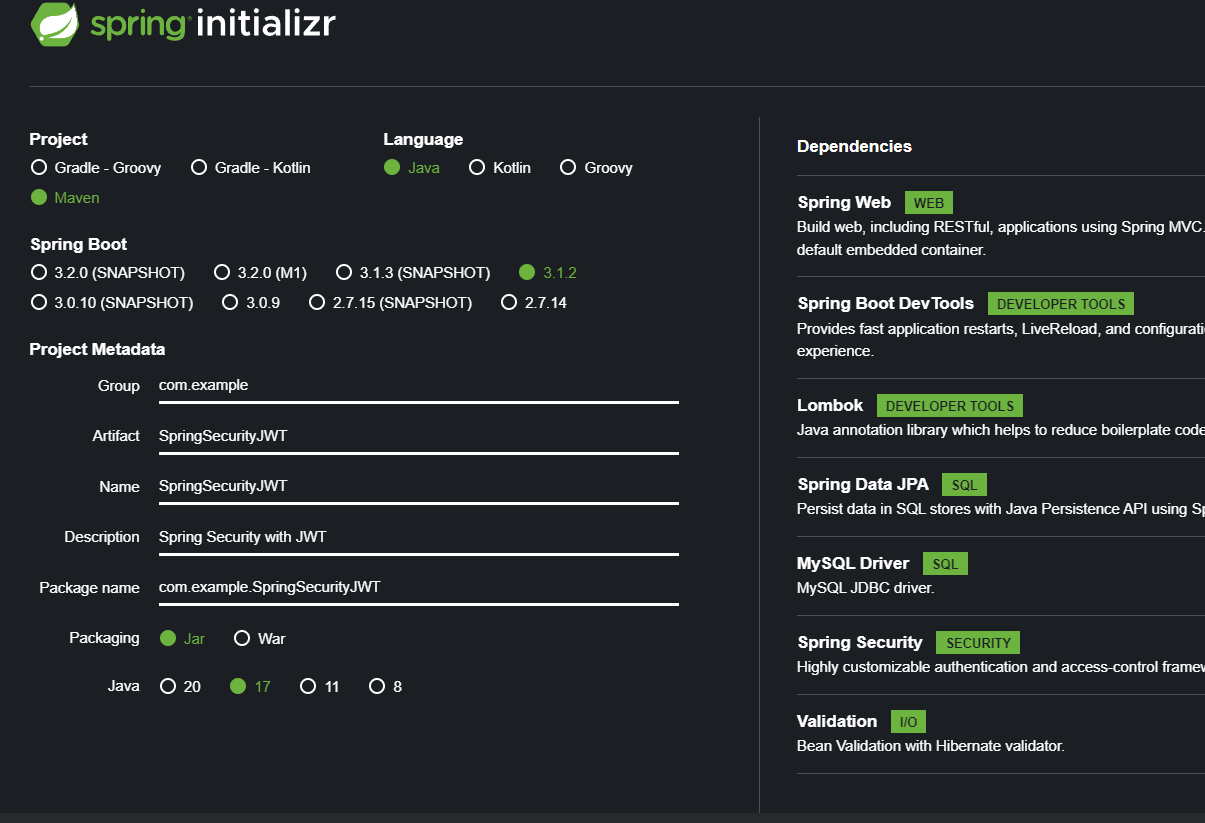
Spring Security with JWT

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

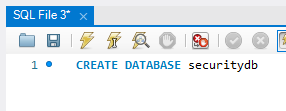
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

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

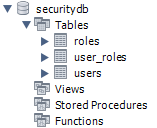
*@JoinTable*(name = "user\_roles", joinColumns = *@JoinColumn*(name = "user\_id")

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";

}

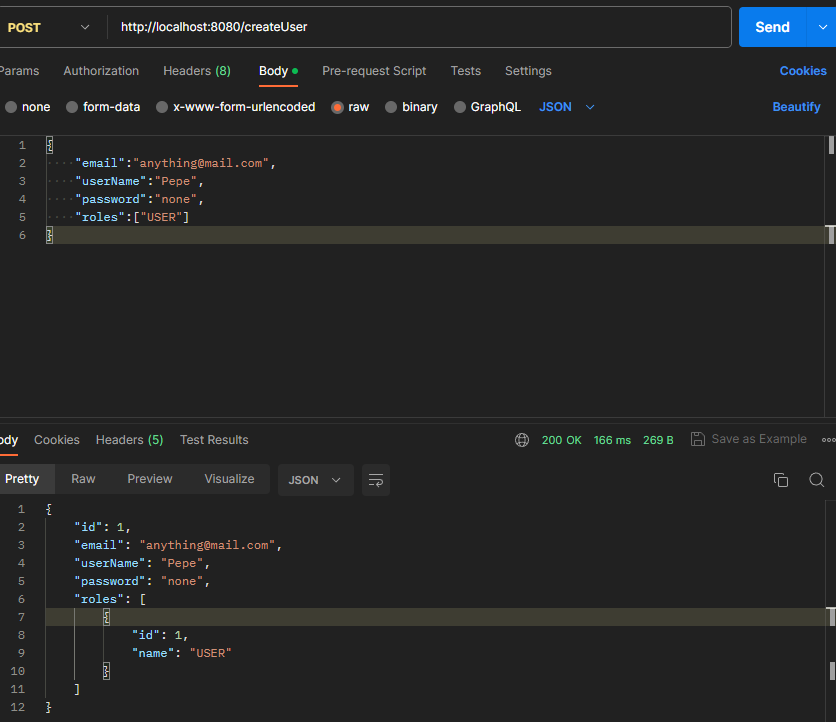
Ahora podemos poner a prueba si la app sigue funcionando y vamos a utilizar POSTMAN para enviar por header el body para crear el usuario

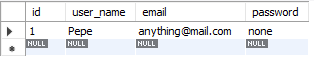
A tener en cuenta:

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)

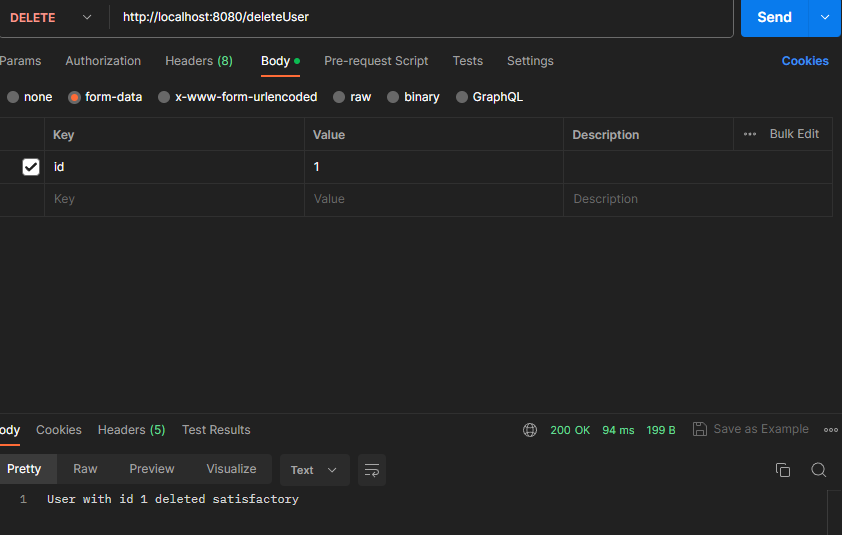
Luego de la creacion del usuario podemos corroborar en la DB si se guardó el nuevo usuario y a su vez la creación del ROL “USER” debido al cascade.PERSIST

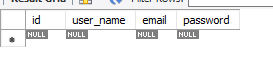






Ahora ponemos a prueba el metodo delete y usando Postman vamos al endpoint deseado  
Esta vez enviamos por BODY>FORM-DATA” el id 1 (o el id del usuario deseado).  
Al revisar la DB debería haber sido borrado el usuario pero NO ASÍ el rol, ya que por el Cascade.PERSIST el rol debería persistir pese al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

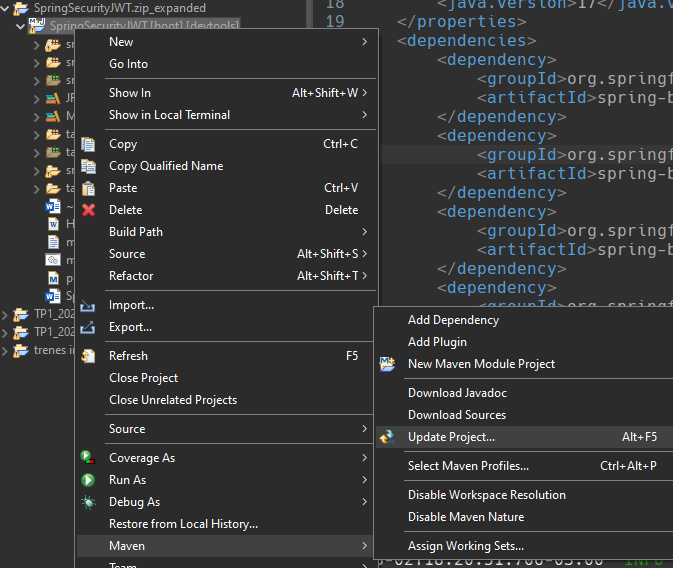
<dependency>

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

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

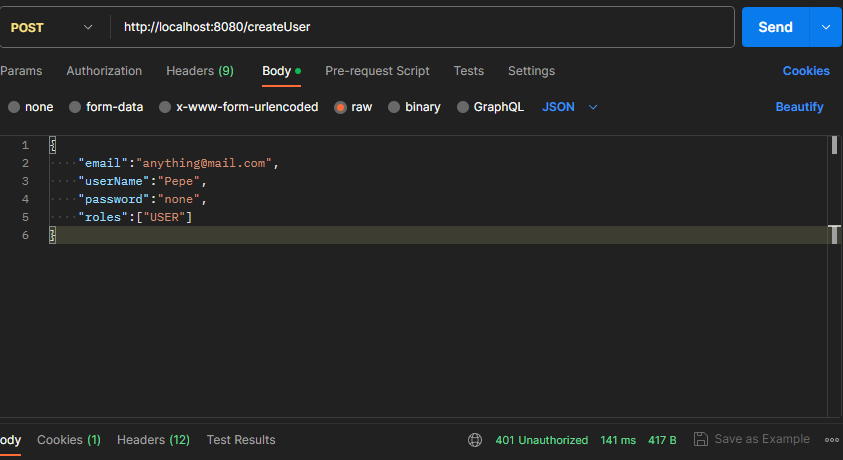
</dependency>

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)

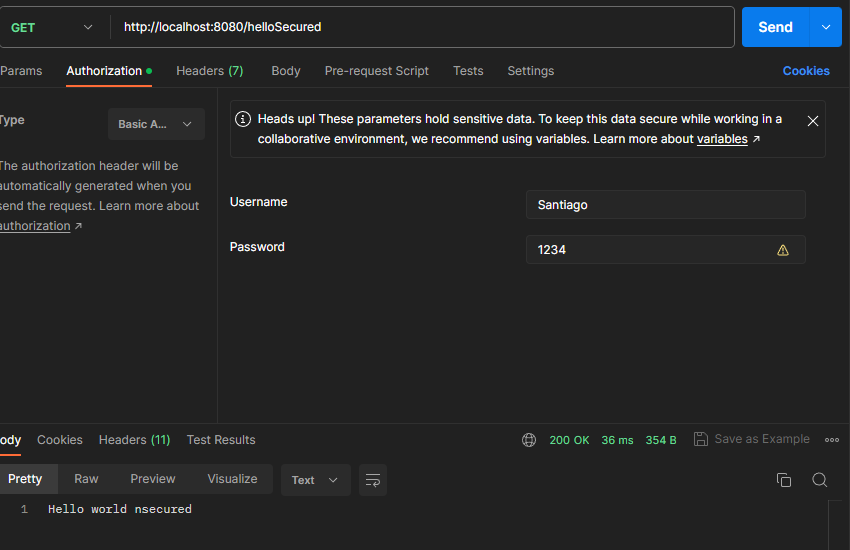
.userDetailsService(userDetailsService())

.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>

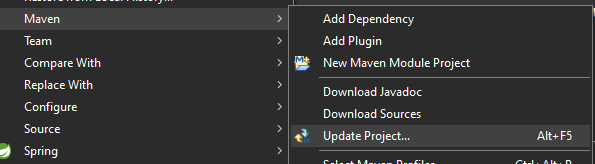
<groupId>io.jsonwebtoken</groupId>

<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()));

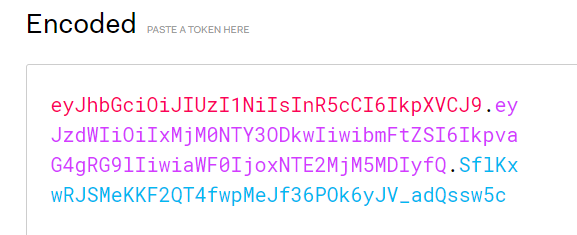
return false;

}

}

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

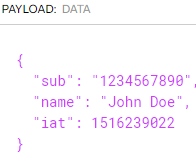


En sí, el token está compuesto por 3 partes

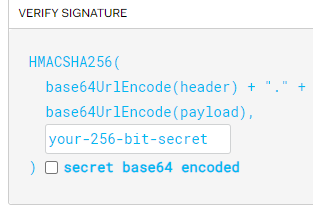
La primer parte (color rojo) es el header, ahí viaja el algoritmo en el que está encriptado y el tipo de token



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);

}