01 Projeto da aula anterior

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

02 Interceptando requisições

Transcrição

Hoje nosso foco será a liberação de requisições da *API*.

No *Insomnia*, podemos ver que o processo de liberação é relativamente simples: consiste em uma nova requisição, com a *URL* "http://localhost:8080/login".

O corpo de requisição consiste em um arquivo *.json* com login e senha da pessoa.

O *token* precisa ser armazenado pelo aplicativo mobile do cliente. E, nas próximas requisições, o *token* precisará ser enviado junto a elas. De volta à *IDE*, acessaremos "src > main > java > med.voll.api > controller > MedicoController". É nesse arquivo que mapeamos a URL "/medicos".

Quando dispararmos a requisição, é por *MedicoController* que ela passará primeiro. Em `@GetMapping`, poderíamos criar a variável `token` e passar o `if (token ==null)`. Levaríamos a variável `ResponseEntity` e seu *return* para dentro da chaves.

E, por fim, fora das chaves, construiremos um *else* para devolver o bloqueio da requisição. Porém, teríamos que passar esse código em todos os métodos e em todos os *controllers*. Portanto, não faz sentido aplicar esse método.

Para otimizar o código e evitar código repetido, criaremos uma classe separada para validar o token. Assim, o *Spring* conseguirá chamá-la automaticamente antes de acessar os métodos dos *controllers*.

Obs: Será preciso chamar a nova classe antes da requisição do controller.

O *Spring* tem uma classe chamada `DispatcherSevlet`, responsável por receber todas as requisições do projeto. Ela descobre qual controller será preciso chamar em cada requisição.

Depois que a requisição passa pelo `DispatcherSevlet`, os *Handler Interceptors* são executados. Com ele, identificamos o *controller* a ser chamado e outras informações relacionadas ao *Spring*.

Já os *filters* aparecem antes mesmo da execução do *Spring*, onde decidimos se a requisição será interrompida ou se chamaremos, ainda, outro *filter*.

Portanto, precisaremos criar um *filter* ou um *interceptor* no nosso projeto, para que o código, com a validação do *token*, sejam colocado dentro deles. Ele terá, então, o papel de ser executado como o "interceptador" da requisição.

Em outras palavras, a requisição passará pelo filtro antes de cair no *controller*.

03 Para saber mais: Filters

Filter é um dos recursos que fazem parte da especificação de Servlets, a qual padroniza o tratamento de requisições e respostas em aplicações Web no Java. Ou seja, tal recurso não é específico do Spring, podendo assim ser utilizado em qualquer aplicação Java.

É um recurso muito útil para isolar códigos de infraestrutura da aplicação, como, por exemplo, segurança, logs e auditoria, para que tais códigos não sejam duplicados e misturados aos códigos relacionados às regras de negócio da aplicação.

Para criar um *Filter*, basta criar uma classe e implementar nela a interface `Filter` (pacote jakarta.servlet). Por exemplo:

```
@WebFilter(urlPatterns = "/api/**")
public class LogFilter implements Filter {


    @Override
    public void doFilter(ServletRequest servletRequest,
ServletResponse servletResponse, FilterChain filterChain) throws
IOException, ServletException {
        System.out.println("Requisição recebida em: " +
LocalDateTime.now());
        filterChain.doFilter(servletRequest, servletResponse);
    }
```

```
} COPIAR CÓDIGO
```

O método `doFilter` é chamado pelo servidor automaticamente, sempre que esse *filter* tiver que ser executado, e a chamada ao método `filterChain.doFilter` indica que os próximos *filters*, caso existam outros, podem ser executados. A anotação `@WebFilter`, adicionada na classe, indica ao servidor em quais requisições esse *filter* deve ser chamado, baseando-se na URL da requisição.

No curso, utilizaremos outra maneira de implementar um *filter*, usando recursos do Spring que facilitam sua implementação.

04 Criando o filter de segurança

Transcrição

Vamos criar um filtro no projeto, para interceptar requisições. O que queremos é fazer a validação do *token* antes que ele caia no *controller*. Para isso, voltaremos à *IDE*.

Vamos criar o projeto dentro da pasta "med.voll.api > infra > security". Vamos usar o atalho "Alt + Insert" e escolher a opção "Class". O nome da classe será "SecurityFilter".

Como o *Spring* não conseguirá carregar a classe automaticamente no projeto, precisaremos passar a anotação `@Component` no código. Vamos implementar `Filter`, de `jakarta.servlet`, passando `implements`.

Passaremos também `extends OncePerRequestFilter`. Com o atalho "Alt + Enter", vamos implementar o método `doFilterInternal`.

Para garantir que o método está sendo passado adequadamente, passaremos `System.out.println("FILTRO CHAMADO")`:

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


import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;


import java.io.IOException;


@Component
public class SecurityFilter extends OncePerRequestFilter {


    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        System.out.println("FILTRO CHAMADO");
    }
} COPIAR CÓDIGO
```

Para garantir que tudo está funcionado, abriremos a abra "run" e limparemos tudo, clicando em "botão direito do mouse > Clear All".

Vamos até o *Insomnia* disparar uma requisição para a nossa *API*. Se chamarmos, lá, a requisição "Listagem de médicos", teremos o código 200 como retorno, mas não receberemos o arquivo *.json*.
Após isso, se verificarmos a *IDE*, veremos que o filtro foi chamado. Então, o erro acontece porque o próximo filtro não foi chamado pelo filtro que criamos.

De volta ao arquivo "SecurityFilter.java", vamos apagar `System.out.println("FILTRO CHAMADO");`. No lugar dele, passaremos o código `filterChain.doFilter(request, response)`, para seguir o fluxo da requisição:

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


import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;


import java.io.IOException;


@Component
public class SecurityFilter extends OncePerRequestFilter {


    @Override
```

```
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        filterChain.doFilter(request, response);
    }
} COPIAR CÓDIGO
```

Voltando ao console e reiniciando-o, podemos disparar a requisição no *Insomnia* novamente. Depois dessas alterações, teremos o arquivo *.json* como retorno. Portanto, aprendemos a criar o nosso filtro com sucesso!

05 Recuperando o token

Transcrição

Agora vamos implementar a lógica do nosso filtro, recuperando o *token* para fazer sua validação.

No *Insomnia*, se acessarmos a aba "Auth", conseguiremos acessar as configurações referentes à autorização. Clicaremos na seta à direita da aba e vamos escolher a opção "Bearer".

Vamos deixar a *checkbox* de "Enabled" selecionada. No primeiro campo de texto, à direita de "Token", vamos passar o texto do *token*. Se dispararmos o *token*, nada acontecerá ainda, porque ele ainda não está sendo recuperado.

De volta à *IDE*, vamos recuperá-lo em "SecurityFilter.java". Em `@Override`, logo abaixo de `protected void`, passaremos `var`

`tokenJWT = recuperarToken(request)`. Com o método `recuperarToken`, recuperaremos o *token*.

Vamos nos deparar com um erro de compilação, porque esse método ainda não existe. Vamos criá-lo com o atalho "Alt + Enter".

No retorno do novo método, substituiremos `void` por `String`. Dentro dele, abriremos a variável `authorizationHeader = request.getHeader("Authorization");`.

Depois disso, faremos a validação com `if (authorizationHeader == null)`. Abaixo, passaremos `throw new RuntimeException("Token JWT não enviado no cabeçalho Authorization!")`. Fora do `if`, passaremos `return authorizationHeader;`.

Abaixo de `var tokenJWT`, passaremos `System.out.println(tokenJWT);`.

No método `recuperarToken`, na última linha do código, adicionaremos `.replace("Bearer ", "");`

```
@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        var tokenJWT = recuperarToken(request);

        System.out.println(tokenJWT);

        filterChain.doFilter(request, response);
```

```
}


private String recuperarToken(HttpServletRequest request) {
        var authorizationHeader = request.getHeader("Authorization");
        if (authorizationHeader == null) {
                throw new RuntimeException("Token JWT não enviado no
cabeçalho Authorization!");`
        }

        return authorizationHeader.replace("Bearer ", "");
} COPIAR CÓDIGO
```

Depois que salvarmos, voltaremos ao *Insomnia*. Lá, vamos disparar a requisição outra vez. Se checarmos o *console*, veremos que o *token* foi impresso sem prefixo.

06 Validando o token recebido

Transcrição

Vamos apagar `System.out.println(tokenJWT)` do método `@Override`. Nosso próximo passo será fazer a validação do *token*.

Faremos isso acessando "TokenService.java". Nesse arquivo, criaremos o método "getSubject", que receberá como parâmetro `String tokenJWT`.

Dentro das chaves, passaremos o código de validação de *token*. Vamos encontrá-lo na biblioteca https://github.com/auth0/java-jwt. O código

poderá ser encontrado na seção "Verify a JWT". Vamos substituir o algoritmo padrão pelo algoritmo do método "gerarToken", do mesmo arquivo.

Faremos algumas adaptações. Se a requisição cair em `catch`, lançaremos uma exceção, com o código `throw newRuntimeException("Token JWT inválido ou expirado!")`:

```
public String getSubject(String tokenJWT) {
        try {
                var algoritmo = Algorithm.HMAC256(secret);
                return JWT.require(algoritmo)
                                .withIssuer("API Voll.med")
                                .build()
                                .verify(tokenJWT)
                                .getSubject();
        } catch (JWTVerificationException exception) {
                throw new RuntimeException("Token JWT inválido ou
expirado!");
        }
} COPIAR CÓDIGO
```

Agora voltaremos à classe "SecurityFilter.java", para chamar o método "TokenService.java". Vamos declará-la como atributo, abaixo de `@Component`. Vamos pedir que o *Spring* injete essa classe para nós.

A anotação do atributo será `@Autowired`. Vamos importar com `private TokenService tokenService;`:

```
@Autowired

private TokenService tokenService;COPIAR CÓDIGO
```

Obs: Cuidado na hora de importar. Precisa ser o "TokenService" do nosso projeto, e não o do Spring.

No método `doFilterInternal`, já recuperamos o *token* do cabeçalho na primeira linha. Agora, precisaremos validar se o *token* está correto. Faremos isso chamando o método `getSubject` e criando também a variável `var subject = tokenService.getSubject(tokenJWT);`.

Logo, abaixo, passaremos um `System.out`, para ver se tudo está funcionando:

```
@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response, FilterChain filterChain)

        var tokenJWT = recuperarToken(request);


        var subject = tokenService.getSubject(tokenJWT);

        System.out.println(subject);


        filterChain.doFilter(request, response);

} COPIAR CÓDIGO
```

Depois de salvar, vamos até o *Insomnia* para disparar uma requisição.

Vamos disparar a requisição "Detalhar Médico". O *token* não será enviado e nós receberemos um erro 500 como resultado, porque nós não configuramos o *token*.

Vamos selecionar, agora, a opção "Listagem de médicos". Lá, clicaremos na caixa de seleção à direita de "Enabled". Com isso, habilitaremos o cabeçalho, que é o responsável pelo envio do *token*.

Se dispararmos a requisição, ela será processada com sucesso. Como retorno, teremos um arquivo *.json* com os dados do médico.

Se dermos uma olhada no console da *IDE*, veremos que o e-mail foi impresso.

Com isso, conseguimos recuperar o *token*, que vem de dentro do cabeçalho "Authorization", na requisição.

Podemos remover o `System.out`, porque verificamos que a autenticação está funcionando:

```
@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
        var tokenJWT = recuperarToken(request);

        var subject = tokenService.getSubject(tokenJWT);

        filterChain.doFilter(request, response);
} COPIAR CÓDIGO
```

No próximo vídeo, veremos o que pode substituir o `System.out` no *Spring*.

07 Autenticando o usuário

Transcrição

Agora vamos fazer as configurações, no *Spring*, que substituirão o `System.out` que removemos no vídeo anterior.

Para que o *Spring* controle a liberação, ou não, das requisições, precisaremos fazer algumas adaptações.

Lembrando que, como estamos construindo uma *API Rest*, precisamos efetuar login novamente a cada requisição. Por isso que enviamos o *token*, que funciona como a garantia de que o usuário se conectou previamente.

Vamos acessar a classe "SecurityConfigurations.java" para trabalhar em algumas alterações. No método `SecurityFilterChain`, vamos dar um enter antes de `.and().build()`. Nessa linha em branco, passaremos `.and().authorizeRequests()`.

Com o método `authorizeRequest`, configuramos a autorização das requisições. Vamos dar um enter e, na nova linha em branco, passaremos `antMatchers(HttpMethod.POST, "/login").permitAll()`. Essa definição diz para o *Spring* que a única requisição que deve ser liberada é a de login.

Abaixo, passaremos `.anyRequest().authenticated()`. Com essa duas linhas, definimos o controle de acesso na *API*.

Todas as requisições que não forem de login, não passarão. O controle disso será feito pelo próprio *Spring*:

Obs: Devido a mudanças no Spring Security, o código abaixo não funciona mais. Na atividade "authorizeRequests deprecated", localizada logo depois desse vídeo, o instrutor deixou uma explicação sobre o novo código.

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

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
                        .and().authorizeRequests()
                        .antMatchers(HttpMethod.POST,
"/login").permitAll()
                        .anyRequest().authenticated()
                        .and().build();
} COPIAR CÓDIGO
```

Vamos salvar e volta ao *Insomnia*, para executar um teste. Se tentarmos disparar a requisição "Listagem de médicos", teremos como retorno o erro "403 Forbidden", mesmo enviando o *token*.

Isso acontece porque, para o *Spring*, ainda não estamos logados. Porém, quando tentamos executar a requisição "Efetuar login", recebemos o mesmo código como retorno, indicando erro.

Vamos voltar para o código da classe "SecurityFilter.java", para entender porque todas as requisições estão sendo bloqueadas.

Em `recuperarToken`, vamos remover o `if` e passar `if (authorizationHeader != null)`. Entre as chaves, diremos que `return authorizationHeader.replace("Bearer ", "");`.

Fora das chaves, passaremos `return null`.

Em `@Override`, logo abaixo de `var tokenJWT`, passaremos um `if (tokenJWT != null)`. Dentro das chaves, passaremos `var subject = tokenService.getSubject(tokenJWT)`:

```
@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
    var tokenJWT = recuperarToken(request);

    if (tokenJWT != null) {
        var subject = tokenService.getSubject(tokenJWT);
    }

    filterChain.doFilter(request, response);
}

private String recuperarToken(HttpServletRequest request) {
        var authorizationHeader = request.getHeader("Authorization");
        if (authorizationHeader != null) {
                return authorizationHeader.replace("Bearer ", "");
```

```
        }


        return null;
} COPIAR CÓDIGO
```

Depois de salvar, vamos disparar a requisição novamente no *Insomnia*. A requisição de login será permitida e, como retorno, receberemos um *token*.

Se tentarmos disparar qualquer outra requisição, elas não serão liberadas, mesmo com o token. Isso acontece porque configuramos para que só requisições autenticadas sejam liberadas pelo *Spring*.

Para dizer que as outras requisições estão autenticadas, em outras palavras, para pedir para que o *Spring* considere que a pessoa está logada, acessaremos outra vez "SecurityFilter.java".
Para acessar o banco de dados, porém, precisamos do *repository*. Por isso, vamos declarar e injetar mais um atributo, abaixo de `private TokenService`. Passaremos `@Autowired` e, abaixo, `private UsuarioRepository repository`.

Dentro do `if (tokenJWT != null)`, vamos passar `var usuario = repository.findByLogin(subject);`, para passar uma autenticação forçada.

Também precisaremos chamar a classe `SecurityContextHolder`, que traz consigo o método
estático `getContext().setAuthentication(authentication)`.

Na linha de cima, criaremos a variável `authentication`, igualando-a à classe `UsernamePasswordAuthenticationToken(usuario, null, usuario.getAuthorities())`:

```
@Autowired
private TokenService tokenService;

@Autowired
private UsuarioRepository repository;

@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
    var tokenJWT = recuperarToken(request);

    if (tokenJWT != null) {
        var subject = tokenService.getSubject(tokenJWT);
        var usuario = repository.findByLogin(subject);

        var authentication = new
UsernamePasswordAuthenticationToken(usuario, null,
usuario.getAuthorities());


SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    filterChain.doFilter(request, response);
```

```
} COPIAR CÓDIGO
```

Agora a autenticação está feita. O *Spring* considerará que o usuário está logado.

08 authorizeRequests deprecated

Atenção!

Na versão 3.0.0 final do Spring Boot uma mudança foi feita no Spring Security, em relação aos códigos que restrigem o controle de acesso.

Ao longo das aulas o método `securityFilterChain(HttpSecurity http)`, declarado na classe **SecurityConfigurations**, ficou com a seguinte estrutura:

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

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().authorizeRequests()
            .antMatchers(HttpMethod.POST, "/login").permitAll()
            .anyRequest().authenticated()
            .and().build();
} COPIAR CÓDIGO
```

Entretanto, desde a versão **3.0.0** final do Spring Boot o método **authorizeRequests()** se tornou **deprecated**, devendo ser substituído pelo novo método **authorizeHttpRequests()**. Da mesma forma, o método **antMatchers()** deve ser substituído pelo novo método **requestMatchers()**:

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

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
            .and().authorizeHttpRequests()
            .requestMatchers(HttpMethod.POST, "/login").permitAll()
            .anyRequest().authenticated()
            .and().build();
} COPIAR CÓDIGO
```

09 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))
            .authorizeHttpRequests(req -> {
                req.requestMatchers(HttpMethod.POST,
"/login").permitAll();
                req.anyRequest().authenticated();
            })
            .addFilterBefore(securityFilter,
UsernamePasswordAuthenticationFilter.class)
            .build();
}
COPIAR CÓDIGO
```

10 Filtrando requisições

Em relação às classes `Filter`, conforme abordado ao longo dessa aula, escolha as opções que indicam os objetivos do ***Filter Chain***:

- Pode ser utilizado para bloquear uma requisição.
  É possível interromper o fluxo de uma requisição com o objeto *Filter Chain*.
- Alternativa correta
  Representa o conjunto de filtros responsáveis por interceptar requisições.
  Esse é um dos objetivos do *Filter Chain*.
- Alternativa correta

Pode ser utilizado para validar *tokens JWT*.

- Alternativa correta

  Pode ser utilizado para autenticar o usuário.

Parabéns, você acertou!

11 Testando o controle de acesso

Transcrição

Vamos executar alguns testes no *Insomnia*, para saber se tudo está funcionando corretamente.

Para simular o login na *API*, vamos disparar a requisição "Efetuar Login". A requisição não será bloqueada e, como retorno, receberemos o *token*.

Se dispararmos a requisição "Detalhar Médico" sem levar o *token*, ela não estará autenticada. Por isso, teremos como resposta o código "403 Forbidden".

Agora vamos tentar disparar essa mesma requisição, mas levando o *token*. Vamos acessar a aba "Auth" e clicar na seta à direita do nome. Selecionaremos a opção "Bearer Token".

Vamos colar o *token* da requisição "Efetuar Login" no espaço de texto à direita de "Token". Com isso, deveríamos receber o código "200", porque a requisição deveria passar. Porém, quando disparamos a requisição, continuamos recebendo o código "403 Forbidden".

Vamos voltar à IDE para entender o que aconteceu. Isso acontece porque o *Insomnia* não consegue acessar o `if` em "SecurityFilter.java". O *filter* não está funcionando.

Isso acontece porque há um segundo filtro, do *Spring*, sendo chamado na aplicação. Precisaremos, portanto, determinar a ordem de aplicação dos filtros.

Se não fizermos isso, por padrão, o *Spring* executará primeiro o filtro dele. Precisamos que ele chame primeiro o que configuramos, para verificar se o *token* está vindo e autenticar o usuário.

Faremos essa alteração na classe "SecurityConfigurations.java". Abaixo de `.anyRequest().authenticated()`, passaremos `.and().addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)`

Vamos injetar a classe, passando, acima de `@Bean`, `private SecurityFilter securityFilter`. Passaremos `@Autowired` acima desse atributo.

Vamos remover o `.and()` da última linha:

```
@Autowired
private SecurityFilter securityFilter;


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

```
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
                .and().authorizeHttpRequests()
                .requestMatchers(HttpMethod.POST,
"/login").permitAll()
                .anyRequest().authenticated()
                .and().addFilterBefore(securityFilter,
UsernamePasswordAuthenticationFilter.class)
                .build();
} COPIAR CÓDIGO
```

Depois que salvarmos, vamos limpar o console e disparar a requisição novamente. Agora deu certo. Teremos como retorno o código "200".

Se quisermos disparar as outras requisições, precisaremos clicar na seta para baixo da aba "Auth", selecionar "Bearer Token" e informar um token válido no campo de texto à direita da opção "Token".

Conseguimos implementar autorização e autenticação usando *tokens*.

12 Ainda com erro 403?

Atenção!

A utilização do Spring Security para implementar o processo de autenticação e autorização via JWT exige bastante mudanças no código, com a criação de novas classes e alteração de algumas já existentes no projeto. Tais mudanças devem ser feitas com muita

atenção, para que o processo de autenticação e autorização na API funcione corretamente.

É bem comum receber **erro 403** nas requisições disparadas no Insomnia, mesmo que você tenha implementado todo o código que foi demonstrado ao longo das aulas. Tal erro vai ocorrer somente no caso de você ter cometido algum descuido ao realizar as mudanças no projeto. Entretanto, existem diversas possibilidades que podem causar o erro 403 e veremos a seguir quais podem estar causando tal erro.

1) Erro ao recuperar o token JWT

Na classe `SecurityFilter` foi criado o método `recuperarToken`:

```
private String recuperarToken(HttpServletRequest request) {
    var authorizationHeader = request.getHeader("Authorization");
    if (authorizationHeader != null) {
        return authorizationHeader.replace("Bearer ", "");
    }

    return null;
} COPIAR CÓDIGO
```

Na linha do return, dentro do if, utilizamos o método **replace** da classe String do Java para apagar a palavra **Bearer**. Repare que existe um **espaço em branco** após a palavra Bearer. Um erro comum é esquecer de colocar esse espaço em branco e deixar o código assim:

```
return authorizationHeader.replace("Bearer", ""); COPIAR CÓDIGO
```

Verifique se você cometeu esse erro no seu código! Uma dica é utilizar também o método **trim** para apagar os espaços em branco da String:

```
return authorizationHeader.replace("Bearer ", "").trim(); COPIAR CÓDIGO
```

2) Issuer diferente ao gerar o token

Na classe `TokenService` foram criados os métodos `gerarToken` e `getSubject`:

```
public String gerarToken(Usuario usuario) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.create()
                .withIssuer("API Voll.med")
                .withSubject(usuario.getLogin())
                .withExpiresAt(dataExpiracao())
                .sign(algoritmo);
    } catch (JWTCreationException exception){
        throw new RuntimeException("erro ao gerar token jwt",
exception);
    }
}

public String getSubject(String tokenJWT) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.require(algoritmo)
                .withIssuer("API Voll.med")
```

```
                .build()
                .verify(tokenJWT)
                .getSubject();
    } catch (JWTVerificationException exception) {
        throw new RuntimeException("Token JWT inválido ou expirado!");
    }
} COPIAR CÓDIGO
```

Repare que nos dois métodos é feita uma chamada ao método **withIssuer**, da classe `JWT`:

```
.withIssuer("API Voll.med") COPIAR CÓDIGO
```

Tanto no método `gerarToken` quanto no `getSubject` o issuer deve ser **exatamente o mesmo**. Um erro comum é digitar o issuer diferente em cada método, por exemplo, em um método com letra maiúscula e no outro com letra minúscula.

Verifique se você cometeu esse erro no seu código! Uma dica é converter essa String do issuer em uma constante da classe:

```
private static final String ISSUER = "API Voll.med";


public String gerarToken(Usuario usuario) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.create()
                .withIssuer(ISSUER)
                .withSubject(usuario.getLogin())
                .withExpiresAt(dataExpiracao())
```

```
                .sign(algoritmo);
    } catch (JWTCreationException exception){
        throw new RuntimeException("erro ao gerar token jwt",
exception);
    }
}

public String getSubject(String tokenJWT) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.require(algoritmo)
                .withIssuer(ISSUER)
                .build()
                .verify(tokenJWT)
                .getSubject();
    } catch (JWTVerificationException exception) {
        throw new RuntimeException("Token JWT inválido ou expirado!");
    }
} COPIAR CÓDIGO
```

Também é possível deixar essa String declarada no arquivo application.properties e injetá-la em um atributo na classe, similar ao que foi feito com o atributo **secret**.

3) Salvar a senha do usuário em texto aberto no banco de dados

Na classe `SecurityConfigurations` ensinamos ao Spring que nossa API vai utilizar o BCrypt como algoritmo de hashing de senhas:

```
@Bean
public PasswordEncoder passwordEncoder() {
```

```
    return new BCryptPasswordEncoder();
} COPIAR CÓDIGO
```

Com isso, ao inserir um usuário na tabela do banco de dados, sua senha deve estar no formato BCrypt e não em texto aberto:

```
mysql> select * from usuarios;
+----+--------------------+--------------------------------------------
-------------------+
| id | login              | senha
|
+----+--------------------+--------------------------------------------
-------------------+
|  1 | ana.souza@voll.med |
$2a$10$Y50UaMFOxteibQEYLrwuHeehHYfcoafCopUazP12.rqB41bsolF5. |
+----+--------------------+--------------------------------------------
-------------------+
1 row in set (0,00 sec) COPIAR CÓDIGO
```

Verifique se a senha do usuário que você inseriu na sua tabela de usuários está no formato BCrypt! Um erro comum é inserir a senha em texto aberto. Por exemplo:

```
mysql> select * from usuarios;
+----+--------------------+--------+
| id | login              | senha  |
+----+--------------------+--------+
|  1 | ana.souza@voll.med | 123456 |
+----+--------------------+--------+
1 row in set (0,00 sec) COPIAR CÓDIGO
```

Se esse for o seu caso, execute o seguinte comando sql para atualizar a senha:

```
update usuarios set senha =

'$2a$10$Y50UaMFOxteibQEYLrwuHeehHYfcoafCopUazP12.rqB41bsolF5.'; COPIAR

CÓDIGO
```

Obs: No json enviado pelo Insomnia, na requisição de efetuar login, a senha deve ser enviada em **texto aberto** mesmo, pois a conversão para BCrypt, e também checagem se ela está correta, é feita pelo próprio Spring.

No caso do erro 403 ainda persistir, alguma exception pode estar sendo lançada mas não sendo capturada pela classe `TratadorDeErros` que foi criada no projeto. Isso acontece porque o Spring Security intercepta as exceptions referentes ao processo de autenticação/autorização, antes da classe `TratadorDeErros` ser chamada.

Você pode alterar a classe `AutenticacaoController` colocando um try catch no método `efetuarLogin`, para conseguir ver no console qual exception está ocorrendo:

```
@PostMapping

public ResponseEntity efetuarLogin(@RequestBody @Valid

DadosAutenticacao dados) {

    try {

        var authenticationToken = new

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

```
        var authentication =
manager.authenticate(authenticationToken);

        var tokenJWT = tokenService.gerarToken((Usuario)
authentication.getPrincipal());

        return ResponseEntity.ok(new DadosTokenJWT(tokenJWT));
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.badRequest().body(e.getMessage());
    }
} COPIAR CÓDIGO
```

Outra dica é também imprimir no console o token que está chegando na API, para você ter a certeza de que ele está chegando corretamente. Para isso, altere o método `getSubject`, da classe `TokenService`, modificando a linha que lança a `RuntimeException` dentro do bloco catch:

```
public String getSubject(String tokenJWT) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.require(algoritmo)
                .withIssuer(ISSUER)
                .build()
                .verify(tokenJWT)
                .getSubject();
    } catch (JWTVerificationException exception) {
        throw new RuntimeException("Token JWT inválido ou expirado: "
+tokenJWT);
    }
```

```
} COPIAR CÓDIGO
```

Agora será mais fácil identificar qual exception de fato está ocorrendo na API, causando o erro 403 nas requisições.

13 Para saber mais: controle de acesso por url

Na aplicação utilizada no curso não teremos perfis de acessos distintos para os usuários. Entretanto, esse recurso é utilizado em algumas aplicações e podemos indicar ao *Spring Security* que determinadas URLs somente podem ser acessadas por usuários que possuem um perfil específico.

Por exemplo, suponha que em nossa aplicação tenhamos um perfil de acesso chamado de **ADMIN**, sendo que somente usuários com esse perfil possam excluir médicos e pacientes. Podemos indicar ao *Spring Security* tal configuração alterando o método `securityFilterChain`, na classe `SecurityConfigurations`, da seguinte maneira:

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

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
        .and().authorizeHttpRequests()
        .requestMatchers(HttpMethod.POST, "/login").permitAll()
```

```
        .requestMatchers(HttpMethod.DELETE,
"/medicos").hasRole("ADMIN")
        .requestMatchers(HttpMethod.DELETE,
"/pacientes").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and().addFilterBefore(securityFilter,
UsernamePasswordAuthenticationFilter.class)
        .build();
} COPIAR CÓDIGO
```

Repare que no código anterior foram adicionadas duas linhas, indicando ao *Spring Security* que as requisições do tipo `DELETE` para as URLs `/medicos` e `/pacientes` somente podem ser executadas por usuários autenticados e cujo perfil de acesso seja **ADMIN**.

14 Para saber mais: controle de acesso por anotações

Outra maneira de restringir o acesso a determinadas funcionalidades, com base no perfil dos usuários, é com a utilização de um recurso do Spring Security conhecido como **Method Security**, que funciona com a utilização de anotações em métodos:

```
@GetMapping("/{id}")
@Secured("ROLE_ADMIN")
public ResponseEntity detalhar(@PathVariable Long id) {
    var medico = repository.getReferenceById(id);
    return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
} COPIAR CÓDIGO
```

No exemplo de código anterior o método foi anotado com `@Secured("ROLE_ADMIN")`, para que apenas usuários com o perfil **ADMIN** possam disparar requisições para detalhar um médico. A anotação `@Secured` pode ser adicionada em métodos individuais ou mesmo na classe, que seria o equivalente a adicioná-la em **todos** os métodos.

Atenção! Por padrão esse recurso vem desabilitado no spring Security, sendo que para o utilizar devemos adicionar a seguinte anotação na classe `Securityconfigurations` do projeto:

```
@EnableMethodSecurity(securedEnabled = true) COPIAR CÓDIGO
```

Você pode conhecer mais detalhes sobre o recurso de method security na documentação do Spring Security, disponível em: https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html

15 Para saber mais: Tratando mais erros

No curso não tratamos todos os erros possíveis que podem acontecer na API, mas aqui você encontra uma versão da classe `TratadorDeErros` abrangendo mais erros comuns:

```
@RestControllerAdvice
public class TratadorDeErros {


    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity tratarErro404() {
```

```
        return ResponseEntity.notFound().build();
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity
tratarErro400(MethodArgumentNotValidException ex) {
        var erros = ex.getFieldErrors();
        return
ResponseEntity.badRequest().body(erros.stream().map(DadosErroValidacao
::new).toList());
    }

    @ExceptionHandler(HttpMessageNotReadableException.class)
    public ResponseEntity
tratarErro400(HttpMessageNotReadableException ex) {
        return ResponseEntity.badRequest().body(ex.getMessage());
    }

    @ExceptionHandler(BadCredentialsException.class)
    public ResponseEntity tratarErroBadCredentials() {
        return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Credenciais
inválidas");
    }

    @ExceptionHandler(AuthenticationException.class)
    public ResponseEntity tratarErroAuthentication() {
```

```
        return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Falha na
autenticação");
    }

    @ExceptionHandler(AccessDeniedException.class)
    public ResponseEntity tratarErroAcessoNegado() {
        return
ResponseEntity.status(HttpStatus.FORBIDDEN).body("Acesso negado");
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity tratarErro500(Exception ex) {
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Erro: "
+ex.getLocalizedMessage());
    }

    private record DadosErroValidacao(String campo, String mensagem) {
        public DadosErroValidacao(FieldError erro) {
            this(erro.getField(), erro.getDefaultMessage());
        }
    }
} COPIAR CÓDIGO
```

16 Faça como eu fiz: autorizando requisições

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, implementando os códigos necessários para realizar o **controle de acesso na API**.

Opinião do instrutor

-
-

Você precisará criar uma classe `Filter`, responsável por interceptar as requisições e realizar o processo de autenticação e autorização:

```
@Component
public class SecurityFilter extends OncePerRequestFilter {

    @Autowired
    private TokenService tokenService;

    @Autowired
    private UsuarioRepository repository;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        var tokenJWT = recuperarToken(request);

        if (tokenJWT != null) {
            var subject = tokenService.getSubject(tokenJWT);
            var usuario = repository.findByLogin(subject);
```

```
            var authentication = new
UsernamePasswordAuthenticationToken(usuario, null,
usuario.getAuthorities());

SecurityContextHolder.getContext().setAuthentication(authentication);
        }

        filterChain.doFilter(request, response);
    }

    private String recuperarToken(HttpServletRequest request) {
        var authorizationHeader = request.getHeader("Authorization");
        if (authorizationHeader != null) {
            return authorizationHeader.replace("Bearer ", "");
        }

        return null;
    }

} COPIAR CÓDIGO
```

Você precisará também atualizar o código da classe `SecurityConfigurations`:

```
@Configuration
@EnableWebSecurity
public class SecurityConfigurations {
```

```
    @Autowired
    private SecurityFilter securityFilter;

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

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
                .and().authorizeRequests()
                .antMatchers(HttpMethod.POST, "/login").permitAll()
                .anyRequest().authenticated()
                .and().addFilterBefore(securityFilter,
UsernamePasswordAuthenticationFilter.class)
                .build();
    }

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

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

```
} COPIAR CÓDIGO
```

E, por fim, vai precisar atualizar o código da classe `TokenService`:

```
@Service
public class TokenService {

    @Value("${api.security.token.secret}")
    private String secret;

    public String gerarToken(Usuario usuario) {
        try {
            var algoritmo = Algorithm.HMAC256(secret);
            return JWT.create()
                    .withIssuer("API Voll.med")
                    .withSubject(usuario.getLogin())
                    .withExpiresAt(dataExpiracao())
                    .sign(algoritmo);
        } catch (JWTCreationException exception){
            throw new RuntimeException("erro ao gerar token jwt",
exception);
        }
    }

    public String getSubject(String tokenJWT) {
        try {
            var algoritmo = Algorithm.HMAC256(secret);
            return JWT.require(algoritmo)
```

```
                    .withIssuer("API Voll.med")
                    .build()
                    .verify(tokenJWT)
                    .getSubject();
        } catch (JWTVerificationException exception) {
            throw new RuntimeException("Token JWT inválido ou
expirado!");
        }
    }

    private Instant dataExpiracao() {
        return
LocalDateTime.now().plusHours(2).toInstant(ZoneOffset.of("-03:00"));
    }

} COPIAR CÓDIGO
```

17 Projeto final do curso

Caso queira, você pode baixar aqui o projeto completo implementado neste curso.

18 O que aprendemos?

Nessa aula, você aprendeu como:

- Funcionam os `Filters` em uma requisição;

- Implementar um *filter* criando uma classe que herda da classe `OncePerRequestFilter`, do Spring;
- Utilizar a biblioteca ***Auth0 java-jwt*** para realizar a validação dos *tokens* recebidos na API;
- Realizar o processo de autenticação da requisição, utilizando a classe `SecurityContextHolder`, do Spring;
- Liberar e restringir requisições, de acordo com a URL e o verbo do protocolo HTTP.

19 Conclusão

Transcrição

Parabéns por ter concluído o segundo curso de Spring Boot.

Vamos recapitular o que aprendemos.

Iniciamos com o projeto do curso anterior, e implementamos melhorias e novos recursos. Fizemos uma organização nos pacotes da aplicação, agora temos três principais, sendo eles: `controller`, `domain` e `infra`.

No `domain` ficam as classes de domínio, relacionadas com os médicos e pacientes. Já no pacote `infra` é onde estão as configurações de infraestrutura, como as de framework e as bibliotecas que usaremos no projeto.

Do lado esquerdo do IntelliJ, vamos selecionar a pasta controller. Nela, teremos mais quatro arquivos:

- `controller`
    - `AutenticacaoController`
    - `HelloController`
    - `MedicoController`
    - `PacienteController`

Vamos clicar no arquivo `MedicoController`.

Iniciamos o curso padronizando os retornos dos métodos da API com `ResponseEntity`. Fizemos alguns ajustes aplicando as boas práticas do protocolo HTTP.

MedicoController:

```
//código omitido

    @PostMapping
    @Transactional
    public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {
        var medico = new Medico(dados);
        repository.save(medico);

        var uri =
uriBuilder.path("/medicos/{id}").buildAndExpand(medico.getId()).toUri(
);

        return ResponseEntity.created(uri).body(new
DadosDetalhamentoMedico(medico));
    } COPIAR CÓDIGO
```

No método `cadastrar`, devolvemos o código `401` com o cabeçalho *location*. Aprendemos a usar o `UriComponentsBuilder`, para criarmos uma URI dinâmica e, no `return`, devolvemos o código `401 created`. Isso quando o cadastro for efetuado com sucesso.
Nos outros métodos, também alteramos para retornar `ResponseEntity` e para devolver o código mais adequado para cada situação.

Por exemplo, no método `excluir`, devolvemos o código `204` que não contém conteúdo no corpo da resposta.

```
MedicoController

   @DeleteMapping("/{id}")
    @Transactional
    public ResponseEntity excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();

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

Portanto, a primeira coisa que fizemos foi padronizar todos os métodos.

Em seguida, no pacote "infra > exception", no arquivo `TratadorDeErros`, aprendemos a criar uma classe conforme o conceito de Controller Advice do Spring Boot.

Com isso, temos um tratamento de erros. Sempre que ocorrer uma exceção dentro de qualquer classe controller, o Spring chama a classe `TratadorDeErros`.

Neste arquivo, aprendemos a lidar com os erros que podem ocorrer na API.

Tratamos o erro `404`, o `EntityNotFoundException`. Personalizamos o erro `400`, para devolver um JSON mais simples, sem muitas informações.

```
TratadorDeErros

@ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity tratarErro404() {
        return ResponseEntity.notFound().build();
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity
tratarErro400(MethodArgumentNotValidException ex) {
        var erros = ex.getFieldErrors();
        return
ResponseEntity.badRequest().body(erros.stream().map(DadosErroValidacao
::new).toList());
    } COPIAR CÓDIGO
```

Poderíamos adicionar mais métodos para tratar todos os tipos de exceções. Portanto, aprendemos a tratar os erros de forma simples, sem códigos duplicados nas classes controllers.

Além disso, dedicamos três aulas para explorarmos a utilização do **Spring Security**.

Adicionamos o Spring Security no projeto, e no pacote de `infra` temos o sub-pacote `security`. Nele, implementamos o conceito de autenticação e autorização por meio de tokens JWT (JSON Web Token).
Escrevemos bastante código. Criamos a classe `SecurityConfigurations`, aprendemos como funcionam as configurações de segurança para o Spring Security.

Neste arquivo `SecurityConfigurations`, configuramos para o Spring Security não seguir o processo padrão de autenticação e autorização.

No caso, estamos realizando a autenticação com a configuração *stateless*. E, também, configuramos para que o *hashing* da senha seja feito utilizando o algoritmo **BCrypt**.

Ensinamos, também, que o Spring precisar usar um filtro nas requisições, sendo a classe `SecurityFilter`.

`SecurityFilter`:

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


import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import med.voll.api.domain.usuario.UsuarioRepository;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthentica
tionToken;
import
org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
public class SecurityFilter extends OncePerRequestFilter {

    @Autowired
    private TokenService tokenService;

    @Autowired
    private UsuarioRepository repository;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        var tokenJWT = recuperarToken(request);

        if (tokenJWT != null) {
            var subject = tokenService.getSubject(tokenJWT);
            var usuario = repository.findByLogin(subject);
```

```
            var authentication = new
UsernamePasswordAuthenticationToken(usuario, null,
usuario.getAuthorities());

SecurityContextHolder.getContext().setAuthentication(authentication);
        }

        filterChain.doFilter(request, response);
    }

    private String recuperarToken(HttpServletRequest request) {
        var authorizationHeader = request.getHeader("Authorization");
        if (authorizationHeader != null) {
            return authorizationHeader.replace("Bearer ", "");
        }

        return null;
    }

} COPIAR CÓDIGO
```

Neste arquivo, temos a lógica para interceptar as requisições, verificar se está vindo o token JWT e logar o usuário. Isto porque aprendemos que em uma API Rest, o conceito de *stateless* significa que não temos um usuário logado. Logo, em cada requisição, precisamos autenticar o usuário para o Spring.

Fizemos essa implementação com o código o `SecurityContextHolder`.

Trecho do código selecionado pelo instrutor:

```
            var authentication = new
UsernamePasswordAuthenticationToken(usuario, null,
usuario.getAuthorities());


SecurityContextHolder.getContext().setAuthentication(authentication);
COPIAR CÓDIGO
```

Porém, fazemos isso somente se o usuário estiver logado, isto é, se antes efetuou o login, recebeu um token e está enviando esse token no cabeçalho `Authorization`. Inclusive, aprendemos a recuperar cabeçalhos do protocolo HTTP.
Logo após, criamos a classe `TokenService` para fazer a geração e validação de tokens com a biblioteca **Auth0 Java JWT**. Aprendemos a adicionar essa biblioteca no projeto.

Escrevemos o código seguindo a documentação da biblioteca, para gerarmos o token no formato JWT. Depois, validamos o token e recuperamos os dados adicionados dentro dele.

Agora, na pasta `domain`, criamos o pacote de `usuario`. Nele, geramos as classes que representam o `usuario`, a entidade JPA. Aprendemos que precisamos implementar a interface `UserDetails` do Spring, para que ele identifique que essa classe representa o usuário.

No arquivo `Usuario`, temos os métodos da interface do Spring para devolvermos os atributos referente ao `username`, `password` e qual possui os perfis de acesso deste usuário.

Criamos também o `UsuarioRepository`, e uma classe `AutenticacaoService`. Neste, implementamos um service `UserDetailsService` do Spring e, portanto, ele sabe que deve chamar essa classe quando o processo de autenticação for acionado.

Por fim, criamos um DTO chamado `DadosAutenticacao`, para representar os dados de login e senha.

```
DadosAutenticacao:

package med.voll.api.domain.usuario;


public record DadosAutenticacao(String login, String senha) {

} COPIAR CÓDIGO
```

A parte de segurança é bem extensa e se trata de um assunto bastante delicado. Autenticação e autorização é um processo um pouco complexo nas aplicações.

O Spring possui algumas vantagens relacionadas a isso, porém, mesmo assim é necessário digitarmos bastante código. O ideal é você assistir novamente às aulas que focamos no Spring Security, já que se trata de um assunto complexo e com vários detalhes.

Pratique o que viu! Temos os *challenges* na plataforma da Alura, em que você pode analisar outros projetos para aplicar esse conceito de segurança.

Com isso, concluímos o curso de **Spring Boot 3: aplique boas práticas e proteja uma API Rest**. No entanto, ainda não acabamos, temos diversos recursos que podemos usar no Spring.

Por exemplo, documentar a nossa API, implementar novas funcionalidades conforme o Trello, como a parte de agendamento das consultas. Ainda não inserimos essa funcionalidade no projeto.

Há testes automatizados que ainda não aprendemos a fazer usando o Spring Boot. Temos diversos recursos que podemos explorar.

Vamos aprender tudo isso ainda, porém, em um terceiro curso sobre o Spring Boot.

Te espero lá!