

Passos realizados na aula de Spring – Parte 2 – Security

Aula 01

Configuração inicial do Spring Security.

Incluir as dependências no pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Executar o projeto, acessar os endpoints e ver o resultado.

Criar a classe SecurityConfig:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private Environment env;

    private static final String[] PUBLIC_MATCHERS = {
        "/h2-console/**"
    };

    private static final String[] PUBLIC_MATCHERS_GET = {
        "/categorias/**"
    };

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.headers().frameOptions().disable(); //just for test profile
        http.cors().and().csrf().disable();
        http.authorizeRequests()
            .antMatchers(PUBLIC_MATCHERS).permitAll()
            .antMatchers(HttpMethod.GET, PUBLIC_MATCHERS_GET).permitAll()
            .anyRequest().authenticated();

        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS
    }
```

```
);  
}
```

Aula 02

Adicionando senha a Cliente

Criar o atributo 'senha' na classe Cliente (atualizar o construtor, criar set e get):

```
@JsonIgnore  
private String senha;
```

Criar o atributo 'senha' na classe ClienteNewDTO (criar set e get):

```
@NotEmpty(message="Preenchimento obrigatório")  
private String senha;
```

Salvando perfis de usuário na base de dados

Adicionar a URL de cliente na classe SecurityConfig:

```
private static final String[] PUBLIC_MATCHERS_GET = {  
    "/produtos/**",  
    "/categorias/**",  
    "/clientes/**"  
};
```

Criar o enum Perfil.

```
public enum Perfil {  
  
    ADMIN(1, "ROLE_ADMIN"),  
    CLIENT(2, "ROLE_CUSTOMER");  
  
    private int cod; //criar apenas get  
    private String descricao; //criar apenas get  
  
    private Perfil(int cod, String descricao) {  
        this.cod = cod;  
        this.descricao = descricao;  
    }  
    public static Perfil toEnum(Integer cod) { //implementar...  
}
```

Incluir os seguintes elementos na classe cliente:

```
@ElementCollection(fetch=FetchType.EAGER)
```

```
@CollectionTable(name="PERFIS")
private Set<Integer> perfis = new HashSet<>();
```

Adicionar a seguinte linha nos 2 construtores:

```
addPerfil(Perfil.CLIENTE);
```

Adicionar 1 cliente do Application como sendo ADMIN:

```
cli2.addPerfil(Perfil.ADMIN);
```

Aula 02 (extra)

Configurar o Lombok e utilizar as anotações nos beans.

Aula 03 (autenticação)

Parte 1 - Implementando autenticação e geração do token JWT

Criar a classe **UserPrincipal** (pronta): analisar seu conteúdo

Criar a classe **UserDetailsServiceImpl** (pronta): analisar seu conteúdo

Criar a classe **CredentialsDTO** (criar set, get e construtor default):

```
@NoArgsConstructor
@AllArgsConstructor
@Setter
@Getter
public class CredentialsDTO implements Serializable {
    private String email;
    private String password;
```

Parte 2 - Geração do token JWT

Incluir os seguintes atributos no arquivo application.properties

```
jwt.secret=SequenciaDeCaracteresParaAssinarToken
jwt.expiration.time=300000
```

Criar a classe **LoginError**:

```
@Getter
public class LoginError extends StandardError {

    String error;
    String path;

    public LoginError(Integer status, String error, String message, String path) {
```

```

super(message, status, new Date());
this.error = error;
this.path = path;
}

```

Criar a classe **UserService** (pronta): analisar seu conteúdo

Criar a classe **JWTUtil** (pronta): analisar seu conteúdo

Implementar a classe **JWTAuthenticationFilter** (pronta): analisar seu conteúdo

Adicionar os seguintes itens na classe **SecurityConfig**:

```

@Autowired
private UserDetailsService userDetailsService;
@Autowired
private JWTUtil jwtUtil;

```

Adicionar a seguinte linha no método `configure(...)`:

```

http.addFilter(new JWTAuthenticationFilter(authenticationManager(), jwtUtil));

```

Criar o seguinte método:

```

@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}

```

Configurar o `passwordEncoder` na classe **SecurityConfig**:

```

@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}

```

Injetar o `passwordEncoder` na classe `Application/run`

```

@Autowired
BCryptPasswordEncoder

```

Criptografar a senha do cliente de teste.

Parte 3 – Testar a aplicação

Fazer a chamada para o endpoint de login, usando as seguintes informações

<http://localhost:8080/login>

POST

Body:

```
{
  "email": "c1@gmail.com",
  "password": 12345
}
```

Verificar o status code e o header Authorization de resposta.

Aula 05 (autorização)

Criar a classe JWTAuthorizationFilter (pronta): analisar seu conteúdo

Descomentar os métodos na classe **JWTUtil**.

Autorizando endpoints para perfis específicos

Adicionar a seguinte anotação na classe SecurityConfig:

```
@EnableGlobalMethodSecurity(prePostEnabled = true) //anotação da classe
```

Injetar o userDetailsService:

```
@Autowired
private UserDetailsService userDetailsService;
```

Comentar o vetor de String de URL's de POST: PUBLIC_POST_URLS

Adicionar o filtro de autorização no método configure:

```
http.addFilter(new JWTAuthorizationFilter(authenticationManager(),
jwtUtil, userDetailsService));
```

Configurar os endpoints de CategoryController e CustomerController para autorizar o acesso de cada perfil.

Incluir a autorização de ROLE_ADMIN nos métodos de POST, PUT e DELETE de categorias:

```
@PreAuthorize("hasAnyRole('ROLE_ADMIN')")
```

Incluir a autorização de ROLE_CUSTOMER no método de GET do cliente:

```
@PreAuthorize("hasAnyRole('ROLE_CUSTOMER')")
```

Fazer o teste de acesso aos endpoints usando token:

- 1) Fazer a chamada GET no endpoint /login com perfil de ADMIN para obter o token
- 2) Fazer a chamada POST no endpoint /categories usando o token do passo anterior.
- 3) Fazer outras chamadas nos outros endpoints e verificar o resultado.
Atenção: caso o token expirar, é necessário gerar outro token válido.
- 4) Fazer outros testes de chamada os endpoints usando o perfil de CUSTOMER.

Aula 06

Restrição de conteúdo: cliente só recupera ele mesmo

Criar a classe **AuthorizationException**:

```
public class AuthorizationException extends RuntimeException {  
    public AuthorizationException(String message) {  
        super(message);  
    }  
}
```

Tratar a exceção na classe ResourceExceptionHandler:

```
@ExceptionHandler(AuthorizationException.class)  
public ResponseEntity<StandardError>  
authorization(AuthorizationException e, HttpServletRequest request) {  
  
    StandardError err = new  
StandardError(HttpStatus.FORBIDDEN.value(), e.getMessage(),  
System.currentTimeMillis());  
    return ResponseEntity.status(HttpStatus.FORBIDDEN).body(err);  
}
```

Criar o método na classe UserSS:

```
public boolean hasRole(Perfil perfil) {  
    return getAuthorities().contains(new  
SimpleGrantedAuthority(perfil.getDescricao()));  
}
```

Atualizar o método find da classe ClienteService:

```
public Cliente find(Integer id) {  
  
    UserSS user = UserService.authenticated();  
    if (user==null || !user.hasRole(Perfil.ADMIN) && !  
id.equals(user.getId())) {  
        throw new AuthorizationException("Acesso negado");  
    }  
}
```

```
}
```

Aula 07

Restrição de conteúdo: cliente só recupera seus pedidos

Criar o método na interface PedidoRepository:

```
@Transactional(readonly=true)
Page<Pedido> findByCliente(Cliente cliente, Pageable pageRequest);
```

Atualizar o método de busca de pedidos para fazer o filtro do cliente autenticado:

```
UserSS user = UserService.authenticated();
if (user == null) {
    throw new AuthorizationException("Acesso negado");
}
Cliente cliente = clienteService.find(user.getId());
return repo.findByCliente(cliente);
```

Refresh token

Implementar a classe AuthController:

```
@RestController
@RequestMapping(value = "/auth")
public class AuthController {

    @Autowired
    private JWTUtil jwtUtil;

    @RequestMapping(value = "/refresh_token", method = RequestMethod.POST)
    public ResponseEntity<Void> refreshToken(HttpServletResponse response) {
        UserSS user = UserService.authenticated();
        String token = jwtUtil.generateToken(user.getUsername());
        response.addHeader("Authorization", "Bearer " + token);
        return ResponseEntity.noContent().build();
    }
}
```

Adicionar a URL para permitir o acesso ao /auth na classe SecurityConfig

```
private static final String[] PUBLIC_MATCHERS_POST = {
    "/clientes/**",
```

```
        "/auth/**"  
    };
```

Aula 08 (extra)

Criptografar a senha:

Injetar o objeto de criptografia nas classes CustomerService e no Application:

```
@Autowired  
private BCryptPasswordEncoder pe;
```

Incluir a chamada para codificar a senha quando instanciar o cliente:

```
pe.encode("senha_do_cliente")
```

Aula 09 – refresh token (extra)

Criar o endpoint para gerar um token novo na classe **AuthenticatorController** (pronta):
analisar seu conteúdo.

```
@RestController  
@RequestMapping("/auth")  
public class AuthenticatorController {  
  
    @Autowired  
    private JWTUtil jwtUtil;  
  
    @PostMapping("/refresh_token")  
    @ResponseStatus(HttpStatus.NO_CONTENT)  
    public void refreshToken(HttpServletResponse response) {  
        UserPrincipal user = UserService.getAuthenticated();  
        String token = jwtUtil.generateToken(user.getUsername());  
        UserService.setAuthorizationInHeader(response, token);  
    }  
}
```

Aula 10 – esqueci a senha (extra)

Exercício: pensar em como pode ser implementado o: Esqueci a senha