentialsNonExpired()`, siempre retornan `true`, indicando que la cuenta está activa sin restricciones.

Gracias a esta implementación, Spring Security **puede gestionar los accesos según los roles de usuario**, restringiendo o permitiendo el uso de determinados recursos.

---

### UsuarioDetailsService: Carga de Información del Usuario

`UsuarioDetailsService` actúa como un puente entre la base de datos y Spring Security, proporcionando los detalles del usuario al sistema de autenticación. Implementa `UserDetailsService`, permitiendo a Spring Security validar usuarios a partir de su nombre de usuario.

El método `loadUserByUsername()` consulta la base de datos mediante `UsuarioRepositorio`. Si el usuario no existe, **lanza una excepción `UsernameNotFoundException`**. En caso contrario, crea una instancia de `UsuarioDetails`, la cual contiene la información de credenciales y permisos.

Este servicio garantiza que **solo los usuarios registrados puedan autenticarse**, integrando la base de datos con el sistema de seguridad.

---

## JWTService: Generación y Validación de Tokens JWT

La clase `JWTService` gestiona la autenticación mediante **JSON Web Tokens (JWT)**, encargándose de su **generación, validación y extracción de información**.

En su inicialización, genera una **clave secreta** (`secretKey`) utilizando el algoritmo **HmacSHA256**, la cual es utilizada tanto para **firmar** como para **validar** los tokens.

El método `generateToken()` crea un JWT con:

- El nombre de usuario como **sujeto**.
- La fecha de emisión (`issuedAt`).
- Una fecha de expiración (`expiration`) de aproximadamente **12 minutos**.
- La firma con la clave secreta, garantizando **seguridad e integridad**.

Para extraer información del token, `extractUserName()` recupera el nombre de usuario, mientras que `extractClaim()` permite acceder a otros datos dentro de los **claims**.

El método `validateToken()` verifica que:

1. **El usuario del token coincide con el registrado** en `UserDetails`.
2. **El token no ha expirado**.

Esto garantiza el **control de sesiones activas** y la protección de los recursos de la aplicación.

---

## UsuarioService: Validación de Credenciales y Manejo de Autenticación

La clase `UsuarioService` gestiona la autenticación de usuarios y la seguridad de credenciales.

El método `verificarUsuario()` recibe un `UsuarioDTO` y valida sus credenciales utilizando `AuthenticationManager`. Para ello, crea un `UsernamePasswordAuthenticationToken` y, si la autenticación es exitosa, **genera y devuelve un token JWT** mediante `JWTService`.

Para proteger las contraseñas, `nuevoUsuario()` utiliza `BCryptPasswordEncoder`, aplicando un cifrado robusto antes de almacenarlas en la base de datos, evitando que sean guardadas en texto plano.

Gracias a esta implementación, `UsuarioService` **asegura la autenticación mediante JWT, protege las credenciales de los usuarios y gestiona el inicio de sesión de manera segura.**

## UsuarioController

El endpoint `signUp` permite registrar nuevos usuarios. Primero, valida que el tipo de documento proporcionado exista en la base de datos. Si no es válido, responde con un error `404 Not Found`. Si la validación es exitosa, se crea un nuevo usuario utilizando el servicio `UsuarioService`, se guarda en la base de datos y se devuelve la información del usuario registrado en formato `UsuarioDTO` con un estado `200 OK`.

Por otro lado, el endpoint `logIn` gestiona la autenticación de usuarios. Recibe las credenciales (`userName` y `password`) en el cuerpo de la solicitud y las valida con el servicio `UsuarioService`. Si las credenciales son correctas, se genera un token JWT que es devuelto en la respuesta. En caso de autenticación fallida, se devuelve un mensaje de error con el estado `401 Unauthorized`, indicando que las credenciales son incorrectas.

## Resumen Implementación JWT y Spring Security

Para implementar autenticación con JWT y Spring Security, primero se creó `UsuarioDetails` y `UsuarioDetailsService`, que permiten a Spring Security gestionar la autenticación cargando los detalles del usuario desde la base de datos. Luego, en `SecurityConfig`, se configuraron las reglas de seguridad, habilitando CORS, desactivando CSRF y asegurando los endpoints con autenticación basada en JWT.

El servicio `JWTService` se encargó de la generación y validación de tokens, utilizando una clave secreta con `HmacSHA256`. `UsuarioService` implementó la lógica de autenticación, validando credenciales con `AuthenticationManager` y generando tokens para sesiones exitosas. Finalmente, `JwtFilter` intercepta cada solicitud, extrae el token y valida su autenticidad, asegurando que solo usuarios autenticados accedan a los recursos protegidos.

## Lista de Reproducción Que se Siguió

Enlace:

<https://youtube.com/playlist?list=PLsyebzWxl7qbKoSgR5ub6joll8-ocxCF&si=c80tWCq1Wbr2Kpmo>



Específicamente desde el video #29 hasta el #38