06

Testando Controllers

Transcrição

Perceba que sempre que adicionamos um recurso na aplicação, sempre vamos até o navegador e verificamos se esta se comporta da forma que esperamos. Este é um processo natural no devenvolvimento web, mas e se quisessemos realizar este tesde dentro do próprio *Eclipse*, como faríamos?

Por exemplo, como podemos garantir que o `HomeController` em seu método `index` realmente esta retornando como resposta a *view* `home.jsp` ? E como podemos garantir que os produtos estão sendo carregados para esta página? A resposta é simples! Por meio dos testes automatizados!

Atenção: Caso não tenha conhecimentos sobre testes de software, não se preocupe. Neste curso, veremos apenas alguns pontos sobre isso, apesar de não ser o foco deste curso. Por isso, recomendamos que faça os cursos de testes em Java (https://cursos.alura.com.br/category/programacao) disponíveis aqui na Alura.

Da mesma forma que fizemos os testes do `ProdutoDAO` , criando a classe `ProdutoDAOTest` . Criaremos testes para a classe `ProdutosController` com a classe `ProdutosControllerTest` no *Source Folder* de testes e no pacote de *controllers*.

O primeiro teste será fazer uma requisição para a página inicial da nossa aplicação e verificar se a *view* `home.jsp` está realmente sendo retornada. O primeiro passo ao criar a classe é fazer as devidas configurações, bem parecidas com as da classe `ProdutoDAOTest` .

```
@WebAppConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class, AppWebConfiguration.class, DataSourceC
@ActiveProfiles("test")
public class ProdutosControllerTest {

}
```

As únicas diferenças são que na anotação `@ContextConfiguration` em vez de carregarmos a classe `ProdutoDAO` , carregamos a classe que tem todas as configurações de *MVC* da aplicação, `AppWebConfiguration` . A outra diferença é a presença da anotação `@WebAppConfiguration` que faz o carregamento das demais configurações de *MVC* do *Spring*.

Para que possamos fazer requisições sem o uso de um navegador, precisamos de um objeto capaz de simular este procedimento. Estes objetos que simulam comportamentos são conhecidos como Mock s. E para a criação de um *mock* no *Spring*, precisaremos definir um contexto. Por isso, criaremos mais dois atributos na classe `ProdutosControllerTest` .

```
@Autowired
private WebApplicationContext wac;

private MockMvc mockMvc;
```

O *Spring* já conhece o contexto da aplicação, por isso o atributo `WebApplicationContext` é anotado com `@Autowired`. O Objeto `MockMvc` será o objeto que fará as requisições para os *controllers* da nossa aplicação.

Apesar do *Spring* criar o objeto de contexto da aplicação de forma automática. Este não cria o objeto *Mock*, mas facilita essa tarefa através de classes auxiliadoras. Para a criação do objeto `MockMvc`, definiremos um método que será executado antes dos testes e instanciaremos o objeto usando a classe `MockMvcBuilders`. Iremos fornecer para o método, que se chamará `setup`, o contexto `webAppContextSetup`.

```
@Before
public void setup(){
    mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
}
```

A anotação `@Before` é quem faz com que o método seja chamado antes que qualquer teste seja executado. Métodos de *setup* são muito comuns para carregamento de recursos e definição de configurações que devem ser executadas antes de qualquer teste. A classe `ProdutosControllerTest` até então se encontra com o seguinte código:

```
@WebAppConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class, AppWebConfiguration.class, DataSourceC
@ActiveProfiles("test")
public class ProdutosControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup(){
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

}
```

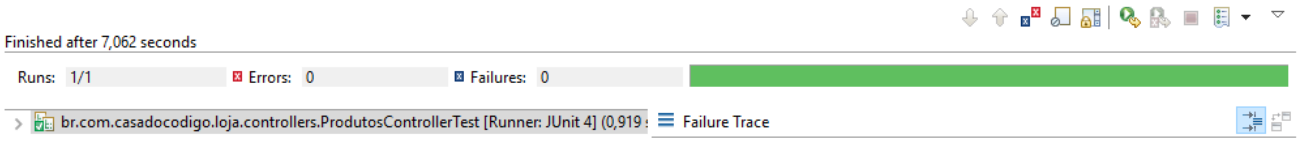
Com tudo configurado, faremos o primeiro teste! Para esta tarefa, definiremos um novo método que se chamará `deveRetornarParaHomeComOsLivros` que usará o método `perform` do objeto `mockMvc` para simular uma requisição. Mas para tal, precisaremos de algum outro objeto que construa um *Request*, objeto de requisição. Este objeto será fornecido pela classe `MockMvcRequestBuilders` por meio do método `get` para o qual será passado o caminho da requisição.

```
@Test
public void deveRetornarParaHomeComOsLivros() {
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
}
```

Com a requisição feita, nos resta verificar o resultado da mesma por meio do método `andExpect` do objeto `mockMvc` que receberá o resultado do método `forwardedUrl` da classe `MockMvcResultMatchers` no qual verificará se foi feito um redirecionamento no servidor para a *view* localizada em `WEB-INF/views/home.jsp`. Assim teremos:

```
@Test
public void deveRetornarParaHomeComOsLivros() throws Exception{
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
           .andExpect(MockMvcResultMatchers.forwardedUrl("/WEB-INF/views/home.jsp"));
}
```

Agora, ao executarmos o teste, veremos que o mesmo passa!



E para verificarmos a presença dos produtos na resposta da requisição? Faremos praticamente a mesma coisa que já fizemos: usaremos o `andExpect` novamente e também a classe `MockMvcResultMatchers`, mas com a pequena diferença que, desta vez, usaremos o método `model` para que consigamos obter informações sobre o objeto retornado pela requisição e neste objeto, verificaremos a existencia do atributo `produtos` utilizando o método `attributeExists`.

```
@Test
public void deveRetornarParaHomeComOsLivros() throws Exception{
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
           .andExpect(MockMvcResultMatchers.model().attributeExists("produtos"))
           .andExpect(MockMvcResultMatchers.forwardedUrl("/WEB-INF/views/home.jsp"));
}
```