03. Spring Security

Date	@31/01/2023
Categoria	Java API
Curso	Spring Boot 3: aplique boas práticas e proteja uma API Rest

Tópicos

- Projeto da aula anterior
- Autenticação e autorização
- Para saber mais: tipos de autenticação em APIs Rest
- Adicionando o Spring Security
- Entidade usuário e migration
- Para saber mais: hashing de senha
- Repository e Service
- Para saber mais: documentação Spring Data
- Configurações de segurança
- Classe de configurações
- Controller de autenticação
- Faça como eu fiz: autenticação na API
- O que aprendemos?

Autenticação e autorização

Objetivos

- Autenticação

- Autorização (controle de acesso)
- Proteção contra ataques (CSRF, clickjacking, etc.)
- Camada de segurança, o acesso será liberada se e se somente se o token de authorization estiver valido

---

Para saber mais: tipos de autenticação em APIs Rest

Existem diversas formas de se realizar o processo de autenticação e autorização em aplicações Web e APIs Rest, sendo que no curso utilizaremos ***Tokens JWT***.

Você pode conferir as principais formas de autenticação lendo este artigo: Tipos de autenticação.

Para saber mais: hashing de senha

Ao implementar uma funcionalidade de autenticação em uma aplicação, independente da linguagem de programação utilizada, você terá que lidar com os dados de login e senha dos usuários, sendo que eles precisarão ser armazenados em algum local, como, por exemplo, um banco de dados.

Senhas são informações sensíveis e não devem ser armazenadas em texto aberto, pois se uma pessoa mal intencionada conseguir obter acesso ao banco de dados, ela conseguirá ter acesso às senhas de todos os usuários. Para evitar esse problema, você deve sempre utilizar algum algoritmo de ***hashing*** nas senhas antes de armazená-las no banco de dados.

Hashing nada mais é do que uma *função matemática* que converte um texto em outro texto totalmente diferente e de difícil dedução. Por exemplo, o texto *Meu nome é Rodrigo* pode ser convertido para o texto *8132f7cb860e9ce4c1d9062d2a5d1848*, utilizando o algoritmo de *hashing MD5*.

Um detalhe importante é que os algoritmos de *hashing* devem ser de *mão única*, ou seja, não deve ser possível obter o texto original a partir de um *hash*. Dessa forma, para saber se um usuário digitou a senha correta ao tentar se autenticar em uma aplicação, devemos pegar a senha que foi digitada por ele e gerar o *hash* dela, para então realizar a comparação com o *hash* que está armazenado no banco de dados.

Existem diversos algoritmos de *hashing* que podem ser utilizados para fazer essa transformação nas senhas dos usuários, sendo que alguns são mais antigos e não

mais considerados seguros hoje em dia, como o MD5 e o SHA1. Os principais algoritmos recomendados atualmente são:

- **Bcrypt**
- **Scrypt**
- **Argon2**
- **PBKDF2**

Ao longo do curso utilizaremos o algoritmo BCrypt, que é bastante popular atualmente. Essa opção também leva em consideração o fato de que o *Spring Security* já nos fornece uma classe que o implementa.

Para saber mais: documentação Spring Data

Conforme aprendido em vídeos anteriores, o Spring Data utiliza um padrão próprio de nomenclatura de métodos que devemos seguir para que ele consiga gerar as *queries SQL* de maneira correta.

Existem algumas palavras reservadas que devemos utilizar nos nomes dos métodos, como o `findBy` e o `existsBy`, para indicar ao Spring Data como ele deve montar a consulta que desejamos. Esse recurso é bastante flexível, podendo ser um pouco complexo devido às diversas possibilidades existentes.

Para conhecer mais detalhes e entender melhor como montar consultas dinâmicas com o Spring Data, acesse a sua documentação oficial.

Classe de configurações

Um colega de trabalho pede sua ajuda para identificar um problema em seu código, referente a uma classe de configurações do *Spring Security*:

```
@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and().build();
    }
```

```
}
```

Ele afirma que mesmo após criar essa classe, o *Spring Security* ainda está bloqueando todas as requisições que chegam na API, devolvendo o código HTTP **401 (*Unauthorized*)**.

Analise o código anterior e escolha a opção que indica o que está causando esse problema.

- O método `securityFilterChain` deveria ter sido declarado como ***protected***.
  - Não é esse o motivo do problema, pois declarar o método como protected não altera o comportamento do Spring Security.
- O método `securityFilterChain` deveria ter sido anotado com `@Bean`.
  - Sem essa anotação no método, o objeto `SecurityFilterChain` não será exposto como um *bean* para o Spring.
- O parâmetro passado para o método `sessionCreationPolicy` deveria ser `SessionCreationPolicy.STATEFUL`.
  - Não existe a constante `STATEFUL` no enum `SessionCreationPolicy`.
- A classe não está herdando da classe `SpringSecurityConfigurationsBean`.
  - Não existe essa classe no *Spring Security*.

O que aprendemos?

Nessa aula, você aprendeu como:

- Funciona o processo de autenticação e autorização em uma API Rest;
- Adicionar o ***Spring Security*** ao projeto;
- Funciona o comportamento padrão do *Spring Security* em uma aplicação;
- Implementar o processo de autenticação na API, de maneira *Stateless*, utilizando as classes e configurações do *Spring Security*.
