Segurança II

Controlando acesso a Endpoints

Alguns ***endpoints não precisam de uma restrição de segurança***, métodos **GET por exemplo**, onde eu só estou **devolvendo informações para leitura**, **sem nenhum dado sensível é claro**. Eu posso liberar o acesso a **eles** no método configure que recebe **HttpSecurity:**

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers(HttpMethod.GET, ...antPatterns: "/topicos").permitAll()
            .antMatchers(HttpMethod.GET, ...antPatterns: "/topicos/*").permitAll();

}
```

Eu posso restringir o acesso aos demais endpoints dizendo que **qualquer outra requisição é necessária estar autenticado:**

```
    http.authorizeRequests()
            .antMatchers(HttpMethod.GET, ...antPatterns: "/topicos").permitAll()
            .antMatchers(HttpMethod.GET, ...antPatterns: "/topicos/*").permitAll()
            .anyRequest().authenticated();
```

Autenticando usuário

Uma maneira de **fazer autenticação é dizer para o Spring qual classe representa um usuário no sistema**, dá pra fazer isso implementando uma **Interface de UserDetails,** essa interface contém uma **série de métodos** que representam detalhes do usuário ***(senha, login, booleans que representam informações como conta expirada, conta habilitada e etc)*** e que precisam ser sobrescritos com as **informações da sua entidade da API:**

```java
@Entity
public class Usuario implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;
    private String senha;
```

```java
@Override
public String getPassword() {
    return this.senha;
}

@Override
public String getUsername() {
    return this.email;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}
```

Como não tenho uma implementação detalhada para uso dos booleans, por default deixei tudo true*

Um dos métodos da **Interface de UserDetails** pede uma coleção de **Classes** que implementem **GrantedAuthority**, basicamente falando são Perfis de permissão que um usuário pode ter. Por exemplo: administrador, moderador, etc...

Então uma **classe de Perfil** precisa implementar **GrantedAuthority** e retornar o "tipo" do perfil no **método sobrescrito:**

```java
@Entity
public class Perfil implements GrantedAuthority {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String perfil;

    @Override
    public String getAuthority() {
        return this.perfil;
    }
```

```java
@Entity
public class Usuario implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;
    private String senha;

    @ManyToMany(fetch = FetchType.EAGER)
    private List<Perfil> perfis = new ArrayList<>();

    public Usuario(String nome, String email, String senha) {...}

    public Usuario() {}

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.perfis;
    }
```

Agora que o user ta mapeado, é necessário ensinar ao Spring como autentica-lo. Fazemos isso usando o **método de configuração** que recebe o **AuthenticationManagerBuilder.**

Falamos pro **Spring** qual **classe de serviço** (essa classe deve implementar a **Interface UserDetailsService**) **ele vai utilizar para fazer a lógica de autenticação e qual o tipo da criptografia que ele vai utilizar na senha**. Feito isso, o processo é todo automático: