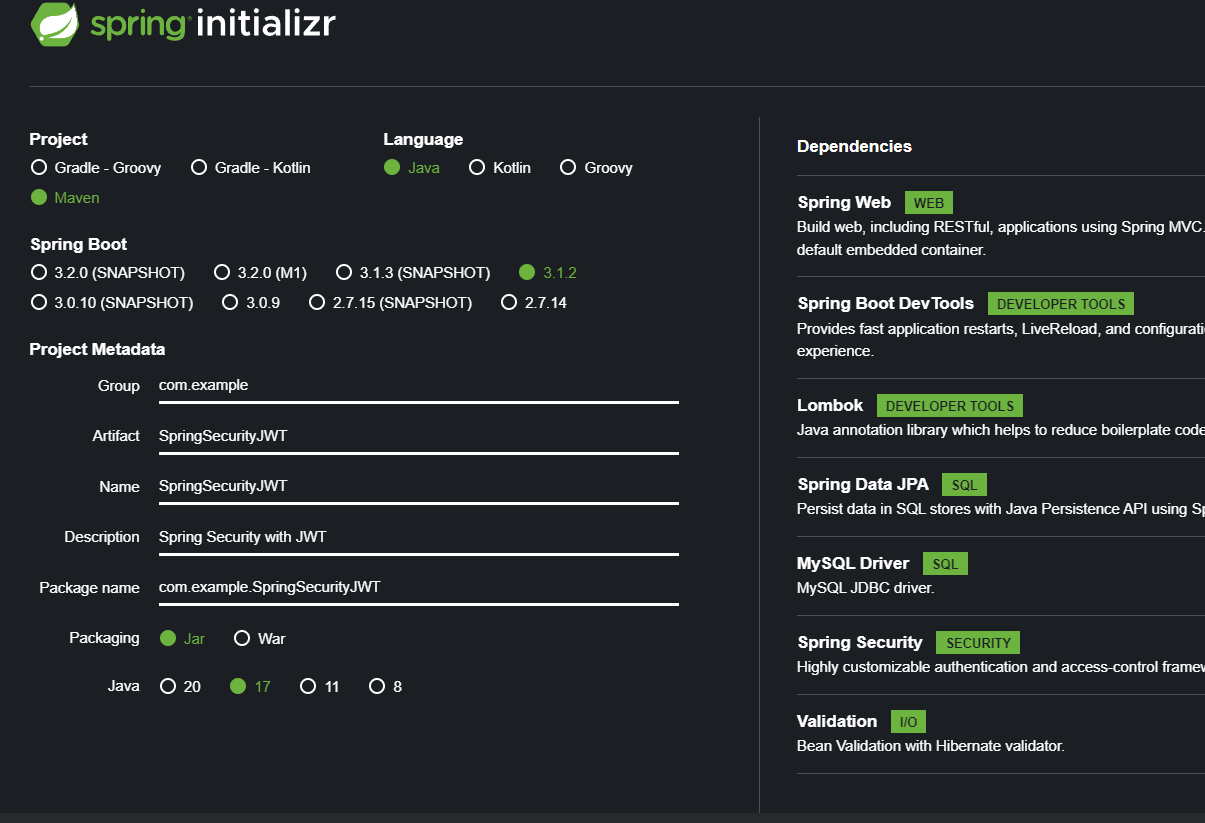
Spring Security with JWT

1 - Vamos a generar el Proyecto en spring initializr



2 – Una vez importado el Proyecto buscamos Spring boot banner en inet y generamos el siguiente banner

,---. ,--. ,---. ,--. ,--. | | ,--.,--. ,--.,--------.

' .-' ,---. ,--.--.`--',--,--, ,---. ' .-' ,---. ,---.,--.,--.,--.--.`--',-' '-.,--. ,--. ,---| |---. | || | | |'--. .--'

`. `-.| .-. || .--',--.| \| .-. |`. `-.| .-. :| .--'| || || .--',--.'-. .-' \ ' / '---| |---' ,--. | || |.'.| | | |

.-' | '-' '| | | || || |' '-' '.-' \ --.\ `--.' '' '| | | | | | \ ' | | | '-' /| ,'. | | |

`-----'| |-' `--' `--'`--''--'.`- / `-----' `----' `---' `----' `--' `--' `--' .-' / `--' `-----' '--' '--' `--'

“Spring Security + JWT”

3 – Ahora vamos a las dependencias y comentamos la siguiente (y luego recargamos el maven/proyecto)

<!-- <dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency> -->

4 – Ahora vamos a configurar la DB

Vamos a application.properties y escribimos la siguiente configuracion

spring.datasource.url=jdbc:mysql://localhost:3306/securitydb

spring.datasource.username=root

spring.datasource.password=Asdf7928

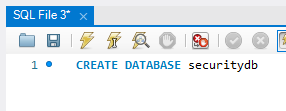
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=create

Vamos a mySQL y creamos la table securitydb



Una vez hecho esto, la app ya funciona y se puede comenzar a trabajar en ella

5 – Dentro del paquete com.example.SpringSecurityJWT creamos el paquete models donde iran los modelos de las clases

La primer clase a crear es UserEntity

*@Entity*

*@Data*

*@AllArgsConstructor*

*@NoArgsConstructor*

*@Builder*

*@Table*(name="users")

public class UserEntity {

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private Long id;

*@Email*

*@NotBlank*

*@Size*(max = 80)

private String email;

*@NotBlank*

*@Size*(max = 30)

private String userName;

*@NotBlank*

private String password;

}

Para la proxima clase que se va a llamar RoleEntity, primero creamos un enum llamado ERole que indicara el tipo de rol para cada usuario

public enum *ERole* {

***ADMIN***, ***USER***, ***GUEST***

}

Una vez creado el enum, creamos la clase RoleEntity

*@Data*

*@AllArgsConstructor*

*@NoArgsConstructor*

*@Builder*

*@Entity*

*@Table*(name = "roles")

public class RoleEntity {

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private Long id;

*@Enumerated*(*EnumType*.***STRING***)

private *ERole* name;

Tener en cuenta que la anotacion Enumerated lleva dentro de los args un EnumType.String ya que si bien es un enum, aquello que esta dentro del mismo, al fin y al cabo es un String

6 – Es necesario configurar la relacion entre las entidades mediante una tabla intermedia ya que la relacion que existe entre UserEntity y RoleEntity es de muchos a muchos; y, dado que la clase que nos interesa desarollar es UserEntity, solo en ella vamos a declarer lo necesario

Empezamos por agregar un atributo privado del tipo . Esto es ya que Set, a diferencia de List u otro tipo de arreglo, no permite duplicados (esto es para que el usuario no tenga 2 veces declarado un mismo valor de tipo de usuario)

Ahora le agregamos las anotaciones:

*@ManyToMany*(fetch = *FetchType*.***EAGER***)

Establece el tipo de relacion entre las clases. Se seleciona FetchType.Eager ya que esta eleccion nos va a traer todos los roles que contiene la clase; a diferencia de Lazy que solo nos trae lo solicitado

*@ManyToMany*(fetch = *FetchType*.***EAGER***, targetEntity = RoleEntity.class

Esto va a declarar la clase con la que se va a generar la relacion

*@ManyToMany*(fetch = *FetchType*.***EAGER***, targetEntity = RoleEntity.class, cascade = *CascadeType*.***PERSIST***)

El .PERSIST hace que al momento de eliminar un usuario, no se borre también el Rol. (esto romperia la db y dejaria inhabilitados a los demas usuarios con roles invalidos) y al momento de crear el usuario, también se crea automaticamente el ROL

Ahora creamos la table intermedia:

*@JoinTable*(name = "user\_roles"

Indica el nombre de la table intermedia

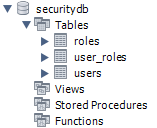
*@JoinTable*(name = "user\_roles", joinColumns = *@JoinColumn*(name = "user\_id")

El joinColumn indica cual es la clave foranea de la clase UserEntity

*@JoinTable*(name = "user\_roles", joinColumns = *@JoinColumn*(name = "user\_id"), inverseJoinColumns = *@JoinColumn*(name = "role\_id"))

El inverseJoinColumns indica la clave foranea de la clase RoleEntity

Si volvemos a correr la app, se ejecutaran directamente las querys de creacion de las tablas y las relaciones entre las mismas y al refrescar la DB deberian aparecer las nuevas tablas



7 – Ahora creamos el paquete Repositories y la interfaz UserRepository, a dicha interfaz la extendemos de CrudRepository<UserEntity, Long> y le agregamos la anotacion de @Repository

*@Repository*

public interface UserRepository extends CrudRepository<UserEntity, Long> {

Ahora vamos a crear un metodo por el cual nos devuelva un usuario buscado mediante su nombre (no me parece buena la idea pero en este caso asi se hara a modo explicativo)

Optional<UserEntity> findByUserName(String userName);

Esta forma de escribir el metodo utiliza directamente un metodo propio de Java para retornar lo buscado mediante params

La otra opcion es indicar mediante Query lo que queremos que el metodo nos devuelva

*@Query*("select u from UserEntity u where u.useName = ?1")

Optional<UserEntity> getName(String userName);

(la u se utiliza como comodin y el = ?1 significa que debe ser igualado al primer parametro indicado)

Ahora es necesario crear el repositorio de Rol

*@Repository*

public interface RoleRepository extends CrudRepository<RoleEntity, Long>{

}

8 – Una vez creados los repositories, vamos a crear el PrincipalController (anotacion de @RestController) dentro del paquete controller y le agregamos 2 metodos que devuelvan un String; con 2 EndPoints distintos para poder mostrar la diferencia entre un endpoint con seguridad y otro sin

*@RestController*

public class PrincipalController {

*@GetMapping*("/hello")

public String hello() {

return "Hello world not secured";

}

*@GetMapping*("/helloSecured")

public String helloSecured() {

return "Hello world nsecured";

}

9 – Agregamos un paquete llamado “request” y creamos la clase CreateUserDto

*@Data*

*@AllArgsConstructor*

*@NoArgsConstructor*

public class CreateUserDTO {

*@Email*

*@NotBlank*

private String email;

*@NotBlank*

private String userName;

*@NotBlank*

private String password;

private Set<String> roles;

10 – Volvemos al controller PrincipalController y creamos el metodo createUser con un return ResponseEntity<?> que va a recibir por parametro un CreateUserDTO (va a llevar las anotaciones @RequestBody para avisar que va a recibir la informacion de un CreateUserDTO complete y @Valid para que automaticamente corrobore dichos datos)

El metodo lleva la anotacion de PostMapping con el endpoint “/createUser”

*@PostMapping*("/createUser")

public ResponseEntity<?> createUser(*@Valid* *@RequestBody* CreateUserDTO createUserDTO){

\*A tener en cuenta = ya que por params vamos a recibir los datos de un usuario, vamos a tener que desarmar la info para poder asignarla al usuario que vamos a crear\*

Primero vamos a crear un Set<RoleEntity> roles , aqui le vamos a asignar el valor que llega por param y para hacer esto primero hay que hacer un getRoles().Stream().map, esto nos devuelve un map con los roles que estan asignados en los datos recibidos.

Set<RoleEntity> roles = createUserDTO.getRoles().stream()

.map

Utilizamos una function Lambda y asignamos el nombre role a la variable recuperada y utilizamos el metodo builder (para esto usamos la anotacion @Builder que nos permite ir creando un objeto con metodos automaticos y por partes)

(role -> RoleEntity.*builder*()

Luego hacemos .name para asignar el nombre del Rol. Al ser un enum Rol, vamos a utilizar el metodo valuOf de ERole para recuperar el valor en String de dicho enum enviado recibido y luego llamamos al metodo .build().

Por ultimo usamos el metodo .collect(Collectors.toSet()) para transformer la info a un Set

.name(*ERole*.*valueOf*(role))

.build())

.collect(Collectors.*toSet*());

El bloque complete es el siguiente

Set<RoleEntity> roles = createUserDTO.getRoles().stream()

.map(role -> RoleEntity.*builder*()

.name(*ERole*.*valueOf*(role))

.build())

.collect(Collectors.*toSet*());

Ahora vamos a crear el UserEntity (que es el motive del metodo per se) y lo igualamos a UserEntity.build() (de nuevo, vamos a ir creandolo parte por parte)

UserEntity userEntity = UserEntity.*builder*()

.userName(createUserDTO.getUserName())

.password(createUserDTO.getPassword())

.email(createUserDTO.getEmail())

.roles(roles)

.build();

(tener en cuenta que roles es el Set que recopilamos en el bloque anterior)

Ahora creamos un atributo del tipo UserRepository en la clase y le agregamos la anotacion @Autowired ya que la informacion la va a recibir por inyeccion

A este atributo lo llamamos desde el metodo y hacemos .save(userEntity) este userEntity es el resultado de la creacion del user

this.userRepository.save(userEntity);

Ahora para finalizer el metodo debemos retornar un una respuesta .ok y enviar el usuario creado

return ResponseEntity.*ok*(userEntity);

el metodo coompleto es el siguiente

*@PostMapping*("/createUser")

public ResponseEntity<?> createUser(*@Valid* *@RequestBody* CreateUserDTO createUserDTO){

Set<RoleEntity> roles = createUserDTO.getRoles().stream()

.map(role -> RoleEntity.*builder*()

.name(*ERole*.*valueOf*(role))

.build())

.collect(Collectors.*toSet*());

UserEntity userEntity = UserEntity.*builder*()

.userName(createUserDTO.getUserName())

.password(createUserDTO.getPassword())

.email(createUserDTO.getEmail())

.roles(roles)

.build();

this.userRepository.save(userEntity);

return ResponseEntity.*ok*(userEntity);

}

Por ultimo, creamos un metodo que va a borrar a un usuario a traves de su id

*@DeleteMapping*("/deleteUser")

public String deleteUser(*@RequestParam* String id) {

userRepository.deleteById(Long.*parseLong*(id));

return "User deleted satisfactory";

}

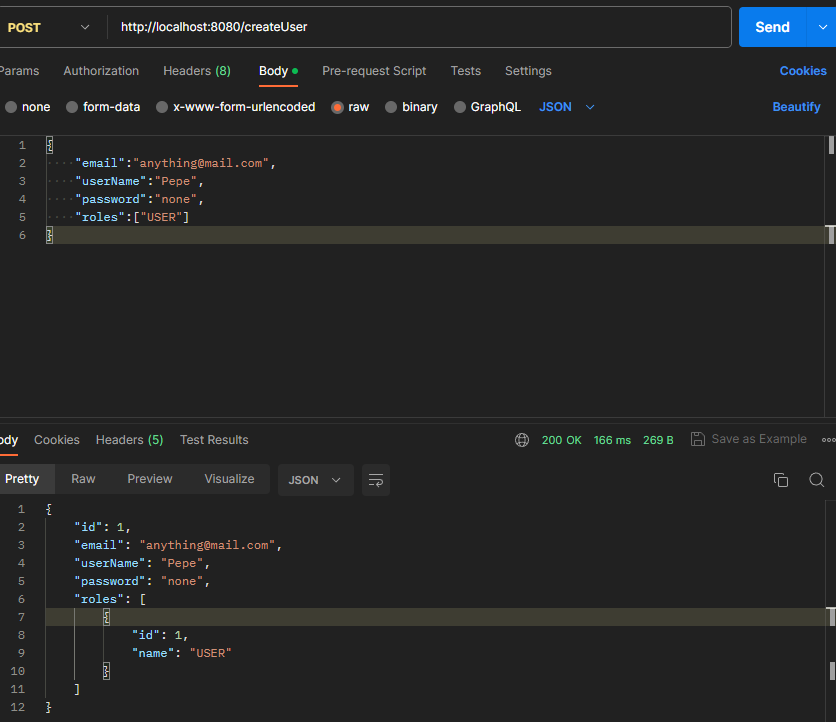
Ahora podemos poner a prueba si la app sigue funcionando y vamos a utilizar POSTMAN para enviar por header el body para crear el usuario

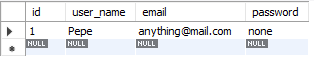
A tener en cuenta:

El metodo debe ser POST y la info que enviamos va a ser por “BODY > RAW”

La info que enviamos debe ser en formato JSON y esto también debe ser indicado en POSTMAN (un error es que esté enviando un text u otro formato)

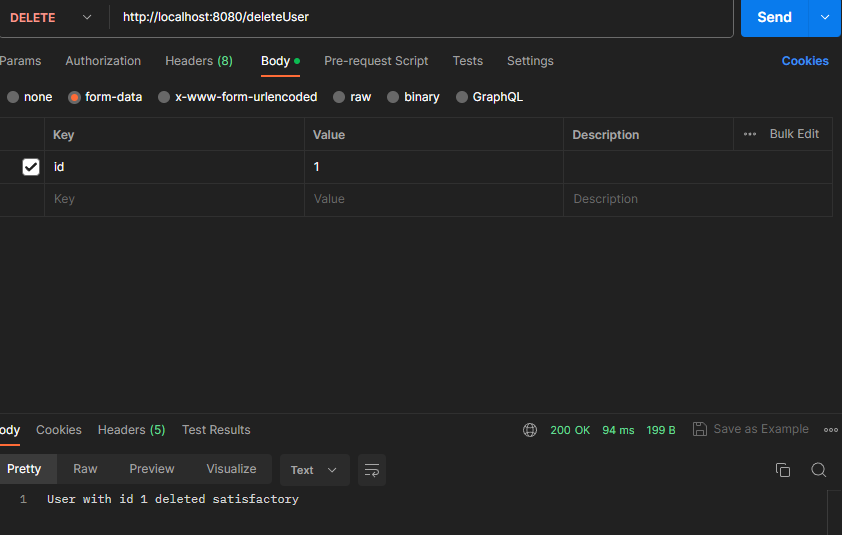
Luego de la creacion del usuario podemos corroborar en la DB si se guardó el nuevo usuario y a su vez la creación del ROL “USER” debido al cascade.PERSIST

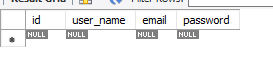






Ahora ponemos a prueba el metodo delete y usando Postman vamos al endpoint deseado  
Esta vez enviamos por BODY>FORM-DATA” el id 1 (o el id del usuario deseado).  
Al revisar la DB debería haber sido borrado el usuario pero NO ASÍ el rol, ya que por el Cascade.PERSIST el rol debería persistir pese al delete del user







11 – Vamos a comenzar a utilizar las dependencias de seguridad por lo que primero que hacemos es ir a pom.xml y descomentamos la siguiente dependencia

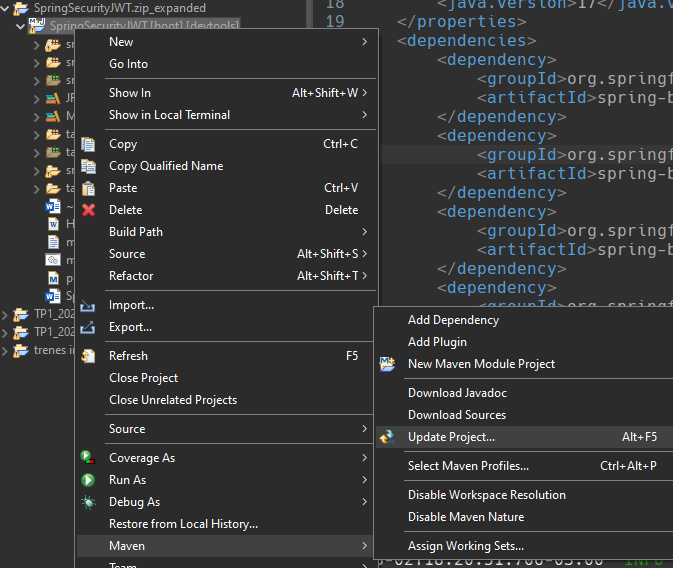
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

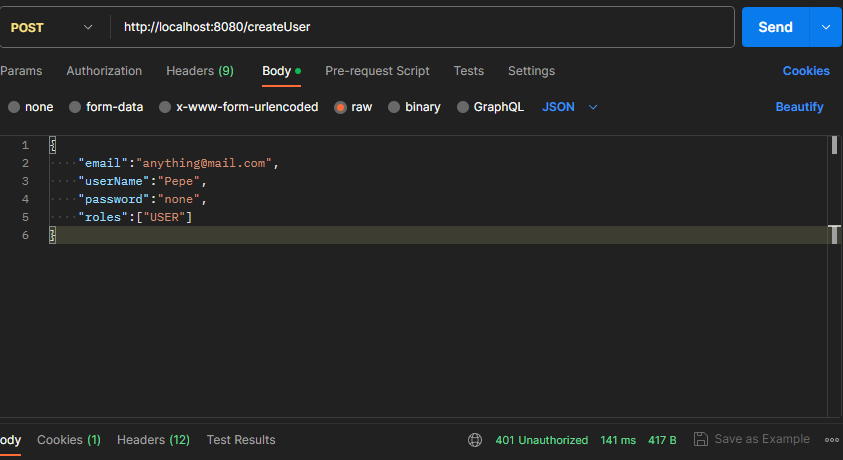
</dependency>

Lo siguiente es refrescar el Proyecto desde Maven



Automaticamente, todos los ENDPOINTS creados quedaran anulados ya que se levanto la seguridad del Proyecto y no se puede acceders a los ENDPOINTS sin estar logueado

Al intentar crear un usuario vía Postman, la respuesta del servidor sera “401 unauthorized”. Lo mismo sucedera si queres elminar un usuario o acceder a los otros ENDPOINTS

Ahora vamos a configurar la seguridad del Proyecto:  
Creamos un paquete llamado “security” y luego la clase “SecurityConfiguration” (agregamos la anotacion Configuration que indica que dicha clase va a utilizar @Beans)

Agregamos el siguiente metodo con la anotacion @Bean y el throws Exception

*@Bean*

SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {

return httpSecurity

.build();

}

HttpSecurity es una conexion segura que por defecto viene con varias implementaciones que protegen las vulnerabilidades de la conexion via URL

Ahora al return le agregaremos los siguientes metodos para controlar o no ciertas vulnerabilidades.

La primera que vamos a modificar/remover es la “.csrf.disable()”

“csrf” significa CROSS-SITE-REFQUEST-FORGERY” y lo que hace es proteger el envio de archivos xml, json y todo tipo de formularios, previniendo que al momento de enviar las peticiones al servidor no se filtren nuestros datos (ya sea nombre de usuario, password u otras informaciones sensibles)  
En este caso vamos a deshabilitar esta proteccion de la siguiente manera

.csrf(config -> config.disable())

La siguiente es “.authorizeHttpRequests” que nos permite modificar el acceso a los EP segun nuestra necesidad

.authorizeHttpRequests(auth -> {

auth.requestMatchers("/hello").permitAll();

auth.anyRequest().authenticated();

})

En este caso estamos indicando que al EP “/hello” puede acceder cualquier persona con o sin autorizacion; luego con el anyRequest indicamos que al resto de los EP solo se podrá ingresar con autenticacion

Ahora vamos a establecer la forma en que se maneja la creacion de las sesiones . Existen 4 posibilidades

ALWAYS: Siempre que se ingrese al enpoint creara una nueva session

IF\_REQUIRED: Solo creara una session si no existe una en uso, caso contrario, utilizara la existente

NEVER: No creara ninguna session pero si existe una en uno, la utilizara; caso contrario, intentara acceder al ENDPOINT sin session

STATELESS: Intentara acceder a todos los ENDPOINT sin session alguna y retornara los resultados de dicha request

En este caso se utilizara STATELESS

.sessionManagement(session ->{

session.sessionCreationPolicy(*SessionCreationPolicy*.***STATELESS***);

Ahora se le debería agregar un “.httpBasic()” pero está deprecado y no entiendo como se utiliza ahora con lambda

.~~httpBasic~~()

(si no se usa el httpBasic aunque esté deprecado, todas las consultas seran respuestas con Forbidden)

Creamos un usuario para poder probar (tener en cuenta que está hardcodeado de momento)

*@Bean*

UserDetailsService userDetailsService() {

InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

manager.createUser(User.*withUsername*("Santiago")

.password("1234")

.roles()

.build());

return manager;

}

Ahora vamos a crear un password Encoder.

Tener en cuenta que en el tutorial se llama a la clase NoOpPasswordEncoder qeu está deprecada (De igual forma hay que usarla ya que caso contrario, tira forbidden)

*@Bean*

PasswordEncoder passwordEncoder() {

return ~~NoOpPasswordEncoder~~.~~getInstance~~();

Este password Encoder será utilizado por un Authentication Manager para evitar que se use un usuario sin contraseña encriptada

*@Bean*

AuthenticationManager authenticationManager(HttpSecurity httpSecurity, PasswordEncoder passwordEncoder) throws Exception {

return httpSecurity.getSharedObject(AuthenticationManagerBuilder.class)

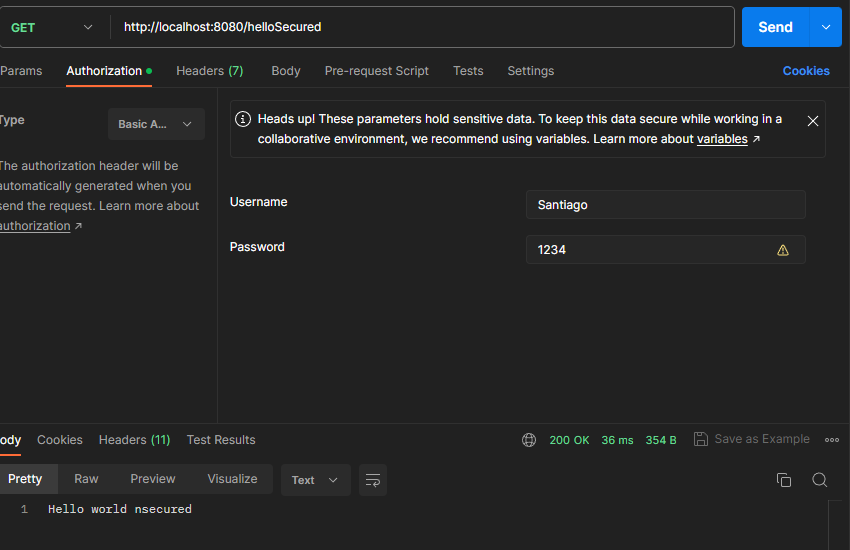
.userDetailsService(userDetailsService())

.passwordEncoder(passwordEncoder)

.~~and~~().build();

}

Una vez terminado este codigo podremos probar la seguiridad de los EP. Para acceder a los mismos deberemos siempre autenticar datos.



12 – Ahora vamos a modificar y aplicar JWT.   
Lo primero es modificar el método passwordEncoder de SecurityConfiguration

*@Bean*

PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

BCryptPasswordEncoder crea un algoritmo de encriptación el cual marcará la pólitica con la que se trabajaran las contraseñas.

Ahora vamos a PrincipalController y declaramos un atributo del tipo PasswordEncoder con una anotación de @Autowired para que reciba por inyección la información

*@Autowired*

private PasswordEncoder passwordEncoder;

Y ahora en el método createUser de la misma clase modificamos la creación del userEntity y enviamos el password Encriptado con el siguiente metodo

.password(this.passwordEncoder.encode(createUserDTO.getPassword()))

Dentro del paquete “security” creamos un nuevo paquete llamado “jwt” y dentro del mismo una clase llamada “JwtUtils” donde trabajaremos todas las utilidades necesarias de JWT.  
Primero a la clase le agregamos una anotación de @Component y declaramos dos variables del tipo String: “secretKey” y “timeExpiration”

*@Component*

public class JwtUtils {

private String secretKey;

private String timeExpiration;

}

secretKey va a ser un token interno generado en hexadecimal que servira a modo de garantizar que nadie podrá corromper o romper la capa de seguridad de JWT.   
timeExpiration marcara el tiempo de duración de dicho token

Buscamos en google “encrypted key generator hex” y generamos un token de esta manera (use la web seanwasere.com)

Luego marcamos el tiempo de duracion del token. En este caso 1 dia (siempre debe ser expresado en milisegundos. 1 día = 86400000 ms)

#JWT

jwt.secret.key=41fd894208cfe5026e0cd5ddce74070f58c6c63f26f98dc26208fce5e94aa0cb

jwt.time.expiration=86400000

Vamos a JwtUtils y a los atributos declarados les agregamos la anotacion @Value para trazar la referencia entre los mismos y los declarados en la application.properties

Dentro de los params debemos escribir la referencia deseada

*@Value*("${jwt.secret.key}")

private String secretKey;

*@Value*("${jwt.time.expiration}")

private String timeExpiration;

Ahora vamos a importar las dependencias necesarias para utilizar JWT:  
Vamos al pom.xml y escribimos lo siguiente

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

Una forma de corroborar si tenemos la ultima version, es ir a maven dependency en google y buscar la api jjwt-api para que nos diga cual es la última version disponible

También agregamos

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

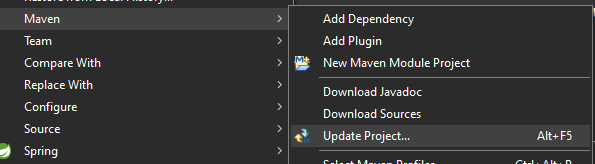
<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

</dependency>

Por último, recargamos el Maven (click derecho sobre el Proyecto>Mave>update project o Alt+F5)



13 – Para la creación del token creamos el método “generateAccessToken” que retorna un String (dado que el token es un String) y que recibe por parametro un String que contiene el username  
En el return ponemos Jwts.builder() (Jwts crea una instancia de las posibles interfaces que ofrece Jwts, en este caso sera utilizado para una instancia de String con formato de token y luego utiliza el método builder() para ir creando el objeto paso a paso)

public String generateAccessToken(String userName) {

return Jwts.*builder*()

Lo primero es agregarle el usuario que va a generar el token mediante el siguiente método

.setSubject(userName)

Lo siguiente es asignarle una fecha de creación al token (esto sera en MiliSegundos)

.setIssuedAt(new Date(System.*currentTimeMillis*()))

Ahora hay que asignarle una fecha de expiración (también en ms)  
La forma correcta de hacer esto es utilizar la fecha de creación y a esta le el tiempo de expiración, ósea, sumamos el atributo de “this.timeExpiration” el cual está expresado en String por lo que habrá que parsearlo a LONG (dado que los ms son expresados en este tipo de dato)

.setExpiration(new Date(System.*currentTimeMillis*() + Long.*parseLong*(this.timeExpiration)))

Lo que sigue es asignarle una “firma” (una key) que será obtenida desde el atributo inyectado “this.secretKey”, la misma primero será desencriptada para ser utilizada y luego enviada nuevamente encriptada

Para esto vamos a crear un método que retorna una Key llamado “getSignatureKey”. Dentro del mismo vamos a decodificar la clave encerrada en el atributo inyectado creando un arreglo del tipo byte[].  
Para decodificar la clave utilizamos el siguiente método

byte[] keyBytes = Decoders.***BASE64***.decode(this.secretKey);

y en el return envíamos el array de bytes de la siguiente forma

return Keys.*hmacShaKeyFor*(keyBytes)

El método entero quedaría así

//Token's sign maker

public Key getSignatureKey() {

byte[] keyBytes = Decoders.***BASE64***.decode(this.secretKey);

return Keys.*hmacShaKeyFor*(keyBytes);

}

Ahora volvemos al método generador del token y agregamos el método signWith y por params enviamos primero el resultado del método decodificador y luego le enviamos el tipo de encriptamiento que queremos asignarle a la key asignada. El método quedara así (Y SIEMPRE DEBE USAR ESTE ALGORITMO)

.signWith(this.getSignatureKey(), *SignatureAlgorithm*.***HS256***)

Para finalizar agregamos el método compact() para unir los datos; el método entero quedaría así

//Access token generator

public String generateAccessToken(String userName) {

return Jwts.*builder*()

.setSubject(userName)

.setIssuedAt(new Date(System.*currentTimeMillis*()))

.setExpiration(new Date(System.*currentTimeMillis*() + Long.*parseLong*(this.timeExpiration)))

.signWith(this.getSignatureKey(), *SignatureAlgorithm*.***HS256***)

.compact();

}

Lo siguiente es corroborar si el token es correcto

Creamos el método isValidToken que recibe por params un String (el token) y retorna un Boolean  
Dentro del mismo utilizamos un try catch.   
En el try utilizamos el método paserBuilder() de la clase “Jwts” para descomprimir la información (es el método contrario a builder())

//Token's Validation

public boolean isTokeValid(String token) {

try {

Jwts.parserBuilder()

} catch(Exception e) {

}

}

Lo siguiente es el método setSigningKey que recibe por params la firma para validar que sea correcta o que no esté nula.

.setSigningKey(this.getSignatureKey())

Luego le agregamos un .build()

.build()

Luego utilizamos el método “parseClaimsJws”para corroborar que el token que recibe por params fue correctamente encriptado

.parseClaimsJws(token)

Y por ultimo el método getBody() para retornar todo en un formato del tipo String o Claims de Jwts

.getBody();

Si nada de esto tira error, se retorna true (el método retorna true, que significa que el token es valido)

Caso contrario, en el catch retornamos false y para imprimir el error agregamos a la clase la anotacion @Slf4j lo que nos permite usar el log para enviar los mensajes de la siguiente manera

***log***.error("Token invalido, error".concat(e.getMessage()));

El método entero queda así

//Token's Validation

public boolean isTokeValid(String token) {

try {

Jwts.*parserBuilder*()

.setSigningKey(this.getSignatureKey())

.build()

.parseClaimsJws(token)

.getBody();

return true;

} catch(Exception e) {

***log***.error("Token invalido, error".concat(e.getMessage()));

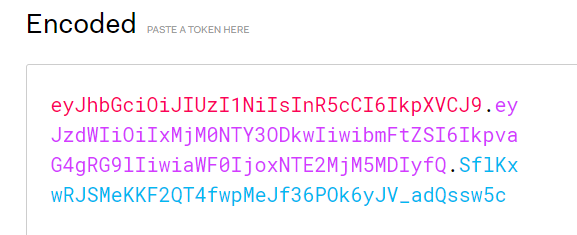
return false;

}

}

Algo importante para tener en cuenta es como está conformado un token. Si vamos a <https://jwt.io/> vamos a ver como está compuesto el mismo

El token encriptado se ve así

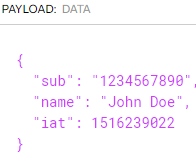


En sí, el token está compuesto por 3 partes

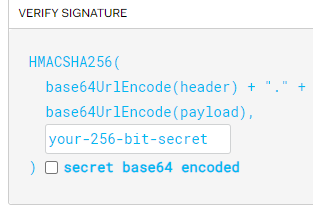
La primer parte (color rojo) es el header, ahí viaja el algoritmo en el que está encriptado y el tipo de token



La Segunda parte (violeta) está compuesta por la información que encripta el token (hay que tener en cuenta que es visible por todos y puede ser desencriptado, por lo que no debe haber info sensible)



En el tercer termino va el JSON que encierra el Header, el JSON que encierra el Payload y la firma secreta (Key) para confirmer que la información viaja sin ser modificada



14 – Vamos a crear los métodos para recuperar la info que viaja en los token, para ellos primero vamos a hablar de los Claims  
Los Claims son piezas de información del usuario , extienden de Map, por lo que pueden guardar mucha información y existen 3 tipos :

1. Registrados: Son Claims predeterminados que no son obligatorios pero si son recomendados para operar la información entre ellos. Tales como “iss”(issuer, tiempo de creación), “exp” (expiration date), “sub” (subject), “aud” (audience), etc.
2. Públicos: Pueden ser definidos por aquellos que usan JWTs pero para evitar que se repitan, deben ser declarados en la registro IANA JSON de los web tokens.
3. Privados: Son utilizados entre grupos cerrados para compartir info y no son registrados

Ahora, vamos a crear el método extractAllClaims que recibe por params el token (String) y retorna Jwts.parseBuilder() que cumple la function inversa al builder() (ósea que decodifica la información en segmentos).   
A este return le iremos agregando los siguientes métodos:

Método setSigningKey que recibe por params la firma para validar que sea correcta o que no esté nula.

.setSigningKey(this.getSignatureKey())

Luego le agregamos un .build()

.build()

Luego utilizamos el método “parseClaimsJws”para corroborar que el token que recibe por params fue correctamente encriptado

.parseClaimsJws(token)

Y por ultimo el método getBody() para retornar todo en un formato del tipo String o Claims de Jwts

.getBody();

El método complete queda así

//Get token's claims

public Claims extractAllClaims(String token) {

return Jwts.*parserBuilder*()

.setSigningKey(this.getSignatureKey())

.build()

.parseClaimsJws(token)

.getBody();

}

Ahora creamos el método getClaim() para recuperar un Claim especifico. Para esto vamos a recibir por params el Token y una \*Function<Claims, T> “calimsTFunction”\* y vamos a retornar una <T> “T” (usamos el comodín genérico para que marcar que el Claim puede contener cualquier tipo de dato)

Lo primero que hacemos es declarar un Claims (recordar que extiende de Map) y en el guardaremos todos los Claims obtenidos del método “this.extractAllClaims(String)”. Luego retornaremos la Function que ingresó por parametro y mediant el método “apply()” devolveremos unicamente el claim especificado.

\* Para aclarar: La function que recibimos por params contiene dos comodines <T, R>. En el Type ingresa un Claim que contendrá toda la información, en la R ira el return deseado, ósea un método que devolvera un valor que querramos averiguar u obtener (que en este caso es un X tipo de claim, por lo que usaremos el generic de T). Lo que hace esto es que en el return (del método getClaim) podamos aplicar el Método “apply” (de la interfaz Function) que lo que hace es devolver el resultado del método en la R que debería ser un dato del Claims (el input T).

//Get specific claim

public <T> T getClaim(String token, Function<Claims, T> claimsTFunction) {

Claims claims = this.extractAllClaims(token);

return claimsTFunction.apply(claims);

}

Para dar un ejemplo claro vamos a crear el método “getUserNameFromToken” el mismo recibira por params al token y retornara un String (el nombre)  
Directamente en el return llamaremos al método “this.getClaim” y enviaremos por params el token y en el lugar de la function enviaremos “Claims::getSubject” (los “::” representan la function, del lado Izquierdo va el input y del derecho va el tipo de retorno <R,T>)



El método sería el siguiente

//Get token's userName

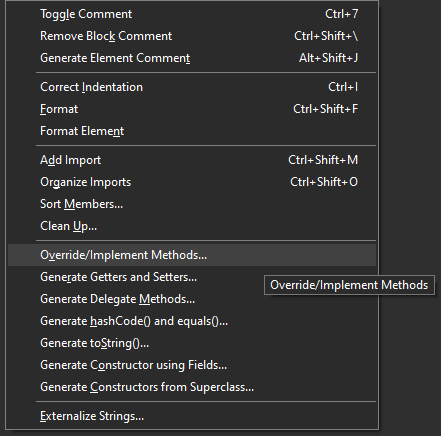
public String getUserNameFromToken(String token) {

return this.getClaim(token, Claims::getSubject);

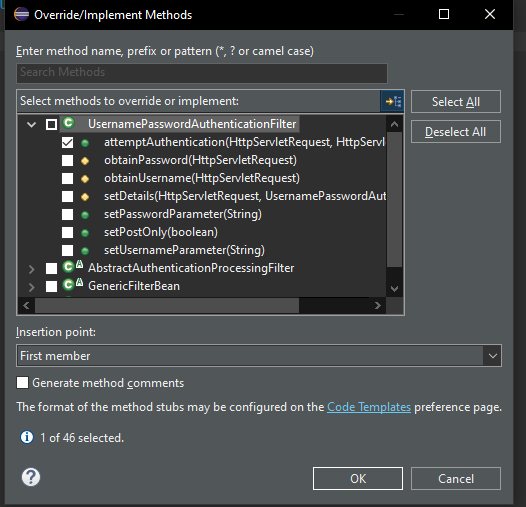
}

15 – Vamos a crear el paquete “security.filter” para crear los filtros necesarios para loguear.  
Dentro del paquete creamos la clase “JwtAuthenticationFilter” y la extendemos de UsernamePasswordAuthenticationFilter. Esto nos habilita a utilizar los métodos propios de dicha clase.

Apretamos “Ctrl+Alt+S” y elegímos la opción “Override” para podes sobreescribir un método propio de la clase extendida



Ahora eleccionamos el método “attemptAuthentication” y damos “OK”



Automáticamente se nos agregara el siguiente método. Este método manejara el intento de autenticación

*@Override*

public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)

throws AuthenticationException {

// **TODO** Auto-generated method stub

return super.attemptAuthentication(request, response);

}

También de la misma forma agregamos el método “successfulAuthentication” para la occasion en la que la autenticasión sea correcta. Dicho método se encuentra dentro de los métodos abstractos



*@Override*

protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain,

Authentication authResult) throws IOException, ServletException {

// **TODO** Auto-generated method stub

super.successfulAuthentication(request, response, chain, authResult);

}

Ahora vamos a modificar el método de “attemptAuthentication”:  
Por params vamos a recibir la request que va a encerrar los datos solicitados (en este caso “username” y “password”)  
Dentro del método creamos una UserEntity (recordemos que esta entidad está en models y es la que le da forma al usuario) y la igualamos a null. También creamos un String “username” y otro “password” (ambos vacíos).

UserEntity userEntity = null;

String username = "";

String password = "";

Utilizamos un tryCatch para intentar igualar la userEntity a la data recibida por request de la siguiente manera

try {

userEntity = new ObjectMapper().readValue(request.getInputStream(), UserEntity.class);

username = userEntity.getUsername();

password = userEntity.getPassword();

ObjectMapper es una librería propia de Spring que se llama “JACSON” y sirve para leer los JSON recibidos (El cuerpo del request llega en formato JSON y es por esto que utilizamos la librería para leer dicho JSON). Luego usamos el método readValue para leer el JSON y dentro de los params llamamos a request.getInputStream() (aquí está la info del request a modo de JSON) y luego hacer UserEntity.class() para transformer dicho JSON a una clase. (todo dentro del try)

userEntity = new ObjectMapper().readValue(request.getInputStream(), UserEntity.class);

Luego igualamos el username y el password creado a la info que encierra, ahora, userEntity.

username = userEntity.getUsername();

password = userEntity.getPassword();

Dado que puede haber errores al momento de recibir o comprobar la request agregaremos los siguientes catchs

} catch(StreamReadException e) {

throw new RuntimeException(e);

} catch (DatabindException e) {

throw new RuntimeException(e);

} catch (IOException e) {

throw new RuntimeException(e);

}

Dado que la clase “JwtAuthenticationFilter” está extendiendo de “UsernamePasswordAuthenticationFilter” podemos acceder a la clase UsernamePasswordAuthenticationToken que nos servirá para autenticarnos al momento de ingresar a la applicación (por params le tenemos que enviar el usuario y el password)

UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username, password);

Por último, modificamos el return y ponemos un “getAuthenticationManager” que es el encargado de autenticar y utilizamos el método “authenticate()” y lke pasamos por params el “authenticationToken” que acabamos de crear

return getAuthenticationManager().authenticate(authenticationToken);

El método complete queda así

*@Override*

public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)

throws AuthenticationException {

// **TODO** Auto-generated method stub

UserEntity userEntity = null;

String username = "";

String password = "";

try {

userEntity = new ObjectMapper().readValue(request.getInputStream(), UserEntity.class);

username = userEntity.getUsername();

password = userEntity.getPassword();

} catch(StreamReadException e) {

throw new RuntimeException(e);

} catch (DatabindException e) {

throw new RuntimeException(e);

} catch (IOException e) {

throw new RuntimeException(e);

}

UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username, password);

return getAuthenticationManager().authenticate(authenticationToken);

}

Ahora vamos a modificar el método “succesfulAthentication”

Creamos un User (este User viene de SpringFramework)



Y lo vamos a igualar al resultado del intento de autenticación del método anterior (esto llega por params y se llama “authResult”) de la siguiente manera

authResult.getPrincipal();

Va a ser necesario usar un Cast a “User” para que no tire error

User user = (User)authResult.getPrincipal();

Luego vamos a necesitar generar el token y para esto, primero importamos a esta clase la clase JwtUtils.  
En este caso, en vez de usar un bean vamos a recibir la jwtUtils desde el constructor de la clase

private JwtUtils jwUtils;

public JwtAuthenticationFilter(JwtUtils jwtUtils) {

this.jwUtils = jwtUtils;

}

Luego desde el método creamos un String (token) y lo igualamos al resultado inyectado y llamamos al método “generateAccessToken(user.getUsername())”

String token = this.jwUtils.generateAccessToken(user.getUsername());

Ahora vamos a articular el Response de la accion

Primero utilizamos el response que nos llegó por params y con el método addHeader(“Authorization”, token)” agregamos un cuerpo a la respuesta, este método nos deja agregarle multiples valores al response

response.addHeader("Authorization", token);

Ahora creamos un Map<String, Object> y lo igualamos a un HashMap para luego poder enviar la info (ya que está viajará en este formato)

Map<String, Object> httpResponse = new HashMap<>();

A dicho atributo le agregaremos primero un Token, en la siguiente línea un mensaje confirmando la autenticación y por ultimo el nombre de usuario al que se le ha generado el token de la siguiente manera (al ser información a la que se accede desde URL se utiliza la clase Map para que viaje y se utilizan los métodos PUT para enviar la información. También podría utilizar GET, DELETE o el que sea necesario)

httpResponse.put("token", token);

httpResponse.put("Message", "Correct Authentication");

httpResponse.put("Username", user.getUsername());

Ahora vamos a utilizar el método “getWriter()” para escribir la respuesta que interactuará con el usuario

response.getWriter().write(new ObjectMapper().writeValueAsString(httpResponse));

Hay que recordar que utilizamos la clase ObjectMapper para transformer la información de Map a JSON y luego el método “writeValueAsString(httpResponse)” para que se transforme dicha info a un String.

Luego en otra línea enviamos el Status “200”, en esté caso con un “OK”

response.setStatus(*HttpStatus*.***OK***.value());

Ahora indicamos el tipo de contenido (es un JSON)

response.setContentType(MediaType.***APPLICATION\_JSON\_VALUE***);

y por último usamos el método “flush()” propio del Writer para enviar toda la información

response.getWriter().flush();

El método complete queda así

*@Override*

protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain,

Authentication authResult) throws IOException, ServletException {

// **TODO** Auto-generated method stub

User user = (User)authResult.getPrincipal();

String token = this.jwUtils.generateAccessToken(user.getUsername());

response.addHeader("Authorization", token);

Map<String, Object> httpResponse = new HashMap<>();

httpResponse.put("token", token);

httpResponse.put("Message", "Correct Authentication");

httpResponse.put("Username", user.getUsername());

response.getWriter().write(new ObjectMapper().writeValueAsString(httpResponse));

response.setStatus(*HttpStatus*.***OK***.value());

response.setContentType(MediaType.***APPLICATION\_JSON\_VALUE***);

response.getWriter().flush();

super.successfulAuthentication(request, response, chain, authResult);

}

16 – En SecurityConfiguration comentamos el método “userDetailsService” dado que de momento estamos hardcodeando la información del usuario y lo que necesitamos es que compare la info que viene por Request contra la la BD.  
Vamos a implementar una clase personalizada de UserDetails para que haga esto.

Creamos un nuevo paquete llamado service y una clase (dentro del paquete) llamada UserDetailsServiceImp (imp de implements)



Dicha clase va a implementar a la interface UserDetailsService y con esta importamos el método que trae consigo. A la clase le agregamos la anotación @Service e importamos

*@Service*

public class UserDetailsServiceImp implements UserDetailsService{

*@Override*

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

// **TODO** Auto-generated method stub

return null;

}

}

Ahora declaramos un atributo del tipo UserRepository y le agregamos la anotación de @Autowired

*@Autowired*

private UserRepository userRepository;

Ahora en el método “loadUserByUsername” vamos a declarar un UserEntity (que es el que está en la BD) y lo vamos a igualar al return del método “findByUsername()" de la clase UserRepository (es el atributo declarado que es inyectado). En caso de que no lo encuentre, va a tirar un “UsernameNotFoundException”

*@Override*

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

UserEntity userEntity

= this.userRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException("User " + username + " not found"));

Lo que sigue es modificar el return en el que enviaremos un new “User” (este user es el importado de SpringFramework), para formar el mismo necesitaremos varios parametros



Primero necesitamos el nombre y el password, para esto tenemos los getters, luego necesitamos 4 booleans:  
- “enabled” usuario active : true  
-“accountNonExpired” que la cuenta no expire : true  
-“credentialsNonExpired” que las credenciales no expiren : true  
-“accountNonLocked” usuario no bloqueado : true

Por último necesitamos una Colección de “GrantedAuthority” (el rol autorizado) por lo que deberemos declarar dicho atributo dentro del método

Declaramos una Collection y entre los diamantes pondremos que sea igual a cualquier cosa que extienda de “GrantedAuthority”. A dicha coleccion la igualamos al resultado de userEntity.getRoles()

Collection<? extends GrantedAuthority> authorities = userEntity.getRoles()

Luego le agregamos un .stream() para que nos devuelva los valores de los roles existentes.  
Después ponemos un .map() para transformer la info en un map y utilizamos una function lambda para asignar los valores a un new SimpleGrantedAuthority(String) (para dicho String vamos a asignar siempre al principio la cadena “ROLE\_” luego concatenamos el role.getName().name())  
Por último usamos el método “.collect(Collectors.toSet())” para tratar la info cómo un “Set” dado que este no permite que se repita Código (a diferencia de una “list”)

Collection<? extends GrantedAuthority> authorities = userEntity.getRoles()

.stream()

.map(role -> new SimpleGrantedAuthority("ROLE\_".concat(role.getName().name())))

.collect(Collectors.*toSet*());

Por último agregamos dicha colección al final del return del User.

El método queda así

*@Override*

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

UserEntity userEntity = this.userRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException("User " + username + " not found"));

Collection<? extends GrantedAuthority> authorities = userEntity.getRoles()

.stream()

.map(role -> new SimpleGrantedAuthority("ROLE\_".concat(role.getName().name())))

.collect(Collectors.*toSet*());

return new User(userEntity.getUsername(),

userEntity.getPassword(),

true,

true,

true,

true,

authorities);

}

Ahora volvemos a la clase SecurityConfiguration y declaramos un atributo de la instancia de UserDetailsServiceImp (con un @Autowired), teniendo en cuenta que para crear dicha clase necesitamos recibir por params un JwtUtils, por lo que también instanciamos dicha clase y esta sí va a recibir un bean de @Autowired.

*@Autowired*

JwtUtils jwtUtils;

@Autowired

UserDetailsServiceImp userDetailsServiceImp;

Lo próximo es modificar el método authenticationManager y en el return, en la parte de .userDetailsService, ponemos en los params la instancia de UserDetailsServiceImp.

.userDetailsService(this.userDetailsServiceImp)

Ahora vamos al método securityFilterChain y creamos una instancia de JwtAuthenticationFilter que recibe por params el jwtUtils instanciado en la clase (inyectado)

JwtAuthenticationFilter jwtAuthenticationFilter = new JwtAuthenticationFilter(this.jwtUtils);

Luego utilizamos esta instancia y llamamos al método “setAuthenticationManager()” el cual necesita recibir una instancia de dicha clase para actuar (ya la tenemos declarada en la clase con el método del mismo nombre). Para esto, modificaremos los params del método y le agregamos una instancia de dicha clase

SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity, AuthenticationManager authenticationManager) throws Exception {

Ahora usamos está instancia que llega por params para completar el metodo setAuthenticationManager

jwtAuthenticationFilter.setAuthenticationManager(authenticationManager);

Lo siguiente es eliminar el .httpBasic() del return y cambiarlo por .addFilter() al que le pasaremos por params el filtro declarado en el mismo método

.addFilter(jwtAuthenticationFilter)

18 – Si bien este filtro ya está bien definido, lamentablemente todavía no podemos probarlo ya que el password de la BD no está encriptado, por lo que haremos lo siguiente:

Primero iremos a la clase main de SpringSecurityJwtApplication y fuera del main escribiremos un CommandLineRunner (esto se hace para que siempre que inicie la app se corra este Código y creemos un usuario) llamado init() y con una anotacion @Bean

*@Bean*

CommandLineRunner init(){

Ahora directamente en el return haremos una function lambda en la que crearemos un UserEntity (hay que tener en cuenta que el UserEntity es la clase que encierra al usuario tal cual aparece en la base de datos y está en el paquete models). El userEntity creado debe ser igualado a un USerEntity.builder() (es decir que iremos creando parte por parte al usuario).

return args -> {

UserEntity userEntity = UserEntity.*builder*()

Primero agregamos el email, luego el username, luego el password, el cual debe estar encriptado, es po resto que creamos (también fuera del main y fuera de este método) un PasswordEncoder con una anotacion @Autowired.

*@Autowired*

PasswordEncoder passwordEncoder;

Luego lo utilizamos dentro del builder para hacer un encode hardcodeado del password

.email("s@s.com")

.username("pepe")

.password(passwordEncoder.encode("1234"))

Por último tenemos que indicarle el rol, primero llamaremos a Set (los roles estan dentro de un Set dado que esto no permite datos iguales), luego usaremos el método of() para indicar que solo retornaremos un rol de dicho set; entre los params enviaremos un RoleEntity.builder() para crear el role n este momento (hardcodeado) y a name le asignaremos el ERole.valuOf (recordar que son un enum) “ERole.ADMIN.Name” y luego usaremos el build() para crear este rol. Por ultimo volveremos a llamar al método build() para cerrar la creacion hardcodeada de este UserEntity

*@Bean*

CommandLineRunner init(){

return args -> {

UserEntity userEntity = UserEntity.*builder*()

.email("s@s.com")

.username("pepe")

.password(passwordEncoder.encode("1234"))

.roles(Set.*of*(RoleEntity.*builder*()

.name(*ERole*.*valueOf*(*ERole*.***ADMIN***.name()))

.build()))

.build();

};

}

Por último grabaremos instanciaremos fuera del método (y del main) un UserRepository (este es el que interactua con la BD) con la anotación @Autowired

*@Autowired*

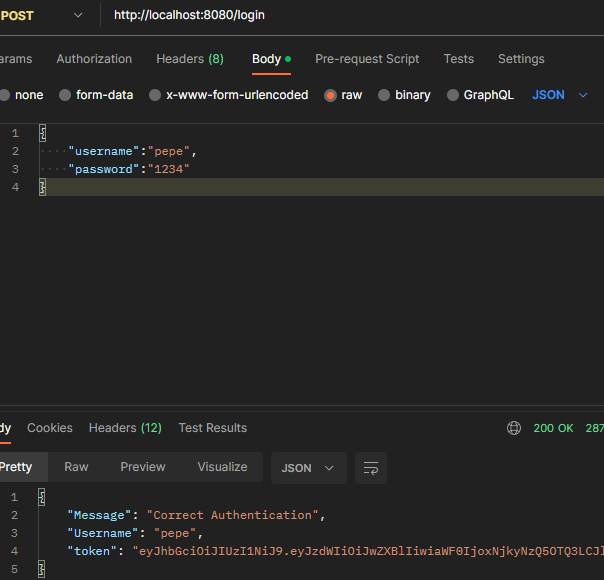
UserRepository userRepository;

Y, nuevamente dentro del método, al finalizar el build() del userEntity, llamaremos al userRepository y utilizaremos el método save() al que le enviaremos por params al userEntity creado.

userRepository.save(userEntity);

A modo de prueba crearemos 2 usuarios mas y los grabaremos de la misma forma

Ahora si vamos a Postman y enviamos un método post a la Url “/login” en el que haya un body raw en formato JSON con los datos de alguno de los usuarios creados al inicar la app, nos dara una respuesta status 200



19 – Si bien el filtro está funcionando, ahora debemos implementar un filtro que valide el token (hasta ahora solo estamos validando las credenciales del usuario)  
Vamos a crear un nuevo filtro (clase) dentro del paquete “filters” llamado “JwtAuthorizationFilter” y lo vamos a extender de la clase “OncePerRequestFilter” que nos obligará a siempre validar el token.  
También le agregamos la anotación @Component

*@Component*

public class JwtAuthorizationFilter extends OncePerRequestFilter {

Implementamos el método obligatorio de la clase (dado que es una clase abstracta)

*@Override*

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

throws ServletException, IOException {

Lo primero a hacer es controlar que los params no ingresen siendo null (y en caso de que así lo sea se rechaza automaticamente la petición)  
Para esto se le va a agregar la anotcación @NonNull a todos los params

(*@NonNull* HttpServletRequest request, *@NonNull* HttpServletResponse response, *@NonNull* FilterChain filterChain)

Ahora vamos a necesitar las instancias de JwtUtils y UserDetailsServiceImpl, ambos con la anotación @Autowired

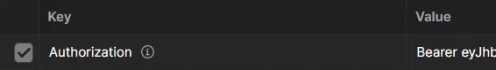
Volviendo al método, lo primero es declarar un String llamado tokenHeader y lo igualamos a la request (que llega por params) y de dicha request extraemos el header mediante el método getHeader(“Authorization”) (es muy importante por params enviar la cadena “Authorization” para que no rompa la app, dado que el header que estamos RECUPERANDO es el Authorization)



String tokenHeader = request.getHeader("Authorization");

Ahora debemos validar si el token es distinto de null (si es null se rechaza la conexion) y si el token empieza con “Bearer “ (dado que este siempre va a empezar con la cadena Bearer y un espacio)

(Para ver el valor, es necesario en la solapa de Autorizacion, cambiar a la opción “bearer token” y pegar el token encriptado)



if(tokenHeader != null && tokenHeader.startsWith("Bearer ")) {

Lo siguiente es quitar del tokenHeader la palabra “Bearer ” y el espacio, por lo que declaramos un nuevo String llamado token y utilizando el método “substring(7) “ removemos la parte no deseada (el método también cuenta con una sobrecarga para decidir hasta donde queremos que recorte, en este caso podría ser tokenheader.lenght() pero dado que si no aplicamos la sobrecarga, el método directamente toma desde lo indicado hasta el final de la cadena). Tenemos en cuenta que por params se declara 7 dado que si contamos la cadena desde el inicio “Bearer “ ocupa 7 caracteres (empezando desde 0 y tomando en cuenta el espacio) y finaliza en la posición 6, por lo que nosotros vamos a obtener el token desde la posicion 7

String token = tokenHeader.substring(7);

Ahora debemos validar si el token es válido, para esto utilizamos el método isTokenValid de la clase jwtUtils

if(this.jwtUtils.isTokeValid(token)) {

Si el token es válido, entonces debemos recuperar el usuario que viene dentro del token. Para eso, creamos un String llamado username y lo igualamos al resultado del método “getUsernameFromToken” de la clase jwtUtils (por params enviamos el token)

String username = this.jwtUtils.getUserNameFromToken(token);

Ahora que tenemos el usuario, necesitamos recuperar los detalles del usuario (password y otros atributos) para esto declaramos una variable de la interaz UserDetails y la instanciamos como un userDetailsService.loadUserByUsername (username)

UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

(estamos usando el UserDetailsServiceImpl declarado en la clase para utilizar el método que ira a la BD a recuperar los datos del usuario y nos devolvera un User propio de SpringSecurity con las autorizaciones y demas atributos)

Una vez que tenemos este usuario, vamos a autenticarnos declarando una variable de la clase “UsernamePAsswordAuthenticationToken”

Para instanciar esta clase tenemos que enviarle por params primero el username, luego ponemos null (esto es por la contraseña que SpringSecurity recupera del userDetails) y por último las autorizaciones; para esto usamos el método getAuthorities de userDetails

UsernamePasswordAuthenticationToken authenticationToken =

new UsernamePasswordAuthenticationToken(username, null, userDetails.getAuthorities());

Ahora debemos llamar al SecurityContextHolder que es la clase que contiene la auntenticacion propia en la aplicación.

Lo que debemos hacer es guardar la autenticacion que hicimos en el SecurityContextHolder, por lo que usamos el método getContext()setAuthentication y por params le enviamos el “authenticacionToken” que acabamos de crear

SecurityContextHolder.*getContext*().setAuthentication(authenticationToken);

Por último debemos especificar que pasa si el método no accede al primer if (esto podría ser porque el token es null o la autorizacion no empieza con “Bearer “).  
En caso de que esto suceda, vamos a usar el “filterChain” que recibimos por params y vamos a utilizar el método “doFilter”y le enviamos la request y la response como params

filterChain.doFilter(request, response);

Esto hara que SpringSecurity se de cuenta que no tiene un token de acceso (o uno válido) y denegará el acceso

Ahora lo que debemos hacer es agregar el filtro creado al paquete “security”.  
Creamos una instancia de JwtAuthorizationFilter con una anotación de @Autowired y en el método securityFilterChain previo al build final agregamos el método “addFilterBefore”, esto agregará un filtro y marcará que se ejecute antes que otro cierto filtro, en este caso quedaría así

.addFilterBefore(this.jwtAuthorizationFilter, UsernamePasswordAuthenticationFilter.class)

Lo que hace esto es que se ejecute el filtro recién creado antes que el filtro JwtAuthenticationFilter que habiamos creado previamente

JwtAuthenticationFilter jwtAuthenticationFilter = new JwtAuthenticationFilter(this.jwtUtils);

(en definitive, lo primero que se va a corroborar al momento de autenticar, es si el usuario tiene un token válido)

20 – Para finalizar, vamos a restringir el acceso a los EndPoints críticos (por ejemplo, el createUser) permitiendo el acceso unicamente a los ADMINS. Para esto vamos a ir al paquete “controller” y crear la clase “TestRolesController”; a la misma le agregamos la anotacion de @RestController

*@RestController*

public class TestRolesController {

Ahora vamos a crear tres endPoints con una URL para cada tipo de Rol respesctivamente (tener en cuenta que se utiliza el método GET en estos EP)

*@GetMapping*("/accessAdmin")

public String accesAdmin() {

return "You have logged in as an ADMIN";

}

*@GetMapping*("/accessUser")

public String accesUser() {

return "You have logged in as an USER";

}

*@GetMapping*("/accessGuest")

public String accessGuest() {

return "You have logged in as an GUEST";

}

Ahora vamos a la clase SecurityConfiguration y a la clase le agregamos la anotación “EnableMethodSecurity(prePostEnable=true)”

*@Configuration*

*@EnableMethodSecurity*(prePostEnabled = true)

public class SecurityConfiguration {

Esto hará que en los endpoints podamos directamente establecer que tipo de rol tiene acceso al mismo

Volvemos a TestRolController y debajo del endpoint agregamos la anotacion @PreAuthorize y entre parentesis le enviamos el “hasRole(‘ADMIN’)” en caso de que queramos que solo accedan los admins, en caso de que queramos que varios roles accedan podemos poner “hasRole(‘ROL’) OR hasRole(‘ROL’)”, también tenemos la opcion de usar el método “hasAnyRole(‘ROL1’, ‘ROL2’, ‘ROL3’, ETC)

*@GetMapping*("/accessAdmin")

*@PreAuthorize*("hasRole('ADMIN')")

public String accesAdmin() {

return "You have logged in as an ADMIN";

}

*@GetMapping*("/accessUser")

*@PreAuthorize*("hasRole('USER') OR hasRole('ADMIN')")

public String accesUser() {

return "You have logged in as an USER";

}

*@GetMapping*("/accessGuest")

*@PreAuthorize*("hasAnyRole('GUEST', 'USER', 'ADMIN')")

public String accessGuest() {

return "You have logged in as an GUEST";

}

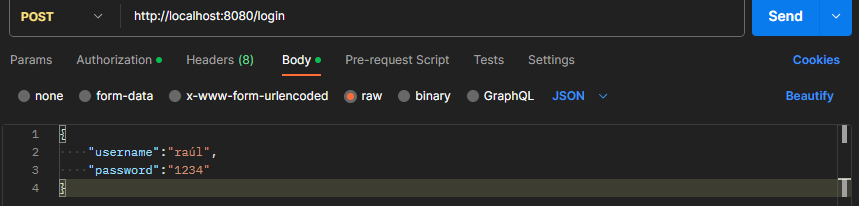
Una vez comprobado el funcionamiento de dichas autorizaciones hacia los endpoints podemos utilizar el PreAuthorize para limitar el acceso a los endPoints (por ejemplo en el createUser)

*@PostMapping*("/createUser")

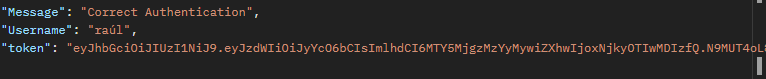
*@PreAuthorize*("hasRole('ADMIN')")

public ResponseEntity<?> createUser(*@Valid* *@RequestBody* CreateUserDTO createUserDTO){

Para corroborar el funcionamiento debemos ir al PostMan y primero hacer login

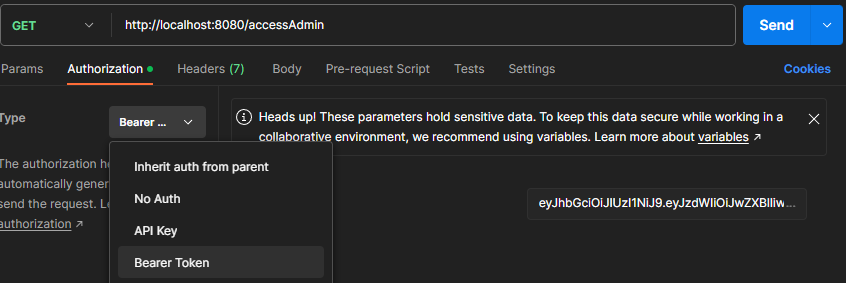


Una vez autenticado el login, nos devolverá el token necesario para acceder a los endPoints



Lo siguiente es enviar una peticion Post a una de las URL (endpoint) con acceso limitado. En este caso /accessAdmin (en este caso, el usuario con el que logueamos tiene ROL de GUEST, por lo que no podrá acceder a dicho ep)

En Postman en la parte de Authorization seleccionamos Bearer Token y copiamos el token

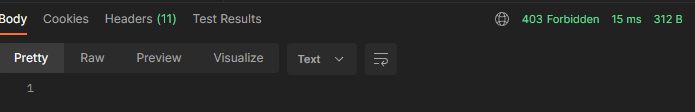




Token

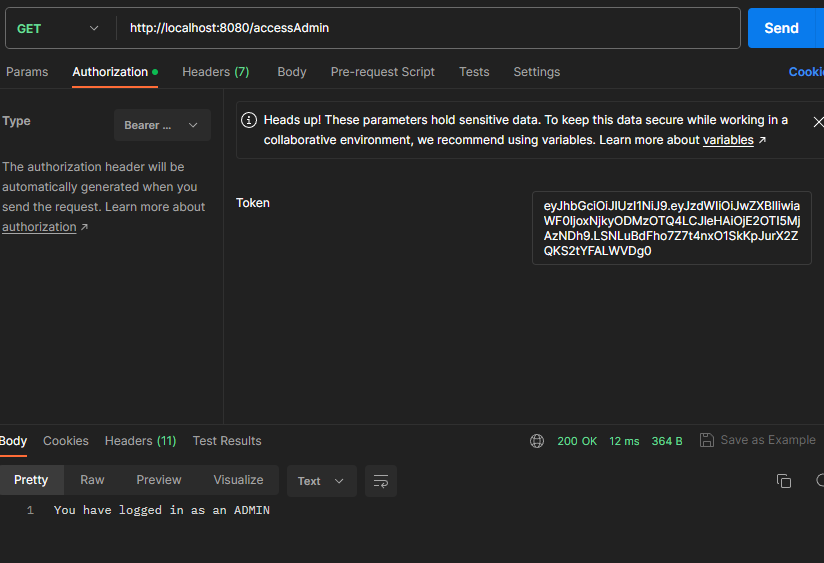
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyYcO6bCIsImlhdCI6MTY5MjgzMzYyMywiZXhwIjoxNjkyOTIwMDIzfQ.N9MUT4oL821wtRspa05Gj45Fo6Ec8-Nz5AKApojBulc

Cuando enviemos la petición, la misma será rechazada debido a la falta de permisos de rol para acceder



Ahora vamos a probar logueando con “pepe” que es el usuario con rol de ADMIN y vamos a repetir el proceso de loguear, copiar el token, ir a la peticion GET con el EP /accessAdmin y enviar el token nuevo

El resultado sera STATUS 200



Si hacemos la misma prueba con el EP createUser el resultado será el mismo (si es que le agregamos la anotacion @PreAuthorize)

FIN