08

Testando um Controller

Transcrição

[00:00] Olá, bem-vindos de volta ao curso de Spring Boot na Alura. Agora que já vimos como funciona o teste de um Repository, como são as abordagens para você testar um Repository, seja usando o banco de dados em memória, o próprio banco de dados utilizado pela aplicação, já podemos partir para outras classes, outros componentes.

[00:22] Além do Repository, outra classe importante que temos no projeto são os Controllers. Nós temos o TopicosController, o AutenticacaoController. E se você julgar interessante você poderia escrever um teste automatizado também do Controller.

[00:37] Por exemplo, nós temos o nosso `AutenticacaoController`. Nós poderíamos testar o método `autenticar`, se ele está funcionando corretamente; se quando eu passo os dados de *login* e senha corretos ele devolve, se quando eu passo os dados incorretos ele devolve o `badRequest`, enfim. Poderíamos testar o funcionamento do Controller conforme as regras da sua aplicação.

[01:00] Vamos criar a classe de teste e escrever alguns testes para o nosso Controller, pode ser do AutenticacaoController. Então clico com o botão direito na classe "AutenticacaoController", vou em "New > Other > JUnit Test Case".

[01:15] Um erro que eu cometi no teste do Repository, eu não percebi que estava marcada a opção "New JUnit Jupiter test". Na verdade vamos usar a do meio, a "New JUnit 4 test".

[01:28] O resto está tudo igual, vai criar no source folder de teste, o nome da classe também está certo, dou um “Finish”. E ele criou a classe no source folder de teste. Agora veio tudo com `public` e veio o `@Test` do “org.junit”.

[01:45] E eu quero testar o Controller. O Controller é um componente do Spring, então em cima da classe vamos precisar daquelas mesmas anotações. Vou pegá-las do CursoRepositoryTest. A princípio vou copiar tudo, mas logo vamos discutir sobre isso.

[02:00] Vou colar e agora vem uma diferença. Agora eu não estou testando banco de dados, então eu não vou ter a anotação `@AutoConfigureTestDatabase` e nem o `@DataJpaTest`. Não é um teste de JPA da camada de persistência do Spring Data.

[02:18] Eu vou colocar uma anotação chamada `@WebMvcTest`. Quando vamos testar a camada Controller, na verdade temos que utilizar essa anotação. Assim o Spring carrega no contexto da aplicação só as classes da parte MVC. Então só os Controllers, RestController, ControllerAdvice, tudo que tem a ver com a camada MVC do projeto.

[02:48] Só que no nosso caso não vamos utilizar isso, porque como eu quero testar o AutenticacaoController e ele vai chamar o `authManager`, o `tokenService`, ele precisa chamar a parte de autenticação da nossa aplicação.

[03:05] E isso daria problema, porque o Spring não carregaria esses outros módulos da nossa aplicação, ele só carregaria o módulo MVC. Então ele não encontraria a nossa classe `AutenticacaoService`, não encontraria o restante das classes que são chamadas pelo Controller.

[03:22] E no caso eu quero testar de verdade, eu não quero simular só o Controller. Eu quero chamar o Controller, e tudo que ele chamar eu também quero simular.

[03:29] Então ao invés da `@WebMvcTest`, vamos utilizar a anotação `@SpringBootTest` mesmo, que é aquela de exemplo que já veio naquela classe padrão. Com essa anotação ele vai carregar todas as nossas classes.

[03:42] E eu vou deixar o `@ActiveProfiles("test")`, porque eu quero simular com o banco de dados vazio para não ter aquele problema.

[03:52] Só que isso vai gerar outro problema na verdade. Lembra da nossa classe `AutenticacaoController`? Na aula anterior que discutimos sobre *profile* nós colocamos a anotação `@Profile("prod")` em cima do nosso Controller.

[04:07] Lembra que essa anotação serve para dizer para o Spring carregar esse Controller apenas se o *profile* ativo for o *profile* de produção.

[04:16] Só que na classe de teste nós colocamos `@ActiveProfiles("test")`, dizendo para o Spring forçar que o *profile* ativo no momento seja o de teste.

[04:24] Então o Spring não vai carregar essa classe Controller. E nós não podemos trocar o `@Profile("prod")` para `@Profile("test")`, porque essa classe é só para carregar em produção.

[04:32] Então agora vamos ver outro recurso bacana do Spring. Na verdade, toda classe, todo componente do Spring, a anotação `@Profile` você pode passar um ou vários *profiles*. E nós só passamos um. Então vamos ter que trocar.

[04:47] Temos que colocar o `value` e entre chaves nós passamos separado por vírgula o nome dos *profiles*: `@Profile(value = {"prod", "test"})`. Então uma classe pode ser carregada para um ou mais *profiles* ao mesmo tempo, não tem problema.

[05:02] Estava sendo só carregada para o ambiente de produção, agora eu quero que essa mesma classe seja carregada também no *profile* de teste. Agora sim - Spring vai carregar isso.

[05:10] Mas não é só isso. Eu também quero carregar no ambiente de testes a nossa classe SecurityConfigurations, que tem todas as regras de segurança e do ambiente de produção. Então só copiei e colei para o Spring carregar não só no ambiente de produção, mas no ambiente de testes também.

[05:31] Agora sim, tudo vai ser carregado normalmente, podemos escrever o nosso teste. Vou apagar o teste que já vem por padrão. No nosso Controller temos aquele método para autenticar.

[05:44] E eu quero testar o cenário em que ele cai nesse *catch*, o cenário em que ele devolve um `badRequest`, que é quando eu faço passando o *login* passando um e-mail e uma senha inválidos, que não estão cadastradas no banco de dados.

[05:54] Então quero verificar se o Controller de fato está devolvendo `badRequest` nesse cenário. Na classe de teste vou renomear o método para `public void deveriaDevolver400CasoDadosDeAutenticacaoEstejamIncorretos()`. Então é isso que eu quero testar.

[06:24] Eu vou simular exatamente esse cenário. Agora não estamos mais testando persistência, estamos testando Controller. Então o Controller precisa de uma URL. Qual é a URL do Controller que eu quero chamar? Qual é o corpo da requisição? Eu preciso passar um JSON.

[06:43] Então eu estou simulando parecido com o teste que fizemos no Postman, só que agora vou fazer no JUnit, com teste automatizado. Eu vou criar as coisas que eu quero simular.

[06:53] E eu preciso de uma URI. No Java tem essa classe `uri` para representar. Vou criar um objeto do tipo URI. A URI no caso é `/auth`, porque se olharmos em cima do Controller tem o `@RequestMapping(“/auth”)`. E a requisição atende se ela for via Post. Então vai ser `URI uri = new URI(“/auth”);`.

[07:18] Vai dar um problema porque tem alguns métodos que lançam a *exception*. Vou colocar um *throws*. Então se der alguma *exception* no meio do caminho ele vai considerar que o teste falhou.

[07:30] Então já tem a URI que eu quero testar. Eu também preciso ter o JSON que vai ser o corpo da requisição. O JSON é aquele que está sendo esperado pelo método de autenticar, em que vão dois parâmetros, o e-mail e a senha, então: `String json = "{email:invalido@email.com,senha:123456}";` . Então não existe esse usuário no banco de dados.

[08:00] Tem um problema, o JSON é delimitado por aspas, então eu tenho que colocar aspas. Só que eu estou no meio da String, então eu tenho que colocar uma barra e aspas, porque senão o Java vai achar que eu estou fechando a String. Tudo que for texto eu tenho que colocar entre aspas, então fica: `String json = "{\"email\":\"invalido@email.com\", \"senha\":\"123456}\"";` .

[08:23] Provavelmente deve ter algum jeito mais elegante de fazer isso, eu estou fazendo assim só para não perder muito tempo com o que não é o foco.

[08:31] Então tenho o endereço, tenho o JSON que eu quero levar. E agora vem um problema: como eu faço para disparar essa requisição, como eu faço para chamar o Controller? Será que vou ter que dar *new* no Controller? Será que vou ter que injetar o Controller, como eu fiz no teste do Repository?

[08:47] E aqui vem uma diferença: para testar um Controller não vamos injetar o Controller, vamos injetar outra classe, então `@Autowired` . A classe que vamos injetar é uma classe utilitária do módulo de teste do Spring e ela se chama `MockMvc` : `private MockMvc mockMvc` . [09:06] Essa é a classe que faz um Mock, que simula uma requisição MVC. Então não vamos injetar diretamente o Controller, eu vou injetar como se fosse o Postman. Vamos pensar que esse é o Postman e ele vai disparar a requisição. E ela vai cair automaticamente no nosso Controller.

[09:23] Como funciona esse MockMvc? Voltando para o meu teste, eu tenho que falar `mockMvc.perform` . Eu quero que ele performe uma requisição. E eu tenho que passar como parâmetro qual é a requisição. E essa é uma parte chata, que tem umas classes meio chatas do Spring que temos que colocar.

[09:43] Então tem uma classe chamada `MockMvcRequestBuilders.post` . E eu passo para ele qual é a URI que eu quero disparar: `.perform(MockMvcRequestBuilders.post(uri));` .

[10:00] Eu preciso dizer também qual é o conteúdo da requisição. Então além de passar esse `post` , também tem que passar o `.content` , que é o conteúdo. Eu estou dizendo que a requisição é do tipo post para esse endereço, mas e o conteúdo? É aquela minha variável JSON: `.content(json));` .

[10:18] Outra coisa também: eu tenho que dizer qual é o `contentType` . Lembra do cabeçalho `contentType` , que colocávamos no Postman? O Spring precisa disso, senão vai dar problema na hora de ele chamar o nosso Controller. Então fica: `.contentType(MediaType.APPLICATION_JSON));` .

[10:42] Com isso eu estou montando a requisição. Estou falando para o Mock MVC fazer uma requisição do tipo post para a URI que é `/auth` ; com esse conteúdo, que é o JSON que tem o meu usuário inválido, não tem um usuário com esse e-mail e com essa senha; com o cabeçalho `APPLICATION_JSON` . E ele já vai conseguir disparar a requisição.

[11:03] Agora preciso fazer um `assert` para verificar se o retorno foi um 400, um `badRequest` . E tem uma diferença para verificar isso, eu não vou fazer o `assert` padrão do JUnit. Já na sequência nós colocamos `.andExpect();` . E por baixo dos panos ele já vai fazer o `assert` .

[11:28] E nós dizemos o que estamos esperando como resposta. Para isso tem mais uma classe: `.andExpect(MockMvcResultMatchers.status().is());` . E dentro eu passo qual é o status que eu estou esperando, que no caso é o `badRequest`

Eu vou passar como parâmetro, por exemplo, o 400. Então fica `.andExpect(MockMvcResultMatchers.status().is(400));` .

[12:07] Então esse é o meu teste. Eu crio o URI, crio o JSON, e só é chato porque tem a API do Mock MVC, e tem que passar o parâmetro, o JSON. Se tivesse cabeçalho tem um método chamado `Headers` , tem o `cookie` . Você vai configurando conforme o teste que você quer simular.

[12:27] E agora o meu teste está pronto, já posso rodar. Porém, vai dar um problema. Para injetar o Mock MVC, como não colocamos aquela anotação `@WebMvcTest` , temos que colocar mais uma, que é o `@AutoConfigureMockMvc` . Senão eu não consigo injetar o Mock MVC no tese. [12:47] E agora finalmente o teste está pronto. É um teste um pouco chato, mas é assim que você escreve um teste de um Controller. Vamos rodar agora e ver o que vai acontecer. Vou dar um “Run As > JUnit Test”.

[13:03] E é aquele mesmo esquema, é um teste com as anotações do Spring Boot. Quando você rodar ele vai subir o servidor, vai carregar.

[13:12] No caso, como eu usei o `@SpringBootTest` ele não vai carregar só a camada MVC, ele vai carregar todas as classes de teste, todas as camadas, a camada de persistência.

[13:23] Então vale você avaliar se você quer fazer um teste que só simula a camada MVC ignorando o resto ou se você quer testar o Controller e tudo que ele chamar você quer que ele carregue, que é esse cenário que eu estou testando.

[13:36] Eu quero que ele chame o SpringSecurity, o Repository. Eu quero que ele simule tipo o Postman. Está chegando uma requisição e eu quero simular de ponta a ponta, é um teste *end-to-end*, do Controller até o banco de dados da minha aplicação.

[13:52] E eu vou verificar se o que o Controller está devolvendo é o HTTP, é o status code. Nós poderíamos verificar também o corpo da resposta, se veio a String, o JSON corretamente, mas não vem ao caso.

[14:06] E passou. Se passou significa que para essa requisição ele devolveu 400.

[14:11] Então esse é um exemplo de como você pode testar um Controller. Então tem todas as anotações a mais; tem essa questão do MockMVC, que é a classe que vai simular a requisição, que vai cair no Controller; tem todas as classes chatas. Você poderia fazer *imports* estáticos para melhorar um pouco a legibilidade do teste, mas não é o foco o teste em si, e sim o Spring Boot.

[14:35] Então dá para melhorar, dá para complementar e colocar cabeçalho, fazer toda a simulação conforme o esperado, conforme o cenário que você quer testar.

[14:44] Então esse é um exemplo de como testar um Controller. Espero que vocês tenham gostado do vídeo.

[14:49] Além do Controller e além do Repository, temos, por exemplo, as classes de modelo no nosso projeto. Temos os DTO´s, temos os Forms. Só que essas classes não são componentes do Spring, não tem nenhuma anotação do Spring.

[15:05] Então no curso eu não vou mostrar como testar essas classes, porque para isso é só você fazer teste de unidade com JUnit puro. O Spring não vai influenciar diretamente nessas classes.

[15:15] Então no curso vamos ver só os testes do Repository e do Controller, que são os dois principais componentes diretos do Spring.

[15:27] Para esses outros tipos de classe não vamos cobrir porque não é o foco do curso, isso é teste unidade com JUnit puro.