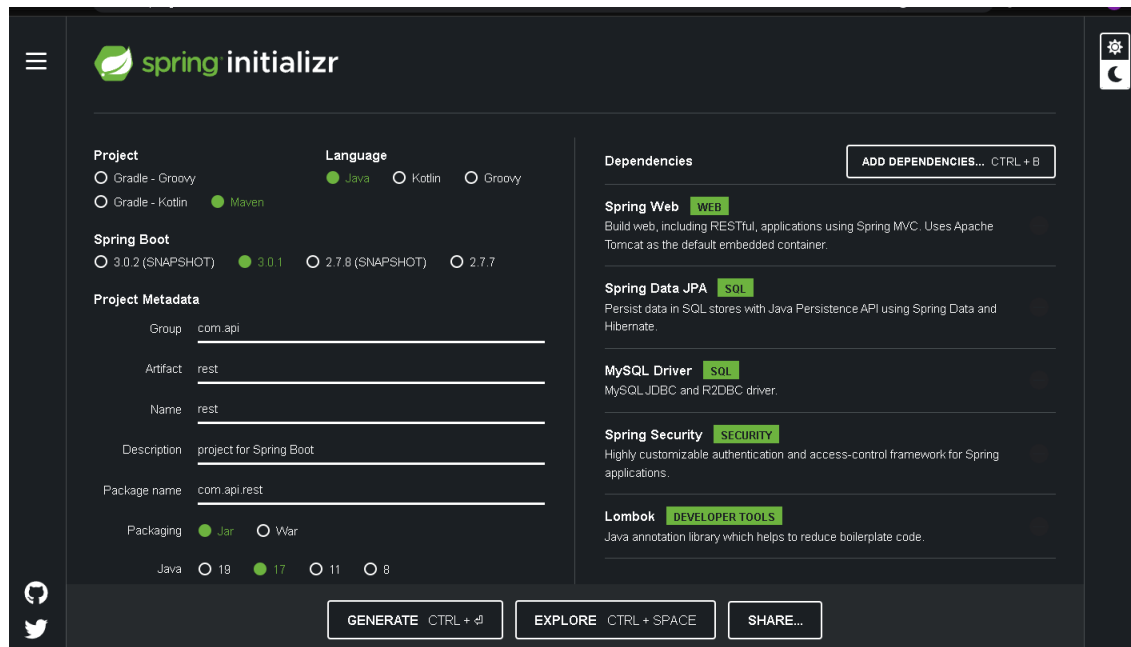


JWT 2023, SPRINGBOOT, TUTORIAL ESPAÑOL:

Esta guía está basada en el video “Spring Boot 3 + Spring Security 6 - JWT Authentication and Authorisation [NEW] [2023]” del canal “AmigosCode”

1er Paso:

Crear un proyecto SpringBoot con el cual trabajar desde la página spring boot initializr:

The screenshot shows the Spring Initializr web application interface. It has a dark theme. On the left, there's a sidebar with a hamburger menu icon and a settings gear icon. The main area is divided into sections: 'Project' with radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', and 'Maven' (selected); 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for '3.0.2 (SNAPSHOT)', '3.0.1' (selected), '2.7.8 (SNAPSHOT)', and '2.7.7'; 'Project Metadata' with input fields for 'Group' (com.api), 'Artifact' (rest), 'Name' (rest), 'Description' (project for Spring Boot), and 'Package name' (com.api.rest); 'Packaging' with radio buttons for 'Jar' (selected) and 'War'; and 'Java' with radio buttons for '19', '17' (selected), '11', and '8'. On the right, there's a 'Dependencies' section with a button 'ADD DEPENDENCIES... CTRL + B'. Below this, there are several dependency cards: 'Spring Web' (WEB) with a description, 'Spring Data JPA' (SQL) with a description, 'MySQL Driver' (SQL) with a description, 'Spring Security' (SECURITY) with a description, and 'Lombok' (DEVELOPER TOOLS) with a description. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. There are also social media icons in the bottom left corner.

Si ya cuenta con uno, asegúrese de tener las dependencias necesarias.

- Spring Web.
- Spring Data JPA.
- MySQL Driver
- Spring Security
- Lombok

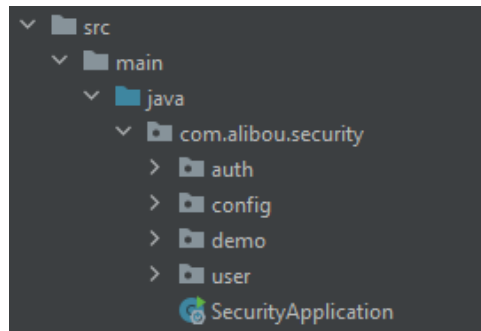
Mas adelante se agregarán las dependencias de JWT.

2do paso: Realizar la conexión a la base de datos, mediante el archivo application.properties.

```
application.properties x
1  spring.jpa.hibernate.ddl-auto=update
2  spring.datasource.url=jdbc:mysql://localhost:3306/restapi
3  spring.datasource.username=root
4  spring.datasource.password=
5  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

3er paso:

Crear una estructura de trabajo, mediante la organización en paquetes: Aquí un ejemplo



Otra organización podría ser:

- Entity
- Service
- Repository
- Controller
- Security

Esto dependerá de como quiera organizar su proyecto.

4to paso:

En el paquete **user** debemos crear todo lo relacionado al usuario:

- La clase User que se verá reflejada como una tabla en nuestra base de datos.
- Un objeto de tipo Enum, que nos permita identificar nuestros roles.
- Un UserRepository, de tipo interfaz, que nos permite agregar un método que utilizaremos más adelante.

Clase user: utilizar etiqueta **@Entity**, **lombok** e implementar **UserDetails**.

```

User.java x
1 package com.alibou.security.user;
2
3 import ...
18
4 usages alibouali
19 @Data
20 @Builder
21 @NoArgsConstructor
22 @AllArgsConstructor
23 @Entity
24 @Table(name = "user")
25 public class User implements UserDetails {
26
27 no usages
28 @Id
29 @GeneratedValue
30 private Integer id;
31 no usages
32 private String firstname;
33 no usages
34 private String lastname;
35 1 usage
36 private String email;
37 1 usage
38 private String password;
39 1 usage
40 @Enumerated(EnumType.STRING)
41 private Role role;
42
43 @Override
44 public Collection<? extends GrantedAuthority> getAuthorities() {
45     return List.of(new SimpleGrantedAuthority(role.name()));
46 }
47
48 alibouali
49 @Override
50 public String getPassword() { return password; }
51
52 alibouali
53 @Override
54 public String getUsername() { return email; }
55
56 no usages alibouali
57 @Override
58 public boolean isAccountNonExpired() { return true; }
59
60 no usages alibouali
61 @Override
62 public boolean isAccountNonLocked() { return true; }
63
64 no usages alibouali
65 @Override
66 public boolean isCredentialsNonExpired() { return true; }
67
68 alibouali
69 @Override
70 public boolean isEnabled() { return true; }
71 }

```

Dato: ¿Qué es UserDetails?

Es una interfaz que nos brinda SpringSecurity, esta proporciona información básica del usuario.

Spring Security no utiliza directamente las implementaciones por motivos de seguridad. Simplemente almacenan información del usuario que luego se encapsula en Authentication objetos. Esto permite que la información del usuario no relacionada con la seguridad (como direcciones de correo electrónico, números de teléfono, etc.) se almacene en una ubicación conveniente.

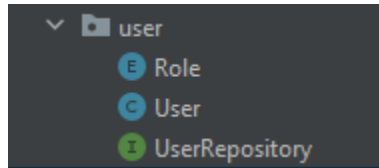
Objeto Role de tipo enum

```
1 package com.alibou.security.user;  
2  
3 public enum Role {  
4  
5     USER,  
6     ADMIN  
7 }
```

UserRepository interface, extiende de JPA, este nos permite obtener funcionalidad como `save()`, `findAll()`, etc.

```
1 package com.alibou.security.user;  
2  
3 import ...  
4  
5  
6 public interface UserRepository extends JpaRepository<User, Integer> {  
7  
8     Optional<User> findByEmail(String email);  
9  
10 }
```

Luego de la creación de lo anteriormente mencionado, nuestra carpeta user, debería quedar de la siguiente manera:



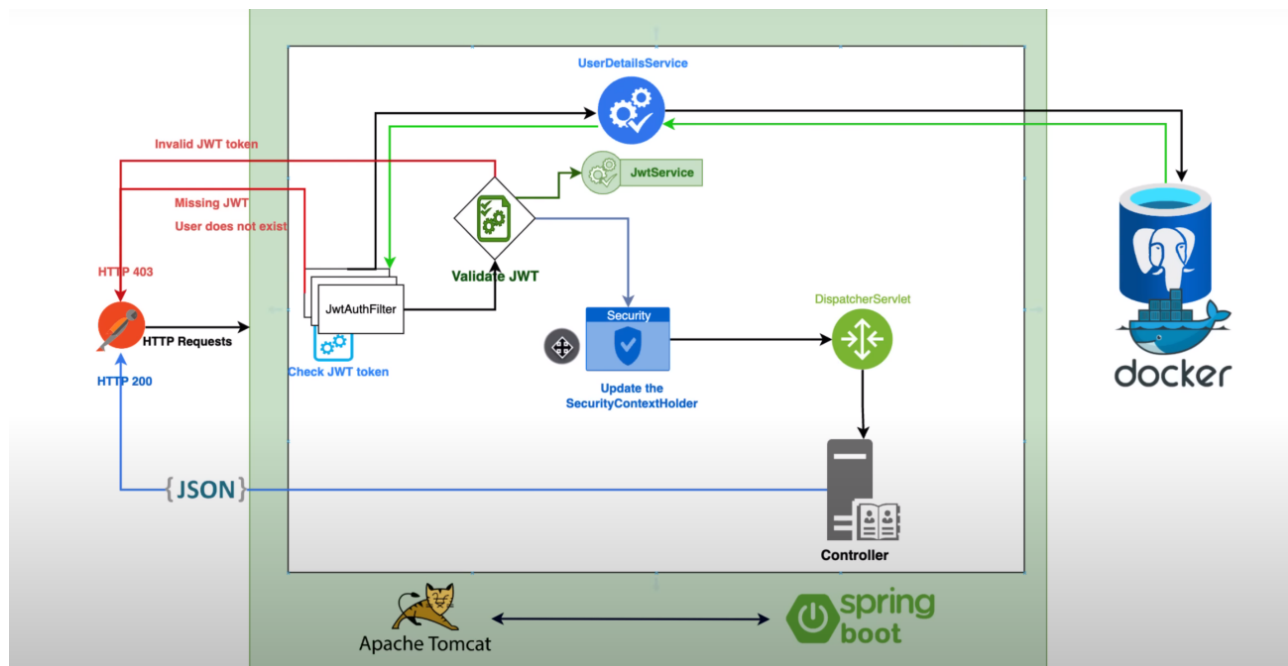
5^{to} paso: Agregar “jjwt-api”, “jjwt-impl”, “jjwt-jackson” como dependencias a nuestro proyecto.

```
<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>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
</dependency>
```

Con las mismas podremos comenzar con la implementación de JWT, pero primero debemos entender su funcionamiento:

En el siguiente diagrama se muestra el proceso de ejecución que tiene JWT:

- Se realiza una llamada a la API, mediante peticiones HTTP.
- Se ejecuta una vez por cada solicitud el `JwtAuthenticationFilter`, tiene la funcionalidad de validar y verificar todo lo relacionado con el token. Algunas de las funcionalidades que realizará será verificar si en la base de datos mediante `UserDetailsService` existe el usuario que está realizando la petición.
- Si el token existe, se ejecuta `JwtService`, que se encarga de validar el token.
- Si el token es válido, se actualizará el security context holder, indicando que el usuario esta ahora actualmente conectado.
- Cuando todo este proceso termine, el usuario podrá obtener las respuesta de la API REST, mediante el controller.



Con estos principios fundamentales podemos comenzar con la implementación.

6^{to} paso: en la carpeta **config** crearemos la clase JwtAuthenticationFilter:

```

1  package com.alibou.security.config;
2
3  import ...
18
19  @Component
20  @RequiredArgsConstructor
21  public class JwtAuthenticationFilter extends OncePerRequestFilter {
22

```

Como pueden observar la clase extiende de OncePerRequestFilter (Indica que el filtro se ejecuta una vez por cada solicitud).

También tiene sus etiquetas @Component, y @RequiredArgsConstructor

Dato:

@RequiredArgsConstructor genera un constructor con 1 parámetro para cada campo que requiere un manejo especial. Todos los **final** campos no inicializados obtienen un parámetro, así como los campos que están marcados como **@NotNull** que no se inicializan donde se declaran. Para aquellos campos marcados con **@NotNull**, también se genera una verificación nula explícita. El constructor arrojará un **NullPointerException** si alguno de los parámetros destinados a los campos marcados con **@NotNull** contiene **null**. El orden de los parámetros coincide con el orden en que aparecen los campos en su clase.

```
JwtAuthenticationFilter.java x
1 package com.alibou.security.config;
2
3 import ...
18
19 @Component
20 @RequiredArgsConstructor
21 public class JwtAuthenticationFilter extends OncePerRequestFilter {
22
23     private final JwtService jwtService;
24     private final UserDetailsService userDetailsService;
```

Como vimos en el diagrama, el filtro involucrara tanto a JwtService como UserDetailsService, el servicio debemos crearlo más adelante, el segundo es un servicio propio de SpringSecurity.

Método doFilterInternal:

```
@Override
protected void doFilterInternal(
    @NonNull HttpServletRequest request,
    @NonNull HttpServletResponse response,
    @NonNull FilterChain filterChain
) throws ServletException, IOException {
    final String authHeader = request.getHeader("Authorization");
    final String jwt;
    final String userEmail;
    if (authHeader == null || !authHeader.startsWith("Bearer ")) {
        filterChain.doFilter(request, response);
        return;
    }
}
```

Es un método que se hereda de OncePerRequestFilter, tiene como parámetros: request, response, filterchain.

doFilterInternal: obtiene el header, pregunta si esta vacío o si no comienza con “Bearer ”, si esto da verdadero, le dice a la cadena de filtros que no continúe con el proceso y retorne un error.

Si el primer check da false, continua con la siguiente secuencia:

- Obtener el token del header
- Obtener el email/username del usuario (dependiendo lo que declaro en la tabla User y el repository) --- Esto lo hace mediante el servicio que crearemos más adelante
- Valida que el email no sea nulo y que no esté autenticado.
- Carga al usuario mediante el userDetailsService.
- Pregunta al JwtService si el token es valido
- Autentica al usuario.
- Llama a la cadena de filtrado para que siga su proceso.

```
jwt = authHeader.substring( beginIndex: 7);
userEmail = jwtService.extractUsername(jwt);
if (userEmail != null && SecurityContextHolder.getContext().getAuthentication() == null) {
    UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);
    if (jwtService.isTokenValid(jwt, userDetails)) {
        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
            userDetails,
            credentials: null,
            userDetails.getAuthorities()
        );
        authToken.setDetails(
            new WebAuthenticationDetailsSource().buildDetails(request)
        );
        SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}
filterChain.doFilter(request, response);
}
```

Con esto hemos terminado el JwtAuthenticationFilter.

7^{mo} paso: es hora de continuar y crear nuestro JwtService. (Podemos crearlo en la misma carpeta de config o en una carpeta de Service)

```
@Service
public class JwtService {

    1 usage
    private static final String SECRET_KEY = "404E635266556A586E3272357538782F413F4428472B4B6250645367566B5970";
```

Definimos nuestra SECRET_KEY: pueden obtener la suya personalizada entrando al siguiente link:

<https://www.allkeysgenerator.com/>

All keys generator

The all-in-one ultimate online toolbox that generates all kind of keys ! Every coder needs [All Keys Generator](#) in its favorites !
It is provided for free and only supported by ads and donations.

[Donate](#)

[GUID](#) [MachineKey](#) [WPA Key](#) [WEP Key](#) [Encryption key](#) [Password](#)

Security level

[64-bit](#) [128-bit](#) [256-bit](#) [512-bit](#) [1024-bit](#) [2048-bit](#) [4096-bit](#)

Hex ?

☒ Yes

How many ?

1

Result

3777217A25432A462D4A404E635266556A586E3272357538782F413F4428472B

Con este servicio debemos poder generar el token, validarlo, preguntar si esta expirado o no, extraer el usuario, decodificar nuestra secret_key, etc.

extraer el usuario:

```
public String extractUsername(String token) { return extractClaim(token, Claims::getSubject); }
```

Generar el token:

```
public String generateToken(
    Map<String, Object> extraClaims,
    UserDetails userDetails
) {
    return Jwts
        .builder()
        .setClaims(extraClaims)
        .setSubject(userDetails.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 24))
        .signWith(getSignInKey(), SignatureAlgorithm.HS256)
        .compact();
}
```

Preguntar si es válido:

```
public boolean isValidToken(String token, UserDetails userDetails) {
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);
}
```

Preguntar si expiro:

```
private boolean isTokenExpired(String token) { return extractExpiration(token).before(new Date()); }
```

Extraer datos:

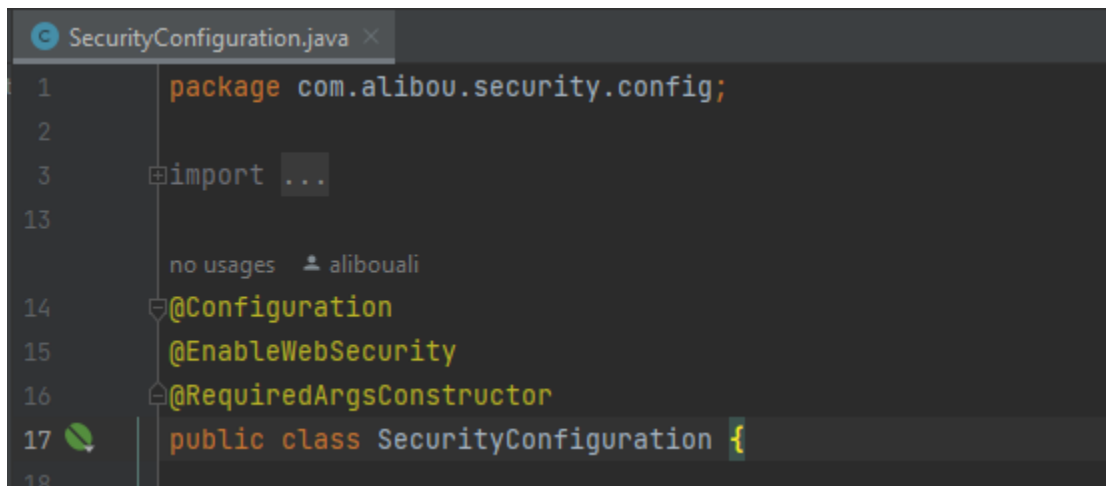
```
private Claims extractAllClaims(String token) {  
    return Jwts  
        .parserBuilder()  
        .setSigningKey(getSignInKey())  
        .build()  
        .parseClaimsJws(token)  
        .getBody();  
}
```

Decodificar la SECRET_KEY:

```
private Key getSignInKey() {  
    byte[] keyBytes = Decoders.BASE64.decode(SECRET_KEY);  
    return Keys.hmacShaKeyFor(keyBytes);  
}
```

Con estos métodos tendríamos completado nuestro JwtService.

8º paso: Configurar nuestro archivo de seguridad global. El mismo puede ser creado en el propio archivo config.



```
SecurityConfiguration.java x  
1 package com.alibou.security.config;  
2  
3 import ...  
13  
14 @Configuration  
15 @EnableWebSecurity  
16 @RequiredArgsConstructor  
17 public class SecurityConfiguration {  
18
```

Necesitaremos las siguientes etiquetas @Configuration @EnableWebSecurity @RequiredArgsConstructor.

También debemos declarar nuestro filtro y el proveedor de autenticación que aun no lo hemos creado.

```
private final JwtAuthenticationFilter jwtAuthFilter;
1 usage
private final AuthenticationProvider authenticationProvider;
```

Por último, inyectaremos un @Bean con el método SecurityFilterChain, asegurarse de que este método sea público, ya que los Bean tienen la característica de que no pueden ser de tipo privado.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf() .csrfConfigurer<HttpSecurity>
        .disable() HttpSecurity
        .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
        .requestMatchers( ...patterns: "/api/v1/auth/**") AuthorizeHttpRequestsConfigurer<...>.AuthorizedUrl
        .permitAll() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
        .anyRequest() AuthorizeHttpRequestsConfigurer<...>.AuthorizedUrl
        .authenticated() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
        .and() HttpSecurity
        .sessionManagement() SessionManagementConfigurer<HttpSecurity>
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and() HttpSecurity
        .authenticationProvider(authenticationProvider)
        .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

En estas líneas de código se expresa lo siguiente: Deshabilitar el csrf, declarar la “White list” (todas aquellas rutas que se pueden acceder sin estar autenticado, lo habitual son aquellas rutas de registro y login), luego para las demás rutas se declara que se necesita autorización, y se crea la sesión y por último se declara el filtro.

Con esto tendremos configurado nuestra seguridad global.

9º paso: Crear una clase ApplicationConfig, para declarar todos aquellos métodos requeridos para el manejo y encriptado de datos.

```

@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {

    1 usage
    private final UserRepository repository;

    1 usage  alibouali
    @Bean
    public UserDetailsService userDetailsService() {
        return username -> repository.findByEmail(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
    }

    no usages  alibouali
    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService());
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

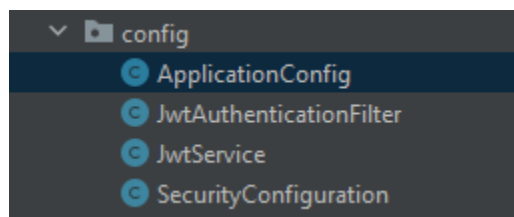
    no usages  alibouali
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    1 usage  alibouali
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }

}

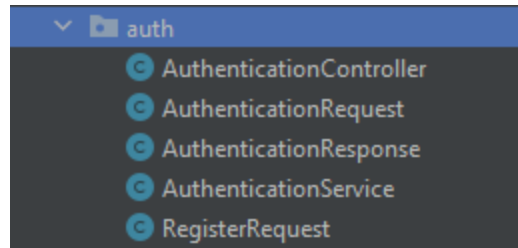
```

Finalizado todos estos pasos la carpeta config, les debería quedar de la siguiente manera:



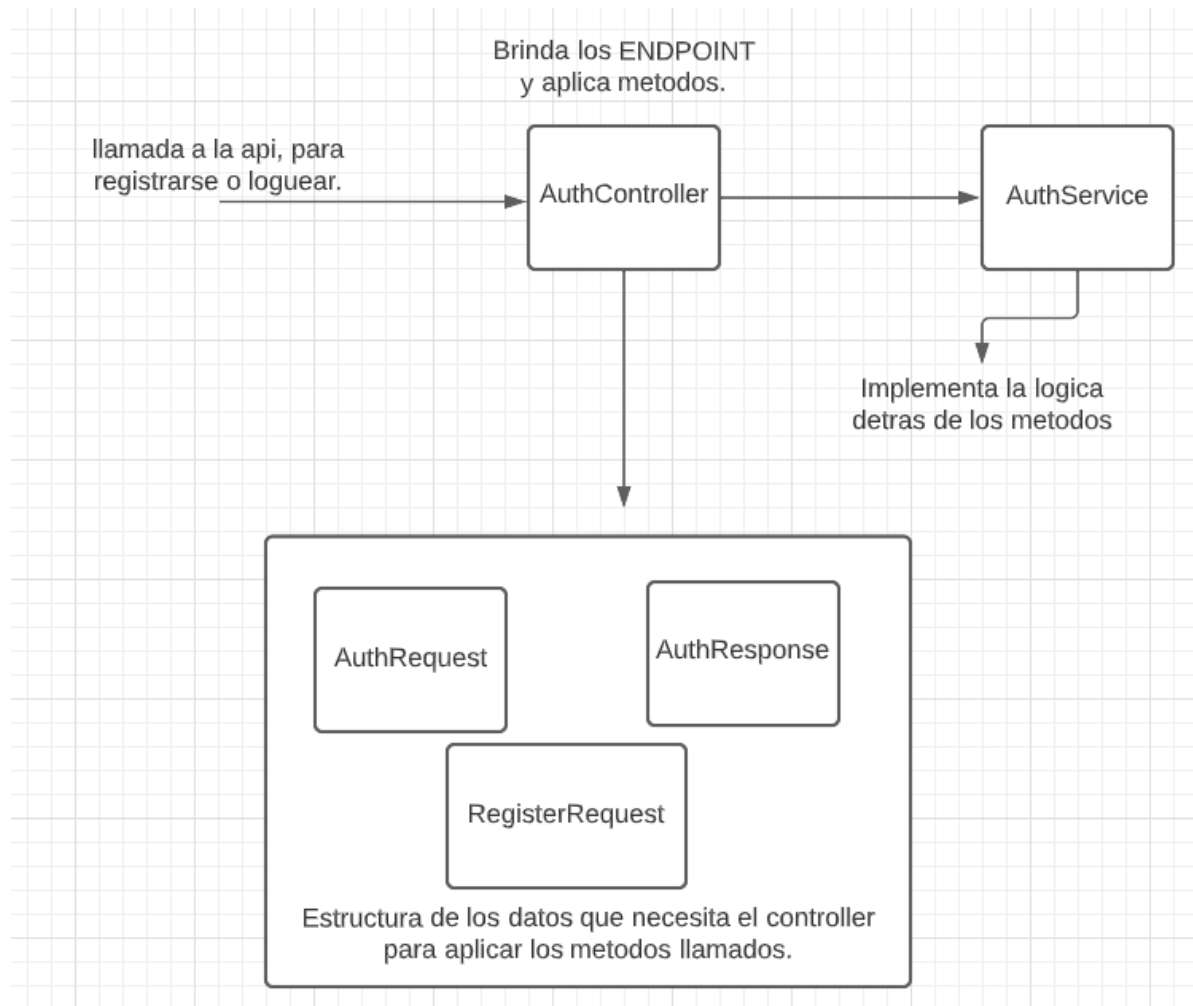
10^{mo} paso: Como ultimo proceso debemos realizar el controlador, el cual contendrá los métodos para llamar a los endpoint, como la creación de usuario y el login.

Al finalizar este proceso deberían obtener una carpeta como la siguiente:

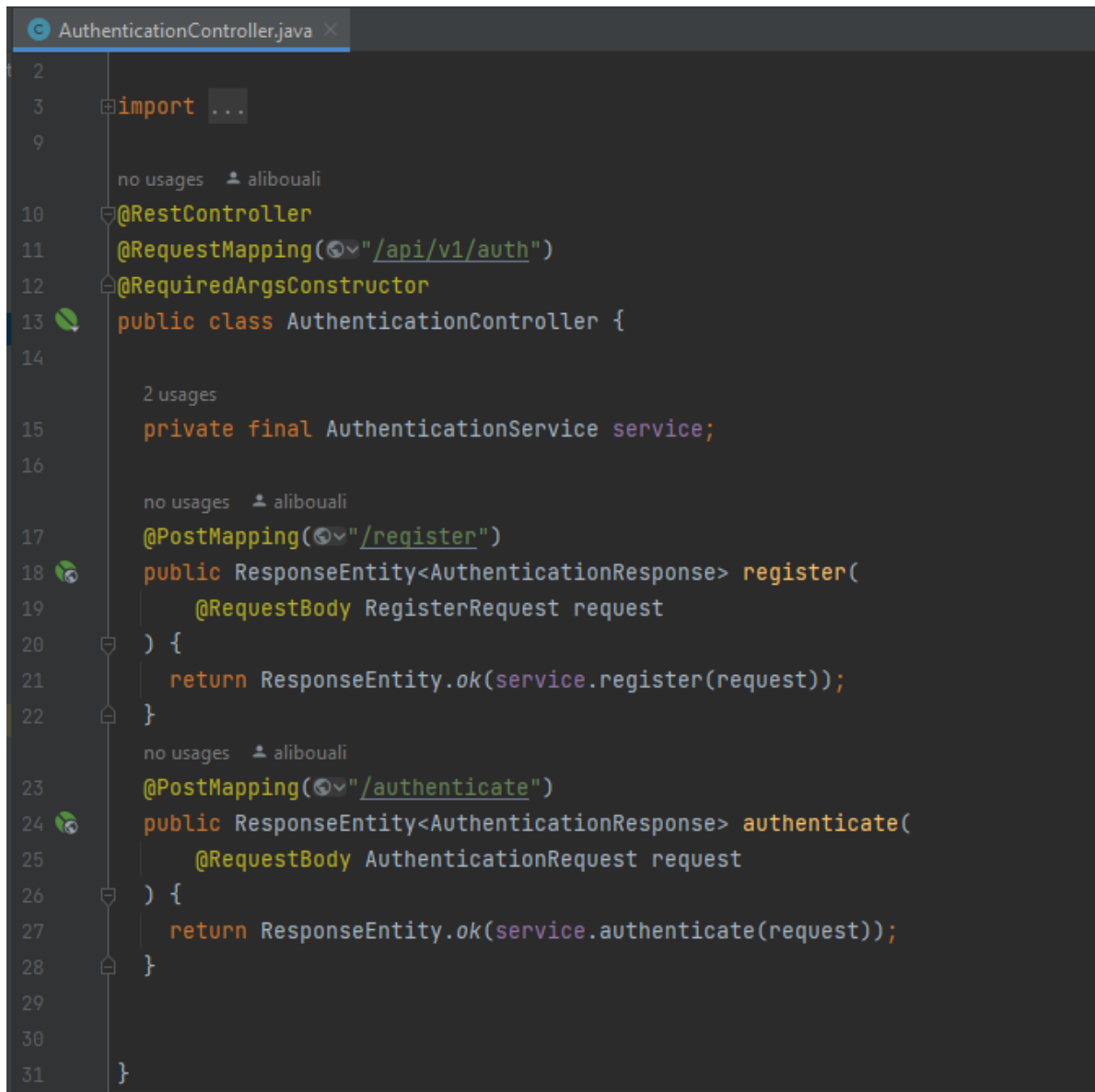


En este ejemplo se llama auth, pero podrían etiquetarla como Controller.

Lógica detrás de esta carpeta:



Primero realizamos el controlador:



```
1 2
3  import ...
9
10 no usages  alibouali
11 @RestController
12 @RequestMapping("/api/v1/auth")
13 @RequiredArgsConstructor
14 public class AuthenticationController {
15
16     2 usages
17     private final AuthenticationService service;
18
19     no usages  alibouali
20     @PostMapping("/register")
21     public ResponseEntity<AuthenticationResponse> register(
22         @RequestBody RegisterRequest request
23     ) {
24         return ResponseEntity.ok(service.register(request));
25     }
26
27     no usages  alibouali
28     @PostMapping("/authenticate")
29     public ResponseEntity<AuthenticationResponse> authenticate(
30         @RequestBody AuthenticationRequest request
31     ) {
32         return ResponseEntity.ok(service.authenticate(request));
33     }
34 }
```

Controlador principal.

Luego la estructura de datos:

```
AuthenticationRequest.java x
1 package com.alibou.security.auth;
2
3 import ...
7
8 @Data
9 @Builder
10 @AllArgsConstructor
11 @NoArgsConstructor
12 public class AuthenticationRequest {
13
14     no usages
15     private String email;
16     no usages
17     String password;
18 }
```

Datos necesarios para el login. Que se deben pasar en el @RequestBody

```
AuthenticationResponse.java x
1 package com.alibou.security.auth;
2
3 import ...
7
8 @Data
9 @Builder
10 @AllArgsConstructor
11 @NoArgsConstructor
12 public class AuthenticationResponse {
13
14     no usages
15     private String token;
16 }
```

Debe retornar el token una vez logueado.

```

1  package com.alibou.security.auth;
2
3  import ...
7
   2 usages  alibouali
8  @Data
9  @Builder
10 @AllArgsConstructor
11 @NoArgsConstructor
12 public class RegisterRequest {
13
   no usages
14     private String firstname;
   no usages
15     private String lastname;
   no usages
16     private String email;
   no usages
17     private String password;
18 }

```

Datos necesarios para el registro. Que se deben pasar por el @RequestBody.

Ahora por último debemos crear el servicio que maneja toda la lógica de los métodos utilizados en el controlador principal.


```

1 package com.alibou.security.auth;
2
3 import ...
4
12
13 @Service
14 @RequiredArgsConstructor
15 public class AuthenticationService {
16     private final UserRepository repository;
17     private final PasswordEncoder passwordEncoder;
18     private final JwtService jwtService;
19     private final AuthenticationManager authenticationManager;
20
21 @ public AuthenticationResponse register(RegisterRequest request) {
22     var user = User.builder()
23         .firstname(request.getFirstname())
24         .lastname(request.getLastname())
25         .email(request.getEmail())
26         .password(passwordEncoder.encode(request.getPassword()))
27         .role(Role.USER)
28         .build();
29     repository.save(user);
30     var jwtToken = jwtService.generateToken(user);
31     return AuthenticationResponse.builder()
32         .token(jwtToken)
33         .build();
34 }

```

En el método register, podemos implementar el Role.ADMIN para crear un nuevo usuario que luego pueda tener mayor privilegio que los demás. Como el de modificar datos y que los usuarios comunes solo sean capaces de leerlos.

```

1 usage alibouali
2
3 public AuthenticationResponse authenticate(AuthenticationRequest request) {
4     authenticationManager.authenticate(
5         new UsernamePasswordAuthenticationToken(
6             request.getEmail(),
7             request.getPassword()
8         )
9     );
10    var user = repository.findByEmail(request.getEmail())
11        .orElseThrow();
12    var jwtToken = jwtService.generateToken(user);
13    return AuthenticationResponse.builder()
14        .token(jwtToken)
15        .build();
16 }

```

Finalmente, para probar nuestra implementación podemos utilizar Postman, para realizar llamadas a la API REST.

¡Espero que les sirva esta información, saludos!

BF 😊