01 Projeto da aula anterior

Caso queira, você pode baixar aqui o projeto do curso no ponto em que paramos na aula anterior.

02 Autenticação e autorização

Transcrição

Agora que implementamos algumas boas práticas na API, continuaremos com o estudo de outras funcionalidades que realizaremos no projeto.
O foco desta aula será na parte de **segurança**. Até o momento, não lidamos com essa questão de controle de acesso, autenticação, autorização, entre outras funcionalidades.

Vamos apresentar alguns conceitos e diagramas antes de iniciarmos a parte do código.

Spring Security

O Spring contém um módulo específico para tratar de segurança, conhecido como **Spring Security**.

Para usarmos no Spring Boot, também vamos utilizar esse mesmo módulo, que já existia antes do Boot, o Spring Security, sendo um módulo dedicado para tratarmos das questões relacionadas com segurança em aplicações.

Essas aplicações podem ser tanto Web quanto uma API Rest, este último sendo o nosso caso. Portanto, esse módulo é completo e contém diversas facilidades e ferramentas para nos auxiliar nesse processo de implementar o mecanismo de autenticação e autorização da aplicação ou API.

Vamos entender o que é o **Spring Security**.

Objetivos

- Autenticação
- Autorização (controle de acesso)
- Proteção contra-ataques (CSRF, clickjacking, etc.)

Em suma, o Spring Security possui três objetivos. Um deles é providenciar um serviço para customizarmos como será o controle de **autenticação** no projeto. Isto é, como os usuários efetuam login na aplicação.

Os usuários deverão preencher um formulário? É autenticação via token? Usa algum protocolo? Assim, ele possui uma maior flexibilidade para lidar com diversas possibilidades de aplicar um controle de autenticação.

O Spring Security possui, também, a **autorização**, sendo o controle de acesso para liberarmos a requisição na API ou para fazermos um controle de permissão.

Por exemplo, temos esses usuários e eles possuem a permissão "A", já estes usuários possuem a permissão "B". Os usuários com a permissão

"A" podem acessar as URLs, os que tiverem a permissão "B", além dessas URLs, podem acessar outras URLs.

Com isso, conseguimos fazer um controle do acesso ao nosso sistema.

Há, também, um mecanismo de proteção contra os principais ataques que ocorre em uma aplicação, como o CSRF (*Cross Site Request Forgery*) e o clickjacking.

São esses os três principais objetivos do Spring Security, nos fornecer uma ferramenta para implementarmos autenticação e autorização no projeto e nos proteger dos principais ataques. Isso para não precisarmos implementar o código que protege a aplicação, sendo que já temos disponível.

No caso da nossa API, o que faremos é o controle de autenticação e autorização, o **controle de acesso**.

Até o momento, não nos preocupamos com essas questões de segurança. Logo, a nossa API está pública. Isso significa que qualquer pessoa que tiver acesso ao endereço da API, consegue disparar requisições, assim como fizemos usando o Insomnia.

Sabemos o endereço da API, no caso é `localhost:8080` - dado que está rodando local na máquina. Porém, após efetuarmos o *deploy* da aplicação em um servidor, seria o mesmo cenário: caso alguém descubra a endereço, conseguirá enviar requisição para cadastrar um médico ou paciente, etc.

O projeto não deveria ser público, somente os funcionários da clínica deveriam ter acesso. Claro que eles utilizarão o front-end ou aplicativo

mobile, mas essas aplicações de clientes irão disparar requisições para a nossa API back-end, e esta deve estar protegida.

A API back-end não deve ser pública, ou seja, receber requisições sem um controle de acesso. A partir disso, entra o Spring Security para nos auxiliar na proteção dessa API no back-end.

ATENÇÃO!
Autenticação em aplicação Web (Stateful) **!=** Autenticação em API Rest (Stateless)

Um detalhe importante é que no caso dos cursos de Spring Boot, estamos desenvolvendo uma API Rest, não uma aplicação Web tradicional.

Essas aplicações desenvolvidas em Java usando o Spring, com o Spring MVC (*Model-View-Controller*) ou JSF (*JavaServer Faces*). A nossa ideia é desenvolvermos só o back-end, o front-end é separado e não é o foco deste curso.
O processo de autenticação em uma aplicação Web tradicional **é diferente** do processo de autenticação em uma API Rest. Em uma aplicação Web, temos um conceito chamado de *stateful*.

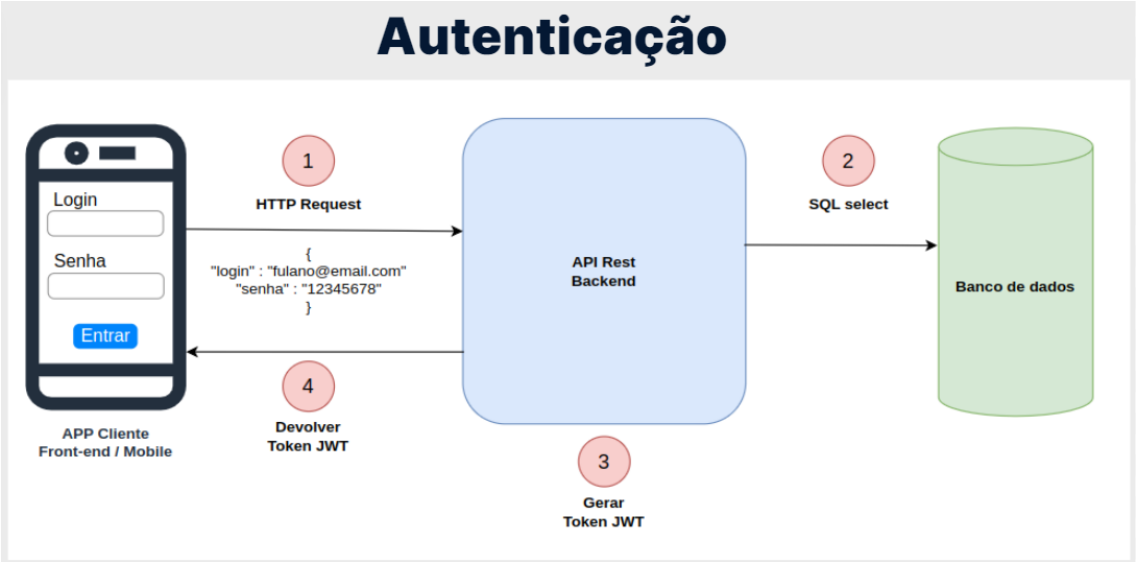
Toda vez que um usuário efetua o login em uma aplicação Web, o servidor armazena o estado. Isto é, cria as sessões e, com isso, consegue identificar cada usuário nas próximas requisições.

Por exemplo, esse usuário é dono de determinada sessão, e esses são os dados de memória deste usuário. Cada usuário possui um espaço na memória. Portanto, o servidor armazena essas sessões, espaços em memória e cada sessão contém os dados específicos de cada usuário.

Esse é o conceito de ***Stateful***, é mantido pelo servidor.
Porém, em uma API Rest, não deveríamos fazer isso, porque um dos conceitos é que ela seja ***stateless***, não armazena estado. Caso o cliente da API dispare uma requisição, o servidor processará essa requisição e devolverá a resposta.

Na próxima requisição, o servidor não sabe identificar quem é que está enviando, ele não armazena essa sessão. Assim, o processo de autenticação funciona um pouco diferente, caso esteja acostumado com a aplicação Web.

Como será o processo de autenticação em uma API? Temos diversas estratégias para lidarmos com a autenticação. Uma delas é usando **Tokens**, e usaremos o *JWT - JSON Web Tokens* como protocolo padrão para lidar com o gerenciamento desses tokens - geração e armazenamento de informações nos tokens.



Esse diagrama contém um esquema do processo de autenticação na API. Lembrando que estamos focando no back-end, e não no front-end. Esta será outra aplicação, podendo ser Web ou Mobile.

No diagrama, o cliente da API seria um aplicativo mobile. Assim, quando o funcionário da clínica for abrir o aplicativo, será exibida uma tela de login tradicional, com os campos "Login" e "Senha" com um botão "Entrar", para enviar o processo de autenticação.

O usuário digita o login e senha, e clica no botão para enviar. Deste modo, a aplicação captura esses dados e dispara uma requisição para a API back-end - da mesma forma que enviamos pelo Insomnia.

Logo, o primeiro passo é a requisição ser disparada pelo aplicativo para a nossa API, e no corpo desta requisição é exibido o JSON com o login e senha digitados na tela de login.

O segundo passo é capturar esse login e senha e verificar se o usuário está cadastrado no sistema, isto é, teremos que consultar o banco de dados. Por isso, precisaremos ter uma tabela em que vamos armazenar os usuários e suas respectivas senhas, que podem acessar a API.

Da mesma maneira que temos uma tabela para armazenar os médicos e outra para os pacientes, teremos uma para guardar os usuários. Logo, o segundo passo do processo de autenticação é: a nossa API capturar esse login e senha, e ir ao banco de dados efetuar uma consulta para verificar a existência dos dados desse usuário.

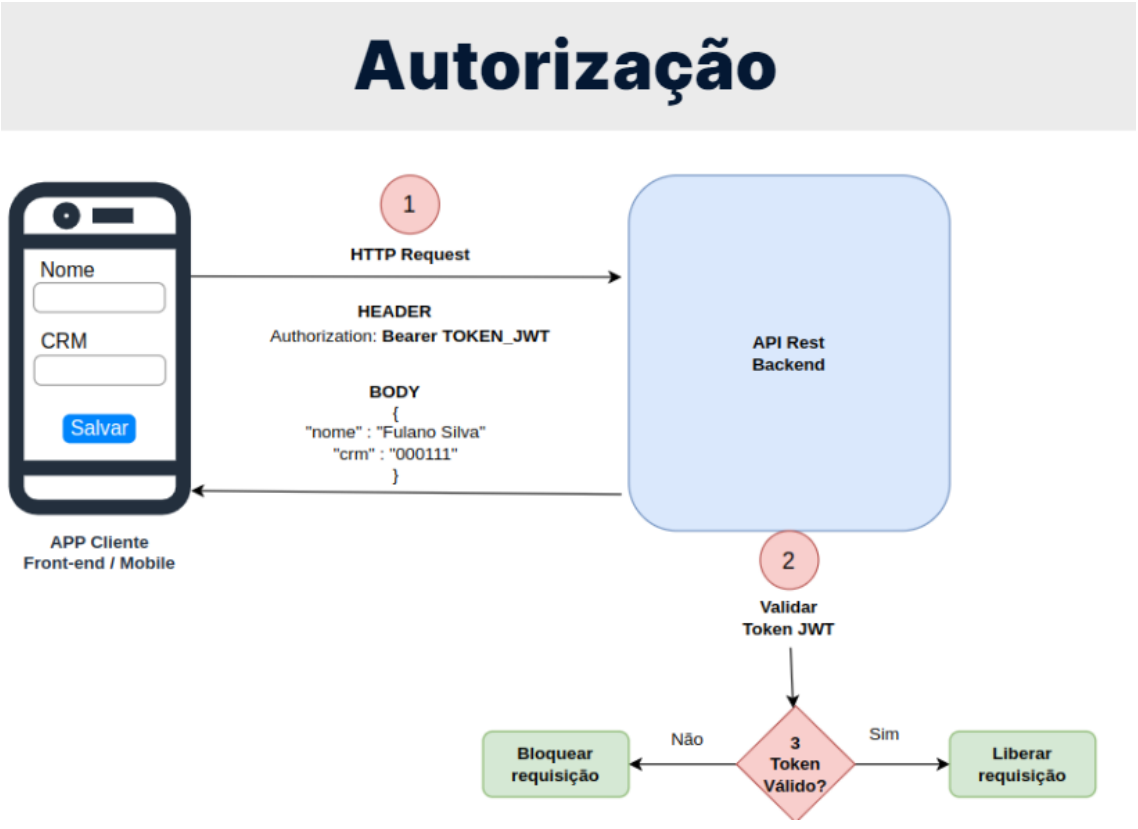
Se for válido, a API gera um Token, que nada mais é que uma string. A geração desse Token segue o formato JWT, e esse token é devolvido na resposta para a aplicação de cliente, sendo quem disparou a requisição.

Esse é o processo de uma requisição para efetuar o login e autenticar em uma API Rest, usando tokens. Será esse processo que seguiremos neste curso.

Isto é, teremos um controller mapeando a URL de autenticação, receberemos um DTO com os dados do login e faremos uma consulta no banco de dados. Se tiver tudo certo, geramos um token e devolvemos para o front-end, para o cliente que disparou a requisição.

Esse token deve ser armazenado pelo aplicativo mobile/front-end. Há técnicas para guardar isso de forma segura, porque esse token que identifica se o usuário está logado.

Assim, nas requisições seguintes entra o processo de autorização, que consta no diagrama a seguir:



Na requisição de cadastrar um médico, o aplicativo exibe o formulário de cadastro de médico - simplificamos no diagrama, mas considere que é um formulário completo - e após preenchermos os dados, clicamos no botão "Salvar".

Será disparada uma requisição para a nossa API - da mesma forma que fizemos no Insomnia. No entanto, além de enviar o JSON com os dados do médico no corpo da resposta, a requisição deve incluir um cabeçalho chamado *authorization*. Neste cabeçalho, levamos o token obtido no processo anterior, de login.
A diferença será essa: todas as URLs e requisições que desejarmos proteger, teremos que validar se na requisição está vindo o cabeçalho *authorization* com um token. E precisamos validar este token, gerado pela nossa API.

Portanto, o processo de autorização é: primeiro, chega uma requisição na API e ela lê o cabeçalho authorization, captura o token enviado e valida se foi gerado pela API. Teremos um código para verificar a validade do token.

Caso não seja válido, a requisição é interrompida ou bloqueada. Não chamamos o controller para salvar os dados do médico no banco de dados, devolvemos um erro `403` ou `401`. Há protocolos HTTP específicos para esse cenário de autenticação e autorização.

Pelo fato do token estar vindo, o usuário já está logado. Portanto, o usuário foi logado previamente e recebeu o token. Este token informa se o login foi efetuado ou não. Caso seja válido, seguimos com o fluxo da requisição.

O processo de autorização funciona assim justamente porque a nossa API deve ser *Stateless*. Ou seja, não armazena estado e não temos uma sessão informando se o usuário está logado ou não. É como se em cada requisição tivéssemos que logar o usuário.

Todavia, seria incomum enviar usuário e senha em todas as requisições. Para não precisarmos fazer isso, criamos uma URL para realizar a autenticação (onde é enviado o login e senha), e se estiver tudo certo a API gera um token e devolve para o front-end ou para o aplicativo mobile.

Assim, nas próximas requisições o aplicativo leva na requisição, além dos dados em si, o token. Logo, não é necessário mandar login e senha, somente o token. E nesta string, contém a informação de quem é esse usuário e, com isso, a API consegue recuperar esses dados.

Essa é uma das formas de fazer a autenticação em uma API Rest.

Caso já tenha aprendido a desenvolver uma aplicação Web tradicional, o processo de autenticação em uma API Rest é diferente. Não possui o conceito de sessão e cookies, é stateless - cada requisição é individual e não armazena o estado da requisição anterior.

Como o servidor sabe se estamos logados ou não? O cliente precisa enviar alguma coisa para não precisarmos enviar login e senha em toda requisição. Ele informa o login e a senha na requisição de logar, recebe um token e nas próximas ele direciona esse mesmo token.

Nas próximas aulas adicionaremos o Spring Security, e implementar esse conceito de autenticação e autorização analisando cada detalhe no Spring Boot.

Até mais!

03 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.

04 Adicionando o Spring Security

Transcrição

Agora que entendemos como funciona o processo de autenticação e autorização, começaremos a utilizar o Spring Security.

Com o projeto aberto no IntelliJ, a primeira etapa que precisamos fazer é adicionar o Spring Security no projeto. Ao criá-lo no site do Spring Initializr, não adicionamos as dependências do Spring Security.

À esquerda do IntelliJ, clicaremos no arquivo `pom.xml`. Perceba que não consta nenhuma dependência referente ao Spring Security, para adicioná-la abriremos o navegador no site do Spring Initializr .

Em "*Project*" estamos com "*Maven Project*" selecionado, na seção "*Language*" estamos com "Java" e em "Spring Boot" estamos com a

versão "3.0.0(RCI)". Pode ser que no momento em que estiver assistindo a este vídeo, a versão `3.0.0` esteja finalizada.

Não preencheremos os outros campos do lado esquerdo. À direita, em "*Dependencies*", clicaremos no botão "*Add dependencies*" ou podemos usar o atalho do teclado "Ctrl + B". Será mostrada uma pop-up, em digitaremos "*security*" na caixa de texto, na parte superior. Selecionaremos a opção "Spring Security".

Perceba que a pop-up fechou e a dependência já foi adicionada em "*Dependencies*". Após isso, clicaremos no botão "*Explore*" na parte inferior central, ou podemos usar o atalho "Ctrl + Space".

É exibido o arquivo `pom.xml` do projeto que estamos criando agora. Descendo o código até a parte de dependências, note que foi incluída duas dependências: o `spring-boot-starter-security` e o `spring-security-test`.

pom.xml

```
//código omitido


<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
</dependency>


<dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
```

```
            <scope>test</scope>
</dependency>


//código omitido COPIAR CÓDIGO
```

Precisaremos das duas, copiaremos primeiro a `spring-boot-starter-security` completa, voltaremos ao IntelliJ e colaremos após a última dependência `mysql-connector-java`. Isso antes de fechar a tag `dependencies`.

Após isso, voltaremos ao site do Spring Initializr e copiaremos a dependência `spring-security-test`. Em seguida, vamos voltar ao IntelliJ, selecionaremos "Enter" para dar um espaço antes de fechar a tag e colaremos a dependência.

Podemos salvar o arquivo `pom.xml` do projeto.

Como estamos alterando as dependências do projeto, o ideal é pararmos de rodar a aplicação. Para isso, clicaremos no botão vermelho "■" na parte superior direita ou podemos usar o atalho "Ctrl + F12".

Para garantir que as dependências do Maven foram baixadas com sucesso, selecionaremos "Maven" no canto superior direito, escrito verticalmente.

Será exibida uma aba, em que clicaremos no primeiro ícone de *reload* "⟳" no canto superior esquerdo, para recarregarmos todos os projetos do Maven.

É sempre bom fazer esse processo para garantir que o Maven baixou as dependências corretamente no projeto.

Agora, iniciaremos o projeto clicando no botão com um ícone de play "▶", na parte superior. Vamos verificar se algo foi alterado após incluirmos o módulo de segurança abrindo a aba "Run", no canto inferior esquerdo.

Analisando os logs, perceba que no final foi gerado uma linha que antes não era exibida no retorno.

Linha selecionada pelo instrutor:
Using generated security password: 520dbc8a-b02a-44dq-a259-0afdb628a93c

Ele nos informa que gerou uma senha aleatória: "Usando a senha de segurança gerada". E em seguida nos passa a senha.

Na linha seguinte é comunicado que essa senha gerada é somente para ambiente de desenvolvimento, e que não devemos usar em ambiente de produção:

Linha selecionada pelo instrutor:
This generated password is for development use only. Your security configuration must be updated before running your application in production ("Essa senha gerada é apenas para uso de desenvolvimento. Sua configuração de segurança deve ser atualizada antes de executar seu aplicativo em produção")

Mais para frente vamos entender qual é esse usuário e senha gerado para nós usarmos em ambientes de desenvolvimento.

Ao adicionarmos o Spring Security como dependência do projeto, ele gera uma configuração padrão, criando um usuário e senha e exibindo no log. E, também, faz uma alteração no projeto, bloqueando todas as requisições.

Se tentarmos enviar qualquer requisição na API, ela será bloqueada e será gerado um erro. Além disso, seremos redirecionados para uma tela de login do Spring Security. Vamos verificar isso acontecendo?

Voltaremos ao Insomnia para disparar a requisição de listagem de médicos. No endereço, temos:

```
http://localhost:8080/medicos COPIAR CÓDIGO
```

Após verificar o endereço, clicaremos no botão "Send". Nos devolveu o código `401 Unauthorized`, e em "Preview", temos a mensagem: "*No body returned for response*" ("Nenhum corpo retornou para resposta").

Isto é, tentamos disparar uma requisição para a URL `/medicos`, mas está tudo bloqueado e não estamos logados. Por isso, não temos autorização para disparar essa requisição.

Isso acontece com qualquer URL. Por exemplo, na requisição "Detalhar Médico", se clicarmos no botão "Send", também, nos retorna o código `401 Unauthorized`.

Todas as requisições estão sendo bloqueadas pelo Spring Security. Portanto, esse é o comportamento padrão do Spring Security ao

adicioná-lo como dependência no projeto, ele bloqueia tudo e gera uma senha aleatória.

Onde podemos usar essa senha gerada? Vamos disparar a requisição pelo navegador, agora. Por ele conseguimos analisar melhor.

URL do navegador:

```
http://localhost:8080/medicos COPIAR CÓDIGO
```

Ao clicarmos na tecla "Enter" é exibida uma tela de Login, com a frase "*Please sign in*" ("Por favor, inscreva-se") e os campos username e password. Abaixo, um botão azul "*Sign in*", centralizado.

Somos redirecionados para esse formulário, sendo do próprio Spring Security. Assim, esse é o padrão: bloquear todas as URLs e se tentarmos acessar do navegador, ele bloqueia e nos redireciona para o formulário de login.

Para efetuar esse login, vamos voltar ao IntelliJ e copiar a senha gerada no console, usando o atalho "Ctrl + C".

Linha selecionada pelo instrutor para copiar a senha:
Using generated security password: 520dbc8a-b02a-44dq-a259-0afdb628a93c

Após copiarmos a senha, colaremos no campo *password* do formulário. Por padrão, o Spring gera um usuário chamado `user`, por isso, no campo *username* digitamos "user".

Please sign in

- Username: user

- Password: 520dbc8a-b02a-44dq-a259-0afdb628a93c

Esse é o usuário padrão para acessarmos o formulário. Já a senha, cada vez que o projeto for reiniciado, será gerada uma nova senha no console.

Clicaremos no botão "*Sign in*". Ao entrarmos no sistema, percebemos que a lista de médicos foi liberada e exibida em um JSON. Tivemos que nos autenticar previamente .

JSON exibido após efetuar o login:


```
content:


0:


    id: 7
    nome: "Bruna Silva"
    email: "bruna.silva@voll.med"
    crm: "1234580"
    especialidade: "ORTOPEDIA"


1:


    id: 6
    nome: "Juliana Queiroz"
    email: "juliana.queiroz@voll.med"
    crm: "233444"
    especialidade: "ORTOPEDIA"
```

```
//retorno omitido COPIAR CÓDIGO
```

Portanto, esse é o comportamento padrão do Spring Security: ele bloqueia todas as URLs, disponibiliza um formulário de login - que e possui um usuário padrão chamado *user* e a senha devemos copiar do console ao inicializar o projeto.

Contudo, no nosso projeto não é exatamente isso que desejamos. Esse comportamento padrão funciona para aplicações Web, que trabalha no modelo de *stateful*. Porém, estamos trabalhando com uma API Rest, em que a autenticação será feita no modelo *stateless*.

Não teremos esse formulário de login no back-end, isso é responsabilidade do front-end ou do aplicativo mobile.

Portanto, vamos precisar realizar algumas configurações, e alterar esse mecanismo padrão do Spring Security. Isso para implementarmos o nosso mecanismo de autenticação personalizado no projeto.

Na sequência, iniciaremos essas mudanças. Até mais!

05 Entidade usuário e migration

Transcrição

Na aula anterior, entendemos o comportamento padrão do Spring Security ao ser adicionado no projeto. Neste vídeo, vamos

implementar as alterações que desejamos nesse comportamento padrão.

São diversas coisas que precisamos implementar, conforme o diagrama que vimos anteriormente do processo de autenticação e autorização, usando tokens.

Para autenticar o usuário na API, precisamos ter uma tabela no banco de dados em que serão armazenados os usuários e suas respectivas senhas. Essa é uma das modificações que vamos fazer no projeto.

Além da tabela de médicos e pacientes, incluiremos uma de usuário. Precisamos representar esse conceito de usuário no código, como estamos usando a JPA, criaremos uma entidade JPA para simbolizar esse usuário e temos que gerar a tabela no banco de dados. Faremos isso usando as ***migrations*** já existentes no projeto.

Para criarmos a entidade JPA, vamos em "src > main > java > med.voll.api > domain", na pasta `domain` ficam os dados relacionados ao domínio da aplicação.

para criar um pacote para guardar essas classes e conceitos referentes ao usuário.

Agora, criaremos um pacote para guardar essas classes e conceitos referentes ao usuário.

Para isso, vamos selecionar o pacote `domain`, e depois usamos o atalho "Alt + Insert". Será exibida uma aba com algumas opções, dentre elas escolheremos a "*Package*" e na aba seguinte digitaremos "usuario".

Assim, ficamos com o seguinte pacote: `med.voll.api.domain.usuario`.
Selecionaremos a tecla "Enter", para salvar. Agora, dentro do pacote `usuario` criaremos uma classe. Com esse objetivo, selecionamos o pacote `usuario` e depois as teclas "Ctrl + Insert".

Na aba exibida, vamos clicar na opção "*Java Class*". Após isso, será mostrado um pop-up em que digitaremos o nome da classe, no caso será somente "Usuario". Podemos selecionar "Enter" novamente, para criar.

```
Usuario.java:

package med.voll.api.domain.usuario;


public class Usuario {


} COPIAR CÓDIGO
```

A classe foi criada, por ser uma entidade JAP terá as mesmas anotações e atributos. Da mesma forma quando mapeamos a entidade médico.

No caso do usuário, será necessário somente três atributos: `id`, `login` e `senha`. Toda tabela possui um ID (chave primária) e, portanto, devemos incluir na entidade também.

Por esse motivo, dentro da classe `Usuario` colocaremos: `private Long id;`. O usuário possui somente duas informações: login e senha. Ambos são *strings*, ou seja, texto.

Para adicionarmos esses campos, na linha seguinte do `id` digitaremos `private String login;` e na próxima linha colocamos a senha: `private String senha;`.

```
Usuario.java:

package med.voll.api.domain.usuario;


public class Usuario {


    private Long id;

    private String login;

    private String senha;


} COPIAR CÓDIGO
```

São esses os atributos necessários para o objeto usuário. Agora, vamos inserir as anotações do JPA e do Lombok, para gerarmos os *getters* e *setters*, construtores, entre outras coisas.

Para facilitar, clicaremos no arquivo `Medico` que se encontra em "src > main > java > med.voll.api > domain > medico". Nele, copiaremos as anotações nas linhas iniciais:

```
Medico:

//código omitido


@Table(name = "medicos")

@Entity(name = "Medico")

@Getter
```

```
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")

//código omitido COPIAR CÓDIGO
```

Após copiar, voltaremos ao arquivo `Usuario` e colocaremos em cima da classe. Perceba que apareceram diversos `imports` no início do código.

Usuario.java:

```
package med.voll.api.domain.usuario;

import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;

@Table(name = "medicos")
@Entity(name = "Medico")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")

public class Usuario {
```

```
    private Long id;

    private String login;

    private String senha;


} COPIAR CÓDIGO
```

Porém, ainda precisamos adaptar. Na anotação `@Table`, no parâmetro `name` não colocaremos a tabela de médicos e sim de usuários.

```
@Table(name = "usuarios")COPIAR CÓDIGO
```

Na anotação `@Entity` o parâmetro `name` também será "Usuario", porém, com a letra inicial maiúscula (nome da classe). Não vamos alterar as anotações restantes, trocamos somente o nome da tabela e da entidade.

```
@Entity(name = "Usuario") COPIAR CÓDIGO
```

Por enquanto, temos:

```
Usuario.java:

package med.voll.api.domain.usuario;


import jakarta.persistence.Entity;

import jakarta.persistence.Table;

import lombok.AllArgsConstructor;

import lombok.EqualsAndHashCode;

import lombok.Getter;

import lombok.NoArgsConstructor;
```

```
@Table(name = "usuarios")
@Entity(name = "Usuario")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")

public class Usuario {

    private Long id;
    private String login;
    private String senha;

} COPIAR CÓDIGO
```

Voltaremos à entidade `medico` e copiaremos a anotação acima do ID:

```
Medico:
//código omitido

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)

//código omitido COPIAR CÓDIGO
```

Em seguida, voltamos ao arquivo `Usuario` e colaremos acima do atributo `id`.

```
Usuario.java:
```

```
package med.voll.api.domain.usuario;


import jakarta.persistence.*
import lombok.AllArgsConstructor;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;


@Table(name = "usuarios")
@Entity(name = "Usuario")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")


public class Usuario {


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


} COPIAR CÓDIGO
```

Com isso, mapeamos a entidade JPA.

Essa entidade está representando uma tabela no banco de dados e, por isso, essa tabela precisa ser criada no banco de dados.

Portanto, criaremos uma *migration* para fazer a evolução do *schema* do banco de dados. Toda vez que precisarmos alterar algum dado no banco de dados (excluir coluna, incluir linha, etc), geramos uma nova *migration*.

ATENÇÃO: sempre interrompa o projeto ao codificar as migrations!

Por isso, no IntelliJ, clicaremos no botão vermelho "■", para interromper o projeto. Logo após, à esquerda, vamos em "src > main > resources > db.migration". Note que temos quatro pastas, sendo elas:

- `db.migration`
    - `V1__create-table-medicos.sql`
    - `V2__alter-table-medicos-add-column-telefone.sql`
    - `V3__alter-table-medicos-add-column-ativo.sql`
    - `V4__create-table-pacientes.sql`

Vamos criar uma quinta *migration*. Para isso, selecionaremos a primeira *migration* `V1__create-table-medicos.sql`, e clicaremos nas teclas "Ctrl + C" e depois em "Ctrl + V".

Será aberta uma pop-up *copy*, com os campos "*new name*" e "*to directory*". No campo "New name" alteraremos para `V5__create-table-usuarios.sql`.

```
V5__create-table-usuarios.sql COPIAR CÓDIGO
```

Seremos redirecionados para o arquivo:

`V5__create-table-usuarios.sql`:

```
create table medicos(

    id bigint not null auto_increment,
    nome varchar(100) not null,
    email varchar(100) not null unique,
    crm varchar(6) not null unique,
    especialidade varchar(100) not null,
    logradouro varchar(100) not null,
    bairro varchar(100) not null,
    cep varchar(9) not null,
    complemento varchar(100),
    numero varchar(20),
    uf char(2) not null,
    cidade varchar(100) not null,

    primary key(id)

); COPIAR CÓDIGO
```

No entanto, precisamos alterar esse código. No comando `create table`, não usaremos "medicos" e sim "usuarios".

```
create table usuários COPIAR CÓDIGO
```

Nos campos, podemos incluir "login" no lugar de "nome" e "senha" ao invés de "email". Após essas alterações, podemos remover os outros campos, dado que não usaremos para o usuário. Podemos excluir o `unique` da senha e aumentar de 100 para 255 caracteres.

`V5__create-table-usuarios.sql:`

```
create table usuarios(

    id bigint not null auto_increment,
    login varchar(100) not null,
    senha varchar(255) not null,

    primary key(id)

); COPIAR CÓDIGO
```

Com isso, temos a estrutura da nossa tabela de usuários.

Ao inserir um usuário, no campo "login" digitamos o e-mail do usuário (informação única) e em "senha", não vamos armazenar na tabela "123456", isto é, em texto explícito. Usaremos um algoritmo de *hash*, aprenderemos mais adiante e com calma.

Podemos salvar o arquivo da quinta *migration*.

Desse modo, temos a entidade usuário e a *migration*. Podemos rodar o projeto clicando no botão verde de play "▶", na parte superior direita do IntelliJ.

Ao rodarmos o projeto, o *flyway* detecta a nova *migration* e executar esse script no banco de dados.
Será exibida a aba "*Run*", procuraremos por algo que nos informe que uma nova versão subiu.

Linhas selecionadas pelo instrutor:
Current version of schema 'vollmedapi': 4

Migrating schema 'vollmedapi' to version "5 - create-table-usuarios"

Nessas linhas, ele nos comunica a versão atual do projeto, depois que rodou uma *migration* e subiu à quinta versão. Na linha seguinte, nos informa que foi executado com sucesso e estamos na versão 5:

Linha selecionada pelo instrutor:
Successfully applied 1 migration to schema 'vollmedapi', now at version v5

Isso significa que a tabela de usuários foi criada!

Com isso, concluímos o primeiro passo: criamos uma entidade JPA simbolizando o usuário e depois geramos uma *migration* no projeto, para criar a tabela no banco de dados.

Na sequência, seguiremos implementando as novas alterações para termos o controle de autenticação do nosso projeto.

06 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.

07 Repository e Service

Transcrição

Na aula anterior, criamos a entidade JPA e a *migration*. Agora, criaremos outras classes no projeto, por exemplo, o *repository*. Isso porque, em geral, cada entidade JPA terá uma interface *repository*, sendo quem faz o acesso ao banco de dados.

Para criarmos o *repository* do usuário, vamos em "src > main > java > med.voll.api > domain > usuario". Com o pacote `usuario` selecionado, usaremos o atalho "Alt + Insert", e na aba exibida escolheremos a opção "Java Class".

Na aba "*New Java Class*" digitaremos o nome "UsuarioRepository" e deixaremos a segunda opção "*Interface*" marcada. Logo após, clicaremos na tecla "Enter", para criar a *interface*.

UsuarioRepository

```
package med.voll.api.domain.usuario;


public interface UsuarioRepository {

} COPIAR CÓDIGO
```

Por ser um *repository* precisamos herdar, por isso, acrescentaremos `extends JpaRepository`, após o nome da interface. Sendo `JpaRepository` a interface do Spring Data.

```
UsuarioRepository
package med.voll.api.domain.usuario;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UsuarioRepository extends JpaRepository {
} COPIAR CÓDIGO
```

Logo após a interface `JpaRepository`, vamos abrir e fechar um sinal de menor que ("<") e outro sinal de maior que (">"), os *generics*. Dentro dos *generics* passaremos os dois tipos: quem é a entidade e o tipo de chave primária.

No caso, a entidade é o `Usuario` e o tipo da chave é `Long`, para representar os IDs.

```
UsuarioRepository
package med.voll.api.domain.usuario;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UsuarioRepository extends JpaRepository <Usuario,
Long> {
} COPIAR CÓDIGO
```

Sempre que precisarmos consultar a tabela de usuários no banco de dados, usaremos o `UsuarioRepository`. Além disso, vamos criar algumas classes de configurações para o Spring Security.

Nessa parte de autenticação, o Spring Security detecta algumas coisas de forma automática no projeto. Por exemplo, a classe que vai ter a lógica de usar o *repository* e acessar o banco de dados para realizar a consulta.

O Spring possui um comportamento padrão, ele procura por uma classe específica no projeto. Portanto, precisamos criar essa classe seguindo o padrão do Spring e, com isso, ele consegue identificá-la no projeto e usá-la para fazer o processo de autenticação.

Para isso funcionar, vamos criar uma classe dentro do pacote `usuario`. Com a pasta `usuario` selecionada, utilizamos o atalho "Alt + Insert", e depois escolhemos a opção "Java Class". No pop-up seguinte, digitamos o nome "AutenticacaoService" e selecionamos "Enter".

AutenticacaoService

```
package med.voll.api.domain.usuario;


public class AutenticacaoService {


} COPIAR CÓDIGO
```

Esta classe terá o código com a lógica de autenticação no projeto.

Até o momento é uma classe Java tradicional, não contém nenhuma anotação e, portanto, o Spring não a reconhece. Para o Spring

identificar essa classe, usamos a anotação `@Service` em cima da classe.

A anotação `@Service` serve para o Spring identificar essa classe como um componente do tipo serviço. Isto é, ela executará algum serviço no projeto, no caso, será o de autenticação. Assim, o Spring carrega a classe e executa o serviço ao inicializarmos o projeto.

```
AutenticacaoService

package med.voll.api.domain.usuario;


import org.springframework.stereotype.Service;


@Service
public class AutenticacaoService {


} COPIAR CÓDIGO
```

O Spring Security também precisa ser informado que esta classe é a responsável pelo serviço de autenticação. Para isso, teremos que implementar uma interface.

Portanto, na assinatura da classe, antes de abrir as chaves vamos digitar `implements UserDetailsService`.
Sendo `UserDetailsService` uma interface do Spring Security.

```
AutenticacaoService

package med.voll.api.domain.usuario;
```

```
import
org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Service;

@Service
public class AutenticacaoService implements UserDetailsService {

} COPIAR CÓDIGO
```

Com isso, o Spring será o responsável por chamar essa classe. Não injetaremos a classe `AutenticacaoService` em nenhum controller, o Spring consegue identificá-la e chamá-la quando ocorrer o processo de autenticação.

Isso desde que ela possua a anotação `@Service` e implemente a interface `UserDetailsService`, própria do Spring Security.

Perceba que está dando erro de compilação na linha da classe, está sublinhado na cor vermelha. Isso acontece porque ao implementarmos uma interface, **precisamos implementar os métodos dessa interface**.

```
AutenticacaoService
package med.voll.api.domain.usuario;

import
org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Service;
```

```
@Service
public class AutenticacaoService implements UserDetailsService {


} COPIAR CÓDIGO
```

Clicando na interface `UserDetailsService`, e usando o atalho "Atl + Enter", será exibido um pop-up com algumas opções. Dentre elas, escolheremos a "*Implement methods*" ("Implementar métodos").

`AutenticacaoService`

```
package med.voll.api.domain.usuario;


import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExceptio
n;
import org.springframework.stereotype.Service;


@Service
public class AutenticacaoService implements UserDetailsService {



        @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return null;
    }
```

```
} COPIAR CÓDIGO
```

Precisamos detalhar somente um método na classe dessa interface. No caso, o `load User By Username`, sendo o método que o Spring chama de forma automática ao efetuarmos o login.

Com isso, quando o usuário efetuar o login, o Spring busca pela classe `AutenticacaoService` - por ser a responsável por implementar a `UserDetailsService` - e chama o método `loadUserByUsername`, passando o `username` digitado no formulário de login.

Em suma, essa é a lógica deste método.

Precisamos devolver alguma coisa, no caso, precisamos acessar o banco de dados, ou seja, usaremos o *repository*. Logo, vamos injetar o *repository* como uma dependência da classe.

Para isso, acima do método `loadUserByUsername` vamos declarar o atributo `private UsuarioRepository` nomeando como *repository*.

AutenticacaoService

```
package med.voll.api.domain.usuario;


import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExceptio
n;
import org.springframework.stereotype.Service;
```

```
@Service
public class AutenticacaoService implements UserDetailsService {

        private UsuarioRepository repository;

        @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return null;
    }
} COPIAR CÓDIGO
```

Para injetarmos a dependência na classe, usaremos a anotação `@Autowired` ou podemos usar o Lombok para gerar um construtor com esse atributo. Usaremos a primeira opção.

AutenticacaoService

```
package med.voll.api.domain.usuario;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExceptio
n;
import org.springframework.stereotype.Service;
```

```
@Service
public class AutenticacaoService implements UserDetailsService {

        @Autowired
        private UsuarioRepository repository;

        @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return null;
    }
} COPIAR CÓDIGO
```

Agora, no método `loadUserByUsername` vamos alterar o retorno de *null* para *repository*. Na sequência colocamos um ponto ("."), e incluímos o método `findByLogin()` (nome do nosso atributo que representa o login), passando como parâmetro o `username`.

AutenticacaoService

```
package med.voll.api.domain.usuario;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExceptio
n;
import org.springframework.stereotype.Service;
```

```
@Service
public class AutenticacaoService implements UserDetailsService {

        @Autowired
        private UsuarioRepository repository;

        @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return repository.findByLogin(username);
    }
} COPIAR CÓDIGO
```

Teremos um erro de compilação em `findByLogin`, porque não existe esse método na interface `UsuarioRepository`. Para ajustar isso, selecionamos o método, usamos o atalho "Alt + Enter" e escolhemos a opção "*Create method 'findByLogin' in 'UsuarioRepository'*".

Após clicarmos nesta opção, seremos redirecionados para o arquivo:

`UsuarioRepository`

```
package med.voll.api.domain.usuario;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.security.core.userdetails.UserDetails;

public interface UsuarioRepository extends JpaRepository<Usuario,
Long> {
    UserDetails findByLogin(String username);
```

```
} COPIAR CÓDIGO
```

Foi criado na nossa interface `UsuarioRepository` com o método `findByLogin()`. Neste método, vamos alterar o parâmetro de `username` para `login`.

```
UsuarioRepository

package med.voll.api.domain.usuario;


import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.security.core.userdetails.UserDetails;


public interface UsuarioRepository extends JpaRepository<Usuario,
Long> {
    UserDetails findByLogin(String login);
} COPIAR CÓDIGO
```

O `findByLogin()` é o método responsável por realizar a consulta do usuário no banco de dados. Portanto, será usado em `AutenticacaoService`.

Assim, criamos a entidade JPA que representa o usuário; temos a *migration* que gerou a tabela no banco de dados; a interface `repository` responsável pelo acesso na tabela de usuários; a classe `AutenticacaoService` que simboliza o serviço de autenticação, será chamada pelo Spring automaticamente quando efetuarmos a autenticação no projeto.

Na sequência, continuaremos com as implementações no projeto.

Te espero na próxima aula!

08 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.

09 Configurações de segurança

Transcrição

A próxima alteração é configurar o Spring Security para ele não usar o processo de segurança tradicional, o *stateful*. Como estamos trabalhando com uma API Rest, o processo de autenticação **precisa ser *stateless***.

Por ser uma configuração, provavelmente tenha pensado no arquivo `application.properties`. No entanto, não é uma configuração de chave/valor, é sim uma configuração mais complexa e dinâmica que envolve classes, entre outros recursos do Spring e do projeto.

Vamos fazer essa configuração via código Java, e não usando propriedades no arquivo `application.properties`.

No IntelliJ, vamos em "src > main > java > med.voll.api > infra". Dentro do pacote `infra`, já que é uma configuração de infraestrutura de segurança, vamos criar uma classe.

Para organizarmos esse pacote de `infra` e não ficarmos com várias classes misturadas, vamos criar sub-pacotes. Por enquanto, temos somente a classe `TratadorDeErros`.

Clicaremos com o botão direito do mouse em `TratadorDeErros` e, na aba exibida vamos escolher as opções "Refactor > Move Class...". Será mostrado um pop-up, em que no campo "*To package*" vamos criar um sub-pacote.

Em "To package" estamos com `med.voll.api.infra`, no final incluiremos `.exception`. Isso porque a classe `TratadorDeErros` tem haver com exceções que podem acontecer no projeto. Assim, ficamos com: `med.voll.api.infra.exception`.

Logo após, vamos clicar no botão "*Refactor*", no canto inferior direito. Será exibida uma aba "*Move*", em que selecionaremos o botão "*Yes*". Com isso, a classe `TratadorDeErros` foi movida para o pacote `infra.exception`.

Com o pacote `infra.exception` selecionado, usaremos o atalho "Alt + Insert" e escolheremos a opção "Package". Na aba seguinte, temos `med.voll.api.infra.exception`. Removeremos o 'exception' e adicionaremos `security`, e depois apertamos "Enter".

No sub-pacote `security`, vamos criar as classes e configurações relacionadas à segurança.

Do lado esquerdo do IntelliJ, ficamos com o seguinte pacote e sub-pacotes:

- infra
    - exception
    - security

Agora, vamos criar a classe de configurações de segurança. Selecionando o pacote `security`, usamos o atalho "Alt + Insert" e escolheremos a opção "Java Class".

Na aba seguinte, intitulado "*New Java Class*", digitaremos "SecurityConfigurations".

```
SecurityConfigurations COPIAR CÓDIGO
```

Podemos clicar na tecla "Enter", para salvar. Seremos redirecionados para o arquivo `SecurityConfigurations`:

```
package med.voll.api.infra.security;


public class SecurityConfigurations {

} COPIAR CÓDIGO
```

É nesta classe que vamos concentrar as informações de segurança do Spring Security. Por enquanto, é uma classe Java e o Spring não

carrega essa classe no projeto, dado que não temos nenhuma anotação nela.

Por ser uma classe de configurações, usaremos a anotação `@Configuration`.

SecurityConfigurations

```
package med.voll.api.infra.security;


import org.springframework.context.annotation.Configuration;


@Configuration
public class SecurityConfigurations {


} COPIAR CÓDIGO
```

Desse modo, o Spring identifica a classe e a carrega no projeto.

Como faremos as configurações de segurança, personalizando o processo de autenticação e autorização, adicionaremos mais uma anotação.

No caso, usaremos uma do Spring Security, sendo a `@EnableWebSecurity`. Isso para informarmos ao Spring que vamos personalizar as configurações de segurança.

SecurityConfigurations

```
package med.voll.api.infra.security;


import org.springframework.context.annotation.Configuration;
```

```
import
org.springframework.security.config.annotation.web.configuration.Enabl
eWebSecurity;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

} COPIAR CÓDIGO
```

Dentro da classe, vamos incluir a configuração do processo de autenticação, que precisa ser *stateless*. Para isso, criaremos um método cujo retorno será um objeto chamado `SecurityFilterChain`, do próprio Spring.
O objeto `SecurityFilterChain` do Spring é usado para configurar o processo de autenticação e de autorização.

Por isso, no bloco da classe, vamos digitar o método `public SecurityFilterChain` nomeado `securityFilterChain()`. Logo após o nome, vamos abrir e fechar chaves.

```
SecurityConfigurations
package med.voll.api.infra.security;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.Enabl
eWebSecurity;
import org.springframework.security.web.SecurityFilterChain;
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

        public SecurityFilterChain securityFilterChain() {

        }

} COPIAR CÓDIGO
```

Neste método, precisamos devolver um objeto `securityFilterChain`, contudo, não vamos instanciar esse objeto. Nos parênteses do método receberemos a classe `HttpSecurity`, do Spring, e o chamaremos de http.

```
SecurityConfigurations
package med.voll.api.infra.security;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecuri
ty;
import
org.springframework.security.config.annotation.web.configuration.Enabl
eWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
```

```
@EnableWebSecurity
public class SecurityConfigurations {

        public SecurityFilterChain securityFilterChain(HttpSecurity
http) {

        }

} COPIAR CÓDIGO
```

Agora, dentro deste método colocaremos um `return http`, esse parâmetro http é o Spring que nos fornece. Perceba que ao digitarmos "http" aparece um pop-up com algumas opções, os métodos.

Na última opção, temos o método `http.build()`, sendo quem cria o objeto `securityFilterChain`.

Precisamos fazer algumas configurações,
no `return` digitaremos `http.csrf().disable()`. Serve para desabilitarmos proteção contra-ataques do tipo CSRF (*Cross-Site Request Forgery*).

Estamos desabilitando esse tipo de ataque porque vamos trabalhar com autenticação via tokens. Nesse cenário, o próprio token é uma proteção contra esses tipos de ataques e ficaria repetitivo.

`SecurityConfigurations`

```
package med.voll.api.infra.security;


import org.springframework.context.annotation.Configuration;
```

```
import
org.springframework.security.config.annotation.web.builders.HttpSecuri
ty;
import
org.springframework.security.config.annotation.web.configuration.Enabl
eWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

        public SecurityFilterChain securityFilterChain(HttpSecurity
http) {
            return http.csrf().disable()
        }

} COPIAR CÓDIGO
```

Após o `disable()` selecionamos "Enter", para melhorar a visualização e quebrar a linha. O próximo método será responsável por configurar a autenticação para ser *stateless*.

Digitaremos um ponto (".") e o método `sessionManagement()`, para mostrar o gerenciamento da sessão. Na sequência, mais um ponto e o método `sessionCreationPolicy()`, qual a política de criação da sessão.

```
.sessionManagement().sessionCreationPolicy() COPIAR CÓDIGO
```

No método `sessionCreationPolicy()`, passamos como parâmetro um objeto. Temos um atributo estático na própria classe `SessionCreationPolicy`, que ao digitarmos no parâmetro virá com `.ALWAYS`, e alteramos para `.STATELESS`.

```
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS) COPIAR CÓDIGO
```

Assim, ficamos com:

```
SecurityConfigurations
//código omitido


@Configuration
@EnableWebSecurity
public class SecurityConfigurations {


        public SecurityFilterChain securityFilterChain(HttpSecurity
http) {
            return http.csrf().disable()


.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)



        }


} COPIAR CÓDIGO
```

Após finalizar essa linha, clicamos na tecla "Enter". Na linha seguinte, iniciamos com um ponto `and().build()` sendo para ele criar o objeto `SecurityFilterChain`.

```
SecurityConfigurations
//código omitido


@Configuration
@EnableWebSecurity
public class SecurityConfigurations {


        public SecurityFilterChain securityFilterChain(HttpSecurity
http) {
            return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();



        }


} COPIAR CÓDIGO
```

Note que na chamada do método `csrf()` temos um sublinhado na cor vermelha. Isso acontece porque esse método lança *exception* e não colocamos `try-catch`. Na verdade, não trataremos isso e, sim, lançaremos no método.

Por isso, na assinatura do método `securityFilterChain` antes de abrirmos as chaves, vamos inserir `throws Exception`. Ou seja, se der alguma exceção é para lançar.

```
SecurityConfigurations

//código omitido


@Configuration

@EnableWebSecurity

public class SecurityConfigurations {


        public SecurityFilterChain securityFilterChain(HttpSecurity

http) throws Exception {

            return http.csrf().disable()


.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE

LESS)

            .and().build();


        }


} COPIAR CÓDIGO
```

Pronto! Essa é a primeira configuração que aplicamos nesta classe.

Estamos desabilitando o tratamento contra-ataque CSRF e, na sequência, desabilitando o processo de autenticação padrão do Spring. Isso porque estamos usando uma API Rest, e queremos que seja *stateless*.

Só faltou um detalhe. O Spring não lê o método de forma automática, precisamos incluir uma anotação. No caso, o `@Bean`, que serve exibir o retorno desse método, que estamos devolvendo um objeto `SecurityFilterChain`.

Para devolvermos um objeto para o Spring, usamos a anotação `@Bean`.

```
SecurityConfigurations
//código omitido


@Configuration
@EnableWebSecurity
public class SecurityConfigurations {


        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
            return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();


        }


} COPIAR CÓDIGO
```

Com isso, criamos a nossa classe de configuração!

Por enquanto, configuramos somente a desabilitação do CSRF e habilitamos a autenticação *stateless* do Spring. Podemos salvar as alterações feitas e abrir a aba "Run", do lado inferior esquerdo.

Perceba que ele está subindo de forma automática, tudo certo.

Voltando ao código, o que muda com essa configuração que fizemos no arquivo `SecurityConfigurations`? Com essa configuração o Spring Security não tem mais aquele comportamento padrão.

Agora, configuramos para o processo de autenticação ser *stateless*. Não será mais gerado o formulário de login e senha, quem faz isso é a nossa aplicação *mobile*, no front-end. E, também, não bloqueará mais a URL.

Nós precisamos configurar como será o processo de autenticação e autorização.

Para testar, voltaremos ao Insomnia. Note que todas as requisições estavam devolvendo o código `401 Unauthorized`, devido ao fato de não estarmos logados no Insomnia.

Agora, o esperado é que a requisição seja liberada, porque desabilitamos o bloqueio padrão. Para testar, clicamos no botão "Send" do método Detalhar Médico.

Perceba que foi devolvido o código `401 Not Found`, porque digitamos um ID inexistente no endereço: `http://localhost:8080/medicos/6999`.

Vamos testar no método Listagem de médicos, clicando nele do lado esquerdo do Insomnia. Logo após, podemos selecionar o botão "Send". Foi devolvido o código `200 OK` com as informações da Bruna Silva, Juliana Queiroz e Renato Amoedo, em "Preview".

Voltando ao método Detalhar Médico, vamos alterar o ID no final do endereço para `7`.

```
http://localhost:8080/medicos/7 COPIAR CÓDIGO
```

Depois da alteração do ID, clicamos no botão "Send". O código devolvido foi o `200 OK`, com as informações referentes ao médico de ID número 7.

Com isso, o Spring Security não bloqueia mais todas as requisições e não exibe o formulário de login.

Vamos testar no navegador, usando o seguinte endereço na URL:

```
localhost:8080/médicos COPIAR CÓDIGO
```

Note que foi devolvido o JSON com as informações da Bruna Silva, Juliana Queiroz e Renato Amoedo, sem redirecionar para o formulário de login conforme vimos em vídeos anteriores.

Desse modo, desabilitamos o processo padrão de autenticação do Spring Security. Agora o nosso processo é *stateless* e desabilitamos o processo que o Spring nos fornecia.

Isto é, se desabilitamos os recursos do Spring, precisamos configurar como será o processo de autenticação no projeto.

Mas vamos aprender isso no próximo vídeo. Até mais!

10 Mudanças na versão 3.1

ATENÇÃO!

A partir da versão **3.1** do Spring Boot algumas mudanças foram realizadas, em relação às **configurações de segurança**. Caso você esteja utilizando o Spring Boot nessa versão, ou em versões posteriores, o código demonstrado no vídeo anterior vai apresentar um aviso de **deprecated**, por conta de tais mudanças.

A partir dessa versão, o método **securityFilterChain** deve ser alterado para:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
    return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm ->
sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
} COPIAR CÓDIGO
```

11 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.STATE
LESS)
                .and().build();
    }

} COPIAR CÓDIGO
```

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***.
- Alternativa correta

  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.
- Alternativa correta

  O parâmetro passado para o método `sessionCreationPolicy` deveria ser `SessionCreationPolicy.STATEFUL`.
- Alternativa correta

  A classe não está herdando da classe `SpringSecurityConfigurationsBean`.

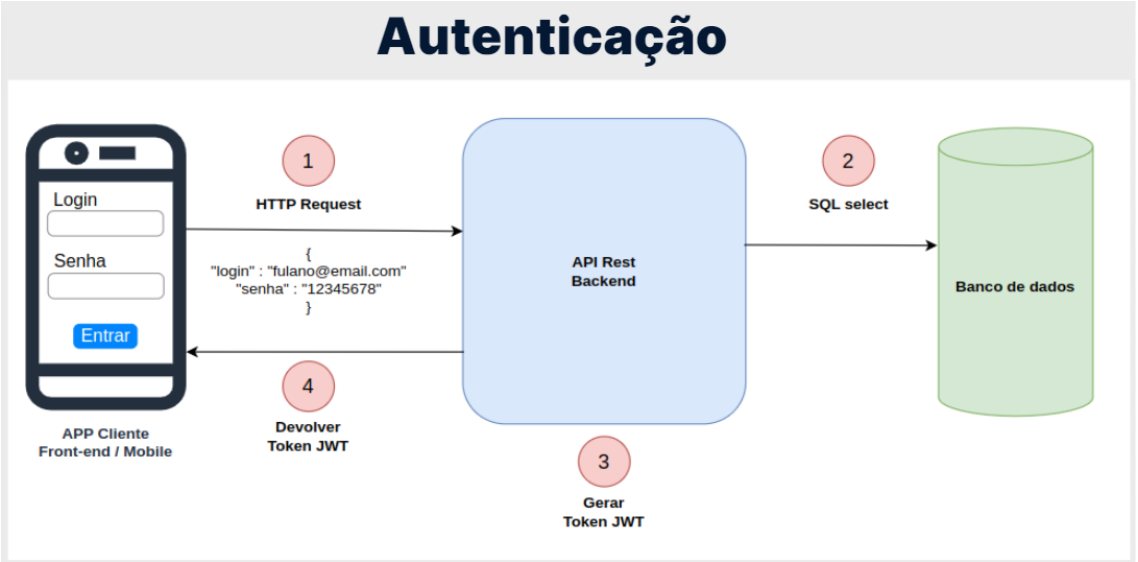
Parabéns, você acertou!

12 Controller de autenticação

Transcrição

Estamos quase finalizando o processo de autenticação, é um pouco mais complexo e são muitas mudanças que precisamos fazer no código.

Vamos relembrar o processo de autenticação na nossa API. Para isso, vamos analisar o seguinte diagrama (já visto em aulas anteriores):



Temos o cliente da nossa API, no caso um aplicativo mobile, e neste aplicativo consta o formulário de login. Ao efetuar o login clicando no botão "Entrar" do aplicativo, uma requisição é enviada para a nossa API, levando no corpo da requisição um JSON com o usuário e senha digitado na tela de login.

Com isso, a nossa API recebe essa requisição e a valida no banco de dados. Caso o usuário esteja cadastrado, é gerado o token como

resposta. Esse é o fluxo de autenticação que precisamos implementar no projeto.

Portanto, a próxima alteração que faremos no projeto é implementar o **tratamento dessa requisição**. Precisamos ter um controller para receber essas requisições, responsável por autenticar o usuário no sistema.

Ainda não construímos esse controller para receber o JSON e disparar o processo de autenticação. Esse será o nosso próximo passo!

O instrutor corrige o nome da classe de `AutenticaoService` para `AutenticacaoService`. Para isso, clicamos com o botão direito no nome, escolhemos as opções "Refactor > Rename" e digitamos o nome correto.

Para criarmos um controller responsável por disparar o processo de autenticação, do lado esquerdo do IntelliJ selecionamos o pacote `controller` e usamos o atalho "Alt + Insert".

Na aba exibida, escolhemos a opção "Java Class", e no pop-up seguinte digitamos o nome "AutenticacaoController". Logo após, podemos clicar na tecla "Enter", para criar.

Seremos redirecionados para o arquivo `AutenticacaoController`:

```
package med.voll.api.controller;


public class AutenticacaoController {


} COPIAR CÓDIGO
```

Agora, precisamos incluir alguma anotação para o Spring reconhecer essa classe. No caso, é uma classe controller, portanto, usaremos a anotação `@RestController`.

AutenticacaoController

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.RestController;


@RestController
public class AutenticacaoController {


} COPIAR CÓDIGO
```

Após incluirmos a anotação, selecionaremos "Enter" e na linha seguinte vamos colocar mais uma, sendo a `@RequestMapping()`. Nesta, precisamos passar qual a URL que o controller vai tratar, no caso será `/login`.

Portanto, ao chegar uma requisição na nossa API para `/login`, o Spring identifica que deve chamar este controller.

AutenticacaoController

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


@RestController
```

```
@RequestMapping("/login")
public class AutenticacaoController {

} COPIAR CÓDIGO
```

Dentro da classe precisamos construir um método chamado `efetuarLogin` para receber essa requisição. Os métodos são geralmente públicos, e o retorno padronizamos para receber um objeto do tipo `ResponseEntity`: `public ResponseEntity efetuarLogin`.

Após o nome do método, vamos abrir e fechar parênteses e abrir e fechar chaves.

```
AutenticacaoController
//código omitido

        public ResponseEntity efetuarLogin() {

        } COPIAR CÓDIGO
```

Precisamos incluir uma anotação no método, informando qual o verbo do protocolo HTTP. Por ser uma requisição que estamos recebendo informações, usaremos o verbo `post` e a anotação será o `@PostMapping`.

```
AutenticacaoController
package med.voll.api.controller;

import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

@RequestMapping("/login")

public class AutenticacaoController {


        @PostMapping

        public ResponseEntity efetuarLogin() {


        }

} COPIAR CÓDIGO
```

No parênteses do método `efetuarLogin()`, vamos receber um DTO com os dados que serão enviados pelo aplicativo front-end: `DadosAutenticacao dados`.

Lembrando que esse parâmetro precisa ser anotado com `@RequestBody`, já que virá no corpo da requisição. E, também, o `@Valid` para validarmos os campos com o *bean validation*.

`AutenticacaoController`

```
package med.voll.api.controller;


import jakarta.validation.Valid;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;
```

```
import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

@RequestMapping("/login")

public class AutenticacaoController {


        @PostMapping

        public ResponseEntity efetuarLogin(@RequestBody @Valid

DadosAutenticacao dados) {


        }

} COPIAR CÓDIGO
```

Perceba que "DadosAutenticacao" está escrito na cor vermelha, isso significa que ocorreu um erro de compilação. Isso porque ainda não criamos a classe `DadosAutenticacao`.

Para criarmos essa classe, selecionamos `DadosAutenticacao` e usamos o atalho "Alt + Enter". Será exibido um pop-up em que escolheremos a opção "*Created record 'DadosAutenticacao'*".

Na aba seguinte, intitulada "*Create Record DadosAutenticacao*", vamos alterar o pacote que consta no campo "*Destination package*". Atualmente estamos com `med.voll.api.controller`, modificaremos para o pacote `usuario`.

Para isso, clicamos no botão de reticências ("...") e selecionamos "domain > usuario". Logo após, apertamos o botão "Ok", no canto inferior direito.

Assim, ficamos com o seguinte caminho no campo "Destination Package": `med.voll.api.domain.usuario`. Tudo certo, podemos selecionar o botão "Ok", no canto inferior direito. Seremos redirecionados para o arquivo que acabamos de criar.

```
DadosAutenticacao
```

```
package med.voll.api.domain.usuario;


public record DadosAutenticacao() {

} COPIAR CÓDIGO
```

No parêntese do `record`, incluiremos dois campos: login e senha. Ambos sendo strings.

```
DadosAutenticacao
```

```
package med.voll.api.domain.usuario;


public record DadosAutenticacao(String login, String senha) {

} COPIAR CÓDIGO
```

Desse modo, concluímos o DTO que representa o JSON que o aplicativo mobile nos envia na requisição de autenticação. Podemos salvar este arquivo, clicando em "Ctrl + S".

Vamos voltar ao arquivo `AutenticacaoController`. Note que agora está compilando, ou seja, o `DadosAutenticacao` não está mais escrito na cor vermelha.

Assim, recebemos o DTO com o login e senha enviados pelo aplicativo mobile. Agora, precisamos consultar o banco de dados e disparar o processo de autenticação.

O processo de autenticação está na classe `AutenticacaoService`. Precisamos chamar o método `loadUserByUsername`, já que é ele que usa o `repository` para efetuar o `select` no banco de dados.

Porém, não chamamos a classe `service` de forma direta no Spring Security. Temos outra classe do Spring que chamaremos e é ela que vai chamar a `AutenticacaoService`.

No arquivo do controller, precisamos usar a classe `AuthenticationManager` do Spring, responsável por disparar o processo de autenticação.

Vamos declarar o atributo na classe `AutenticacaoController`. Será privado, e chamaremos de `manager`. Acima, incluiremos a anotação `@Autowired`, para solicitar ao Spring a injeção desse parâmetro. Não somos nós que vamos instanciar esse objeto, e sim o Spring.

```
@Autowired

private AuthenticationManager manager; COPIAR CÓDIGO
```

Para usarmos o objeto, utilizamos o método `.authenticate()` chamando o objeto `manager`, isso dentro de `efetuarLogin()`. No método `authenticate()`, precisamos passar um objeto do tipo *username authentication token*.

```
manager.authenticate(token);COPIAR CÓDIGO
```

Logo após, vamos guardar o retorno do objeto `token` em uma variável.

```
var authentication = manager.authenticate(token); COPIAR CÓDIGO
```

Este método devolve o objeto que representa o usuário autenticado no sistema.

AutenticacaoController

```
package med.voll.api.controller;


//código omitido


@RestController
@RequestMapping("/login")
public class AutenticacaoController {


        @Autowired
        private AuthenticationManager manager;


        @PostMapping
        public ResponseEntity efetuarLogin(@RequestBody @Valid
DadosAutenticacao dados) {
```

```
            var authentication = manager.authenticate(token);
        }
} COPIAR CÓDIGO
```

Perceba que está dando erro de compilação no parâmetro `token`, isso acontece porque não existe a variável token. Precisamos criá-la na linha de cima.

```
var token = new COPIAR CÓDIGO
```

Esse token é o login e a senha, e já está sendo representado no DTO `DadosAutenticacao`. No entanto, esse DTO não é o parâmetro esperado pelo Spring, ele espera uma classe dele próprio - e não uma classe do projeto.

Portanto, na variável `token` criaremos a classe que representa o usuário e a senha. Após o `new`, vamos instanciar um objeto do tipo `UsernamePasswordAuthenticationToken()` passando como parâmetro o DTO, sendo `dados.login()`, e `dados.senha()`.

```
var token = new UsernamePasswordAuthenticationToken(dados.login(),
dados.senha());COPIAR CÓDIGO
```

Temos o nosso DTO e o Spring contém um próprio, também. O método `authenticate(token)` recebe o DTO do Spring. Por isso, precisamos converter para `UsernamePasswordAuthenticationToken` - como se fosse um DTO do próprio Spring.

`AutenticacaoController`

```
package med.voll.api.controller;


//código omitido


@RestController
@RequestMapping("/login")
public class AutenticacaoController {


        @Autowired
        private AuthenticationManager manager;


        @PostMapping
        public ResponseEntity efetuarLogin(@RequestBody @Valid
DadosAutenticacao dados) {
            var token = new
UsernamePasswordAuthenticationToken(dados.login(), dados.senha())
            var authentication = manager.authenticate(token);
        }
} COPIAR CÓDIGO
```

No fim, precisamos retornar um `.ok().build()`. Isso para recebermos um código `200 OK` quando a requisição for efetuada com sucesso.

```
return ResponseEntity.ok().build(); COPIAR CÓDIGO
```

Assim, ficamos com o seguinte código:

`AutenticacaoController`

```
package med.voll.api.controller;
```

```
//código omitido


@RestController
@RequestMapping("/login")
public class AutenticacaoController {


        @Autowired
        private AuthenticationManager manager;


        @PostMapping
        public ResponseEntity efetuarLogin(@RequestBody @Valid
DadosAutenticacao dados) {
            var token = new
UsernamePasswordAuthenticationToken(dados.login(), dados.senha())
            var authentication = manager.authenticate(token);


            return ResponseEntity.ok().build();
        }
} COPIAR CÓDIGO
```

Com isso, temos o nosso controller responsável pelo processo de autenticação. Podemos salvar o arquivo e abrir a aba "Run". Perceba que ele deu erro ao tentar iniciar o projeto.

Parte do erro selecionada pelo instrutor:

```
***************
APPLICATION FAILED TO START
*************** COPIAR CÓDIGO
```

Em seguida, nos informa o motivo da falha:

Description: Field manager in med.voll.api.controller.AutenticacaoController required a bean of type 'org.springframework.security.authentication.AuthenticationManager' that could not be found.*

Isso significa que o campo `manager` na classe `Autenticacao Controller` requer um bean do tipo `Authentication Manager`, que não pôde ser encontrado.

Isto é, no momento de carregar o `AutenticacaoController`, ele não encontrou o `Authentication Manager`. Não conseguiu injetar o atributo `manager` na classe controller.

A classe `AuthenticationManager` é do Spring. Porém, ele não injeta de forma automática o objeto `AuthenticationManager`, precisamos configurar isso no Spring Security. Como não configuramos, ele não cria o objeto `AuthenticationManager` e lança uma exceção.

Faremos essa configuração.

Por ser uma configuração de segurança, faremos essa alteração na classe `SecurityConfigurations`.

SecurityConfigurations

```
package med.voll.api.infra.security;


//código omitido


@Configuration
```

```
@EnableWebSecurity
public class SecurityConfigurations {

        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();
        }

} COPIAR CÓDIGO
```

Vamos criar mais um método após o fecha chaves do `SecurityFilterChain`. Será um método público, cujo retorno é o objeto `AuthenticationManager` e o nome será `authenticationManager`.

```
public AuthenticationManager authenticationManager() COPIAR CÓDIGO
```

No parêntese deste método, receberemos um objeto do tipo `AuthenticationConfiguration` chamado `configuration`.

```
public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration) {

} COPIAR CÓDIGO
```

No retorno, teremos `configuration.getAuthenticationManager()`.

```
public AuthenticationManager

authenticationManager(AuthenticationConfiguration configuration) {



        return configuration.getAuthenticationManager();

} COPIAR CÓDIGO
```

A classe `AuthenticationConfiguration`, possui o método `getAuthenticationManager()` que cria o objeto `AuthenticationManager`. Portanto, usaremos essa classe.

Note que está gerando um erro de compilação em `getAuthenticationManager()`, isso acontece porque esse método precisa lançar uma *exception*. Portanto, na assinatura do método `AuthenticationManager` incluiremos o `throws Exception`, antes de abrir as chaves.

SecurityConfigurations

```
package med.voll.api.infra.security;



//código omitido



@Configuration

@EnableWebSecurity

public class SecurityConfigurations {



        @Bean
```

```
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();
        }

                @Bean
                public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration)
throws Exception {
                return configuration.getAuthenticationManager();
                }

} COPIAR CÓDIGO
```

Note que sumiu o sublinhado na cor vermelha abaixo de `getAuthenticationManager()`, isso significa que compilou. Esse é o método que estamos informando ao Spring como injetar objetos. Portanto, acima dele incluiremos a anotação `@Bean`.

SecurityConfigurations

```
package med.voll.api.infra.security;

//código omitido

@Configuration
```

```
@EnableWebSecurity
public class SecurityConfigurations {

        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();
        }

                @Bean
                public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration)
throws Exception {
                return configuration.getAuthenticationManager();
                }

} COPIAR CÓDIGO
```

A anotação `@Bean` serve para exportar uma classe para o Spring, fazendo com que ele consiga carregá-la e realizar a sua injeção de dependência em outras classes.

Desse modo, informamos ao Spring como criar um objeto `AuthenticationManager`. Podemos salvar o arquivo e abrir a aba "Run", do lado inferior esquerdo. Agora, sim, foi inicializado corretamente.

Com isso, criamos o nosso controller de autenticação, e já conseguimos disparar uma requisição para efetuar o login no sistema. Vamos testar no Insomnia.

Porém, criaremos uma nova requisição. No painel do lado esquerdo do Insomnia, clicamos no sinal de mais ("+"). Será exibido quatro opções:

- HTTP Request
- GraphQL Request
- gRPC Request
- New Folder

Selecionaremos a primeira opção "HTTP Request". Perceba que à esquerda será mostrada uma aba "*New Request*", clicaremos duas vezes nela para renomearmos para "Efetuar Login".

No painel central, vamos configurar a requisição. No verbo, ao invés de `get` será `post` e a URL vai ser `http://localhost:8080/login`, sendo a URL que configuramos na classe `AutenticacaoController`.

URL da requisição Efetuar Login:

```
http://localhost:8080/login COPIAR CÓDIGO
```

Abaixo do verbo, na aba "*Body*", selecionaremos para expandir. Será exibida uma seção "*Structured*" e outra "*Text*", nesta última clicaremos na opção "JSON".

Note que ao invés de "Body", agora a aba se chama "JSON". Nela, incluímos os dados de login e senha para enviar o JSON.

JSON

```
{
        "login": "ana.souza@voll.med",
        "senha": "123456"
} COPIAR CÓDIGO
```

Logo após, clicaremos no botão "Send", para disparar a requisição.

Perceba que o retorno foi o código `403 Forbidden`, ou seja, o Spring Security não bloqueou a requisição. A requisição foi processada, porém devolveu o código `403`.

No corpo na resposta, temos:

Preview

```
{
        "timestamp": "2022-10-25T19:30:40.428+00:00",
        "status": 403,
        "error": "Forbidden",
        "message": "Access Denied",
        "path": "/login"
} COPIAR CÓDIGO
```

Vamos analisar o que aconteceu no IntelliJ, na aba "Run". Perceba que o `select` foi disparado.

Retorno na aba "Run" do IntelliJ:

```
Hibernate:
        select
            v1_0.id,
            v1_0.login,
            v1_0.senha
        from
            usuarios v1_0
        where
            v1_0.login=? COPIAR CÓDIGO
```

Portanto, no arquivo `AutenticacaoController`, quando chamamos o `manager.authenticate(token)`, o próprio Spring encontrou a classe `AutenticacaoService`. Depois, chamou o método que usa o `repository` e fez a consulta no banco de dados

Assim, o processo de autenticação está sendo disparado corretamente. Isso significa que ele fez a consulta no banco de dados, mas não encontrou nenhum registro com o login e senha informados.

No Insomnia, passamos as seguintes informações:

```
"login": "ana.souza@voll.med",
"senha": "123456" COPIAR CÓDIGO
```

Isto é, não temos esse registro cadastrado no banco de dados. Isso acontece porque a nossa tabela de usuários está vazia. Por isso, precisamos incluir um usuário na tabela.

Na aba inferior do IntelliJ, abriremos um terminal. Nele, logaremos no MySQL, para isso usaremos o seguinte comando:

```
mysql -u root -p vollmed_api COPIAR CÓDIGO
```

- **vollmed_api**: nome do banco de dados.

Ao selecionarmos a tecla "Enter", será solicitada uma senha. Digitaremos "root".

```
Enter password: root
```

Com isso, entramos no banco de dados. Vamos efetuar um `select` na tabela de usuários, utilizando o comando `select * from usuarios;`.

```
select * from usuarios; COPIAR CÓDIGO
```

Como retorno, temos:

```
Empty set (0,00 sec)
```

Isso significa que a tabela está vazia. Precisamos incluir um usuário na tabela do banco de dados, e para isso, usaremos o comando `insert`:

```
insert into usuarios values (1, 'ana.souza@voll.med', '123456');

COPIAR CÓDIGO
```

Porém, em vídeos anteriores, aprendemos que deixar a senha explícita não é uma boa prática de segurança. Por isso, não vamos armazenar `123456`, e sim, algum **algoritmo de *hashing* de senhas**.

Isso para gerarmos o *hashing* da senha 123456, e salvá-la na coluna "senha" da tabela. Neste curso, usaremos o **algoritmo BCrypt**.

No caso do instrutor, ele já tem anotado o que se refere a senha 123456, no formato do algoritmo BCrypt.

Para colar, usamos o atalho "Ctrl + Shift + V".

```
insert into usuarios values (1, 'ana.souza@voll.med',
'$2a$10$Y50UaMFOxteibQEYLrwuHeehHYfcoafCopUazP12.rqB41bsolF5.');
COPIAR CÓDIGO
```

Essa é a boa prática, salvar a senha em *hashing* e não em texto aberto. Podemos selecionar "Enter", para inserir.

Como retorno, obtemos:

```
Query OK, 1 row affected (0,01 sec)
```

Agora, podemos efetuar o login. Contudo, no banco de dados estamos usando o formato BCrypt de hashing da senha. Como o Spring identifica que estamos usando o BCrypt? Não configuramos isso ainda.

Por ser uma configuração de segurança, voltaremos ao arquivo `SecurityConfigurations`.

```
SecurityConfigurations
package med.voll.api.infra.security;


//código omitido


@Configuration
@EnableWebSecurity
public class SecurityConfigurations {
```

```
        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();
        }

                @Bean
                public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration)
throws Exception {
                return configuration.getAuthenticationManager();
                }

} COPIAR CÓDIGO
```

Criaremos mais um método usando a anotação `@Bean`. Será público e devolve um objeto do tipo `PasswordEncoder`, sendo a classe que representa o algoritmo de *hashing* da senha.

```
//código omitido

@Bean
    public PasswordEncoder passwordEncoder() {
```

```
        } COPIAR CÓDIGO
```

O método não receberá nenhum parâmetro, somente retornaremos um novo `BCryptPasswordEncoder()`, sendo uma classe do Spring para instanciarmos como se fosse uma classe Java.

```
//código omitido

@Bean
    public PasswordEncoder passwordEncoder() {
                return new BCryptPasswordEncoder();
        } COPIAR CÓDIGO
```

Com isso, configuramos o Spring para usar esse algoritmo de *hashing* de senha.

Código do arquivo `SecurityConfigurations` completo:

```
package med.voll.api.infra.security;


//código omitido


@Configuration
@EnableWebSecurity
public class SecurityConfigurations {


        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        return http.csrf().disable()
```

```
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().build();
        }

                @Bean
                public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration)
throws Exception {
                return configuration.getAuthenticationManager();
                }

                @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
} COPIAR CÓDIGO
```

Podemos salvar o arquivo e voltar ao IntelliJ. Na aba "Run", conseguimos observar que foi reiniciado e não foi encontrado nenhum erro.

Agora, podemos simular a requisição de login novamente. No Insomnia, vamos alterar a senha do JSON antes de disparar a requisição. Acrescentaremos "78" no final da senha.

JSON do método Efetuar Login:

```
{
        "login": "ana.souza@voll.med",
```

```
        "senha": "12345678"
}
```

COPIAR CÓDIGO

URL do método Efetuar Login:

```
http://localhost:8080/login
```

COPIAR CÓDIGO

Logo após, clicamos no botão "Send", à direita do endereço. Note que o código devolvido foi o `500 Internal Server Error` e no corpo na resposta, obtemos:

Preview:

```
{
        "timestamp": "2022-10-25T19:30:40.428+00:00",
        "status": 500,
        "error": "Internal Server Error",
        "message": "Invalid property 'accountNonLocked' of bean class
[med.voll.api.domain.usuario.Usuaio]: Could not find field for
property during fallback access",
        "path": "/login"
}
```

COPIAR CÓDIGO

Isso aconteceu porque precisamos implementar uma interface para o Spring Security na classe usuário. Para fazer essa configuração voltaremos ao IntelliJ, no arquivo `Usuario.java`.

Usuario.java:

```
package med.voll.api.domain.usuario;


import jakarta.persistence.*
import lombok.AllArgsConstructor;
```

```
import lombok.EqualsAndHashCode;

import lombok.Getter;

import lombok.NoArgsConstructor;


@Table(name = "usuarios")

@Entity(name = "Usuario")

@Getter

@NoArgsConstructor

@AllArgsConstructor

@EqualsAndHashCode(of = "id")


public class Usuario {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String login;

    private String senha;


} COPIAR CÓDIGO
```

Para o Spring Security identificar a classe usuário do nosso projeto, precisamos informar. Por exemplo, como ele vai saber que o atributo login é o campo login? A forma para identificarmos isso é usando uma interface.

Portanto, precisamos implementar uma interface chamada `UserDetails` (própria do Spring Security) na classe que representa o usuário.

```
Usuario.java:

package med.voll.api.domain.usuario;


//código omitido


public class Usuario implements UserDetails {


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


} COPIAR CÓDIGO
```

Por ser uma interface, precisamos implementar os métodos. Note que já até gerou erro de compilação. Para isso, usaremos o atalho "Alt + Enter" e na caixa de opções escolheremos a primeira "*Implement methods*".

Na aba seguinte, será solicitado para selecionarmos os métodos que desejamos incluir no projeto. Deixaremos todos selecionados e clicaremos no botão "Ok".

No projeto, serão exibidos os métodos que deixamos selecionados.

```
Usuario.java:

package med.voll.api.domain.usuario;


//código omitido
```

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

@Override
    public String getPassword() {
        return null;
    }

    @Override
    public String getUsername() {
        return null;
    }

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

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

    @Override
    public boolean isCredentialsNonExpired() {
        return false;
```

```
    }

    @Override
    public boolean isEnabled() {
        return false;
    }
} COPIAR CÓDIGO
```

Note que há alguns objetos que devolvem dados do tipo `boolean`, todos estão com o retorno como `false`. Alteraremos todos para "true", que significa verdadeiro.

Usuario.java:

```
package med.voll.api.domain.usuario;

//código omitido

@Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
                return null;
    }

@Override
    public String getPassword() {
        return null;
    }

    @Override
    public String getUsername() {
```

```
        return null;
    }

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

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

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

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

No `isCredentialsNonExpired()`, é só caso quisermos controlar a conta do usuário: se há uma data de expiração ou se pode ter as credenciais bloqueadas. No caso, não faremos esse controle de conta, portanto, vamos devolver tudo como verdadeiro.

Isso para comunicar ao Spring que o usuário não está bloqueado, está habilitado e a conta não expirou. Assim, retornamos tudo `true`. Caso queira controlar isso, basta criar os atributos e retornar os atributos específicos que representam essas informações.

No método `getUsername()`, devolvemos qual atributo da classe representa o `Username`. Por isso, ao invés de retornar `null`, vamos devolver o `login`. Aplicaremos a mesma lógica no método `getPassword()`.

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

    @Override
    public String getUsername() {
        return login;
    } COPIAR CÓDIGO
```

Assim que informamos ao Spring que o usuário é o atributo login, e que o `getPassword` é o atributo senha.

No primeiro método criado, precisamos devolver um objeto do tipo `Collection` chamado `getAuthorities`. Caso tenhamos um controle de permissão no projeto, por exemplo, perfis de acesso, é necessário criar uma classe que represente esses perfis.

No nosso caso, não controlamos os perfis. Se o usuário estiver cadastrado, pode acessar qualquer tela sem restrições. Mas precisamos devolver para o Spring uma coleção representando os perfis.

Para isso, vamos simular uma coleção para compilarmos o projeto. Não usaremos, mas devolveremos um objeto válido para o Spring.

No retorno, ao invés de `null`, vamos inserir `List.of()`. Dentro do parêntese, criaremos um objeto do tipo `new SimpleGrantedAuthority()`, sendo a classe do Spring que informa qual o perfil do usuário.

Passaremos um perfil estático, em `SimpleGrantedAuthority()`. Por padrão, os perfis do Spring possui um prefixo, `ROLE_`, e o nome do perfil. No caso, será `USER`.

```
  @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));
    } COPIAR CÓDIGO
```

Com isso, informamos ao Spring que o perfil desse usuário é fixo, e se chama *role user*.

Deste modo, ficamos com o seguinte arquivo `Usuario.java`:

```
package med.voll.api.domain.usuario;


import jakarta.persistence.*;
```

```java
import lombok.AllArgsConstructor;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.List;

@Table(name = "usuarios")
@Entity(name = "Usuario")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")
public class Usuario implements UserDetails {

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

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));
```

```
    }

    @Override
    public String getPassword() {
        return senha;
    }

    @Override
    public String getUsername() {
        return login;
    }

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

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

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

    @Override
    public boolean isEnabled() {
```

```
        return true;
    }
} COPIAR CÓDIGO
```

Podemos salvar o arquivo. Assim, a classe usuário está seguindo o padrão do Spring.

Vamos voltar ao Insomnia, e clicar no botão "Send" da requisição Efetuar Login, novamente. Lembrando que estamos com a senha `12345678`. Foi devolvido o código `403 Forbidden`, porque fornecemos a senha errada.

Vamos ajustar a senha para `123456`:

JSON do método Efetuar Login:

```
{
        "login": "ana.souza@voll.med",
        "senha": "123456"
}
COPIAR CÓDIGO
```

Agora, sim, vamos clicar no botão "Send". Foi devolvido o código `200 OK`, e no corpo da resposta não temos nada.

Preview:
No body returned for response

Com isso, conseguimos implementar o processo de autenticação. Porém, neste caso, deveríamos retornar um token.

Se voltarmos ao arquivo `AutenticacaoController`, no método `efetuarLogin`, solicitamos que se o usuário for logado com sucesso, devemos retornar um token na resposta.
No `ResponseEntity.ok()`, devolvemos um JSON com o token.

```
AutenticacaoController

//código omitido


 @PostMapping

    public ResponseEntity efetuarLogin(@RequestBody @Valid

DadosAutenticacao dados) {

        var token = new

UsernamePasswordAuthenticationToken(dados.login(), dados.senha());

        var authentication = manager.authenticate(token);


        return ResponseEntity.ok().build();

    }


} COPIAR CÓDIGO
```

Essa parte de token e do JSON Web Token (JWT), aprenderemos na próxima aula. Neste vídeo, o nosso objetivo era fazer as configurações básicas do Spring Security e implementar todo processo de autenticação.

Na sequência, vamos entender como funciona o token e como é esse processo no projeto. Após entendermos o conceito, vamos implementar o passo a passo.

Até mais!

13 Faça como eu fiz: autenticação na API

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, implementando o processo de autenticação na API.

Opinião do instrutor

Primeiramente, você precisará adicionar o ***Spring Security*** no projeto, incluindo essas dependências no `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
COPIAR CÓDIGO
```

Depois disso, você precisará criar as classes `Usuario`, `UsuarioRepository` e `AutenticacaoService` no projeto, conforme demonstrado a seguir:

```
@Table(name = "usuarios")
@Entity(name = "Usuario")
@Getter
@NoArgsConstructor
```

```java
@AllArgsConstructor
@EqualsAndHashCode(of = "id")
public class Usuario implements UserDetails {

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

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));
    }

    @Override
    public String getPassword() {
        return senha;
    }

    @Override
    public String getUsername() {
        return login;
    }

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

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

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

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

COPIAR CÓDIGO

```
public interface UsuarioRepository extends JpaRepository<Usuario,
Long> {
    UserDetails findByLogin(String login);
}
```

COPIAR CÓDIGO

```
@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
```

```
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return repository.findByLogin(username);
    }
}
```

COPIAR CÓDIGO

Você também deve criar uma nova *migration* no projeto para a criação da tabela de usuários (**IMPORTANTE:** lembre-se de parar o projeto antes de criar a nova *migration*!):

```
create table usuarios(
    id bigint not null auto_increment,
    login varchar(100) not null,
    senha varchar(255) not null,

    primary key(id)
);
```

COPIAR CÓDIGO

Além disso, você também vai precisar criar a classe com as configurações de segurança da API:

```
@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
```

```
        return http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
                .and().build();
    }

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration)
throws Exception {
        return configuration.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
COPIAR CÓDIGO
```

Por fim, você precisará criar uma classe Controller e um DTO para lidar com as requisições de autenticação na API:

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager manager;
```

```
    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody @Valid
DadosAutenticacao dados) {
        var token = new
UsernamePasswordAuthenticationToken(dados.login(), dados.senha());
        var authenticaon = manager.authenticate(token);

        return ResponseEntity.ok().build();
    }
}
COPIAR CÓDIGO
public record DadosAutenticacao(String login, String senha) {
}
COPIAR CÓDIGO
```

Para testar a autenticação, você precisará inserir um registro de usuário em seu banco de dados, na tabela de usuários:

```
insert into usuarios values(1, 'ana.souza@voll.med',
'$2a$10$Y50UaMFOxteibQEYLrwuHeehHYfcoafCopUazP12.rqB41bsolF5.');
COPIAR CÓDIGO
```

14 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*.