



DIPLOMADO  
**Desarrollo de sistemas con  
tecnología Java**

**Módulo 11**  
Spring Security

*Mtro. Alfonso Gregorio Rivero Duarte*



# Acuerdos

- ★ Recesos de 10 minutos.
- ★ Preguntar en cualquier momento.
- ★ La participación en clase es muy importante.

# Consideraciones

- ★ El moderador será el encargado de las sesiones de preguntas.
- ★ Hacer uso del ícono de  levantada para pedir la palabra.
- ★ Si por alguna circunstancia entra tarde a la sesión, no activar el micrófono para no interrumpir la sesión.
- ★ Se prohíbe la violencia y la agresión verbal en contra de cualquier persona.



# ¿Quién soy yo?

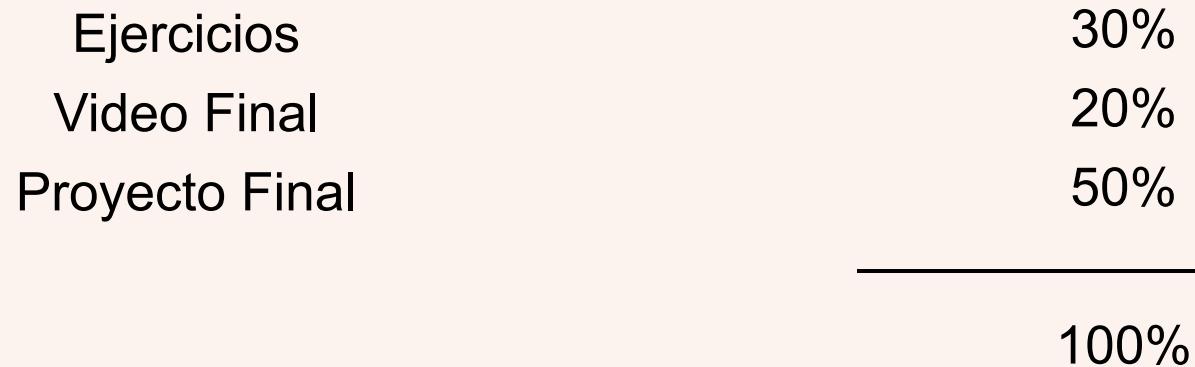
- ★ Experiencia profesional
- ★ Certificaciones
- ★ Academia
- ★ Hobbies
- ★ ¿Por qué elegí aportar con el diplomado?
- ★ Otros...



# Módulos del Diplomado

Módulo	Duración (hrs)
Programación orientada a objetos con Java	40
Manejo de bases de datos con Java	20
Principios y patrones de diseño	20
Persistencia con Hibernate	20
JavaServer Faces	25
Enterprise Java beans y Servicios web	15
Introducción al desarrollo de aplicaciones empresariales con Spring Boot Framework	20
Persistencia con Spring Data	20
Desarrollo de aplicaciones Web con Spring Boot MVC	20
API RESTful con Spring Boot	20
Spring Security	20
	
<b>Total</b>	<b>240</b>

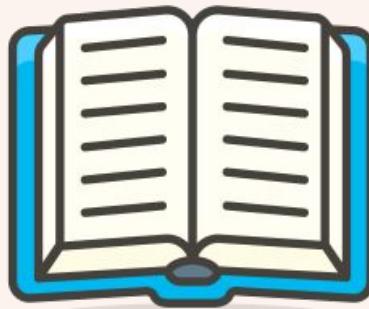
# Evaluación



*Todos los ejercicios y programas deben de subirse al apartado especial que se encuentra en Moodle.*



# Bibliografía



# Temario

- 1. ¿Qué es Spring Security?**
  - Introducción
  - Arquitectura de Spring Security
  - Añadir soporte de Spring Security en Spring Boot



- 2. Configuración Spring Security**
  - Usuario y contraseña por defecto (en memoria)
  - Personalización de usuario y contraseña
  - Renderizar atributos de la sesión con Thymeleaf
  - Creación de formulario de login personalizado
  - Cerrar sesión de usuario

# Temario

## 3. Recuperación de usuarios y roles desde una base de datos

- Creación de las clases de usuario y rol
- Conectar la autenticación con la BD
- Implementar la función de registro de usuarios



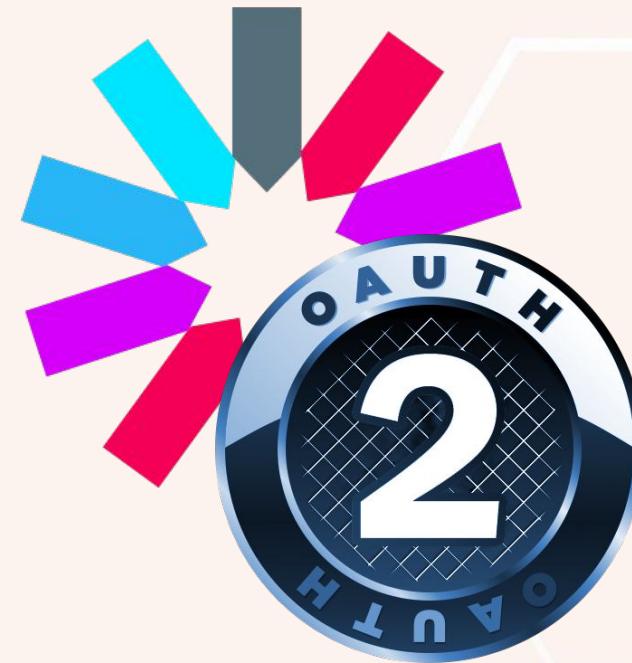
## 4. Recuperación de usuarios y roles respaldados por LDAP

- ¿Qué es LDAP?
- Instalación y configuración de Apache Directory Studio
- Configuración Básica de LDAP en Spring
- Trabajando con UnboundID LDAP

# Temario

## 5. Spring Security y oAuth2 con JWT

- Arquitectura de Spring Security con JWT y Thymeleaf
- Arquitectura de Spring Security con JWT y Rest
- Spring Security con OAuth2.0 y GitHub
- Spring Security con OAuth2.0 y Keycloak



# Objetivos Generales del Módulo de Seguridad

- El alumno trabajará con las últimas versiones de Spring Boot e integrará seguridad a las aplicaciones.
- El alumno trabajará con Spring Security framework 6.0 directamente, aprenderá a configurarlo y trabajará con diferentes tipos de autenticación
  - Básica
  - LDAP
  - JWT
  - OAuth2.0

# 1. ¿Qué es Spring Security?



## 1.1 Introducción

# Application Security 101

La seguridad es un tema muy muy largo.

Hacer aplicaciones seguras envuelve soluciones tecnológicas, adoptar buenas prácticas, y por supuesto, interés personal y físico.

Seguridad comúnmente envuelve analizar riesgos y luego implementar acciones de mitigación.

En 2016, para la elección presidencial en USA, WikiLeaks publicó correos obteniendo información del partido demócrata.

- Dicha revelación de información fue muy dañina!

# Application Security 101

Correos hackeados envolvían 3 diferentes áreas:

- Servidor DNC (Democratic National Committee)
- Correo John Podesta
- Correo Hillary Clinton

La inteligencia de USA cree que los correos hackeados fueron obtenidos por hackers rusos via cyber ataques.

Esto significa se obtuvo los accesos al servidor DNC via internet  
Phishing

Acciones de mitigación

Parches O.S. para prevenir exploits (usaban windows XP)  
Longitud contraseñas / requerimientos de complejidad



# Security Audit Frameworks / Certifications

**PCS-DSS:** Payment Card Industry Data Security Standard

Aplicable a organizaciones que procesan tarjetas crédito o débito

**SOX:** Sarbanes-Oxley

Para empresas USA que cotizan en bolsa

**HIPAA:** Health Insurance Portability and Accountability Act

Industria Médica

**SSAE-16:** Statement on Standards for Attestation Engagements (SSAE) No. 16

CPA - Guía autorizada para informar sobre organizaciones de servicios.

# ¿Por qué “seguridad”?

## ¿Qué es Seguridad?

La seguridad sirve para proteger sus datos y su lógica empresarial dentro de su aplicación web.

## Seguridad es un requerimiento no funcional

La seguridad es muy importante al igual que la escalabilidad, el rendimiento y la disponibilidad.

Ningún cliente me preguntará específicamente que necesita seguridad.

## Seguridad desde la fase de desarrollo

La seguridad debe considerarse desde la fase de desarrollo junto con la lógica empresarial.

## Diferentes tipos de Seguridad

La seguridad para una aplicación web se implementará de diferentes maneras, como mediante firewalls, HTTPS, SSL, autenticación, autorización, etc.

## ¿Por qué es importante la seguridad?

La seguridad no significa sólo perder datos o dinero, sino también la marca y la confianza de tus usuarios que has construido a lo largo de años.

## Evitar los ataques más comunes

Al utilizar la seguridad, también debemos evitar los ataques de seguridad más comunes como CSRF, autenticación rota dentro de nuestra aplicación.

# Vulnerabilidades Web Comunes

OWASP - Open Web Application Security Project <https://owasp.org>

Organización sin fines de lucro que trabaja para mejorar la seguridad del software.

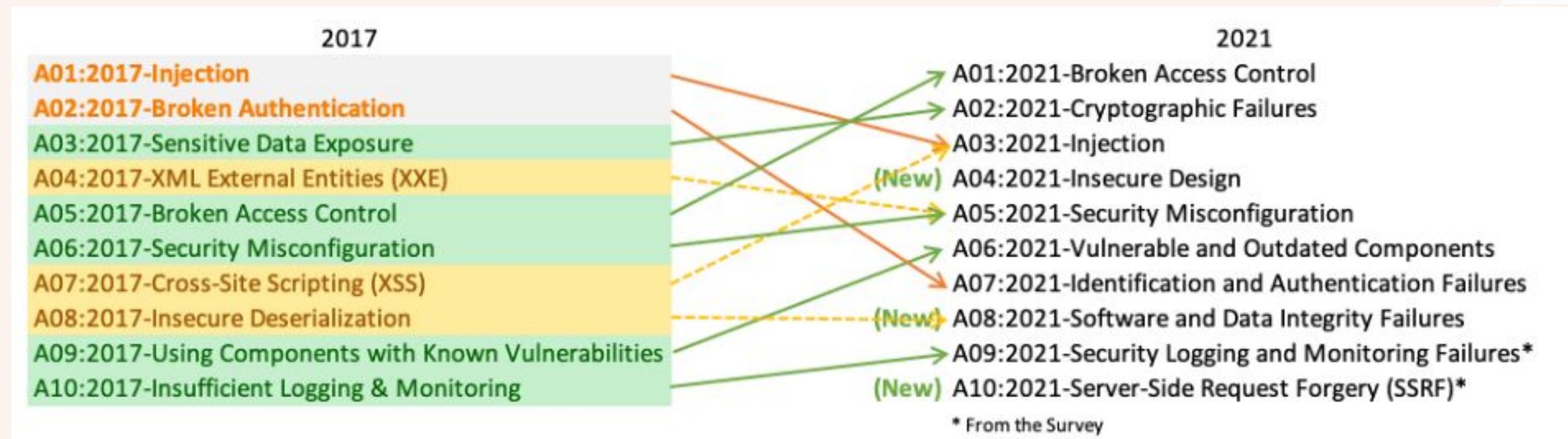
Proporciona:

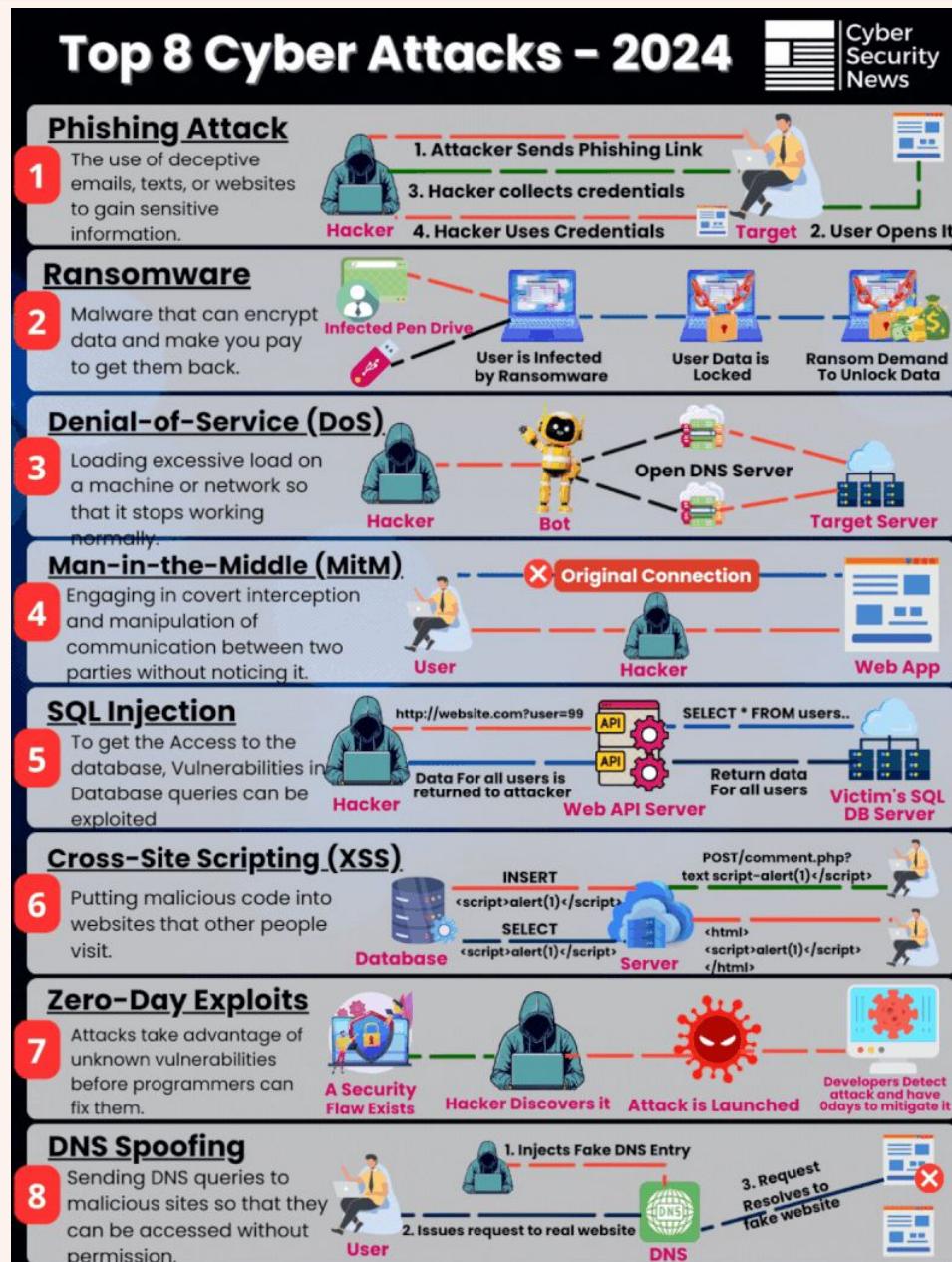
- Herramientas y recursos
- Comunidad y networking
- Educación y entrenamiento

OWASP Los 10 principales riesgos de seguridad de aplicaciones web

<https://owasp.org/www-project-top-ten/>

# OWASP Los 10 principales riesgos de seguridad de aplicaciones web





# Vulnerabilidades

Spring Security ha incorporado soporte para abordar varias vulnerabilidades comunes

- Secuencias de comandos entre sitios (XSS)
- Falsificación de solicitudes entre sitios (CSRF)
- Encabezados de respuesta HTTP de seguridad
  - Se puede configurar una variedad de encabezados para mejorar la seguridad del navegador.
- Redirigir a HTTPS

# Objetivos de la Seguridad

## SEGURIDAD

¿Cómo puedo implementar seguridad en mis aplicaciones web/móviles para que no haya violaciones de seguridad en mis aplicaciones?

## PASSWORDS

¿Cómo almacenar contraseñas, validarlas, codificarlas y decodificarlas utilizando algoritmos de cifrado estándar de la industria?

## USERS & ROLES

¿Cómo mantener la seguridad a nivel de usuario en función de sus roles y subvenciones asociadas a ellos?

## MULTIPLE LOGINS

¿Cómo puedo implementar un mecanismo en el que el usuario inicie sesión, solo use y comience a usar mi aplicación?



## FINE GRAINED SECURITY

¿Cómo puedo implementar seguridad en cada nivel de mi aplicación usando reglas de autorización?

## CSRF & CORS

¿Qué son los ataques CSRF y las restricciones CORS? ¿Cómo superarlos?

## JWT & OAUTH2

¿Qué es JWT y OAUTH2? ¿Cómo puedo proteger mi aplicación web usándolos?

## PREVENTING ATTACKS

¿Cómo prevenir ataques de seguridad como Fuerza Bruta, robo de datos, fijación de sesiones?

# ¿Qué es Spring Security?

Spring Security es el módulo del proyecto Spring para incorporar seguridad de acceso a las aplicaciones hechas con Spring Boot. Permite controles de acceso por URL entre otras muchas opciones y es más que suficiente para proteger tu programa.



# Spring Security

Spring Security es una librería que forma parte del universo del proyecto Spring. Spring tiene más de 25 sub proyectos o módulos que aportan funcionalidad que las aplicaciones pueden utilizar si lo creen conveniente. En este caso Spring Security trata de agrupar todas las funcionalidades de control de acceso de usuarios sobre proyectos Spring.

El control de acceso permite limitar las opciones que pueden ejecutar un determinado conjunto de usuarios o roles sobre la aplicación. En esta dirección, Spring Security controla las invocaciones a la lógica de negocios o limita el acceso de peticiones HTTP a determinadas URLs.

# Spring Security

Para ello, el programador debe realizar una configuración sobre la aplicación indicando a Spring Security cómo debe comportarse la capa de seguridad. Y aquí es una de las grandes ventajas de Spring Security ya que permite realizar toda una serie de parametrizaciones y ajustes para un gran abanico de posibilidades, permitiendo que el módulo se adapte bien a casi cualquier escenario de aplicaciones realizadas con Spring IoC o Spring Boot.

Si la implementación de Spring Security no fuera suficiente, aun así, proporciona toda una serie de interfaces Java que el programador puede implementar para cambiar el modo de comportarse de una determinada funcionalidad, como el ir a buscar los usuarios a una fuente de datos muy particulares.

# Spring Security

Siempre que se pueda, conviene usar las implementaciones provenientes en la librería de Spring Security, ya que es código muy maduro ampliamente testado y además se mantiene actualizado por la comunidad cuando se publica un aviso de seguridad o ante cualquier publicación un nuevo estándar como pasó al liberarse el protocolo HTTP 2.

En definitiva, Spring Security es el método más conveniente para incorporar una capa de seguridad en donde se desea que sólo algunos usuarios tengan acceso a métodos y controladores de una aplicación Spring en base al uso de roles de usuario.

# Spring Security

Además, al ser un proyecto maduro y ampliamente utilizado, es posible encontrar implementaciones incluidas de serie para los principales sistemas de seguridad de usuarios más desplegados en el mundo del desarrollo de software o en la administración de sistemas de TI, como LDAP o Kerberos entre otros.

# Características Spring Security

- Código y diseño del componente excelente
- Gran uso en proyectos de referencia dentro del sector Java
- Solución madura (con versiones estables desde el año 2003).
- Facilidad de configuración y parametrización (gracias al uso de namespace para la configuración y al uso del patrón dependency Injection).

# Características Spring Security

- Integración con los sistemas legacy de autenticación más importantes del mercado: BBDD, LDAP, CAS (para single sign-on), gestión de certificados, etc.
- Como todo componente realizado dentro de la familia Spring es de fácil uso y fácil extensibilidad.
- Gran cantidad de documentación y ejemplo de soporte.

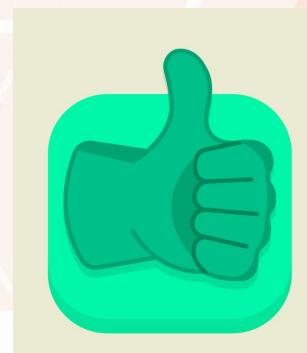


# Pros de utilizar Spring Security

- **Protección contra ataques comunes:** Seguridad de primavera proporciona una protección contra ataques comunes como la inyección SQL y la falsificación de solicitudes. Esto ayuda a garantizar la seguridad de su aplicación y los datos de sus usuarios.
- **Personalización:** Seguridad de primavera es altamente personalizable y se integra fácilmente con otras tecnologías de Spring, como Spring MVC y Spring Boot. Esto significa que se adapta a las necesidades específicas de su proyecto.

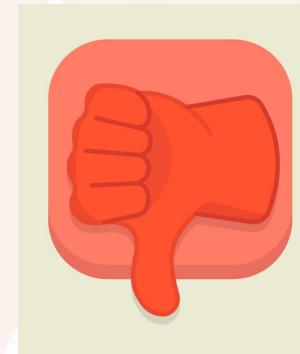
# Pros de utilizar Spring Security

- **Variedad de mecanismos de autenticación:** Spring Security proporciona una gran variedad de mecanismos de autenticación, como autenticación endosada en formularios, autenticación en token y autenticación basada en certificados. Esto significa que puede elegir el mecanismo que mejor se adapte a sus necesidades de seguridad.
- **Comunidad activa:** Seguridad de primavera tiene una gran comunidad activa y una amplia gama de documentación y recursos en línea, lo que facilita la aplicación y la resolución de problemas.



# Contras de utilizar Spring Security

- **Complejidad:** Seguridad de primavera puede ser complejo bastante complejo de configuración y alimentación. Puede requerir un aprendizaje adicional para implementar lo bien en un proyecto.
- **Sobrecarga de recursos:** El uso de Spring Security puede tener un impacto en el rendimiento de su aplicación por la sobrecarga de recursos.



# 1. ¿Qué es Spring Security?

1.2 Arquitectura de Spring Security



# Contexto de Seguridad

En el contexto computacional, tenemos varios niveles de seguridad:

- Hardware
- OS
- Database
- Message Brokers
- Network Security
- **Application Security**

Spring security se enfoca en Application Security

Provee:

- Protección para exploits comunes
- Integración con productos de seguridad externos, i.e. LDAP
- Provee utilerias para Password Encoding

# Términos clave de seguridad de la aplicación

**Identidad** - Un único actor, típicamente un individuo. a.k.a. user

**Credenciales** - usualmente un id y un password

**Autenticación** - es cómo la aplicación verifica la identidad del solicitante

- Spring Security tiene una variedad de métodos para la autenticación
- Típicamente el usuario provee las credenciales, que son validadas

**Autorización** - ¿el usuario puede realizar una acción?

- Usando la identidad del usuario, spring security determina si se está autorizado a realizar dicha acción

# Términos clave de seguridad de la aplicación

## AUTHENTICATION

En la autenticación, se verifica la identidad de los usuarios para proporcionar acceso al sistema.

Autenticación (AuthN) realizada antes de la autorización.

Por lo general, necesita los datos de inicio de sesión del usuario.

Si la autenticación falla, normalmente obtendremos una respuesta de error 401.

Por ejemplo, como cliente/empleado de un banco, para realizar acciones en la aplicación, debemos demostrar nuestra identidad.

## AUTHORIZATION

En la autorización, se verifican las autoridades de la persona o usuario para acceder a los recursos.

La autorización (AuthZ) siempre ocurre después de la autenticación.

Necesita privilegios o roles de usuario.

Si la autorización falla normalmente obtendremos una respuesta de error 403.

Una vez que haya iniciado sesión en la aplicación, mis roles y las autoridades decidirán qué tipo de acciones puedo realizar.

# Authentication Providers

Authentication Providers - Verifican la identidad del usuario

Soportados por Spring Security:

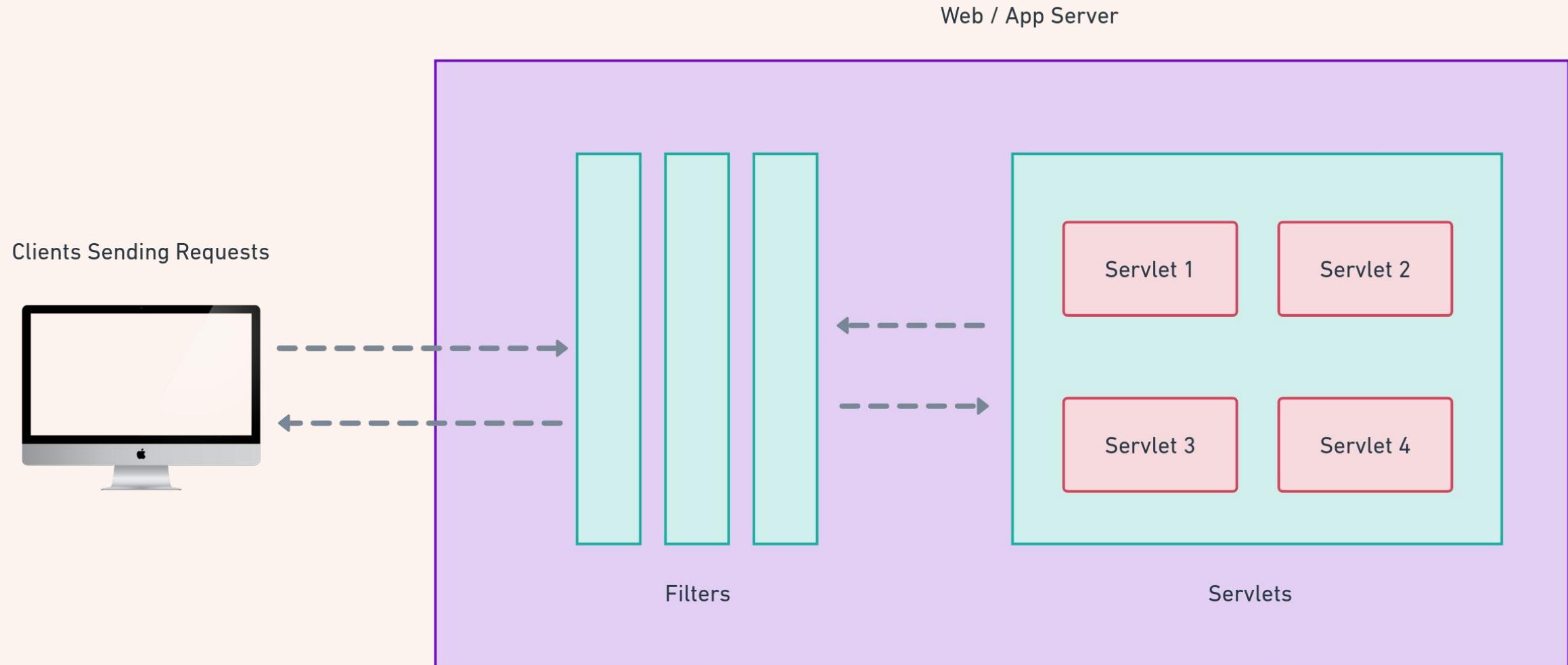
- In Memory
- JDBC / Database
- Custom
- Keycloak
- ACL (Access Control List)
- OpenID
- LDAP / Active Directory
- CAS

# Password Storage

Spring Security soporta varios métodos para almacenaje y verificación de passwords

- NoOpPasswordEncoder (no recomendado para producción)
- StandardPasswordEncoder (no recomendado para producción)
- Pbkdf2PasswordEncoder
- BCryptPasswordEncoder
- SCryptPasswordEncoder
- Argon2PasswordEncoder

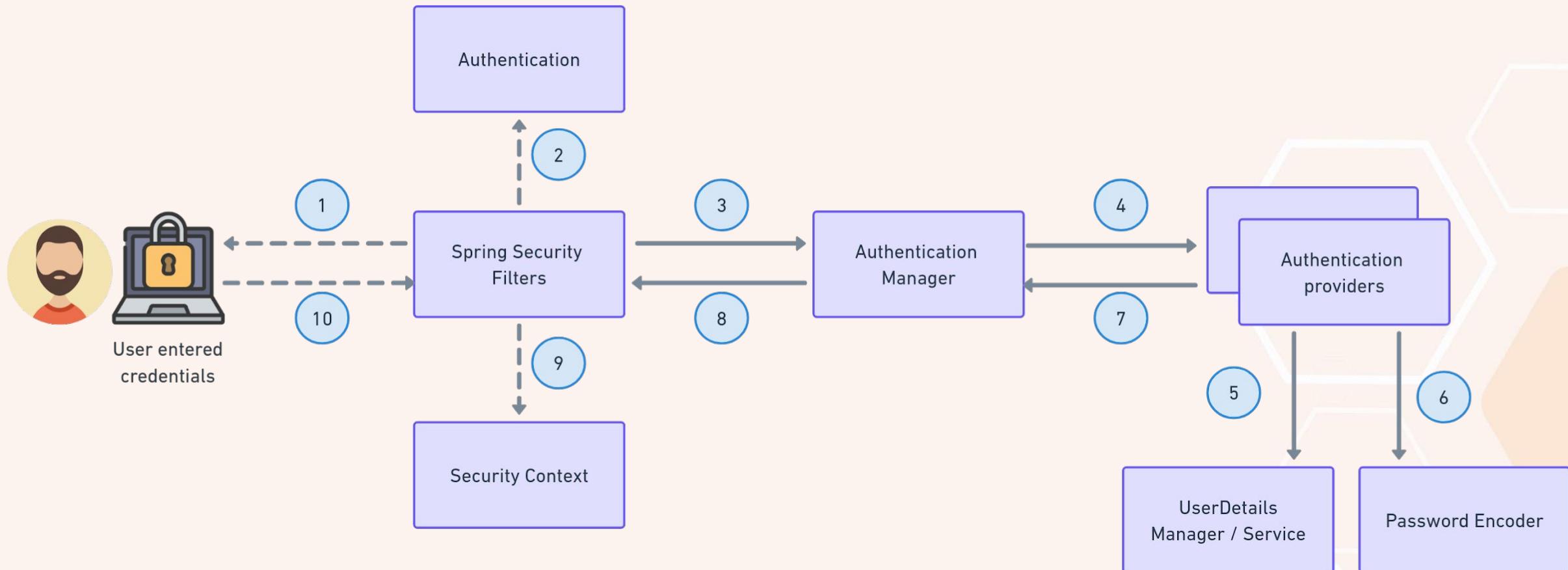
# Servlets y filtros



# Servlets y filtros

- Escenario típico dentro de una aplicación web
  - En las aplicaciones web de Java, Servlet Container (servidor web) se encarga de traducir los mensajes HTTP para que el código Java los entienda. Uno de los contenedores de servlets más utilizados es Apache Tomcat. Servlet Container convierte los mensajes HTTP en ServletRequest y los entrega al método Servlet como parámetro. De manera similar, ServletResponse regresa como salida al Servlet Container desde Servlet. Entonces, todo lo que escribimos dentro de las aplicaciones web Java está controlado por servlets.
- Rol de Filtros
  - Se pueden utilizar filtros dentro de las aplicaciones web Java para interceptar cada solicitud/respuesta y realizar un trabajo previo antes de nuestra lógica empresarial. Entonces, usando los mismos filtros, Spring Security aplica la seguridad en función de nuestras configuraciones dentro de una aplicación web.

# Flujo interno de Spring Security



# Flujo interno de Spring Security

- **Filtros de Spring Security**

Una serie de filtros de Spring Security interceptan cada solicitud y trabajan juntos para identificar si se requiere autenticación o no. Si se requiere autenticación, lleve al usuario a la página de inicio de sesión o utilice los detalles existentes almacenados durante la autenticación inicial.

- **Authentication**

Los filtros como UsernamePasswordAuthenticationFilter extraerán el nombre de usuario/contraseña de la solicitud HTTP y prepararán el objeto de tipo de autenticación. Porque la autenticación es el estándar central para almacenar detalles de usuarios autenticados dentro del marco de Spring Security.

- **AuthenticationManager**

Una vez recibida la solicitud del filtro, delega la validación de los datos del usuario a los proveedores de autenticación disponibles. Dado que puede haber varios proveedores dentro de una aplicación, es responsabilidad del AuthenticationManager administrar todos los proveedores de autenticación disponibles.

- **AuthenticationProvider**

AuthenticationProviders tiene toda la lógica central para validar los detalles del usuario para la autenticación.

- **UserDetailsManager / UserDetailsService**

UserDetailsManager / UserDetailsService ayuda a recuperar, crear, actualizar y eliminar los detalles del usuario de la base de datos/sistemas de almacenamiento

- **PasswordEncoder**

Interfaz de servicio que ayuda a codificar y cifrar contraseñas. De lo contrario, es posible que tengamos que vivir con contraseñas de texto plano.

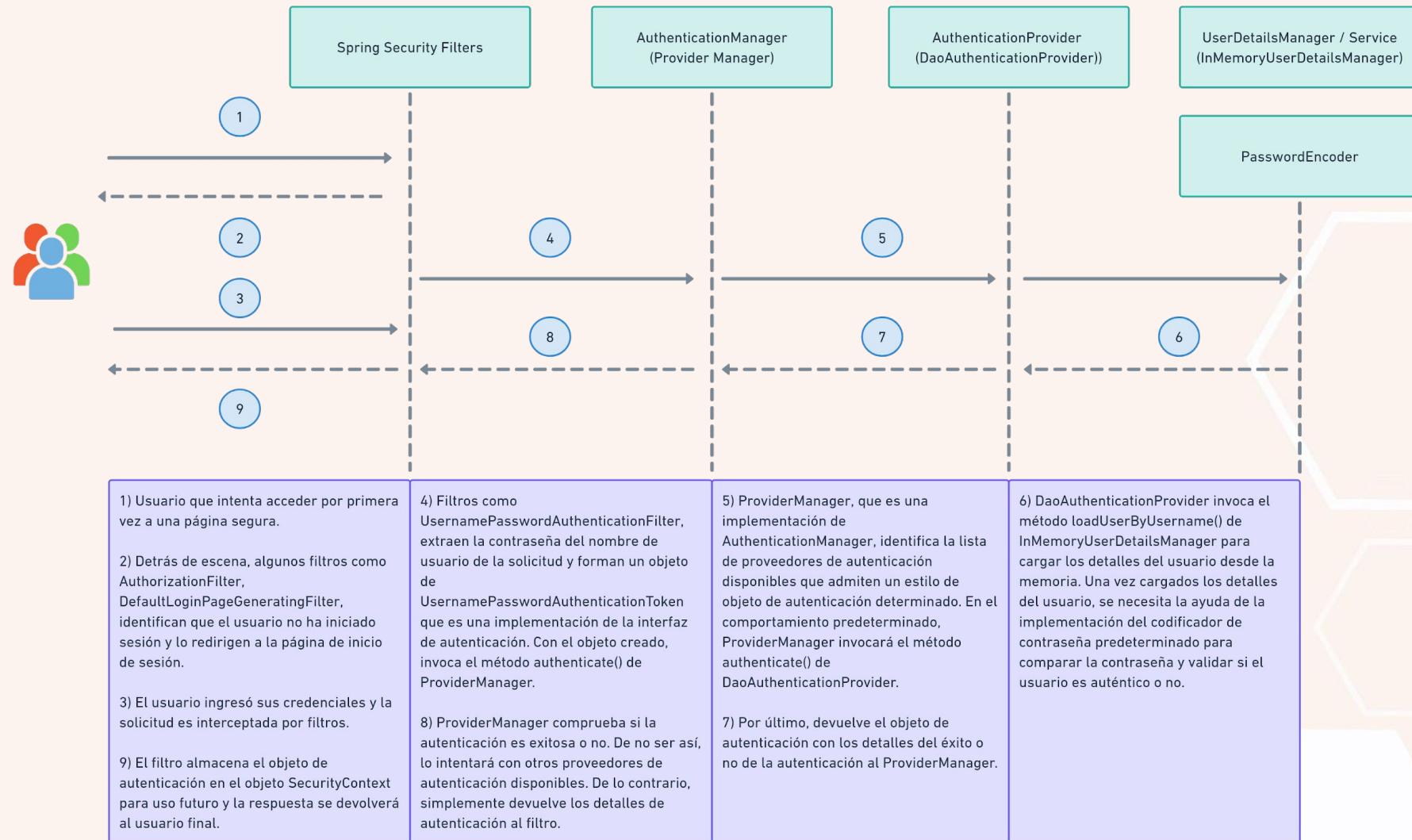
- **SecurityContext**

Una vez que se ha autenticado la solicitud, la autenticación generalmente se almacenará en un SecurityContext local de subproceso administrado por SecurityContextHolder. Esto ayuda durante las próximas solicitudes del mismo usuario.

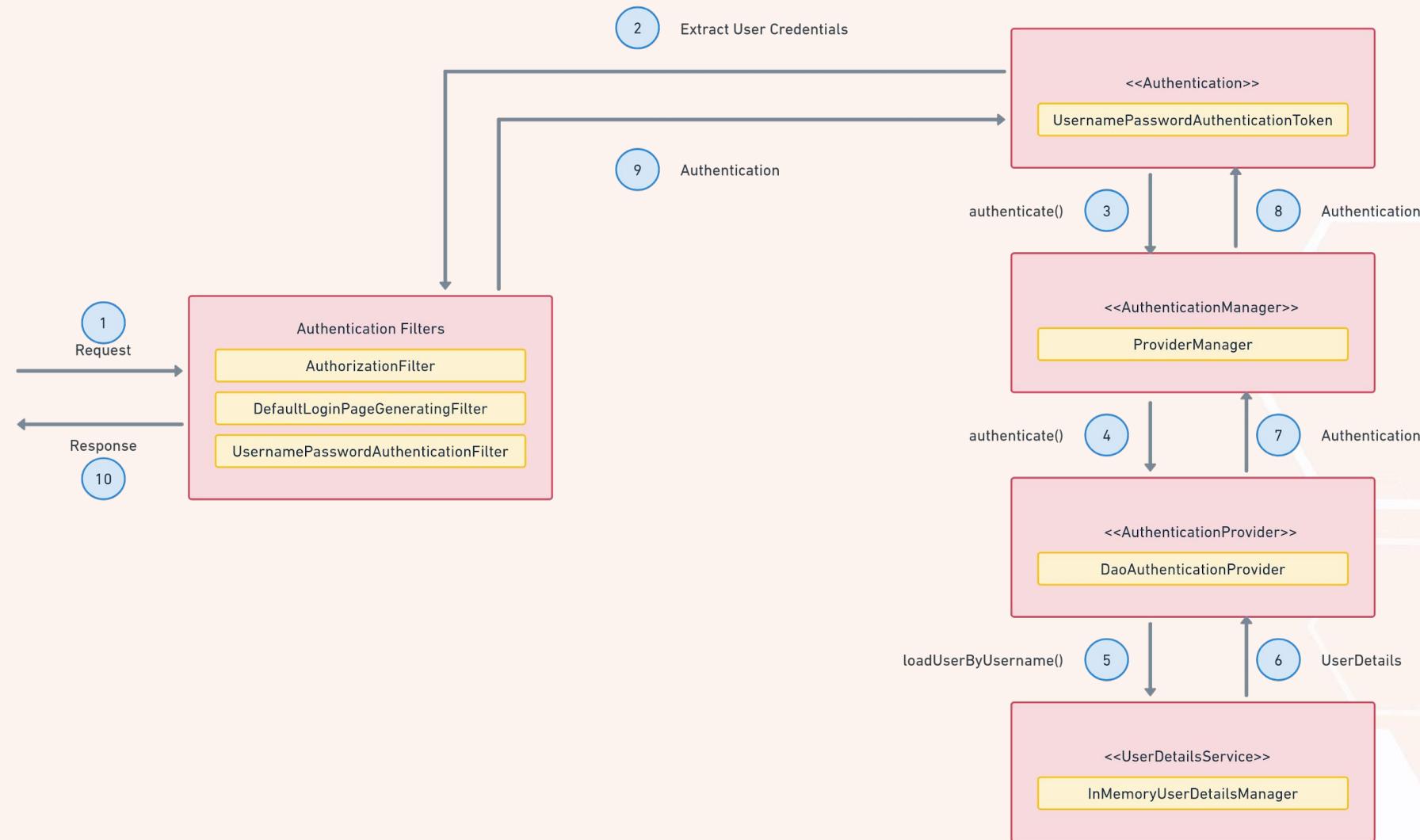
# Flujo interno de Spring Security

- **Authentication Filter:** un filtro para un tipo de autenticación específico en la cadena de filtro de seguridad de resorte. (es decir, autenticación básica, recuérdame cookie, etc.)
- **Authentication Manager:** interfaz API estándar utilizada por filtro
- **Authentication Provider:** la implementación de la autenticación - (en memoria, base de datos, etc.)
- **User Details Service:** servicio para proporcionar información sobre el usuario
- **Password Encoder:** servicio para cifrar y verificar contraseñas
- **Security Context:** contiene detalles sobre entidad autenticada

# Flujo interno de Spring Security



# Flujo interno de Spring Security



# Interfaces y Clases Importantes

Interfaz principal que carga datos específicos del usuario.

UserDetailsService  
(Interface)

Una extensión de UserDetailsService que brinda la capacidad de crear nuevos usuarios y actualizar los existentes.

UserDetailsManager  
(Interface)

Clases de implementación proporcionadas por Spring Security

InMemoryUser  
DetailsManager

JdbcUser  
DetailsManager

LdapUser  
DetailsManager

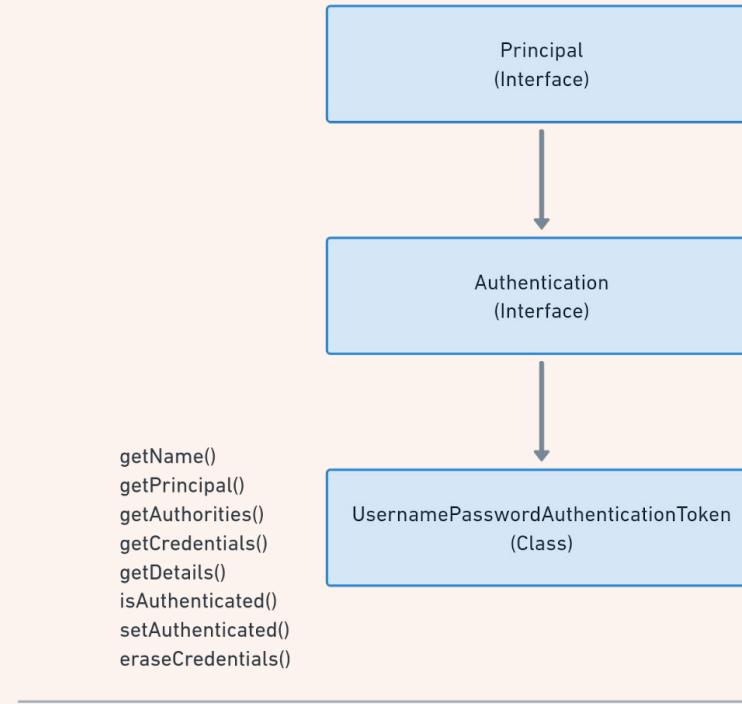
loadUserByUsername(String username)

createUser(UserDetails user)  
updateUser(UserDetails user)  
deleteUser(String username)  
changePassword(String oldPwd, String newPwd)  
userExists(String username)

UserDetails

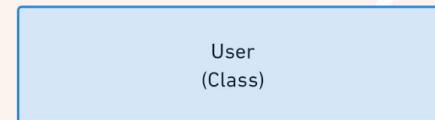
Todas las interfaces y clases anteriores utilizan una interfaz UserDetails y su implementación que proporciona información básica del usuario.

# Relación entre clases



La autenticación es el tipo de retorno en todos los escenarios en los que intentamos determinar si la autenticación es exitosa o no. Como dentro de AuthenticationProvider y AuthenticationManager

¿Por qué tenemos dos formas distintas de almacenar los datos de inicio de sesión del usuario?



getPassword()  
getUsername()  
getAuthorities()  
isAccountNonExpired()  
isAccountNonLocked()  
isEnabled()  
isCredentialsNonExpired()  
eraseCredentials()

UserDetails es el tipo de devolución en todos los escenarios en los que intentamos cargar la información del usuario desde los sistemas de almacenamiento. Como dentro de UserDetailService y UserDetailsManager

# Filtros

Muchas veces tendremos situaciones en las que necesitaremos realizar algunas actividades de mantenimiento durante el flujo de autenticación y autorización. Algunos ejemplos de este tipo son:

- Validación de entrada
- Rastreo. Auditoría e informes
- Registro de entradas como dirección IP, etc.
- Cifrado y descifrado
- Autenticación multifactor mediante OTP

Todos estos requisitos se pueden manejar utilizando filtros HTTP dentro de Spring Security. Los filtros son conceptos de servlet que también se aprovechan en Spring Security.

# Filtros

Ya vimos algunos filtros integrados del marco Spring Security como UsernamePasswordAuthenticationFilter, DefaultLoginPageGeneratingFilter, etc.

Un filtro es un componente que recibe solicitudes, procesa su lógica y la entrega al siguiente filtro de la cadena.

Spring Security se basa en una cadena de filtros de servlet. Cada filtro tiene una responsabilidad específica y dependiendo de la configuración se agregan o eliminan filtros. También podemos agregar nuestros filtros personalizados según la necesidad.

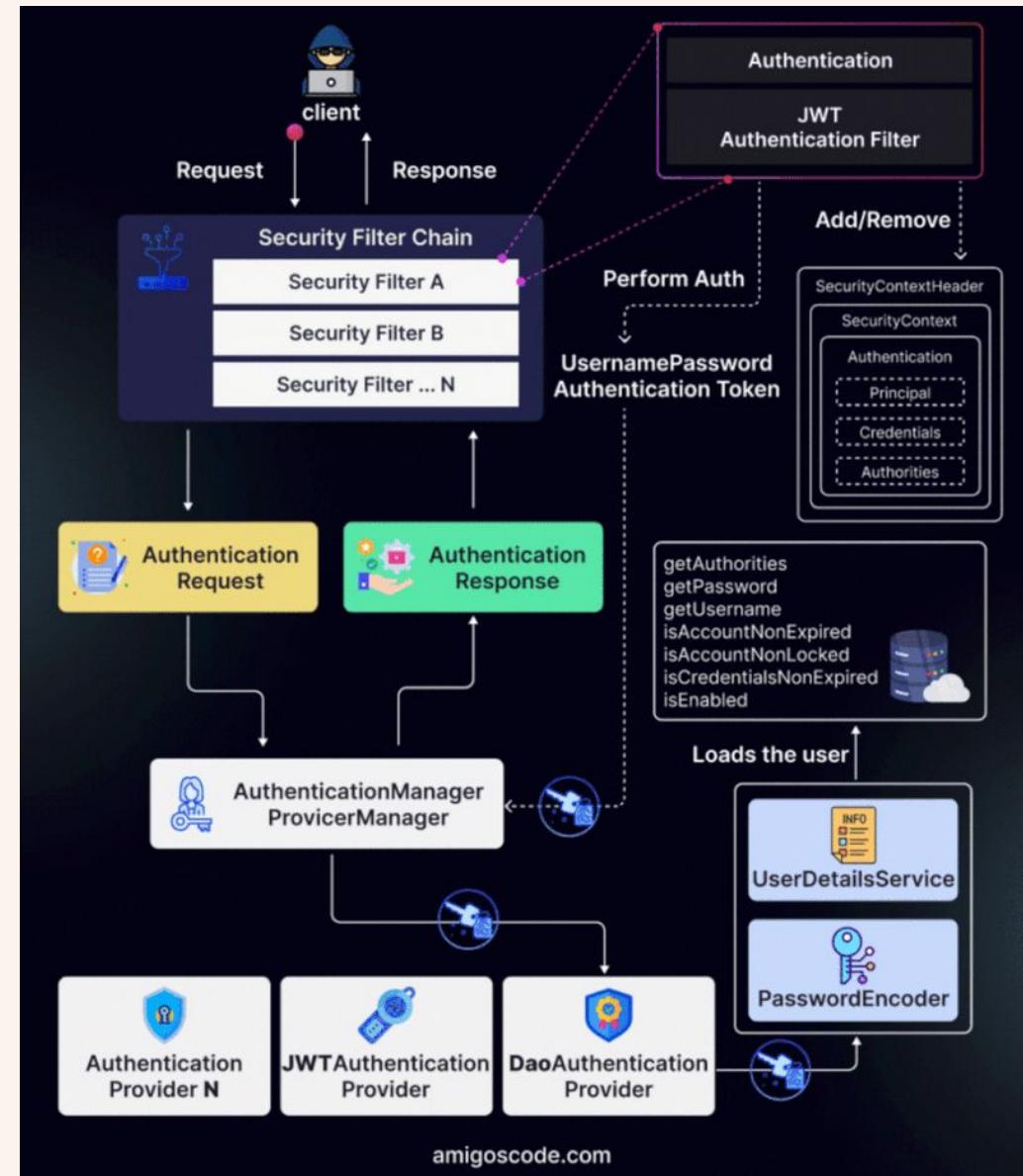
# Filtros

- ✓ We can always check the registered filters inside Spring Security with the below configurations,
  1. `@EnableWebSecurity(debug = true)` – We need to enable the debugging of the security details
  2. Enable logging of the details by adding the below property in application.properties
    - `logging.level.org.springframework.security.web.FilterChainProxy=DEBUG`

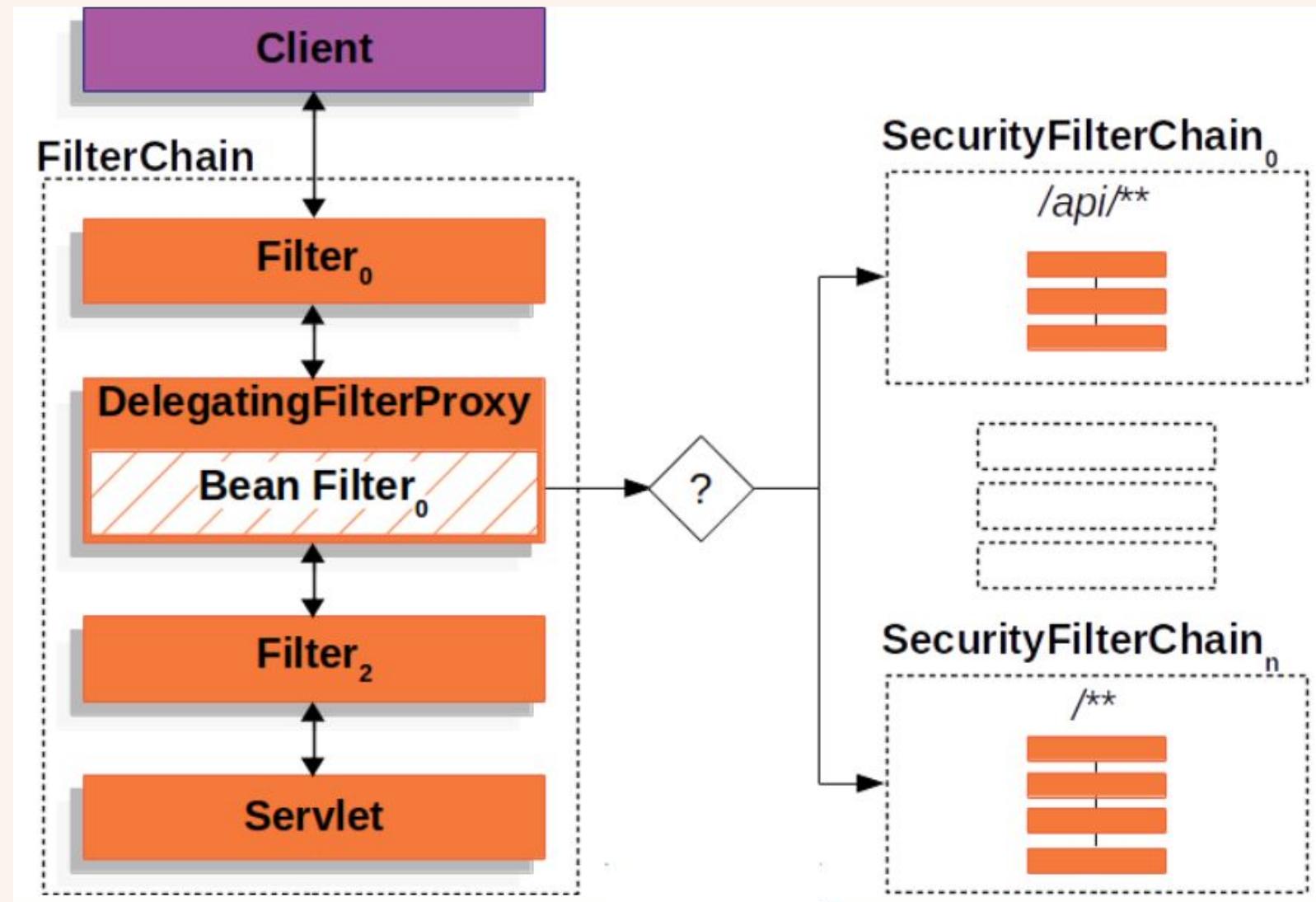
Se adjuntan algunos de los filtros internos de Spring Security que se ejecutan en el flujo de autenticación.

Security filter chain: [  
DisableEncodeUrlFilter  
WebAsyncManagerIntegrationFilter  
SecurityContextHolderFilter  
HeaderWriterFilter  
CorsFilter  
CsrfFilter  
LogoutFilter  
UsernamePasswordAuthenticationFilter  
DefaultLoginPageGeneratingFilter  
DefaultLogoutPageGeneratingFilter  
BasicAuthenticationFilter  
RequestCacheAwareFilter  
SecurityContextHolderAwareRequestFilter  
AnonymousAuthenticationFilter  
SessionManagementFilter  
ExceptionTranslationFilter  
FilterSecurityInterceptor]  
]

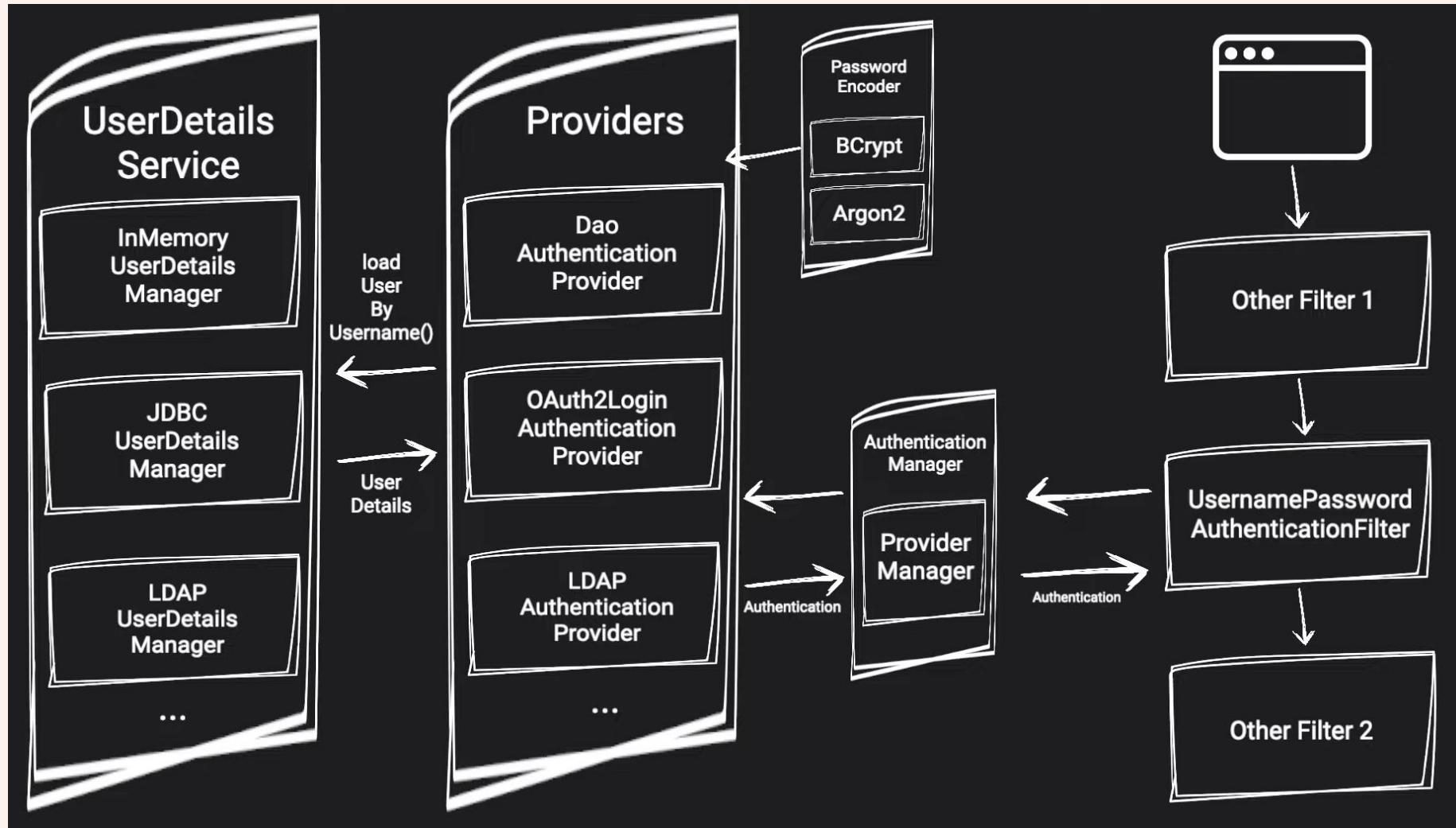
# Filtros



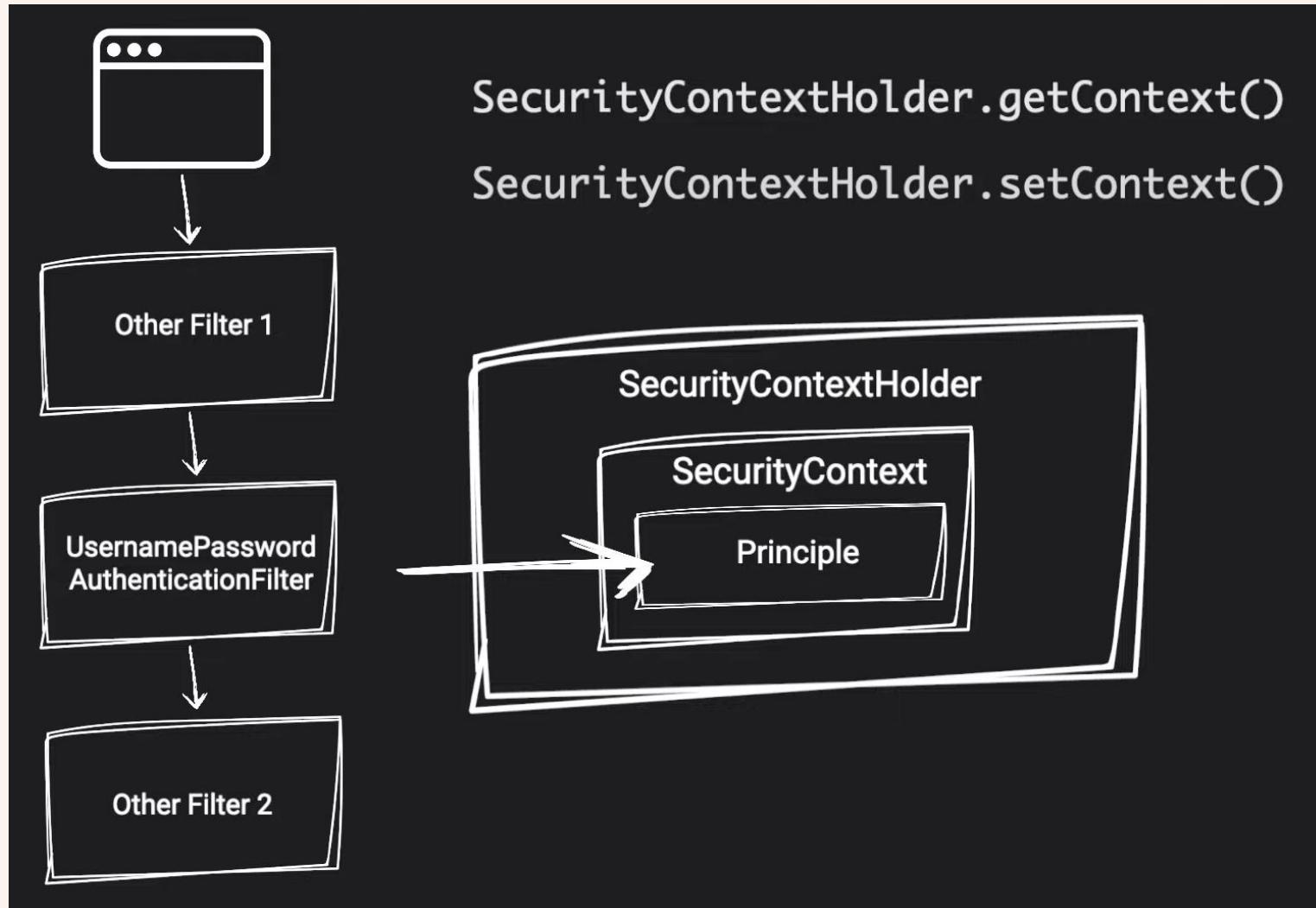
# Filtros



# Filtros



# Filtros



# 1. ¿Qué es Spring Security?

1.3 Añadir soporte de Spring Security en Spring Boot



file: [spring-security-hello-world.zip](#)

# Añadir soporte de Spring Security en Spring Boot

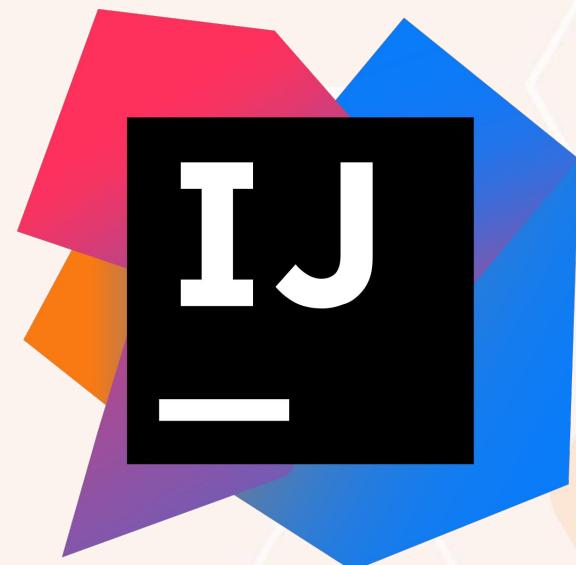
Para poder empezar, tenemos 2 opciones de trabajo:

1. Utilizar IntelliJ Ultimate Edition
2. Utilizar IntelliJ Community Edition

En ambos casos, requerimos trabajar con spring initializer

- <https://start.spring.io/>

A continuación se ejemplifican ambos casos:

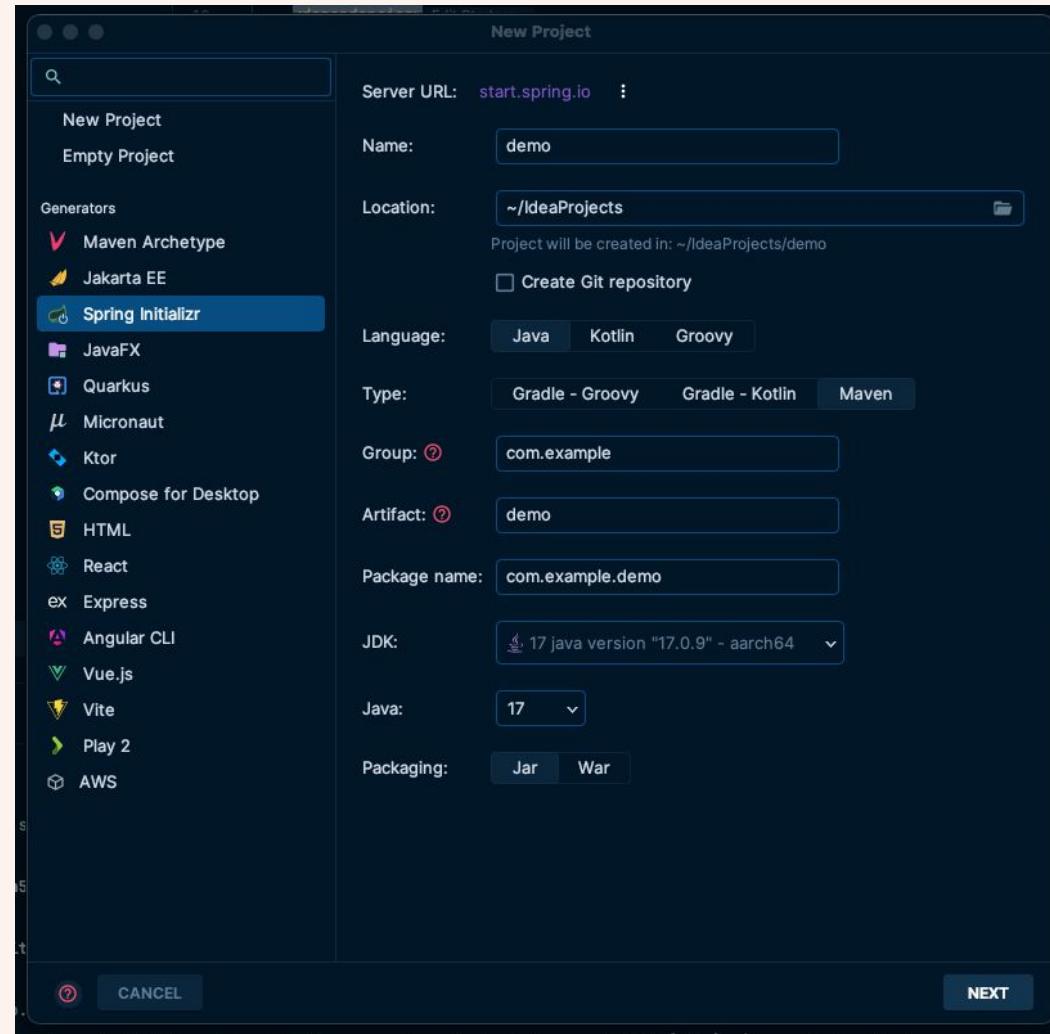


# Comparativa versiones IntelliJ



IntelliJ IDEA Ultimate	IntelliJ IDEA Community Edition
✓ Spring (Spring MVC, Spring Boot, Spring Integration, Spring Security and more)	
✓ Spring Cloud	
✓ Java EE (JSF, JAX-RS, CDI, JPA, etc)	
✓ Jakarta EE (JSF, JAX-RS, CDI, JPA, etc)	
✓ Micronaut, Quarkus, Helidon	
✓ Hibernate, JPA	
✓ Ktor	✓ JavaFX
✓ JavaFX	✓ Swing (incl. UI Designer)
✓ Swing (incl. UI Designer)	✓ Android (includes the Android Studio's functionality)
✓ Android (includes the Android Studio's functionality)	✓ GWT
✓ Thymeleaf, Freemarker, Velocity	✓ Swing (incl. UI Designer)
✓ Liquid, Go Template, Mustache, Qute	✓ Android (includes the Android Studio's functionality)
✓ AspectJ, OSGI	
✓ Akka, SSP, Play2	

# Usando IntelliJ Ultimate Edition



Usando la versión ultimate, cuando se crea un nuevo proyecto, ya nos da la opción de utilizar spring initializer desde origen para la creación de proyectos.

De este modo, nos ahorramos ese pequeño paso sin utilizar la página de Spring Initializer.

Y además de todo, el proyecto queda listo para usarse sin mas ni mas...!

# Usando IntelliJ Community Edition

En este caso necesitaremos crear el proyecto desde spring initializer:

The screenshot shows the Spring Initializr web interface with the following configuration:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 3.2.2 (selected)
- Dependencies:** No dependency selected.
- Project Metadata:**
  - Group: edu.unam
  - Artifact: hello-world-spring-security
  - Name: hello-world-spring-security
  - Description: Demo project for Spring Boot
  - Package name: edu.unam.hello-world-spring-security
- Packaging:** Jar (selected)
- Java:** 21 (selected)

At the bottom, there are three buttons: GENERATE (⌘ + ↩), EXPLORE (CTRL + SPACE), and SHARE... .

# Usando IntelliJ Community Edition

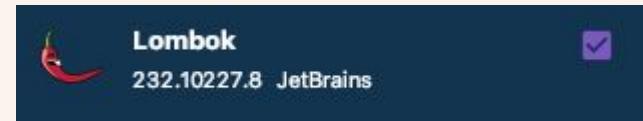
Una vez creado y descargado, procedemos a abrirlo en IntelliJ Community Edition.

Hay que tomar en cuenta que hay que descargar un plugin para apoyarnos con spring boot.

Una vez instalado, ya podemos usar spring boot en IntelliJ Community.



De igual manera, aprovechamos para instalar el plugin de **Lombok**, que utilizaremos para usar menos código espagueti, entre otras cosas.



# Creación de Hello World!

Para empezar, debemos de utilizar dos dependencias importantes:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

# Creación de Hello World!

De la misma manera, necesitamos configurar nuestra aplicación

Empezamos por configurar un puerto en **application.properties**

```
server.port = 8090
```

# Creación de Hello World!

Y seguimos con crear un controlador

En este caso será REST, solo para probar el hola mundo!

```
1 @RestController
2 @RequestMapping("/auth")
3 public class HelloWorldController {
4     @GetMapping("/welcome")
5     public String welcome() {
6         return "Hello World Spring Security!";
7     }
8 }
```

## 2. Configuración Spring Security

2.1 Usuario y contraseña por defecto (en memoria)



# Usuario y contraseña por defecto (en memoria)

Una vez realizado los pasos anteriores, ejecutar el proyecto.

Si somos observadores, notamos que en la consola de nuestro IntelliJ al ejecutarlo, nos manda el siguiente output.

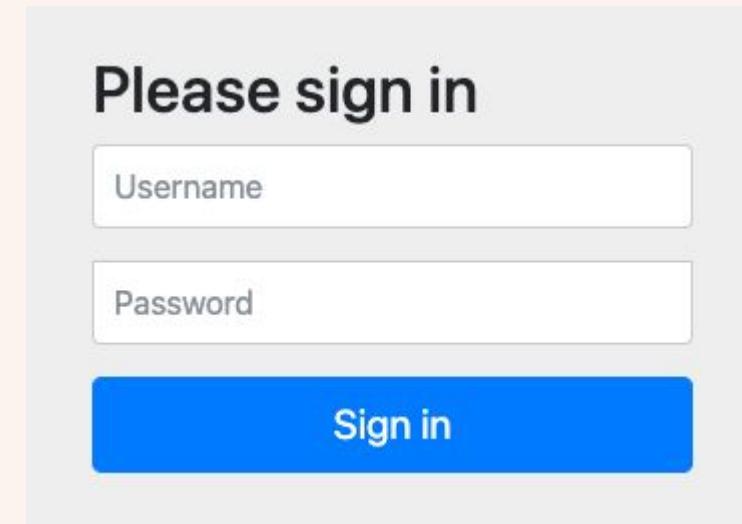
# Usuario y contraseña por defecto (en memoria)

Ahora, procedemos a ir a la siguiente url en nuestro navegador:

- <http://localhost:8090/auth/welcome>

Igual, notamos que no nos deja acceder, y nos redirige en automático a la siguiente url:

- <http://localhost:8090/login>



# Usuario y contraseña por defecto (en memoria)

En este caso, el username por defecto es **user**

Y en el caso de la contraseña, es la que nos generó en automático en la consola.

Nota: Cada vez que ejecutamos el proyecto, nos genera una contraseña nueva en automático.

Al entrar los datos, nos da acceso y ahora sí podemos ver el resultado del endpoint antes creado.

Hello World Spring Security!

# Descripción general de la autenticación básica HTTP

La autenticación básica HTTP es parte de la especificación HTTP

- Originalmente de RFC 2617, 1999, actualizado en RFC 7617 en 2015

La autenticación básica proporciona una forma estándar para que los clientes HTTP envíen el nombre de usuario y contraseña

- Codificación de URL: `https://nombre de usuario:contraseña@www.ejemplo.com`
- Encabezado HTTP: Clave: Autorización, Valor: Básico <cadena codificada en Base64>

Cadena: **nombre\_de\_usuario:contraseña**

# Descripción general de la autenticación básica HTTP

La codificación de URL y la codificación de encabezados no son seguras

- Tarea trivial para revertir una cadena codificada en Base64 a un valor de texto

Para proteger las credenciales de usuario, se recomienda el uso de HTTPS

La autenticación básica HTTP también es criticada por enviar credenciales de usuario en cada solicitud.

- Aumenta el riesgo
- Otros métodos envían un token de autenticación en cada solicitud.

## 2. Configuración Spring Security

2.2 Personalización de usuario y contraseña



file: [spring-security-disable-authentication.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# Cambio de valores por defecto a valores personalizados

Ahora, ¿cómo podemos cambiar los valores de login por defecto?:

Fácil!

Vamos a **application.properties**

```
spring.security.user.name = jonathan
spring.security.user.password = 1234
```

# Cambio de valores por defecto a valores personalizados

Ahora si, hemos cambiado el user y password por defecto.

Con este cambio, ya no aparece la contraseña en el log de consola de spring boot.

Esto apenas es el comienzo, apenas analizaremos el concepto de autenticación.



# Clase SecurityConfiguration

Necesitamos crear una clase de configuración de Spring Security. Esta clase puede tener diferentes nombres. Para este ejemplo la llamaremos

**SecurityConfiguration** con el siguiente código.

```
1 @Configuration // anotación de clase de config de spring security
2 @EnableWebSecurity // habilitamos seguridad web
3 public class SecurityConfiguration {
4     @Bean // método encadenamiento de sentencias de seguridad sobre HttpSecurity
5     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
6         http
7             .authorizeHttpRequests((authz) -> authz // lambda requerida!
8                 // permite todos los endpoints en cierta ruta dada
9                 .requestMatchers("/auth/**").permitAll()
10                .requestMatchers("/**"). permitAll()
11                // si no esta autenticado, no hay acceso a ningún endpoint
12                .anyRequest().authenticated()
13            );
14            // builder pattern
15        return http.build();
16    }
17 }
```

# Clase SecurityConfiguration

Notemos que el `SecurityFilterChain` es una implementación de un patrón de diseño llamado `ChainOfResponsibility`. Recordamos que es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Usamos la clase de `HttpSecurity` que permite configurar la seguridad basada en web para solicitudes http específicas.

Como dato informativo, para las versiones de Spring Boot 2 usaba Spring Security 5, para Spring Boot 3 se usa Spring Security 6.

# Clase SecurityConfiguration

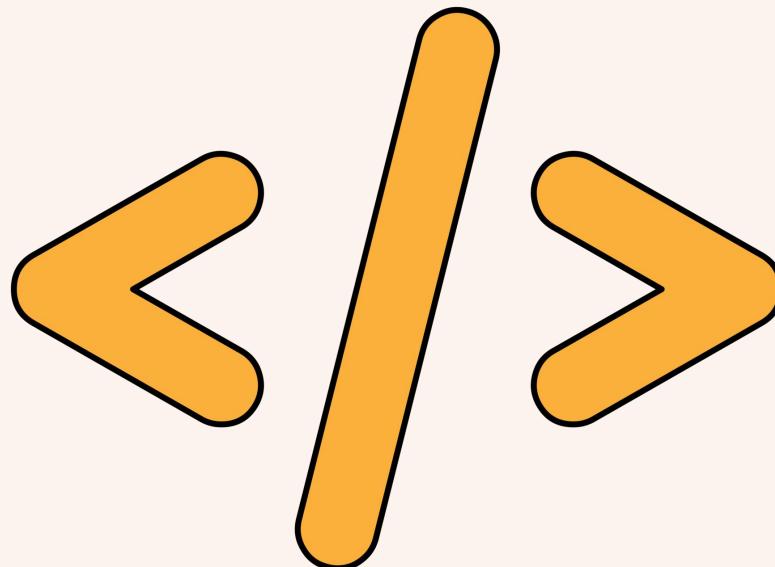
En Spring Security 6 trabajamos ya con lambdas y se cambia completamente la implementación de la seguridad en FilterChain.

- .requestMatchers("/auth/\*\*").permitAll()
- .anyRequest().authenticated()

Estas líneas nos permiten permitir cualquier tipo de petición hacia esa ruta de endpoints.

De igual manera podemos encadenar diferentes peticiones o mismos métodos como el de requestMatchers para ir agregando diferente lógica de permisos en diferentes endpoints o rutas de endpoints.

# Ejercicio 1



## 2. Configuración Spring Security

2.3 Renderizar atributos de la sesión con Thymeleaf



file: [spring-security-thymeleaf-integration.zip](#)  
file: [spring-security-thymeleaf-integration2.zip](#)

# Configuración de Spring

Continuando con el desarrollo, ahora, creamos un nuevo proyecto en Spring y configuramos

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
</dependencies>
```

# Configuración de Spring

Creamos ahora la capa de servicios y agregamos 3 nuevas clases:

1. **AdminService**
2. **HomeService**
3. **UserService**

Las creamos solo como ejemplo para ir repasando las capas de servicio hacia el controlador. Hay que anotarlas con **@Service**

Recordemos que seguimos desarrollando MVC en el proyecto.

```
1 @Service
2 public class AdminService {
3     public String getText() {
4         return "Admin";
5     }
6 }
```

```
1 @Service
2 public class HomeService {
3     public String getText() {
4         return "Home";
5     }
6 }
```

```
1 @Service
2 public class UserService {
3     public String getText() {
4         return "User";
5     }
6 }
```

# Configuración de Spring

Luego creamos un controlador llamado HomeController que es el que se va a conectar con la capa de servicios y ahí inyectamos en constructor.

Notamos que esta clase la anotaremos con **@Controller**.

Aquí hay que revisar varias cosas conceptuales.

```
1  @Controller
2  public class HomeController {
3      private final HomeService homeService;
4      private final UserService userService;
5      private final AdminService adminService;
6      private final UserInfoService userInfoService;
7
8      @Autowired
9      public HomeController(HomeService homeService, UserService userService,
10                          AdminService adminService, UserInfoService userInfoService) {
11          // Inyección por controlador
12      }
13
14     @GetMapping("/")
15     public String home(Model model) {
16         model.addAttribute("text", homeService.getText());
17         return "index";
18     }
19
20     @GetMapping("/index")
21     public String index() {
22         return "redirect:/";
23     }
24
25     @GetMapping("/user")
26     public String user(Model model) {
27         model.addAttribute("text", userService.getText());
28         return "user";
29     }
30
31     @GetMapping("/admin")
32     public String admin(Model model) {
33         model.addAttribute("text", adminService.getText());
34         return "admin";
35     }
36 }
```

# Configuración de Spring

La anotación **@Controller** forma parte del marco Spring MVC.

Normalmente se utiliza para crear aplicaciones web que representan vistas o páginas HTML.

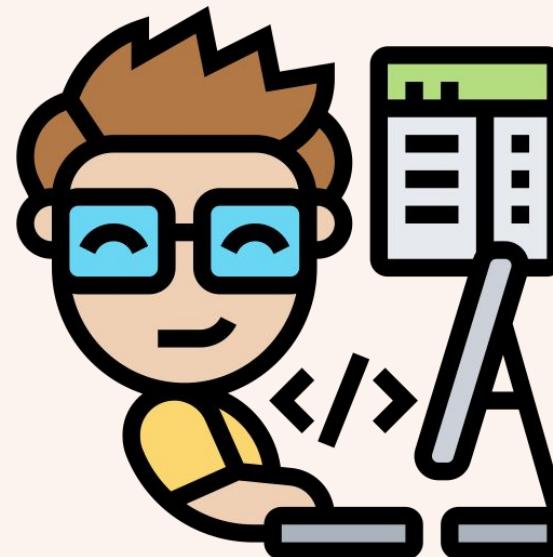
Los controladores anotados con **@Controller** son responsables de manejar las solicitudes HTTP, procesar datos y devolver una vista para que la represente el navegador del cliente.

# Configuración de Spring

Los métodos dentro de una clase `@Controller` a menudo devuelven un nombre de vista lógica, que Spring Boot resuelve en una plantilla HTML.

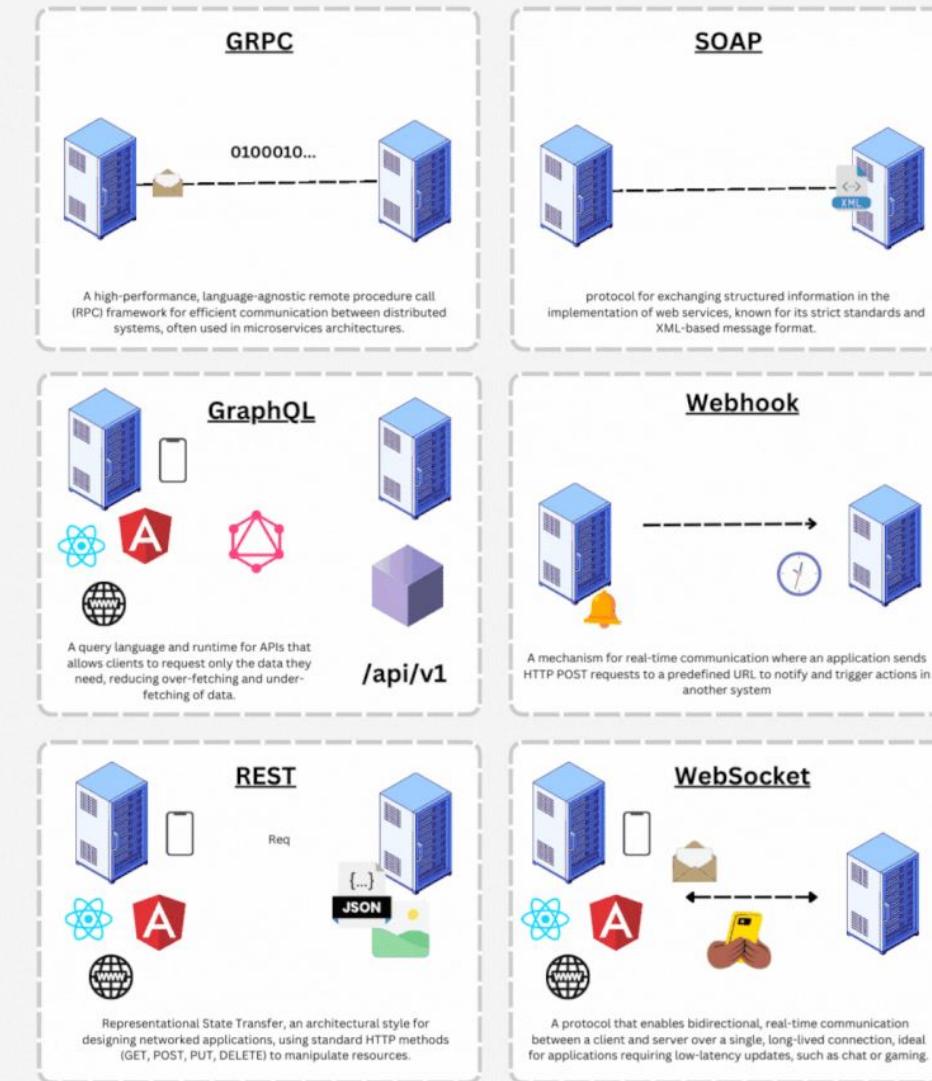
Aquí estaremos trabajando con MVC nativo de Spring, paso directo entre Controlador (java) y Vista (html). Esto es diferente a trabajar con **Arquitecturas API**.

# Configuración de Spring



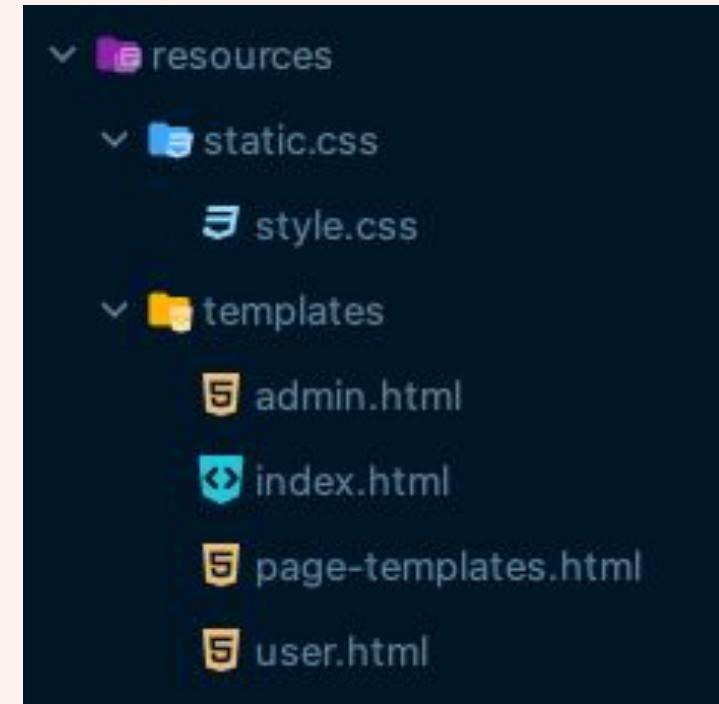
amigoscode.com

## 6 Popular API Architecture Styles



# Configuración de Spring

- Finalmente, agregamos las hojas HTML con Thymeleaf en nuestro proyecto.
- Creamos carpetas y archivos nuevos en la carpeta resources.
- Recordemos que la carpeta static por lo general, es para archivos con poco cambio o casi nulo cambio. Archivos por lo general CSS o incluso templates de HTML estáticos. Si usamos bootstrap, aquí irían las librerías o el CDN local. Lo mismo con archivos locales JS, imágenes, etc.
- En el caso de la carpeta templates, aquí iría nuestro código en thymeleaf listo para ser modificado o trabajado en nuestros proyectos.



# Configuración de Spring

Finalmente, tenemos 4 archivos HTML

- admin.html
- index.html
- page-templates.html
- user.html

**page-templates**: contiene fragmentos thymeleaf del navbar y footer a ser usados en los demás archivos HTML

# Configuración de Spring

**user.html**: llamado a navbar, footer, y texto proveniente del controlador HomeController, apuntando al método GET endpoint /user

**admin.html**: llamado a navbar, footer, y texto proveniente del controlador HomeController, apuntando al método GET endpoint /admin

**index.html**: llamado a navbar, footer, y texto proveniente del controlador HomeController, apuntando al método GET endpoint /index

# Configuración de Spring

Al ejecutar nuestro proyecto nos da como resultado, la página inicial.

\*\*Revisar y jugar con **SecurityConfiguration** para ver si da acceso a las URL o no.

The screenshot shows a web browser window with the following details:

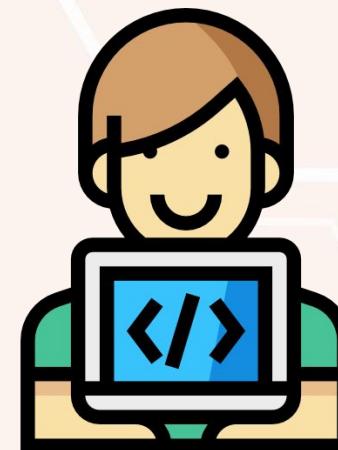
- Address Bar:** localhost:8090
- Toolbar:** Includes standard browser icons like back, forward, search, and refresh, along with a "New Chrome available" notification.
- Bookmark Bar:** Contains links to myapps.cbre.com, Swagger UI, docker-compose..., feign/DefaultError..., DOM, Solicitud de Reinte..., Solicitud de carga..., Certificación - Pro..., Contribuciones -..., Tipo de contribuci..., Aplicaciones Exter..., and All Bookmarks.
- Header Bar:** Shows the title "Diplomado Java - UNAM" in red, and navigation links for Home, User, and Admin.
- Content Area:** Displays the "Home" page of the application, which is currently empty.
- Page Footer:** © Alfonso Rivero - Spring Security

# Mejora del programa

Vamos a realizar algunas modificaciones o mejoras a nuestro programa

Archivos que modificaremos:

- pom.xml
- style.css
- SecurityConfiguration.java
- admin.html
- index.html
- page-template.html
- user.html



# Mejora del programa

Debemos agregar 1 dependencia más al POM.

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

Y agregar unas líneas al final del archivo CSS.

```
input[type=submit].link {
    border: none;
    background: none;
    color: black;
    text-decoration: none;
}

input[type=submit].link hover{
    color: darkgreen;
}

form.link {
    align-items: center;
    display: inline-block;
    padding: 10px 15px;
}
```

# Mejora del programa

Ahora, vamos a configurar de una manera diferente nuestro **SecurityConfiguration**.

Entramos a la clase y modificamos primero el bean de securityFilterChain.

Agregamos correctamente el requestMatchers para cerciorarse de permitir nuestro home, index, css y favicon.

De igual manera asignamos ROLES a ciertos endpoints:

- .requestMatchers("/user").hasAnyRole("USER")
- .requestMatchers("/admin").hasAnyRole("ADMIN")

# Mejora del programa

Luego vemos la linea `.formLogin(Customizer.withDefaults())`

De forma predeterminada, el inicio de sesión del formulario Spring Security está habilitado. Sin embargo, tan pronto como se proporcione cualquier configuración basada en servlet, se debe proporcionar explícitamente el inicio de sesión basado en formulario.

<https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/form.html>

# Mejora del programa

Ya que nos encontramos en esta clase, hay que hacer énfasis en el `formLogin`

```
.formLogin(login -> login
            .defaultSuccessUrl("/")
            .permitAll())
.logout(logout -> logout
        .logoutSuccessUrl("/"));
```

Lo que trabajamos en estos apartados, es definir un redirect en URL para login **SUCCESS** y **LOGOUT**. Ambos apuntan hacia “/”

# Mejora del programa

En el caso de los otros dos beans que agregamos, uno es el **PasswordEncoder** que utilizaremos en nuestra aplicación.

Si notamos aquí, este bean tiene un par de líneas comentadas. Es el parteaguas para ver el archivo **PasswordEncoder.pdf** donde se explican los diferentes PasswordEncoders que se pueden utilizar en las aplicaciones.

- En este caso usamos `PasswordEncoderFactories.createDelegatingPasswordEncoder()` al tener una dependencia directa con el bean faltante.

# Mejora del programa

El otro bean es **UserDetailsManager**, que utilizaremos para crear 2 usuarios “en memoria” (puesto que aún no hemos agregado alguna BD para persistir esta información).

Dichos usuarios se crean sin **PasswordEncoder** (**{noop}**) y es por eso que se define en el bean anterior la delegación directa del **PasswordEncoder**.

Finalmente los últimos archivos a modificar son los HTML.

El cambio principal está en el archivo `page-templates.html`

# Mejora del programa

Dentro del navbar, vamos a modificar los hrefs hacia nuestro controlador.

- Recordemos que ahora ya metimos AUTORIZACIÓN por roles en el proyecto. Por lo tanto, hay que ir a modificar el Front para aceptar dichos cambios.

Y de paso, agregamos el href de logout que apunta hacia un endpoint “/logout”.

Dicho endpoint no existe. Sin embargo, en los cambios realizados en el **SecurityConfiguration**, ahí detallamos un redirect a “/” al realizar un logout.

Los demás archivos HTML tienen cambios básicos sencillos.

- Solo imprimimos los roles del usuario (authorities) que se encuentra autenticado en ese momento.

## 2. Configuración Spring Security

2.4 Creación de formulario de login personalizado



file: [spring-security-thymeleaf-integration.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# Archivos a modificar

Vamos a modificar y agregar cierto código a los archivos:

- HomeController.java
- SecurityConfiguration.java

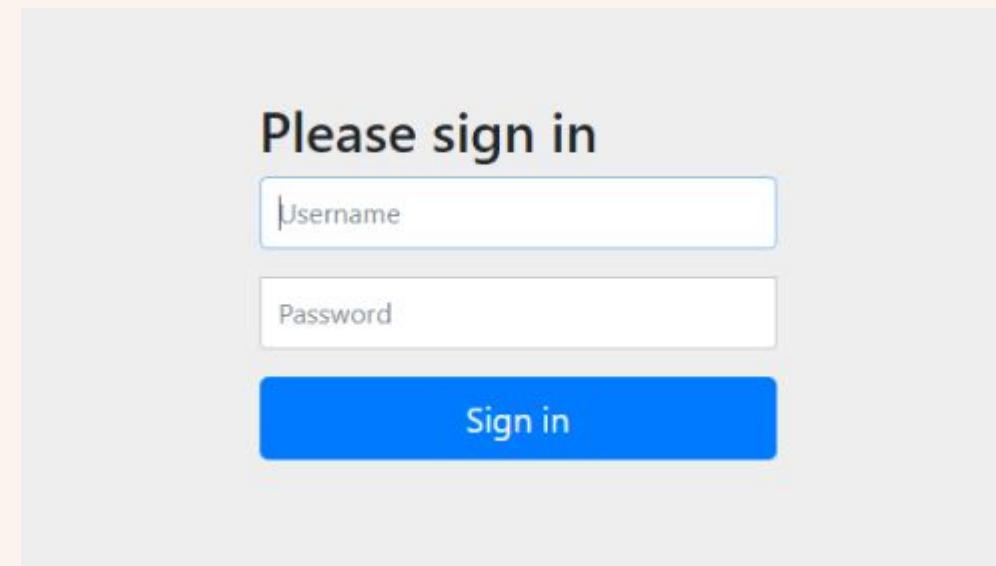
Y creamos un nuevo archivo

- login.html



# Creación de formulario de login personalizado

Cuando se utiliza el método de inicio de sesión mediante formulario, Spring Security mostrará la página de inicio de sesión predeterminada que se ve así:



Please sign in

# Creación de formulario de login personalizado

En caso de que desee utilizar su propia página de inicio de sesión, especifique la URL de la página de inicio de sesión personalizada utilizando este código en el **SecurityFilterChain**:

```
.formLogin(login -> login  
    .loginPage("/login"))
```

# Creación de formulario de login personalizado

De forma predeterminada, Spring Security utiliza los nombres de campo nombre de usuario y contraseña, y la acción del formulario es /iniciar sesión. Si desea utilizar diferentes nombres de campo y URL, especifíquelos usando el código Java como se muestra a continuación

```
http.formLogin()  
    .loginPage("/login")  
    .usernameParameter("email")  
    .passwordParameter("pass")  
    .loginProcessingUrl("/doLogin");
```

# Login Default Success URL

Spring Security redirigirá a los usuarios a la página que visitaron antes de iniciar sesión. Por ejemplo, si un usuario visita la página para crear un nuevo producto que requiere autenticación, será redirigido a la página de inicio de sesión. Y después de iniciar sesión correctamente, será enviado de regreso a la página de creación de nuevo producto. Esta es la lógica correcta en caso de que el usuario haya visitado antes una página segura.

En caso de que desee mostrarle al usuario una página separada después de que visite una página no segura antes de iniciar sesión, especifique la URL de éxito predeterminada de la siguiente manera:

```
http.formLogin()  
    .defaultSuccessUrl("/login_success");
```

# Login Failure URL

De forma predeterminada, Spring Security redireccionará a /login?error si el usuario no pudo iniciar sesión. Si desea cambiar este comportamiento, p. mostrando su propia página que muestra un mensaje de error de inicio de sesión al usuario; luego especifique la página de error de inicio de sesión personalizada usando este código:

```
http.formLogin()  
    .failureUrl("/login_error");
```

# Login Success Forward URL

Si desea ejecutar algún código adicional después de que el usuario haya iniciado sesión correctamente, por ejemplo, para uso de auditoría o logging, luego hay que especificar la URL de reenvío exitosa como esta:

```
http.formLogin()  
    .successForwardUrl("/login_success_handler");
```

# Login Success Forward URL

Para que esto funcione correctamente, debes escribir un método de controlador correspondiente en la clase de controlador:

```
@PostMapping("/login_success_handler")
public String loginSuccessHandler() {
    System.out.println("Logging user login success...");
    return "index";
}
```

# Login Failure Forward URL

Si desea ejecutar algún código adicional si el usuario no pudo iniciar sesión, especifique la URL de reenvío de falla como esta

```
http.formLogin()  
    .failureForwardUrl("/login_failure_handler")
```

# Login Failure Forward URL

Y codifique el método del controlador en el controlador de esta manera:

```
@PostMapping("/login_failure_handler")
public String loginFailureHandler() {
    System.out.println("Login failure handler....");
    return "login";
}
```

En este método de manejo de errores de inicio de sesión, usted decide qué página debe ver el usuario después.

# Login Success Handler

De manera similar a la URL de reenvío de éxito de inicio de sesión, también puede escribir un controlador de éxito de autenticación de la siguiente manera:

```
http.formLogin()
    .successHandler(new AuthenticationSuccessHandler() {
        @Override
        public void onAuthenticationSuccess(HttpServletRequest request,
                                            HttpServletResponse response, Authentication authentication) throws
        IOException, ServletException {
            System.out.println("Logged user: " + authentication.getName());
            response.sendRedirect("/");
        }
    });
}
```

# Login Success Handler

Esto le brinda un mayor control ya que puede acceder a un objeto de autenticación directamente.

# Login Failure Handler

En lugar de utilizar la URL de reenvío de errores de inicio de sesión, puede utilizar un controlador de errores de autenticación de la siguiente manera:

```
http.formLogin()
    .failureHandler(new AuthenticationFailureHandler() {
        @Override
        public void onAuthenticationFailure(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException exception) throws
        IOException, ServletException {
            System.out.println("Login failed");
            System.out.println(exception);
            response.sendRedirect("/login_error");
        }
    });
}
```

# Login Failure Handler

Esto le brinda un mayor control ya que puede acceder a un objeto **AuthenticationException** directamente.

## 2. Configuración Spring Security



2.5 Cerrar sesión de usuario

file: [spring-security-personalized-logout.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# Archivos a modificar

Vamos a modificar y agregar cierto código a los archivos:

- SecurityConfiguration.java
- page-template.html



# Customize Logout Details

Además de la personalización del inicio de sesión, Spring Security también permite a los programadores personalizar el proceso de cierre de sesión. Básicamente codificas el botón Cerrar sesión de esta manera:

```
<form th:action="@{/logout}" method="post">
    <input type="submit" value="Logout" />
</form>
```

Lo primero que se puede cambiar es la URL de cierre de sesión.

# Logout URL

De forma predeterminada, Spring Security procesa la URL /logout mediante el método HTTP POST. Puede configurar el objeto **HttpSecurity** para cambiar esta URL como se muestra a continuación:

```
http.logout()  
    .logoutUrl("/doLogout");
```

Para que esto funcione, debe actualizar la URL de acción del formulario de cierre de sesión en consecuencia:

```
<form th:action="@{/doLogout}" method="post">
```

# Logout Success URL

De forma predeterminada, el usuario verá la página de inicio de sesión después de cerrar sesión en la aplicación. Si desea mostrar una página personalizada al usuario, especifique la URL de cierre de sesión exitoso de la siguiente manera:

```
http.logout()  
    .logoutSuccessUrl("/logout_success");
```

# Logout Success Handler

Si desea realizar pasos adicionales después de que el usuario haya cerrado sesión correctamente, utilice un controlador de cierre de sesión exitoso como este:

```
http.logout()  
    .logoutSuccessHandler(new LogoutSuccessHandler() {  
        @Override  
            public void onLogoutSuccess(HttpServletRequest request,  
HttpServletResponse response, Authentication authentication) throws  
IOException, ServletException {  
                System.out.println("This user logged out: " +  
authentication.getName());  
                response.sendRedirect("/logout_success");  
            }  
    }) ;
```

# Usar Link de Logout en vez de botón

La razón por la que debería usar el botón Cerrar sesión es porque Spring Security genera automáticamente un token de seguridad para evitar el ataque CSRF (falsificación de solicitud entre sitios) en la página de inicio de sesión. Por ejemplo:

```
<input type="hidden" name="_csrf" value="8ec12704-5ab6-4f0c-a758-2fc36f2c9368"/>
```

Es por eso que la solicitud de cierre de sesión debe enviarse mediante el método HTTP POST. Puede deshabilitar la prevención CSRF para usar un enlace de cierre de sesión (a través del método HTTP GET) usando este código:

```
.csrf(csrf -> csrf.disable())
```

ver: [csrf.pdf](#)

# Usar Link de Logout en vez de botón

Sin embargo, esto no se recomienda para aplicaciones que están en producción. Entonces, la mejor manera es ocultar el formulario de cierre de sesión y usar Javascript para el enlace de cierre de sesión, como se muestra en el siguiente ejemplo:

```
<form      name="logoutForm"          th:hidden="true"          method="post"
th:action="@{/doLogout}">
    <input type="submit" value="Logout" />
</form>
<a href="javascript: logoutForm.submit();">Sign Out</a>
```

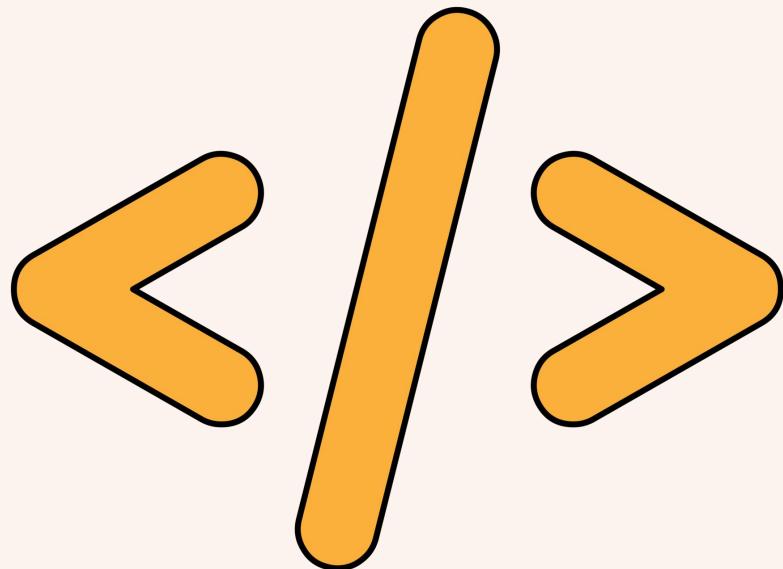
# Usar Link de Logout en vez de botón

Así es como se personalizan los detalles de inicio y cierre de sesión para una aplicación web Java creada en Spring Boot.

Spring Security proporciona API altamente flexibles y personalizables para que pueda usarlas con facilidad.



# Ejercicio 2



# 3. Recuperación de usuarios y roles desde una base de datos

3.1 Creación de las clases de usuario y rol



file: [spring-security-user-role-db-creation.zip](#)  
DDTIC\_DSJ\_PLI\_2024

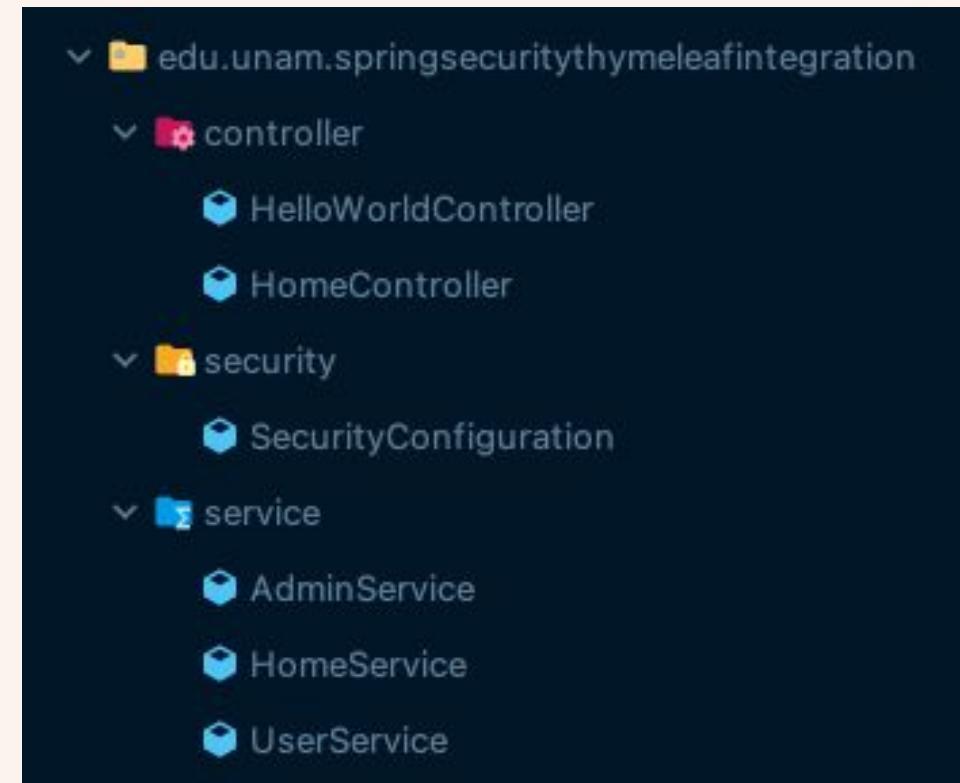
# Creación de las Clases de Usuario y Rol

En este apartado vamos a trabajar con la creación de las clases de usuario y rol.

Antes de crearlas, tenemos que trabajar con un encarpelado fácil de entender.

Anteriormente tenemos este encarpelado, que nos funcionó perfectamente para empezar a trabajar con Spring Security.

Sin embargo, ahora, tenemos que realizar ciertos cambios para darle una mejor forma al proyecto.



# Creación de las Clases de Usuario y Rol

Empezamos por crear una carpeta llamada “system” en la ruta original, donde ahí vamos a arrastrar los otros dos paquetes antes creados, llamados “controller” y “service”.

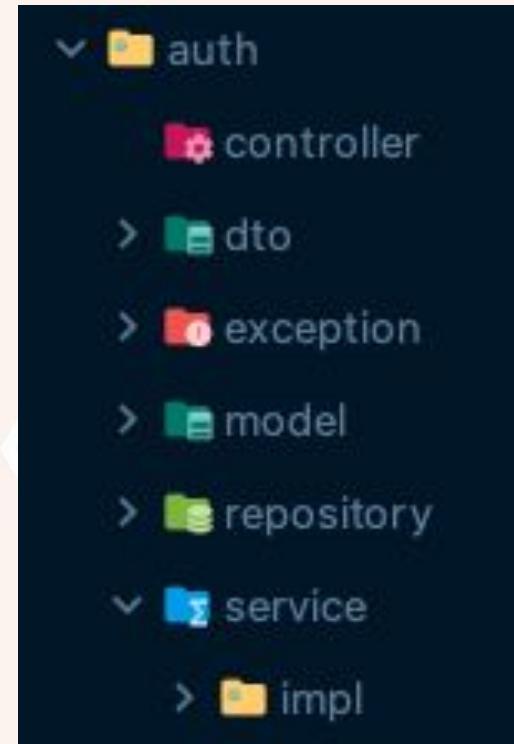
- El archivo HomeController lo vamos a pasar dentro de “auth” / “controller”

Luego, vamos a crear otra carpeta llamada “auth”.

# Creación de las Clases de Usuario y Rol

En general, dentro de la carpeta “auth” vamos a tener la capa de creación de clases orientadas hacia el módulo de seguridad de la aplicación (user / role / user\_role).

Tomando en cuenta nuestro modelo de encarpetado / arquitectura por capas, dentro de “auth” vamos a crear los paquetes base de la autenticación:



# Creación de las Clases de Usuario y Rol

Dando una explicación general:

- **dto**: aquí van nuestras clases DTO. Recordar que no es bueno exponer nuestro modelo hasta la capa de servicio y controlador.
- **exception**: requerido para manejar nuestras propias excepciones
- **model**: modelo de nuestras clases de seguridad
- **repository**: uso de jpa para trabajar con nuestros CRUD
- **service**: para poder trabajar con nuestro CRUD dentro del sistema. Aquí se trabaja principio SOLID en cuanto a la Interface Segregation y Dependency Inversion.

De igual manera, crearemos 2 test cases apuntando directamente al repository

- Dichos test cases solamente nos ayudarán a crear algo de información y escribirla directo en la BD.

# POM

Primero, debemos agregar las librerías de JPA, MariaDB y Lombok, para que desde un principio evitemos la creación de código espagueti.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

# Properties

Ahora, debemos de reemplazar el archivo properties por ese pedazo de configuraciones.

Ya viene con comentarios para su mejor entendimiento.

Básicamente agregamos la configuración de la BD

```
# =====
# = GLOBAL CONFIG
# =====
server.port=8090
spring.security.user.name=jonathan
spring.security.user.password=1234

# =====
# = DATA SOURCE
# =====
# Set here configurations for the database connection
spring.datasource.url=jdbc:mariadb://localhost:3306/springsecurity
spring.datasource.username=root
spring.datasource.password=d1p10m4d0j4v4
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
# Keep the connection alive if idle for a long time (needed in production)
# spring.datasource.testWhileIdle=true
# spring.datasource.validationQuery=SELECT 1

# =====
# = JPA / HIBERNATE
# =====
# Show or not log for each sql query
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update): with "create-drop" the database
# schema will be automatically created afresh for every start of application
# spring.jpa.hibernate.ddl-auto=create
spring.jpa.hibernate.ddl-auto=none
# Allows Hibernate to generate SQL optimized for a particular DBMS
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect
# Allows Hibernate to format SQL in console
spring.jpa.properties.hibernate.format_sql=true
```

# Model

En este apartado vamos a crear 2 clases, UserInfo y UserInfoRole.

Userinfo es la información de la entidad donde almacenaremos la información de los usuarios en la BD. Esta clase tendrá información muy general sobre el usuario.

Aquí, notamos un pedazo de código que relaciona con @ManyToMany hacia UserInfoRole.

- Esto significa que vamos a crear una tabla intermedia para poder relacionar los usuarios con sus roles.

# Model

```
1 @ManyToMany(fetch = FetchType.EAGER)
2 @JoinTable(schema = "user_adm",
3             name = "sec_user_role_relation",
4             joinColumns = @JoinColumn(name = "urr_id_user"),
5             inverseJoinColumns = @JoinColumn(name = "urr_id_user_role")
6 )
7 @JsonManagedReference
8 private Set<UserInfoRole> useInfoRoles = new HashSet<>();
```

Justo aquí, este es el apartado donde cada uno puede modificar esto a su conveniencia. Podrían utilizar relaciones @OneToOne o @OneToMany. En este caso, se expone la alternativa más compleja para que cada quien adecúe a sus necesidades.

# Model

Finalizamos la clase con un método que nos retorna el nombre completo del usuario.

Para el caso de `UserInfoRole`, solo agregaremos la información de la entidad para almacenar los roles.

Notamos que estas clases tienen ciertas particularidades de mejores prácticas en general:

- El nombre de la tabla va en minúsculas, separado por `_` y con prefijo
- El prefijo de cada campo va igual a 3 letras
- Se agregan 4 campos de auditoría al final de cada entidad

# DTO

En este paquete estaremos trabajando clases DTO.

Por el momento trabajaremos en 2 clases:

- UserInfoDTO
- UserInfoRoleDTO

Notaremos que ambas clases son casi iguales al Model, sólo que éstas clases no son entidades y están separadas para crear objetos independientes.

NOTA: cuando trabajamos con una capa de DTO, hay que trabajar TODO con DTO (revisar Set en UserInfoDTO).

# Repository

Siguiente, creamos la capa de repositorio para ambas clases.

En estos dos repositorios, creamos algunos Query Methods (buena alternativa para usar este poderoso feature de Spring y evitar realizar las querys en duro).

Anotamos la interfaz con **@Repository**.

- Esto ya no es necesario, sin embargo, lo realizo por costumbre.

# Service

Este apartado es muy importante. Como ya sabemos, aquí mandamos a llamar al repository para trabajar el CRUD. En este caso, no realizaremos ningún delete.

En estos servicios ya estamos utilizando principio SOLID, creando interfaces y su implementación.

- Dentro de la implementación estaremos trabajando métodos para el CRUD.

Como parte importante, estamos realizando dos métodos “mapper” en cada clase.

- Este método “mapper” nos va a servir para convertir entre DTO y Entity.

# Service

De igual manera estamos utilizando query Methods.

Sirve que repasamos un poco lo visto en módulos anteriores

```
1 @Override
2 public UserInfoDTO findByUseEmail(String email) throws UserInfoNotFoundException {
3         // se crea en automático desde el repositorio!
4     UserInfo object = userInfoRepository.findByUseEmail(email);
5     if(object == null)
6         throw new UserInfoNotFoundException(email);
7     return convertEntityToDTO(object);
8 }
```

# Unit Testing

De la misma manera, vamos a crear 2 pruebas unitarias a partir de los repositorios.

Dichas clases van a tener test cases que van a invocar al crud de nuestra aplicación para crear registros y probar que se han creado dichos registros.

```
1 @DataJpaTest // Anotación JPA Test
2 @AutoConfigureTestDatabase(replace = Replace.NONE)
3 @Rollback(false) // Es decir, SI modifica la BD
4 class UserInfoRepositoryTest {
5 }
```

```
1 @Test
2 public void testCreateUser() {
3     // creación del usuario
4     UserInfo user = new UserInfo();
5     user.setUseFirstName("Alfonso");
6     user.setUseLastName("Rivero");
7     user.setUseEmail("devil861109@gmail.com");
8     user.setUsePasswd("asdf1234"); //user
9     user.setUseIdStatus(1);
10    user.setUseCreatedBy(1L);
11    user.setUseModifiedBy(1L);
12    UserInfo savedUser = userInfoRepository.save(user);
13    // lo busca en la BD
14    UserInfo existUser = testEntityManager.find(UserInfo.class, savedUser.getUseId());
15    // assertion
16    assertThat(user.getUseEmail()).isEqualTo(existUser.getUseEmail());
17 }
```

### 3. Recuperación de usuarios y roles desde una base de datos

3.2 Conectar la autenticación con la BD



file: [spring-security-db-configuration.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# Coneectar la autenticación con la BD

Por fin empezaremos a ver temas más a profundidad. Hay que conectar nuestro modelo de seguridad de la BD hacia Spring Security.

Lo primero que vamos a configurar son los test unitarios.

En la clase **UserInfoRepositoryTest**, vamos a crear nuevos test cases:

- para poder agregar un usuario nuevo
- dos test cases que nos van a servir para asignar roles a nuestros usuarios

Si seguimos bien el desarrollo, el rol USER es id 1 y el rol ADMIN es id 2

Una vez creadas, ejecutamos los test cases para actualizar en la BD el password y asignar roles a los usuarios en la tabla relacional.

# Package “security”

En el paquete de seguridad, vamos a crear nuestro encarpelado base:

- paquete model
- paquete service

En el paquete model, vamos a crear una clase:

- UserDetailsImpl

En el paquete service vamos a crear dos clases:

- AuthenticationProviderImpl
- UserDetailsServiceImpl

# Package “security”

Si notamos bien, la clase `UserDetailsImpl` es nuestra propia implementación de una interfaz llamada `UserDetails`, que forma parte del Core de Spring Security.

De igual manera, la clase `AuthenticationProviderImpl` implementa la interfaz `AuthenticationProvider` y la clase `UserDetailsServiceImpl` implementa `UserDetailsService`, todo forma parte de core de Spring Security

# Package “security”

En nuestro caso, `UserDetailsImpl` es una implementación creada para comunicarnos con Spring Security. Si la analizamos bien, lo que estamos haciendo es crear ese medio de conexión entre nuestro modelo `UserInfo` con la interfaz `UserDetails` de S.S.

Una diferencia notable, es que ya no estaremos hablando de “roles”, ahora hablaremos de “authorities”.

De igual manera vemos varios métodos que nos pide implementar por defecto, que dichos métodos vamos a poder comunicarnos con nuestra BD directamente, por si requerimos desarrollar ciertas lógicas de autenticación.

# Package “security”

En Spring Security, la interfaz **UserDetailsService** es un componente central que se utiliza para cargar datos específicos del usuario. Es responsable de recuperar información del usuario de una fuente de datos de backend, como una base de datos o un servicio externo, y devolver una instancia de la interfaz **UserDetails**.

La interfaz **UserDetailsService** tiene un método único llamado `loadUserByUsername()`, que toma un nombre de usuario como parámetro y devuelve un objeto **UserDetails** completamente poblado. El objeto **UserDetails** representa al usuario autenticado en el marco de Spring Security y contiene detalles como el nombre de usuario, la contraseña, las autoridades (roles) y atributos adicionales del usuario.

Desde aquí, notamos que nuestro username a utilizar será directamente el email del usuario!

# Package “security”

La interfaz **AuthenticationProvider** es un componente clave del marco de autenticación y autorización de Spring Security. Es responsable de autenticar las credenciales de un usuario y devolver un objeto de autenticación que representa al usuario autenticado.

# Package “security”

La interfaz **AuthenticationProvider** está diseñada para admitir una amplia gama de escenarios de autenticación. Por ejemplo, una implementación de esta interfaz puede realizar las siguientes acciones:

1. Verifique las credenciales del usuario con una base de datos o proveedores de identidad externos, como LDAP u OAuth.
2. Realice una validación adicional de las credenciales del usuario, como verificar la caducidad de la contraseña o el bloqueo de la cuenta.
3. Asigne roles y permisos de usuario según las credenciales del usuario.
4. Devuelve información adicional sobre el usuario autenticado, como su nombre completo o dirección de correo electrónico.

# Package “security”

Proveedores de autenticación más comunes en Spring Security:

- DaoAuthenticationProvider
- LdapAuthenticationProvider
- OpenIDAuthenticationProvider
- JwtAuthenticationProvider
- RememberMeAuthenticationProvider

# Package “security”

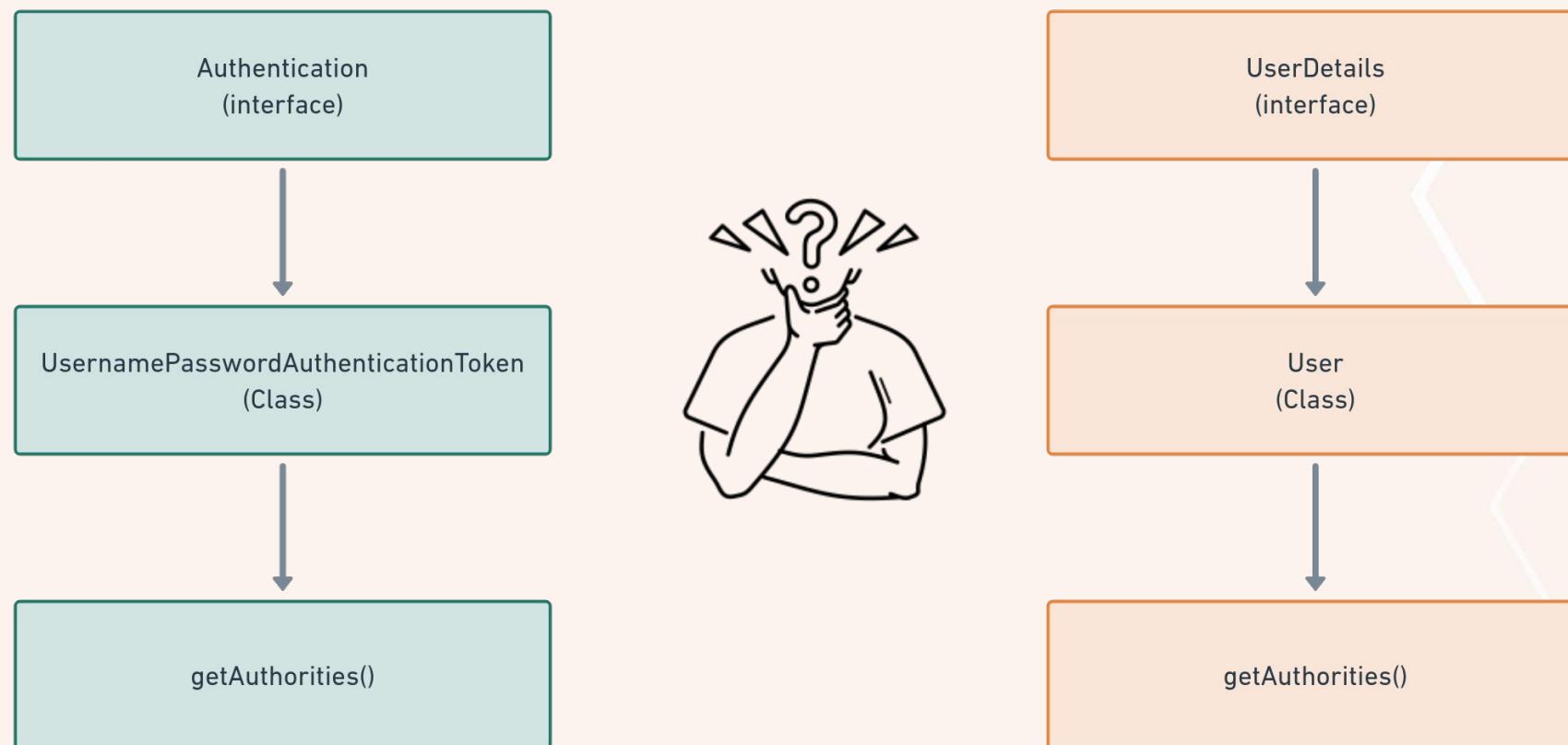
- Authorities/Roles information in Spring Security is stored inside GrantedAuthority. There is only one method inside GrantedAuthority which return the name of the authority or role.
- SimpleGrantedAuthority is the default implementation class of GrantedAuthority interface inside Spring Security framework.

```
public interface GrantedAuthority {  
  
    String getAuthority();  
  
}
```

```
public final class SimpleGrantedAuthority implements GrantedAuthority {  
  
    private final String role;  
  
    public SimpleGrantedAuthority(String role) {  
        this.role = role;  
    }  
  
    @Override  
    public String getAuthority() {  
        return this.role;  
    }  
}
```

# ¿Cómo se almacenan los authorities?

¿Cómo se almacena la información de las Autoridades dentro de los objetos de las interfaces UserDetails & Authentication que juega un papel vital durante la autenticación del usuario?



# ¿Cómo se almacenan los authorities?

## AUTHORITY

La autoridad es como un privilegio individual o una acción.  
Restringir el acceso de forma detallada  
Ej: VER CUENTA, VISTAS, etc.

## ROLE

ROLE es un grupo de privilegios/acciones Restringir el acceso de forma generalizada Ej:  
ROLE\_ADMIN, ROLE\_USER

- Los nombres de las autoridades/roles son de naturaleza arbitraria y estos nombres se pueden personalizar según los requisitos comerciales.
- Los roles también se representan utilizando el mismo contrato GrantedAuthority en Spring Security
- Al definir un rol, su nombre debe comenzar con el prefijo ROLE\_. Este prefijo especifica la diferencia entre un rol y una autoridad.

# Limpieza en SecurityConfiguration

Ahora, ya que hemos entendido un poco cómo funciona la clase de **SecurityConfiguration**, vamos a limpiarla.

- Primero, voy a dejarla como está, solo voy a borrar los comentarios y haré un refactor hacia los imports.

La primera modificación será en el método **PasswordEncoder**. Aquí vamos a trabajar únicamente con bcrypt, para realizarlo más fácil.

```
1 @Bean
2 PasswordEncoder passwordEncoder() {
3     return new BCryptPasswordEncoder(11, new SecureRandom());
4 }
```

# Limpieza en SecurityConfiguration

El siguiente cambio es más agresivo, vamos a borrar el método `inMemoryUserDetailsManager()`, puesto que ya no lo estaremos utilizando porque ahora vamos a conectarnos directamente a la BD para obtener la información.

Luego de eso, vamos a crear un nuevo método llamado `authenticationProvider()`. Como vimos anteriormente, este método va a trabajar con **DaoAuthenticationProvider**. Vamos a pasarle nuestra clase de **UserDetails** y finalizamos con el encoder.

# Limpieza en SecurityConfiguration

De igual manera hay que injectar la clase **UserDetailsService** para que funcione.

```
1 @Bean
2 public AuthenticationProvider authenticationProvider() {
3     DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
4     authenticationProvider.setUserDetailsService(uds);
5     authenticationProvider.setPasswordEncoder(passwordEncoder());
6     return authenticationProvider;
7 }
```

# Limpieza en SecurityConfiguration

Finalmente, realizamos últimos ajustes en el método `securityFilterChain`.

Primero, vamos a agregar la siguiente línea dentro del logout:

- `.deleteCookies("JSESSIONID")`
- esto nos va a permitir borrar las cookies que crea spring, y se borran en cada logout

# Limpieza en SecurityConfiguration

Segundo, vamos a crear 3 chain methods nuevos:

- `.csrf(Customizer.withDefaults())`: trabajar con csrf de manera default
- `.cors(Customizer.withDefaults())`: trabajar con cors de manera default
- `.authenticationProvider(authenticationProvider())`: trabajar con nuestro authentication provider previamente configurado (DAO)

Con esto es más que suficiente! Ahora ya podemos ejecutar el proyecto y podemos ingresar con nuestro user y password de la BD.

# 3. Recuperación de usuarios y roles desde una base de datos

3.3 Implementar la función de registro de usuarios



file: [spring-security-user-registry.zip](#)

# Registro de usuarios

En este apartado, vamos a trabajar creando una pantalla de registro de usuarios.

Hay que recalcar que para muchos proyectos, no es funcional la pantalla de registro.

En este caso se realiza por ser una manera educativa de entender mejor la parte de seguridad y poder enlazarla con la lógica de la aplicación

Para poder empezar, vamos a crear primero dos templates en html usando thymeleaf:

- `signup_form.html`
- `register_success.html`

# Registro de usuarios

**signup\_form**: form donde cargamos la información del nuevo usuario. Utilizaremos Bootstrap con alguna que otra validación en front end.

**register\_success**: mensaje de creación de usuario y nos permite regresar al login de la aplicación



# Auth

En la carpeta 'auth', únicamente crearemos dos métodos nuevos en el paquete 'controller'

- showRegistrationForm
- processRegister

De igual manera, vamos a crear una inyección hacia **UserInfoService**.

Recordemos que **UserInfoService** es la capa de servicio del usuario, donde podemos realizar el CRUD

# SecurityConfiguration

Ya hemos terminado! Ya podemos ejecutar el proyecto y revisar funcionalidad.

Sin embargo, aclaremos un último movimiento en **SecurityConfiguration**. Vamos a trabajar con un **AuthenticationManager**.

- Eso significa eliminar la linea

```
.authenticationProvider(authenticationProvider())
```

del método **securityFilterChain**

- Y agregar un método del **AuthenticationManager**

Con eso terminamos este apartado!

- Fijarse bien en cómo llamamos al modelo desde un form y cuando damos submit, cómo se procesa el evento.
- De igual manera, notar cómo trabajan los campos a nivel de thymeleaf.

# 4. Recuperación de usuarios y roles respaldados por LDAP

## 4.1 ¿Qué es LDAP?



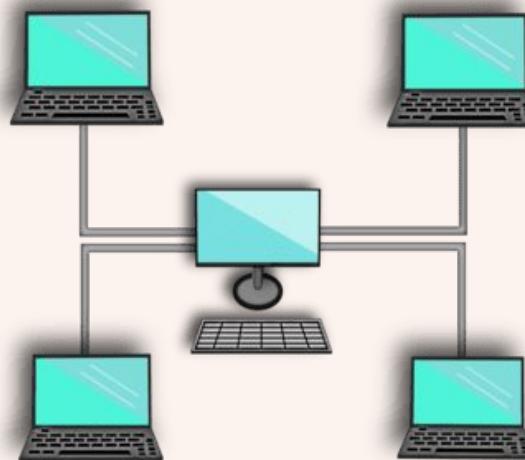
# ¿Qué es LDAP?

Supongamos que trabajamos en una empresa pequeña. Dicha empresa cuenta con un personal de 5 personas, de las cuales tú eres el responsable de T.I.

Por lo tanto, la empresa compra 5 computadoras personales. El administrar la seguridad y las aplicaciones de éstas 5 computadoras no va a resultar complicado, sin embargo, si la empresa en vez de tener 5 personas tiene 500, hay que empezar a pensar desde un principio cómo poder gestionar las credenciales de los diferentes usuarios. Y más si la empresa tiene alta rotación!

# ¿Qué es LDAP?

Cuando tenemos decenas de ordenadores en una red, es necesario organizar los datos correctamente y también las credenciales de los diferentes usuarios. Para poder crear una estructura jerárquica es muy importante contar con un sistema como LDAP, el cual te permitirá almacenar, administrar y proteger la información de todos los equipos adecuadamente, y también se encargará de gestionar todos los usuarios y activos.



# ¿Qué es LDAP?

El protocolo LDAP es muy utilizado actualmente por empresas que apuestan por el software libre al utilizar distribuciones de Linux para ejercer las funciones propias de un directorio activo en el que se gestionarán las credenciales y permisos de los trabajadores y estaciones de trabajo en redes LAN corporativas en conexiones cliente/servidor.

LDAP son las siglas de Protocolo Ligero de Acceso a Directorio, o en inglés Lightweight Directory Access Protocol).

Se trata de un conjunto de protocolos de licencia abierta que son utilizados para acceder a la información que está almacenada de forma centralizada en una red. Este protocolo se utiliza a nivel de aplicación para acceder a los servicios de directorio remoto.

# ¿Qué es LDAP?

Un directorio remoto es un conjunto de objetos que están organizados de forma jerárquica, tales como nombre claves direcciones, etc.

Estos objetos estarán disponibles por una serie de cliente conectados mediante una red, normalmente interna o LAN, y proporcionarán las identidades y permisos para esos usuarios que los utilicen.

# ¿Qué es LDAP?

LDAP está basado en el protocolo **X.500** para compartir directorios, y contiene esta información de forma jerarquizada y mediante categorías para proporcionarnos una estructura intuitiva desde el punto de vista de la gestión por parte de los administradores.

Es, por así decirlo, una guía telefónica, pero con más atributos y credenciales. En este caso utilizamos el término directorio para referirnos a la organización de estos objetos.

# ¿Qué es LDAP?

De forma general, estos directorios se utilizan básicamente para contener información virtual de usuarios, para que otros usuarios accedan y dispongan de información acerca de los contactos que están aquí almacenados.

Pero es mucho más que esto, ya que es capaz de comunicarse de forma remota con otros directorios LDAP situados en servidores que pueden estar en el otro lado del mundo para acceder a la información disponible.

De esta forma se crea una base de datos de información descentralizada y completamente accesible.

# Un directorio LDAP no es una base de datos

A diferencia de una base de datos, no se pueden definir atributos específicos para una entrada, solo pueden utilizarse aquellos que se especifiquen en su conjunto de clases.

Más aún, recordar que estos atributos son multivaluados frente al valor único por entrada de una BD.

Además, un LDAP no mantiene integridad referencial alguna (internamente si, pero la estructura ofrecida a los clientes no se mantiene) ni existe el concepto de transacción.

De nuevo, se hace hincapié en que los datos del árbol LDAP deberán estar organizados para que los cambios que se necesiten sean mínimos.

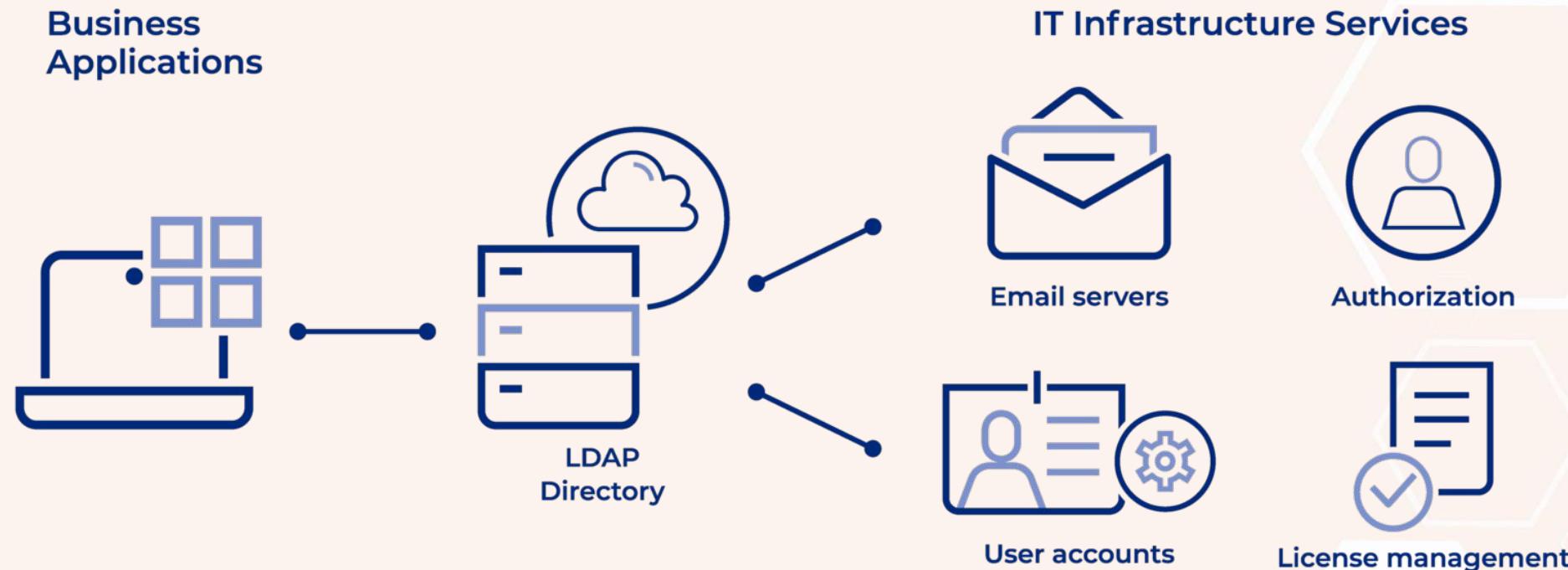
# ¿Cómo funciona LDAP?

El funcionamiento de acceso y administración es muy similar a Active Directory de Windows. Cuando el cliente LDAP se conecta con el servidor, podrá realizar dos acciones básicas, bien consultar y obtener información del directorio, o modificarla.

Si un cliente consulta la información el servidor LDAP puede conectarla directamente si tienen un directorio alojado en él, o bien redirigir la solicitud hasta otro servidor que efectivamente tenga esta información. Este podrá ser local, o remoto.

# ¿Cómo funciona LDAP?

## How LDAP Works



# ¿Cómo funciona LDAP?

Si un cliente quiere modificar la información del directorio, el servidor comprobará si el usuario que está accediendo a este directorio tiene permisos de administrador o no. Entonces, la información y gestión de un directorio LDAP se podrá hacer de forma remota.

El puerto de conexión para el protocolo LDAP es el TCP 389, aunque por supuesto, se podrá modificar por el usuario y establecerlo en el que desee si así se lo indica al servidor.

# Atributos LDAP

La estructura básica de LDAP es un árbol de nodos llamado Directory Information Tree (DIT) donde cada nodo es una entrada.

Cada entrada se define por un DN (Distinguished Name) y contiene un conjunto de atributos (pares nombre/valor).

Este DN es una cadena que indica la ruta en el árbol de dicha entrada y será único en todo el árbol.

Un directorio LDAP está concebido para favorecer las peticiones de lectura sobre las de escritura, de modo que las lecturas serán muy rápidas y se penalizarán las operaciones de escritura.

# Atributos LDAP

Con esta premisa en mente, los datos de un directorio LDAP deberán estar estructurados para cambiar lo menos posible.

Un DN tiene una estructura similar a la ruta de un fichero, solo que en orden inverso.

Esto es, los nodos de la ruta se muestran de hoja a raíz. Un ejemplo del aspecto que ofrece un DN:

```
cn=usuario1,ou=people,ou=division1,dc=miorg,dc=es
```

# Atributos LDAP

Atributo	Significado
cn	"common name", nombre
sn	"surname", apellido
uid	"userid", nombre de usuario
mail	e-mail
ou	"organizational unit", unidad organizativa
telephoneNumber	Número de teléfono

# ¿Qué se requiere para implementar LDAP?

Mientras que la autenticación LDAP ha probado con certeza ser efectiva, la cantidad de tiempo requerido para implementar y personalizar la infraestructura basada en LDAP para satisfacer las necesidades de gestión de identidad de una organización moderna puede ser importante.

Tradicionalmente, LDAP ha sido también una implementación local, que requiere de servidores que deben ser integrados dentro de la infraestructura completa de la identidad de gestión de una organización (la cual ha resultado tradicionalmente ser local).

# ¿Qué se requiere para implementar LDAP?

Esta clase de instalación puede resultar difícil de lograr, especialmente para las organizaciones pequeñas o en la cloud. Después de todo, la mayoría de las organizaciones desearían cambiar su infraestructura completa de gestión de la identidad local hacia la cloud.

# 4. Recuperación de usuarios y roles respaldados por LDAP

## 4.2 Instalación y configuración de Apache Directory Studio



# Instalación

Apache Directory es un proyecto de la Apache Software Foundation dedicado a crear soluciones en torno a los servicios de directorio.

En el contexto de este proyecto, las tecnologías más relevantes son Lightweight Directory Access Protocol y Kerberos; para ambos protocolos Apache Directory ofrece implementaciones libres en Java bajo Licencia Apache.

En la actualidad existen los siguientes subproyectos:

- Apache Directory Server - un servidor LDAP implementado en Java
- Apache Directory Triplesec - un servidor que soporta una autenticación de dos factores
- Apache Directory Studio - herramienta basada en Eclipse para LDAP los servicios de directorio

# MacOS (chip M1/M2/M3)

Si se requiere instalar en MacOS, a veces es un poco complicado

- Ya sea por las versiones de JAVA, hay que usar x86\_64
- Se requiere Java 11 forzoso

Hay que descargar una versión en particular de Java (Temurin) en conjunto con apache directory studio

- <https://github.com/adoptium/temurin11-binaries/releases> (x86\_64)
- <https://directory.apache.org/studio/>

# MacOS (chip M1/M2/M3)

Ahora, hay que entrar a Finder/Applications

- Ahí hay que darle click derecho a ApacheDirectoryStudio.app y elegir “Show Package Contents”
- Ir a Contents/info.plist y modificar el apartado de xml donde apunta a ejecutar cierta jdk

```
<string>-vm</string>
<string>/Library/Java/JavaVirtualMachines/jdk-11.0.22+7/Contents/Home/bin/java</string>
```

# Windows

Se requiere utilizar Java 11 forzosamente

Ingresar a la liga y descargar:

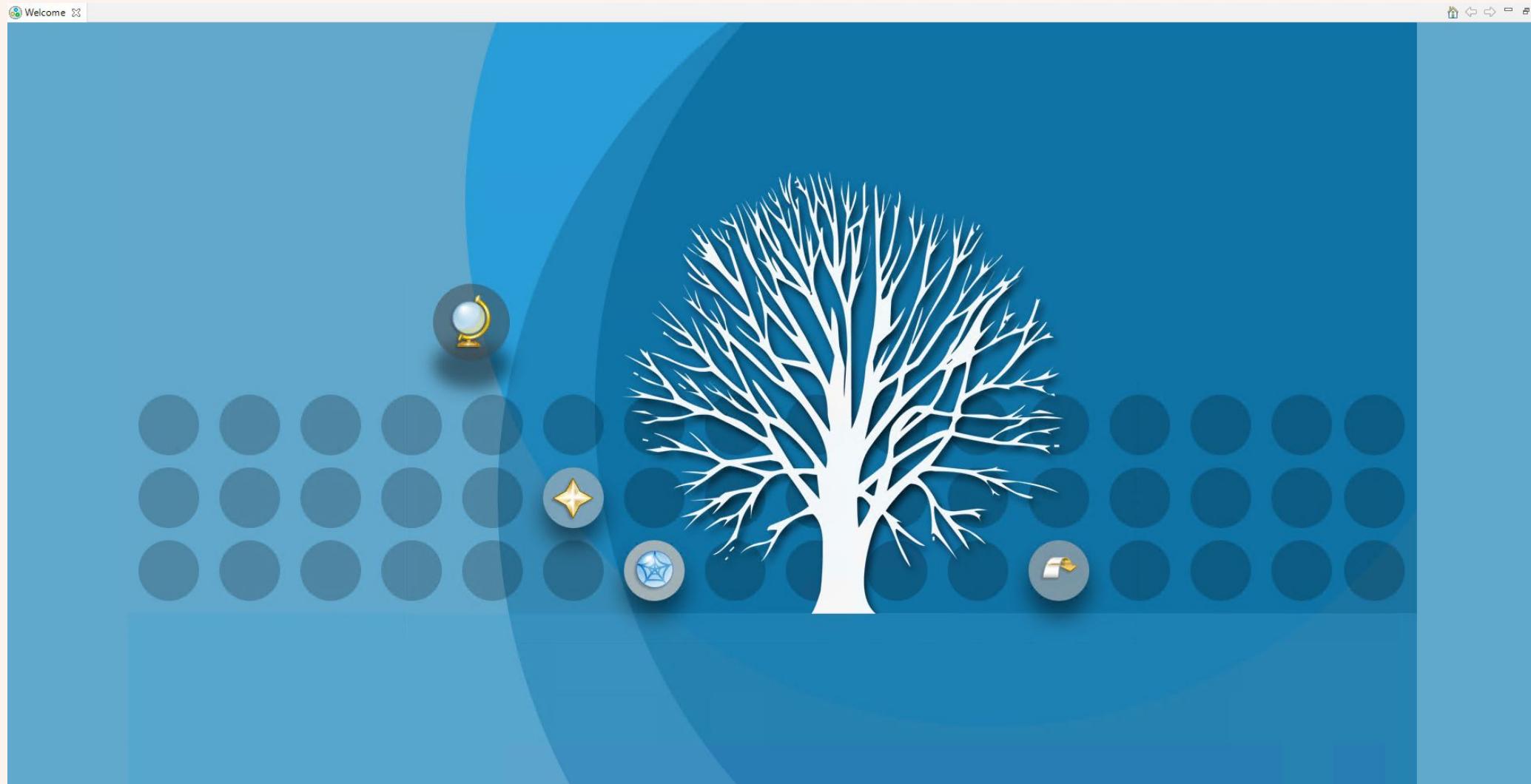
- <https://jdk.java.net/archive/> (version 11)
- <https://directory.apache.org/studio/download/download-windows.html>

Instalar

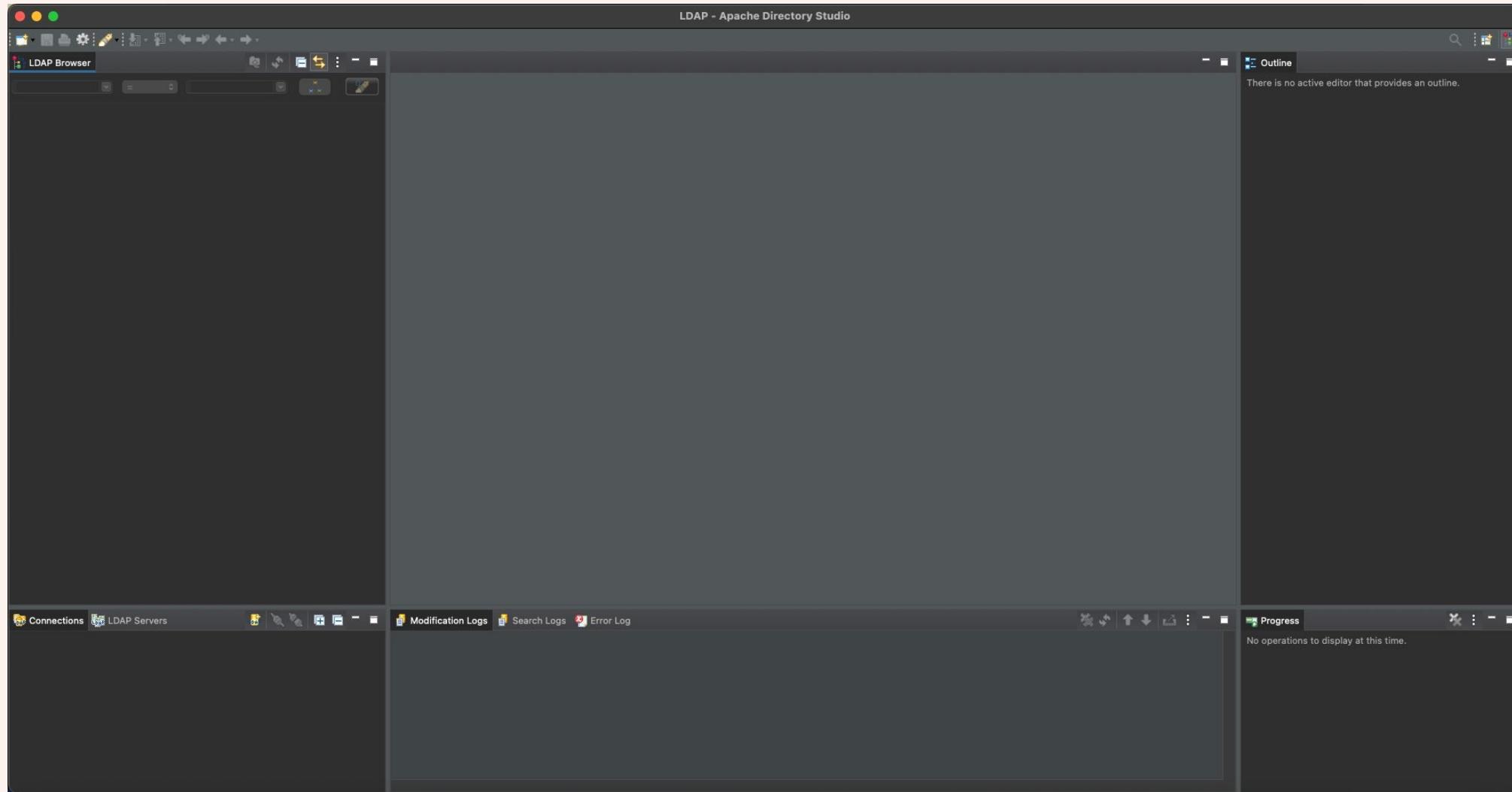
- Next, next, next, finish



# Apache Directory Studio

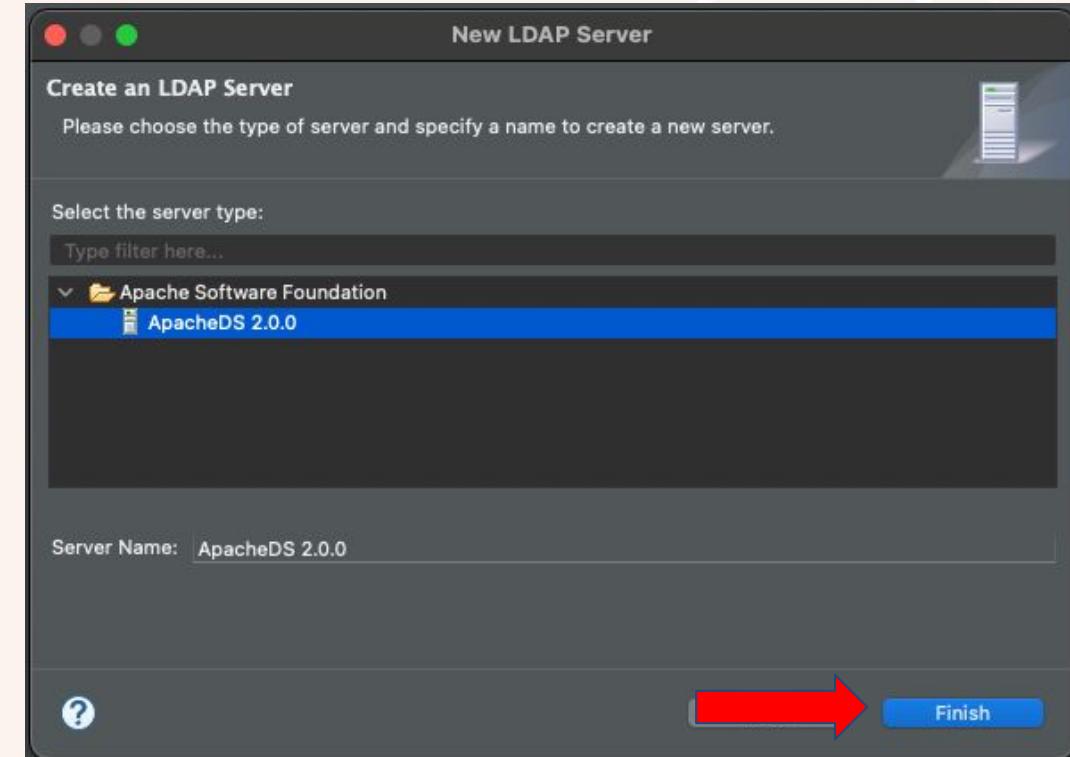
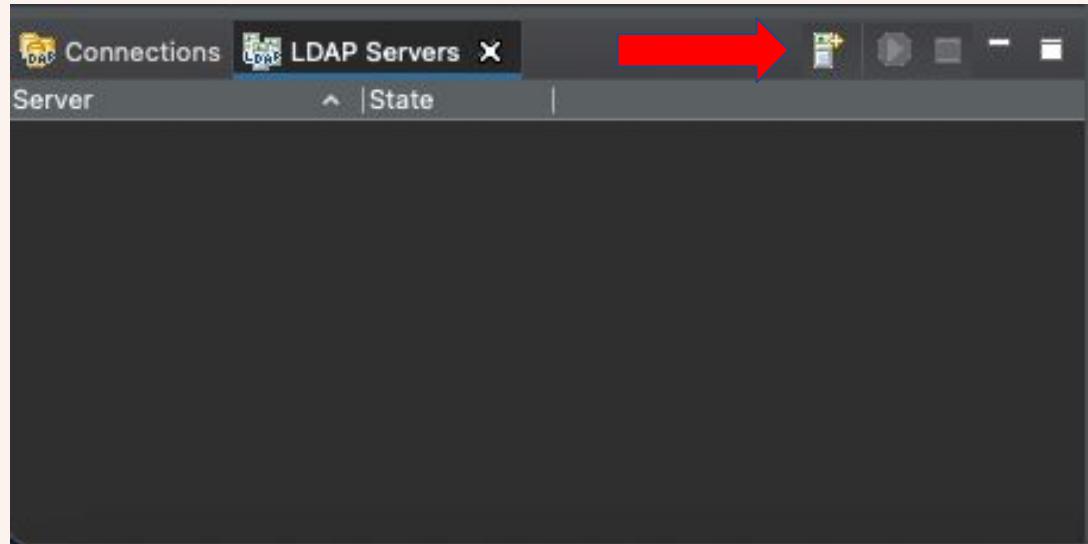


# Apache Directory Studio



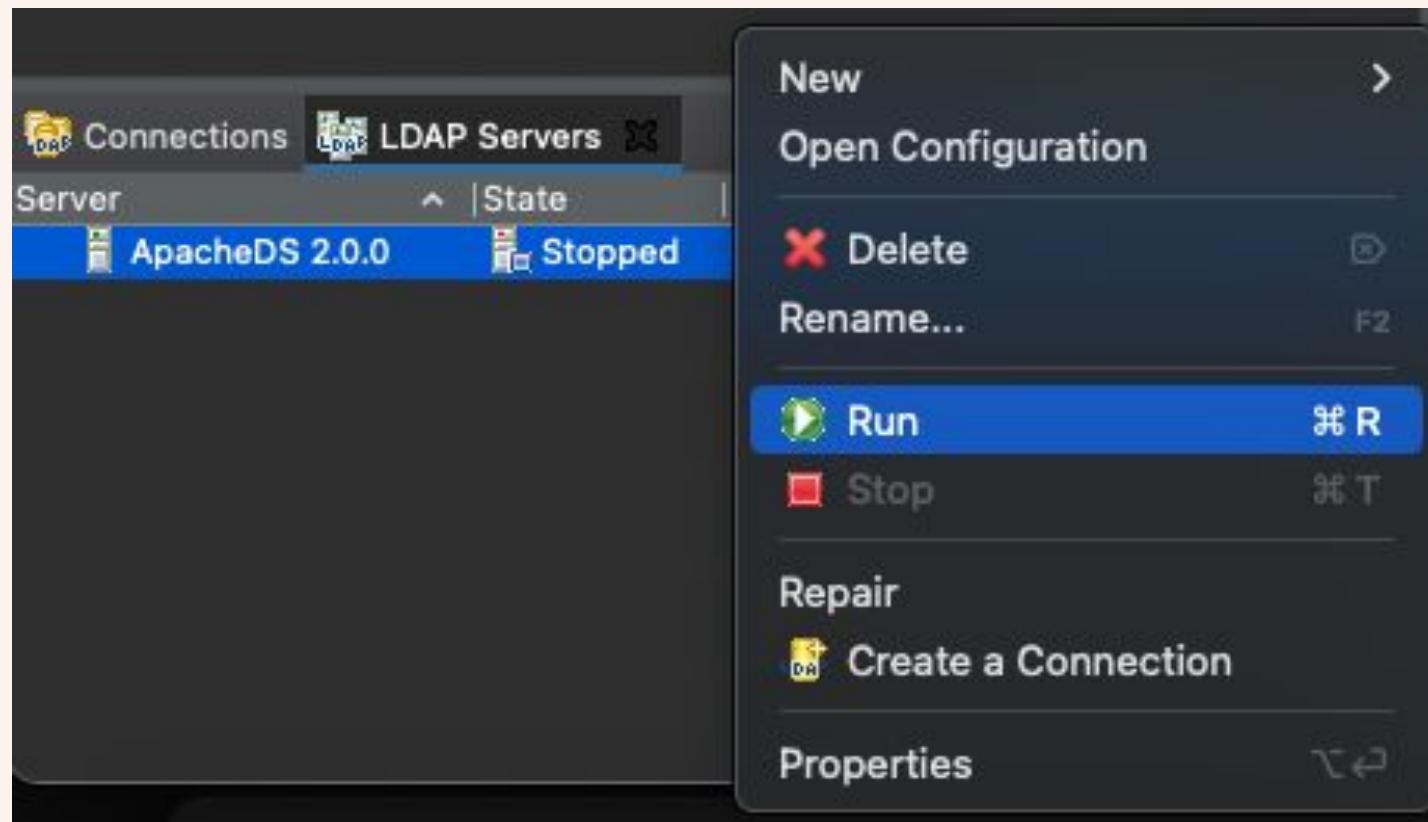
# Apache Directory Studio

Del lado inferior izquierdo, hay que seleccionar LDAP Servers, y crear uno:



# Ejecutar ADS

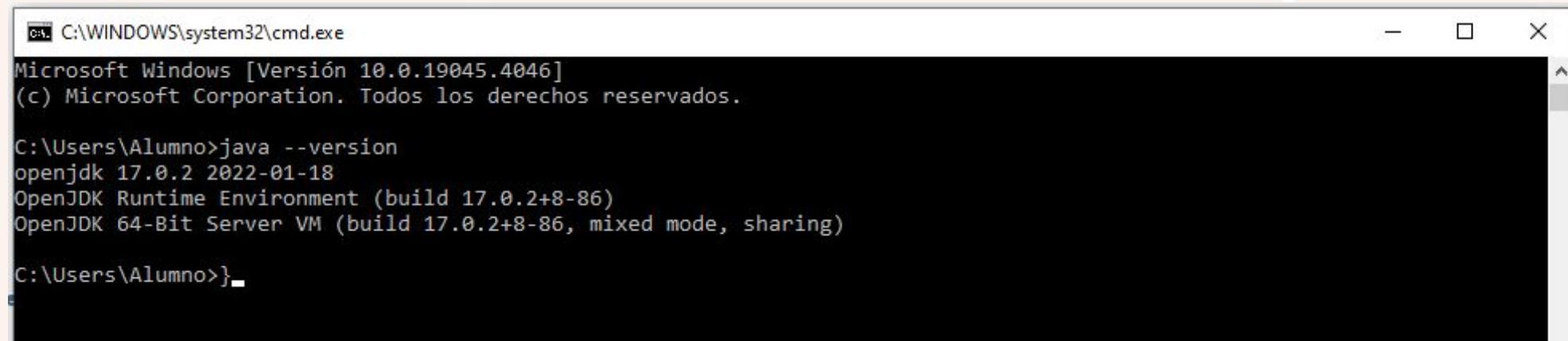
Ahora, hay que darle click derecho y correr servidor



# Ejecutar ADS

Es muy probable que en este proceso se rompa, es decir, que no levante el servidor de LDAP.

En este caso, hay que revisar en CMD la versión de Java



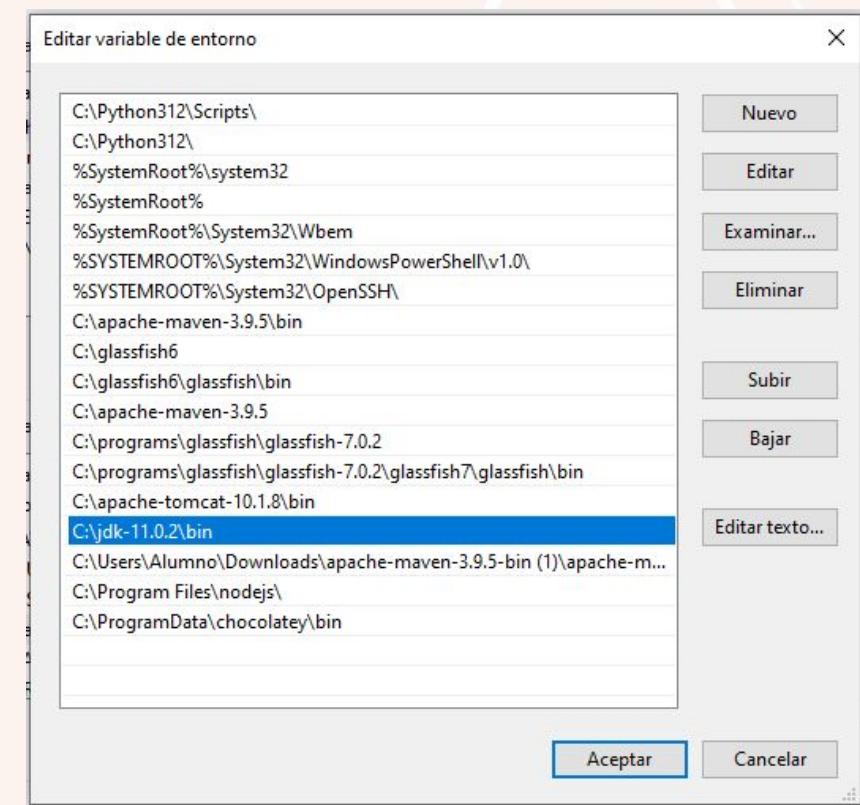
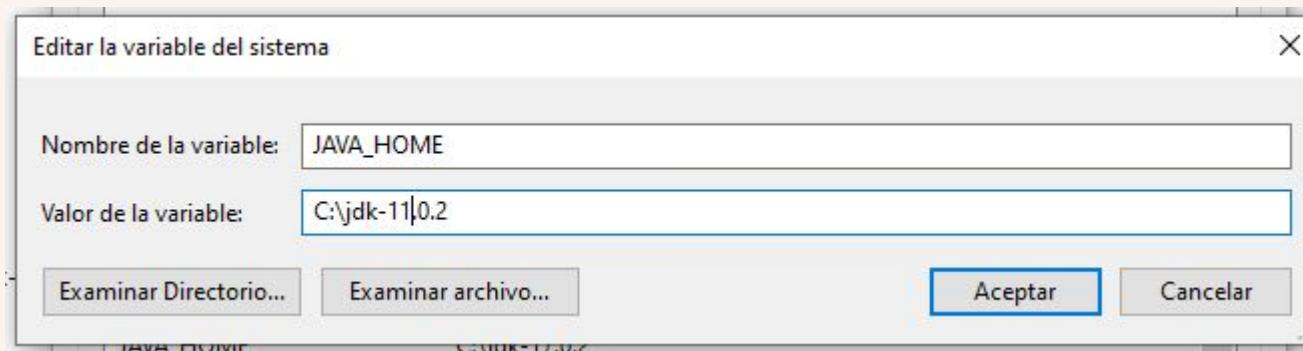
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Versión 10.0.19045.4046]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Alumno>java --version
openjdk 17.0.2 2022-01-18
OpenJDK Runtime Environment (build 17.0.2+8-86)
OpenJDK 64-Bit Server VM (build 17.0.2+8-86, mixed mode, sharing)

C:\Users\Alumno>
```

# Ejecutar ADS

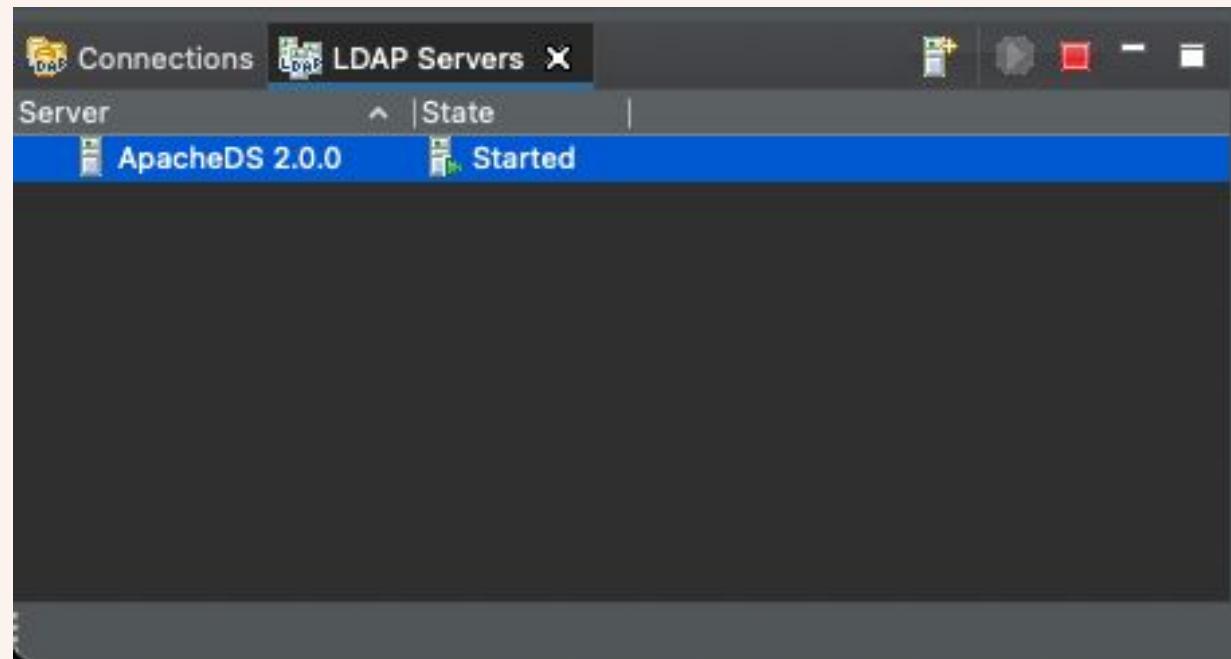
En caso de tener una versión diferente de la 11, entonces hay que cerrar ADS, entrar a las variables de entorno de Windows y modificar el path de Java para que apunte a nuestra descarga de openjdk 11.



# Ejecutar ADS

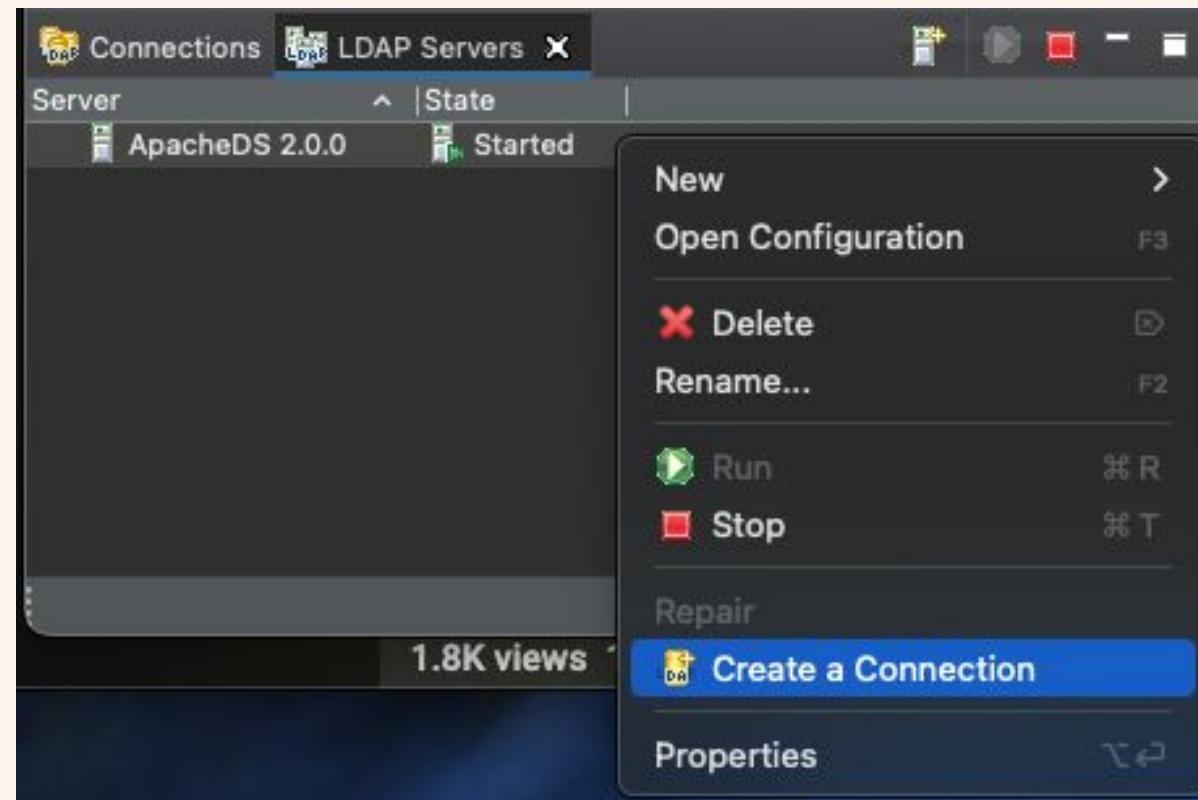
Una vez que tengamos bien referenciada nuestra JDK11, ahora si procedemos a abrir ADS e intentar correr nuestro servidor de LDAP.

En esta ocasión, debería estar funcionando correctamente!



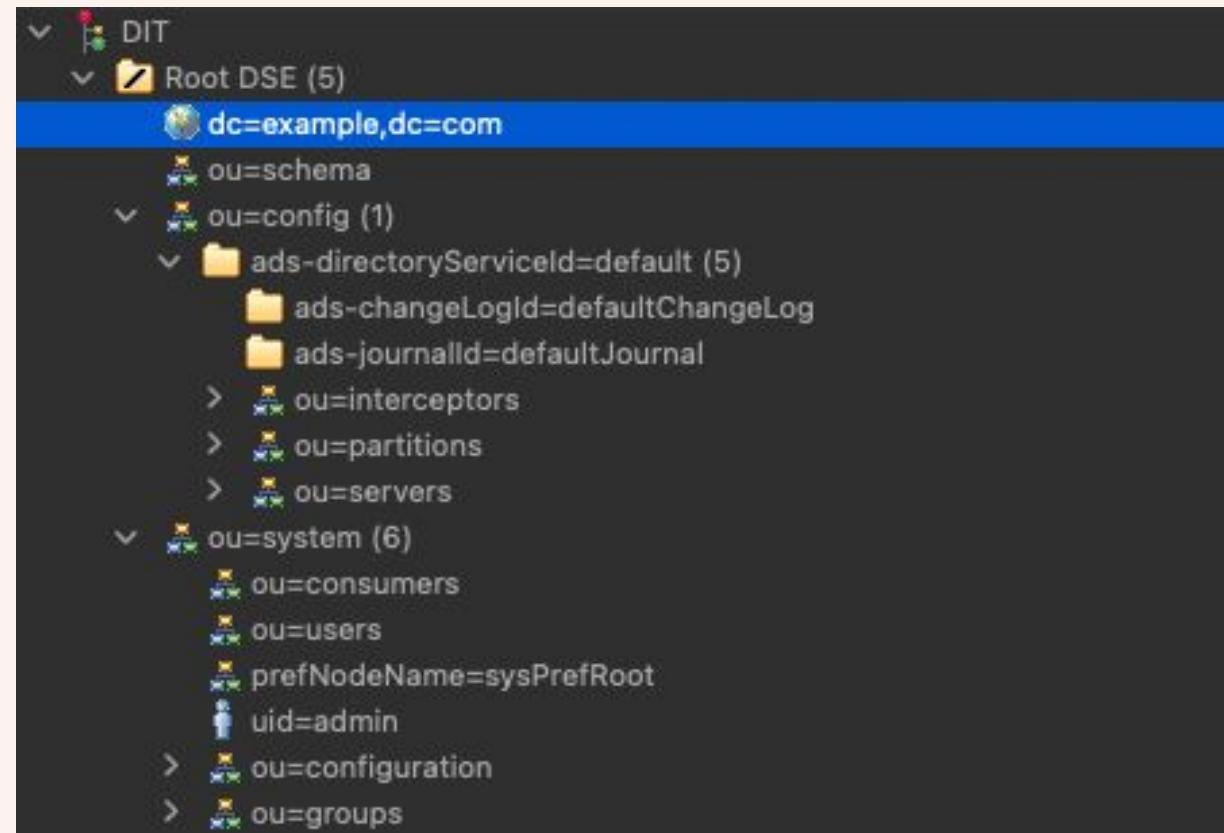
# Crear conexión

Hay que dar click derecho y creamos la conexión



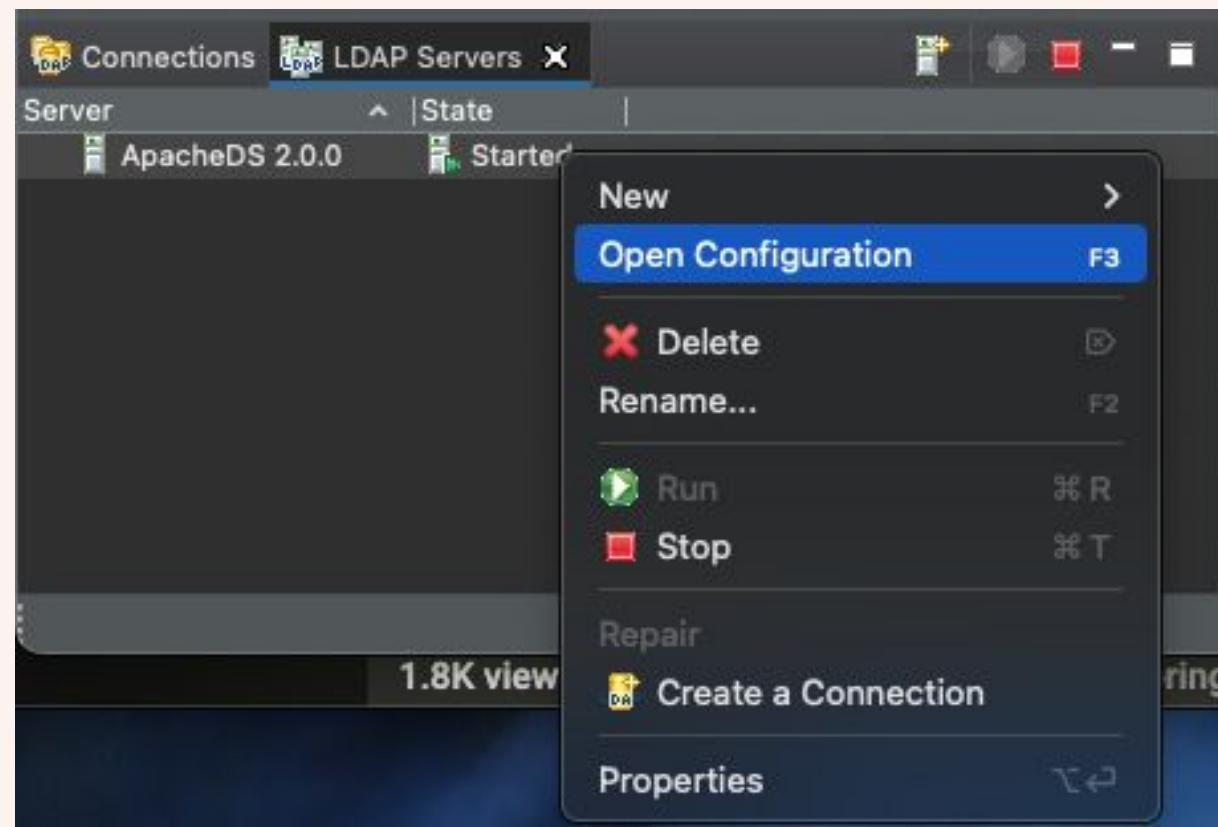
# Crear conexión

Si hacemos eso, nos muestra una pantalla similar:

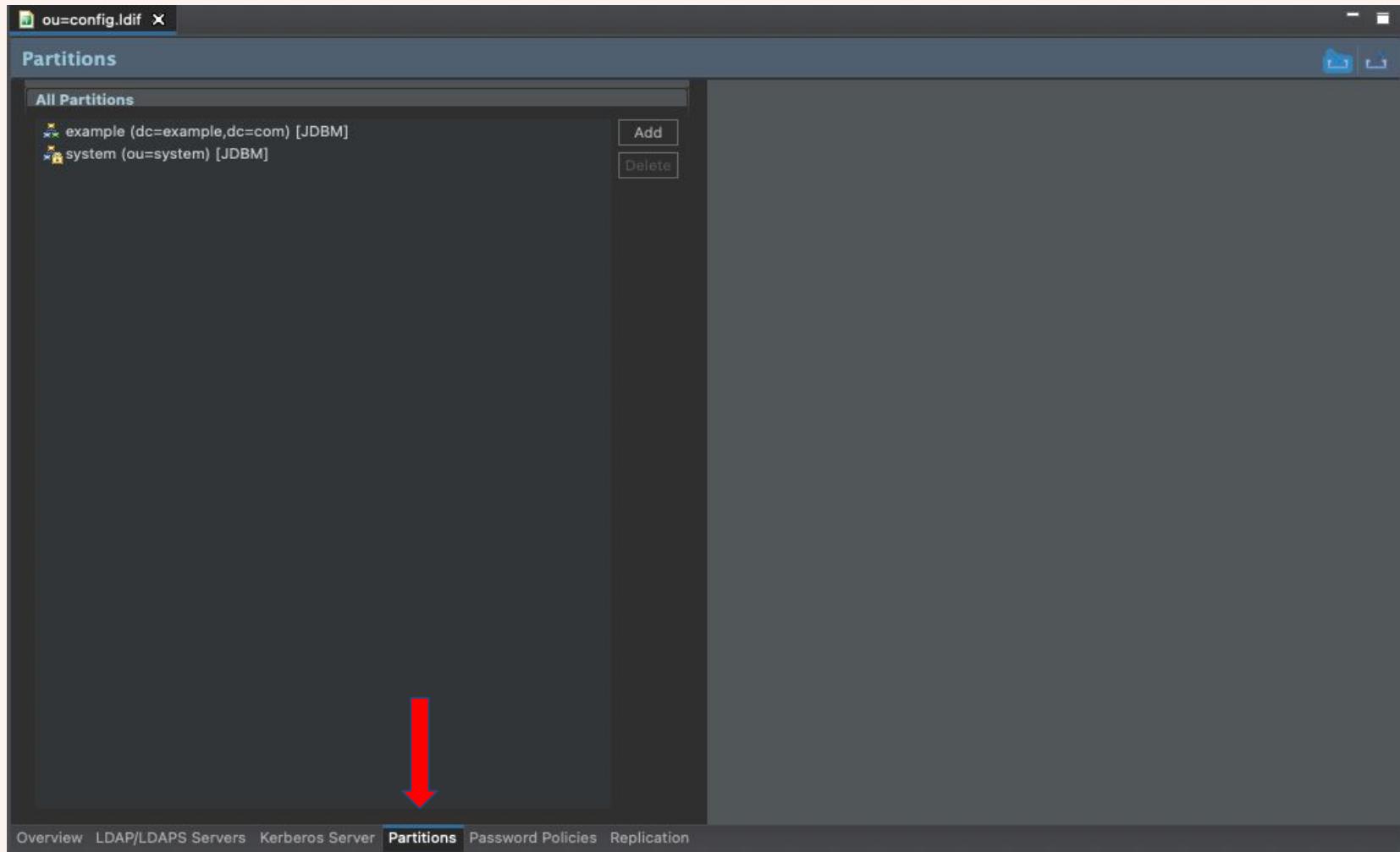


# Crear conexión

Ahora vamos a configurar la conexión



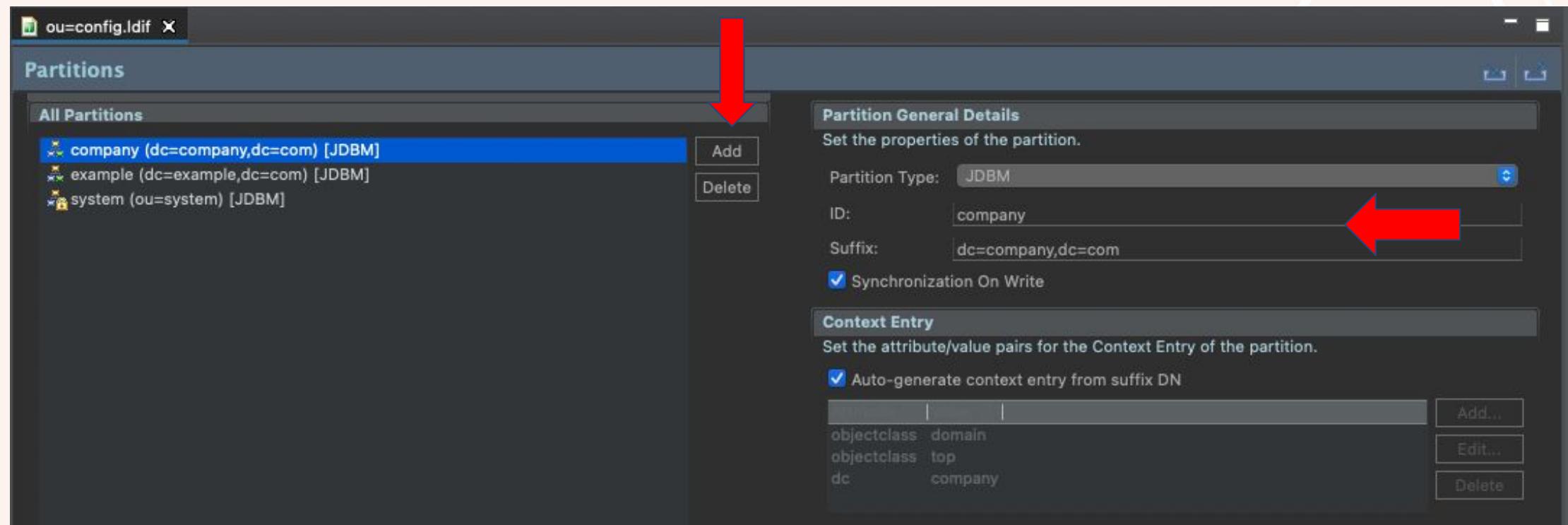
# Creación CN



# Creación CN

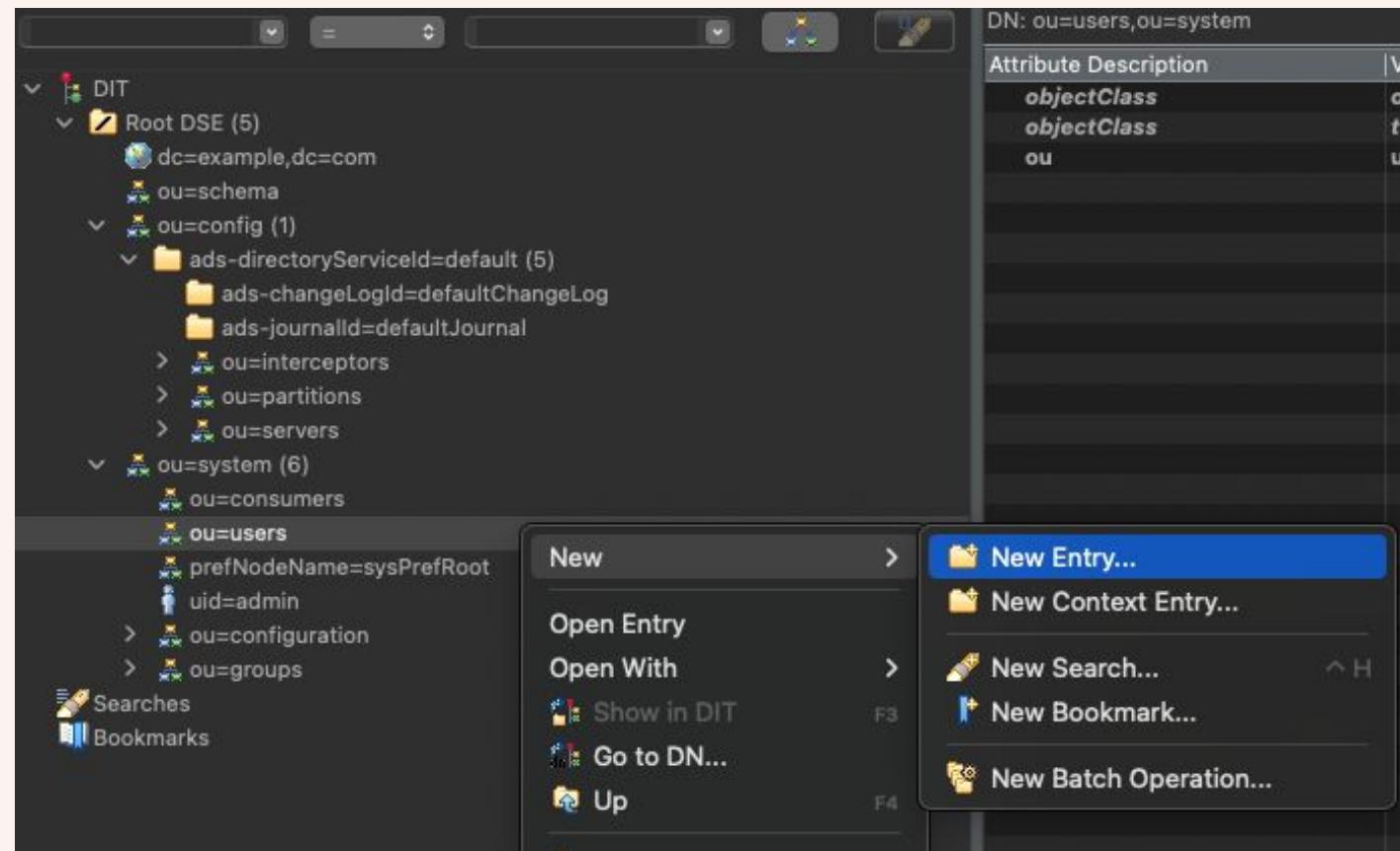
cn (common name) - Atributo de nombre común, que contiene el nombre del objeto

dc (domain component) - Atributo de componente de dominio



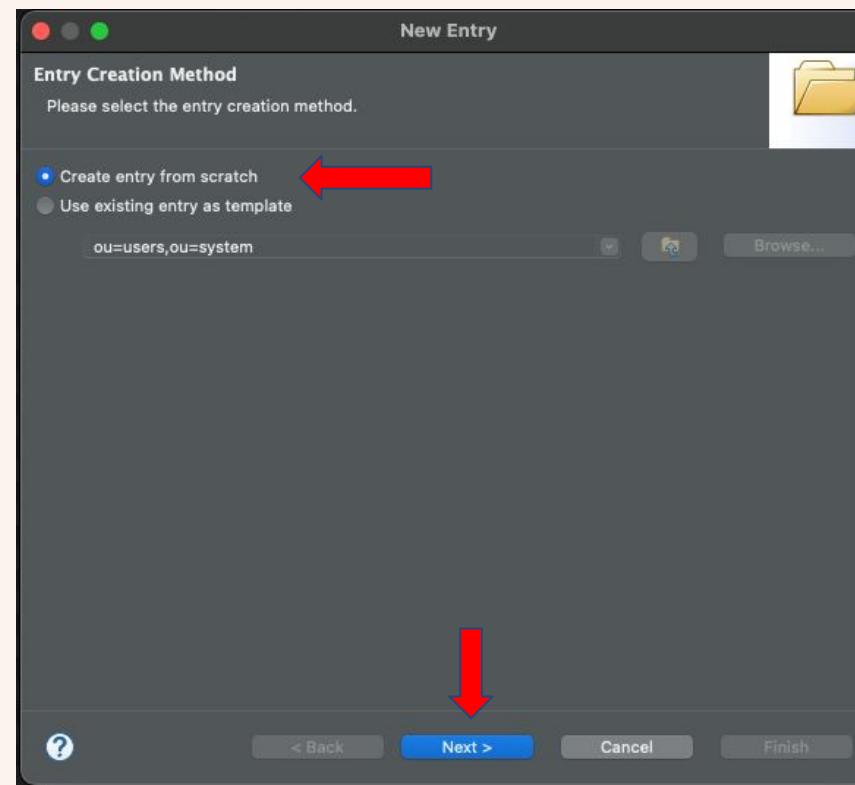
# Create User

Ahora procedemos a crear un usuario nuevo con su contraseña

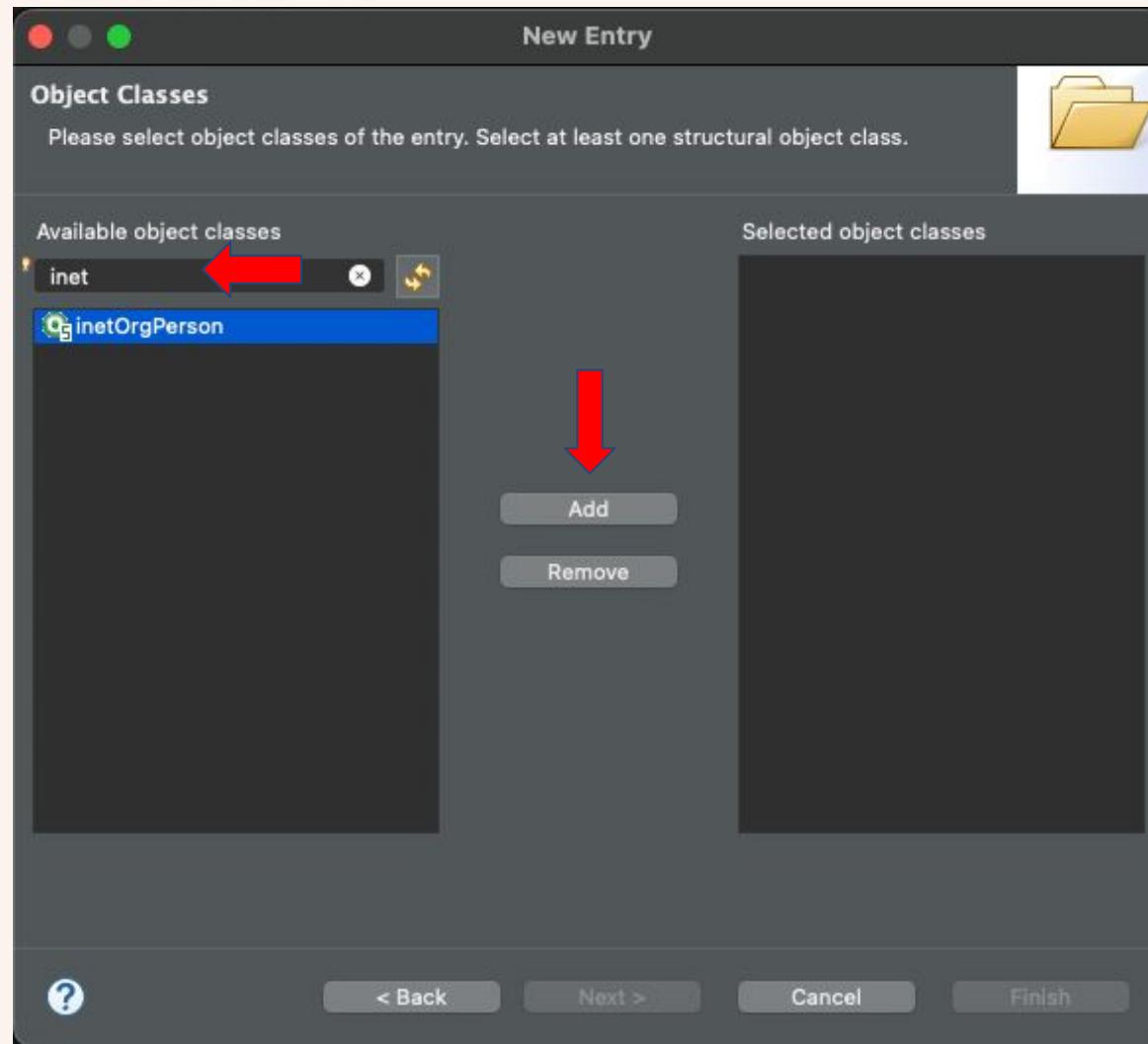


# Create User

ou (organizational unit)- Atributo de nombre de organización que contiene el nombre de la organización

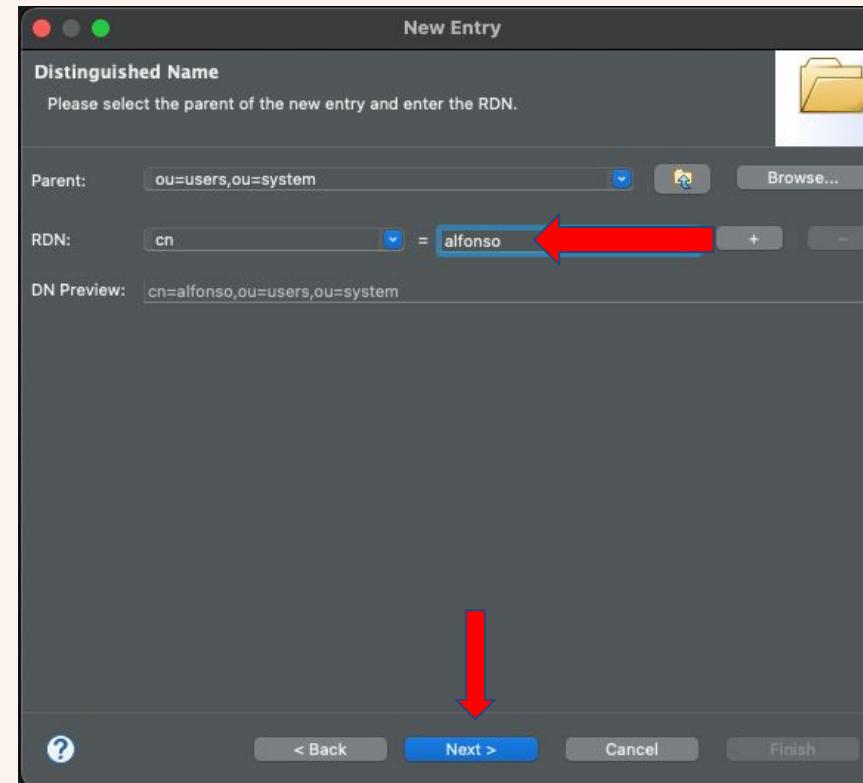


# Create User



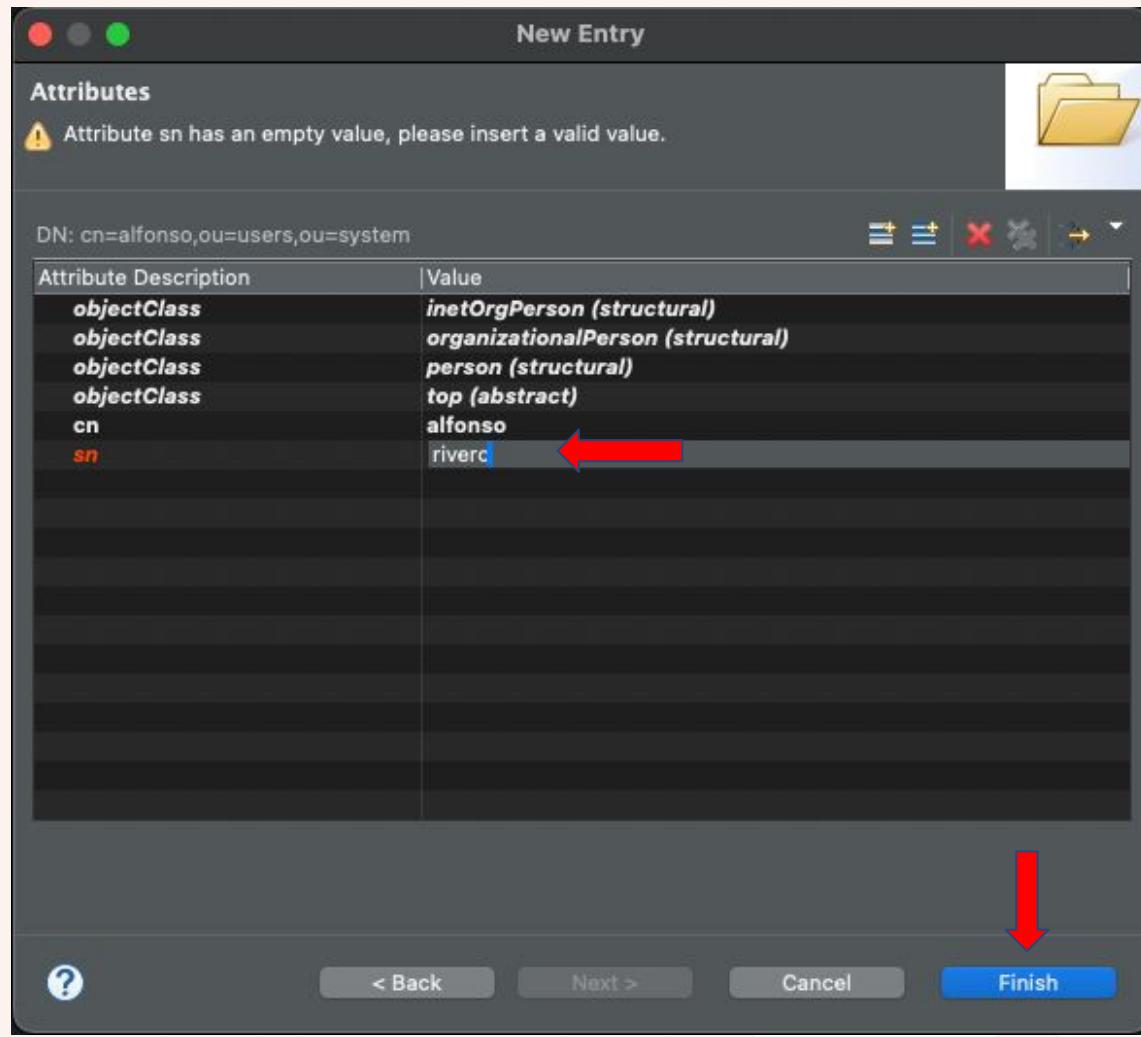
# Create User

RDN (relative distinguished name) - El nombre distinguido relativo (RDN) de un objeto. Un RDN es la parte relativa de un nombre distinguido (DN), que identifica de forma única un objeto LDAP.



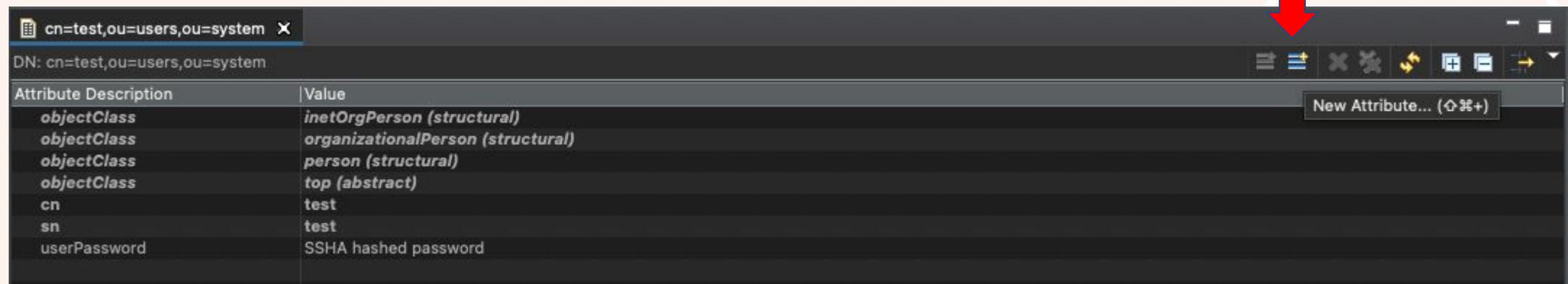
# Create User

sn (surname) - apellido



# Create User

Ahora agregamos atributos al surname



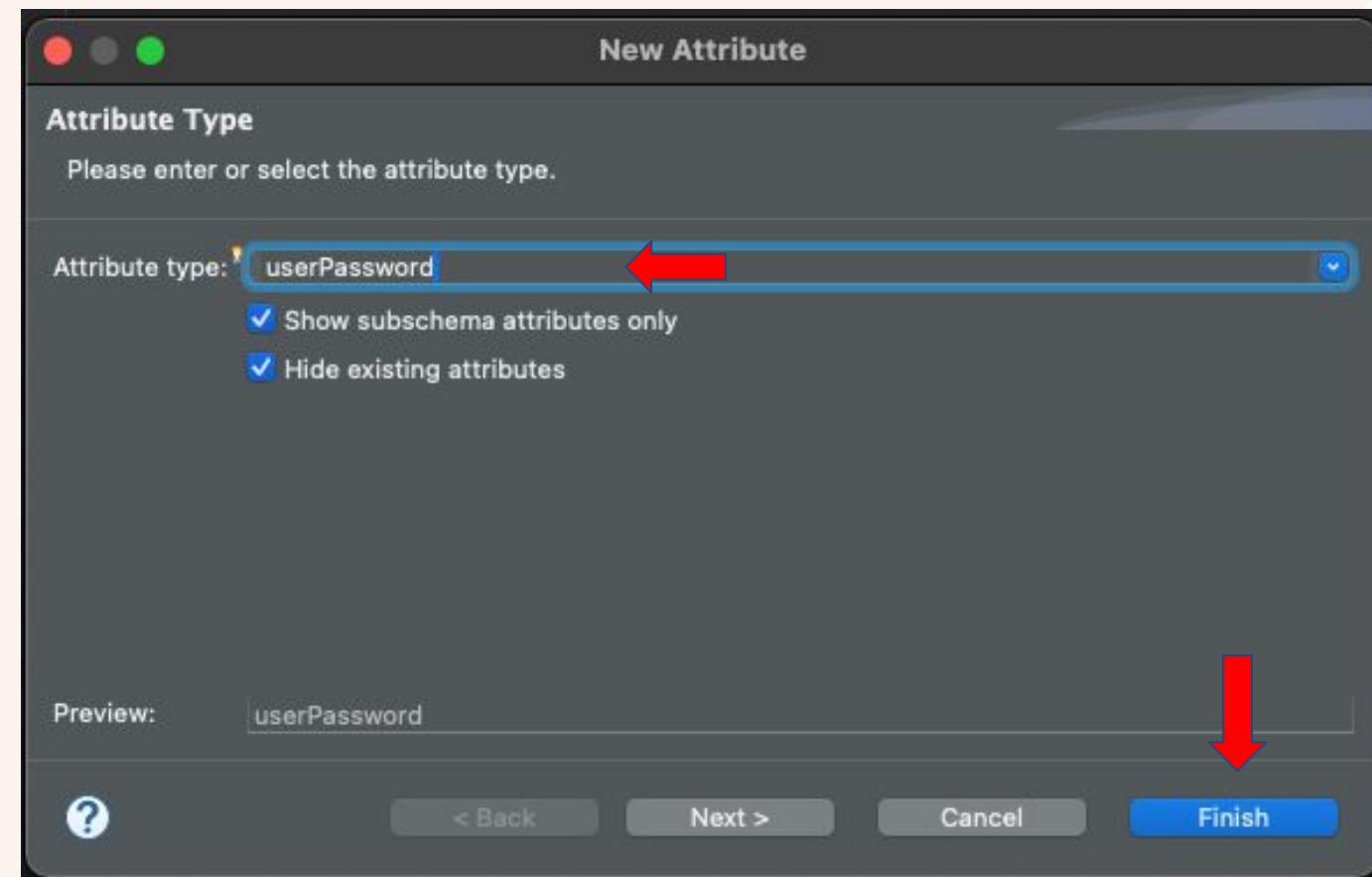
The screenshot shows a user interface for managing LDAP attributes. The title bar indicates the entry is located at `cn=test,ou=users,ou=system`. The DN field also shows `cn=test,ou=users,ou=system`. The main table lists attributes and their values:

Attribute Description	Value
<code>objectClass</code>	<code>inetOrgPerson (structural)</code>
<code>objectClass</code>	<code>organizationalPerson (structural)</code>
<code>objectClass</code>	<code>person (structural)</code>
<code>objectClass</code>	<code>top (abstract)</code>
<code>cn</code>	<code>test</code>
<code>sn</code>	<code>test</code>
<code>userPassword</code>	SSHA hashed password

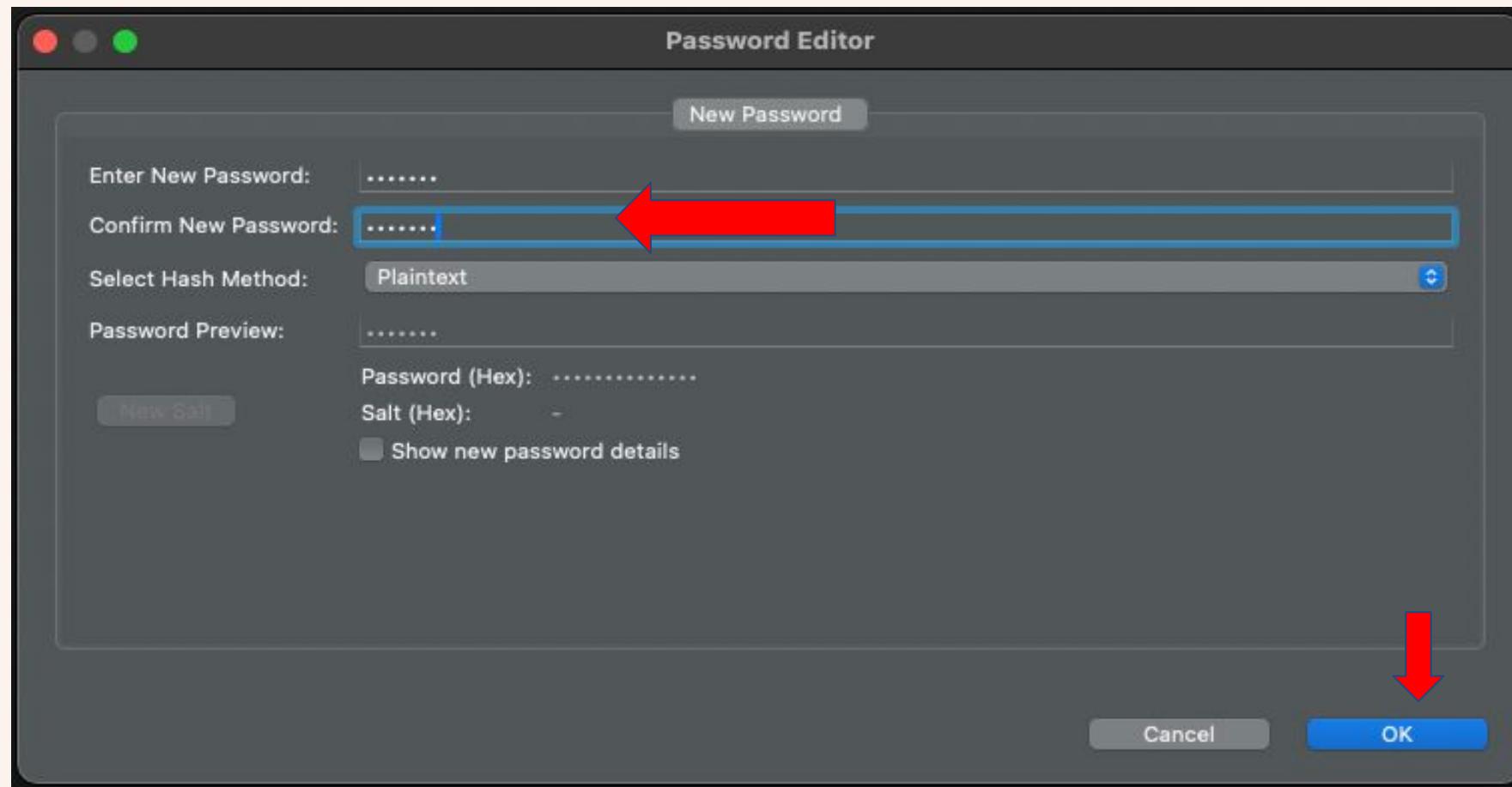
A red arrow points to the "New Attribute..." button in the toolbar.

# Create User

Hay que agregar una contraseña



# Create User



# 4. Recuperación de usuarios y roles respaldados por LDAP

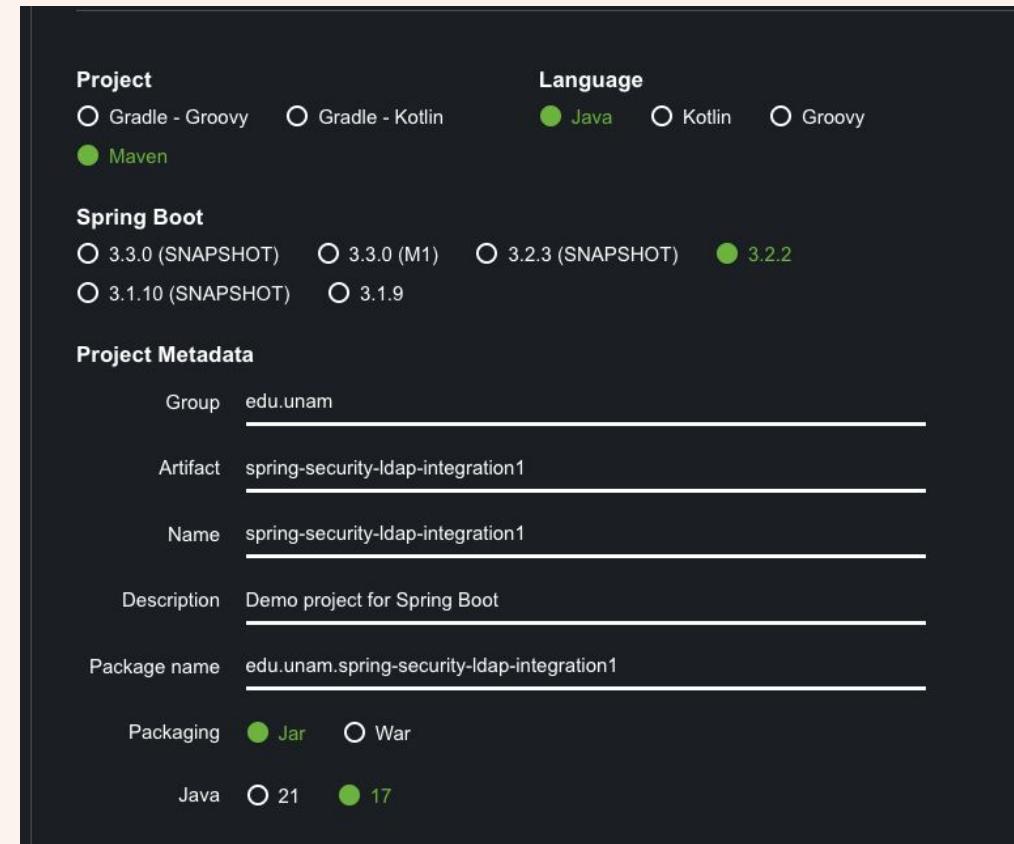
## 4.3 Configuración básica de LDAP en Spring



file: [spring-security-ldap-integration1.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# Creación de un nuevo proyecto

Vamos a tener que crear un nuevo proyecto con spring initializer



# POM

La configuración básica es sumamente sencilla. Lo más complicado es entender cómo trabaja LDAP.

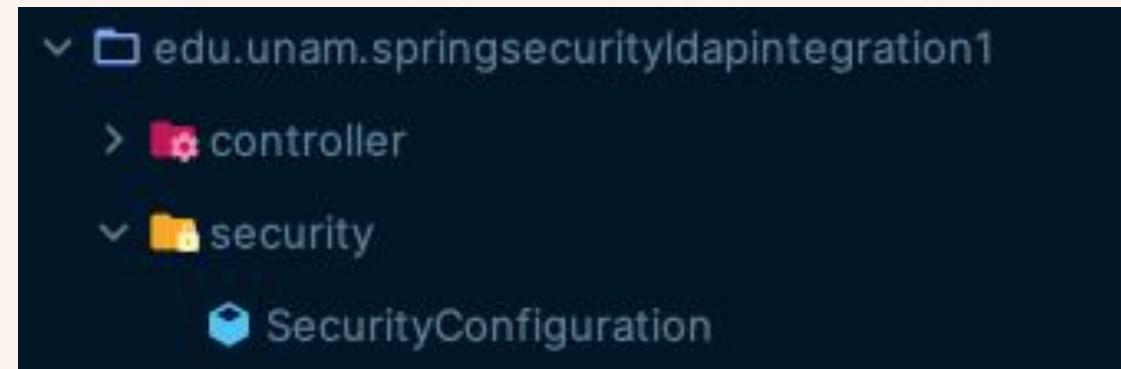
Como ya sabemos, lo primero que debemos hacer es descargar las dependencias de nuestro proyecto.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-ldap</artifactId>
    </dependency>
</dependencies>
```

# Encarpetado

Procedemos a trabajar con el encarpetado ya conocido

Y luego pasamos a crear nuestra clase **SecurityConfiguration**



# SecurityConfiguration

Como ya vimos, esta es nuestra clase principal

Trabajaremos con un limpio, con características default.

Tenemos 3 beans nuevos:

- LdapTemplate
- LdapContextSource
- AuthenticationManager

```
...
1 @Configuration
2 @EnableWebSecurity
3 public class SecurityConfiguration {
4     @Bean
5     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
6         http.authorizeHttpRequests((authorize) -> authorize.anyRequest().fullyAuthenticated())
7             .formLogin(Customizer.withDefaults()); // trabajamos con login default
8         return http.build();
9     }
10    @Bean
11    public LdapTemplate ldapTemplate() {
12        return new LdapTemplate(contextSource()); // agregamos un bean de LdapTemplate como context source
13    }
14    @Bean
15    public LdapContextSource contextSource() {
16        // configuramos parámetros de LDAP
17        LdapContextSource ldapContextSource = new LdapContextSource();
18        ldapContextSource.setUrl("ldap://localhost:10389");
19        ldapContextSource.setUserDn("uid=admin,ou=system");
20        ldapContextSource.setPassword("secret");
21        return ldapContextSource;
22    }
23    @Bean
24    AuthenticationManager authManager(BaseLdapPathContextSource source) {
25        // Trabajamos con el context source de LDAP en nuestro proyecto
26        LdapBindAuthenticationManagerFactory factory = new LdapBindAuthenticationManagerFactory(source);
27        factory.setUserDnPatterns("cn={0},ou=users"); // configuramos el Patrón User DN
28        return factory.createAuthenticationManager();
29    }
30 }
```

# LdapContextSource

LdapContextSource se utiliza para crear un LdapTemplate.

Aquí va la configuración del contexto, es decir, la URL del servidor de LDAP, el UserDn y la contraseña origen para poder loguearse a LDAP.

```
1 @Bean
2 public LdapContextSource contextSource() {
3     // configuramos parámetros de LDAP
4     LdapContextSource ldapContextSource = new LdapContextSource();
5     ldapContextSource.setUrl("ldap://localhost:10389");
6     ldapContextSource.setUserDn("uid=admin,ou=system");
7     ldapContextSource.setPassword("secret");
8     return ldapContextSource;
9 }
```

# LdapTemplate

LdapTemplate se utiliza para la creación y modificación de entradas LDAP.

Se requiere para poder cargar el contexto (una carga de contexto similar a como lo vimos en hibernate)

```
1 @Bean
2 public LdapTemplate ldapTemplate() {
3     return new LdapTemplate(contextSource()); // agregamos un bean de LdapTemplate como context source
4 }
```

# AuthenticationManager

La interfaz **AuthenticationManager** en el contexto de la seguridad de Spring es una parte fundamental del proceso de autenticación de usuarios en una aplicación.

Su función principal es autenticar a los usuarios en función de sus credenciales (como nombre de usuario y contraseña) y decidir si se les permite o no acceder a recursos protegidos.

# AuthenticationManager

**AuthenticationManager**: Es una interfaz que define un solo método: `authenticate`. Su objetivo es tomar una instancia de `Authentication`, que generalmente es un objeto que contiene las credenciales del usuario (como `UsernamePasswordAuthenticationToken`), y determinar si es una autenticación válida.

La implementación de `AuthenticationManager` en Spring se encarga de realizar la autenticación, como verificar si las credenciales son correctas y cargar los detalles del usuario autenticado en un objeto `Authentication`.

Puedes personalizar la implementación de `AuthenticationManager` según tus necesidades, por ejemplo, para utilizar diferentes métodos de autenticación (por contraseña, por token, Idap, etc.).

# AuthenticationManager

```
1 @Bean
2 AuthenticationManager authManager(BaseLdapPathContextSource source) {
3     // Trabajamos con el context source de LDAP en nuestro proyecto
4     LdapBindAuthenticationManagerFactory factory = new LdapBindAuthenticationManagerFactory(source);
5     factory.setUserDnPatterns("cn={0},ou=users"); // configuramos el Patrón User DN
6     return factory.createAuthenticationManager();
7 }
```

# Controller

Crearemos una clase llamada **LdapAuthenticationController**, donde tendremos únicamente un GET, y para realizarlo de manera más sencilla, trabajaremos un poco con REST.

- Recordar que @RestController es diferente que @Controller.

Éste endpoint nos va a funcionar para poder probar que nuestra integración con LDAP está funcionando.

- Recordar que LDAP no nos trabaja las autorizaciones, pero si la autenticación

```
 1 @RestController
 2 public class LdapAuthenticationController {
 3     @GetMapping("/")
 4     public String index() {
 5         return "Welcome to the home page!";
 6     }
 7 }
```

# application.properties

Y no olvidemos nuestro archivo properties

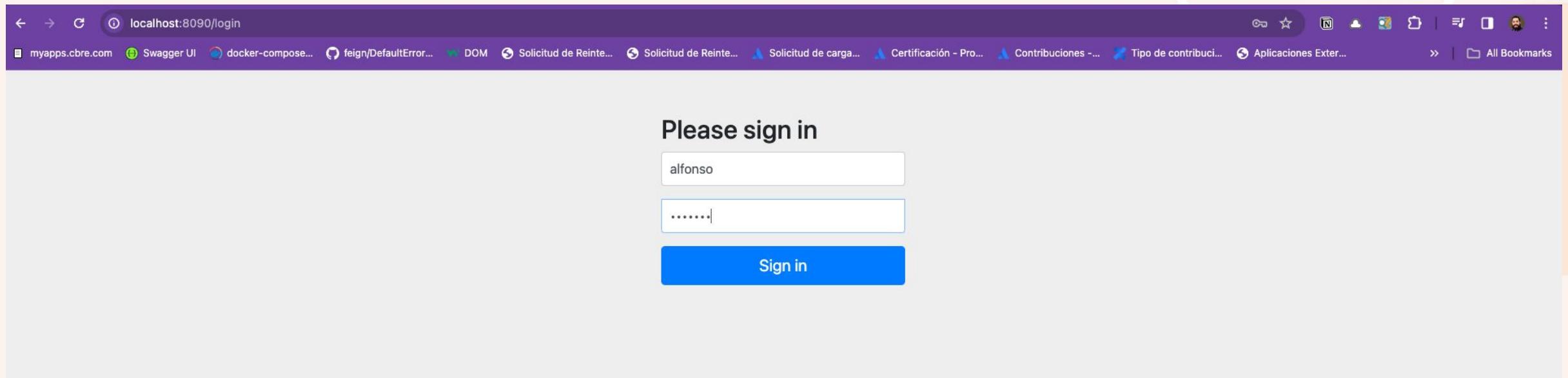
```
# =====  
# = GLOBAL CONFIG  
# =====  
server.port=8090
```

# Ejecución de proyecto

Primero hay que cerciorarnos de que el servidor de LDAP esté funcionando.

Luego, ejecutar nuestro proyecto en Spring.

- Hay que entrar con nuestro usuario y contraseña configuradas en LDAP.



# 4. Recuperación de usuarios y roles respaldados por LDAP

## 4.4 Trabajando con UnboundID LDAP



file: [spring-security-ldap-unboundid.zip](#)  
DDTIC\_DSJ\_PLI\_2024

# ¿Qué es UnboundID LDAP?

El SDK LDAP de UnboundID para Java es una biblioteca Java rápida, potente, fácil de usar y completamente gratuita para comunicarse con servidores de directorio LDAP.

Ofrece mejor rendimiento, mayor facilidad de uso y más funciones que otras API LDAP basadas en Java.

El SDK LDAP de UnboundID para Java se puede usar y redistribuir de forma gratuita en aplicaciones de código abierto o propietarias.

No tiene dependencias de terceros, por lo que un único archivo jar es todo lo que necesita para agregar soporte LDAP de primer nivel a su aplicación Java.

# POM

Hay que agregar un par de dependencias

```
<dependency>
    <groupId>org.springframework.ldap</groupId>
    <artifactId>spring-ldap-core</artifactId>
</dependency>
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
</dependency>
```

# SecurityConfiguration

Tomaremos como base, la documentación oficial de Spring Security.

- <https://spring.io/guides/gs/authenticating-ldap>

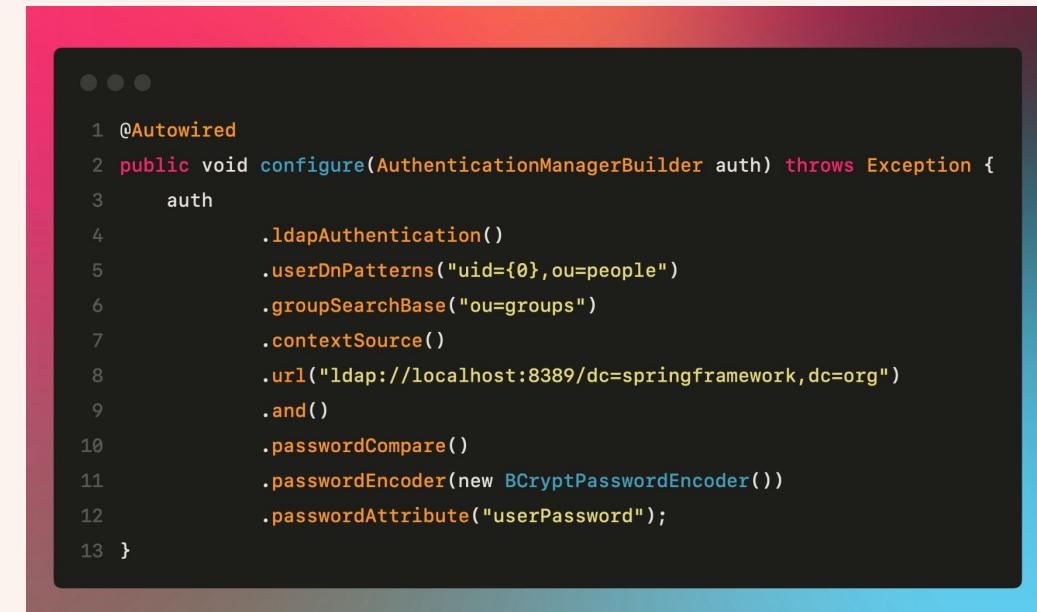
Aquí nos pide configurar esta clase de cierta manera.

- Aquí se define que vamos a trabajar con un archivo extensión Idif

Agregamos un método configure que trabaja con **AuthenticationManagerBuilder**

- Dependiendo de cómo esté configurado el archivo Idif, tenemos que configurar el dnPattern, searchBase, url, etc.

```
1  ...
2  ...
3  ...
4  ...
5  ...
6  ...
7  ...
8  ...
9  ...
10 ...
11 ...
12 ...
13 }
```



# application.properties

De igual manera, hay que agregar unas configuraciones en el archivo properties

```
# =====
# = GLOBAL CONFIG
# =====

server.port=8090

spring.ldap.embedded.ldifclasspath:ldap-data.ldif
spring.ldap.embedded.base-dn=dc=springframework,dc=org
spring.ldap.embedded.port=8389
```

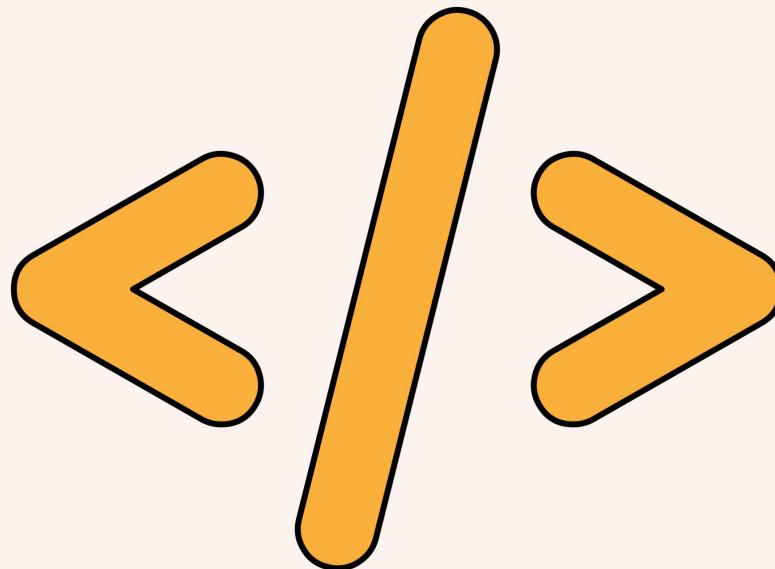
# LDIF

Y finalizamos con el archivo ldif que vamos a tener que crear dentro de la carpeta resources.

Lo vamos a llamar ldap-data.ldif

Este va a ser nuestro archivo a analizar.

# Ejercicio 3



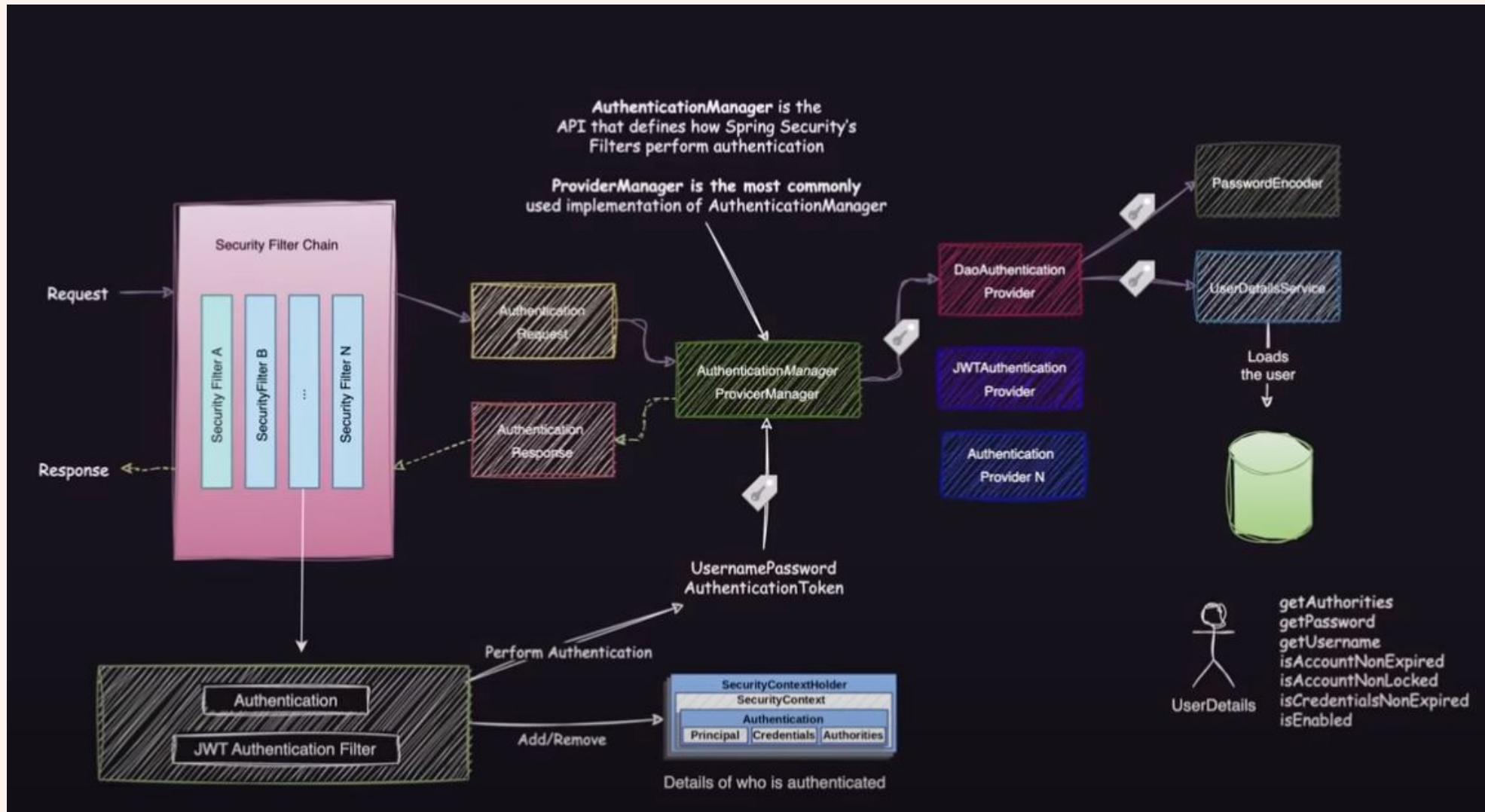
# 5. Spring Security y OAuth2 con JWT

5.1 Arquitectura de Spring Security con JWT y Thymeleaf

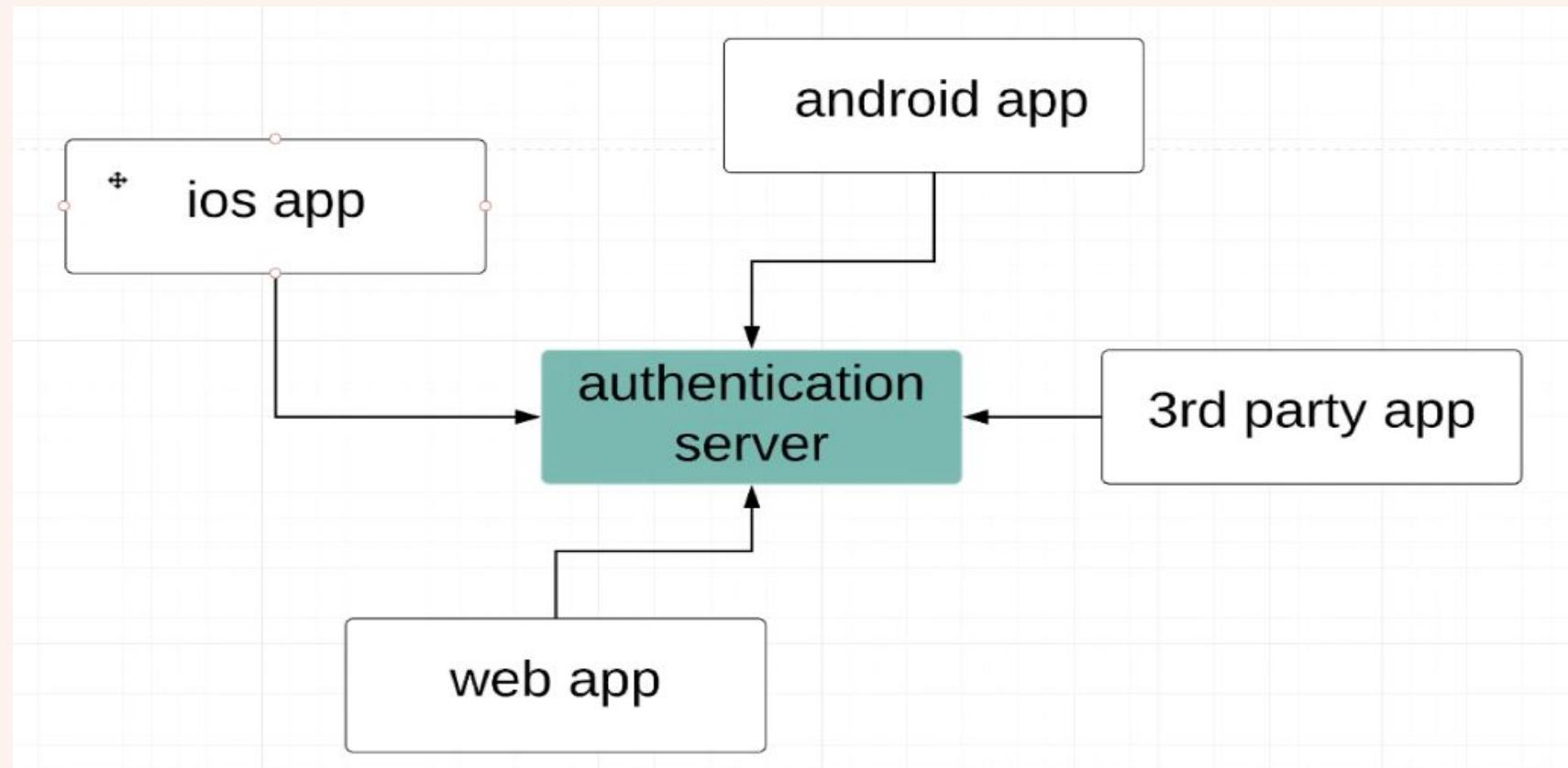


file: [spring-security-jwt-thymeleaf.zip](#)

# Arquitectura de Spring Security con JWT y Thymeleaf



# Arquitectura de Spring Security con JWT y Thymeleaf



ver el archivo: [JWT.pdf](#)

# Archivos a modificar

Tomaremos como base nuestro proyecto de **spring-security-user-registry**

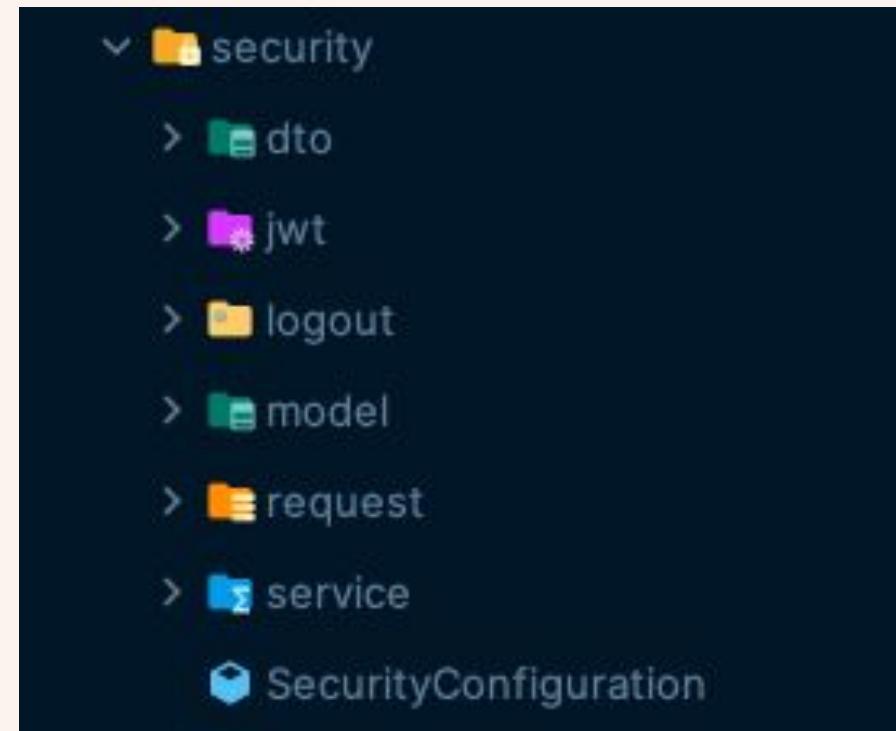
Vamos a modificar y agregar cierto código a varios archivos

Vamos a trabajar con un nuevo encarpetado



# Encarpetado nuevo

En este caso, solo vamos a afectar un encarpetado dentro del paquete ‘security’.



# POM

Como ya sabemos, hay que trabajar con nuestro POM.

La librería oficial de JWT para Java se encuentra en:

- <https://github.com/jwtk/jjwt>

*Siempre hay que tener cuidado! Esta librería está en constante cambio y son cambios fuertes!*

En este caso hay que agregar:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <!-- or jjwt-gson if Gson is preferred -->
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

# Modificación Service Negocio

Vamos a realizar una modificación a UserService.

Básicamente vamos a llamar a `SecurityContextHolder` para que nos regrese las credenciales del usuario logueado y veamos que las podemos obtener desde cualquier parte de nuestro sistema.

```
1 @Slf4j
2 @Service
3 public class UserService {
4     public String getText() {
5         Authentication auth = SecurityContextHolder.getContext().getAuthentication();
6         log.info("Auth {}", auth);
7         log.info("Credentials {}", auth.getCredentials());
8         return "User";
9     }
10 }
```

# DTO

A nivel de DTO, usaremos uno llamado CredentialsDTO

Nos va a servir para almacenar mayor información del token generado (payload)

```
1  @Builder
2  @Getter
3  @Setter
4  @ToString
5  public class CredentialsDTO {
6      private String sub;
7      private String aud;
8      private Long iat;
9      private Long exp;
10 }
```

# JWT

Aquí tenemos dos clases,  
**JWTAuthenticationFilter** y  
**JWTTokenProvider**

La primera, es un filtro. Nos va a validar cada que entra a Spring y validará si existe el token, *se obtiene por Cookies*, y en caso positivo, autentica al usuario.

- Una de las mejores maneras de trabajar con Thymeleaf y JWT es usando cookies
- [Ver URL](#)

```
1  @Override
2  protected void doFilterInternal(@NotNull HttpServletRequest request, @NotNull HttpServletResponse response,
3                                     @NotNull FilterChain filterChain)
4          throws ServletException, IOException {
5      String jwt = "";
6      if(request.getCookies() != null)
7          for(Cookie cookie: request.getCookies())
8              if(cookie.getName().equals("token"))
9                  jwt = cookie.getValue();
10     if(jwt == null || jwt.equals("")){
11         filterChain.doFilter(request, response);
12         return;
13     }
14     try {
15         if (tokenProvider.validateJwtToken(jwt)) {
16             Claims body = tokenProvider.getClaims(jwt);
17             var authorities = (List<Map<String, String>>) body.get("auth");
18             Set<SimpleGrantedAuthority> simpleGrantedAuthorities = authorities.stream()
19                 .map(m -> new SimpleGrantedAuthority(m.get("authority")))
20                 .collect(Collectors.toSet());
21             //String username = tokenProvider.getIssuer(jwt);
22             String username = tokenProvider.getFullName(jwt);
23             CredentialsDTO credentials = CredentialsDTO.builder()
24                 .sub(tokenProvider.getSubject(jwt)).aud(tokenProvider.getAudience(jwt))
25                 .exp(tokenProvider.getTokenExpiryFromJWT(jwt).getTime())
26                 .iat(tokenProvider.getTokenIatFromJWT(jwt).getTime())
27                 .build();
28             UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
29                 username, credentials, simpleGrantedAuthorities);
30             authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
31             SecurityContextHolder.getContext().setAuthentication(authentication);
32         }
33     } catch (Exception exception) {
34         log.error("Can NOT set user authentication -> Message: {}", exception.getMessage());
35     }
36     filterChain.doFilter(request, response);
37 }
```

# JWT

La clase **JWTTokenProvider** nos apoyará con la generación y validación de Tokens.

- Aquí leemos las variables del archivo YAML, generamos tokens, extraemos información de los tokens, validar tokens, en general.
- Esta clase es como una clase de apoyo, por eso se anota como **@Component**

```
1 public String generateJwtToken(Authentication authentication, UserInfoDTO user) {  
2     Claims claims = Jwts.claims().setSubject("UNAM").setIssuer(user.getEmail())  
3         .setAudience("JAVA");  
4     claims.put("principal", authentication.getPrincipal());  
5     claims.put("auth", authentication.getAuthorities().stream().map(s -> new SimpleGrantedAuthority(s.getAuthority()))  
6             .collect(Collectors.toList()));  
7     claims.put("issid", user.getId());  
8     claims.put("issname", user.getFirstName() + " " + user.getLastName());  
9     key = Keys.hmacShaKeyFor(secret.getBytes());  
10    return Jwts.builder()  
11        .setClaims(claims)  
12        .setIssuedAt(new Date())  
13        .setExpiration(new Date(System.currentTimeMillis() + jwtExpirationInMs * 1000L))  
14        .signWith(key, SignatureAlgorithm.HS512)  
15        .compact();  
16 }  
17  
18 public Claims getClaims(String token) {  
19     key = Keys.hmacShaKeyFor(secret.getBytes());  
20     return Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token).getBody();  
21 }
```

# Logout

En este paquete, tenemos una única clase CustomLogoutSuccessHandler.

Cada que hacemos logout, se ejecuta esta clase.

Lo importante es la eliminación de cookies!

```
...
1 @Override
2 public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {
3     log.info("Logout Handler");
4     Cookie[] cookies2 = request.getCookies();
5     for (Cookie cookie : cookies2) {
6         if (cookie.getName().equals("token")) {
7             cookie.setMaxAge(0);
8             response.addCookie(cookie);
9             log.info("Logout successfully");
10            response.sendRedirect(request.getContextPath());
11        }
12    }
13 }
```

# Templates Thymeleaf

También hay que venir aquí a realizar un par de modificaciones a:

- admin.html
- user.html

Básicamente, es modificar los <span> de sec:authentication

- Dicho tag es manejado directamente por thymeleaf-extras-springsecurity6

Lo que realiza estas nuevas acciones, es obtener el objeto autenticado de Spring, y mostrarlo en HTML.

```
<div class="main">
    <div th:replace="~{page-templates :: main-text}"></div>
    <div class="text">
        <h2 th:text="${text}"></h2>
        <div>
            <p>Principal: <span sec:authentication="principal">NOT FOUND</span></p>
            <p>Spring Roles: <span sec:authentication="authorities">NOT FOUND</span></p>
            <p>Credentials: <span sec:authentication="credentials">NOT FOUND</span></p>
        </div>
    </div>
    <div th:replace="~{page-templates :: footer}"></div>
</div>
```

# Templates Thymeleaf

Ésta modificación que realizamos es de suma importancia, ya que desde el **JWTAuthenticationFilter** es cuando creamos el **UsernamePasswordAuthenticationToken**. Se manda a llamar desde **SecurityConfiguration**, y ahí es donde tenemos nuestro AuthenticationManager. El **AuthenticationProvider** debe de regresar un objeto de tipo **UserDetails** para que funcione correctamente!

```
1 UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(  
2                                         username, credentials, simpleGrantedAuthorities);
```

# Templates Thymeleaf

```
1 @Bean
2 public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
3     throws Exception {
4     //your AuthenticationProvider must return UserDetails object
5     return new ProviderManager(authenticationProvider());
6 }
```

# Request

El paquete cuenta con 2 clases

- **JwtRequest:** nos sirve para cuando generamos el JWT, aquí se almacena el payload del token
- **LoginUserRequest:** nos sirve para almacenar la información proveniente del username y password que viene desde el login.

```
 1  @Getter
 2  @Setter
 3  @AllArgsConstructor
 4  @NoArgsConstructor
 5  @JsonPropertyOrder({
 6      "username",
 7      "password"
 8  })
 9  public class LoginUserRequest {
10      @JsonProperty("username")
11      private String username;
12      @JsonProperty("password")
13      private String password;
14 }
```

```
 1  @Getter
 2  @Setter
 3  @AllArgsConstructor
 4  @NoArgsConstructor
 5  public class JwtRequest {
 6      private String token;
 7      private String tokenType;
 8      private Long userId;
 9      private String userName;
10      private Long expiryDuration;
11      private Collection<? extends GrantedAuthority> authorities;
12
13      public JwtRequest(String token, Long userId, String userName, Long expiryDuration,
14                         Collection<? extends GrantedAuthority> authorities) {
15          this.token = token;
16          this.userId = userId;
17          this.userName = userName;
18          this.expiryDuration = expiryDuration;
19          this.authorities = authorities;
20          this.tokenType = "Bearer";
21      }
22 }
```

# Clase SecurityConfiguration

Modificamos el **securityFilterChain**

- Prácticamente deshabilitamos CSRF.

A veces no es necesario crear y mantener una HttpSession, por ejemplo, para conservar la autenticación en todas las solicitudes. Algunos mecanismos de autenticación como HTTP Basic no tienen estado y, por lo tanto, vuelven a autenticar al usuario en cada solicitud.

- Si no desea crear sesiones, puede utilizar SessionCreationPolicy.STATELESS
- Se deshabilita puesto que ahora esa sesión será sustituida por JWT

**Finalmente, agregamos el filtro antes desarrollado!**

```
1 @Bean
2 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
3     http
4         .authorizeHttpRequests((authorize) -> authorize
5             .requestMatchers("/css/**", "/favicon.ico", "/**", "/index").permitAll()
6             .requestMatchers("/user").hasAnyRole("USER")
7             .requestMatchers("/admin").hasAnyRole("ADMIN")
8             .requestMatchers("/api/**").permitAll()
9             .anyRequest().authenticated()
10        )
11        .formLogin(login -> login
12            .loginPage("/login")
13            .defaultSuccessUrl("/")
14            .successForwardUrl("/login_success_handler")
15            .permitAll())
16        .logout(logout -> logout
17            .logoutUrl("/doLogout")
18            .logoutSuccessUrl("/index")
19            .deleteCookies("JSESSIONID") //NEW Cookies to clear
20            .logoutSuccessHandler(customLogoutSuccessHandler)
21            .clearAuthentication(true)
22            .invalidateHttpSession(true))
23        .addFilterAfter(new JWTAuthenticationFilter(tokenProvider), UsernameAndPasswordAuthenticationFilter.class)
24        .csrf(AbstractHttpConfigurer::disable)
25        .cors(Customizer.withDefaults())
26        .sessionManagement(session -> session
27            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
28    ;
29    return http.build();
30 }
```

# Implementación de Filtro Personalizado

Podemos crear nuestros propios filtros implementando la interfaz Filter del paquete jakarta.servlet. Publique que necesitamos anular el método doFilter() para tener nuestra propia lógica personalizada. Este método acepta 3 parámetros, ServletRequest, ServletResponse y FilterChain.

- **ServletRequest:** representa la solicitud HTTP. Usamos el objeto ServletRequest para recuperar detalles sobre la solicitud del cliente.
- **ServletResponse:** representa la respuesta HTTP. Usamos el objeto ServletResponse para modificar la respuesta antes de enviarla de regreso al cliente o más adelante en la cadena de filtro.
- **FilterChain:** la cadena de filtros representa una colección de filtros con un orden definido en el que actúan. Usamos el objeto FilterChain para reenviar la solicitud al siguiente filtro de la cadena.

Puede agregar un nuevo filtro a la cadena Spring Security antes, después o en la posición de uno conocido. Cada posición del filtro es un índice (un número) y es posible que también se le llame "el orden".

A continuación se detallan los métodos disponibles para configurar un filtro personalizado en el flujo de Spring Security.

- **addFilterBefore(filter, class)** - agrega un filtro antes de la posición de la clase
- **addFilterAfter(filter, class)** - agrega un filtro después de la posición de la clase
- **addFilterAt(filter, class)** - agrega un filtro en la ubicación donde se encuentra la clase

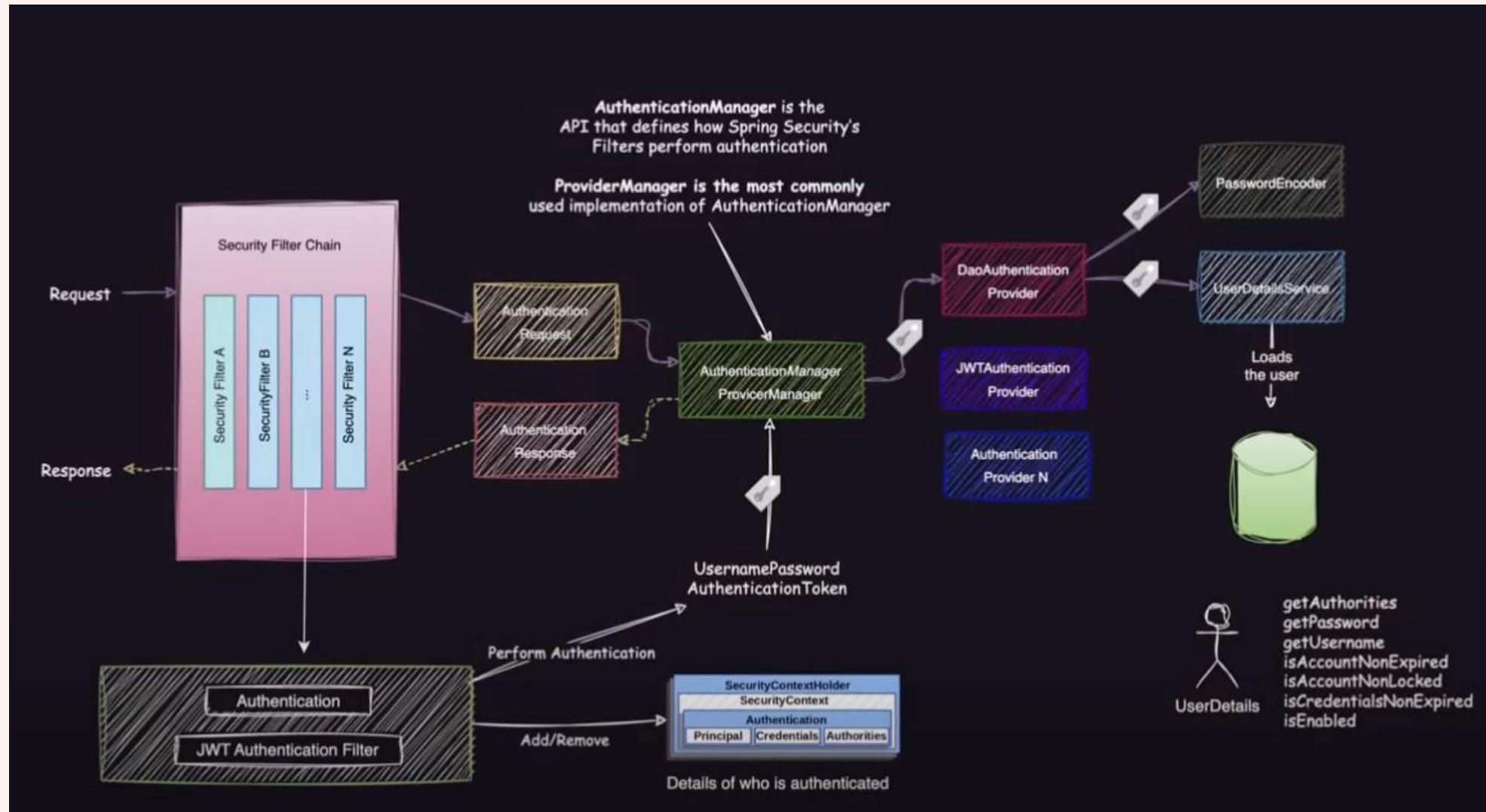
# 5. Spring Security y OAuth2 con JWT

5.2 Arquitectura de Spring Security con JWT y Rest

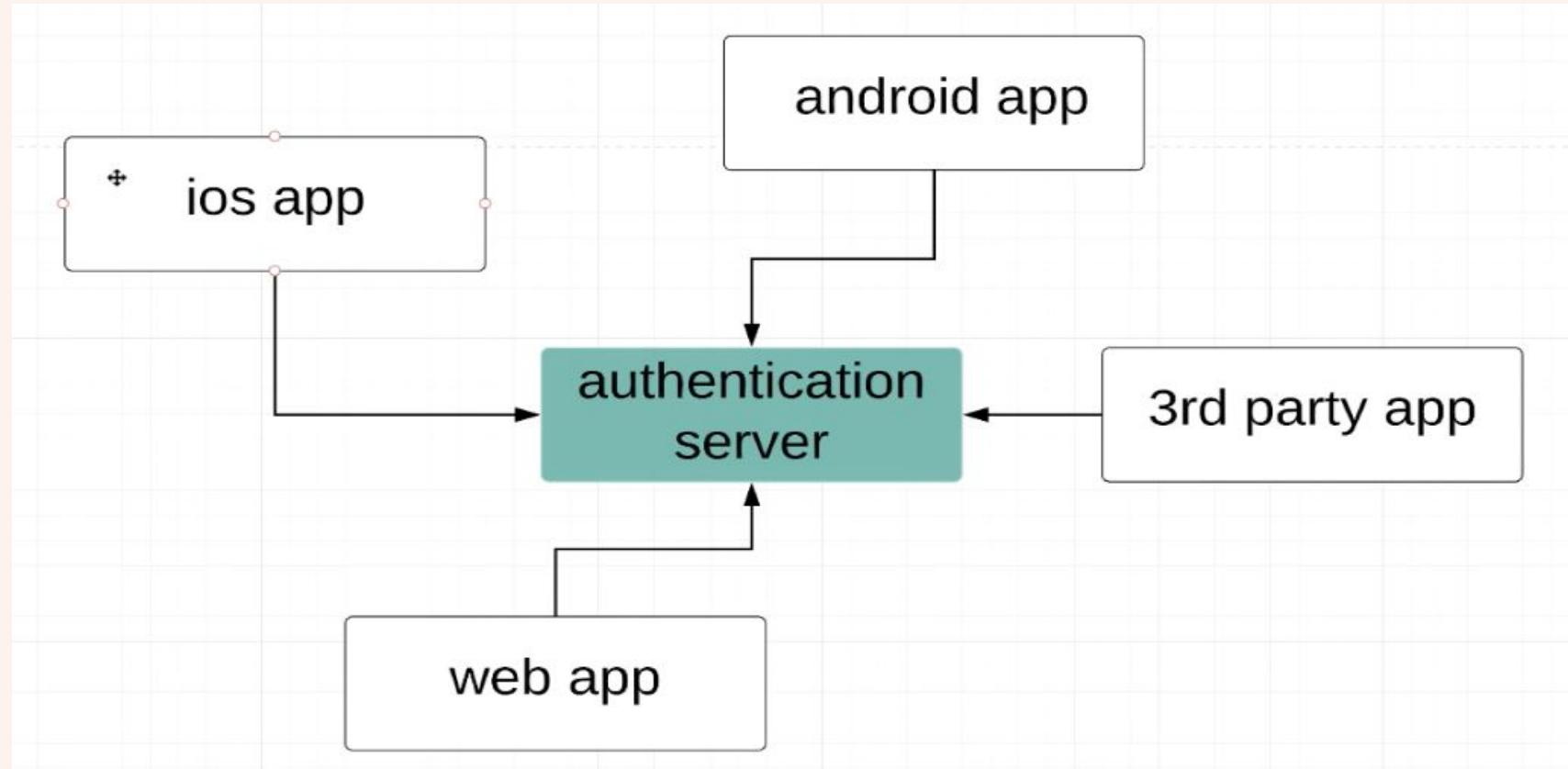


file: [spring-security-jwt-rest.zip](#)

# Arquitectura de Spring Security con JWT y Rest



# Arquitectura de Spring Security con JWT y Rest



ver el archivo: [JWT.pdf](#)

# Archivos a modificar

Tomaremos como base nuestro proyecto de **spring-security-jwt-thymeleaf**

Vamos a modificar y agregar cierto código a varios archivos

Vamos a respetar el mismo encarpetado, agregando nuevos paquetes y clases

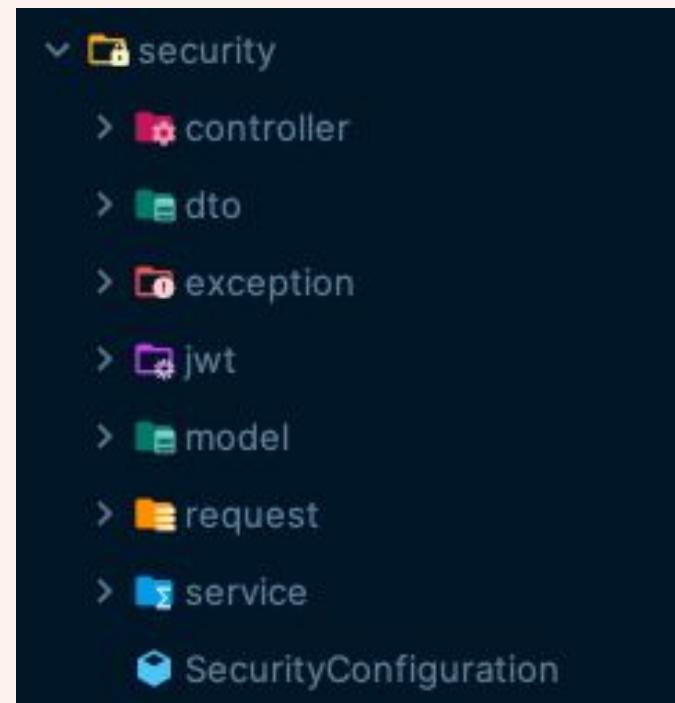
Vamos a limpiar el proyecto!



# Nuevo encarpetado

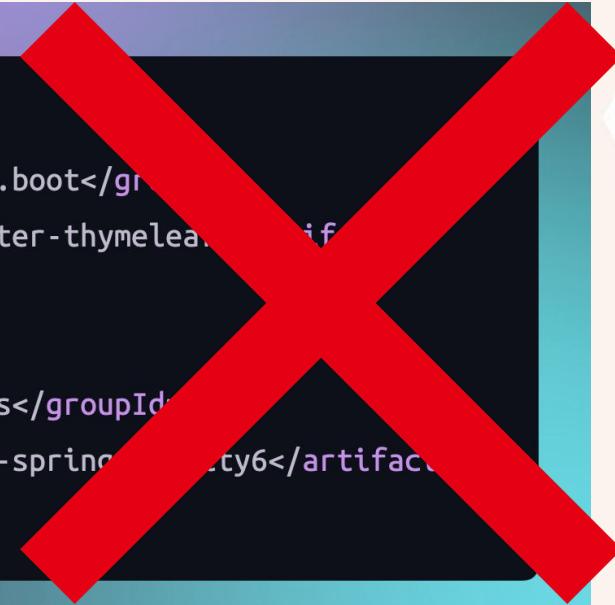
Es el mismo, solo agregamos dos paquetes:

- exception
- controller



# Limpieza POM

Vamos a eliminar las 2 dependencias de Thymeleaf antes creadas, esto para evitar tener tanta carga de librerías y liberar tiempo de compilación.



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-spring-mvc</artifactId>
</dependency>
```

# Eliminar resources

Dentro de la carpeta resources, tenemos dos carpetas, static y templates.

Hay que eliminar estos archivos, recordar, eso solo sirven para MVC, ahora, hay que adecuarnos a proyectos REST.

Dentro del paquete 'auth', tenemos un paquete 'controller'.

- Vamos a eliminar este paquete con todo y su contenido.
- Este contenido es específico para MVC
- El paquete auth solo se queda con la lógica para UserInfo y UserInfoRole

Hay que eliminar paquete 'logout', ya que esto solo sirve para MVC.

# Exception

Vamos a crear 2 clases para trabajar con las excepciones.

RestExceptionHandler

ExceptionResponse

Básicamente, **RestExceptionHandler** nos va a funcionar como un manejador global de excepciones para arquitectura REST.

Y la clase **ExceptionResponse** solo nos va a servir para poder manejar la excepción como una clase de respuesta hacia el cliente.

# JWT

Debemos agregar una clase llamada **JWTUsernameAndPasswordAuthenticationFilter**

- Ésta nos va a servir como un filtro, que sustituiría al filtro original llamado UsernameAndPasswordAuthenticationFilter.

El **JWTUsernameAndPasswordAuthenticationFilter** comprende lo siguiente:

1. JWTUsernameAndPasswordAuthenticationFilter extiende UsernameAndPasswordAuthenticationFilter y, de forma predeterminada, se activa cuando realiza una llamada "POST" a "/login". Esto se puede cambiar llamando a setFilterProcessesUrl(<PATH\_HERE>) en el constructor de JWTUsernameAndPasswordAuthenticationFilter.
2. No guarda la contraseña en el objeto de Autenticación simplemente porque no es necesaria, y es más seguro mantener la contraseña fuera de la memoria porque cualquiera con acceso al volcado de memoria puede recuperar la contraseña.

# JWT

```
● ● ●  
1 @Override  
2 protected void doFilterInternal(@NotNull HttpServletRequest request, @NotNull HttpServletResponse response,  
3                                     @NotNull FilterChain filterChain)  
4     throws ServletException, IOException {  
5     try {  
6         String jwt = getJwt(request);  
7         if (jwt != null && tokenProvider.validateJwtToken(jwt)) {  
8             Claims body = tokenProvider.getClaims(jwt);  
9             var authorities = (List<Map<String, String>>) body.get("auth");  
10            Set<SimpleGrantedAuthority> simpleGrantedAuthorities = authorities.stream()  
11                .map(m -> new SimpleGrantedAuthority(m.get("authority")))  
12                .collect(Collectors.toSet());  
13            //String username = tokenProvider.getIssuer(jwt);  
14            String username = tokenProvider.getFullName(jwt);  
15            CredentialsDTO credentials = CredentialsDTO.builder()  
16                .sub(tokenProvider.getSubject(jwt)).aud(tokenProvider.getAudience(jwt))  
17                .exp(tokenProvider.getTokenExpiryFromJWT(jwt).getTime())  
18                .iat(tokenProvider.getTokenIatFromJWT(jwt).getTime())  
19                .build();  
20            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(  
21                username, credentials, simpleGrantedAuthorities);  
22            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
23            SecurityContextHolder.getContext().setAuthentication(authentication);  
24        }  
25    } catch (Exception exception) {  
26        log.error("Can NOT set user authentication -> Message: {}", exception.getMessage());  
27    }  
28    filterChain.doFilter(request, response);  
29 }
```

El JWTAuthenticationFilter igual tiene cambios importantes!

Ya no usaremos Cookies, ahora vamos a trabajar de manera diferente.

# Controller

Creamos una clase llamada AuthController, donde nuestro endpoint principal es /login

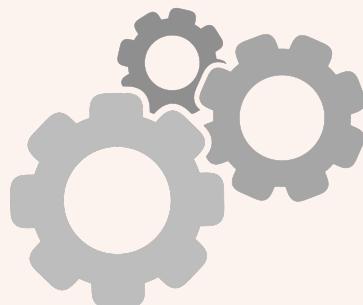
Este endpoint nos va a servir para poder realizar un login hacia el sistema.

prácticamente nos valida si existimos en la BD, nos genera un token y nos regresa el token si tenemos acceso junto con varios datos en general.

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/api/auth") // asignar un request mapping padre
4 @CrossOrigin("*") // Para trabajar con Angular, por ejemplo
5 public class AuthController {
6     private final AuthenticationManager authenticationManager;
7     private final UserInfoService userInfoService;
8     private final JWTTokenProvider jwtTokenProvider;
9
10    @Autowired
11    public AuthController(AuthenticationManager authenticationManager, UserInfoService userInfoService,
12                          JWTTokenProvider jwtTokenProvider) {
13        // Inyección por controlador
14    }
15
16    @PostMapping("/login")
17    public ResponseEntity<?> createAuthenticationToken(@RequestBody LoginUserRequest authenticationRequest,
18                                                       BindingResult bindingResult) throws Exception {
19        // Crea el token de autenticación
20    }
21
22    private Authentication authenticate(String username, String password) throws Exception {
23        // Manda a llamar a UsernamePasswordAuthenticationToken de la interfaz AuthenticationManager
24        // registrada en SecurityConfiguration, para trabajar desde el Filter y lograr autenticarse
25    }
```

# Controller

A nivel de controller



```
1 @PostMapping("/login")
2 public ResponseEntity<?> createAuthenticationToken(@RequestBody LoginUserRequest authenticationRequest,
3                                                               BindingResult bindingResult) throws Exception {
4     if(bindingResult.hasErrors())
5         return new ResponseEntity<>(ExceptionResponse.builder()
6                                         .errorStatus(HttpStatus.BAD_REQUEST)
7                                         .errorCode(HttpStatus.BAD_REQUEST.value())
8                                         .errorMessage(bindingResult.toString())
9                                         .timestamp(LocalDateTime.now())
10                                        .build(), HttpStatus.BAD_REQUEST);
11    UserInfoDTO user = userInfoService.findByUseEmail(authenticationRequest.getUsername());
12    if (user.getUseIdStatus() == 1) {
13        Authentication authentication = authenticate(authenticationRequest.getUsername(),
14                                                      authenticationRequest.getPassword());
15        SecurityContextHolder.getContext().setAuthentication(authentication);
16        log.info("authentication {}", authentication);
17        String jwtToken = jwtTokenProvider.generateJwtToken(authentication, user);
18        JwtRequest jwtRequest = new JwtRequest(jwtToken, user.getUserId(), user.getEmail(),
19                                                jwtTokenProvider.getExpiryDuration(), authentication.getAuthorities());
20        return new ResponseEntity<>(jwtRequest, HttpStatus.OK);
21    }
22    return new ResponseEntity<>(ExceptionResponse.builder()
23                                .errorStatus(HttpStatus.BAD_REQUEST)
24                                .errorCode(HttpStatus.BAD_REQUEST.value())
25                                .errorMessage("User has been deactivated/locked !!")
26                                .timestamp(LocalDateTime.now())
27                                .build(), HttpStatus.BAD_REQUEST);
28 }
```

# SecurityConfiguration

Ahora, lo importante es limpiar el **securityFilterChain** para poderlo trabajar adecuadamente.

Y lo importante, agregar los filtros desarrollados anteriormente.

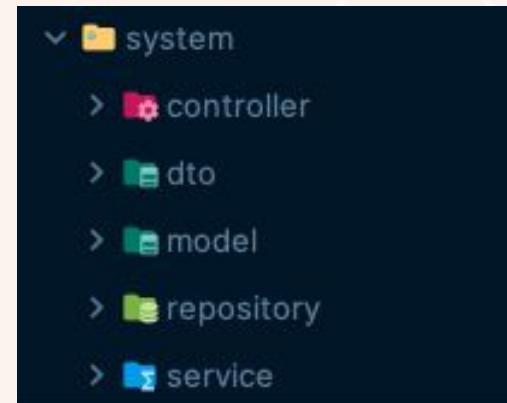
```
1 @Bean
2 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
3     http
4         .authorizeHttpRequests((authorize) -> authorize
5             .requestMatchers("/user").hasAnyRole("USER")
6             .requestMatchers("/admin").hasAnyRole("ADMIN")
7             .requestMatchers("/api/**").permitAll()
8             .anyRequest().authenticated()
9         )
10        .addFilter(new JWTUsernameAndPasswordAuthenticationFilter())
11        .addFilterAfter(new JWTAuthenticationFilter(tokenProvider),JWTUsernameAndPasswordAuthenticationFilter.class)
12        .csrf(AbstractHttpConfigurer::disable)
13        .cors(Customizer.withDefaults())
14        .sessionManagement(session -> session
15            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
16    ;
17    return http.build();
18 }
```

# Paquete 'system'

Recordemos el módulo de hibernate, donde aprendimos a desarrollar entidades y modelos para una agenda.

- Tomaremos una de esas tablas y la vamos a mapear en nuestro proyecto, para que nos sirva como ejemplo.
- Tomaremos el archivo \*.sql que se encuentra dentro de la carpeta resources de nuestro proyecto y lo ejecutaremos en la BD.

Luego, procedemos a desarrollar en el paquete 'system' el encarpelado base para nuestras clases



# Paquete 'system'

Prácticamente revisamos el código y es trabajar con un catálogo de la base de datos que contiene 3 campos. Hay que desarrollar

- Modelo
- Repositorio
- Servicio con su implementación (SOLID)
- Controlador Rest

El controlador nos va a funcionar para accesarlo desde Postman, mandando JWT por header.

```
1 @GetMapping("/get-contact-types")
2 public ResponseEntity<?> getContactTypes(HttpServletRequest request) {
3     List<CatContactTypeDTO> list = catContactTypeService.findAll();
4     return new ResponseEntity<>(list, HttpStatus.OK);
5 }
```

# Ejecución!

Abrimos postman y creamos dos endpoints:

POST> <http://localhost:8090/api/auth/login>

Este debe ir con body

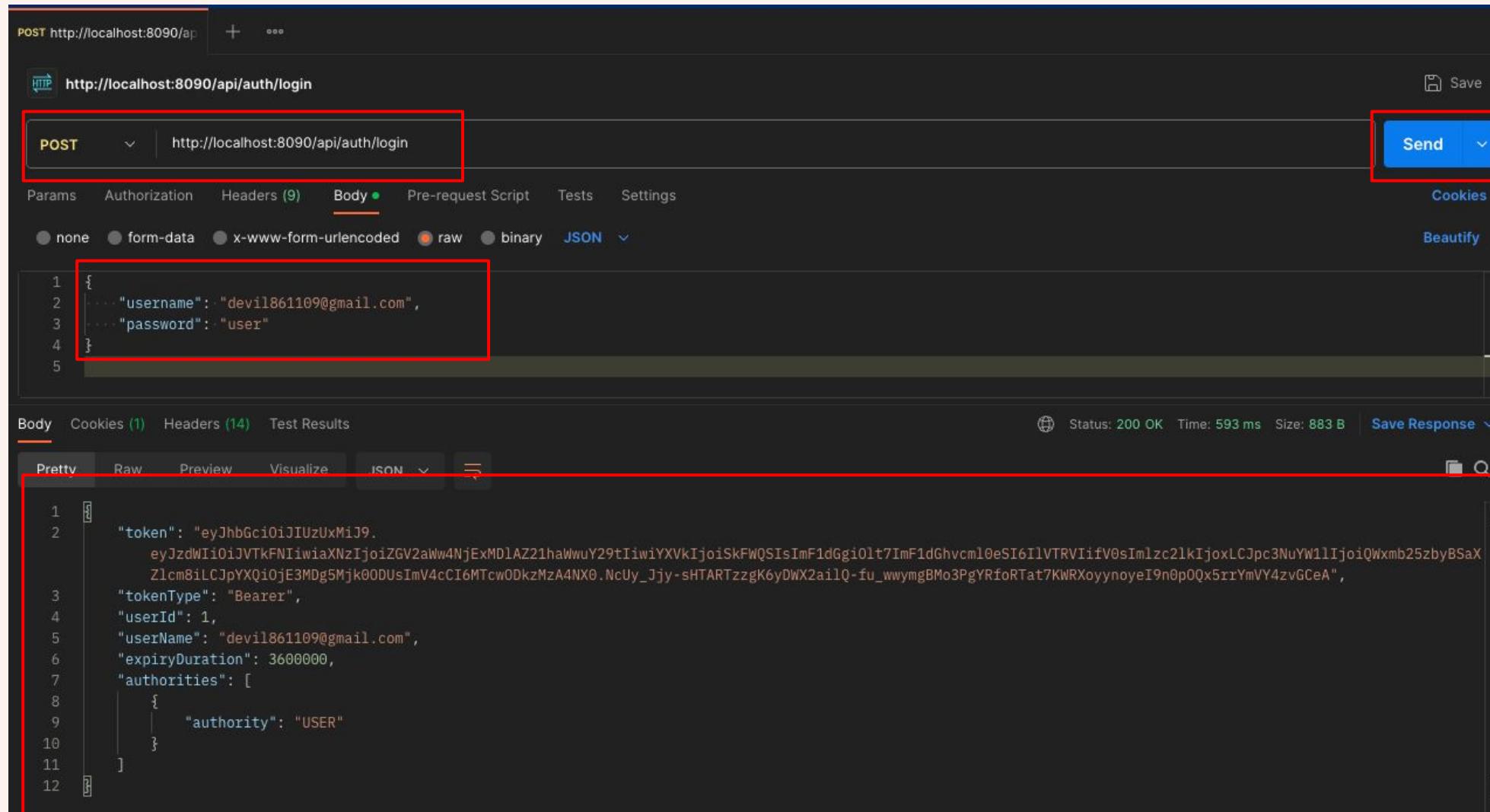


The screenshot shows the 'raw' tab of the Postman interface. The content type is set to 'JSON'. The body contains the following JSON payload:

```
1 {
2   "username": "devil861109@gmail.com",
3   "password": "user"
4 }
```

GET> <http://localhost:8090/v1/contact-type/get-contact-types>

# Ejecución!



The screenshot shows a POST request to `http://localhost:8090/api/auth/login`. The request body is JSON:

```
1 {
2   "username": "devil861109@gmail.com",
3   "password": "user"
4 }
```

The response is a 200 OK status with the following JSON data:

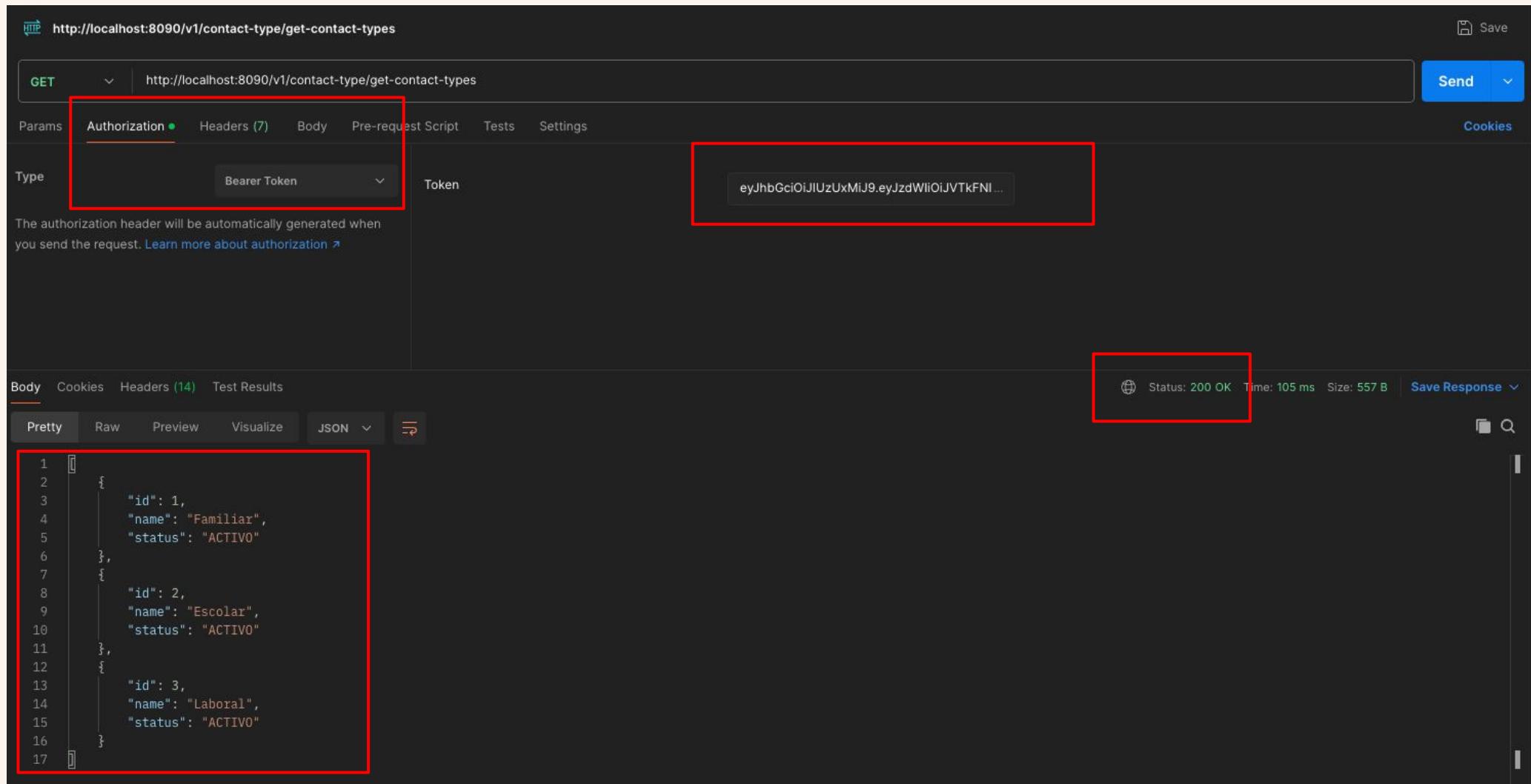
```
1 {
2   "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJVtkFNIIwiaXNzIjoiZGV2aWw4NjExMDlAZ21haWwuY29tIiwiYXVkIjoiSkFWQSIiMf1dGgi0lt7ImF1dGhvcmI0eSI6IlVTRVIifV0sImIzc2lkIjoxLCJpc3NuYW1lIjoiQWxmb25zbyBSaXZlc8iLCJpYXQiOjE3MDg5Mjk0ODUsImV4cCI6MTcwODkzMzA4NX0.NcUy_Jjy-sHTARTzzgK6yDWX2ailQ-fu_wwymgBMo3PgYRfoRTat7KWRXoyynoyeI9n0p0Qx5rrYmVY4zvGCeA",
3   "tokenType": "Bearer",
4   "userId": 1,
5   "userName": "devil861109@gmail.com",
6   "expiryDuration": 3600000,
7   "authorities": [
8     {
9       "authority": "USER"
10    }
11  ]
12 }
```

# Ejecución!

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET (highlighted by a red box)
- URL:** http://localhost:8090/v1/contact-type/get-contact-types (highlighted by a red box)
- Authorization:** No Auth (highlighted by a red box)
- Status:** Status: 403 Forbidden (highlighted by a red box)
- Response Time:** 115 ms
- Response Size:** 389 B

# Ejecución!

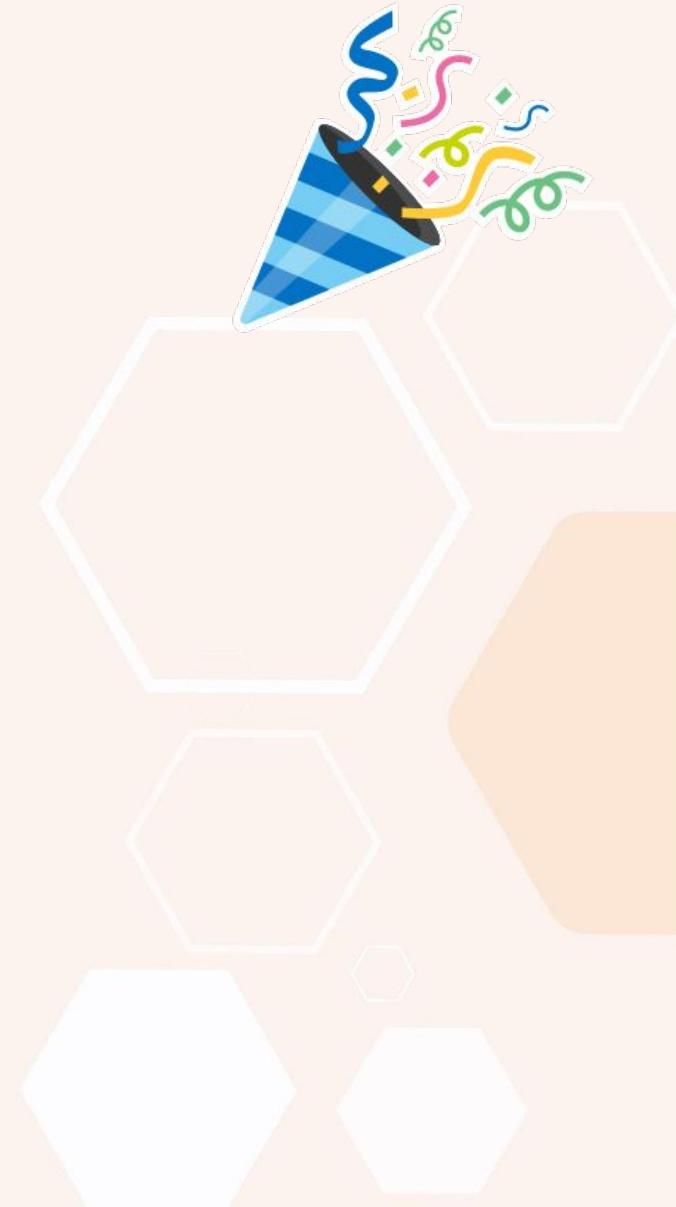
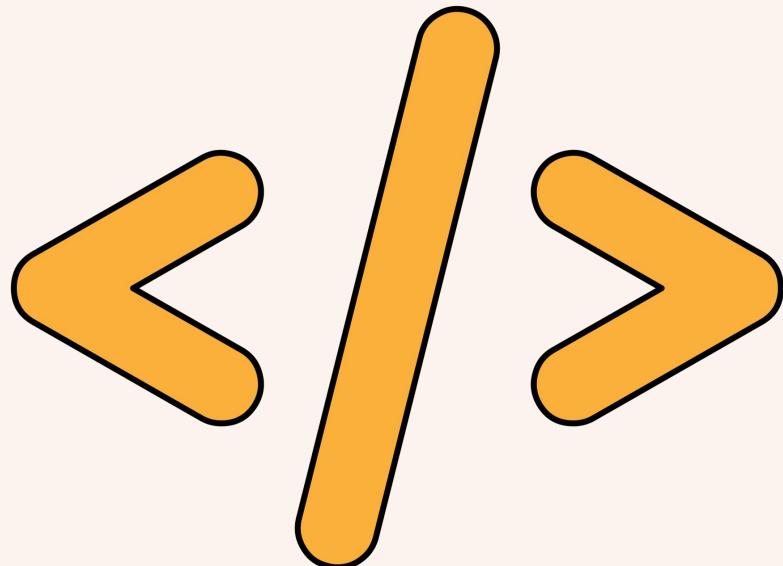


The screenshot shows a POSTMAN interface with the following details:

- URL:** `http://localhost:8090/v1/contact-type/get-contact-types`
- Method:** GET
- Authorization:** Bearer Token (highlighted by a red box)
- Token:** eyJhbGciOiJIUzUxMiJ9.eyJzdWlIiOiJVTkFNI... (highlighted by a red box)
- Status:** 200 OK (highlighted by a red box)
- Response Body (Pretty JSON):**

```
1 [ ]  
2 {  
3   "id": 1,  
4   "name": "Familiar",  
5   "status": "ACTIVO"  
6 },  
7 {  
8   "id": 2,  
9   "name": "Escolar",  
10  "status": "ACTIVO"  
11 },  
12 {  
13   "id": 3,  
14   "name": "Laboral",  
15   "status": "ACTIVO"  
16 }]  
17 ]
```

# Proyecto Final



# 5. Spring Security y OAuth2 con JWT

5.3 Spring Security con OAuth2.0 y GitHub



file: [spring-security-oauth2-github.zip](#)

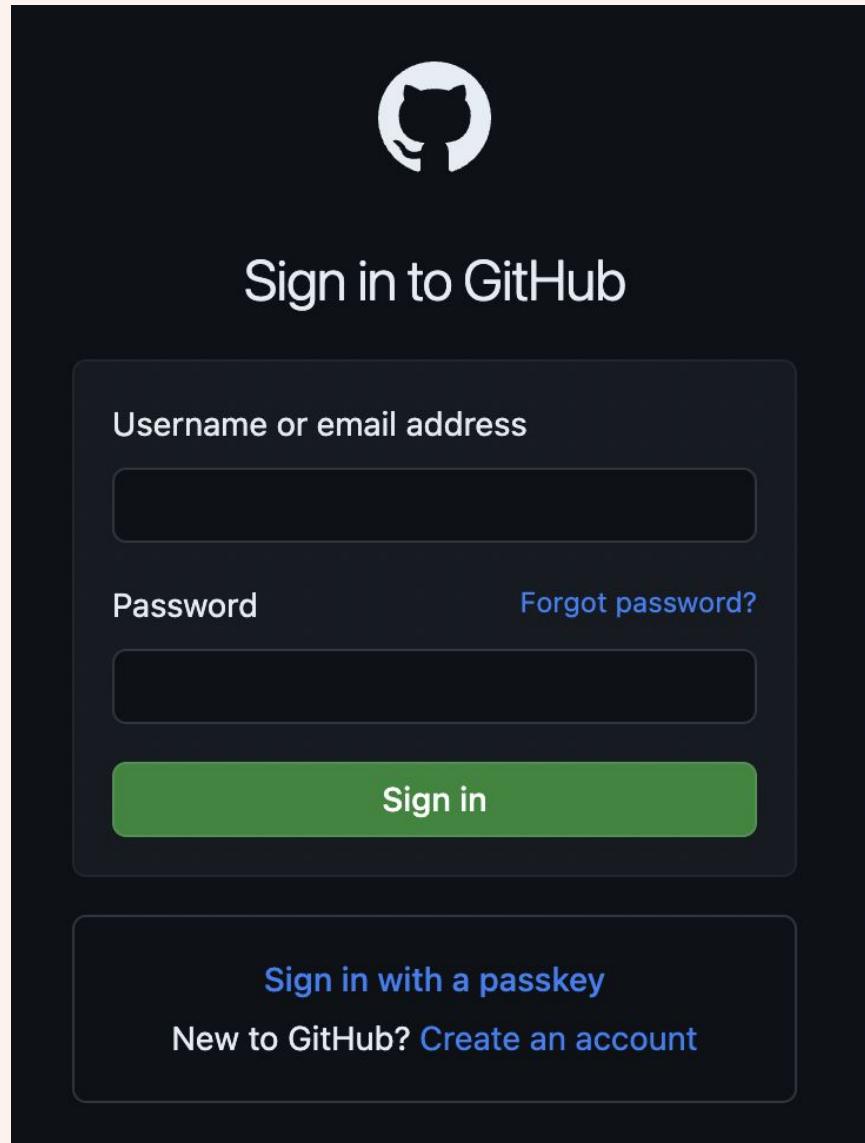
# OAuth2 con Github

Para el desarrollo de este ejemplo, vamos a trabajar con GitHub.

- Así que, si no tienes una cuenta, vela creando!

En este escenario, la aplicación Spring Boot a desarrollarse, actuará tanto como aplicación cliente como servidor de recursos, mientras que el servidor de autorización será GitHub.

Hay que loguearnos a GitHub.

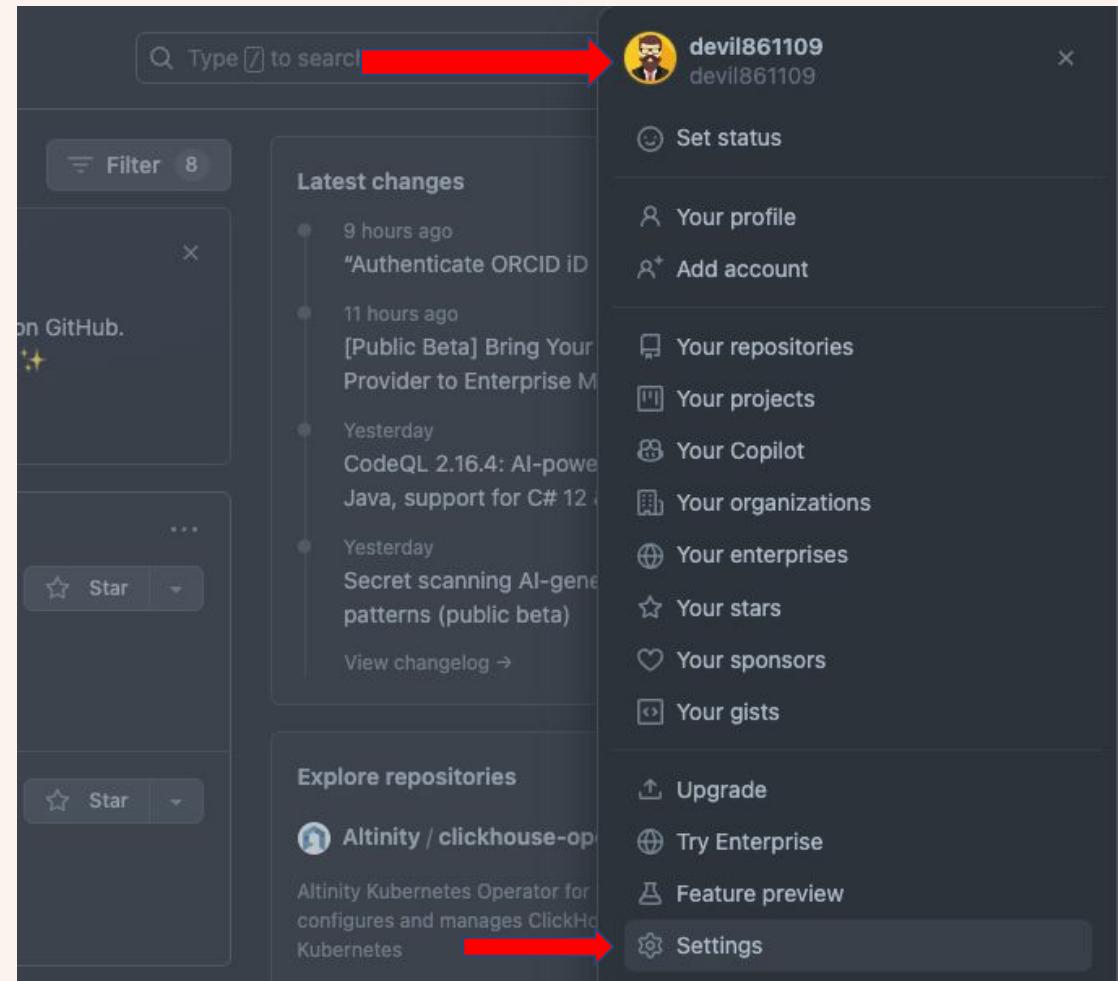
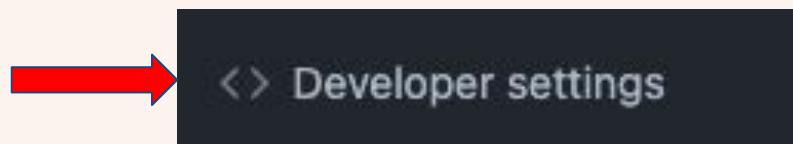


# OAuth2 con Github

Al autenticarnos en GitHub, hay que ir al extremo derecho superior, darle click en nuestro user, e ir hacia Settings.

Ahora, del lado izquierdo nos aparece nuestro menú.

Hay que ir hasta abajo y darle click en Developer Settings.



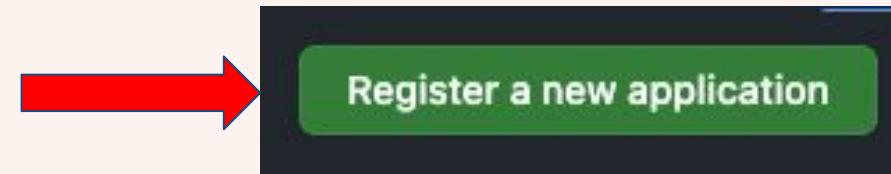
# OAuth2 con Github

A continuación nos aparece otro menú pequeño del lado izquierdo.

Aquí debemos seleccionar Oauth Apps



Y registramos una nueva aplicación



# OAuth2 con Github

Aquí, mi servidor de aplicaciones GitHub está intentando recopilar todos los detalles sobre mi aplicación cliente.

Entonces el nombre que quiero darle a mi aplicación es "AuthGitHubSpringBoot" y la URL de la página de inicio que le daré es <http://localhost:8080>.

- Esta es la página de inicio de mi aplicación cliente.
- En el mundo real, por supuesto, hay que darle un dominio adecuado, pero como estamos desarrollando una aplicación Spring Boot dentro de nuestro sistema local, no hay problema

Register a new OAuth application

**Application name \***  
AuthGitHubSpringBoot  
Something users will recognize and trust.

**Homepage URL \***  
<http://localhost:8080>  
The full URL to your application homepage.

**Application description**  
AuthGitHubSpringBoot  
This is displayed to all users of your application.

**Authorization callback URL \***  
<http://localhost:8080>  
Your application's callback URL. Read our [OAuth documentation](#) for more information.

**Enable Device Flow**  
Allow this OAuth App to authorize users via the Device Flow.  
Read the [Device Flow documentation](#) for more information.

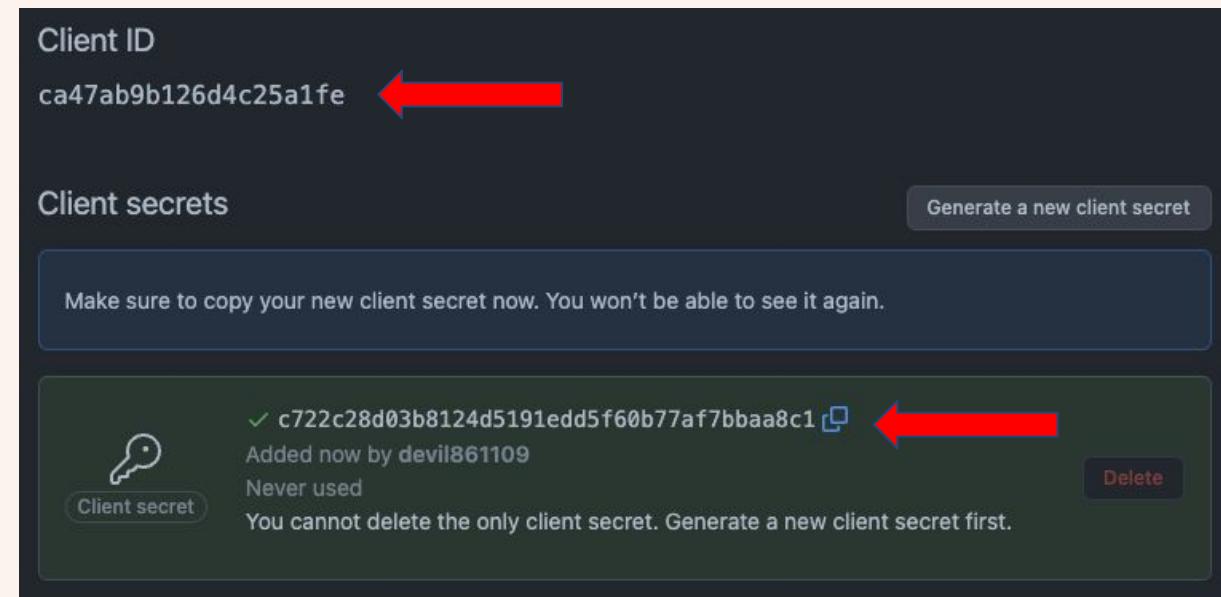
**Register application** **Cancel**

# OAuth2 con Github

Si todo lo realizamos correctamente, nos va a llevar a mostrarnos un Client ID y un Client Secrets.

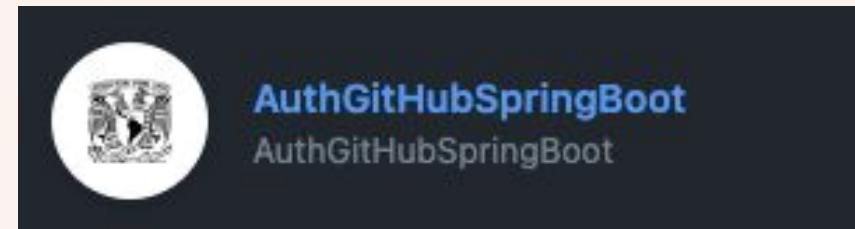
Aún no tenemos ningún secrets, por lo que procedemos a generar uno

- Hay que guardar el secrets generado!



# OAuth2 con Github

Con eso configuramos nuestro servidor en GitHub

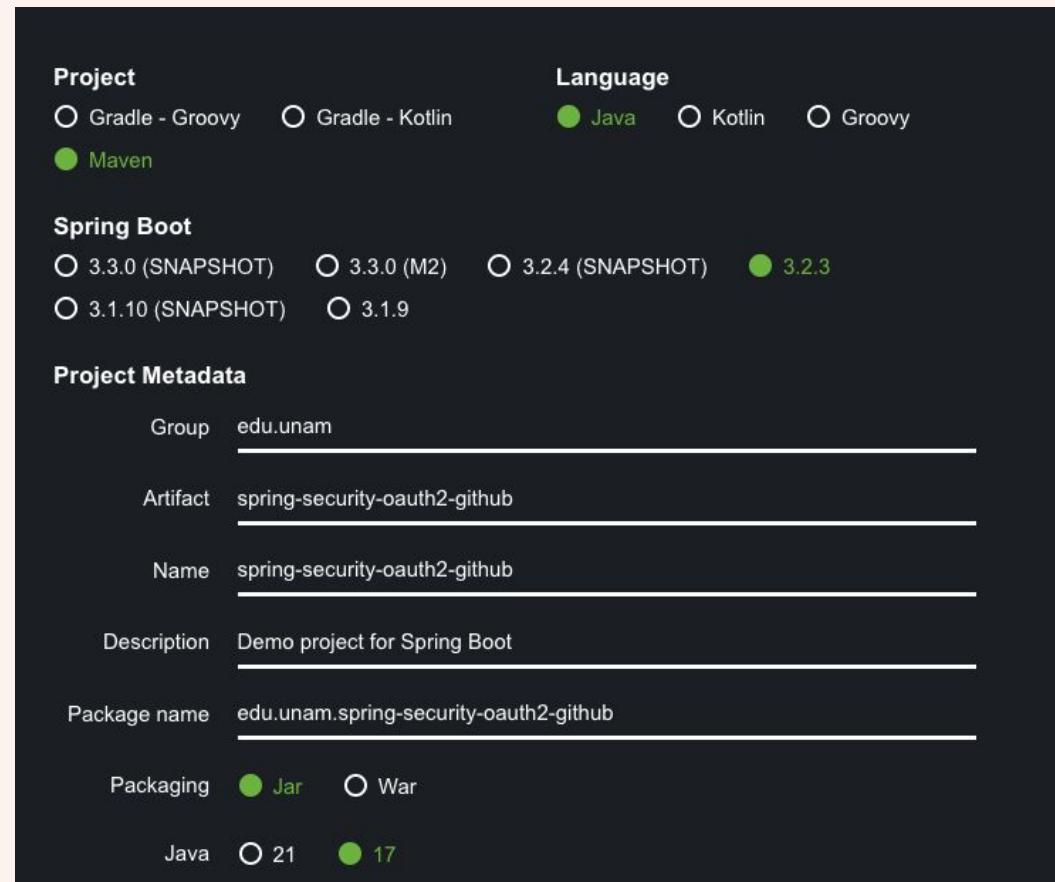


Ahora hay que trabajar con nuestra aplicación en Spring Boot.

# Spring Boot App

Empecemos desde cero.

Vamos a spring initializer y creamos un nuevo proyecto



# POM

Agregamos nuestras librerías

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
</dependencies>
```

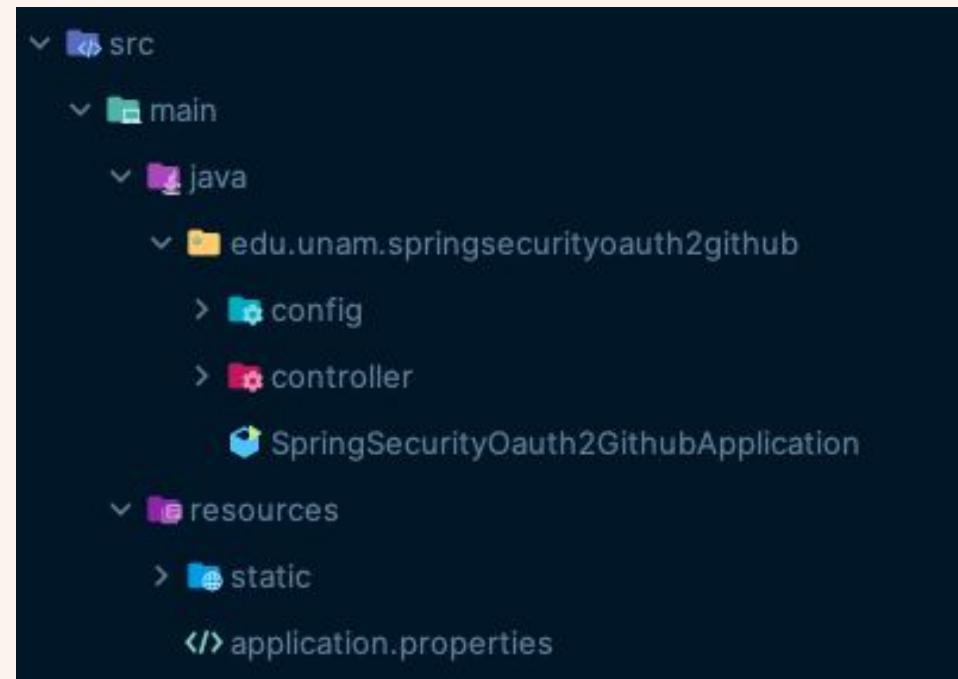
# application.properties

De igual manera, hay que agregar aca el Client ID y el Client Secrets que cada uno de ustedes generó

```
spring.security.oauth2.client.registration.github.client-id=ca47ab9b126d4c25a1f  
spring.security.oauth2.client.registration.github.client-secret=c722c28d03b8124d5191edd5f60b77af7bbaa8c
```

# Encarpetado

Hay que seguir este encarpetado



# Resources

También, hay que crear una carpeta llamada static

Dentro meteremos un HTML simple llamado secure.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sample OAUTH2 App</title>
</head>
<body>
    <h1>Successfully logged in using GitHub & OAUTH2 !!</h1>
</body>
</html>
```

# Paquete 'config'

Creamos una clase llamada SecurityConfiguration donde tendremos el siguiente código.

Es lo mismo que ya venimos manejando, pero ahora con oauth2Login

```
1 @Configuration
2 public class SecurityConfiguration {
3     @Bean
4     SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
5         http.authorizeHttpRequests((requests)->requests.anyRequest().authenticated())
6             .oauth2Login(Customizer.withDefaults());
7         return http.build();
8     }
9 }
```

# Paquete 'controller'

De igual manera, aca tenemos un endpoint programado que va hacia `secure.html`

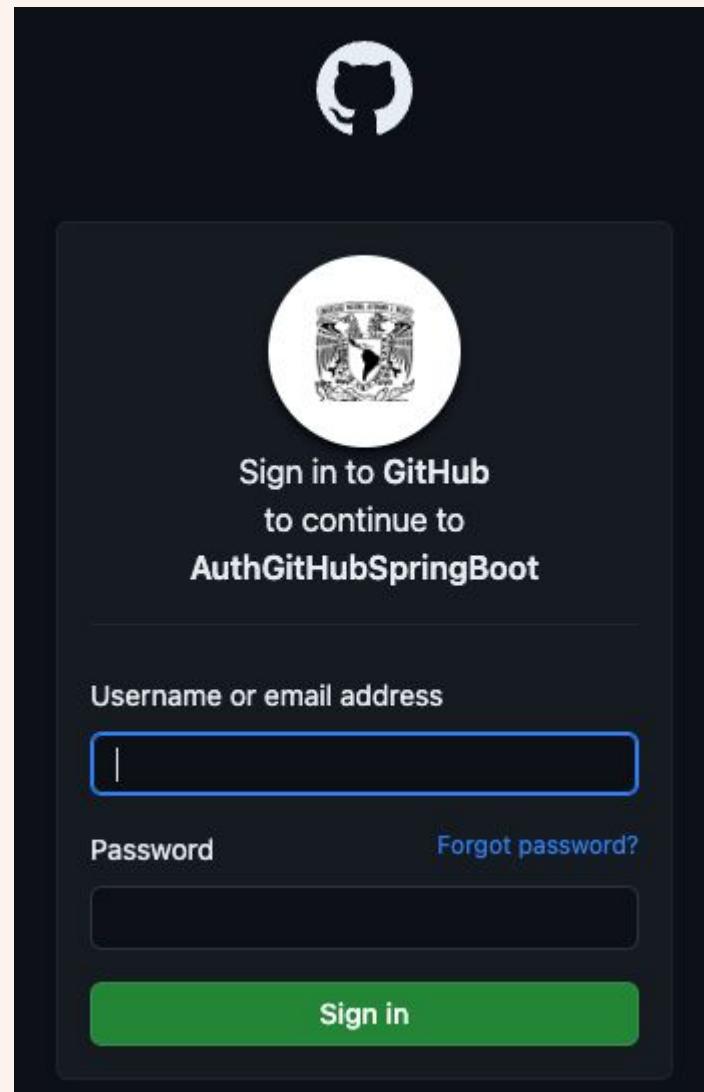
```
1 @Controller
2 public class SecureController {
3     @GetMapping("/")
4     public String main(OAuth2AuthenticationToken token) {
5         System.out.println(token.getPrincipal());
6         return "secure.html";
7     }
8 }
```

# Paquete 'controller'

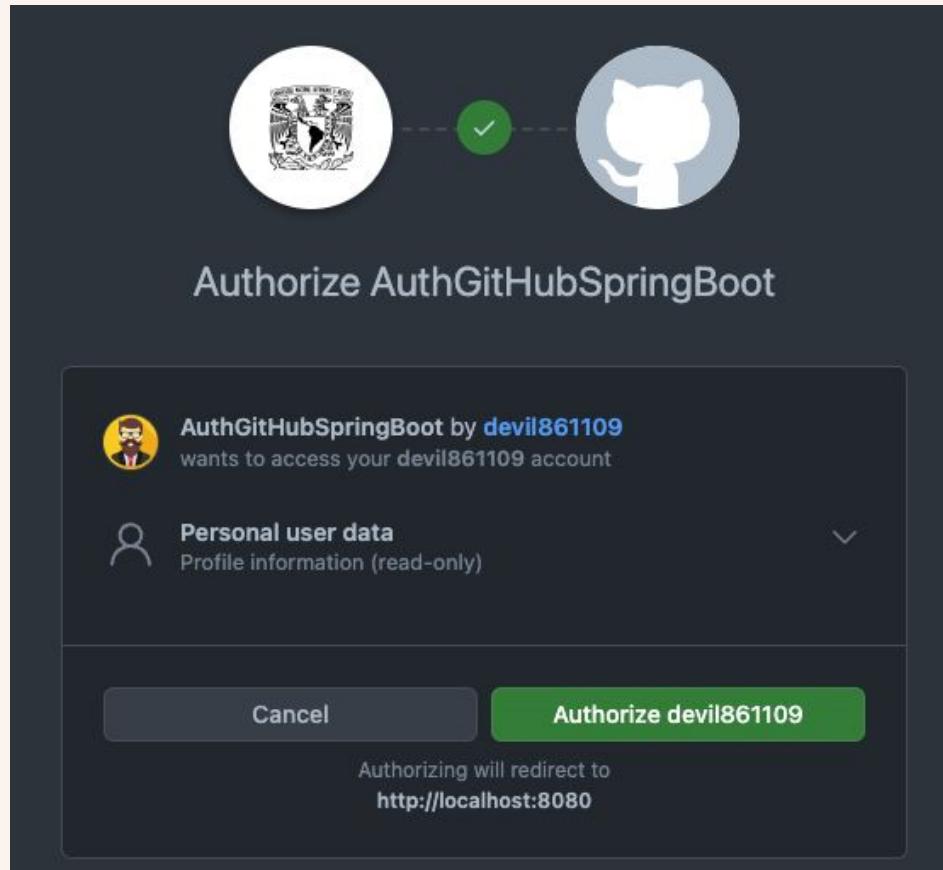
Cuando ejecutamos nuestro proyecto ya creado, recordemos que entraría hacia:

- <http://localhost:8080/>
- Puerto 8080 por defecto, el mismo que configuramos en GitHub.

Si accedemos por URL de navegador (modo incógnito), nos da la siguiente pantalla



# Paquete 'controller'



Al poner nuestras credenciales, nos da la siguiente pantalla.

Hay que darle click en Autorizar, y nos va a redirigir a nuestro proyecto en Spring Boot con la siguiente leyenda:

**Successfully logged in using GitHub & OAUTH2 !!**

# Conclusiones

Este pequeño proyecto nos sirve para entender cómo trabaja OAuth2.

Sin embargo, hay muchas limitantes

- No es suficiente para aplicaciones reales
- No podemos crear usuarios en GitHub
- No podemos crear roles o authorities
- Para tener el control total, hay que utilizar otros Servidores de OAuth2
  - Como por ejemplo, Keycloak

# 5. Spring Security y OAuth2 con JWT

5.4 Spring Security con OAuth2.0 y KeyCloak



file: [spring-security-oauth2-keycloak.zip](#)

# Trabajando con KeyCloak

Hay que ir a la página <https://www.keycloak.org/> y descargar la última versión.

Para este caso, yo trabajo con la 24.0.1

The screenshot shows the 'Downloads' section for Keycloak version 24.0.1. A large red arrow points to the 'ZIP (sha1)' download link at the bottom right of the table row. The table has three columns: 'Keycloak', 'Distribution powered by Quarkus', and 'ZIP (sha1)'. The 'Keycloak' and 'Distribution' columns are empty, while the 'ZIP (sha1)' column contains a download icon and the text 'ZIP (sha1)'.

Keycloak	Distribution powered by Quarkus	ZIP (sha1)
		ZIP (sha1)

# Trabajando con KeyCloak

Podemos leer la documentación que viene para entender qué hacer, pero esto es trabajo del DevOps. Nosotros aprenderemos de manera rápida a configurar KeyCloak, sin embargo, esto no es trabajo de desarrollo.

## Documentation 24.0.1

### Guides

<a href="#">Release Notes</a>	
<a href="#">Getting Started</a>	 <a href="#">How to get started with Keycloak</a>
<a href="#">Server Installation and Configuration</a>	

### Getting started

[OpenJDK](#)  
[Get started with Keycloak on bare metal](#)



DDTIC\_DSJ\_PLI\_2024

# Trabajando con KeyCloak

Si quieren cambiarlo de puerto, intenten con --http-port 8180

## Start Keycloak

1. From a terminal, open the `keycloak-24.0.1` directory.
2. Enter the following command:

- On Linux, run:

```
bin/kc.sh start-dev
```

- On Windows, run:

```
bin\kc.bat start-dev
```



# Trabajando con KeyCloak

Hecho esto, ahora tenemos KeyCloak accesible desde cualquier navegador de internet

## Create an admin user

Keycloak has no default admin user. You need to create an admin user before you can start Keycloak.

1. Open <http://localhost:8080/>. 
2. Fill in the form with your preferred username and password.

## Log in to the Admin Console

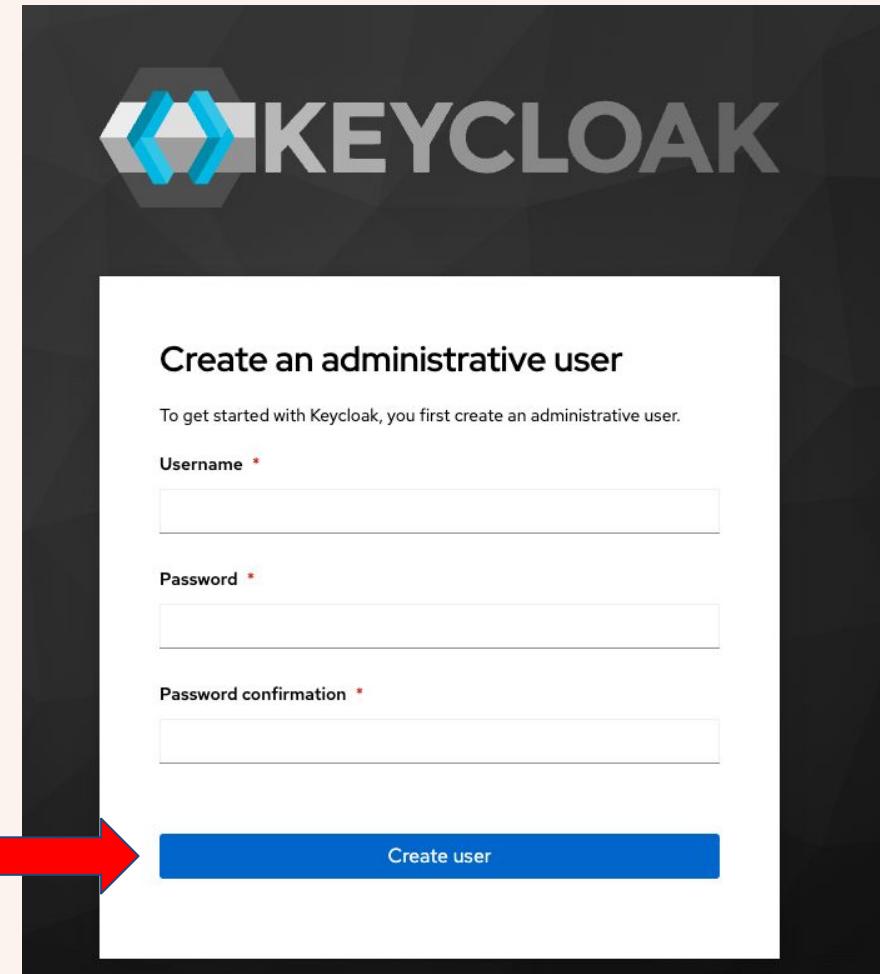
1. Go to the [Keycloak Admin Console](#).
2. Log in with the username and password you created earlier.

# Trabajando con KeyCloak

Al entrar la primera vez, nos va a pedir crear nuestro usuario admin.

Después de crear el usuario, no olviden user y password

Una vez hecho esto, hay que entrar al Administrator Console



# Realms

Una vez dentro, vemos del lado izquierdo que tenemos configurado el Realm Master Keycloak. Un realm en Keycloak es un dominio de administración, esencialmente un espacio aislado donde se definen todos los usuarios, roles, clientes y otros aspectos relacionados con la seguridad.

Esto nos funciona puesto que podríamos tener varios ambientes configurados:

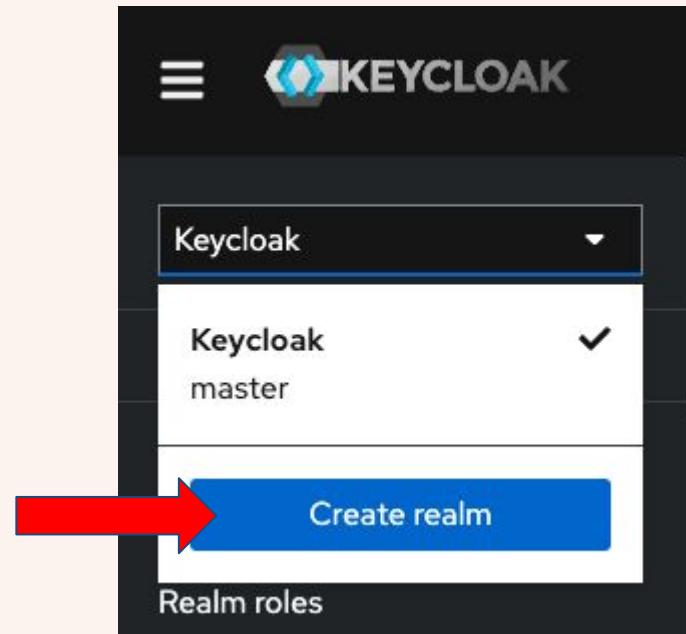
Dev

Prod

Testing

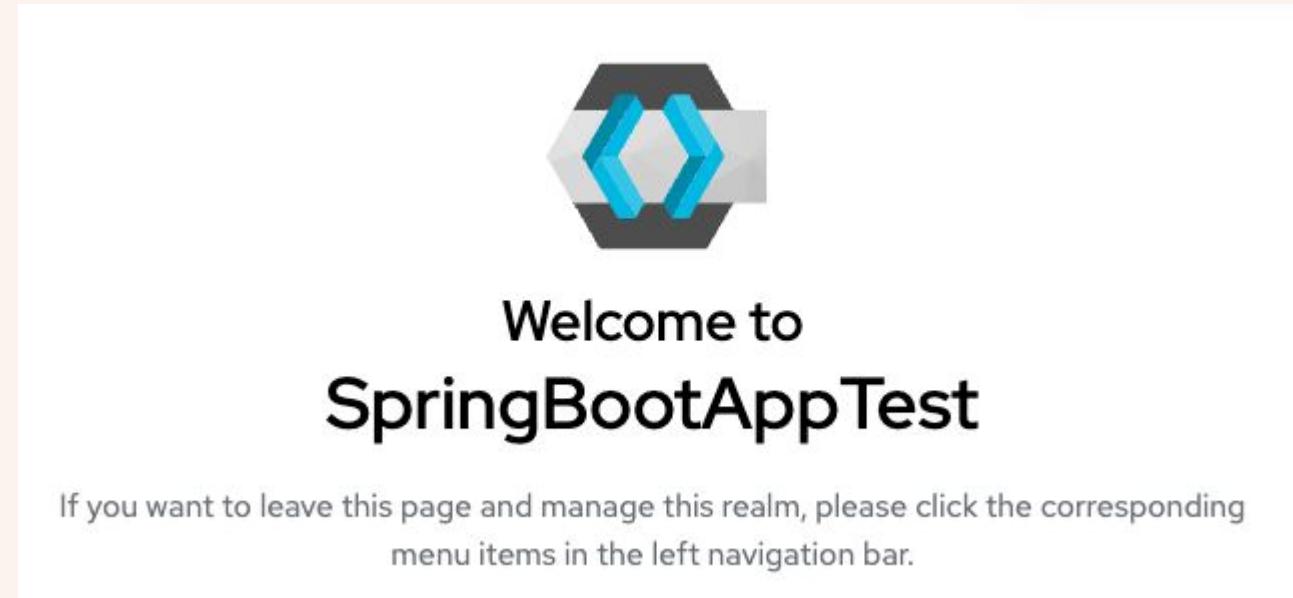
...

Procedemos a crear el nuestro.



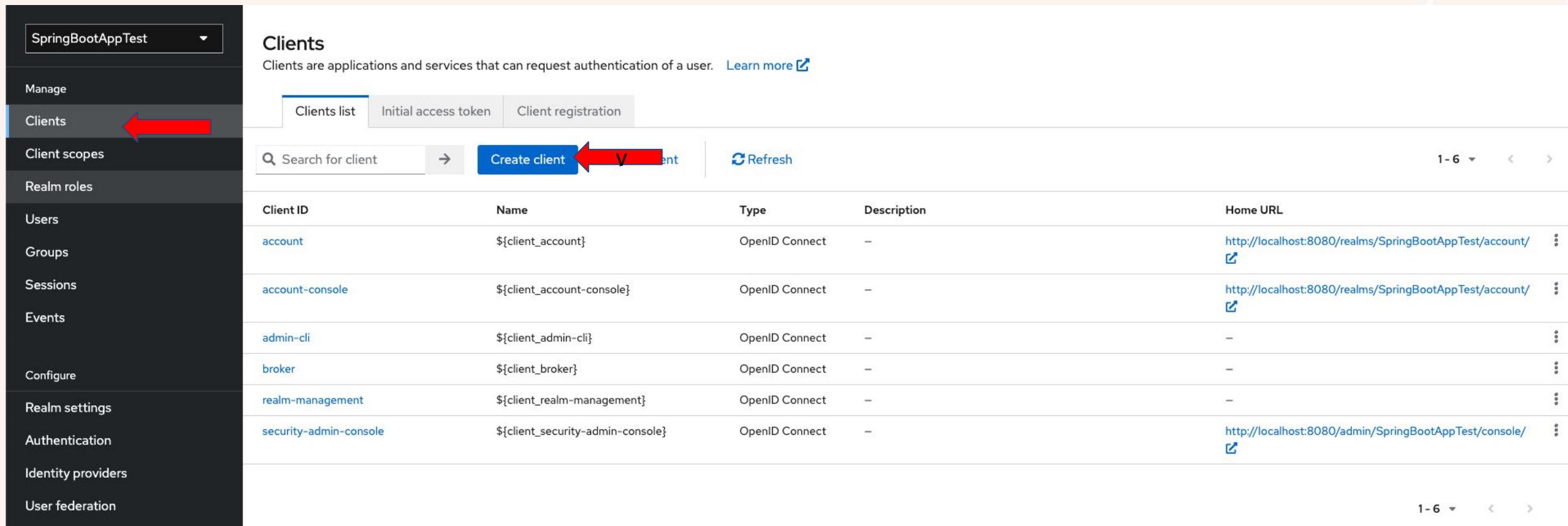
# Realms

Ya creado nuestro realm, nos pondremos a configurar lo básico



# Clients

Ahora seleccionamos la tab de clients y vemos la sig configuración ya existente y crearemos un nuevo cliente



The screenshot shows the Keycloak administration interface for a realm named "SpringBootAppTest". The left sidebar has a dark theme with white text and includes tabs for Manage, Client scopes, Realm roles, Users, Groups, Sessions, Events, Configure, Realm settings, Authentication, Identity providers, and User federation. The "Clients" tab is highlighted with a blue underline and has a red arrow pointing to it from the left.

The main content area is titled "Clients" and contains the following information:

- Description: Clients are applications and services that can request authentication of a user. [Learn more](#)
- Navigation tabs: Clients list (selected), Initial access token, Client registration.
- Search bar: Search for client
- Create client button: Create client
- Refresh button: Refresh
- Pagination: 1 - 6

The table below lists the existing clients:

Client ID	Name	Type	Description	Home URL
account	#{client_account}	OpenID Connect	-	<a href="http://localhost:8080/realm/SpringBootAppTest/account/">http://localhost:8080/realm/SpringBootAppTest/account/</a>
account-console	#{client_account-console}	OpenID Connect	-	<a href="http://localhost:8080/realm/SpringBootAppTest/account/">http://localhost:8080/realm/SpringBootAppTest/account/</a>
admin-cli	#{client_admin-cli}	OpenID Connect	-	-
broker	#{client_broker}	OpenID Connect	-	-
realm-management	#{client_realm-management}	OpenID Connect	-	-
security-admin-console	#{client_security-admin-console}	OpenID Connect	-	<a href="http://localhost:8080/admin/SpringBootAppTest/console/">http://localhost:8080/admin/SpringBootAppTest/console/</a>

# Clients

Hay que seguir los siguientes pasos en la creación del cliente

1 General settings      2 Capability config      3 Login settings

**Client type** OpenID Connect

**Client ID \*** `springbootapitest` 

**Name**

**Description**

**Always display in UI**  Off

# Clients

1 General settings

2 Capability config

3 Login settings

**Client authentication**  On 

**Authorization**  Off

**Authentication flow**

- Standard flow 
- Direct access grants 
- Implicit flow 
- Service accounts roles  
- OAuth 2.0 Device Authorization Grant 
- OIDC CIBA Grant 

# Clients

En este apartado, lo dejamos así, no necesitamos poner nada

- 1 General settings
- 2 Capability config
- 3 Login settings

Root URL 

Home URL 

# Clients

General settings

Client ID \* ⓘ springbootapitest

Name ⓘ

Description ⓘ

Always display in UI ⓘ  Off

Access settings

Root URL ⓘ

Home URL ⓘ

Admin URL ⓘ

[Logout settings](#)

[Save](#) [Revert](#)



Jump to section

- General settings
- Access settings
- Capability config
- Login settings
- Logout settings

# Clients

Aquí, tenemos nuestro ID y nuestro secret:

The screenshot shows the 'Client details' page for a client named 'springbootapitest'. The client is identified as an 'OpenID Connect' type. A red arrow points to the 'Client Id and Secret' dropdown menu, which contains the value 'nx6LU5k91s9Krw5OoLNVoCN8gTdOieKF'. Another red arrow points to the 'Copy' icon next to the client secret value. The page includes tabs for Settings, Keys, Credentials (which is selected), Roles, Client scopes, Service accounts roles, Sessions, and Advanced.

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

Client Authenticator: Client Id and Secret

Client Secret: nx6LU5k91s9Krw5OoLNVoCN8gTdOieKF

Registration access token

DDTIC\_DSJ\_PLI\_2024

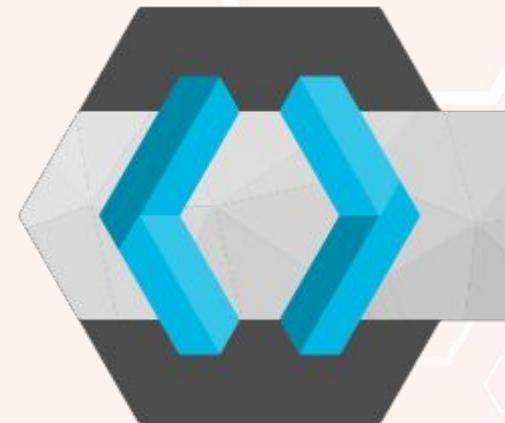
# Clients

Con esto terminamos la configuración básica de nuestro KeyCloak.

Igual, a futuro, podemos meternos más a la configuración de roles y permisos en KeyCloak para hacer pruebas con nuestro desarrollo en Spring.

Notas:

- KeyCloak es mi auth server
- SpringBoot sera resource server



# Desarrollo Spring App

Tomaremos como base el proyecto de spring-security-jwt-rest

Aquí es donde realizaremos nuestros cambios.

Empezaremos por añadir una librería al POM

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

# Clase KeycloakRoleConverter

Hay que crear una clase llamada KeycloakRoleConverter

Ésta la podemos poner en el paquete 'security'

```
1 public class KeycloakRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {
2     @Override
3     public Collection<GrantedAuthority> convert(Jwt jwt) {
4         Map<String, Object> realmAccess = (Map<String, Object>) jwt.getClaims().get("realm_access");
5         if (realmAccess == null || realmAccess.isEmpty()) {
6             return new ArrayList<>();
7         }
8         Collection<GrantedAuthority> returnValue = ((List<String>) realmAccess.get("roles"))
9             .stream().map(roleName -> "ROLE_" + roleName)
10            .map(SimpleGrantedAuthority::new)
11            .collect(Collectors.toList());
12         return returnValue;
13     }
14 }
```

# Clase KeycloakRoleConverter

La lógica que voy a escribir dentro de este KeycloakRoleConverter es que mi servidor de autenticación Keycloak devolverá un token de acceso.

Dentro de este token de acceso habrá detalles sobre las funciones de mi aplicación cliente o las funciones de mi usuario final.

Es mi responsabilidad extraer la información de esos roles o la información de las autoridades del token y convertirlos a un formato que mi marco de Spring Security pueda entender.

- Dentro de esta clase necesito implementar una interfaz con el nombre Converter.

# Clase KeycloakRoleConverter

Este convertidor aceptará el token JWT que recibe de mi servidor Keycloak y extraerá la información de los roles y enviará de vuelta la lista de GrantedAuthority porque mi marco Spring Security solo puede comprender la información de las reglas o información de las autoridades si envío en forma de objetos GrantedAuthority.

# SecurityConfig

Aquí vienen los cambios fuertes.

Hay que quitar todos los filtros antes creados.

De igual manera, quitaremos el passwordEncoder, authenticationProvider y authenticationManager

Agregaremos dentro del securityFilter el JwtAuthenticationConverter, que nos va a apoyar junto con KeycloakRoleConverter a obtener el token y convertir los roles que vendrán de Keycloak.

Igual agregaremos un llamado a oauth2ResourceServer

# SecurityConfig

```
 1  @Configuration
 2  @EnableWebSecurity
 3  public class SecurityConfiguration {
 4      @Bean
 5      public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
 6          JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
 7          jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter());
 8
 9          http
10              .authorizeHttpRequests((authorize) -> authorize
11                  .requestMatchers("/v1/contact-type/**").hasAnyRole("USER", "ADMIN")
12                  .anyRequest().authenticated()
13              )
14              .csrf(AbstractHttpConfigurer::disable)
15              .cors(Customizer.withDefaults())
16              .sessionManagement(session -> session
17                  .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
18              .oauth2ResourceServer(oauth2ResourceServerCustomizer ->
19                  oauth2ResourceServerCustomizer.jwt(jwtCustomizer -> jwtCustomizer
20                      .jwtAuthenticationConverter(jwtAuthenticationConverter)))
21      ;
22
23  }
```

# Limpieza proyecto

Igual, se vienen muchos cambios. Al final va a parecer que el proyecto se queda casi vacío.

Dentro del paquete 'security'

Eliminaremos TODO, solo dejaremos las clases  
SecurityConfigutation  
KeycloakRoleConverter

Igualmente, eliminaremos el paquete 'auth', ya que tiene las clases mapeadas de acceso a la BD login.

Solo nos quedaremos con el paquete 'system'

# KeyCloak

Ahora, necesitamos un Access Token.

Lo vamos a obtener de KeyCloak

Aquí, usando Postman, primero necesito enviar una solicitud al servidor de autorización con todos los detalles para obtener un token de acceso.

Con KeyCloak funcionando:

- Hay que acceder a la siguiente ruta (cada quien la modifica dependiendo de su proyecto)

<http://localhost:8080/realmSpringBootAppTest/.well-known/openid-configuration>

# KeyCloak

```
localhost:8080/realmSpringBootAppTest/.well-known/openid-configuration
myapps.cbre.com Swagger UI docker-compose... feign/DefaultError... DOM Solicitud de Reinte... Solicitud de Reinte... Solicitud de carga... Certificación - Pro... Contribuciones -... Tipo de contribuci... Aplicaciones Exter...
All Bookmarks

{"issuer": "http://localhost:8080/realmSpringBootAppTest", "authorization_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/auth", "token_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/token", "introspection_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/token/introspect", "userinfo_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/userinfo", "end_session_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/logout", "frontchannel_logout_session_supported": true, "frontchannel_logout_supported": true, "jwks_uri": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/certs", "check_session_iframe": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/login-status-iframe.html", "grant_types_supported": ["authorization_code", "implicit", "refresh_token", "password", "client_credentials", "urn:openid:params:grant-type:ciba", "urn:ietf:params:oauth:grant-type:device_code"], "acr_values_supported": ["0", "1"], "response_types_supported": ["code", "none", "id_token", "token", "id_token_token", "code_id_token", "code_id_token_token"], "subject_types_supported": ["public", "pairwise"], "id_token_signing_alg_values_supported": ["PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"], "id_token_encryption_alg_values_supported": ["RSA-OAEP", "RSA-OAEP-256", "RSA1_5"], "id_token_encryption_enc_values_supported": ["A256GCM", "A192GCM", "A128GCM", "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512"], "userinfo_signing_alg_values_supported": ["PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "none"], "userinfo_encryption_alg_values_supported": ["RSA-OAEP", "RSA-OAEP-256", "RSA1_5"], "userinfo_encryption_enc_values_supported": ["A256GCM", "A192GCM", "A128GCM", "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512"], "request_object_signing_alg_values_supported": ["PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512", "none"], "request_object_encryption_alg_values_supported": ["RSA-OAEP", "RSA-OAEP-256", "RSA1_5"], "request_object_encryption_enc_values_supported": ["A256GCM", "A192GCM", "A128GCM", "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512"], "response_modes_supported": [{"query", "fragment", "form_post", "query_jwt", "fragment.jwt", "form_post.jwt", "jwt"}, {"registration_endpoint": "http://localhost:8080/realmSpringBootAppTest/clients-registrations/openid-connect", "token_endpoint_auth_methods_supported": [{"private_key_jwt", "client_secret_basic", "client_secret_post", "tls_client_auth", "client_secret_jwt"}, {"token_endpoint_auth_signing_alg_values_supported": [{"PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"}], "introduction_endpoint_auth_methods_supported": [{"private_key_jwt", "client_secret_basic", "client_secret_post", "tls_client_auth", "client_secret_jwt", "introduction_endpoint_auth_signing_alg_values_supported": [{"PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"}], "authorization_signing_alg_values_supported": [{"PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"}], "authorization_encryption_alg_values_supported": [{"RSA-OAEP", "RSA-OAEP-256", "RSA1_5"}, {"authorization_encryption_enc_values_supported": [{"A256GCM", "A192GCM", "A128GCM", "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512"}], "claims_supported": [{"aud", "sub", "iss", "auth_time", "name", "given_name", "family_name", "preferred_username", "email", "acr"}, {"claim_types_supported": [{"normal"}, {"claims_parameter_supported": true, "scopes_supported": [{"openid", "micropoint-jwt", "offline_access", "profile", "phone", "roles", "web-origins", "email", "address", "acr"}], "request_parameter_supported": true, "request_uri_parameter_supported": true, "require_request_uri_registration": true, "code_challenge_methods_supported": [{"plain", "S256"}], "tls_client_certificate_bound_access_tokens": true, "revocation_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/revoke", "revocation_endpoint_auth_methods_supported": [{"private_key_jwt", "client_secret_basic", "client_secret_post", "tls_client_auth", "client_secret_jwt"}, {"revocation_endpoint_auth_signing_alg_values_supported": [{"PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"}], "backchannel_logout_supported": true, "device_authorization_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/auth/device", "backchannel_token_delivery_modes_supported": [{"poll", "ping"}], "backchannel_authentication_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/ext/ciba/auth", "backchannel_authentication_request_signing_alg_values_supported": [{"PS384", "RS384", "EdDSA", "ES384", "HS256", "HS512", "ES256", "RS256", "HS384", "ES512", "PS256", "PS512", "RS512"}], "require_pushed_authorization_requests": false, "pushed_authorization_request_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/ext/par/request", "mtls_endpoint_aliases": [{"token_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/token", "revocation_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/revoke", "introspection_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/token/introspect", "device_authorization_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/auth/device", "registration_endpoint": "http://localhost:8080/realmSpringBootAppTest/clients-registrations/openid-connect", "userinfo_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/userinfo", "pushed_authorization_request_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/ext/par/request", "backchannel_authentication_endpoint": "http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/ext/ciba/auth"}], "authorization_response_iss_parameter_supported": true}}}
```

# KeyCloak

Ahora, hay que buscar en todo ese texto lo siguiente: token\_endpoint

El primero que aparezca, es nuestro endpoint para solicitar el token

<http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/token>

También vamos a buscar el siguiente texto: jwks\_uri

Ese nos da la siguiente URL que es la que tenemos que ir a configurar a nuestro archivo YAML.

<http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/certs>

# KeyCloak

Solo hay que fijarse bien cómo agregamos este apartado

```
security:  
  oauth2:  
    resourceserver:  
      jwt:  
        jwk-set-uri: http://localhost:8080/realmSpringBootAppTest/protocol/openid-connect/certs
```

Ésto se hace para que mi servidor de recursos pueda descargar los certificados públicos del servidor Keycloak usando esta URL.

# Postman

Ahora, hay que trabajar con postman para conseguir el token.

Creamos un nuevo endpoint POST y pegamos la url obtenida anteriormente.

The screenshot shows the Postman application interface. A red box highlights the URL field and the body parameters section. The URL is set to `http://localhost:8080/realm/SpringBootAppTest/protocol/openid-connect/token`. The body is configured with the following parameters:

Key	Value
client_id	springbootapitest
client_secret	nx6LU5k91s9Krw5OoLNVoCN8gTdOieKF
scope	openid email profile address
grant_type	client_credentials

# Postman

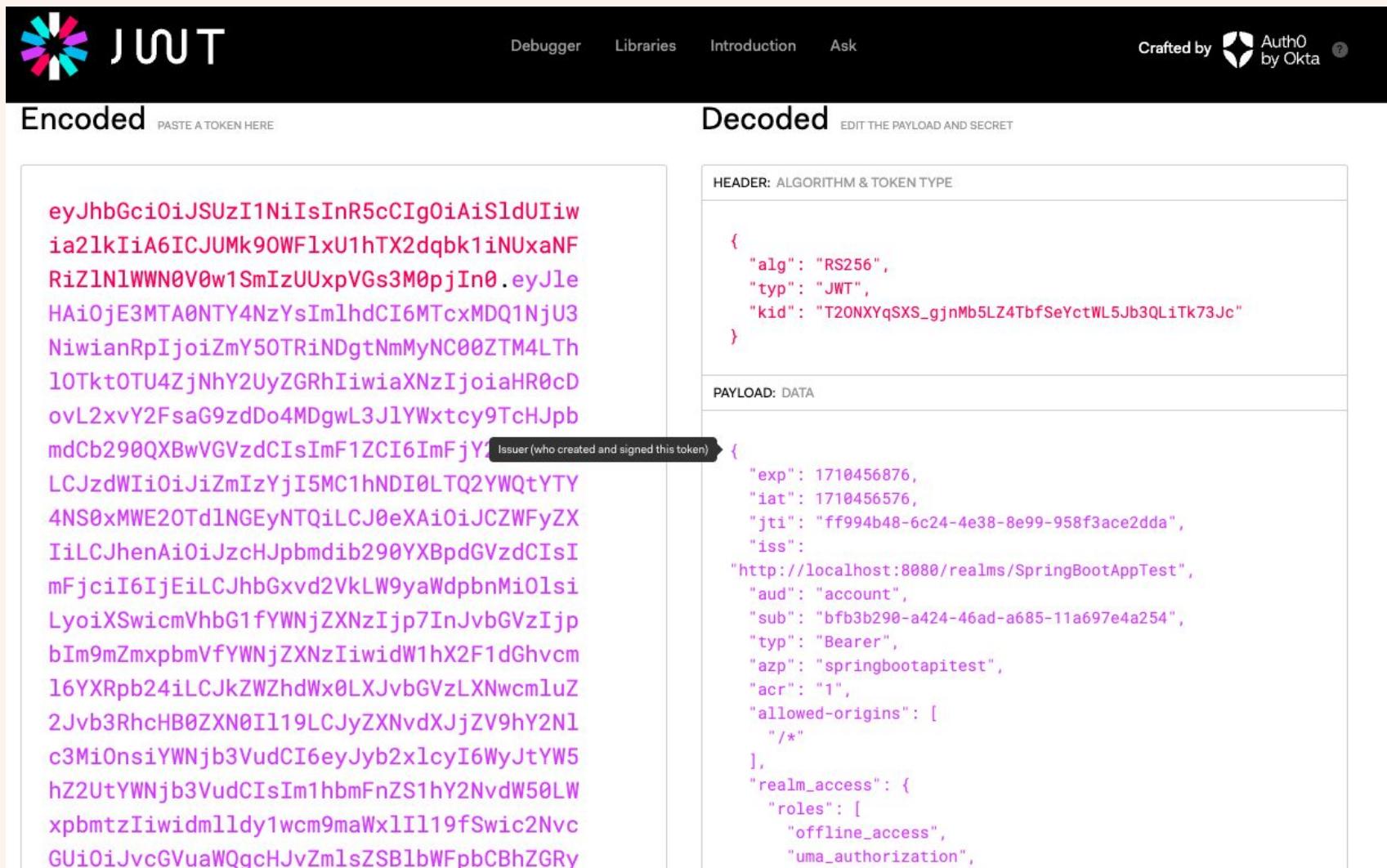
Aquí, ahora hay que poner nuestro client\_id y nuestro client\_secret en la opción de x-www-form-encoded

Aparte de estos, requerimos otros dos datos más:

scope - openid email profile address  
grant\_type - client\_credentials

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJUMk90WF1xU1hTX2dqbk1iNUxaNFRizlN1WWN0V0w1SmIzUUxpVGs3M0pjIn0.eyJleHAiOjE3MTA0NTY4NzYsImIhdCI6MTcxMDQ1NjU3NiwanRpIjoizmY50TRindgtNmMyN00ZTM4LTh10TktOTU4ZjNhY2UyZGRhIiwickXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL3J1YWxtcy9TchJpbmdCb290QBwVGVzdCisImF1ZC16ImFjY291bnQiLCJzdWIoiJiZmIzYjI5MC1hNDI0LTQ2Y0QtYT4N50xMWE20Td1NGEyNTQilCJ0eXAi0iJCZWFyZXiiLCJhenAi0iJzcHJpbmdib290YXBpdGVzdCisImFjciI6IjEiLCJhbGxd2VklW9yaWdpbnMiOlisiLyoiXSwicmVhbG1fYWNjZXNzIjw7InJvbGVfIjpbiM9mZxpbmVfYWNjZXNzIiwidW1hX2F1dGhvcml6YXRpb24iLCJkZWzhWx0LXJvbGVzLNwcmLuZ2Jvb3RhchB0ZXN0i119LCJyZXNvdXJjZ9hY2N1c3MiOnsiYWNj3VudCI6eyJyb2xlcycI6WytYNhZ2UtYWNb3VudCisIm1hbmfNzS1hY2NvdW50LWpbmtzIiwidm1ldy1wcm9maWx1Ii19fSwic2NvcGUioiJvcGVuaWQgcHJvZm1sZSB1bWFpbCBhZGRyZXNzIiwiZWh1haWxfdmVyaWZpZWQi0mZhbHNllCJjbG11bnRtB3N0IjoiMDowOjA6MDowOjA6MDoxIIwiYWRkcmVzcyI6e30sInByZWlcnJlZf91c2VybmtZSI6InNlcnZpY2UtyWNjb3VudC1zcHJpbmdib290YXBpdGVzdCisImNsawVudEFkZHJlc3Mi0iIw0jA6MDowOjEiLCJjbG11bnRfaWQioiJzcHJpbmdib290YXBpdGVzdC39. VBXeQa0wwL7HzBGd1ktDwXvrVJS0t7ER6EvG_2yrcDqWa2ijfpdkwfHhmGwsWikldaAcnxgmwSEAnykfNvqKngFx5zLXxcsD7X_zg_FtizDWovRd-PaTrkRpt08xNHHIKMrZZVnyVt3eSzhbCI6JhWFEV5zp3ygkmakGk7zo6fx1-_y6ESPp003woKFrvhrQwy85cYnESXQpAfw1nx1DN22KEFtel-taFCRgwaZRRLy6bMIDbS2ZdsIMIkSwFetgxU1K5Qxu_DZSDqcIhdEA0pp1p8cza-DZKvqS32iOgW0H27Y_FbLs5MN089cx6MQvrlkedbn36y7G2NMmvQ",
  "expires_in": 300,
  "refresh_expires_in": 0,
  "token_type": "Bearer",
  "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJUMk90WF1xU1hTX2dqbk1iNUxaNFRizlN1WWN0V0w1SmIzUUxpVGs3M0pjIn0.eyJleHAiOjE3MTA0NTY4NzYsImIhdCI6MTcxMDQ1NjU3NiwanRpIjoizmY50TRindgtNmMyN00ZTM4LTh10TktOTU4ZjNhY2UyZGRhIiwickXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL3J1YWxtcy9TchJpbmdCb290QBwVGVzdCisImF1ZC16ImFjY291bnQiLCJzdWIoiJiZmIzYjI5MC1hNDI0LTQ2Y0QtYT4N50xMWE20Td1NGEyNTQilCJ0eXAi0iJJRCisImF6cCI6InNwcmLuZ2Jvb3RhchB0ZXN0i119LCJyZXNvdXJjZ9hY2N1c3MiOnsiYWNj3VudCI6eyJyb2xlcycI6WytYNhZ2UtYWNb3VudCisIm1hbmfNzS1hY2NvdW50LWpbmtzIiwidm1ldy1wcm9maWx1Ii19fSwic2NvcGUioiJvcGVuaWQgcHJvZm1sZSB1bWFpbCBhZGRyZXNzIiwiZWh1haWxfdmVyaWZpZWQi0mZhbHNllCJjbG11bnRtB3N0IjoiMDowOjA6MDowOjA6MDoxIIwiYWRkcmVzcyI6e30sInByZWlcnJlZf91c2VybmtZSI6InNlcnZpY2UtyWNjb3VudC1zcHJpbmdib290YXBpdGVzdC39. VBXeQa0wwL7HzBGd1ktDwXvrVJS0t7ER6EvG_2yrcDqWa2ijfpdkwfHhmGwsWikldaAcnxgmwSEAnykfNvqKngFx5zLXxcsD7X_zg_FtizDWovRd-PaTrkRpt08xNHHIKMrZZVnyVt3eSzhbCI6JhWFEV5zp3ygkmakGk7zo6fx1-_y6ESPp003woKFrvhrQwy85cYnESXQpAfw1nx1DN22KEFtel-taFCRgwaZRRLy6bMIDbS2ZdsIMIkSwFetgxU1K5Qxu_DZSDqcIhdEA0pp1p8cza-DZKvqS32iOgW0H27Y_FbLs5MN089cx6MQvrlkedbn36y7G2NMmvQ",
  "not-before-policy": 0
}
```

# Postman



The screenshot shows the Postman interface with a token analysis tool. On the left, under 'Encoded' (PASTE A TOKEN HERE), a long string of characters is displayed in purple. On the right, under 'Decoded' (EDIT THE PAYLOAD AND SECRET), the token structure is shown in JSON format:

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "T20NXYqSXS_gjnMb5LZ4TbfSeYctWL5Jb3QLiTk73Jc"  
}  
  
PAYLOAD: DATA  
  
{  
  "exp": 1710456876,  
  "iat": 1710456576,  
  "jti": "ff994b48-6c24-4e38-8e99-958f3ace2dda",  
  "iss": "http://localhost:8080/realms/SpringBootAppTest",  
  "aud": "account",  
  "sub": "bfb3b290-a424-46ad-a685-11a697e4a254",  
  "typ": "Bearer",  
  "azp": "springbootapitest",  
  "acr": "1",  
  "allowed-origins": [  
    "*"  
  ],  
  "realm_access": {  
    "roles": [  
      "offline_access",  
      "uma_authorization",  
      "profile"  
    ]  
  }  
}
```

# Creación de Roles en Keycloak

Hay que crear nuestros roles que venían desde la BD en nuestro Authority Server

USER

ADMIN

The screenshot shows the Keycloak administration interface for a realm named 'SpringBootAppTest'. The left sidebar has a dark theme with white text and includes links for Manage, Clients, Client scopes, Realm roles (which is selected and highlighted in grey), Users, Groups, Sessions, and Events. The main content area is titled 'Realm roles' and contains a sub-header: 'Realm roles are the roles that you define for use in the current realm.' followed by a 'Learn more' link. Below this is a search bar with the placeholder 'Search role by name' and a blue 'Create role' button. A large red arrow points to the 'Create role' button. The table below lists three existing roles: 'default-roles-springbootapptest', 'offline\_access', and 'uma\_authorization'. Each row in the table includes columns for 'Role name', 'Composite' (with values True, False, and False respectively), and 'Description' (with values \${role\_default-roles}, \${role\_offline-access}, and \${role\_uma\_authorization}).

Role name	Composite	Description
default-roles-springbootapptest	True	\${role_default-roles}
offline_access	False	\${role_offline-access}
uma_authorization	False	\${role_uma_authorization}

# Creación de Roles en KeyCloak

The screenshot shows the Keycloak interface for creating new roles. There are two separate role creation forms displayed.

**Role 1: USER**

- Role name \***: USER
- Description**: (Empty text area)
- Buttons**: Save (blue button), Cancel
- Tab navigation**: Details (selected), Attributes, Users in role

**Role 2: ADMIN**

- Role name**: ADMIN
- Description**: (Empty text area)
- Buttons**: Save (blue button), Cancel

# Creación de Roles en Keycloak

## Realm roles

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

Role name	Composite	Description
ADMIN	False	—
USER	False	—

Ahora, hay que agregar estos roles al cliente

# Creación de Roles en KeyCloak

The screenshot shows the Keycloak interface for managing clients. On the left, a sidebar menu is visible with options: Manage, Clients (which has a red arrow pointing to it), Client scopes, Realm roles, Users, Groups, Sessions, and Events. The main content area is titled "Clients > Client details" for a client named "springbootapitest". The client is described as "OpenID Connect". A sub-header states: "Clients are applications and services that can request authentication of a user." Below this, there are tabs for Settings, Keys, Credentials, Roles, Client scopes, Service accounts roles, and another partially visible tab. A red arrow points from the "Clients" sidebar option down to the "Roles" tab in the main content area. The "General settings" section contains fields for Client ID (set to "springbootapitest"), Name, and Description.

Clients > Client details

springbootapitest OpenID Connect

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

General settings

Client ID \* ⓘ springbootapitest

Name ⓘ

Description ⓘ

# Creación de Roles en Keycloak

No hacer click en Roles porque Roles es la pestaña que debemos usar siempre que un usuario final esté involucrado, siempre que intentemos asignar roles a un usuario final.

Pero aquí, este cliente, que es **springbootapitest**, es una cuenta de servicio que se utiliza para comunicaciones API a API.

Es por eso que debemos hacer clic en los roles de esta cuenta de servicio.

# Creación de Roles en KeyCloak

The screenshot shows the 'Service accounts roles' tab selected in the navigation bar. A tooltip message says: 'To manage detail and group mappings, click on the username [service-account-springbootapitest](#)'. Below the search bar, there is a checked checkbox for 'Hide inherited roles'. The 'Assign role' button is highlighted with a red arrow. The table lists one role: 'default-roles-springbootapptest'.

Name	Inherited	Description
default-roles-springbootapptest	False	\${role_default-roles}

# Creación de Roles en KeyCloak

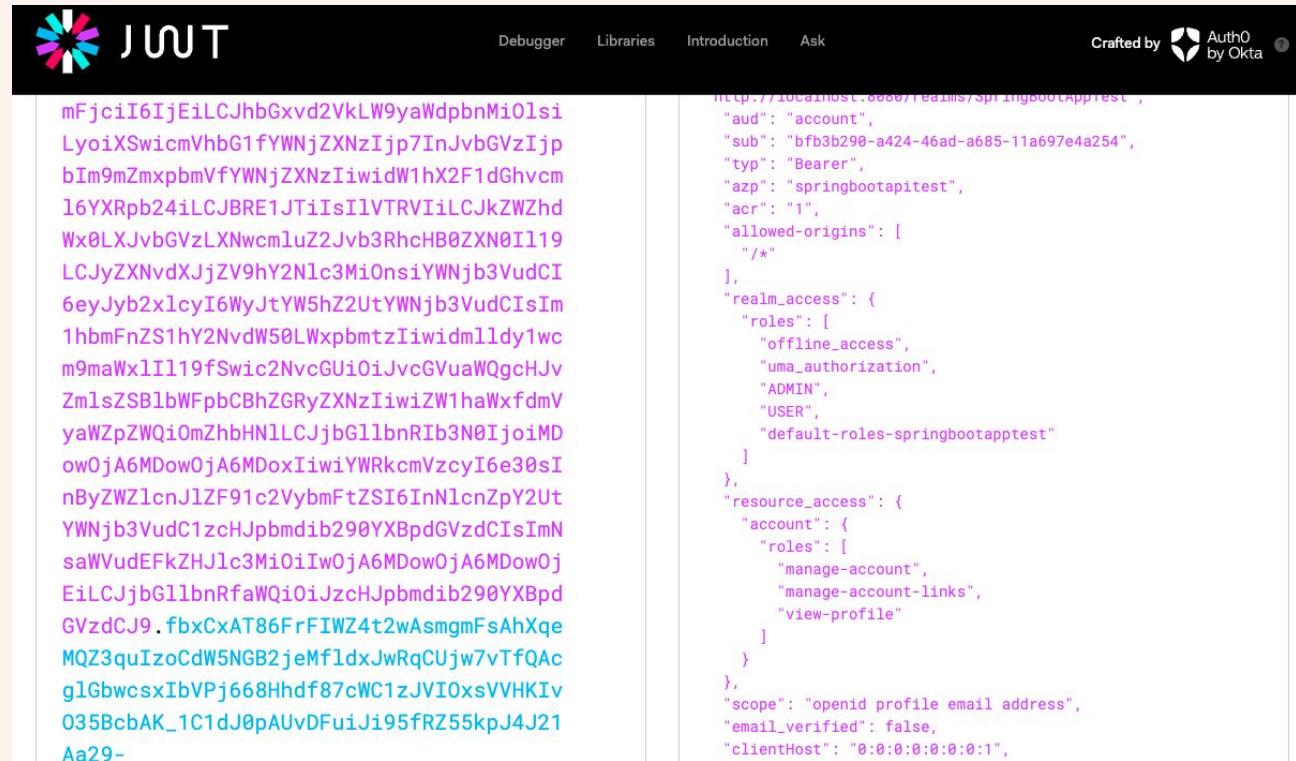
Assign roles to springbootapitest

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



# Creación de Roles en Keycloak

Volvemos a ejecutar Postman, y validamos el token creado en jwt.io y vemos que ya aparece nuestros nuevos roles.



```
http://localhost:8080/realm斯pringbootapptest
{
  "aud": "account",
  "sub": "bf3b290-a424-46ad-a685-11a697e4a254",
  "typ": "Bearer",
  "azp": "springbootapitest",
  "acr": "1",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "uma_authorization",
      "ADMIN",
      "USER",
      "default-roles-springbootapptest"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid profile email address",
  "email_verified": false,
  "clientHost": "0:0:0:0:0:0:1"
}
```

# Recta Final!!!

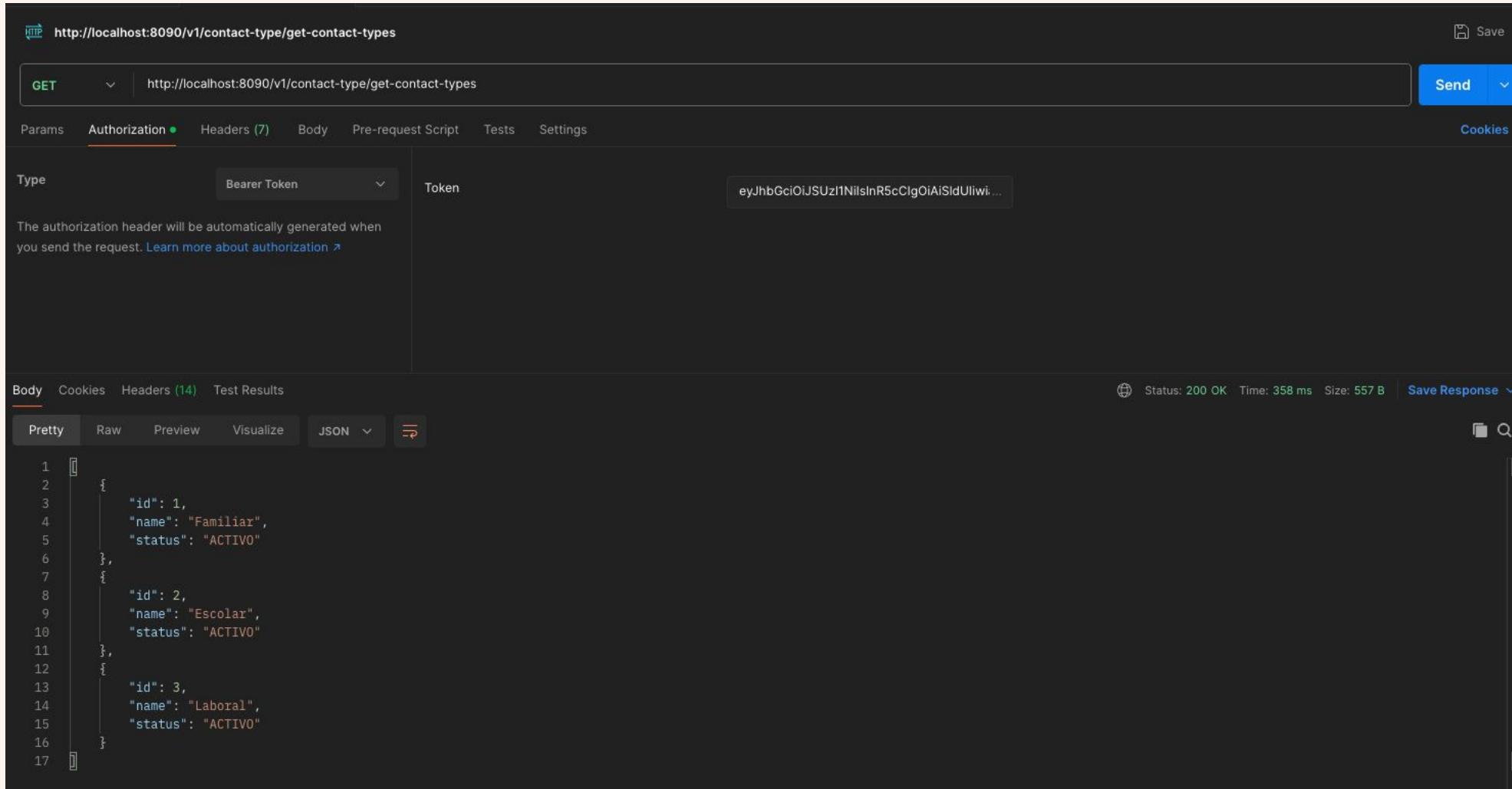
Ya tenemos todo listo y configurado.

Lo más pesado fue el KeyCloak (no podemos ver más, esto no está dentro del scope del backend).

Sin embargo, ya podemos ejecutar de nuevo el proyecto en Spring, y ejecutar en postman el endpoint creado anteriormente

Ahí revisaremos si funciona nuestro lector de Token!

# Recta Final!!!



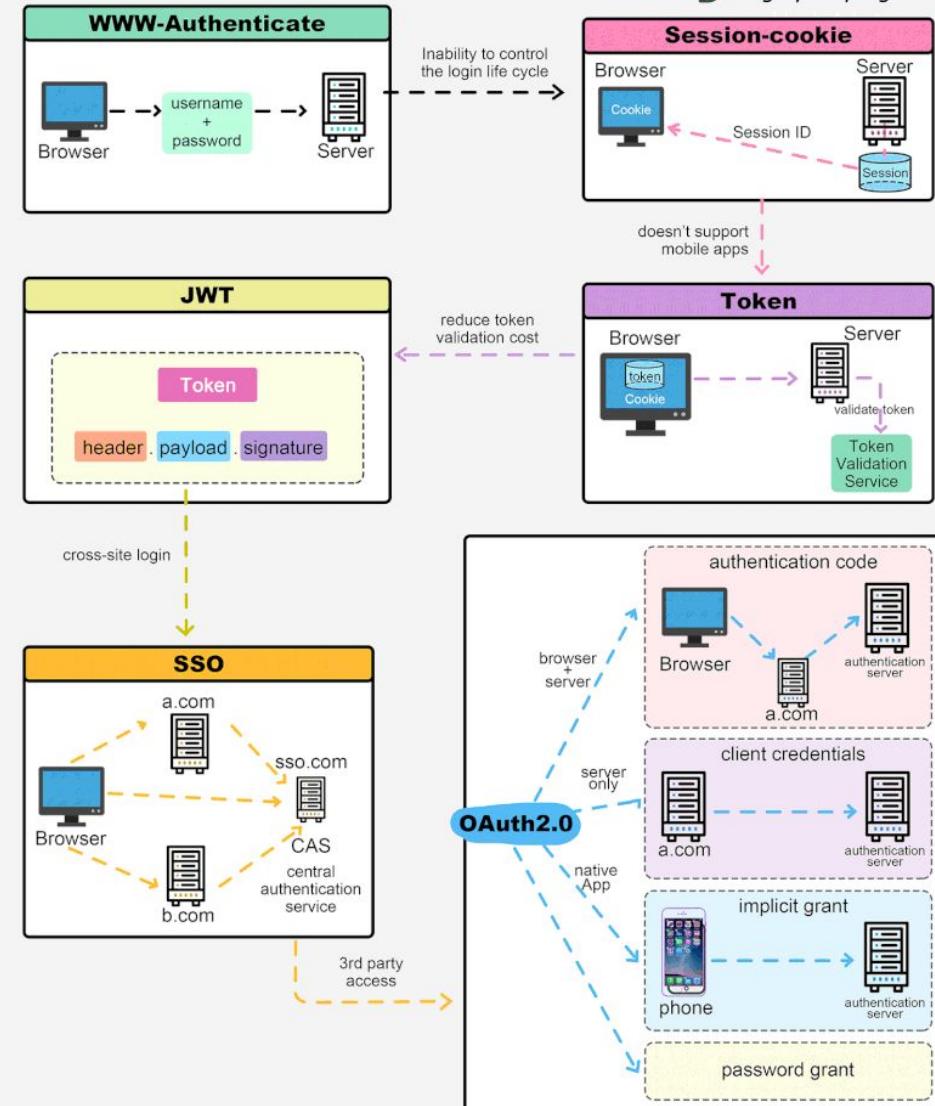
The screenshot shows a Postman API request for `http://localhost:8090/v1/contact-type/get-contact-types`. The request method is `GET`. In the `Authorization` tab, a `Bearer Token` is specified with the value `eyJhbGciOiJSUzI1NiisInR5cC1gOAIaSidUliwi...`. The response status is `200 OK`, with a time of `358 ms` and a size of `557 B`. The response body is a JSON array containing three objects, each representing a contact type:

```
1 [ { "id": 1, "name": "Familiar", "status": "ACTIVO" }, { "id": 2, "name": "Escolar", "status": "ACTIVO" }, { "id": 3, "name": "Laboral", "status": "ACTIVO" } ]
```

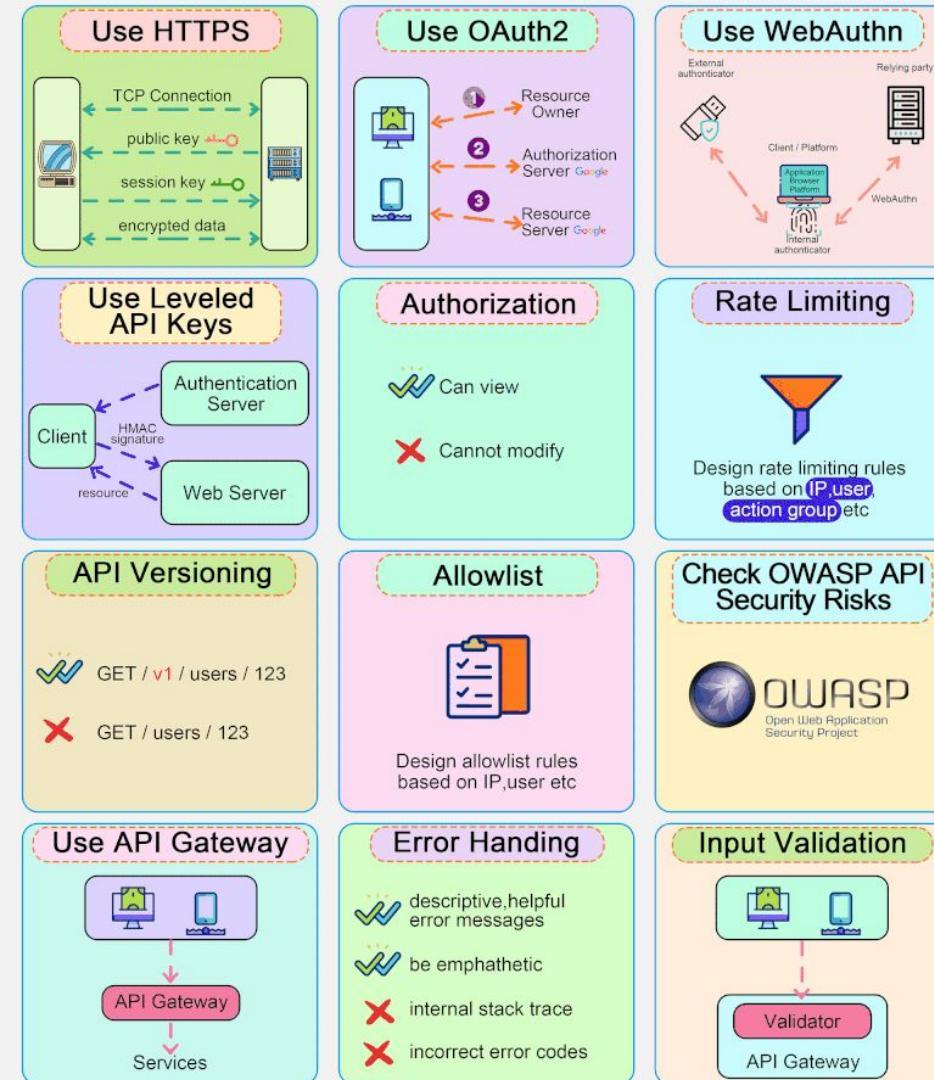
# REST API Authentication Methods

 ByteByteGo

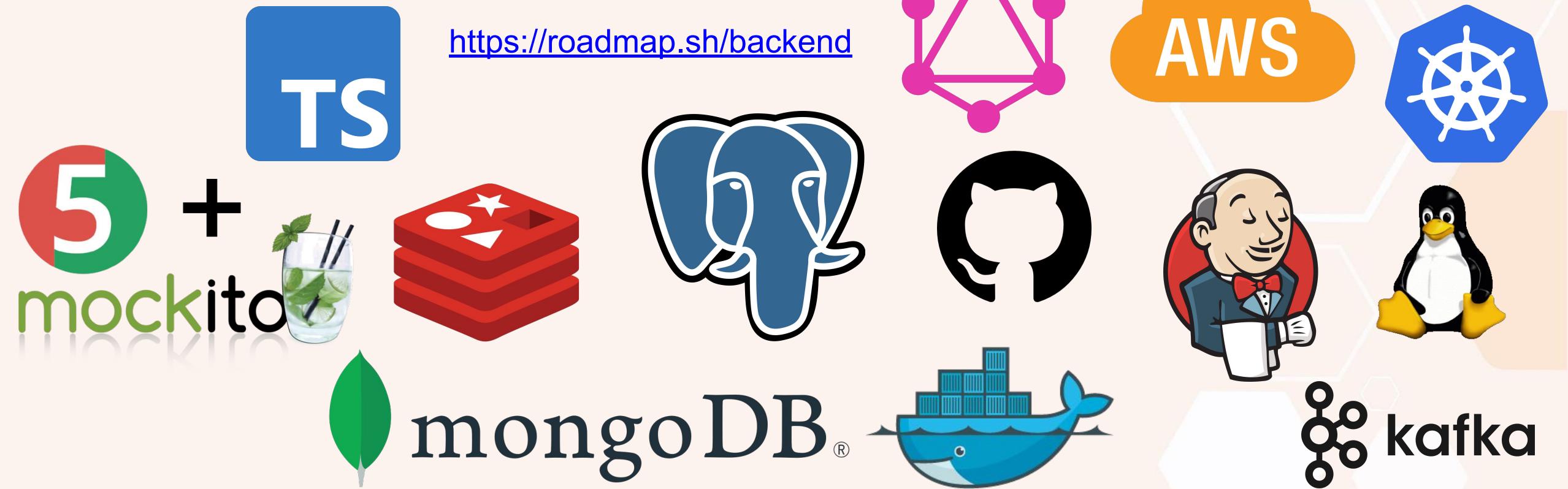

# Session,JWT,Token,SSO,OAuth 2.0

 blog.bytebytogo.com


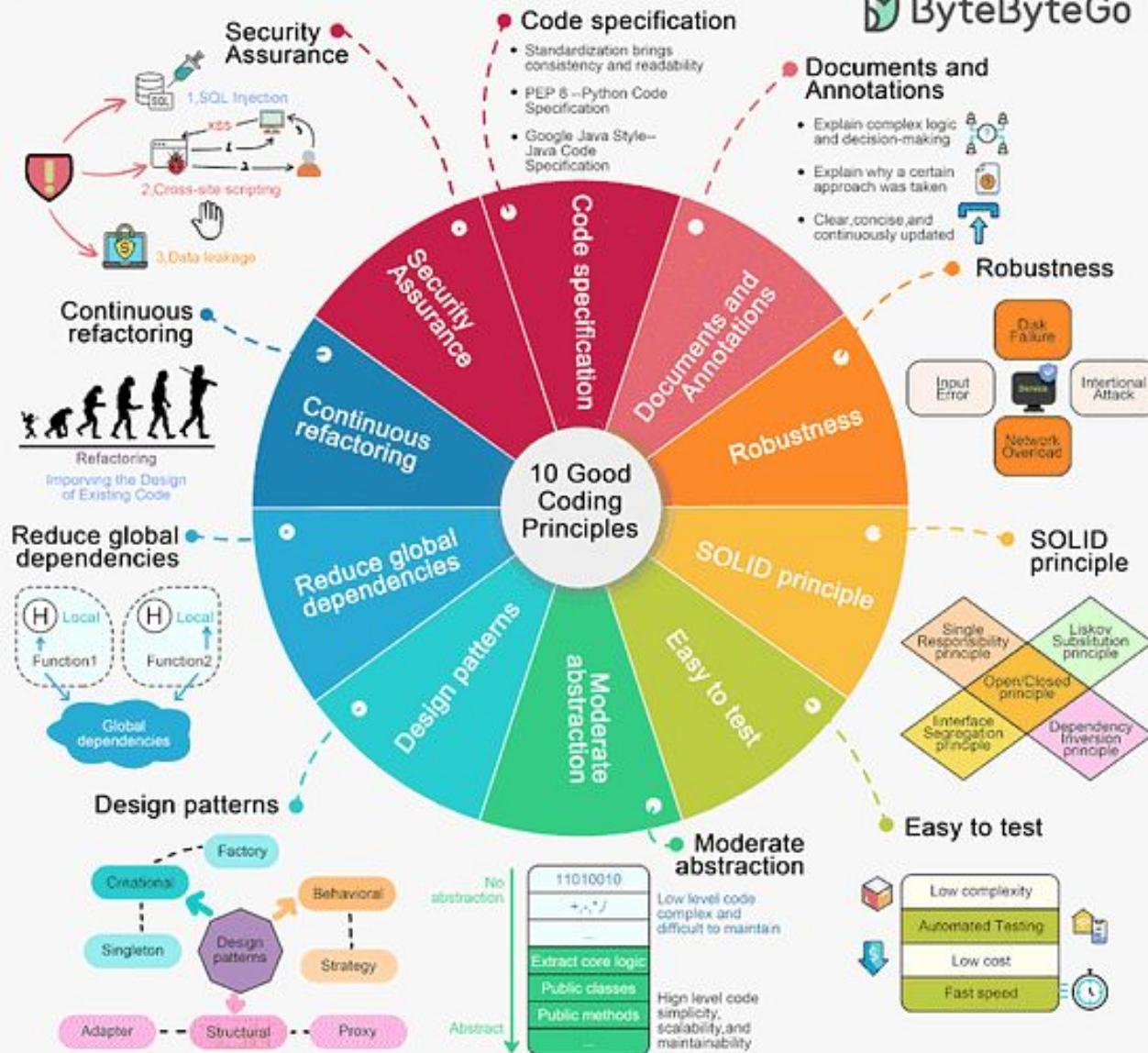
# 12 Tips for API Security

 ByteByteGo

# ¿Qué sigue?



# 10 Good Coding Principles

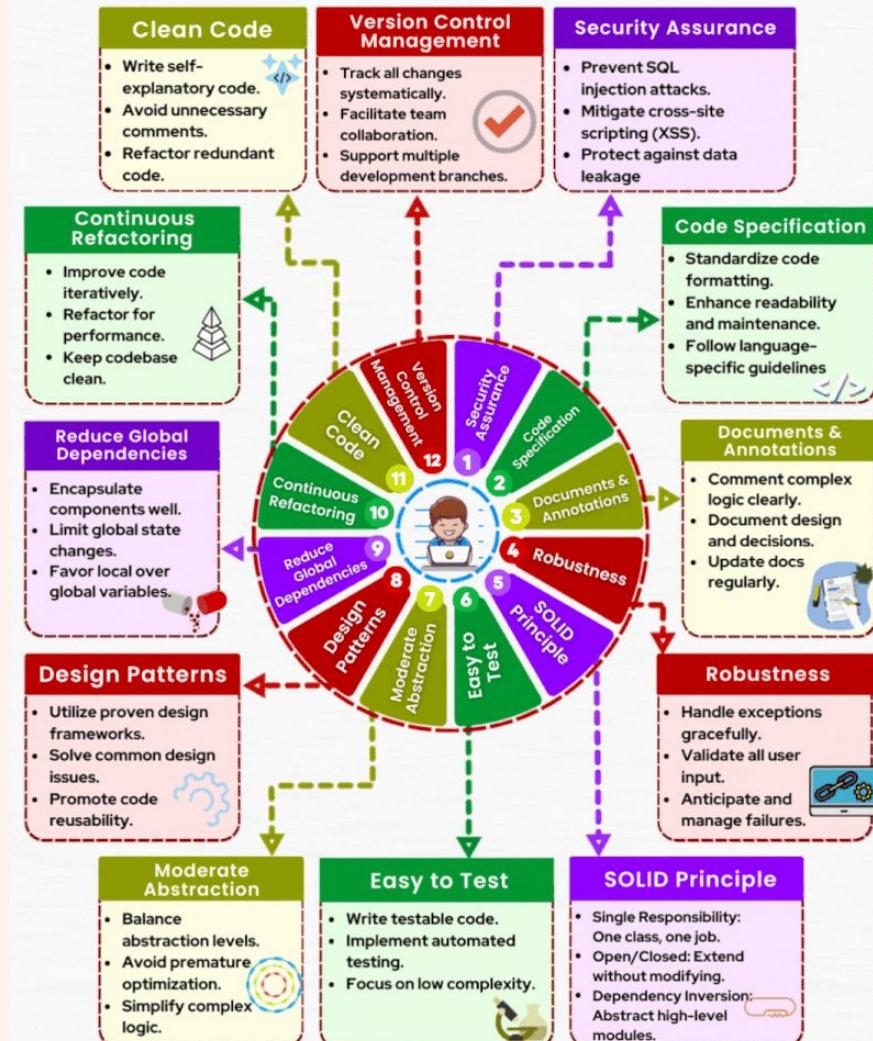


ByteByteGo

Technical Career  
Branding Coach

Hina Arora  
@hinaarora

## Top 12 Coding Principles



## Java Roadmap 2024

<https://www.youtube.com/watch?v=fO9HN3nvDLI>

# Contacto

Mtro. Alfonso Gregorio Rivero Duarte  
*Senior Data Manager - CBRE*

devil861109@gmail.com

Tels: (+52) 55 289970 69

Redes sociales:



<https://www.linkedin.com/in/alfonso-gregorio-rivero-duarte-139a9225/>