02

Teste de requisições assíncronas

Transcrição

Nesta aula conheceremos algumas dicas sobre situações que podem fazer com que os testes falhem, mesmo que a aplicação funcione corretamente quando a testamos manualmente. Isso acontece, especialmente, em aplicações que trabalham muito com JavaScript, algo comum hoje em aplicações front-end. Nesses casos, pode acontecer que o Selenium não encontre um elemento na página devido a algum código JavaScript ou outra coisa que tenha travado a página antes que tal elemento fosse carregado e exibido.

Podemos simular um exemplo na página de login (`login.html`). Antes do navegador desenhar o campo "Senha", podemos incluir um código JS (`<script>window.stop()</script>`) que faz com que o navegador pare de carregar a página, simulando um travamento.

```
<div class="card mb-3">
    <form id="login-form" th:action="@{/login}" class="card-body'
        <div class="form-group">
            <label for="username">Usuário</label>
            <input id="username" name="username"  class="form-cor
        </div>

        <script>window.stop()</script>
        <div class="form-group">
            <label for="password">Senha</label>
```

```
        <input id="password" type="password" name="password"
    </div>
```

COPIAR CÓDIGO

Dessa forma, o campo "Senha" e todo conteúdo após a execução do `window.stop()` não serão desenhados. Claro, nesse caso estamos forçando a parada manualmente, mas poderíamos ter um código JavaScript que chama um API ou serviço externo que cause lentidão, ou uma requisição AJAX.

Ao executarmos os testes de `LoginTest`, somente o campo "Usuário" será exibido na tela. Na aba do JUnit, perceberemos que nossos testes falharam e retornaram uma exceção, `NoSuchElementExceptionError`, ou seja, um elemento - nesse caso o de ID `password` - não foi encontrado.

Se acessarmos manualmente a página de login (http://localhost:8080/login (http://localhost:8080/login)), teremos que o campo "Senha" realmente não é exibido por conta do nosso código JavaScript. Esse é um cenário que pode acontecer em seus testes, e pode fazer com que às vezes eles falhem e às vezes passem.

Uma maneira de lidar com isso é configurar o *timeout* do Selenium, fazendo com que ele espere um determinado tempo antes de acusar uma falha no teste.

Na nossa classe `PageObect`, assim que chamamos o construtor e criamos o `browser`, chamaremos um método `browser.manage()` que permite fazer algumas configurações no WebDriver, uma delas sendo justamente os *timeouts*. Utilizando o método `timeouts()`, conseguiremos fazer configurações como a do método `implicitlyWait()`, que permite ajustar um tempo de espera que deverá existir antes de lançar um erro na busca de um elemento na página.

No nosso caso, passaremos os parâmetros `5`, que é o valor, e `TimeUnit.SECONDS`, representando a unidade de tempo que estamos definindo.

```
public PageObject(WebDriver browser) {
    System.setProperty("webdriver.chrome.driver", "/drivers/chror

    if (browser == null) {
        this.browser = new ChromeDriver();
    } else {
        this.browser = browser;
    }

    this.browser.manage().timeouts().implicitlyWait(5, TimeUnit.S
}
```

COPIAR CÓDIGO

Com isso, toda vez que o Selenium buscar um elemento na página, ele esperará no máximo 5 segundos, caso não encontre tal elemento, antes de lançar um erro. Existem outros *timetouts* que podem ser configurados, como o `pageLoadTimeout()`. Quando nossas páginas demoram um pouco a carregar, podemos, por exemplo, fazer com que o Selenium espere `10` segundos antes de efetuar os testes e lançar os erros.

Esses *timeouts* são bastante úteis quando trabalhamos com AJAX, com requisições assíncronas e com JavaScript, que podem desacelerar ou travar o carregamento da página. Quando estamos cientes de que nosso projeto trabalha com essas tecnologias, podemos, preventivamente, configurar os *timeouts* de acordo com o tempo limite que desejamos esperar. Como em nosso projeto não temos essa situação, vale como dica para aplicações futuras.