04

Testando um repository

Transcrição

[00:00] Oi, bem-vindos de volta o curso de Spring Boot da Alura. Agora que já conhecemos um pouco sobre como funcionam os testes automatizados com o módulo de testes do Spring Boot, podemos de fato começar a testar as nossas classes.

[00:14] A ideia é que testemos as classes que criamos, que são componentes do Spring, por exemplo, os Controllers, ou Repositories, ou qualquer outro tipo de classe do gênero.

[00:23] Para começar nós vamos testar o Repository. Se entrarmos no "src/main/java", lá no pacote "repository", veremos que temos 3 Repositories, o de curso, tópico e usuário. Vamos abrir o "CursoRepository".

[00:38] Se dermos uma olhada nessa interface, lembra que estamos utilizando o Spring Data, então essa é a camada de persistência, faz a integração com o banco de dados utilizando a JPA. E nós temos um método nessa interface, o `findByNome`.

[00:54] Então poderia ser o exemplo: eu quero testar para ver se esse método está funcionando corretamente, se quando eu passo o nome de um curso ele me devolve o curso correto do banco de dados. Quando eu passo o nome de um curso que não existe, não está cadastrado, ele não me devolve nada.

[01:08] Então caso essa fosse uma classe que você julgasse ser importante de testar, você poderia criar o teste automatizado com JUnit e usando o módulo Spring Boot.

[01:18] Vamos criar um teste para nosso CursoRepository. Eu vou clicar com o botão direito em cima da interface no menu lateral esquerdo da tela, escolho a opção “New > Other” e na caixa que foi aberta vou filtrar digitando “junit” e vou escolher a opção “JUnit Test Case”.

[01:37] No caso do Eclipse ele já preenche com algumas informações. Ele já sabe que é uma classe de teste, então ele já vai jogar automaticamente no source folder de teste, o `src/test/java` ; coloca o mesmo pacote do Repository; coloca o mesmo nome da classe de teste com o sufixo “Test”. Está tudo certo, vou clicar no “Finish”.

[01:58] Ele criou a classe, se você olhar na esquerda, no Explorer, você verá que ele criou o nosso `CursoRepositoryTest` dentro do `src/test/java` . Vamos abrir a classe de teste. E ela vem vazia. Na verdade ela vem com um método de exemplo que só força o teste para falhar, mas vou apagar isso.

[02:19] Vou colocar `public` antes da classe, que ele não colocou, para não gerar a classe quando os métodos são públicos. Está com `@Test` .

[02:28] Essa foi uma classe de teste Java, então só tem Java e JUnit. Então o Eclipse não sabe que é uma classe de testes utilizando o Spring Boot.

[02:39] Então vou entrar naquela classe de teste de exemplo que vimos no último vídeo, vou copiar as duas anotações, que precisamos colocar em cima da classe para o Spring Boot subir o servidor e fornecer toda a infra necessária para rodarmos nossos testes. Então colei aquelas anotações `@RunWith` e `@SpringBootTest` . [02:57] Agora já vemos a primeira diferença. Nessa classe não vamos utilizar a anotação `@SpringBootTest` . Nós vamos utilizar outra anotação, porque como é uma classe para testar um Repository, o Spring já tem também uma especialização para testes de Repository.

[03:16] Existe uma outra anotação chamada `@DataJpaTest` . Vou dar um “Ctrl + Shift + O” para importar e ela vem do pacote “org.springframework.boot.test

Essa é uma anotação para quando você quer criar uma classe de teste, só que essa classe vai testar um Repository.

[03:33] Então a diferença é que ele já provê algumas coisas no caso de Repository. Então todos os métodos de teste já vão ter um controle de transação, para injetar um EntityManager específico para testes, então já tem todo um tratamento específico para testes de Repository. Então é mais apropriado utilizar essa anotação ao invés do `@SpringBootTest`, que é uma genérica.

[03:55] E no lugar desse método vai ficar o nosso primeiro teste, então só vou renomear esse teste. Se voltarmos na classe Repository, na interface, no caso, que estamos testando, esse é o método que eu quero testar, o `findByNome`. É dado o nome de um curso e ele tem que me devolver o objeto curso. Então quero testar para ver se isso está funcionando.

[04:14] Vou renomear o método que já vem de exemplo, chamado `test`, e eu vou chamar esse método de `public void deveriaCarregarUmCursoAoBuscarPeloSeuNome()`. Ficou até um pouco grande, mas é para deixar bem explícito qual é o cenário de teste.

[04:33] Então essa vai ser a ideia, deveria carregar um curso ao buscar pelo seu nome. Dado que eu passo o nome de um curso ele deveria conseguir carregar esse curso lá do banco de dados.

[04:48] Só para fazer um passo a passo, vou criar uma String chamada `String nomeCurso =` e vou passar o nome de um curso, por exemplo o HTML 5. Então fica `String nomeCurso = “HTML 5”;`.

[04:58] Agora já tem o nome do curso, agora eu preciso chamar aquele método `findByNome`, que é o método que eu quero testar. Só que para isso vou precisar de uma instância do Repository.

[05:08] E vem o primeiro problema: como é que eu vou dar *new* naquela interface? Como é que eu vou trazer a interface para cá, que é quem eu quero testar?

[05:16] Agora entra a vantagem: como estamos utilizando essas anotações do Spring Boot, dentro da classe de teste, toda vez que você quiser qualquer recurso do Spring você pode declarar como sendo um atributo.

[05:30] Eu vou criar um atributo: `private CursoRepository repository;`. É nesse atributo que vai ter a instância do objeto que eu quero testar.

[05:38] Só que é uma classe do Spring, eu não posso dar *new* nela; é o Spring que dá *new* nessa classe. Nós não injetávamos essa classe no Controller? Aqui também podemos injetar. Só que vamos usar a anotação `@Autowired`, que é a anotação do Spring para fazer injeção de dependências.

[05:55] Então essa é uma diferença: se fosse uma classe de testes pura, Java e JUnit, não daria para fazer injeção de dependências. Como estamos utilizando essas anotações, o Spring provê toda essa infraestrutura. Você consegue injetar qualquer classe; consegue injetar os componentes do Spring como se estivesse numa classe gerenciada pelo servidor, numa classe como se o projeto estivesse sendo executado.

[06:20] Então pronto, injetei o meu Repository, que é o objeto que eu quero testar. Agora voltando para o método de teste, eu consigo chamar o `repository.findByNome(nomeCurso);`.

[06:39] Essa chamada tem que me devolver um objeto do tipo curso. Eu passo o nome e ele me devolve um objeto curso. Vou jogar numa variável. Vou só importar a classe curso e ok: `Curso curso = repository.findByNome(nomeCurso);`

[06:58] Passei o nome “HTML 5”, porque tem esse curso no nosso banco de dados. Agora vou verificar, vou fazer os `asserts` do JUnit para verificar se ele encontrou esse curso.

[07:06] Então `Assert.assertNotNull(curso);` . Então a variável `curso` não pode vir nula, porque tem esse curso no banco de dados.

[07:16] E além disso, eu quero verificar se o curso que veio do banco de dados de fato é o curso com nome HTML 5, então vou fazer um `assertEquals` . Eu quero verificar se `nomeCurso` , que é a minha variável local, é exatamente igual a `curso.getNome` , que veio do banco. Então fica `Assert.assertEquals(nomeCurso, curso.getNome());` .

[07:35] Um teste bem simples, ele recupera o curso pelo nome, verifica se ele foi encontrado, se não está nulo e verifica se o nome do curso que foi trazido do banco é exatamente o nome do que eu estou pesquisando.

[07:46] Já está pronto meu teste, posso rodar. Para rodar, clico com o botão direito na classe, escolho a opção "Run As > JUnit Test". Então você roda da mesma maneira que qualquer classe de teste com o JUnit.

[07:57] O Spring Boot vai ajudar só na parte de você colocar essas anotações em cima da classe, e quando rodar o teste ele vai subir o servidor, vai criar o contêiner, vai inicializar tudo. E como eu tenho o contêiner executando eu posso fazer injeção de dependências, injetar os componentes como se eu estivesse com a aplicação rodando normalmente.

[08:18] Vamos dar uma olhada em como está sendo executado. Ele está subindo o servidor, no caso o Tomcat. Vai inicializar todo o contexto do Spring, criar todos os componentes, fazer o scan do nosso projeto, achar as classes Controller e as classes Repository.

[08:36] É a mesma coisa que você estar rodando normalmente o projeto, rodando aquela classe ForumApplication. Só que ele só sobe, roda todos os testes e finaliza no final.

[08:52] Agora vem um problema, que talvez você esteja pensando: ele vai subir o servidor, então esse teste vai demorar. E é exatamente isso, ele demora. Se

fosse só com Java puro, com JUnit, ele não demoraria tanto assim, ele rodaria em alguns segundos. Só que como ele precisa fazer injeção de dependências, criar todo o contêiner, todo o contexto da aplicação, ele acaba demorando.

[09:15] Deu um problema porque na verdade eu importei a anotação `@Test` errado. Eu importei ela do pacote "org.junit.jupiter", que é o novo pacote do JUnit, só que não vamos utilizar esse.

[09:26] Então eu vou apagar esse *import*. Cuidado, quando vocês forem importar uma anotação `@Test`, é do pacote "org.junit".

[09:41] Vou rodar novamente. Posso vir pela aba na direita e clicar no botão verde para ele rodar novamente os testes. E ele vai começar todo o processo novamente.

[09:51] Então como eu estava dizendo, como os testes com o Spring Boot precisam carregar o servidor, eles demoram. Se fosse com JUnit, se fosse uma classe que você estivesse dando *new* e chamando o método, em menos de um segundo ele rodaria.

[10:04] No caso, como eu dependo da infra do servidor, de todo o contexto sendo inicializado, provavelmente esse teste vai levar algo em torno de 5 a 15 segundos, dependendo da sua máquina e da infraestrutura.

[10:18] No meu caso está demorando bem mais do que isso, mas é porque o computador onde estou gravando o curso está um pouco lento. Então por conta da lentidão do meu computador o teste acaba demorando um pouco mais.

[10:32] Mas na máquina de vocês é algo em torno de 10 a 15 segundos, que é algo aceitável. A não ser que você tenha um computador bem lento, ou que esteja rodando um monte de coisa ao mesmo tempo, nesse caso ele vai travar.

[10:47] Então rodou o teste, está executando. Ele vai gerar os *logs*; ele começou uma transação de JPA, porque tem o `@JPA`, então ele cria tudo que precisa d

camada de persistência. E o teste passou normalmente, então funcionou.

[11:07] Eu consegui carregar o curso de nome HTML 5, ele foi lá no banco, viu que existia e trouxe o nome certo.

[11:14] Posso até criar outro cenário. Vou duplicar a linha do `@Test`. E se eu passar o nome de um curso que não existe, como, por exemplo, "JPA"? Não existe um curso com esse nome. Então seria outro cenário que na verdade vai ser um `assertNull`. Passei o nome de um curso que não existe, então ele deveria devolver nulo: `Assert.assertNull(curso);`.

[11:36] Vou até mudar o nome do método: `public void naoDeveriaCarregarUmCursoCujoNomeNaoEstejaCadastrado()`. Esse é outro cenário de teste.

[11:55] E agora entra a parte de testes. Não é o foco do curso ficar pensando em cenários de teste, quais são as possibilidades, enfim. Nós temos cursos na Alura para você aprender sobre essa parte de testes automatizados e de JUnit. Mas só para termos dois exemplos.

[12:12] Agora você deve estar se perguntando da onde ele tirou esse curso HTML 5, porque estou dizendo que não tem o curso JPA, de onde ele está tirando essas informações? Ele está carregando do nosso banco de dados. Mas onde está o nosso banco de dados?

[12:31] Vou dar um "Ctrl + Shift + R". Lembra que temos no nosso projeto aquele arquivo `data.sql`? Que toda vez que rodamos o projeto ele cria, roda todos esses *inserts*. Então está justamente no `data.sql`.

[12:46] Toda vez que ele roda o projeto, mesmo no teste – porque quando rodamos o teste ele vai subir a aplicação – mesmo quando rodamos com um teste automatizado, o Spring Boot vai ler o `data.sql`.

[13:01] E lembra que toda vez que rodamos o projeto, eu tinha configurado para ele sempre inserir esses registros. Então tem dois alunos, tem dois perfis, tem dois cursos.

```
INSERT INTO USUARIO(nome, email, senha) VALUES('Aluno',
'aluno@email.com',
'$2a$10$sFKmbxbG4ryhwPNx/l3pgOJSt.fW1z6YcUnuE2X8APA/Z3NI/oSpq')

INSERT INTO USUARIO(nome, email, senha) VALUES('Moderador',
'moderador@email.com',
'$2a$10$sFKmbxbG4ryhwPNx/l3pgOJSt.fW1z6YcUnuE2X8APA/Z3NI/oSpq')


INSERT INTO PERFIL(id, nome) VALUES(1, 'ROLE_ALUNO');
INSERT INTO PERFIL(id, nome) VALUES(2, 'ROLE_MODERADOR');

INSERT INTO USUARIO_PERFIS(usuario_id, perfis_id) VALUES(1,
1);
INSERT INTO USUARIO_PERFIS(usuario_id, perfis_id) VALUES(2,
2);

INSERT INTO CURSO(nome, categoria) VALUES('Spring Boot',
'Programação');
INSERT INTO CURSO(nome, categoria) VALUES('HTML 5', 'Front-
end');

INSERT INTO TOPICO(titulo, mensagem, data_criacao, status,
autor_id, curso_id) VALUES('Dúvida', 'Erro ao criar
projeto', '2019-05-05 18:00:00', 'NAO_RESPONDIDO', 1, 1);
INSERT INTO TOPICO(titulo, mensagem, data_criacao, status,
autor_id, curso_id) VALUES('Dúvida 2', 'Projeto não
compila', '2019-05-05 19:00:00', 'NAO_RESPONDIDO', 1, 1);
INSERT INTO TOPICO(titulo, mensagem, data_criacao, status,
autor_id, curso_id) VALUES('Dúvida 3', 'Tag HTML', '2019-05-
05 20:00:00', 'NAO_RESPONDIDO', 1, 2);
```

COPIAR CÓDIGO

[13:10] Toda vez que rodamos o projeto na nossa base de dados vai ter sempre um curso HTML 5 e um curso Spring Boot. Então é por isso que ele encontrou os registros. Ele já rodou meu teste e os dois passaram. Então ele encontrou o HTML 5, não encontrou o JPA, porque não tem esse curso.

[13:28] Esse era o objetivo dessa aula, mostrar para vocês como fazer um teste de um Repository.

[13:34] Talvez você esteja se perguntando se é uma boa ideia eu já ter um banco de dados preenchido, se isso não pode atrapalhar nos testes. Mas essa vai ser uma discussão que vamos fazer no próximo vídeo, sobre outras estratégias no caso de você querer testar um Repository, porque tem algumas outras abordagens distintas.

[13:51] Mas isso fica para o próximo vídeo. Espero que vocês tenham gostado desse. Vejo vocês no próximo vídeo. Um abraço e até lá.