Testes de Sistema - parte 2

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 8 set. 2024, 20:11
Livro:	Testes de Sistema - parte 2		

Descrição

Continuando a discussão que começamos quando falamos de limpar a aplicação, você não deve ter medo de criar código para facilitar a testabilidade da sua aplicação como um todo. Isso pode significar inclusive o desenvolvimento de serviços web que criam cenários para suas aplicações.

Índice

1. Como limpar o banco em testes de sistema?

Quando tínhamos testes de unidade, limpar o cenário era fácil, afinal eram apenas objetos na memória. Depois, discutimos testes de integração, e ali ficou um pouco mais complicado, pois precisávamos limpar um banco de dados. Agora, no teste de sistema, é tudo ainda mais difícil. Não temos acesso “aos detalhes” do sistema, como código, banco de dados etc.

Mas, do mesmo jeito que fazíamos no teste de unidade, e favorecíamos a testabilidade na hora da criação da classe, precisamos criar maneiras da aplicação facilitar o teste. Já que fazer a aplicação estar em um estado inicial é essencial para o teste, ela deve então prover algum serviço para tal. Em aplicações web, uma simples URL, /reseta-aplicacao, que leva a aplicação ao seu estado inicial, é suficiente. Aí, basta o teste, antes de navegar pela aplicação, fazer uma requisição para esse endereço.

Mais para frente, discutiremos sobre APIs para criação de cenários. Neste momento, perceba a aplicação deve facilitar a escrita de testes, mesmo que isso implique na produção de mais código e/ou serviços para tal.

2. Requisições Ajax

Vamos agora à página de detalhes de um leilão. Para isso, basta clicar em “Exibir” ao lado de qualquer leilão. Nessa página, o usuário pode ver os dados de um leilão e fazer lances nele.

Veja o que acontece quando damos um lance. A página não “pisca”, mas é atualizada! Isso acontece porque nossa aplicação web fez uso do que chamamos de Ajax. Ou seja, a requisição aconteceu, mas ela foi por baixo dos panos, sem o usuário ver. Precisamos testar essa ação, e levar em conta que ela é executada via Ajax.

Nosso teste deve ser algo parecido com isso:

```
    @Test
    public void deveFazerUmLance() {
        leiloes.detalhes(1);
        lances.lance("José Alberto", 150);
        assertTrue(lances.existeLance("José Alberto", 150));
    }
```

Não há segredo no método lance() do DetalhesDoLeilaoPage. Precisamos apenas selecionar um usuário no combo e preencher um valor.

Nada de novo até então:

```
public class DetalhesDoLeilaoPage {

    private WebDriver driver;

    public DetalhesDoLeilaoPage(WebDriver driver) {
        this.driver = driver;


    public void lance(String usuario, double valor) {
        WebElement txtValor
                = driver.findElement(By.name("lance.valor"));
        WebElement combo
                = driver.findElement(By.name("lance.usuario.id"));
        Select cbUsuario = new Select(combo);
        cbUsuario.selectByVisibleText(usuario);
        txtValor.sendKeys(String.valueOf(valor));
    }
}
```

Devemos agora submeter o formulário. Mas, dessa vez, não vamos submeter pela caixa de texto. Veja o código-fonte da página! O botão não submete o formulário, ele é um simples “button” do HTML. Precisamos clicar nele. Para isso, basta usar o método click():

```
public class DetalhesDoLeilaoPage {

    private WebDriver driver;

    public DetalhesDoLeilaoPage(WebDriver driver) {
        this.driver = driver;
    }

    public void lance(String usuario, double valor) {
        WebElement txtValor
                = driver.findElement(By.name("lance.valor"));
        WebElement combo
                = driver.findElement(By.name("lance.usuario.id"));
        Select cbUsuario = new Select(combo);
        cbUsuario.selectByVisibleText(usuario);
        txtValor.sendKeys(String.valueOf(valor));
        driver.findElement(By.id("btnDarLance")).click();
    }
}
```

Agora vamos verificar se o novo lance efetivamente apareceu na tela, também da forma que já conhecemos:

```
    public boolean existeLance(String usuario, double valor) {
        return driver.getPageSource().contains(usuario)
                && driver.getPageSource().contains(String.valueOf(valor));
    }
```

Mas temos um problema. Uma requisição Ajax acontece por trás do panos, e pode levar alguns segundos para terminar. Se invocarmos o método existeLance() e a requisição ainda não tiver voltado, o método retornará falso! Precisamos fazer com que esse método aguarde até a requisição terminar.

Pediremos ao Selenium para que espere alguns segundos até que a requisição termine. Isso é conhecido por explicit wait. Para usá-lo, precisamos utilizar a classe WebDriverWait. No construtor, ela recebe o driver e a quantidade máxima de segundos a esperar. Em seguida, ela recebe a condição que faz esse tempo parar. No nosso caso, a condição é se o texto aparecer. Para isso, faremos uso de uma outra classe, a ExpectedConditions, e passaremos a expectativa correta:

```
        Boolean temUsuario
                = new WebDriverWait(driver, 10)
                        .until(ExpectedConditions
                                .textToBePresentInElement(
                                        By.id("lancesDados"), usuario)
                        );
```

Vamos colocar o explicit wait dentro do método existeLance(), completando o código para verificar também se existe o valor:

```
    public boolean existeLance(String usuario, double valor) {
        Boolean temUsuario
                = new WebDriverWait(driver, 10)
                        .until(ExpectedConditions
                                .textToBePresentInElement(
                                        By.id("lancesDados"), usuario)
                        );
        if (temUsuario) {
            return driver.getPageSource().contains(String.valueOf(valor));
        }
        return false;

    }
```

Agora, vamos ao teste. Repare que o cenário é maior: precisamos criar 2 usuários (um que é o dono do produto e outro para dar lance no produto) e um leilão. Vamos também limpar o cenário para facilitar nossa vida. Tudo isso no @Before:

```
public class LanceSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();
        driver.get("http://localhost:8080/apenas-teste/limpa");
        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo()
                .cadastra("Paulo Henrique", "paulo@henrique.com");
        usuarios.novo()
                .cadastra("José Alberto", "jose@alberto.com");
        leiloes = new LeiloesPage(driver);
        leiloes.visita();
        leiloes.novo().preenche("Geladeira", 100,
                "Paulo Henrique", false);
    }

    @Test
    public void deveFazerUmLance() {
        DetalhesDoLeilaoPage lances = leiloes.detalhes(1);
        lances.lance("José Alberto", 150);
        assertTrue(lances.existeLance("José Alberto", 150));
    }
}
```

Falta só um detalhe: o método detalhes(1) do LeiloesPage não está implementado. Precisamos fazê-lo chegar à página de detalhes do produto para darmos os lances. Vamos pegar o detalhes do primeiro item da lista e pedir ao Selenium para achar todos os elementos da página com o texto “exibir”, e clicar no primeiro deles. Cuidado com maiúsculas e minúsculas! O link é “exibir” (tudo minúsculo). O Selenium é case-sensitive. Vamos lá:

```
    public DetalhesDoLeilaoPage detalhes(int posicao) {
        List<WebElement> elementos
                = driver.findElements(By.linkText("exibir"));
        elementos.get(posicao - 1).click();
        return new DetalhesDoLeilaoPage(driver);
    }
```

3. Builders em testes de sistema

Perceba que algumas páginas precisam de cenários complicados para começarem. Por exemplo, a página de teste de lances precisa de dois usuários e um lance cadastrado. Para deixar nosso código de teste mais simples ainda, sem precisar escrever todo o código necessário para levantar cada um desses cenários, podemos escondê-lo em uma classe mais simples.

Por exemplo, veja a classe CriadorDeCenario:

```
class CriadorDeCenarios {

    private WebDriver driver;

    public CriadorDeCenarios(WebDriver driver) {
        this.driver = driver;


    public CriadorDeCenarios umUsuario(String nome, String email) {
        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo().cadastra(nome, email);
        return this;
    }

    public CriadorDeCenarios umLeilao(String usuario,
            String produto,
            double valor,
            boolean usado) {
        LeiloesPage leiloes = new LeiloesPage(driver);
        leiloes.visita();
        leiloes.novo().preenche(produto, valor, usuario, usado);
        return this;
    }
}
```

Com essa classe, podemos fazer com que os testes da classe Lance fiquem mais simples ainda de serem escritos.

```
public class LanceSystemTest {

    @Before
    public void criaCenario() {
        lances = new DetalhesDoLeilaoPage(driver);
        new CriadorDeCenarios(driver)
                .umUsuario("Paulo Henrique", "paulo@henrique.com")
                .umUsuario("José Alberto", "jose@alberto.com")
                .umLeilao("Paulo Henrique", "Geladeira", 100, false);
    }
// código continua aqui...
}
```

Isso nos garante que, caso um dia o cenário para criar um usuário mude, mudaremos apenas o CriadorDeCenarios.

Uma boa prática é sempre utilizar classes como essa quando precisamos montar um cenário que depende de outras páginas. Assim, o teste de uma página não precisa saber como a outra página funciona. É a ideia do encapsulamento (conceito da orientação a objetos) aplicado ao nosso código de testes.

4. API para criação de cenários

Continuando a discussão que começamos quando falamos de limpar a apli- cação, você não deve ter medo de criar código para facilitar a testabilidade da sua aplicação como um todo. Isso pode significar inclusive o desenvolvimento de serviços web que criam cenários para suas aplicações.

Você já percebeu ao longo deste cirza que criar cenários é geralmente a parte mais trabalhosa do teste de qualquer nível. Criar entidades com valores predefinidos, e depois na integração, persisti-los, não é fácil. Em um teste de sistema, isso é exponencial: você precisa criar 2, 3, 4 cenários maiores, para conseguir testar aquela funcionalidade que faz uso de todos os cenários juntos.

Portanto, não é má ideia pensar em serviços web que montem os cenários que são requeridos pelo teste. Serviços web podem ter o formato ideal e fazer o que você achar melhor. Por exemplo, a aplicação pode prover serviços web que recebem as entidades serializadas (em XML, JSON, ou qualquer outro formato) e os insira no banco. Dessa forma, os testes montam os objetos (por meio de Test Data Builders), serializa-os e envia para a aplicação web. Outro caso seria a aplicação ter serviços já específicos, do tipo “cria notas fiscais de pessoas físicas”, que já instanciariam e persistiriam tudo o que for necessário.

Novamente, não há regra pra isso. Faça com que o código seja fácil de ser mantido e fácil de ser consumido pelos seus testes. Não pense que é besteira investir nessa infraestrutura. Ela lhe economizará muito tempo no futuro.

5. Referência Bibliográfica

Extraído de:
Aniche, M. **Testes Automatizados de Software**. São Paulo: Casa do Código, 2015.