Protractor

Lições sobre testes end-to-end automatizados

Casa do Código

WALMYR FILHO

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

Revisão técnica

Carlos Panato

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

ISBN

Impresso e PDF: 978-85-5519-228-9

EPUB: 978-85-5519-229-6

MOBI: 978-85-5519-230-2

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

O uso do logo oficial do Protractor na arte da capa deste livro foi autorizado pelo time do Protractor e Angular, sob a licença CC 4.0 (Creative Commons).

AGRADECIMENTOS

Dedico este livro aos meus pais e minha esposa, que sempre me apoiaram em minhas iniciativas.

Agradeço a todos os colegas que me acompanharam na carreira de desenvolvimento de software e que fizeram parte de meus aprendizados como profissional focado em qualidade de software.

Agradeço ao Dionatan Moura pelo apoio e incentivo em escrever este livro, além das diversas dicas que me ajudaram a fazer isso acontecer.

Agradeço aos beta readers, que me ajudaram revisando o conteúdo do livro.

Agradeço a editora Casa do Código, por me oportunizar lançar este livro com eles.

Agradeço também a você, leitor, por se interessar em aprender sobre testes end-to-end automatizados com o framework Protractor.

PREFÁCIO

Por Carmen Popoviciu

O teste é uma parte importante do desenvolvimento de software. Ele é nossa principal ferramenta, como desenvolvedores, para garantir que as aplicações que estamos construindo são livres de erros e funcionam conforme o esperado.

Eu gosto de acreditar que a arte de escrever bons testes se encontra em um bom entendimento do que o teste realmente é, em compreender os seus conceitos básicos e paradigmas e, finalmente, em aplicar essas e outras melhores práticas com sabedoria. Assim como com a escrita de código, testes levam tempo e esforço para serem feitos da maneira certa. É preciso leitura e prática para entender completamente o que testar e onde, para entender os diferentes tipos de testes, ou qual usar e quando, para então chegar a um nível completo de proficiência. Mas uma vez que você aprende, o conhecimento está lá para ficar, e nada é tão bom quanto isso.

O foco principal deste livro é o Protractor, uma ferramenta de testes *end-to-end*, usado principalmente no ecossistema de aplicações AngularJS. A ferramenta foi desenvolvida pelo time principal do framework AngularJS e está aí desde seus primórdios. Apesar disso, por um longo tempo literaturas relacionadas ao Protractor e outros recursos eram escassos, e não tão bem representadas como aquelas relacionados a testes de unidade, por exemplo. Até hoje eu ainda vejo alguns rostos confusos quando se fala de testes E2E e Protractor. Esta foi uma das principais razões pelas quais Andres Dominguez e eu resolvemos escrever o guia de estilos do Protractor, um conjunto de regras e boas práticas, que mais tarde se tornaram o *Styleguide* oficial deste framework. Nosso objetivo era compartilhar nossa experiência com o uso do

Protractor em projetos de grande dimensão, e fornecer um conjunto de diretrizes para ajudar outros desenvolvedores a começar facilmente a escrever testes E2E eficientes para suas aplicações.

Este livro é uma excelente continuação do *Styleguide* do Protractor, não apenas porque reforça as melhores práticas descritas lá, mas porque leva um passo adiante e examina em profundidade o fluxo de trabalho com o Protractor como um todo, e aborda diferentes cenários comuns e casos de uso extremos. Este livro pode ser um excelente ponto de entrada para quem está começando com o Protractor, mas também um recurso valioso para os desenvolvedores que já estão familiarizados com a ferramenta e querem obter mais conhecimento sobre o assunto.

Carmen Popoviciu

@CarmenPopoviciu

Desenvolvedora front-end

Por Stefan Teixeira

Tive o primeiro contato com testes automatizados em 2011. Desde então, isso mudou a minha carreira.

É algo fascinante poder usar skills de desenvolvimento e testes, reduzir o trabalho repetitivo e oferecer feedback mais rápido para a equipe. Hoje, com o crescimento da popularidade da cultura DevOps e com a busca pela entrega contínua, a automação de testes é algo indispensável. É impossível termos um processo de deploy automatizado confiável sem uma boa estratégia de testes automatizados.

Quando falamos de aplicações web, torna-se essencial termos testes automatizados que simulam usuários interagindo com a aplicação. Estes geralmente são chamados de testes de UI, ou testes end-to-end, e são o foco deste livro.

No último GitHub Octoverse, é possível ver o JavaScript prevalecendo como linguagem mais usada no mundo *open source*. Na parte de desenvolvimento web, existem diversos frameworks JavaScript sendo utilizados extensivamente, como o AngularJS.

O Protractor, criado pela própria equipe do AngularJS, se mostra como um dos frameworks mais completos para testes e2e em JavaScript, pela simples curva de aprendizado e várias facilidades que oferece. Além disso, o Protractor também pode ser usado perfeitamente em aplicações que não usam AngularJS, o que favorece ainda mais o seu uso.

Acompanho o trabalho do Walmyr desde 2014, quando conheci seu blog *Talking About Testing* (http://talkingabouttesting.com). Desde Outubro de 2014, Walmyr tem criado conteúdo de qualidade sobre Protractor, seja em forma de posts, palestras ou nos vídeos da

excelente iniciativa *Aprendendo Protractor*.

Este livro consolida, de forma concisa, toda a experiência adquirida nos últimos anos pelo Walmyr sobre Protractor e testes em aplicações web. A leitura é extremamente recomendada tanto para iniciantes quanto para os mais experientes, se tornando um ótimo livro de referência sobre o assunto. Boa leitura!

Stefan Teixeira

@stefan_teixeira

Engenheiro de QA com foco em automação na Toptal

http://stefanteixeira.com.br

SOBRE O AUTOR

Walmyr Lima e Silva Filho trabalha com engenharia de software desde 2004, tendo bacharelado em Administração de Empresas com ênfase em Análise de Sistemas de Informação pela PUC-RS, em 2012. Trabalhou em organizações nacionais e multinacionais em Porto Alegre e Florianópolis.

Atualmente, trabalha como engenheiro de software com foco em qualidade de software no Appear.in, na Noruega. Dentre suas atividades, está a escrita de testes e2e automatizados com o framework Protractor.

Ativo membro de comunidades de tecnologia, sempre gostou de participar de eventos, seja como expectador, coordenador, organizador, voluntário e algumas vezes como palestrante. Ele já palestrou em eventos como: The Developers Conference, Agile Trends, Conferência Agile Testers, e eventos dos Grupos de Usuários de Teste de Software do Rio Grande do Sul e Santa Catarina.

Foi voluntário do Agile Brazil 2014, em Florianópolis, e Coordenador do The Developers Conference Florianópolis em 2015. Além disso, também ajudou em iniciativas internas dentro de empresas em que trabalhou, pois acredita que o conhecimento é algo que deve ser compartilhado.

É autor do blog *Talking About Testing* (http://talkingabouttesting.com) e possui um canal no YouTube onde compartilha conteúdo "mão na massa" em formato de vídeos sobre a utilização do framework Protractor (https://www.youtube.com/user/wlsf82/videos). Além disso, recentemente começou a escrever conteúdos em inglês em sua conta

no Medium (https://medium.com/@walmyrlimaesilv).

Pode ser encontrado no Twitter como *@walmyrlimaesilv*.

SOBRE O LIVRO

Neste livro, você encontrará uma coleção de práticas para implementar testes end-to-end automatizados ao processo de desenvolvimento de aplicações web, utilizando o framework Protractor.

A ideia de escrever este livro veio como uma forma de reunir uma coleção de aprendizados ao longo de minha carreira usando o framework Protractor. Este livro pode servir como fonte de consulta a profissionais que já utilizam a ferramenta, ou profissionais e estudantes que estejam interessados em aprender.

Conhecimentos básicos de JavaScript são recomendados para a leitura do livro, visto que o Protractor é baseado em Node.js.

Durante a leitura, você verá: questões relacionadas a configuração inicial do framework para a criação dos primeiros testes; boas práticas para o desenvolvimento de testes automatizados; o padrão Page Objects; algumas funções ajudantes; node modules úteis; como realizar ações e verificações durante a escrita de testes; testes de revisão visual, testes na nuvem e testes para *mobile*; como executar testes automatizados utilizando práticas de integração contínua; a mais nova versão do JavaScript (ECMAScript 2015); algumas configurações avançadas; o processo criativo para o desenvolvimento de testes end-to-end; e algumas dicas.

Aproveite a leitura!

Sumário

Versão: 20.2.10

Capítulo 1

INTRODUÇÃO

A ideia de escrever este livro veio como uma forma de reunir uma coleção de aprendizados ao longo de minha carreira utilizando o *framework* Protractor. O objetivo era servir como fonte de consulta a profissionais que já utilizam a ferramenta ou profissionais, e estudantes que estejam interessados em aprender. Porém, antes de entrar em detalhes relacionados ao framework, gostaria de abordar algumas questões que julgo importantes para um melhor proveito do que virá a seguir.

Testes automatizados são uma parte muito importante no processo de desenvolvimento de software, sendo estes a base para garantir feedback rápido após mudanças em aplicações. Além disso, eles ajudam no *design* delas e servem como fonte de documentação.

Quando comecei a trabalhar com desenvolvimento de software com foco em testes, no ano de 2004, trabalhava com uma abordagem de testes manuais. Nela, analistas de testes escreviam extensos casos de testes, que eram posteriormente executados manualmente por testadores de software. Eles ajudavam na atualização destes documentos, e também no cadastro de inconformidades em ferramentas de gestão de defeitos (*bug tracking*).

Com o tempo, percebi que executar testes de forma manual, além de ser um processo cansativo, repetitivo e chato, não era um processo confiável. Isso porque, na maioria dos casos, novas

mudanças eram adicionadas nas aplicações antes mesmo do ciclo de testes manuais acabar, ciclos estes que às vezes levavam até uma semana para que todos os testes de regressão de uma aplicação fossem executados.

Dessa forma, no meio da execução dos testes, uma nova versão da aplicação era disponibilizada para ser testada. Caso os testes tivessem de iniciar desde seu início, corria-se o risco de que ele viesse a ocorrer, ou seja, que antes de acabar a execução dos testes, uma nova versão fosse disponibilizada. E se os testes simplesmente continuassem a ser executados na nova versão, de onde pararam, não havia garantia alguma de que as funcionalidades testadas no início dos testes (na versão anterior) estariam ainda passando.

Então resolvi começar a automatizar estes testes manuais, que poderiam ser executados quantas vezes fossem necessárias, de forma rápida e confiável, contra diferentes tipos de navegadores e em diferentes versões de sistemas operacionais.

Além disso, existe um conceito em desenvolvimento de software chamado de pirâmide dos testes, que foi prescrito por Mike Cohn no livro *Succeeding With Agile*. Ele define que, para garantir que aplicações de software tenham qualidade, estas devem possuir uma boa base de testes em nível de unidade (base da pirâmide), alguns testes de integração — para verificar que as partes da aplicação quando integradas funcionam sem problemas (meio da pirâmide) —, e um menor número de testes de UI (*user interface*) — os testes que o framework Protractor se propõe a testar, também conhecidos como testes *end-to-end* (e2e).

Veja a seguir uma ilustração da pirâmide dos testes:

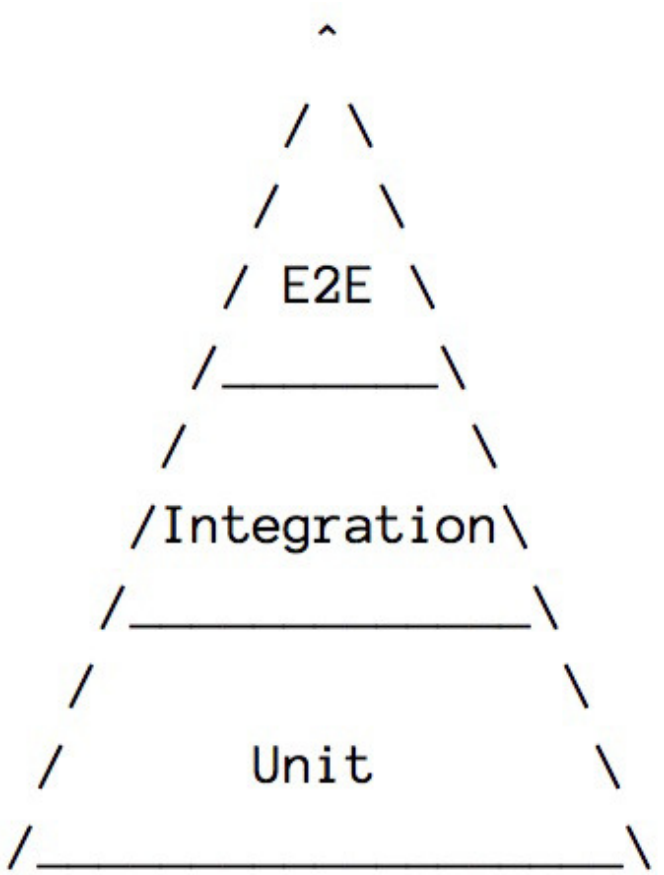

Figura 1.1: Pirâmide dos testes

Portanto, gostaria de ressaltar que, apesar de o livro se tratar de técnicas para os testes do topo do pirâmide (testes e2e), como profissionais da área de engenharia de software, não podemos nos esquecer dos testes que sustentam a pirâmide. São testes que podem ser executados de maneira muito mais rápida quando comparados a testes e2e, além de garantirem que cada parte da aplicação funciona conforme projetada, tanto testadas de forma isolada quanto integradas entre si.

Agora que já temos um boa base e alguns fundamentos, vamos

aos testes e2e e ao Protractor.

1.1 O QUE É PROTRACTOR?

Protractor é um framework *open source* de testes end-to-end automatizados para aplicações AngularJS (criado pelo próprio time que mantém o framework AngularJS). Ele é utilizado para a execução de testes em aplicações, interagindo com elas como um usuário real faria, em navegadores reais, tais como Chrome e Firefox.

Apesar de ter sido criado com algumas funcionalidades específicas para aplicações AngularJS, com uma única linha de código, o Protractor também pode ser utilizado para a criação de testes para aplicações não Angular.

O Protractor é baseado no WebDriverJS. Porém, conforme mencionado, possui algumas funcionalidades específicas para aplicações AngularJS e uma sintaxe um pouco diferente. Ele funciona em conjunto com Selenium para prover essa infraestrutura de testes que simula a utilização de aplicações web.

O servidor Selenium (framework de teste de software portável para aplicações web) interpreta comandos de teste e os envia ao navegador. Já o Protractor cuida dos scripts de testes escritos em JavaScript, usando Node.js

Para exemplificar, funciona mais ou menos assim: primeiro, o Protractor se comunica através da API do WebDriver para enviar comandos ao Selenium, atráves de HTTP. Então, o Selenium se comunica com o navegador usando um protocolo chamado JSON Webdriver Wire Protocol e, por fim, o navegador executa os comandos recebidos como um usuário real faria.

1.2 PRÉ-REQUISITOS NECESSÁRIOS PARA COMEÇAR

Protractor é um programa baseado em Node.js, portanto, para seu uso, você precisa possuir o Node.js instalado, em uma versão superior a v0.10.0.

Para verificar a versão do Node.js instalada em seu computador, basta executar no terminal o seguinte comando:

```
node --version
```

Caso o Node.js não esteja instalado, basta baixá-lo a partir da seguinte URL: https://nodejs.org/en/. E a partir do arquivo baixado, instalá-lo.

1.3 INSTALAÇÃO

Visto que o Protractor é baseado em Node.js, ele utiliza o NPM (*node package manager*) para sua instalação. NPM é um gerenciador de pacotes baseado em Node.js.

Para instalar o Protractor, basta executar o seguinte comando no terminal:

```
npm install -g protractor
```

Este comando instalará o Protractor globalmente em seu computador. Após a instalação, você pode verificar a versão do Protractor instalada executando o seguinte comando:

```
protractor --version
```

Ele deve exibir algo como o seguinte:

```
Version 4.0.2
```

Após a instalação do Protractor, você precisará atualizar o

`webdriver-manager`. Utilize o seguinte comando:

```
webdriver-manager update
```

Você também pode instalar o Protractor e o `webdriver-manager` como dependências de desenvolvimento do seu projeto em vez de instalá-los globalmente. Para isso, basta executar os seguintes comandos:

```
npm install protractor --save-dev
npm install webdriver-manager --save-dev
```

1.4 CONFIGURAÇÕES BÁSICAS

Com o Protractor instalado e o `webdriver-manager` atualizado, o próximo passo é criar um arquivo de configuração. Ele vai definir questões como o endereço onde o servidor do Selenium deve estar rodando, em qual navegador os testes serão executados, quais testes serão executados, qual a URL base para o início de cada teste, dentre diversas outras possíveis configurações. Para saber mais sobre outras configurações, consulte o capítulo *12. Configurações avançadas*.

Criando o arquivo de configuração

Dentro da estrutura de diretórios de seu projeto, caso já exista um diretório onde são armazenados testes, tais como testes de unidade e testes de integração, recomendo a criação de um diretório chamado `e2e`. Caso ainda não exista um diretório para testes, crie o diretório `tests`, e então dentro dele o diretório `e2e`. Para mais detalhes, veja a seção *Estrutura de projeto* do capítulo *2. Boas práticas*.

Dentro do diretório `e2e`, crie um arquivo chamado `protractor.conf.js`. Abra o arquivo `protractor.conf.js` em seu editor, e então adicione as seguintes configurações básicas:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

OBSERVAÇÃO

O valor do atributo `baseUrl` deve ser substituído posteriormente pela URL base da aplicação do projeto que será testado utilizando o *framework* Protractor. Por hora, entenda esta URL somente como um exemplo.

Com isto, você já possui o mínimo necessário para começar a escrever seus testes.

1.5 ESCREVENDO O PRIMEIRO TESTE

Com as configurações básicas para execução de testes com o framework Protractor, o próximo passo é a criação dos testes propriamente ditos.

A partir do diretório `e2e` (anteriormente criado, que contém o arquivo de configurações de Protractor), crie um subdiretório chamado `specs` e, dentro dele, crie um arquivo chamado `homepage.spec.js`.

Abra o arquivo `homepage.spec.js` em seu editor e adicione o seguinte código (que será explicado em detalhes logo adiante):

```
// homepage.spec.js
```

```
describe('Homepage', function() {
  it('perform a search into the api page', function() {
    browser.get('#/api');

    element(by.model('searchTerm')).sendKeys('restart');
    element(by.css('.depth-1')).click();

    expect(element(by.css('.api-title')).getText()).toContain('bro
wser.restart');
  });
});
```

Antes de executar o teste recém-criado, vamos entender a estrutura de um teste escrito com o Protractor. Inicialmente, é definido um `describe`, que é a definição da suíte de teste. Este recebe uma descrição (neste caso, `'Homepage'`) e executa uma função (ou um callback).

Então, é definido um `it`, que é o caso de teste propriamente dito. Este também recebe uma descrição, que é o que exatamente o teste se propõe a testar (neste caso *'perform a search into the api page'*) e também executa uma função (ou callback).

Em seguida, os passos do teste são definidos e são explicados a seguir, linha a linha. O primeiro passo do teste é o seguinte:

```
browser.get('#/api');
```

Esse passo basicamente acessa através do navegador a URL relativa `#/api`. Porém, como no arquivo de configuração do Protractor (`protractor.conf.js`) foi definida uma URL base (`baseUrl`), essa URL relativa é então concatenada à URL base, gerando a seguinte URL absoluta: **http://www.protractortest.org/#/api**.

No segundo passo, é digitado o valor `'restart'` em um campo definido por seu modelo `(by.model('searchTerm'))`.

```
element(by.model('searchTerm')).sendKeys('restart');
```

No terceiro passo, um link com a classe `css 'depth-1'` é clicado.

```
element(by.css('.depth-1')).click();
```

Por fim, é feita a verificação:

```
expect(element(by.css('.api-title')).getText()).toContain('browser
.restart');
```

Neste caso, é verificado que um elemento com a classe `css 'api-title'` contém o texto `'browser.restart'`.

1.6 EXECUTANDO O PRIMEIRO TESTE

Agora que o Protractor já possui as configurações básicas para execução de testes e um primeiro teste já foi escrito, é a hora de executá-lo para verificar o comportamento da aplicação. No terminal, digite o seguinte comando para iniciar o servidor do Selenium:

```
webdriver-manager start
```

Em uma segunda aba do terminal, a partir do diretório no qual se encontra o arquivo de configuração do Protractor (`protractor.conf.js`), execute o seguinte comando:

```
protractor
```

Com o webdriver-manager iniciado e quando o comando para o Protractor executar os testes, o navegador definido no arquivo de configuração do Protractor deve ser aberto (neste caso, o navegador Chrome). Então, os passos descritos no testes devem ser executados. Após a execução do teste, o navegador deve ser fechado automaticamente.

Em caso de sucesso na execução do teste, você deve ver um resultado como o seguinte:

```
$ protractor
[17:26:44] I/hosted - Using the selenium server at http://localhos
t:4444/wd/hub
[17:26:44] I/launcher - Running 1 instances of WebDriver
Started
.


1 spec, 0 failures
Finished in 1.724 seconds
[17:26:48] I/launcher - 0 instance(s) of WebDriver still running
[17:26:48] I/launcher - chrome #01 passed
```

Perceba que é exibido o endereço do servidor do Selenium, definido no arquivo de configuração do Protractor (http://localhost:4444/wd/hub). E após aproximadamente 2 segundos, o teste é finalizado sem nenhuma falha. Além disso, é exibida a informação de que o teste foi executado no navegador Chrome.

1.7 A IMPORTÂNCIA DO RESULTADO DOS TESTES

Na escrita de scripts de testes automatizados, é importante garantir não só que os testes passem, mas que também falharão quando a aplicação não se comportar conforme o esperado. Uma técnica que costumo utilizar para verificar se meus testes estão realmente testando alguma coisa é modificar a *assertion* (expect) para simular uma falha, e verificar que o teste falhará caso o resultado esperado seja diferente do que foi definido. Uma técnica que costumo utilizar para verificar se meus testes estão realmente testando alguma coisa é modificar a *assertion* (expect) para simular uma falha, e verificar que o teste falhará caso o resultado esperado seja diferente do que foi definido.

Simulando uma falha

Modifique o último passo do teste para o seguinte:

```
expect(element(by.css('.api-title')).getText()).not.toContain('bro
wser.restart');
```

Perceba que agora a verificação é que o elemento com classe `css 'api-title'` **não** contenha o texto `'browser.restart'`.

Execute novamente o teste e verifique o resultado. Após a execução, o resultado deve ser algo como o seguinte:

```
$ protractor
[17:57:42] I/hosted - Using the selenium server at http://localhos
t:4444/wd/hub
[17:57:42] I/launcher - Running 1 instances of WebDriver
Started
F

Failures:
1) Homepage perform a search into the api page
  Message:
    Expected 'browser.restart View code' not to contain 'browser.r
estart'.
  Stack:
    Error: Failed expectation
        at Object.<anonymous> (/Users/Walmyr/www/Protractor-eBook/
e2e/specs/homepage.spec.js:8:57)
        at /usr/local/lib/node_modules/protractor/node_modules/jas
minewd2/index.js:96:23
        at new Promise (/usr/local/lib/node_modules/protractor/nod
e_modules/selenium-webdriver/lib/promise.js:1043:7)
        at controlFlowExecute (/usr/local/lib/node_modules/protrac
tor/node_modules/jasminewd2/index.js:82:18)
        at TaskQueue.execute_ (/usr/local/lib/node_modules/protrac
tor/node_modules/selenium-webdriver/lib/promise.js:2790:14)
        at TaskQueue.executeNext_ (/usr/local/lib/node_modules/pro
tractor/node_modules/selenium-webdriver/lib/promise.js:2773:21)
        at asyncRun (/usr/local/lib/node_modules/protractor/node_m
odules/selenium-webdriver/lib/promise.js:2697:25)
        at /usr/local/lib/node_modules/protractor/node_modules/sel
enium-webdriver/lib/promise.js:639:7
        at process._tickCallback (internal/process/next_tick.js:10
3:7)

1 spec, 1 failure
Finished in 2.049 seconds
[17:57:45] I/launcher - 0 instance(s) of WebDriver still running
```

```
[17:57:45] I/launcher - chrome #01 failed 1 test(s)
[17:57:45] I/launcher - overall: 1 failed spec(s)
[17:57:45] E/launcher - Process exited with error code 1
```

Veja que ocorreu um erro em relação ao resultado esperado, onde se esperava que o valor `'browser.restart'` não contivesse esse valor.

É de extrema importância forçar falhas nos testes escritos para garantir que eles vão realmente falhar quando houver uma falha real na aplicação. Sem essa garantia, pode ocorrer o que é chamado de falsos positivos, ou seja, testes que passam quando na verdade deveriam estar falhando.

Corrija o teste para que ele volte a passar e execute-o novamente para garantir que o comportamento está ok.

> **DICA**
>
> Para que o teste volte a passar, remova o `not` do seguinte código: `expect(element(by.css('.api-title')).getText()).not.toContain('browser.restart');` .

1.8 O PADRÃO AAA (ARRANGE, ACT, ASSERT)

Para finalizar o capítulo introdutório, resolvi trazer um padrão amplamente utilizado, não só na escrita de testes e2e, como também na escrita de testes de unidade ou testes de integração. Este padrão é chamado AAA (*arrange, act e assert*).

Arrange é a pré-condição do teste. É o que deve ocorrer para o teste estar na condição necessária para que seja realizado. No caso do exemplo deste capítulo, o *arrange* é a navegação até a URL

relativa `'#/api'`.

Em um exemplo de um sistema de gerenciamento de conteúdo, em que para criar um conteúdo o usuário precisaria estar logado, por exemplo, o *arrange* seria fazer o login na aplicação.

Act são os passos executados na aplicação (são os *inputs*). No caso do exemplo deste capítulo, o *act* são os passos onde é digitado o valor `'restart'` no campo definido pelo modelo `'searchTerm'`, e quando o campo definido pela classe `css 'depth-1'` é clicado.

No exemplo do sistema de gerenciamento de conteúdo, no teste de criação de conteúdo, o *act* seria os passos para a criação do conteúdo, tais como a navegação até a tela de criação do conteúdo desejado, o preenchimento de um formulário e sua submissão.

Assert é a verificação (o *output*), o passo que torna esta automação realmente um teste. No exemplo deste capítulo, o *assert* é a verificação de que o elemento com classe `css 'api-title'` contém o texto `'browser.restart'`.

Novamente ao exemplo de um sistema de gerenciamento de conteúdo, o *assert* em um caso de testes de criação de um conteúdo poderia ser a verificação de uma mensagem de sucesso após a criação. Ou ainda melhor, a verificação de que tal conteúdo foi realmente criado, tal como a sua verificação em uma listagem.

Agora que já vimos alguns fundamentos sobre teste de software e você já sabe o que é o Protractor e o básico para utilizá-lo, veremos algumas boas práticas quando se trata de automação de testes e2e.

Capítulo 2

BOAS PRÁTICAS

Resolvi abordar boas práticas em desenvolvimento de testes e2e logo no segundo capítulo do livro. Isto porque elas serão a base para muitos dos outros itens abordados ao longo dos próximos capítulos, e também a fundação para o desenvolvimento de testes automatizados robustos e de fácil manutenção. As boas práticas e padrões de projeto são tão importantes quando se fala de testes automatizados como quando se fala de código de produção.

Neste capítulo, veremos:

- Algumas regras gerais, tais como duplicidade de testes e boas práticas referentes a configurações;
- Questões relacionadas a estrutura de projeto, importantes para sua manutenção;
- Estratégias de localizadores, essenciais para a escrita de testes robustos;
- O padrão Page Objects para ajudar na manutenabilidade e legibilidade dos testes;
- E suítes de testes, tais como suítes de *smoke* teste e testes de regressão.

2.1 REGRAS GERAIS

Para manter boas práticas na escrita de scripts de testes e2e automatizados, é necessário evitar a duplicidade de testes e de configurações. Para isso, existem algumas regras gerais que é

importante ressaltar.

Não crie testes e2e para funcionalidades já cobertas por testes de unidade ou testes de integração

Testes e2e que simulam a real utilização da aplicação como se fosse um usuário possuem um custo maior com relação ao tempo de execução quando comparados a testes de unidade ou teste de integração. Isso porque testes e2e de UI (*user interface*) precisam abrir o navegador, visitar páginas (que podem demorar para carregar), preencher formulários, clicar em botões etc. Portanto, se já existem testes para tal funcionalidade em uma destas outras camadas, eles serão executados de forma muito mais rápida.

Além disso, quando se trata de desenvolvimento de software utilizando práticas de código limpo, por exemplo, sugere-se a não duplicação de código. Testes e2e para funcionalidades já cobertas por testes em outras camadas (unidade ou integração) é duplicação de testes, ou seja, algo desnecessário e que somente adiciona complexidade, sem agregar valor.

Utilize somente um arquivo de configuração

Conforme recém-falado, duplicação de código é algo que devemos evitar, e o mesmo vale para configurações. Quando existe a necessidade de mudar uma configuração devido a alguma modificação na aplicação ou sua arquitetura, por exemplo, esta mudança teria de ser feita em mais de um arquivo de configuração, algo que é passível de ser esquecido.

Além disso, ferramentas de build e automatizadores de tarefa, tais como Gulp e Grunt, podem lhe ajudar na tarefa de sobrescrever tais configurações em tempo de execução. Assim, podem, por exemplo, possibilitar a execução de testes em diferentes ambientes, tais como ambiente local de desenvolvimento, ambiente de QA

(*quality assurance*), ambiente de UAT (*user acceptance testing*), ambiente de testes de performance, ou mesmo no ambiente produção.

> Gulp e Grunt são ferramentas para execução automatizada de tarefas, como mimificação e/ou concatenação de arquivos, compilação de front-end, execução de testes etc. Além disso, com ambas as ferramentas, a automação das tarefas é realizada escrevendo código JavaScript.

Portanto, evite algo como:

- `protractor.conf.local.js`
- `protractor.conf.qa.js`
- `protractor.conf.uat.js`
- `protractor.conf.perf.js`
- `protractor.conf.prod.js`

que seriam arquivos de configuração por ambiente. Em vez disso, tenha somente o arquivo `protractor.conf.js` . Porém, com diferentes tasks para sobrescrever a `baseUrl` , por exemplo, para execução dos mesmos testes nos mais diversos ambientes. Algo como:

- `gulp e2e-test-local`
- `gulp e2e-test-qa`
- `gulp e2e-test-uat`

E assim por diante. O mesmo pode ser feito para sobrescrever, por exemplo, o navegador no qual os testes vão executar, como:

- `gulp e2e-test-chrome`
- `gulp e2e-test-firefox`

- `gulp e2e-test-ie`

2.2 ESTRUTURA DE PROJETO

Projetos bem estruturados trazem diversas vantagens no desenvolvimento de softwares. Vantagens estas que serão apresentadas a seguir, além de algumas práticas que utilizei com sucesso em projetos reais.

Agrupe os testes e2e de forma que faça sentido ao projeto

Projetos de software, incluindo subprojetos de testes automatizados, devem ser bem estruturados por questões de organização, facilidade de localização de cada uma de suas partes e separação com relação a outros tipos de testes, como testes de unidade, de integração, de performance etc. Além disso, também possibilitam uma estrutura limpa, na qual as coisas façam sentido.

Em casos nos quais não existe a preocupação da organização dos testes em uma estrutura de diretórios específica por tipo de teste, por exemplo, torna-se difícil realizar a manutenção de tais testes. Isto porque todos estão misturados uns com os outros, ou mesmo com códigos que nem mesmo relacionados a testes são.

Um exemplo de estrutura de projeto de testes e2e que gosto de usar é a seguinte:

```
|-- project-root-directory
    |-- client
    |-- server
    |-- tests
        |-- e2e
            |-- page-objects
                  contact.po.js
                  homepage.po.js
                  signIn.po.js
                  signUp.po.js
```

```
        |-- specs
              contact.spec.js
              hompepage.spec.js
              signIn.spec.js
              signUp.spec.js
        helper.js
        protractor.conf.js
        README.md
    |-- integration
    |-- unit
.gitignore
gulpfile.js
package.json
README.md
```

Perceba que, nesse exemplo, dentro do diretório e2e, há um arquivo `README.md` , que utilizo para documentar brevemente questões importantes relacionadas ao propósito de tais testes e explicações sobre a estrutura deste subprojeto, pré-requisitos e setup inicial necessário para começar. Além disso, também existem algumas dicas úteis e qualquer outra informação que possa fazer sentido.

Exemplo de README.md

```
# Sample project end-to-end (e2e) tests.

The Sample project e2e tests are written using Protractor, the off
icial test framework for e2e testing of AngularJS applications.

This test project tries to follow the best practices described in
the Protractor's official website.

The architecture of this project is described below:

* The `protractor.conf.js` file stores all the configuration neede
d for the tests to be executed, like the selenium address, the bro
wser that the tests will be executed in, the base url from where t
he tests will start from, etc.

* There is also a `helper.js` file for general functions that can
be used in the tests, like sleeping for some seconds, refreshing t
he page, waiting for element visibility, etc.

* The `page-objects` directory contains the web elements and funct
```

```
ions for specific pages or parts of the pages. This is done this w
ay for better maintenance and for separation of responsibilities.

* And the **specs** directory contains the tests, where each test
suite is a separate spec file. Each spec file has a describe secti
on, which basically describes the functionality being tested (and
it names the test suite), and there is an it section for each of t
he different test cases.

## Local installation:

For installing protractor locally, follow the below steps:
```

npm install protractor -g

webdriver-manager update

```
## Running protractor:

For running protractor just execute the below grunt tasks.
```

grunt server:e2e // run this in one console's tab grunt test:protractor // run this in another console's tab

```
## Tips:

Protractor uses Jasmine syntax, so:

* if you need to run only a specific test case, change the 'it' fo
r 'fit'
* if you want to skip a specific test case, change the 'if' for 'x
it'
* if you want to run only a specific test suite, change the 'descr
ibe' for 'fdescribe'

Also:

* for running protractor against your local environment, make a co
py of the config.local.example.js file, but change the name to con
fig.local.js (this will be ignored by git). With this file, some n
ot important configurations (for running on local) are overwritten
. You can also edit this config local file for overwritten other c
onfigurations you might need (e.g: baseUrl).
```

___

```
For more information, take a look at the Protractor's official doc
umentation [http://www.protractortest.org/#/](http://www.protracto
rtest.org/#/).
```

2.3 ESTRATÉGIAS DE LOCALIZADORES

A escolha de bons localizadores é um dos segredos para a escrita de testes e2e robustos e de fácil manutenção. É por meio de localizadores que elementos HTML são identificados para posterior interação ou verificação quando se trata de testes automatizados.

Portanto, usar bons localizadores para identificar tais elementos é essencial, visto que quando se trata de testes automatizados que interagem com a aplicação com um usuário real faria, é necessário garantir que se está interagindo ou verificando o elemento correto. Além disso, a escolha de bons localizadores ajuda na legibilidade dos testes.

Nesta seção, apresento algumas estratégias recomendadas para a escolha dos localizadores de elementos HTML, além de práticas que não devem ser seguidas.

Evite utilizar xpath

Segundo o próprio guia de estilos do site oficial do Protractor, não é recomendado utilizar xpath para a localização de elementos HTML em testes e2e. Alguns motivos são:

- Markup sujeito a alterações
- Problemas de desempenho
- Legibilidade ruim

Um exemplo de elemento localizado pelo `xpath` (retirado do site oficial do Protractor) é demonstrado a seguir:

```
var elem = element(by.xpath('/*/p[2]/b[2]/following-sibling::node(
```

```
)' +
   '[count(.|/*/p[2]/b[2]/following-sibling::br[1]/preceding-sibli
ng::node())' +
   '=' +
   ' count((/*/p[2]/b[2]/following-sibling::br[1]/preceding-siblin
g::node()))' +
   ']'));
```

Perceba a dificuldade para entender qual elemento HTML tal localizador se propõe a identificar. Portanto, **não utilize xpath** para localizar elementos.

Dê preferência a localizadores específicos do Protractor quando possível

Visto que o Protractor é o framework oficial para testes e2e de aplicações AngularJS, ele possui uma série de localizadores específicos para este tipo de aplicação, como:

- `by.binding` — Este localiza elementos pela diretiva `ngBind`, diretiva esta que substitui o texto de um elemento HTML específico pelo valor de uma expressão, e atualiza o texto quando o valor de tal expressão muda.
- `by.model` — Este localiza elementos por seu modelo (diretiva `ngModel`), visto que aplicações AngularJS usam a arquitetura MVC (*model, view, controller*).
- `by.repeater` — Este localiza elementos que possuem a diretiva `ngRepeat` quando definidos, diretiva esta que ajuda na escrita de código reutilizável quando se trata de aplicações AngularJS, em que se pode definir somente um elemento em um template HTML. Mas dependendo de quantos itens existirem em uma coleção, estes são multiplicados no HTML.

Tendo em vista a existência destes localizadores específicos para aplicações desenvolvidas em AngularJS, recomenda-se que estes

sejam sua primeira opção para escolha de localizadores. Alguns motivos são:

- Facilidade no acesso aos elementos
- O código é menos propenso a mudanças com relação ao markup
- São localizadores mais legíveis

Veja a seguir alguns exemplos de elementos localizados através de localizadores específicos do Protractor:

```
var personField = element(by.binding('person.name'));
var searchField = element(by.model('ctrl.findBy'));
var firstDocumentLink = element(by.repeater('item in trail')).row(
0);
```

Ao entender as diretivas de aplicações AngularJS, a definição de localizadores para elementos HTML fica mais clara e intuitiva, além de ajudar na legibilidade do código.

Dê preferência a by.id e by.css quando não houver localizador específico do Protractor disponível

Nem sempre será possível localizar elementos na aplicação através de localizadores específicos do Protractor. Por vezes, certas partes do HTML não possuirão nenhuma referência às diretivas específicas do AngularJS, como `ngModel`, `ngBind` ou `ngRepeat`.

Nestes casos, recomenda-se, primeiro, identificá-los por `id`, e caso não haja um id, utilizar o localizador por `css` (*css selector*), de preferência por classe.

Algumas vantagens da utilização de tais localizadores são:

- Facilidade no acesso aos elementos
- Bom desempenho na localização de elementos no DOM (*document object model*)

- Utilização de markup menos sujeito a alterações
- Legibilidade

Veja a seguir alguns exemplos de elementos localizados através de `id` e classe `css`:

```
var footer = element(by.id('footer'));
var backToTop = footer.element(by.css('.back-to-top')):
```

Veja como o código demonstrado é legível, além de utilizar localizadores que tendem a não mudar com frequência.

Evite localizadores por texto

Em alguns projetos que trabalhei, cometi o erro de usar localizadores de elementos por texto, e de repente, os testes começaram a falhar, causando falsos negativos. Vale lembrar de que resultados falsos negativos são prejudiciais a projetos de software, visto que o time deve poder confiar nos resultados dos testes. Quando estes falham, devem estar indicando bugs reais na aplicação em teste, e não resultados negativos por testes mal escritos.

Nos casos dos projetos em que presenciei tais problemas, eu havia localizado alguns elementos através de seus textos. Quando a aplicação passou por internacionalização, em que vários textos foram traduzidos, os testes começaram a falhar e precisaram ser corrigidos.

Alguns motivos para evitar localizadores por texto são:

- Textos de botões, links e rótulos tendem a mudar ao longo do tempo
- Testes e2e não podem falhar por simples alterações de textos
- Internacionalização (traduções)

Por fim, caso não haja um bom localizador (específico do

Protractor, por `id` ou por classe `css`, por exemplo) para o elemento com o qual o teste precisa interagir ou mesmo fazer uma verificação, é mais sensato modificar a aplicação para a adição de um `id` ou uma classe `css` ao elemento do que utilizar um localizador frágil. Isto se chama testabilidade.

2.4 PAGE OBJECTS

Page Objetcs são uma padrão de projeto na escrita de testes automatizados, principalmente quando se fala de testes de UI (*user interface*), ou teste e2e. Tal padrão serve para facilitar a manutenção dos testes e ajudar na escrita de testes legíveis. Com o uso de *Page Objects*, basicamente separa-se a responsabilidade entre a definição de elementos HTML e funções que podem ser realizadas em uma página em teste, da implementação do teste em si.

Conforme descrito no site oficial do Protractor, é comum utilizar Page Objects na escrita de testes e2e, visto que tal padrão ajuda na escrita de testes limpos. Uma vez que os elementos da aplicação são encapsulados nestes objetos, quando por algum motivo determinado *template* da aplicação muda, somente o Page Object que define tal elemento precisa ser alterado em vez dos testes que o utilizam, que continuam intactos. Além disso, Page Objects podem ser reutilizados ao longo de diferentes testes, sempre que necessário.

Utilize Page Objects para interagir com a página em teste

Somente reforçando, as vantagens da utilização do padrão Page Objects são:

- Reutilização de código
- Testes limpos e legíveis

- Quando um elemento muda, somente o *Page Object* precisa ser alterado, não todos os testes que utilizam tal elemento

Veja um exemplo de um teste escrito usando o padrão Page Objects:

```
// contactPage.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');

describe('contact page', function() {
  var contactPage = new ContactPage();

  it('successfull contact form submission', function() {
      contactPage.visit();

      contactPage.fillForm('walmyr', 'walmyr@email.com', 'My conta
ct message.');
      contactPage.submitButton.click();

      expect(contactPage.successMessage.isDisplayed()).toBe(true);
  });
});
```

Esse teste define um *Page Object* chamado `ContactPage`, logo no início do arquivo. Logo após a definição do `describe`, que é a definição da suíte de teste por assim dizer, uma instância desse *Page Object* é definida (`contactPage`).

Então as funções e os elementos de tal *Page Object* ficam expostos ao longo dos casos de teste (`it`), para realizar os passos do teste propriamente dito. Este visita a página de contato, preenche um formulário com os valores passados como argumento à função (`fillForm`), clica no botão para submissão do formulário e, por fim, verifica que uma mensagem de sucesso é exibida.

Perceba como é fácil entender o que o teste se propõe a fazer/testar.

O arquivo `contactPage.po.js` é demonstrado mais adiante.

Declare um Page Object por arquivo

Visto que um Page Object se propõe a encapsular os elementos de uma tela específica da aplicação (ou de uma parte da tela, em algumas situações), recomenda-se declarar somente um Page Object por arquivo. Assim, o código fica limpo e existe uma maior facilidade em encontrar o que se procura.

Veja um exemplo, retirado da seção sobre estrutura de projeto:

```
|-- page-objects
      contact.po.js
      homepage.po.js
      signIn.po.js
      signUp.po.js
```

Neste caso, existem quatro Page Objects, cada um encapsulando elementos de partes distintas da aplicação.

Utilize somente um module.exports ao final do arquivo Page Objects

Visto a recomendação anterior, tendo somente um Page Object por arquivo, só há uma classe a ser exportada. Veja a seguir o arquivo `contactPage.po.js` que encapsula os elementos do teste demonstrado no tópico *Utilize Page Objects para interagir com a página em teste*:

```
// contactPage.po.js

var ContactPage = function() {
  this.emailField = element(by.id('email-field'));
  this.messageTextAreaField = element(by.id('message-text-area-fie
ld'));
  this.nameField = element(by.id('name-field'));
```

```
  this.submitButton = element(by.id('submit'));
  this.successMessage = element(by.css('.success-message'));
};

ContactPage.prototype.fillForm = function(name, email, message) {
  this.nameField.sendKeys(name);
  this.emailField.sendKeys(email);
  this.messageTextAreaField.sendKeys(message):
};

ContactPage.prototype.visit = function() {
  browser.get('contact');
};

module.exports = ContactPage;
```

Perceba que o Page Object `ContactPage` possui cinco elementos expostos de forma pública, que podem ser usados diretamente no teste, apesar de só dois deles estarem sendo utilizados diretamente neste caso. Adiante há uma seção que explica melhor esta abordagem. Os elementos em questão são:

- `emailField`
- `messageTextAreaField`
- `nameField`
- `submitButton`
- `successMessage`

> Costumo organizar os elementos de forma alfabética quando fazem parte do mesmo contexto, o que também pode ser considerado uma boa prática.

Além disso, o Page Object `ContactPage` possui duas funções (em Orientação a Objetos, dois métodos), que são explicados isoladamente a seguir:

- **Método ou função `fillForm` :** recebe três argumentos

(`name` , `email` e `message`) e preenche 3 elementos com estes valores, nesta ordem. Os elementos preenchidos são exatamente os que não são utilizados diretamente nos testes em questão, visto que, neste caso, são usados por este método ou função. Adiante há uma seção que explica melhor esta abordagem.

- **Método ou função `visit` :** basicamente acessa uma página pela URL relativa `'contact'` .

Requeira e instancie todos os node modules no topo

Somente para fins de esclarecimento, um Page Object neste caso é considerado um *node module*, ou seja, também são considerados módulos escritos em Node.js.

No exemplo do teste demonstrado na seção *Utilize Page Objects para interagir com a página em teste*, é possível perceber que, na primeira linha do arquivo, o Page Object `ContactPage` é requerido (`var ContactPage = require('../page-objects/contactPage.po.js')`), e então um objeto do tipo `ContactPage` é instanciado (`var contactPage = new ContactPage();`), logo no início do `describe` .

Um exemplo de teste que requer mais de um Page Object é demonstrado a seguir:

```
// contactPageWithPageObjectWrapper.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');
var MessagesWrapper = require('..page-objects/messagesWrapper.po.j
s');

describe('contact page', function(){
  var contactPage = new ContactPage();
  var messagesWrapper = new MessagesWrapper();

  it('successfull contact form submission', function() {
      contactPage.visit();
```

```
      contactPage.fillForm('walmyr', 'walmyr@email.com', 'My conta
ct message.');
      contactPage.submitButton.click();

      expect(messagesWrapper.successMessage.isDisplayed()).toBe(tr
ue);
  });
});
```

Veja que o teste é, na verdade, o mesmo já demonstrado, porém, este foi refatorado para usar um Page Object chamado `MessageWrapper`. Isso porque outras telas da mesma aplicação possuem também estas mensagens, portanto, é algo como um Page Object genérico.

Neste caso, agora a verificação (`expect`) é feita sobre a instância `messageWrapper` do Page Object `MessageWrapper` em vez da instância `contactPage` do Page Object `ContactPage`. Dessa forma, outras telas que precisam verificar a exibição de uma mensagem de sucesso podem utilizar exatamente a mesma linha de código, ou seja:

```
expect(messagesWrapper.successMessage.isDisplayed()).toBe(true);
```

Declare todos os elementos do construtor como públicos

Conforme comentado no tópico *Utilize somente um module.exports ao final do arquivo Page Objects*, no Page Object exemplo, comentei que, dos cinco elementos expostos de forma pública, somente dois estavam sendo diretamente usados pelo teste, porém, mesmo assim todos foram expostos de forma pública. Isto se explica pois o usuário de um Page Object (desenvolvedor ou testador que vai escrever os testes) deve possuir acesso rápido aos elementos disponíveis na página.

No teste demonstrado, nem todos os elementos estavam sendo

usados. Mas em outros testes, estes podem ser necessários de forma isolada em vez de através do método que os utilizou (`fillForm`). Veja um exemplo:

```
// contactPageWithPageObjectWrapper.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');
var MessagesWrapper = require('..page-objects/messagesWrapper.po.j
s');

describe('contact page', function(){
  var contactPage = new ContactPage();
  var messagesWrapper = new MessagesWrapper();

  it('try to submit contact form filling only the email field', fu
nction() {
      contactPage.visit();

      contactPage.emailField.sendKeys('email@example.com');
      contactPage.submitButton.click();

      expect(messagesWrapper.errorMessage.isDisplayed()).toBe(true
);
  });
});
```

Neste caso, visto que o elemento `emailField` também está exposto de forma pública, este pode ser usado diretamente no teste. Como informação adicional e informativa (não relacionada a este contexto), perceba que agora a instância `messageWrapper` (do Page Object `MessageWrapper`) possui também um elemento exposto de forma pública, chamado `errorMessage` , que está sendo usado na verificação (`expect`).

Declare funções para operações que necessitam de mais de um passo

Conforme comentado no tópico *Utilize somente um module.exports ao final do arquivo Page Objects*, no exemplo do Page Object demonstrado, há um método chamado `fillForm` . Veja a seguir somente o método em questão:

```
ContactPage.prototype.fillForm = function(name, email, message) {
  this.nameField.sendKeys(name);
  this.emailField.sendKeys(email);
  this.messageTextAreaField.sendKeys(message):
};
```

Este método se preocupa exatamente em atender a boa prática: **declare funções para operações que necessitam de mais de um passo**. Visto que três passos são necessários para preencher o formulário de contato, estes foram encapsulados em uma função que recebe três argumentos (`name` , `email` e `message`), e então preenche os três campos em questão com os valores destes argumentos, nesta ordem.

Esta abordagem é muito importante por questões de reaproveitamento de código e eliminação de duplicidade. Veja a seguir um novo teste que utiliza tal método:

```
it('try to submit contact form with invalid email format', functio
n() {
    contactPage.visit();

    contactPage.fillForm('walmyr', 'invalidformat', 'My contact me
ssage');
    contactPage.submitButton.click();

    expect(messagesWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Perceba que o mesmo método foi usado, porém alterando o segundo argumento, assim podendo verificar o comportamento da aplicação em outro cenário de teste.

Não faça assertions nos Page Objects

Considero esta uma regra de grande importância quando se fala de testes automatizados. Seguem alguns motivos que a justificam:

- A responsabilidade pelos *assertions* é dos testes;
- As pessoas que vão ler os scripts de teste devem ser

capazes de entender o comportamento esperado da aplicação lendo somente o teste.

Ou seja, o `expect` deve sempre estar no arquivo com extensão `.spec.js`. Isto está relacionado com a questão de separação de responsabilidades.

A responsabilidade do teste é, a partir de um contexto, realizar ações, e então verificar o resultado. Já a responsabilidade do Page Object é encapsular elementos, funções ou métodos para operações que necessitam de mais de um passo, conforme explicado na seção anterior.

Adicione wrappers para diretivas, diálogos e elementos comuns

Conforme demonstrado na seção *Requeira e instancie todos os node modules no topo*, o teste utilizou um Page Object chamado `MessageWrapper`, criado a partir de uma refatoração, visto que as mensagem são exibidas em telas diversas. Este é o motivo desta boa prática.

Caso existam elementos que se repetem ao longo da aplicação em diferentes telas, em vez de sair duplicando tais elementos em diferentes Page Objects, deve-se criar estes *wrappers* que são Page Objects mais genéricos. Assim, eles podem ser utilizados em diferentes contextos.

Alguns motivos para utilizar Page Objects do tipo *wrappers*:

- Podem ser utilizados em múltiplos testes;
- Evita duplicidade de código;
- Quando uma diretiva, diálogo ou elemento comum muda, você só precisa mudar o *wrapper*, e somente uma vez.

Mais sobre Page Objects será abordado no próximo capítulo *Page Objects*.

2.5 SUÍTES DE TESTES

Para fechar o capítulo de boas práticas, abordarei alguns pontos importantes quando se trata de suítes de testes e2e automatizados.

Não utilize mock a não ser que seja totalmente necessário

Não é recomendado a utilização de *mocks* em testes e2e, exatamente pois testes e2e se propõem a testar a aplicação como se fosse um usuário real. E quando um usuário real está usando uma aplicação, este interage com ela por completo, com todas suas integrações.

Alguns motivos do porque não utilizar *mocks* em testes e2e automatizados são:

- Utilizar a aplicação real com todas suas dependências provê alta confiança;
- Caso uma parte da aplicação que está sendo testada usando *mock* for alterada e esquecemos de alterar o *mock*, o teste que o utiliza pode continuar passando, mesmo que a alteração tenha quebrado a aplicação. Isso se chama falso positivo. Ou seja, o teste estará passando, quando deveria estar falhando, visto que determinada parte da aplicação parou de se comportar conforme o esperado, porém um *mock* desatualizado causa a impressão de que a aplicação continua

funcionando.

Utilize Jasmine2

Jasmine é um framework de testes de unidade para JavaScript, no qual os scritps de teste são escritos utilizando a técnica de desenvolvimento guiado por testes. Sua sintaxe é simples e de fácil compreensão.

O framework Protractor por padrão usa a sintaxe do framework de testes Jasmine, porém, em suas versões iniciais, ele utilizava o Jasmine1. Entretanto, com sua evolução, passou a suportar o Jasmine2, que pode ser definido no arquivo de configurações, na configuração de framework, assunto abordado no capítulo *12. Configurações avançadas*.

Algumas vantagens da utilização do Jasmine2 são:

- É bem documentado;
- É também suportado pelo time do Protractor;
- Possui as funções `beforeAll` e `afterAll` , não existentes na versão anterior.

Veja um exemplo do uso da função `beforeAll` :

```
beforeAll(function() {
  homepage.visit();
  homepage.accessNewForm();
  newAccountForm.createAccount(accountData);
});
```

Veja um exemplo da utilização da função `afterAll` :

```
afterAll(function() {
  dashboard.delete(accountData);
});
```

Veja que tais funções podem ser utilizadas para pré e pós-condições da suíte de testes.

Faça com que os testes sejam independentes ao menos no nível de arquivo

O ideal é que cada caso de teste seja totalmente independente, mas quando isso não for possível, por questões de um grande custo com relação ao tempo e execução, devido a pré e pós-condições muito custosas, faça com que ao menos sejam independentes em nível de arquivo (arquivo com extensão `.spec.js`).

Algumas vantagens desta abordagem são:

- Execução de testes em paralelo com a utilização de *sharding* (este assunto é abordado em detalhes no capítulo *12. Configurações avançadas*);
- A ordem de execução dos testes nem sempre é garantida;
- É possível executar suítes de testes de forma isolada.

Faça com que os casos de teste sejam totalmente independentes uns dos outros

Testes independentes é uma boa prática não só na escrita de scripts de testes e2e, como também na escrita de testes de unidade e de integração.

Algumas vantagens da criação de testes totalmente independentes são:

- Possibilidade da execução dos testes de forma isolada;
- Facilidade de depuração.

Exceção

Quando as operações realizadas para iniciar os testes forem muito custosas, é recomendado que estes sejam então independentes em nível de arquivo. De outra forma, demorariam muito para serem executados, não provendo rápido feedback, que é um dos princípios do uso de testes automatizados.

Navegue até a página em teste antes de cada teste

Ao navegar até a página que será testada antes de cada caso de teste, garantimos que a aplicação está em um estado limpo, no qual não há o risco de que testes anteriormente executados afetem o teste em execução. Uma maneira prática para utilizar tal abordagem é o uso da função `beforeEach`. Veja um exemplo:

```
beforeEach(function() {
  homepage.visit();
});
```

Tenha uma suíte de testes que navega pelas principais rotas da aplicação

É interessante possuir uma suíte de testes voltada à navegação através das principais rotas da aplicação, tal como navegação por menus, para garantir que quando um usuário real estiver usando tal aplicação, será direcionado para a página correta.

Em testes como estes, pode haver verificações de que o usuário foi levado à URL correta, como também verificações de que os elementos corretos estão sendo exibidos (elementos que são específicos de cada tela, tais como títulos, por exemplo). Minha

recomendação neste tipo de testes é que os dois tipos de verificações sejam combinadas. Ou seja, além de verificar que o usuário foi levado à URL correta, verificar também que os elementos corretos estão sendo exibidos.

Além disso, testes voltados à navegação são normalmente testes mais rápidos de serem executados, quando comparadas com testes que interagem mais com a aplicação. Algumas vantagens desta abordagem são:

- Garantia de que as principais partes da aplicação estão corretamente conectadas;
- Usuários não navegam pela aplicação digitando URLs;
- Confiança com relação a questões relacionadas a permissões de acesso.

A definição de suítes de testes no arquivo de configuração é detalhada no capítulo *12. Configurações avançadas*.

Tenha uma suíte de smoke test

Uma suíte de *smoke test* é uma suíte de testes rápida e que garante que os principais cenários da aplicação estão funcionando conforme esperado. Estes também são chamados de cenários de "caminho feliz" (*happy path tests*).

Além disso, quando usando práticas como entrega contínua (*continuous delivery*), uma suíte de *smoke test* pode ser muito útil na utilização de *deployment pipelines*, evitando que uma suíte mais demorada, como uma de testes e2e de regressão seja executada caso exista uma falha na suíte de *smoke test*, por exemplo.

Com isso, consolidamos algumas boas práticas no que diz respeito a escrita de testes e2e automatizados, como evitar

duplicidade de teste e configurações, estratégias para bons localizadores de elementos HTML (para tornar seus testes mais robustos e legíveis), suítes de teste (o que ajuda no uso de *pipelines* de *deploy*, por exemplo), e o padrão de projetos *Page Objects* (que ajuda na manutenabilidade do projeto de testes).

Caso você tenha ficado muito interessado no padrão *Page Objects*, este será abordado no próximo capítulo em mais detalhes.

Capítulo 3

PAGE OBJECTS

Conforme já comentado no capítulo anterior, o uso do padrão Page Objects ajuda na escrita de testes limpos e com código reutilizável. Neste capítulo, demonstrarei com exemplos reais a utilização do padrão Page Objects, para que você, leitor, perceba ainda mais suas vantagens, como legibilidade e manutenabilidade dos testes.

Lembra do teste a seguir, demonstrado no capítulo introdutório do livro? Ele, após fazer uma busca na página da API do site do Protractor, verificava se o título exibido era o esperado.

```
// homepage.specs.js

describe('Homepage', function() {
  it('perform a search into the api page', function() {
    browser.get('#/api');

    element(by.model('searchTerm')).sendKeys('restart');
    element(by.css('.depth-1')).click();

    expect(element(by.css('.api-title')).getText()).toContain('bro
wser.restart');
  });
});
```

Segue a versão do mesmo teste, porém refatorado para a utilização do padrão Page Objects.

```
// homepageWithPageObjects.specs.js

var ApiPage = require('../page-objects/apiPage.po.js');
```

```
describe('Homepage', function() {
  var apiPage = new ApiPage();

  it('perform a search into the api page', function() {
    apiPage.visit();

    apiPage.searchField.sendKeys('restart');
    apiPage.firstLinkOnLeftSide.click();

    expect(apiPage.itemTitle.getText()).toContain('browser.restart
');
  });
});
```

Perceba como o teste em si ficou mais fácil de ser lido, sem expressões de difícil entendimento, tais como `element`, `browser`, `by.id`, `by.model` e `by.css`. A ideia do uso de Page Objects é tornar os testes legíveis, e encapsular os elementos e funções de páginas específicas nos arquivos de Page Objects, neste caso, arquivos com extensão `.po.js`.

Veja a seguir o Page Object da ApiPage utilizada no teste demonstrado anteriormente:

```
// apiPage.po.js

var ApiPage = function() {
  this.firstLinkOnLeftSide = element(by.css('.depth-1'));
  this.itemTitle = element(by.css('.api-title'));
  this.searchField = element(by.model('searchTerm'));
};

ApiPage.prototype.visit = function() {
  browser.get('#/api');
};

module.exports = ApiPage;
```

Tal código demonstra exatamente o encapsulamento citado, em que três elementos são expostos de forma pública, com nomes legíveis para serem usados no arquivo de teste, além da função que visita a página em questão.

3.1 REFATORANDO TESTES PARA UTILIZAÇÃO DE PAGE OBJECTS

Caso você já utilize o framework Protractor para escrita de testes e2e automatizados e ainda não utiliza o padrão Page Objects, recomendo que comece a usar. Isso facilitará sua vida ou de quem tiver que dar manutenção dos testes no futuro, além de facilitar na criação de novos casos de teste quando novas funcionalidades forem adicionadas à aplicação.

Uma forma que uso para refatorar testes que não utilizam Page Objects para que comecem a usar é a seguinte. Primeiro, reescrevo o teste como se o(s) Page Object(s) já existisse(m). Foi exatamente isso que fiz para refatorar o teste demonstrado no capítulo introdutório.

Inicialmente, requeri o Page Object, antes da definição do `describe` e antes mesmo da existência deste arquivo.

```
var ApiPage = require('../page-objects/apiPage.po.js');
```

Depois, criei uma instância deste objeto logo no início do `describe`, para que os elementos que eu viesse a implementar no Page Object estivessem disponíveis para uso durante os testes.

```
var apiPage = new ApiPage();
```

Então, para ajudar na legibilidade do teste, substituí o `browser.get` e as definições dos elementos no teste, pelo seguinte:

```
apiPage.visit()
apiPage.searchField
apiPage.firstLinkOnLeftSide
apiPage.itemTitle
```

Com isso, eu já sabia exatamente o que meu Page Object necessitaria. Criei o diretório `page-objects` dentro do diretório `e2e`, e dentro deste novo diretório criei o arquivo

`apiPage.po.js`. O diretório `page-objects` foi criado por questões de organização de código, já o arquivo em si para encapsular os elementos e funções da página em teste, separando as responsabilidades.

Então, implementei uma função chamada `visit`, da seguinte forma:

```
ApiPage.prototype.visit = function() {
  browser.get('#/api');
};
```

Gosto de ter funções chamadas `visit` para navegar até a página em questão, por questões de legibilidade, visto que é mais fácil entender um código com o seguinte:

```
apiPage.visit();
```

Do que o seguinte:

```
browser.get('#/api');
```

Implementei também três elementos públicos à função `ApiPage`, da seguinte forma:

```
this.firstLinkOnLeftSide = element(by.css('.depth-1'));
this.itemTitle = element(by.css('.api-title'));
this.searchField = element(by.model('searchTerm'));
```

Por fim, somente exportei tal Page Object, para que ele pudesse ser instanciado dentro de qualquer testes, após requerido:

```
module.exports = ApiPage;
```

Ou seja, com tal Page Object exportado, basta o seguinte para que os elementos expostos de forma pública e sua(s) função(ões) sejam acessíveis no arquivo de teste (`*.spec.js`):

```
var apiPage = new ApiPage();
```

Este mesmo processo pode ser realizado para a refatoração de

qualquer teste que não utilize o padrão Page Objects, para que passe a usar. Também torna o processo de refatoração fácil e intuitivo, visto que quando você vai enfim implementar o Page Object já sabe exatamente o que ele precisará para que seu teste continue funcionando.

É parecido com fazer TDD (*test-driven development*), em que você primeiro desenvolve o teste para depois desenvolver a funcionalidade em questão. No caso de refatoração de testes e2e para uso do padrão Page Objects, você primeiro modifica o teste como se um Page Object já existisse, e então implementa tal Page Object.

3.2 OUTROS EXEMPLOS DE PAGE OBJECTS

No capítulo anterior, falei sobre como Page Objects ajudam na eliminação de duplicidade de código, já que, uma vez que um Page Object é criado, este pode ser reaproveitado ao longo de diferentes casos de teste, ou mesmo em diferentes suítes de teste, como nos casos dos Page Objects do tipo *wrappers*. Veja a seguir um exemplo de uma suíte de testes que não utiliza Page Objects, e procure perceber códigos duplicados.

```
// todoMvc.spec.js

describe('Todo MVC Angular', function() {
  var newTodoField = element(by.id('new-todo'));

  it('add an item in the todo list', function() {
    browser.get('http://todomvc.com/examples/angularjs/#/');

    newTodoField.sendKeys('Create test without page object');
    newTodoField.sendKeys(protractor.Key.ENTER);

    expect(element.all(by.css('.view')).count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    browser.get('http://todomvc.com/examples/angularjs/#/');
```

```
    newTodoField.sendKeys('Create new test without page object');
    newTodoField.sendKeys(protractor.Key.ENTER);

    expect(element.all(by.css('.view')).count()).toEqual(2);
  });
});
```

Além do código não ser um dos mais legíveis, pois a definição dos elementos fica exposta nos testes, tanto o primeiro teste quanto o segundo executam as mesmas ações de digitar um item no campo `newTodoField`, e depois simular a tecla `ENTER` sendo pressionada.

Veja como fica a mesma suíte de teste utilizando Page Objects:

```
// todoMvcWithPageObjects.spec.js

var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  beforeEach(function() {
    todoMvc.visit();
  });

  it('add an item in the todo list', function() {
    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    todoMvc.addItemOnTodoList('Create new test without page object
');

    expect(todoMvc.listOfItems.count()).toEqual(2);
  });
});
```

Perceba que os dois testes, que antes possuíam quatro passos cada, agora possuem somente dois, visto que os passos que eram repetidos foram:

1. movidos para uma função `beforeEach` (no caso do código `todoMvc.visit();`); e
2. transformados em uma função, chamada `addItemOnTodoList`. Esta recebe como parâmetro uma string com o valor do item que se deseja adicionar a lista.

Perceba também a utilização do padrão AAA (*arrange*, *act*, *assert*), comentado no capítulo introdutório, em que agora o *arrange* está no `beforeEach`.

> A única duplicação não removida foi o `expect`, e isto é de propósito, pois, como visto no capítulo *2. Boas práticas*, todo caso de teste deve possuir uma verificação, ou seja, um `expect`.

Veja também a implementação do Page Object a seguir:

```
// todoMvc.po.js

var TodoMvc = function() {
  this.listOfItems = element.all(by.css('.view'));
  this.newTodoField = element(by.id('new-todo'));
};

TodoMvc.prototype.addItemOnTodoList = function(item) {
  this.newTodoField.sendKeys(item);
  this.newTodoField.sendKeys(protractor.Key.ENTER);
};

TodoMvc.prototype.visit = function() {
  browser.get('http://todomvc.com/examples/angularjs/#/');
};

module.exports = TodoMvc;
```

Veja que, além do método `addItemOnTodoList`, também foi criado um método `visit`. Ele encapsula o código

`browser.get('http://todomvc.com/examples/angularjs/#/');` , tornando o teste em si mais legível e sem informações desnecessárias.

3.3 CRIANDO E UTILIZANDO PAGE OBJECTS DO TIPO WRAPPER

Conforme já mencionado no capítulo *2. Boas práticas*, na seção sobre Page Objects, recomenda-se criar Page Objects do tipo *wrapper* para elementos que são comuns em diferentes telas da aplicação. A seguir, veja exemplos de testes para duas diferentes telas da mesma aplicação, porém que compartilham de elementos comuns, e que portanto usam um Page Object do tipo *wrapper*.

```
// choko.specs.js

var CreateAccountPage = require('../page-objects/chokoCreateAccoun
t.po.js');
var MessagesWrapper = require('../page-objects/chokoMessagesWrappe
r.po.js');
var SignInPage = require('../page-objects/chokoSignIn.po.js');

var messageWrapper = new MessagesWrapper();

describe('Choko - Sign in', function() {
  var signInPage = new SignInPage();

  it('try to sign in without filling any field', function() {
    signInPage.visit();

    signInPage.signInButton.click();

    expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
  });
});

describe('Choko - Create account', function() {
  var createAccountPage = new CreateAccountPage();

  it('try to create account without filling any field', function()
 {
    createAccountPage.visit();
```

```
    createAccountPage.createAccountButton.click();

    expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
  });
});
```

O primeiro teste tenta fazer um login sem preencher nenhum campo, ou seja, só pressionando o botão de *sign in*. O segundo teste tenta criar uma conta da mesma forma, ou seja, sem preencher nenhum campo, somente clicando no botão *create account*.

Visto que a mensagem de erro exibida em ambos os testes fica encapsulada em um mesmo elemento, esta é definida em um Page Object do tipo *wrapper*. Perceba o Page Object em questão (`chokoMessagesWrapper.po.js`):

```
// chokoMessagesWrapper.po.js

var MessageWrapper = function() {
  this.errorMessage = element(by.css('.alert-danger'));
};

module.exports = MessageWrapper;
```

Perceba que este é um Page Object bem simples, que somente possui um elemento exposto de forma pública. Porém, pode ser utilizado em diferentes testes. Somente para fins de exemplos adicionais, veja também os Page Objects `chokoCreateAccount.po.js` e `chokoSignIn.po.js` :

```
// chokoCreateAccount.po.js

var chokoCreateAccountPage = function() {
  this.createAccountButton = element(by.id('element-create-account
-submit'));
};

chokoCreateAccountPage.prototype.visit = function() {
  browser.get('http://choko.org/create-account');
};

module.exports = chokoCreateAccountPage;
```

Este expõe de forma pública o botão de criação de conta e uma função para visitar tal página.

```
// chokoSignIn.po.js

var ChokoSignInPage = function() {
  this.signInButton = element(by.id('element-sign-in-submit'));
};

ChokoSignInPage.prototype.visit = function() {
  browser.get('http://choko.org/sign-in');
};

module.exports = ChokoSignInPage;
```

Ele expõe de forma pública o botão de *sign in* e também uma função para visitar tal página.

Conforme mencionado no início deste capítulo, todos os exemplos demonstrados neste capítulo são exemplos reais. Portanto, caso tenha interesse, tente fazer os mesmos testes sozinho para praticar. E no próximo capítulo, veremos como criar funções ajudantes (*helpers*).

Capítulo 4

HELPERS

Em alguns projetos costumo também utilizar *helpers*, além da utilização de Page Objects para encapsular elementos específicos das telas da aplicação em teste, para a criação de funções para operações que necessitam de mais de um passo, ou mesmo para a criação de funções que ajudem na legibilidade dos testes.

Helpers podem existir para diversos motivos. Conforme o nome já diz, estes são **ajudantes**. Ou seja, podemos criar funções dentro de *helpers* para, por exemplo, ajudar em determinadas operações que podem vir a se repetir ao longo dos testes. Assim, evita-se a duplicidade de código e facilita a manutenção, além de ajudarem na legibilidade dos testes.

A diferença entre um *helper* e um Page Object é que o *helper* não encapsula elementos específicos de uma página, nem mesmo funções referentes a uma tela específica da aplicação. Ele é utilizado para questões mais genéricas, como será mostrado a seguir. Já o Page Object existe exatamente para encapsular elementos e funções de telas específicas da aplicação. Portanto, ambos podem ser usados em conjunto.

Um caso de *helper* que costumo utilizar, por exemplo, é um para a criação de strings randômicas que possam ser usadas em formulários de criação de conta, em que caso uma conta já tenha sido criada para um usuário, esta não poderá ser utilizada para o mesmo teste, em uma execução futura. Neste caso, posso gerar uma

string randômica que será o nome do usuário, garantindo que ele não se repita no próximo teste.

Outra abordagem para o mesmo problema pode ser uma rotina que limpe os dados criados durante os testes. Esta rotina pode também ser um *helper*.

Veja a seguir um exemplo de *helper* para a criação de strings randômicas, conforme comentado:

```
// helper.js

var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

O *helper* aqui demonstrado utiliza um node module chamado `node-uuid`. Por agora, saiba que este é usado para a geração de strings randômicas. Mais detalhes sobre este node module serão discutidas no próximo capítulo *5. Node modules úteis*.

Agora, veja o exemplo de um teste utilizando esta função `generateRandomString`, a partir do arquivo `helper.js`. Este visita uma página, adiciona um item em uma lista e verifica que tal item foi corretamente adicionado:

```
// todoMvcWithPageObjects.spec.js

var Helper = require('../helper')
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Anguar', function() {
  var helper = new Helper();
  var todoMvc = new TodoMvc();

  it('add random value in the todo list', function() {
```

```
    var randomString = helper.generateRandomString();

    todoMvc.visit();

    todoMvc.addItemOnTodoList(randomString);

    expect(todoMvc.listOfItems.getText()).toContain(randomString);
  });
});
```

Veja que o *helper* é também requerido como um node module, no topo do arquivo, e então instanciado logo no início do `describe`.

O teste demonstrado usa o *helper* para gerar uma string randômica, armazenada em uma variável chamada de `randomString`. Então, o teste visita a página da aplicação TODO MVC, adiciona a string randômica gerada a partir do *helper* na lista de TODOs, e por fim verifica que a lista contém o texto da string randômica recém-adicionada.

Veja que o *helper* em questão não é um Page Object, pois este não possui relação alguma com a aplicação. Ele é somente um ajudante para ser consumido pelos testes.

Veja mais um exemplo:

```
it('try to sign in with a random email and random password', funct
ion() {
  var randomEmail = helper.generateRandomEmail();
  var randomPassword = helper.generateRandomString();

  signInPage.visit();

  signInPage.usernameField.sendKeys(randomEmail);
  signInPage.passwordField.sendKeys(randomPassword);
  signInPage.signInButton.click();

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Este novo teste simula um usuário tentando realizar *sign in* na

aplicação com um usuário randômico e uma senha randômica. Perceba que, para a geração da senha randômica, o mesmo *helper* do exemplo anterior é utilizado (`generateRandomString`). Já para a geração do e-mail randômico, é usado um novo *helper*.

Veja a seguir o arquivo `helper.js` refatorado, agora contendo também a função `generateRandomEmail` :

```
// helper.js

var shortid = require('shortid');
var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

A função `generateRandomEmail` basicamente retorna um id, concatenado com a string `'@email.com'` , para garantir que é um e-mail com formato válido. Perceba que, para esta nova função, um novo node module está sendo usado (`shortid`). Da mesma forma que o node module `node-uuid` , este também será visto em detalhes no próximo capítulo.

O teste recém-demonstrado realiza uma operação que necessita de mais de um passo: preencher o campo de usuário, preencher o campo de senha e clicar no botão `sign in` . Agora, veja como ele fica refatorado, utilizando uma função para a tentativa de login, que também pode ser usada para um login bem-sucedido, caso o usuário e senha fornecidos sejam válidos:

```
it('try to sign in with a random email and random password - refac
tored', function() {
```

```
  var randomEmail = helper.generateRandomEmail();
  var randomPassword = helper.generateRandomString();

  signInPage.visit();

  signInPage.signIn(randomEmail, randomPassword);

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

> **OBSERVAÇÃO**
>
> Esta refatoração não é relacionada ao uso de *helpers*, mas visto que é uma boa prática sugerida, resolvi demonstrá-la aqui.

Ou seja, agora em vez de o usuário preencher os campos e clicar no botão em passos diferentes, estes são encapsulados na função `signIn`. Veja a função em questão:

```
ChokoSignInPage.prototype.signIn = function(email, password) {
  this.usernameField.sendKeys(email);
  this.passwordField.sendKeys(password);
  this.signInButton.click();
};
```

4.1 HELPER UTILIZANDO EXPECTED CONDITIONS

Vejamos agora um *helper* um pouco diferente dos vistos até então. Há algum tempo, escrevi alguns testes automatizados que estavam falhando mesmo sem haver algo realmente errado na aplicação, causando falsos negativos.

Conforme já comentado no capítulo sobre **2. Boas práticas**, testes que resultam em falsos negativos são prejudiciais ao time. Testes automatizados devem prover resultados reais sobre o

comportamento da aplicação em teste e devem ser confiáveis.

No caso destes testes em específico, o problema era que, quando o script de teste escrito com o framework Protractor tentava interagir com um determinado elemento, ocorria um erro de que o elemento não havia sido encontrado. O caso era que, quando se tentava clicar em tal elemento, ele não estava visível ainda.

A princípio, o webdriver deveria lidar com estas questões, mas nem sempre isso é o que ocorre. Isso pode ocorrer por diversas questões, como problemas de *timing*, e nestes casos podemos utilizar *expected conditions* (EC).

Para resolver tal problema, criei um *helper*. Veja a seguir o código deste *helper* (na verdade, é o mesmo helper já demonstrado até então, mas com uma nova função, a última):

```
// helper.js

var EC = protractor.ExpectedConditions;
var shortid = require('shortid');
var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

Helper.prototype.waitElementVisibility = function(element) {
  browser.wait(EC.visibilityOf(element), 3000);
};

module.exports = Helper;
```

Ou seja, criei um *helper* que recebia um elemento como argumento, e então aguardava por **até** 3000 milissegundos (3 segundos) para que este elemento estivesse visível. Assim, só

interagindo com ele quando este já estivesse disponível para isso.

Um ponto interessante do uso de tal abordagem é que, caso o elemento esteja visível logo no primeiro segundo, por exemplo, o teste não vai aguardar os próximos dois, prosseguindo com o próximo passo, que normalmente é a interação com o próprio elemento, ou uma verificação.

No exemplo seguinte, imagine que o campo `usernameField` não está disponível de imediato logo após a visita à tela de *Sign in*. Neste caso, o *helper* vem a calhar. Ou seja, antes de interagir com tal elemento, posso aguardar que ele esteja visível.

```
it('try to sign in just filling the email field', function() {
  var randomEmail = helper.generateRandomEmail();

  signInPage.visit();
  helper.waitElementVisibility(signInPage.usernameField);

  signInPage.usernameField.sendKeys(randomEmail);
  signInPage.signInButton.click();

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Somente para fins de curiosidade, outros exemplos que *helpers* podem ser úteis são: geração de telefones de um determinado país, para refrescar a tela, ou para fazer o *sign up* em uma aplicação. Ou seja, é possível criar *helpers* para as mais diversas situações, e isso depende de contexto.

Além disso, você pode utilizar *helpers* para questões mais complexas, dependendo de suas necessidades. Alguns exemplos de *helpers* mais complexos que já usei são: para obter o ID de um dispositivo a partir do armazenamento local do navegador, e a geração de um código para login ou *sign up*, para sistemas que exigem tal código de verificação, sendo estes casos bem específicos.

No próximo capítulo, veremos node modules úteis.

Capítulo 5

NODE MODULES ÚTEIS

Uma das partes boas de desenvolver testes e2e automatizados com Protractor é que você pode se beneficiar de diversos node modules existentes para complementar esta atividade. Isso porque o Protractor é um framework baseado em Node.js.

Neste capítulo, há uma lista de node modules úteis que utilizo para resolver diferentes problemas quando estou escrevendo testes e2e automatizados.

Antes de começar, caso seu projeto ainda não possua um arquivo para o gerenciamento das dependências do projeto, execute o seguinte comando para gerar o arquivo `package.json` (aceite todas as opções padrão):

```
npm init
```

Com isso, sempre que você instalar um node module utilizando no NPM, o arquivo `package.json` será atualizado, adicionando as novas dependências.

5.1 JASMINE-SPEC-REPORTER

O `jasmine-spec-reporter` melhora o feedback dos testes executados em seu console (ou terminal). Ou seja, após executar os testes, em vez de seu terminal exibir algo como o seguinte:

```
[22:10:31] I/hosted - Using the selenium server at http://localhos
t:4444/wd/hub
```

```
[22:10:31] I/launcher - Running 1 instances of WebDriver
Started
...


3 specs, 0 failures
Finished in 4 seconds
[22:10:37] I/launcher - 0 instance(s) of WebDriver still running
[22:10:37] I/launcher - chrome #01 passed
```

Utilizando o node module `jasmine-spec-reporter`, você terá feedback mais detalhado, como:

```
[22:13:20] I/hosted - Using the selenium server at http://localhos
t:4444/wd/hub
[22:13:20] I/launcher - Running 1 instances of WebDriver
Spec started
Started

  1 Todo MVC Anguar
    ✓ add an item in the todo list (3 secs)
.    ✓ add new item in the todo list (0.958 sec)
.    ✓ add random value in the todo list (1 sec)
.
Executed 3 of 3 specs SUCCESS in 5 secs.


3 specs, 0 failures
Finished in 4.623 seconds
[22:13:26] I/launcher - 0 instance(s) of WebDriver still running
[22:13:26] I/launcher - chrome #01 passed
```

Utilizando o jasmine-spec-reporter

Para começar a utilizar o `jasmine-spec-reporter`, siga os seguinte passos: execute o seguinte comando, a partir da raiz do seu projeto, para instalar o `jasmine-spec-reporter` como dependência de desenvolvimento do seu projeto.

```
npm install jasmine-spec-reporter --save-dev
```

Então atualize o arquivo de configuração do Protractor, adicionando a ele o seguinte:

- Antes do `module.exports.config` :

```
var SpecReporter = require('jasmine-spec-reporter');
```

- Dentro do `module.exports.confg` :

```
onPrepare: function() {
jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
}));
}
```

Com o node module instalado e essa nova configuração, você já terá um feedback muito melhor após executar os testes em seu console (ou terminal), conforme já demonstrado. Alguns detalhes sobre esta configuração:

- `displayFailuresSummary: true,` — exibe um resumo de todas as falhas após a execução dos testes;
- `displayFailedSpec: true,` — exibe cada teste que falhou;
- `displaySuiteNumber: true,` — exibe o número de cada suíte de testes (de forma hierárquica, caso este seja o caso);
- `displaySpecDuration: true` — exibe o tempo de duração de cada arquivo com extensão `.spec` `duration` .

Todas as configurações disponíveis para o node module `jasmine-spec-reporter` podem ser encontradas na seguinte URL: https://www.npmjs.com/package/jasmine-spec-reporter

Mais detalhes sobre a configuração `onPrepare` podem ser consultados no capítulo *12. Configurações avançadas.*

5.2 PROTRACTOR-JASMINE2-HTML-REPORTER

O `protractor-jasmine2-html-reporter` é uma alternativa para a geração de relatório de execução de testes e2e em formato HTML, com a possibilidade da adição de screenshots de cada teste.

Utilizando o protractor-jasmine2-html-reporter

Para instalar o `protractor-jasmine2-html-reporter` como dependência de desenvolvimento de seu projeto, execute o seguinte comando:

```
npm install protractor-jasmine2-html-reporter --save-dev
```

Após a instalação do node module, atualize o arquivo de configuração do Protractor, conforme o seguinte:

```
// protractor.conf.js

var Jasmine2HtmlReporter = require('protractor-jasmine2-html-repor
ter');
var SpecReporter = require('jasmine-spec-reporter');

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  onPrepare: function() {
    jasmine.getEnv().addReporter(new SpecReporter({
      displayFailuresSummary: true,
      displayFailedSpec: true,
      displaySuiteNumber: true,
      displaySpecDuration: true
    }));

    jasmine.getEnv().addReporter(new Jasmine2HtmlReporter({
      takeScreenshots: true,
      fixedScreenshotName: true
    }));
```

```
  }
};
```

Atenção à linha no início do arquivo, onde o node module `protractor-jasmine2-html-reporter` é requerido. Veja também as linhas finais, onde um novo *reporter* é adicionado, instanciando um objeto `Jasmine2HtmlReporter` e definindo para este dois atributos: `takeScreenshots: true, fixedScreenshotName: true` . O primeiro para que, além do relatório em formato HTML, também sejam tiradas screenshots de cada teste, e o segundo para que as screenshots tiradas tenham nome fixo em vez de randômico.

Todas as configurações disponíveis para o node module `protractor-jasmine2-html-reporter` podem ser encontradas em: https://www.npmjs.com/package/protractor-jasmine2-html-reporter. Execute os testes com a nova configuração e, após a execução, abra no navegador o arquivo `htmlReport.html` gerado. Ele deve lhe mostrar algo como:

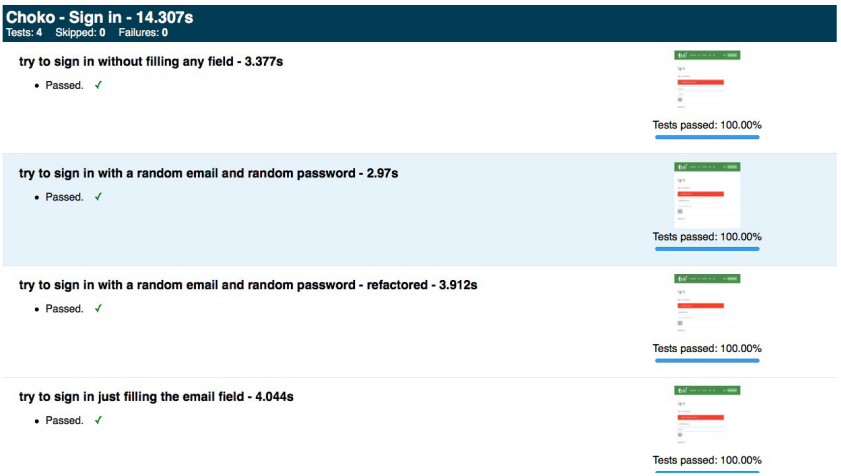

Figura 5.1: protractor-jasmine2-html-reporter

Com esta configuração, a cada nova execução dos testes, o relatório e as screenshots serão sobrescritos.

5.3 SHORTID

Com `shortid`, é possível criar IDs curtos e não sequenciais.

Utilizando o shortid

Para instalar o node module `shortid` como dependência do seu projeto, execute o seguinte comando:

```
npm install shortid --save-dev
```

Após a instalação, ele estará pronto para ser usado. Veja a seguir um exemplo da utilização deste node module, retirado do capítulo *4. Helpers*:

```
// helper.js

var shortid = require('shortid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

module.exports = Helper;
```

Perceba que, após o node module ser requerido, com apenas uma linha de código, um id curto pode ser gerado:

```
shortid.generate();
```

Segue a lista de caracteres padrão utilizados pelo `shortid`:

```
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-_'
```

Mais detalhes sobre o node module `shortid` podem ser encontrados na seguinte URL: https://www.npmjs.com/package/shortid.

5.4 NODE-UUID

Com `node-uuid`, é possível criar IDs baseados em data e hora, ou IDs randômicos.

Utilizando o node-uuid

Para instalar o node module `node-uuid` como dependência do seu projeto, execute o seguinte comando:

```
npm install node-uuid --save-dev
```

Após a instalação, eLe estará pronto para ser utilizado. Segue um exemplo, retirado do capítulo *4. Helpers*:

```
// helper.js

var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

Após requerido o node module `node-uuid` com uma única linha de código, você pode gerar IDs randômicos:

```
return uuid.v4();
```

Para gerar IDs baseados em data e hora, use o seguinte:

```
uuid.v1();
```

Mais detalhes sobre o node module `node-uuid` podem ser encontrados em: https://www.npmjs.com/package/node-uuid.

5.5 FS

O node module `fs` possibilita interações com o sistema de arquivos (*file system*).

Utilizando o fs

Para instalar o node module `fs` como dependência do seu projeto, execute o seguinte comando:

```
npm install fs --save-dev
```

Após a instalação, este estará pronto para ser utilizado.

Um exemplo do uso deste node module, que já usei em alguns projetos para fins de sobrescrever as configurações do Protractor com base em um arquivo de configuração local (não versionado pelo git) é o seguinte:

```
// protractor.conf.js

var fs = require('fs');

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
};

// Local config customization.
try {
  fs.statSync(__dirname + '/config.local.js');
  require(__dirname + '/config.local.js')(module.exports.config);
} catch(error) {
  return;
}
```

No início, o node module `fs` é requerido. Após a definição das configuração, um `try catch` verifica a existência de um arquivo chamado `config.local.js`, no mesmo diretório onde o arquivo `protractor.conf.js` se encontra. Caso o arquivo seja encontrado, este é requerido, sobrescrevendo as propriedades do arquivo `protractor.conf.js` pelas configurações definidas neste arquivo. Caso o arquivo não seja encontrado, o `catch`

simplesmente executa um `return`.

Veja a seguir o arquivo `config.local.js`, que sobrescreve a propriedade `baseUrl` pelo valor `'http://localhost:8000/'`, e adiciona uma nova propriedade `directConnect` com o valor `true`. Esta, quando verdadeira, desconsidera a propriedade `seleniumAddress`, utilizando o WebDriver do próprio navegador (nos casos do Chrome e Firefox).

```
// config.local.js

/**
 * Configuration alter method.
 * @param {object} Current Protractor configuration, as defined in
 * protractor.conf.js file.
 */

module.exports = function (config) {
  config.baseUrl = 'http://localhost:8000/';
  config.directConnect = true;
};
```

Mais detalhes sobre o node module `fs` podem ser encontrados na seguinte URL: https://nodejs.org/api/fs.html.

5.6 BROWSERSTACK-LOCAL

O BrowserStack (https://www.browserstack.com/) é um SaaS (*Software as a Service*) que provê uma combinação de diferentes sistemas operacionais, navegadores e dispositivos para testes manuais e automatizados. Além disso, com este serviço, é possível realizar testes em ambiente local de desenvolvimento, possibilitando que, mesmo durante o desenvolvimento, testes possam ser executados em diferentes navegadores, por exemplo.

Para isto, existe o node module `browserstack-local`. Ou seja, com este node module, é possível executar testes automatizados em navegadores na nuvem, porém contra seu ambiente local de

desenvolvimento.

Mais detalhes sobre o BrowserStack no capítulo *8. Testes na nuvem*.

Utilizando o browserstack-local

Para instalar o node module `browserstack-local` como dependência do seu projeto, execute o seguinte comando:

```
npm install browserstack-local --save-dev
```

Com o node module `browserstack-local` instalado, atualize o arquivo de configuração do Protractor, requerindo o node module `browserstack-local` e criando uma instância dele. Também adicione as capabilities `browserstack.user`, `browserstack.key` e `browserstack.local`, com os valores `process.env.BROWSERSTACK_USERNAME`, `process.env.BROWSERSTACK_ACCESS_KEY` e `true`, respectivamente. Então adicione os atributos `beforeLaunch` e `onComplete`, conforme a seguir:

```
// protractor.conf.js

var browserstack = require('browserstack-local');
var browserstackLocal = new browserstack.Local();

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome',
    'browserstack.user': process.env.BROWSERSTACK_USERNAME,
    'browserstack.key': process.env.BROWSERSTACK_ACCESS_KEY,
    'browserstack.local': 'true'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://localhost:8080/',
  beforeLaunch: function() {
    return new Promise(function(resolve, reject) {
      const browserstackLocalArgs = {
        'key': process.env.BROWSERSTACK_ACCESS_KEY,
        'force': 'true',
```

```
        'forcelocal': 'true'
      };
      browserstackLocal.start(browserstackLocalArgs, function(erro
r) {
        if (error) {
          reject(error);
          return;
        }
        resolve();
      });
    });
  },
  onComplete: function() {
    return new Promise(function(resolve, reject) {
      browserstackLocal.stop(function(error) {
        if (error) {
          reject(error);
          return;
        }
        resolve();
      });
    });
  }
};
```

Este código depende da definição das seguintes variáveis de ambiente, que devem conter as credenciais de sua conta no serviço do BrowserStack:

- `BROWSERSTACK_USERNAME`
- `BROWSERSTACK_ACCESS_KEY`

A documentação do BrowserStack para projetos em Node.js pode ser encontrada na seguinte URL: https://www.browserstack.com/automate/node. Detalhes sobre as funções `beforeLaunch` e `onComplete` são apresentados no capítulo *12. Configurações avançadas*.

Por hora, entenda que o `browserstack-local` é iniciado antes dos testes começarem e finalizado após a execução de todos os testes, habilitando que os testes sejam executados em seu ambiente local. Perceba que a `baseUrl` está apontando para

`http://localhost:8080` , para fins de exemplo.

Mais detalhes sobre o node module `browserstack-local` podem ser encontrados em: https://www.npmjs.com/package/browserstack-local.

Diversos outros node modules são encontrados a partir da URL: https://www.npmjs.com/. Portanto, caso tenha alguma necessidade específica, lembre de procurar por algo pronto antes de implementar sua própria solução. É muito provável que alguém já tenha resolvido tal problema, salvando-lhe tempo.

No próximo capítulo, veremos ações e verificações.

Capítulo 6

AÇÕES E VERIFICAÇÕES

Ações e verificações são uma parte importante na escrita de testes automatizados e2e. Elas fazem parte do padrão AAA (*arrange*, *act* e *assert*), mencionado no capítulo introdutório, em que as ações são o A de *act* e as verificações o A de *assert*.

O framework Protractor dispõe de uma lista de diversas ações e verificações que podem ser realizadas quando escrevemos scripts de testes automatizados, que serão discutidas a seguir.

6.1 AÇÕES

Ações servem para simular usuários interagindo com as aplicações, como: navegação por meio de URLs, cliques em botões, preenchimentos de campos, limpeza de campos, entre outros. A seguir, são apresentadas algumas ações comuns que podem ser realizadas usando o framework Protractor.

Ação de navegação: browser.get(url);

Navega até uma página por uma URL, podendo ser o argumento `url` uma URL relativa ou absoluta.

OBSERVAÇÃO

Caso uma `baseUrl` esteja definida no arquivo de configuração, basta passar como argumento à função `get()` uma URL relativa, que será concatenada com a `baseUrl`, formando a URL absoluta na qual se deseja navegar.

Veja alguns exemplos:

```
browser.get('https://leanpub.com/testes-e2e-com-protractor'); // n
avegação através de URL absoluta

browser.get('testes-e2e-com-protractor'); // navegação através de
URL relativa. Imagine que a baseUrl definida no arquivo de configu
ração seja a 'https://leanpub.com/'
```

O primeiro exemplo navega até a página deste livro, através de sua URL absoluta. Já o segundo exemplo navega até uma página a partir de uma URL relativa. Neste caso, uma `baseUrl` deve estar definida no arquivo de configuração, que poderia ser `https://leanpub.com/`.

A principal vantagem da utilização de URL relativa com relação a URL absoluta está na possibilidade de sobrescrever a `baseUrl` definida no arquivo de configuração. Isso permite que os mesmos testes sejam executados, por exemplo, em diferentes ambientes (como ambiente de testes, local de desenvolvimento ou de produção).

Ação de clique: element.click();

Clica em um determinado elemento HTML. Veja alguns exemplos:

```
var button = element(by.id('submit'));
```

```
button.click(); // clique em um botão com id 'submit'

element(by.css('a .my-link')).click(); // clique em um link com um
a classe css 'my-link'
```

Ação de preenchimento de campo: element.sendKeys('text');

A função `sendKeys()` é normalmente usada para o preenchimento de elementos HTML do tipo `input`. Esta ação pode ser utilizada para o preenchimento de campos de texto, simulação de pressionamento de teclas do teclado, ou até mesmo para envio de arquivos em campos `input` do tipo `file`.

Veja alguns exemplos:

```
element(by.model('searchTerm')).sendKeys('restart'); // digitação
do valor 'restart' em um campo que possui o model 'searchTerm'

var newTodoField = element(by.id('new-todo'));
newTodoField.sendKeys(protractor.Key.ENTER); // simulação do press
ionamento da tecla ENTER em um campo com id 'new-todo'

var element = element(by.css('input[type=file]'));
element.sendKeys('/path/to/file.txt'); // simulação do envio de um
 arquivo em um campo input do tipo file
```

Ação de limpeza de campo: element.clear();

Algumas vezes antes de simular a digitação em um campo, é necessário limpá-lo. Para isso, o Protractor possui a função `clear()`. Veja um exemplo:

```
var searchField = element(by.model('searchTerm'));
var text = 'restart';
searchTerm.sendKeys(text);
searchField.clear(); // limpeza de um elemento definido pelo model
 'searchField' após ele ter recebido um valor do tipo string (um t
exto)
expect(searchField.getText()).not.toContain(text)
```

Ação de refrescar a tela: browser.refresh();

Em um determinado caso de teste, pode ser necessário simular a tela sendo refrescada. Para isso, o Protractor possui a função `refresh()`.

Um exemplo pode ser para verificar que uma lista de TODOs continua com os valores previamente preenchidos, mesmo após a tela do navegador ser refrescada. Veja um exemplo:

```
var newTodoField = element(by.id('new-todo'));
var text = 'Create test without page object';

newTodoField.sendKeys(text);
newTodoField.sendKeys(protractor.Key.ENTER);
browser.refresh(); // refrescando a tela do navegador

expect(element.all(by.css('.view')).getText()).toContain(text);
```

6.2 VERIFICAÇÕES

Verificações são o que tornam a automação um teste propriamente dito, pois após uma pré-condição e uma ou mais ações, verifica-se se um resultado esperado ocorreu. Portanto, as verificações são parte importante na escrita de script de testes automatizados e2e.

A seguir são apresentadas algumas verificações comuns que podem ser realizadas utilizando o framework Protractor.

toEqual()

A verificação de igualdade pode ser utilizada, por exemplo, para verificar que o texto de um determinado elemento HTML é exatamente igual ao texto passado como argumento. Veja um exemplo:

```
var foo = element(by.id('my-id'));
expect(foo.getText()).toEqual('some text'); // verificação de que
o elemento localizado pelo id 'my-id' possui exatamente o texto 's
ome text'
```

Esta verificação pode também ser usada em conjunto com a função `count()`, por exemplo. Veja alguns exemplos:

```
expect(element.all(by.css('.view')).count()).toEqual(1); // verifi
cação de que o array de elementos identificados pela classe css 'v
iew' contém somente um elemento

expect(element.all(by.css('.view')).count()).toEqual(3); // verifi
cação de que o array de elementos identificados pela classe css 'v
iew' contém três elementos
```

toContain()

A verificação `toContain` pode ser usada para verificar que um determinado elemento HTML contém um determinado texto, podendo ser verificado só parte do texto contido em tal elemento. Veja um exemplo:

```
var foo = element(by.id('my-id'));
expect(foo.getText()).toContain('some text'); // verificação de qu
e o elemento localizado pelo id 'my-id' contém o texto 'some text'
```

toBe()

Verificação normalmente utilizada para o retorno de promessas e valores booleanos. Veja alguns exemplos:

```
expect(messageWrapper.errorMessage.isDisplayed()).toBe(true); // v
erificação de que uma mensagem de erro está sendo exibida

expect(browser.getCurrentUrl()).toBe('https://leanpub.com/testes-e
2e-com-protractor'); // verificação de que o retorno da promessa d
a URL atual é 'https://leanpub.com/testes-e2e-com-protractor'
```

not

Verificação de negação, que pode ser usada em conjunto com qualquer uma das outras verificações já vistas, para negar tal expectativa. Veja alguns exemplos:

```
var foo = element(by.id('my-id'));
```

```
expect(foo.getText()).not.toEqual('some text'); // verificação de
que o elemento localizado pelo id 'my-id' não possui exatamente o
texto 'some text'

var foo = element(by.id('my-id'));
expect(foo.getText()).not.toContain('some text'); // verificação d
e que o elemento localizado pelo id 'my-id' não contém o texto 'so
me text'

expect(messageWrapper.errorMessage.isDisplayed()).not.toBe(true);
// verificação de que uma mensagem de erro não está sendo exibida

expect(browser.getCurrentUrl()).not.toBe('https://leanpub.com/test
es-e2e-com-protractor'); // verificação de que o retorno da promes
sa da URL atual não é 'https://leanpub.com/testes-e2e-com-protract
or'
```

Neste capítulo, foram vistas as principais ações e verificações que podem ser realizadas durante a escrita de scripts de testes e2e com o framework Protractor. No próximo capítulo, veremos como adicionar um passo adicional após cada verificação, para a realização de testes de revisão visual.

Capítulo 7

TESTES DE REVISÃO VISUAL

Durante o desenvolvimento de software, mesmo automatizando testes, ainda são necessárias verificações manuais, como testes exploratórios e testes para verificar questões relacionadas ao estilo (alterações de CSS que podem quebrar a aplicação visualmente, por exemplo).

Testes e2e automatizados usualmente existem para realizar testes funcionais, ou seja, verificar que as funcionalidades da aplicação estão de acordo com a perspectiva do usuário final. Porém, hoje em dia, é possível integrar testes e2e automatizados com outras ferramentas, a fim de ajudar a realizar verificações de estilo somente quando diferenças visuais são encontradas.

Uma alternativa de integração ao framework Protractor para facilitar tais atividades é o `VisualReview-protractor` , API do **VisualReview**. Ele tem o objetivo de fornecer um fluxo de trabalho produtivo e amigável para testes e revisões de layout de aplicações web para qualquer teste de regressão.

A ideia é a seguinte: com base em testes de regressão e2e já existentes, ele adiciona passos para tirar screenshots de cada página acessada, para posterior revisão visual, com a facilidade de uma interface que ajuda neste fluxo do trabalho.

O Visual Review trabalha com um conceito de screenshots que

já foram revisadas previamente, que são usadas como base para comparação dos screenshots tiradas no momento dos testes de regressão e2e. Caso alguma diferença seja encontrada com relação aos screenshots base e aos do momento dos testes, esta diferença é exibida através de uma interface gráfica. Isso possibilita a quem for fazer a revisão visual aprovar ou rejeitar tal screenshot.

Quando um screenshot é aprovado, ele é então substituído pelo screenshot base. Quando rejeitado, continua valendo o screenshot base, e neste caso, provavelmente um bug visual foi encontrado, necessitando de uma ação para sua correção. Quando nenhuma diferença é encontrada entre o screenshot base e o do momento da execução dos testes e2e, este é automaticamente aprovado, não necessitando da revisão visual.

Links para o VisualReview-protractor e VisualReview

https://github.com/xebia/VisualReview-protractor

https://github.com/xebia/VisualReview

7.1 INTEGRANDO O VISUALREVIEW AO PROTRACTOR

Para instalar o `VisualReview-protractor` como dependência de desenvolvimento de seu projeto, execute o seguinte comando:

```
npm install visualreview-protractor --save-dev
```

Após a instalação, é necessário adicionar algumas configurações. No arquivo `protractor.conf.js`, antes do `module.exports.config`, adicione as seguintes linhas:

```
const VisualReview = require('visualreview-protractor');
```

```
var vr = new VisualReview ({
  hostname: 'localhost',
  port: 7000
});
```

Esta configuração define a biblioteca do visual review em uma constante chamada `VisualReview` e, logo após, um objeto do tipo `VisualReview` é instanciado e armazenado em uma variável `vr`. Esta possui os atributos `hostname` com valor `localhost`, e `port` com valor `7000`.

Em seguida, dentro do `module.exports.config`, adicione as seguintes configurações (os nomes do projeto e da suíte de testes podem ser alterados conforme sua necessidade):

```
beforeLaunch: function () {
  vr.initRun('Visual-Review-Sample-Project', 'visualReviewSuite');
},

afterLaunch: function (exitCode) {
  return vr.cleanup(exitCode);
},

params: {
  visualreview: vr
}
```

Ou seja, defina uma função `beforeLaunch`, que inicializa o **VisualReview** passando como argumento o nome do projeto e da suíte de testes a sua escolha, em formato de strings. Defina também a função `afterLaunch`, que retorna a execução do `cleanup` e serve para a limpeza de arquivos temporários gerados durante a execução dos testes.

Também é definido um parâmetro `visualreview` com valor `vr`, que será utilizado no arquivo de testes (`*.spec.js`). Com estas configurações, basta adicionar duas linhas de código nos testes (arquivos com extensão `.spec.js`) para fazer o **Visual Review** funcionar. Segue um exemplo:

```
// sample.spec.js

var vr = browser.params.visualreview;

describe ('Sample', function () {
  it ('title is correct', function () {
    browser.get('/#');
    expect(element(by.id('title')).getText()).toEqual('Sample');
    vr.takeScreenshot('sample-home');
  });
});
```

No início do arquivo, a variável `vr` é definida com o valor do atributo `visualreview`, definido na configuração `params`. E dentro do teste, logo após a verificação (`expect`), um screenshot é tirado e nomeado.

Além da instalação do **VisualReview-protractor**, antes da execução dos testes, é necessário baixar e descompactar a última versão do **Visual Review**. Para encontrar a última versão, consulte a seguinte URL: https://github.com/xebia/VisualReview/releases.

Em seguida, execute o seguinte comando a partir do diretório onde o **Visual Review** foi descompactado, para inicializá-lo:

```
./start.sh
```

Com isso, bastar executar o Protractor para que os testes já estejam integrados ao **Visual Review**. Após a execução dos testes, acessando pelo navegador o endereço http://localhost:7000, você verá algo com o seguinte:

Figura 7.1: visualreview-screen1

Acessando o link **Visual-Review-Sample-Project**, a seguinte tela deve ser exibida:

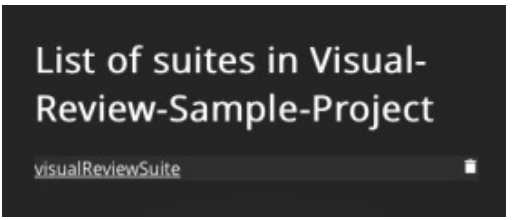

Figura 7.2: visualreview-screen2

Clicando no link **visualReviewSuite**, você terá acesso à lista das últimas execuções dos testes. Algo como o seguinte:

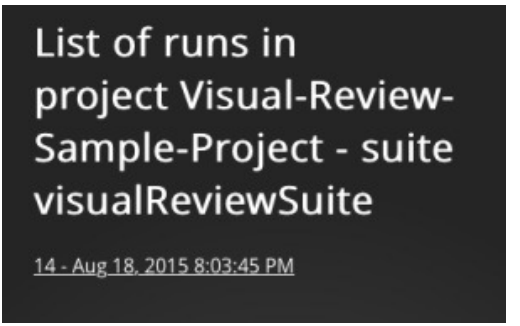

Figura 7.3: visualreview-screen3

Enfim, acessando o link da última execução dos testes, você terá acesso à interface gráfica para aprovação ou rejeição das screenshots. Algo como o seguinte:

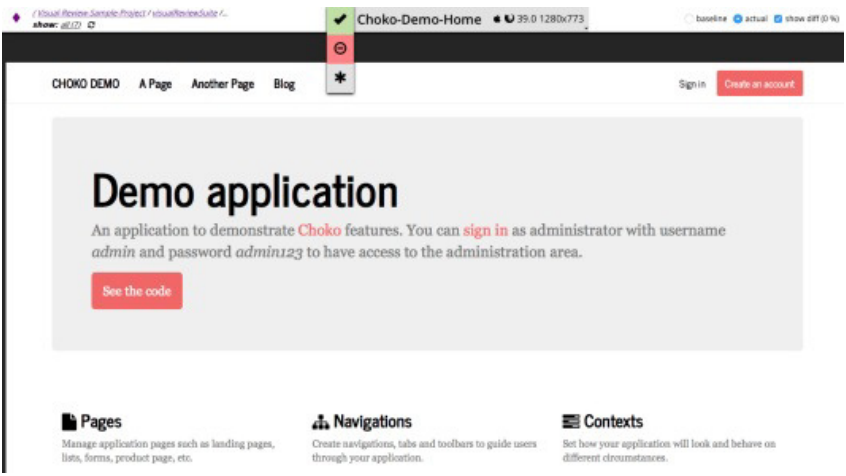

Figura 7.4: visualreview-screen4

Na primeira execução dos testes, caso todos os screenshots estejam exibindo a aplicação conforme esperado, eles devem ser aprovados, para serem usados como base de comparação para as próximas execuções dos testes. E a partir das próximas execuções, caso alguma diferença seja encontrada com relação ao screenshot base e ao tirado no momento da execução dos testes, ele então pode ser aprovado ou rejeitado, no momento da revisão visual.

Com isso, você pode incrementar facilmente o processo de testes de seus projetos de desenvolvimento de software que rodam testes com o framework Protractor, para garantir ainda mais qualidade nas aplicações entregues.

7.2 O QUE TESTAR E O QUE NÃO TESTAR COM O VISUAL REVIEW

- Analise seus testes e2e antes de adicionar passos de revisão visual a eles. Ao analisá-los, procure entender em quais momentos é interessante adicionar tal integração, como momentos em que mudanças visuais acontecem na aplicação (mudanças de telas, exibição e mensagens etc.).

- Procure entender quais testes são mais adequados para este tipo de abordagem. Os testes de revisão visual existem para lhe poupar tempo. Portanto, se em todos os testes sempre existe diferença visual, algo está errado na escolha dos testes que estão usando esta abordagem. Ou seja, procure adicionar passos de revisão visual em parte da aplicação que não vai ficar mudando toda hora visualmente.

- Se o layout de uma determinada tela ainda está em fase de experimentação, não utilize revisão visual nela.

- Adicione passos de revisão visual às telas da aplicação que já são estáveis e as quais não há previsão de serem alteradas em um futuro próximo.

- Adicione passos de revisão visual às telas que não possuem elementos dinâmicos, tais como anúncios, data, hora e outros.

- Também não é recomendado usar tais testes em telas que possuem gifs animados ou vídeos autoexecutáveis, visto que a cada execução haverá diferenças, mesmo que a aplicação não esteja "quebrada" visualmente.

Como fonte de leitura adicional, recomendo o seguinte post que escrevi em meu blog em um momento que fui utilizar o

VisualReview e me deparei com um erro. Após sua correção, resolvi compartilhar: https://talkingabouttesting.com/2016/06/03/cannot-read-property-platform-of-undefined/.

Agora que você já possui uma boa base sobre o que são e para que servem testes de revisão visual, como integrar o framework **Protractor** com o **VisualReview**, quando utilizar ou não, e também uma dica relacionada à resolução de problemas, recomendo-lhe começar a colocar tal aprendizado em prática.

No próximo capítulo, veremos como executar testes e2e automatizados na nuvem.

Capítulo 8

TESTES NA NUVEM

O que fazer quando precisamos testar a mesma aplicação em diferentes navegadores? Ou mesmo em diferentes combinações de navegadores e sistemas operacionais? E quando precisamos testar a mesma aplicação em navegadores e dispositivos móveis, como *smartphones*?

Uma das vantagens dos testes e2e é a possibilidade de executarmos suítes de testes em diferentes combinações de sistemas operacionais, navegadores e dispositivos. Em nossos próprios computadores, até podemos ter mais de um navegador para executarmos os testes, ou até mesmo máquinas virtuais com outros sistemas operacionais. Porém, existem serviços especializados que fornecem uma variedade destas combinações, inclusive com versões mais antigas de navegadores, para quando o sistema desenvolvido precisar atender uma necessidade específica.

Além disso, a utilização de tal tipo de serviço junto com ferramentas de integração contínua diminuiu o tempo de feedback após alterações realizadas na aplicação.

Alguns destes serviços para testes automatizados na nuvem são o BrowserStack e o SauceLabs, ambos disponíveis para integração com o Protractor. Vejamos cada um deles.

Links para o BrowserStack e SauceLabs:

http://browserstack.com/

http://saucelabs.com/

8.1 BROWSERSTACK

O **BrowserStack** é um serviço de testes na nuvem baseado em **Selenium**, e oferece mais de 1.000 combinações diferentes entre sistemas operacionais, navegadores e dispositivos. É um sistema pago, mas dispõe de uma versão *trial* com 100 minutos para execução de testes automatizados.

Integrando scripts de teste escritos com Protractor ao BrowserStack

Para habilitar a execução de testes escritos com o framework Protractor ao BrowserStack, é necessário fazer algumas alterações no arquivo de configurações do Protractor, já visto no capítulo introdutório. Para fins de relembrar, o arquivo de configuração basicamente define o endereço em que o servidor do Selenium estará rodando, o navegador contra o qual os testes serão executados, os testes propriamente ditos, e a URL base. Porém, neste caso, algumas novas configurações são adicionadas.

Veja um exemplo:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://hub.browserstack.com/wd/hub',
  capabilities: {
    'browserName': 'chrome',
```

```
    'browser_version': '52',
    'os': 'OS X',
    'os_version': 'El Capitan',
    'browserstack.user': process.env.BROWSERSTACK_USERNAME,
    'browserstack.key': process.env.BROWSERSTACK_ACCESS_KEY
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Basicamente o que muda é o `seleniumAddress`. Ele recebe o endereço do servidor do Selenium do BrowserStack e as `capabilities`. Estas, além de agora definirem versão do navegador, sistema operacional e sua versão, também recebem dois novos atributos (`browserstack.user` e `browserstack.key`).

Observação

Veja que o valor das credenciais do BrowserStack (`browserstack.user` e `browserstack.key`) apontam para as variáveis de ambiente `BROWSERSTACK_USERNAME` e `BROWSERSTACK_ACCESS_KEY` . Isto é feito dessa forma para que tais dados sensíveis não fiquem *hard coded* no arquivo de configuração, por questões de segurança. Portanto, para isso, é preciso que tais variáveis sejam criadas com os valores corretos (usuário e chave de sua conta no BrowserStack).

Para definir variáveis de ambiente em um ambiente Linux ou OS X, edite o arquivo `/ect/bashrc` . Ao final deste arquivo, adicione suas variáveis e salve o arquivo. Veja um exemplo:

```
BROWSERSTACK_USERNAME=meu-usuario-no-browserstack
BROWSERSTACK_ACCESS_KEY=minha-chave-no-browserstack
```

Após salvar o arquivo, execute o seguinte comando, para que o sistema operacional reconheça as novas variáveis definidas:

```
source /etc/bashrc
```

Com isso, ao rodar o Protractor, seus testes serão executados em um navegador Chrome na nuvem do BrowserStack. Você pode também executar os mesmos testes usando a configuração de `multiCapabilities` para disparar testes em diferente combinações de navegadores, sistemas operacionais e dispositivos. Veja um exemplo:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://hub.browserstack.com/wd/hub',
  browserstack.user: process.env.BROWSERSTACK_USERNAME,
  browserstack.key: process.env.BROWSERSTACK_ACCESS_KEY,
```

```
  multiCapabilities: [
    {
      'browserName': 'chrome',
      'browser_version': '52.0',
      'os': 'OS X',
      'os_version': 'El Capitan',
    },
    {
       browserName: 'firefox',
       browser_version: '47.0',
       os: 'Windows',
       os_version: '7'
    },
  ]
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Neste exemplo, ao executar os testes, eles vão rodar em paralelo em um OS X El Capitan no navegador Chrome versão 52 e, em um Windows 7, no navegador Firefox versão 47, na nuvem do BrowserStack. Além disso, conforme demonstrado no capítulo *5. Node modules úteis*, é possível executar testes na nuvem do BrowserStack contra um ambiente local de desenvolvimento, através de um túnel, utilizando o node module `browserstack-local`

8.2 SAUCELABS

O **SauceLabs** é outro serviço de testes na nuvem também baseado em **Selenium** e oferece mais de 700 combinações diferentes entre sistemas operacionais, navegadores e dispositivos. É um sistema pago, dispõe de uma versão *trial* por duas semanas, 90 minutos para execução de testes automatizados e até 8 testes rodando em paralelo.

Integrando scripts de teste escritos com Protractor ao SauceLabs

Para habilitar a execução de testes escritos com o framework Protractor ao SauceLabs, é necessário fazer algumas alterações no arquivo de configurações do Protractor, semelhante ao que foi feito quando configuramos o BrowserStack. Veja um exemplo:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  username: process.env.SAUCELABS_USERNAME,
  accessKey: process.env.SAUCELABS_ACCESS_KEY,
  multiCapabilities: [
    {
      'browserName': 'chrome',
      'browser_version': '52.0',
      'os': 'OS X',
      'os_version': 'El Capitan',
    },
    {
       browserName: 'firefox',
       browser_version: '47.0',
       os: 'Windows',
       os_version: '7'
    },
  ]
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Assim como o BrowserStack, o que muda é basicamente o `seleniumAddress`. Ele aponta para o servidor Selenium do SauceLabs e as credenciais de acesso, que também são obtidas por meio de variáveis de ambiente, visto que são dados sensíveis. Além disso, também é possível executar testes com diferentes `capabilities` para rodar a mesma suíte de testes utilizando diferentes combinações de sistemas operacionais, navegadores e dispositivos.

O SauceLabs também dispõe da habilidade de execução de testes em ambiente local de desenvolvimento, usando o Sauce Connect.

SauceConnect

SauceConnect é um servidor *proxy*. Ou seja, ele fica entre a aplicação no ambiente local de desenvolvimento e o **SauceLabs**.

Com o SauceConnect, é possível estabelecer uma conexão segura para que o **SauceLabs** execute testes em seus navegadores em ambiente virtual contra a aplicação, rodando em seu ambiente local de desenvolvimento, ou também contra uma aplicação que roda em uma rede com *firewall* que bloqueia o acesso à aplicação.

Mais detalhes sobre o Sauce Connect podem ser obtidos pela seguinte URL: https://wiki.saucelabs.com/display/DOCS/Setting+Up+Sauce+Connect.

Ambos os serviços (BrowserStack e SauceLabs) suportam diversas linguagens de programação, dentre elas o **JavaScript**, esta utilizada pelo framework Protractor. Eles também podem ser integrados com ferramentas de integração contínua, como **Jenkins**, **SemaphoreCI**, **Travis CI** etc. E ambos também dispõem de suporte para ajudar na resolução de problemas.

Algumas funcionalidades interessantes disponíveis em ambos os serviços são os screenshots tirados automaticamente durante a execução dos testes (podendo ser usados como evidências dos testes). Além disso, também há os vídeos que são gravados durante a execução dos testes, que podem ser posteriormente reproduzidos para ajudar na identificação de problemas quando testes falham.

No próximo capítulo, veremos questões sobre como integrar testes e2e automatizados, que podem ser executados na nuvem, com serviços de integração contínua.

Capítulo 9

INTEGRAÇÃO CONTÍNUA

Integração contínua (ou CI) é uma prática de desenvolvimento de software, na qual o código é continuamente integrado (ao menos uma vez por dia, pelo desenvolvedor) e de forma automatizada. CI também diz respeito a verificar se o novo código que foi escrito quebrou ou não o que já estava funcionando, uma vez que testes e outras tarefas automatizadas (como verificações de sintaxe) são executados quando integrando o código.

Utilizar CI permite que times de desenvolvimento de software tenham feedback rápido sobre as mudanças que estão fazendo em uma aplicação específica. Além disso, também é uma maneira mais barata para correção de problemas quando estes são encontrados, visto que o código modificado ainda está fresco na memória dos desenvolvedores.

CI também é uma das práticas da disciplina de programação eXtrema, criada por Kent Beck e Ron Jeffries, em 1997. Após utilizarem CI (e outras práticas da XP) e terem bons resultados nos projetos nos quais trabalhavam, decidiram escrever a respeito, como uma forma de compartilhar tal conhecimento com o resto do mundo, para então termos melhores softwares.

Um dos pontos cruciais ao utilizar CI é a diminuição de conflitos quando integramos código. Uma vez que o código é frequentemente integrado (de um ramo específico para o ramo principal), ele tem menos chances de quebrar o que já existia. E

mesmo que ele quebre algo que já funcionava, fica mais fácil de resolver.

Outra questão importante quando se fala de CI é que tal prática precisa ser suportada por uma suíte de **testes automatizados**. Não apenas testes de unidade, mas também testes de integração, e ainda melhor, se possível, por testes end-to-end.

9.1 TESTES E2E NO PROCESSO DE INTEGRAÇÃO CONTÍNUA

Com a ajuda de ferramentas para a realização de testes na nuvem, é possível adicionar testes e2e no processo de integração contínua, juntamente com a utilização de alguma ferramenta específica para isto, como o **SemaphoreCI** (http://semaphoreci.com/), demonstrado a seguir.

Integrando testes e2e escritos com o framework Protractor ao SemaphoreCI

O **SemaphoreCI** é uma ferramenta de integração contínua para agilizar os ciclos de desenvolvimento de software e permitir um fluxo de trabalho escalável. É possível utilizá-lo de forma gratuita para projetos públicos, além de poder fazer um *trial* por 30 dias para projetos privados, ou pagar por diferentes planos, de acordo com suas necessidades.

Com o **SemaphoreCI**, é possível testar seu código a cada mudança, e realizar *deploys* de forma rápida e segura. Além disso, ele é facilmente customizável a diferentes necessidades, como a definição de variáveis de ambiente criptografadas (como credenciais para acesso a serviços de testes na nuvem), integração com ferramentas de comunicação para notificações dos status dos *builds* (integração com ferramentas de chat como Slack e Hipchat, ou

ferramentas de e-mail), dentre outras.

Para integrar testes e2e escritos com o Protractor ao SemaphoreCI, para a execução de testes na nuvem do SauceLabs, por exemplo, as seguintes alterações são necessárias no arquivo de configurações do Protractor.

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  username: process.env.SAUCELABS_USERNAME,
  accessKey: process.env.SAUCELABS_ACCESS_KEY,
  capabilities: {
      'browserName': 'chrome',
      'name': 'SemaphoreCI, SauceLabs and Protractor test suite'
  }
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Veja que as alterações necessárias são basicamente as já mencionadas no capítulo *8. Testes na nuvem*, além de um nome que agora também foi dado a suíte de testes. Os passos para iniciar a integração são:

1. Crie uma conta no **SemaphoreCI** através da seguinte URL: http://semaphoreci.com/.

2. Crie um projeto. Caso seu projeto utilize **GitHub** para o versionamento de código, após a criação de sua conta no **SemaphoreCI**, é possível criar um novo projeto a partir de sua conta no **GitHub**. Veja a imagem:

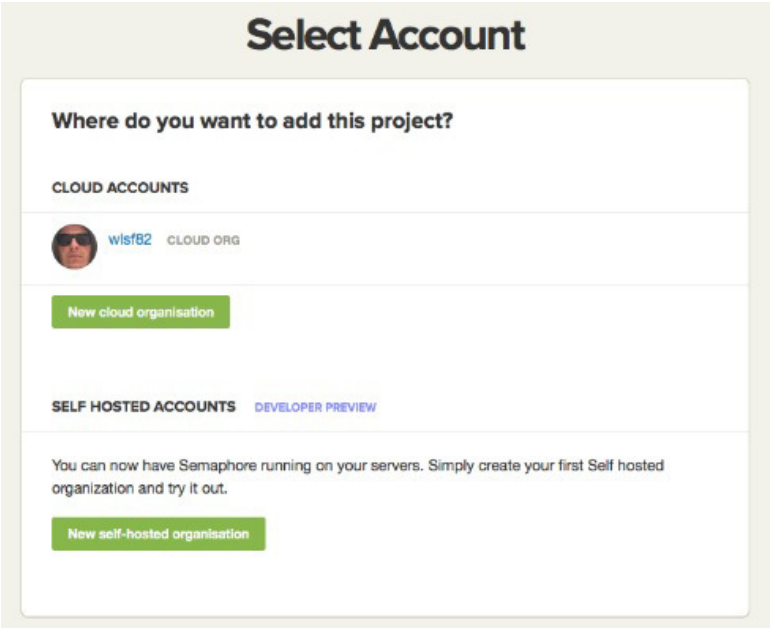

Figura 9.1: semaphoreci-screen1

Clicando no link de seu usuário no **GitHub**, você será direcionado para a próxima tela, para buscar pelo projeto que você deseja integrar ao **SemaphoreCI**. Veja:

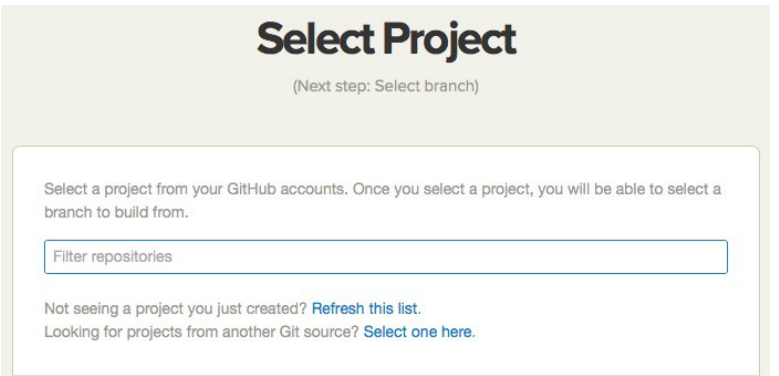

Figura 9.2: semaphoreci-screen2

Busque e selecione o projeto.

3. Configure o *build*. Após buscar e selecionar o projeto, é necessário definir as configurações do **build**. Veja:

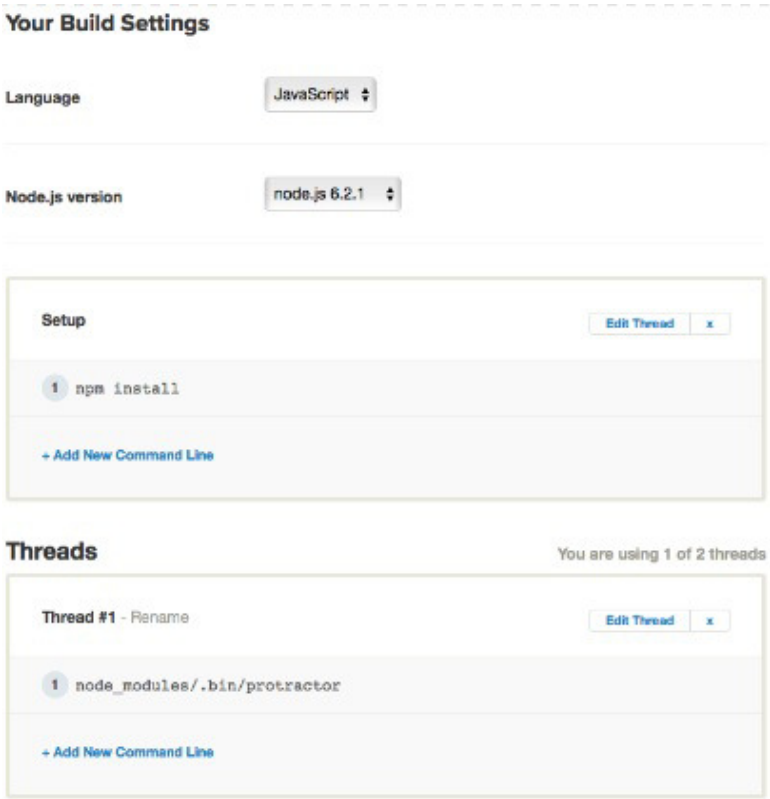

Figura 9.3: semaphoreci-screen3

Visto que o Protractor é baseado em Node.js, a linguagem selecionada deve ser o JavaScript, e também é necessário selecionar a versão do Node.js. Nas configurações do build, no setup é preenchido o `npm install` para instalar as dependências do projeto. Na *thread*, a execução do Protractor é preenchida a partir do binário instalado no diretório node modules.

4. Defina as variáveis de ambiente. Visto que as credenciais de acesso do **SauceLabs** são dados sensíveis, é possível armazená-los no **SemaphoreCI** em variáveis de ambiente, criptografadas. Veja:

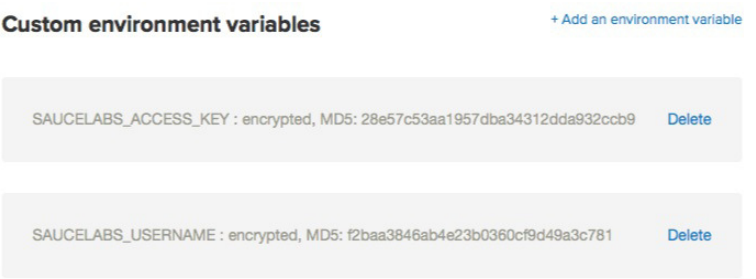

Figura 9.4: semaphoreci-screen4

   Também é possível definir diferentes tarefas para serem executadas em diferentes *branchs* do repositório, definir notificações por e-mail ou ferramentas de chat, como **Slack**, para quando os testes falharem, por exemplo.

Mais informações sobre o **SemaphoreCI** podem ser obtidas através da seguinte URL: https://semaphoreci.com/docs/javascript-continuous-integration.html.

Com estas configurações, após fazer um git push no *branch* configurado para disparar os testes pelo SemaphoreCI, ele automaticamente executará os testes na nuvem do **SauceLabs** (neste exemplo) contra a aplicação definida pela `baseUrl` . Esta foi definida no arquivo de configuração do Protractor, neste caso, o site oficial do **Protractor**.

Veja a seguir um exemplo do dashboard do **SemaphoreCI** após a execução de alguns builds, que resultaram em *builds* verdes (os testes passaram) e um *build* vermelho (os testes falharam). Além disso, também é possível identificar tarefas que foram disparadas

automaticamente (por um git push no branch configurado), ou disparadas manualmente (também possível através da interface do **SemaphoreCI**).

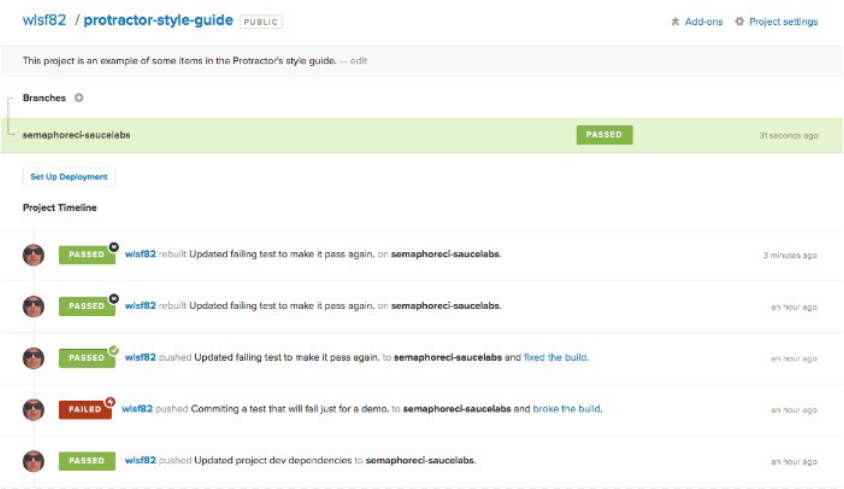

Figura 9.5: semaphoreci-screen5

Concluindo, com poucos passos, é possível adicionar testes escritos com o framework **Protractor** para serem executados na nuvem a cada momento que o código é enviado para um repositório remoto, com a ajuda de uma ferramenta de CI. Isso ajudará em um feedback rápido ao time de desenvolvimento, principalmente quando alterações na aplicação quebraram algo que funcionava, podendo o time agir rapidamente para remediar tal problema, seja fazendo um *hotfix*, ou mesmo um *rollback* para a versão anterior.

No próximo capítulo, veremos testes para *mobile*, que também podem tirar proveito de integração contínua para suas execuções.

Capítulo 10

TESTES PARA MOBILE

Quando se fala de automação de teste de software atualmente, além de se pensar em testes para aplicações web que rodam em computadores convencionais ou notebooks, também precisamos pensar em testes para dispositivos móveis. Isso porque eles, hoje em dia, são tão ou até mesmo mais utilizados que os computadores convencionais.

O que fazer quando precisamos testar uma aplicação em dispositivos móveis, como um *smartphone*? Neste capítulo, serão demonstradas algumas alternativas para testes de aplicações em dispositivos móveis.

Redefinindo o tamanho do navegador

Uma maneira de testar uma aplicação responsiva, que na mesma versão é apresentada de formas diferentes dependendo da dimensão do navegador, é redimensionando a janela do navegador. Ao redimensionar um navegador para 320 por 568, por exemplo, é possível simular as dimensões de um iPhone 5.

Veja a seguir como redimensionar a janela do navegador para simular as dimensões de um dispositivo móvel:

```
browser.driver.manage().window().setSize(320, 568);
```

Esta pode ser uma abordagem válida, quando se fala de aplicações responsivas.

10.1 SIMULANDO UM DISPOSITIVO MÓVEL NO NAVEGADOR

Ao criar scripts de teste, algumas vezes não basta somente redimensionar a janela do navegador para simular um dispositivo móvel. Esta abordagem pode não funcionar, por exemplo, se a aplicação que você está testando possui uma versão web específica para mobile, diferente de uma aplicação apenas responsiva.

Em casos como este, uma abordagem que pode ser usada é a simulação de um dispositivo móvel através da configuração de *capabilities*. Veja um exemplo:

```
capabilities: {
  'browserName': 'chrome',
  'chromeOptions': {
    'mobileEmulation': {
      'deviceName': 'Google Nexus 5'
    }
  }
}
```

Utilizando esta configuração, os testes serão executados no navegador Chrome, porém utilizando o simulador de um dispositivo Google Nexus 5 do próprio navegador.

10.2 UTILIZANDO SIMULADORES DE DISPOSITIVOS MÓVEIS NA NUVEM

Outra forma para testar aplicações em dispositivos móveis pode ser usando os recursos disponíveis em serviços de testes na nuvem, como o **BrowserStack** ou **SauceLabs**, mencionados no capítulo *8. Testes na nuvem*. Para a realização de testes para *mobile*, podemos usar um servidor web chamado Appium, baseado em **Selenium**, que consegue "pilotar" dispositivos móveis.

APIUM:

Acesse http://appium.io.

Veja a seguir um exemplo de configuração para executar testes no **SauceLabs**, simulando um iPhone 4s:

```
module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  specs: ['*.spec.js'],
  capabilities: {
    browserName: 'safari',
    'appium-version': '1.5',
    platformName: 'iOS',
    platformVersion: '9.2',
    deviceName: 'iPhone 4s',

    username: process.env.SAUCELABS_USERNAME,
    accessKey: process.env.SAUCELABS_ACCESS_KEY,

    'name': 'iOS simulation'
  },
  baseUrl: 'http://appium.io/'
};
```

Veja que, além do `seleniumAddress` apontando para o servidor do **SauceLabs** e das *capabilities*, agora também há uma propriedade chamada `'appium-version'`, com valor `'1.5'`. Esta é a única diferença necessária que permitirá a execução de testes em dispositivos móveis.

OBSERVAÇÃO

Conforme já mencionado em outros capítulos, as credenciais de acesso aos serviços de testes na nuvem são passadas através de variáveis de ambiente, visto que são dados sensíveis. Portanto, estas variáveis devem estar definidas com os valores corretos no servidor que vai executar os testes.

Para relembrar como definir tais variáveis de ambiente, consulte o capítulo *8. Testes na nuvem*, na seção sobre o serviço BrowserStack.

Veja mais um exemplo, agora simulando um celular Android:

```
module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  specs: ['*.spec.js'],
  capabilities: {
    browserName: 'chrome',
    'appium-version': '1.5',
    platformName: 'Android',
    platformVersion: '4.4',
    deviceName: 'Samsung Galaxy S4 Emulator',

    username: process.env.SAUCELABS_USERNAME,
    accessKey: process.env.SAUCELABS_ACCESS_KEY,

    'name': 'Android simulation'
  },
  baseUrl: 'http://appium.io/'
};
```

Ou seja, existem diversas maneiras de realizar testes para aplicações em dispositivos móveis, basta entender qual é a mais adequada para cada situação. Pode ser que você tenha uma aplicação responsiva, e somente redefinir as dimensões da janela do

navegador pode atender sua necessidade. Já para casos em que existe uma aplicação *mobile* específica, a simulação de um dispositivo no navegador pode ser a melhor opção. Ou então, uma opção que serve para ambos os casos é o uso de dispositivos reais ou simuladores em serviços na nuvem.

Como conteúdo adicional, neste vídeo demonstro como realizar testes simulando um dispositivo, rodando o sistema operacional iOS em ambiente local de desenvolvimento: https://talkingabouttesting.com/2016/02/29/integrando-protractor-e-appium-para-testes-automatizados-de-dispositivos-moveis/.

Para mais informações sobre testes para *mobile*, consulte a documentação oficial do **Protractor** através da seguinte URL: http://www.protractortest.org/#/mobile-setup.

No próximo capítulo, veja como escrever testes e2e com **Protractor** em ECMAScript 2015.

Capítulo 11

ECMASCRIPT 2015

ECMAScript é a linguagem de programação base do JavaScript, utilizada pelo framework **Protractor** para a escrita dos *scripts* de testes automatizados. Até então, os códigos demonstrados neste livro usaram a versão 5 do ECMAScript (ES5 ou ECMAScript 5).

Em junho de 2015, foi lançada a versão 6 do ECMAScript, nomeada como ECMAScript 2015, que neste capítulo será referenciada por **ES2015**.

Um dos objetivos do **ES2015** é prover um melhor suporte a um grande número de aplicações, a criação de bibliotecas, e para o uso de ECMAScript para a compilação de outras linguagens. Algumas das melhorias incluem módulos, declaração de classes, escopagem, iteradores e geradores, promessas para programação assíncrona, padrões de desestruturação e *tail calls* mais adequadas.

> *Tail call* é uma sub-rotina executada ao final de uma ação ou procedimento.

A partir da versão 6 do Node.js, a maioria das utilidades do **ES2015** são suportadas. Visto que o **Protractor** é baseado em Node.js, é possível tirar proveito dessas mudanças para a escrita de código ainda mais legível e com suas facilidades.

Veja adiante algumas mudanças quando escrevendo código de testes com **Protractor** em **ES2015**.

11.1 ARQUIVO DE CONFIGURAÇÃO EM ES2015

Segue um exemplo de um arquivo de configuração do Protractor, utilizando `use strict`, que será explicado a seguir.

```
// protractor.conf.js ES2015

'use strict';

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Neste exemplo do arquivo de configuração do **Protractor**, a única diferença é a utilização do `'use strict'`, para garantir que o JavaScript seja executado somente em modo estrito. Com isso, por exemplo, variáveis não podem ser usadas se não forem definidas. Além disso, ao utilizar `'use strict'`, garantimos a escrita de código JavaScript mais seguro, assegurando que erros sintáticos causarão erros.

Veja um exemplo:

```
'use strict';

a = 123; // Isto causa um erro dizendo que 'a' não está definido.
```

Também é possível utilizar o modo estrito em ES5, porém, quando se trata de **ES2015**, a utilização de tal modo é obrigatória, a não ser na definição de módulos, onde isso já é implícito. Mas vale utilizar mesmo na definição de módulos para manter o padrão.

Veja outro arquivo de configuração quando comparada com ES5, agora também com a função `onPrepare` definida de forma diferente quando estava em **ES2015**:

```
// protractor.conf.js ES2015

'use strict';

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
  onPrepare() {
    browser.driver.manage().window().maximize();
  },
};
```

A função `onPrepare` é usada neste exemplo para maximizar a janela do navegador antes de os testes começarem. Já em **ES2015**, é definida como vista neste exemplo. Veja como a mesma função seria definida utilizando ES5:

```
// função onPrepare definida utilizando ES5

onPrepare: function() {
  browser.driver.manage().window().maximize();
}
```

Veja que, em **ES2015**, a definição da função `onPrepare` é simplificada, não necessitando a definição de um atributo, sendo seu valor uma função (`function() {}`). Mais sobre a função `onPrepare` pode ser visto no capítulo *12. Configurações avançadas*.

11.2 ARQUIVOS DE TESTE (SPEC FILES) EM ES2015

Veja a seguir um exemplo de arquivo de teste escrito utilizando **ES2015**. Ele visita uma página, adiciona um item a uma lista, e

verifica que tal item foi corretamente adicionado:

```
'use strict';

const Helper = require('../helper');
const TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', () => {
  const helper = new Helper();
  const todoMvc = new TodoMvc();

  it('add random value in the todo list', () => {
    const randomString = helper.generateRandomString();

    todoMvc.visit();

    todoMvc.addItemOnTodoList(randomString);

    expect(todoMvc.listOfItems.getText()).toContain(randomString);
  });
});
```

A primeira mudança com relação aos códigos já vistos nos outros capítulos é novamente a utilização do `'use strict'`, já explicado anteriormente, quando falamos sobre o arquivo de configuração.

A segunda mudança é que agora o *helper*, o *page object*, suas instâncias e uma string randômica são definidos utilizando `const` em vez de `var`. Em **ES2015**, `const` é utilizado para a definição de valores que não poderão ser alterados ao longo do código, no mesmo escopo. Veja alguns exemplos para facilitar o entendimento:

```
'use strict';

const a = 'texto'

a = 'outro texto' // isso não é permitido é causa um erro de compi
lação
```

Em **ES2015**, para valores que precisam ser alterados ao longo do código (diferente do ES5, em que se utiliza `var`), usamos `let`. Veja:

```
'use strict';

let a = 'texto'

a = 'outro texto' // alteração permitida!
```

A terceira e última mudança é no *callback* da função `it`. Ele, em **ES2015**, utiliza o que chamamos de *array functions*. Veja a seguir:

```
'use strict';

it('add random value in the todo list', () => {});
```

Este mesmo código em ES5 seria assim:

```
it('add random value in the todo list', function() {});
```

Ou seja, é necessário escrever menos código para a definição do *callback*. Em **ES2015**, funções não nomeadas são definidas como *array functions*. O mesmo vale para funções não nomeadas que passam argumentos. Veja um exemplo:

```
'use strict';

const sayHello = (name) => {
    console.log('Hello ' + name + '!');
}

sayHello('João'); // Imprime no console 'Hello João!'
```

11.3 PAGE OBJECTS E HELPERS EM ES2015

As diferenças mais notáveis quando escrevendo testes com **Protractor** utilizando **ES2015**, em minha opinião, está na escrita de *Page Objects* ou *Helpers*. Veja alguns exemplos adiante.

Veja o arquivo *Helper*, que gera e-mails e strings randômicas:

```
// helper.js

'use strict';
```

```
const shortid = require('shortid');
const uuid = require('node-uuid');

class Helper {
  generateRandomEmail() {
    return shortid.generate() + '@email.com';
  }

  generateRandomString() {
    return uuid.v4();
  }
}

module.exports = Helper;
```

Perceba que:

- O *helper* utiliza a boa prática do mode estrito;
- O *helper* define os node modules utilizando `const` ;
- O *helper* é definido como uma classe em vez de uma simples função;
- O *helper* possui métodos em vez de definir funções a partir do protótipo da função *helper*. Neste caso, os métodos são: `generateRandomEmail` e `generateRandomString` .

Agora, o arquivo *Page Object*. Ele encapsula os elementos da página em questão e expõe métodos para adicionar itens em uma lista e visitar tal página:

```
// todoMvc.po.js

'use strict';

class TodoMvc {
  constructor() {
    this.listOfItems = element.all(by.css('.view'));
    this.newTodoField = element(by.id('new-todo'));
  }

  addItemOnTodoList(item) {
    this.newTodoField.sendKeys(item);
    this.newTodoField.sendKeys(protractor.Key.ENTER);
```

```
  }

  visit() {
    browser.get('http://todomvc.com/examples/angularjs/#/');
  }
}

module.exports = TodoMvc;
```

Já no *Page Object*, perceba que:

- O *Page Object* utiliza a boa prática do mode estrito;
- O *Page Object* é definido como uma classe em vez de uma simples função;
- O *Page Object* define um construtor para definição de seus elementos expostos de forma pública;
- O *Page Object* possui métodos, sendo neste caso os métodos `addItemOnTodoList` e `visit` .

A utilização de **ES2015** para a escrita de *Page Objects* e *Helpers*, na minha opinião, torna o código mais legível, facilitando ainda mais a manutenção dos testes sempre que necessário.

Veja no link a seguir um projeto exemplo de testes escritos com o framework **Protractor** utilizando **ES2015**, que criei para fins de expor alguns exemplos: https://github.com/wlsf82/protractor-es6.

Para mais informações sobre **ES2015**, recomendo a leitura da documentação oficial, em: http://www.ecma-international.org/ecma-262/6.0/.

No próximo capítulo, veremos as configurações avançadas.

Capítulo 12

CONFIGURAÇÕES AVANÇADAS

No capítulo introdutório do livro, foram vistas as configurações básicas para a execução de testes automatizados com o framework **Protractor**. Porém, diversas outras configurações podem ser feitas, possibilitando outras facilidades, como: a utilização do *webdriver* do próprio navegador, a definição de diferentes frameworks de testes (Jasmine e Mocha), a execução de testes em paralelo, a definição de suítes de teste, dentre outras questões. Neste capítulo, serão abordadas estas outras configurações.

12.1 UTILIZANDO O WEBDRIVER DO PRÓPRIO NAVEGADOR: DIRECTCONNECT

A configuração `directConnect` pode ser utilizada quando usamos os navegadores Firefox ou Chrome. Ela dispensa a necessidade de um servidor Selenium rodando, visto que, quando utilizamos os navegadores já mencionados, é possível usar o *webdriver* deles próprios. Veja um exemplo:

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
```

```
  baseUrl: 'http://www.protractortest.org/'
};
```

Por padrão, a configuração `directConnect` tem o valor `false`. Porém, quando definida com o valor `true`, a configuração `seleniumAddress` é ignorada, ou como neste exemplo, nem mesmo necessária. Neste caso, o **Protractor** usa o *webdriver* do próprio navegador Chrome para a execução dos testes.

Esta é uma configuração útil para testes em ambiente local de desenvolvimento.

12.2 DEFININDO UM FRAMEWORK BASE PARA A ESCRITA DE TESTES

Com o framework **Protractor**, é possível definir qual framework base será utilizado para a escrita dos scripts de teste. O framework base padrão é o Jasmine versão 2.x.

Jasmine

Jasmine é um framework de testes de unidade para JavaScript no qual os scripts de teste são escritos utilizando a técnica de desenvolvimento guiado por testes. Sua sintaxe é simples e de fácil compreensão.

Além disso, conforme já mencionado, **Jasmine** é o framework base padrão do framework **Protractor**. Portanto, se você já leu o livro até aqui, você já está acostumado com a sintaxe do **Jasmine**.

Atualizando do Jasmine 1.x para 2.x

Nas versões mais antigas do **Protractor**, o framework base padrão era o **Jasmine** versão 1.3. Porém, mesmo usando uma versão mais antiga do **Protractor**, é possível utilizar uma versão mais atual do **Jasmine**.

Uma das vantagens do **Jasmine** versão 2.x é a possibilidade do uso das funções `beforeAll` e `afterAll`, não disponíveis na versão 1.x. Estas configurações podem ser utilizadas para a definição de pré e pós-condições para a suíte de testes, como, por exemplo, a criação de uma conta de usuário antes de os testes começarem, e sua deleção ao final dos testes.

Para usar a versão 2.x do **Jasmine** em uma versão antiga do **Protractor**, defina o framework conforme demonstrado no código a seguir:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  framework: 'jasmine2'
};
```

> **Observação**
>
> Apesar da possibilidade da utilização do **Jasmine** versão 2.x em uma versão mais antiga do **Protractor**, o recomendado é usar a versão mais atual do **Protractor**, na qual o **Jasmine** versão 2.x já é o padrão.

Mais informações sobre o framework **Jasmine** podem ser encontrados no site oficial da ferramenta, através da seguinte URL: http://jasmine.github.io.

Mocha

Mocha é um framework de testes de unidade para JavaScript rico em funcionalidades e que simplifica a escrita de testes assíncronos. Para a utilização do framework **Protractor** com o framework base **Mocha**, os seguintes node modules precisam ser instalados: `mocha` , `chai` e `chai-as-promised` .

É recomendado instalar tais node modules como dependências de desenvolvimento. Portanto, utilize os seguintes comandos:

```
npm install mocha --save-dev
npm install chai --save-dev
npm install chai-as-promised --save-dev
```

Após a instalação das dependências, para o uso do **Mocha**, este deve ser definido no arquivo de configurações do **Protractor**, conforme:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  framework: 'mocha'
};
```

E para a utilização do framework **Mocha** nos arquivos de teste do **Protractor**, é necessário requerer e configurar o **Chai**. Veja:

```
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');

chai.use(chaiAsPromised);
var expect = chai.expect;
```

Então é possível utilizar **Chai as Promised** da seguinte forma, em que o retorno da promessa `getText()` pode ser igual ao valor passado entre aspas. Eventualmente visto que o teste é assíncrono.

```
expect(myElement.getText()).to.eventually.equal('some text');
```

A seguir, é exibido um exemplo simples de um teste utilizando como base o framework **Mocha**. Ele basicamente acessa uma página e verifica a existência do logo:

```
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');

chai.use(chaiAsPromised);
var expect = chai.expect;

describe('Protractor website', function() {
  it('logo', function() {
    var logo = element(by.css('.protractor-logo'));

    browser.get('#');

    expect(logo.getAttribute('class')).to.eventually.equal('protra
ctor-logo');
  });
});
```

A principal diferença está na verificação. Em vez de esperarmos que a classe do logo seja `protractor-logo` , espera-se que ela **eventualmente** seja igual a este valor, visto que o Mocha é usado para realização de testes de códigos que executam de forma assíncrona.

Mais informações sobre o framework **Mocha** podem ser encontradas no site oficial da ferramenta, pela seguinte URL: http://mochajs.org.

> Para mais detalhes sobre a escolha do framework base para utilizar com o **Protractor**, consulte: http://www.protractortest.org/#/frameworks.

12.3 EXECUTANDO TESTES EM PARALELO:

SHARDTESTFILES

O que fazer quando existem tantos testes e2e, e a execução destes começa a demorar? Como já foi visto, testes automatizados existem para prover rápido feedback ao time após mudanças na aplicação. Porém, quando os testes começam a demorar para prover tal feedback, uma ação precisa ser tomada.

Tal demora pode ser explicada por uma grande quantidade de casos de teste. Uma forma de resolver tal problema é a paralelização dos testes.

Visto que o **Protractor** é baseado no **Selenium**, é possível utilizar os recursos de testes em **grid** para a execução de testes em paralelo. Para executar de testes em paralelo com o **Protractor**, basta adicionar algumas propriedades às *capabilities* definidas no arquivo de configurações. Veja:

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'
    shardTestFiles: true,
    maxInstances: 2

  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Nesta configuração, a propriedade `shardTestFiles: true` é usada para permitir a paralelização dos testes. Já a propriedade `maxInstances: 2` é utilizada para definir quanto navegadores estarão disponíveis para a execução de testes em paralelo (neste caso, 2).

Ou seja, com esta configuração, os testes serão paralelizados em

dois navegadores diferentes. Por exemplo, caso haja dois arquivos com extensão `.spec.js`, cada um seria executado em paralelo em um navegador, diminuindo pela metade o tempo de execução dos testes. Caso necessário, para rodar testes em paralelo em mais de dois navegadores, basta definir o valor desejado a propriedade `maxInstances`.

SUGESTÕES COMPLEMENTARES DE VÍDEOS

Há um tempo atrás, gravei dois vídeos demonstrando com a "mão na massa" como executar testes em paralelo, tanto em ambiente de desenvolvimento como na nuvem. Os vídeos podem ser acessados através das seguintes URLs, respectivamente:

- https://www.youtube.com/watch?v=KllelI_Cd30
- https://www.youtube.com/watch?v=rhWT3Ev1ROA

SUGESTÃO DE LEITURA COMPLEMENTAR

Paul Yoder escreveu um post bem explicativo sobre paralelização de testes com **Protractor**. O post pode ser consultado em: http://blog.yodersolutions.com/run-protractor-tests-in-parallel/.

12.4 SUÍTES DE TESTE

Uma suíte de teste pode ser definida por um arquivo com extensão `.spec.js`, por exemplo. Caso os arquivos de teste sejam

separados por funcionalidades, poderíamos dizer que existem suítes de teste diferentes para diferentes funcionalidades da aplicação.

Porém, além disso, podemos definir diferentes tipos de suítes de teste, como uma suíte de *smoke test* (para verificar as principais funcionalidades da aplicação e só os cenários de caminho feliz), ou uma para as principais rotas da aplicação (para verificar que os usuário são direcionados para as rotas certas quando navegando pela aplicação), ou ainda, uma que roda somente os testes de *mobile*, por exemplo.

Com o **Protractor**, é possível definir suítes de teste no arquivo de configuração. Assim, podemos executar a partir da linha de comando somente os testes de uma determinada suíte.

Veja a seguir um exemplo de definição de uma suíte de *smoke test*, ou seja, uma suíte de teste para verificar que as principais funcionalidades da aplicação funcionam em seus cenários de caminho feliz:

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'

  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  suites: {
    smoke: 'specs/smokeTests.spec.js',
  },
};
```

A propriedade `suites` recebe como valor um objeto, o qual pode definir uma ou mais suítes de teste. Nesse exemplo, é definida uma suíte de *smoke teste* que aponta para um arquivo encontrado no diretório `specs` e com nome `smokeTests.spec.js`.

Para execução de uma determinada suíte, tal como a de *smoke test* recém-definida, o seguinte comando pode ser executado a partir da linha de comando. Ele rodará somente os arquivos definidos para tal suíte, desconsiderando qualquer outro arquivo definido pelo array de specs:

```
protractor --suite smoke
```

Esta configuração pode ser útil quando usamos *pipelines* de *deploy* onde, antes da execução de uma suíte de testes de regressão, pode ser executada uma suíte de *smoke test*. Assim, garante-seque não haverá perda de tempo executando testes mais demorados caso algum falhe na fase anterior.

12.5 ANTES DE QUALQUER CONFIGURAÇÃO DE AMBIENTE: BEFORELAUNCH

A função `beforeLaunch` pode ser útil para a definição de questões que são necessárias antes de qualquer configuração de ambiente, como, por exemplo, a inicialização de um túnel para execução de testes em serviços na nuvem contra ambiente local de desenvolvimento, ou a inicialização de algum outro serviço externo, como o **Visual Review** para testes de revisão visual. Esta função é executada somente uma vez e antes da função `onPrepare` . Veja dois exemplos.

- **Teste na nuvem contra ambiente local de desenvolvimento:** nesse exemplo a configuração `beforeLaunch` inicializa um túnel com o serviço de testes na nuvem **BrowserStack** antes de qualquer outra configuração de ambiente.

```
beforeLaunch: function() {
  return new Promise(function(resolve, reject) {
    var browserstackLocalArgs = {
      'key': process.env.BROWSERSTACK_ACCESS_KEY,
```

```
      'force': 'true',
      'forcelocal': 'true'
    };
    browserstackLocal.start(browserstackLocalArgs, function(error)
 {
      if (error) {
        reject(error);
        return;
      }
      resolve();
    });
  });
}
```

- **Testes de revisão visual:** nesse exemplo, o **Visual Review** é inicializado possibilitando a revisão visual posterior a execução dos testes.

```
beforeLaunch: function () {
  vr.initRun('Visual-Review-Sample-Project', 'visualReviewSuite');
}
```

12.6 ANTES DA EXECUÇÃO DOS TESTES: ONPREPARE

A função `onPrepare` é executada assim que o **Protractor** está pronto e disponível. Ela é executada antes do início da execução dos testes.

Um uso simples para tal configuração pode ser, por exemplo, a maximização do navegador antes do início dos testes. Veja:

```
onPrepare: function () {
    browser.driver.manage().window().maximize()
}
```

Outros exemplos do uso da função `onPrepare` foram vistos no capítulo *5. Node modules úteis*, para a definição de diferentes *reports* de testes. Veja:

- `jasmine-spec-reporter` : nesse exemplo, a função

`onPrepare` é definida para a utilização do node module `jasmine-spec-reporter` para um melhor feedback após a execução dos testes no console.

```
onPrepare: function() {
  jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
  }));
}
```

- `protractor-jasmine2-html-reporter` : nesse exemplo, a função `onPrepare` é definida para a utilização do node module `protractor-jasmine2-html-reporter` , possibilitando um relatório de testes em formato HTML com screenshots de cada teste executado, podendo ser usado como evidência de testes.

```
onPrepare: function() {
  jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
  }));

  jasmine.getEnv().addReporter(new Jasmine2HtmlReporter({
    takeScreenshots: true,
    fixedScreenshotName: true
  }));
}
```

12.7 ASSIM QUE OS TESTES SÃO FINALIZADOS: ONCOMPLETE

A função `onComplete` é executada assim que os testes são finalizados. Tal função pode ser usada, por exemplo, para finalizar o

túnel com um serviço de testes na nuvem, conforme exemplo:

```
onComplete: function() {
  return new Promise(function(resolve, reject) {
    browserstackLocal.stop(function(error) {
      if (error) {
        reject(error);
        return;
      }
      resolve();
    });
  });
}
```

Neste exemplo, o túnel com o serviço **BrowserStack** é finalizado após a execução dos teste para evitar que outros sejam disparados e executados contra o ambiente local de desenvolvimento. Tal configuração é útil quando usamos processos de integração contínua.

12.8 APÓS A EXECUÇÃO DOS TESTES: AFTERLAUNCH

A função `afterLaunch` é executada após a execução dos testes e quando o *webdriver* já está desligado. Essa função passa como parâmetro um `exitCode` que possui valor 0 quando os testes passam.

Um exemplo da utilização de tal função, por exemplo, é a limpeza de arquivos temporários gerados pelo **Visual Review** após a execução de testes e2e usando tal integração. Veja:

```
afterLaunch: function (exitCode) {
  return vr.cleanup(exitCode);
}
```

Para a lista completa das configurações disponíveis, consulte a documentação oficial do **Protractor**, em:

https://github.com/angular/protractor/blob/master/docs/referenceConf.js.

Com isso, é possível utilizarmos recursos mais avançados relacionados a configurações para testes e2e automatizados, visando a escrita de testes mais robustos, e que atendem diversas necessidades. Temos como exemplo a escolha de um framework base que o time já esteja acostumado; a execução de teste em ambiente local sem a necessidade de um servidor do Selenium; a execução de testes em paralelo para poupar tempo; a definição de suítes de teste, tal como uma suíte de smoke teste, ou uma de testes para mobile; e funções que são executadas antes ou depois da execução dos testes, servindo como *setup* ou *cleanup* dos testes.

No próximo capítulo, veremos um assunto interessante: o processo criativo na escrita de testes e2e automatizados.

Capítulo 13

PROCESSO CRIATIVO EM TESTE DE SOFTWARE

Quando se fala em processo criativo em teste de software no contexto de testes e2e, a ideia é responder algumas das seguintes perguntas:

- Quais cenários/casos de teste criar?
- Como evoluir a suíte de testes?
- Como organizar o projeto de testes para ajudar em sua manutenção?
- Como evoluir os testes?

Neste capítulo, são apresentadas algumas ideias para responder estas perguntas.

13.1 DEFININDO OS CASOS DE TESTE

Ao iniciar a escrita de testes e2e automatizados, é importante garantir que as principais funcionalidades da aplicação estejam funcionando. Portanto, uma boa ideia quando definimos os casos de teste é começar pelos cenários de caminho feliz. Ou seja, os cenários que o usuário vai percorrer para realizar um procedimento com sucesso na aplicação em teste, por exemplo, um login com sucesso, a adição com sucesso de um produto no carrinho de compras, ou a criação com sucesso de uma conta de usuário.

Além disso, é interessante, antes de implementar a lógica dos casos de teste, escrever ao menos a descrição deles. Veja alguns exemplos:

```
describe('Test system', function() {
  it('successful account creation', function() {});

  it('successfully login', function() {});

  it('add product to the cart', function() {});
});
```

Perceba que os testes ainda não têm nenhuma implementação, porém, tal técnica ajuda a estruturar a suíte de teste para posterior implementação e melhoria. Além disso, tal visualização ajuda na criação de novos casos de teste e eliminação de testes redundantes ou desnecessários.

Após pensar nos cenários de caminho feliz, fica até mais fácil pensar nos cenários alternativos. Veja alguns exemplos:

```
describe('Test system', function() {
  it('try to create account without filling all mandatory fields',
 function() {});

  it('try to create account with password that does not match', fu
nction() {});

  it('try to create account using password that does not respect t
he rules', function() {});

  it('try to login with invalid user', function() {});

  it('try to login without filling any field', function() {});
});
```

Com uma ideia dos cenários de teste, é possível começar a evoluir a suíte e partir para a implementação.

13.2 EVOLUINDO A SUÍTE DE TESTE

Uma boa tática para começar é seguir a técnica *red, green,*

refactoring, do TDD (*test-driven development*). Ou seja, primeiro se cria um teste que falha, após se implementa o mínimo para que ele passe, e então se refatora o teste para torná-lo robusto.

Veja um exemplo:

```
describe('Test system', function() {
  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    element(by.id('element-sign-in-username')).sendKeys('validuser
');
    element(by.id('element-sign-in-password')).sendKeys('validpass
word');
    element(by.id('element-sign-in-submit')).click();

    expect(element(by.css(.authenticated)).isPresent()).not.toBe(t
rue);
  });
});
```

No caso do usuário e senha utilizados nesse teste serem realmente válidos, tal teste deve falhar, pois o usuário deve estar autenticado e a aplicação deve possuir um elemento com a classe CSS `'authenticated'`. Entretanto, o teste espera que tal elemento não esteja presente.

É bom lembrar de que é sempre bom fazer o teste falhar, para garantir que, quando este passa, não é um falso positivo. Ou seja, um teste que sempre passa, mesmo quando devia estar falhando.

Após, segundo a técnica *red, green, refactoring*, deve-se fazer o mínimo para o teste passar. Veja:

```
describe('Test system', function() {
  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    element(by.id('element-sign-in-username')).sendKeys('validuser
');
    element(by.id('element-sign-in-password')).sendKeys('validpass
word');
    element(by.id('element-sign-in-submit')).click();
```

```
    expect(element(by.css(.authenticated)).isPresent()).toBe(true)
;
  });
});
```

Com o teste passando, é então possível refatorá-lo para torná-lo mais robusto. Veja uma alternativa:

```
describe('Test system', function() {
  function login(user, password) {
    element(by.id('element-sign-in-username')).sendKeys(user);
    element(by.id('element-sign-in-password')).sendKeys(password);
    element(by.id('element-sign-in-submit')).click();
  }

  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    login('validuser', 'validpassword');

    expect(element(by.css(.authenticated)).isPresent()).toBe(true)
;
  });
});
```

Perceba que uma função chamada `login` foi criada. Ela recebe como parâmetro usuário e senha, preenche os campos necessários com estes valores e clica no botão de login.

Com esta simples função, o que antes eram três passos no caso de teste agora é apenas um. Além disso, tal função pode ser reutilizada para outros cenários de login, tal como a tentativa de login sem o preenchimento de usuário e senha. Veja:

```
it('try to login without filling any field', function() {
  browser.get('http://choko.org/sign-in');

  login('', '');

  expect(element(by.css(.authenticated)).isPresent()).not.toBe(tru
e);
});
```

Mas ainda dá para ficar melhor. Os testes demonstrados até aqui

estão misturando a definição de elementos e funções e os passos dos casos de teste propriamente ditos. Como já foi visto em outros capítulos, uma forma organizar os testes para facilitar sua manutenção é o uso do padrão *Page Objects*.

13.3 ORGANIZANDO O PROJETO DE TESTES PARA MANUTENÇÃO EVOLUTIVA

Organizar os testes é uma tarefa primordial para ajudar em sua manutenção, como também para torná-los mais legíveis. A utilização do padrão *Page Objects* ajuda exatamente neste sentido, separando definição de elementos e funções em arquivos específicos que podem ser alterados uma só vez. Assim, todos os testes que os utilizam passam a desfrutar de tais mudanças, deixando os testes em si com uma linguagem mais próxima a uma linguagem de negócio, em que os passos necessários para a execução dos casos de teste é definida, ajudando na legibilidade.

Veja o teste de login já demonstrado, porém, agora refatorado para a utilização do padrão *Page Objects*:

```
var LoginPage = require('../page-objects/loginPage.po.js');

describe('Test system', function() {
  var loginPage = new LoginPage();

  it('successful account creation', function() {
    loginPage.visit();

    loginPage.login('validuser', 'validpassword');

    expect(loginPage.isLoggedIn.isPresent()).toBe(true);
  });
});
```

Veja como a leitura do teste fica muito mais legível e próxima de uma linguagem de negócio, na qual não existem informações desnecessárias, como IDs de elementos etc. A seguir, é demonstrado

o *Page Object* em questão:

```
var LoginPage = function() {
  this.usernameField = element(by.id('element-sign-in-username'));
  this.passwordField = element(by.id('element-sign-in-password'));
  this.loginButton = element(by.id('element-sign-in-submit'));
};

LoginPage.prototype.login = function(user, password) {
  this.usernameField.sendKeys(user);
  this.passwordField.sendKeys(password);
  this.loginButton.click();
};

LoginPage.prototype.visit = function() {
  browser.get('http://choko.org/sign-in');
};

module.exports = LoginPage;
```

Já *Page Object* define os elementos da página de teste em questão e funções para ações que necessitam de mais de um passo, como o login, ou mesmo funções que simplesmente ajudam na legibilidade, como a função `visit`, que visita a página em questão.

Assim a suíte de teste vai ficando mais organizada e tomando mais forma. Mas não paramos por aí.

13.4 EVOLUINDO AINDA MAIS

Softwares sempre podem ser melhorados. Uma vez que a suíte de teste começa a crescer, sua arquitetura deve evoluir.

Na primeira seção deste capítulo, foram demonstrados cenários de teste para 3 funcionalidades diferentes da mesma aplicação (criação de conta, login e carrinho de compras). Uma forma de evoluir tais testes para uma melhor organização, por exemplo, é separá-los em diferentes arquivos de teste. Estes propõem a testar diferentes funcionalidades, ou seja, os testes passam a ser separados dependendo de seus contextos.

Veja um exemplo da evolução de tais testes.

- **Teste de criação de conta de usuário**

```
describe('Account creation', function() {
  it('successful account creation', function() {});

  it('try to create account without filling all mandatory fields',
 function() {});

  it('try to create account with password that does not match', fu
nction() {});

  it('try to create account using password that does not respect t
he rules', function() {});
});
```

- **Teste de login**

```
describe('Login', function() {
  it('successfully login', function() {});

  it('try to login with invalid user', function() {});

  it('try to login without filling any field', function() {});
});
```

- **Teste de carrinho de compras**

```
describe('Shopping cart', function() {
  it('add product to the cart', function() {});
});
```

Eles podem possuir seus *Page Objects* específicos e genéricos. Obviamente, há espaço para outras melhorias que podem depender de diferentes contextos –como a organização de *Page Objects* por funcionalidades em subdiretórios (para aplicações muito grandes), criação de helpers para questões bastante genéricas ao longo de toda aplicação etc. Porém, a ideia aqui foi ao menos mostrar um pouco do processo criativo que envolve a criação de testes e2e.

Sugestão de leituras complementares

No blog **Talking About Testing**, escrevi dois posts que são relacionados ao assunto deste capítulo. Deixo-os como leitura complementar. Espero que ajude! Consulte as seguintes URLs:

- https://talkingabouttesting.com/2016/05/09/processo-criativo-em-teste-de-software/
- https://talkingabouttesting.com/2016/06/27/testes-de-aceitacao-automatizados-por-onde-comecar/

No próximo capítulo, serão apresentadas algumas dicas úteis relacionadas a testes e2e automatizados.

CAPÍTULO 14

DICAS ÚTEIS

Este capítulo reúne uma série de dicas úteis para utilização do framework **Protractor**, a fim de facilitar o desenvolvimento de software unido aos benefícios da utilização de testes automatizados.

Algumas dessas dicas são: um gerador de estrutura de testes, facilidades do **Jasmine**, questões relacionadas a depuração de testes, testes para aplicações não AngularJS, dicas para demonstrações de aplicações, e como sobrescrever configurações via linha de comando.

14.1 GERADOR DE ESTRUTURA DE TESTES

A cada novo projeto web, alguns passos precisam ser repetidos para a utilização do Protractor como framework de testes e2e. Esses passos são: a definição do arquivo de configurações e algumas de suas configurações (como a `baseUrl` e o navegador no qual os testes vão rodar), o arquivo para o gerenciamento de dependências dos testes e o diretório onde os testes propriamente ditos serão armazenados.

Para tornar fácil a configuração inicial da estrutura de testes com o **Protractor**, é possível utilizar o *generator-protractor*, explicado a seguir. O *generator-protractor* é um gerador de código para utilização do **Protractor** e pode ser instalado da seguinte forma:

```
npm install -g yo
npm install -g generator-protractor
```

Após instalado, ele já pode ser utilizado.

Dentro do diretório do projeto que será testado com o **Protractor**, crie um diretório chamado `tests` e, dentro deste, um diretório chamado `e2e` . Acesse o diretório `e2e` recém-criado a partir da linha de comando:

```
cd tests/e2e
```

Execute o seguinte comando:

```
yo protractor
```

Algumas perguntas serão feitas, as quais podem ser respondidas com os valores padrões sugeridos, ou então conforme sua escolha. Seguem as perguntas com sugestão dos valores padrões:

```
Welcome to the protractor code generator.
? Choose a name for the protractor configuration file (protractor.
conf.js)
? Choose a base URL (http://localhost:8000)
? Which browsers do you want to run? (Use arrow keys)
❯ Chrome
  Firefox
  Both, at the same time
```

Após a resposta das perguntas, o básico para começar é gerado, inclusive com um arquivo de testes de exemplo. Então, é só ir customizando o resto conforme as necessidades específicas do projeto. Uma sugestão adicional é a criação do diretório `page-objects` , visto como uma boa prática para ajudar na manutenabilidade e legibilidade dos testes.

Mais detalhes sobre o *generator-protractor* podem ser encontradas na seguinte URL: https://www.npmjs.com/package/generator-protractor.

14.2 FACILIDADES DO JASMINE

Quando criando testes automatizados, às vezes precisamos rodar somente um novo teste recém-criado, sem a necessidade de testar todo o resto. Ou mesmo, podemos precisar rodar somente um arquivo de testes específico, sem a necessidade de rodar todos os outros. Também há casos nos quais é útil deixar de rodar um teste específico sem afetar o restante dos outros, quando tal teste está falhando e precisando de manutenção, visto que este não está robusto o suficiente e causando resultados falsos negativos, por exemplo.

O framework base **Jasmine**, usado como padrão pelo Protractor, dispõe de algumas facilidades para resolver estas questões, que serão abordadas a seguir.

Executando um só caso de teste

Vejamos o seguinte exemplo de uma suíte de teste, onde no primeiro caso de teste, uma página é visitada, um item é adicionado a uma lista vazia e então é feita uma verificação de que a lista possui um item, e no segundo caso de teste um novo item é adicionado à lista e é feita uma verificação de que o texto do item recém-adicionado está contido na lista:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
```

```
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

Caso haja a necessidade de executar somente um dos casos de teste (o segundo, por exemplo), basta modificar a definição do `it` para `fit` . Assim, quando o Protractor for executado, somente o teste `fit` será executado, e qualquer outro será desconsiderado. Veja:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  fit('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

Visto que o segundo caso de teste está definido como `fit` , somente este é executado.

Pulando um caso de teste específico

O contrário também é possível. Digamos que você queira pular um caso de teste específico para manutenção posterior para torná-lo mais confiável, como um teste que às vezes falha e às vezes passa, mesmo sem alteração alguma na aplicação. Tal objetivo é atingido utilizando outra facilidade do **Jasmine**, substituindo o `it` por `xit`. Veja:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  xit('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

No mesmo exemplo, agora somente o primeiro teste é executado, pois o teste com a definição `xit` é desconsiderado na hora da execução. Além disso, quando um teste é pulado, é possível adicionar uma explicação de por que tal teste está sendo pulado. Ela fica disponível após a execução dos testes, na lista de testes pendentes (testes definidos com `xit`). Veja:

```
xit('add new item in the todo list', function() {
  todoMvc.visit();

  todoMvc.addItemOnTodoList('Create new test without page object')
;
```

```
  expect(todoMvc.listOfItems.count()).toEqual(2);
}).pend('This test needs refactoring, because it is not independen
t');
```

Perceba que o teste possui um `.pend();` no final, recebendo uma string exibida no resultado da execução dos testes, na lista de testes pendentes. Veja um exemplo:

```
$ protractor
[18:49:48] I/direct - Using ChromeDriver directly...
[18:49:48] I/launcher - Running 1 instances of WebDriver
Started
.*.

Pending:

1) angularjs homepage todo list should list todos
  Needs refactoring

3 specs, 0 failures, 1 pending spec
Finished in 5.761 seconds
[18:49:55] I/launcher - 0 instance(s) of WebDriver still running
[18:49:55] I/launcher - chrome #01 passed
```

Executando somente um describe

Outra facilidade disponível pela utilização do **Jasmine** é a execução de somente um `describe` . Tal opção pode ser útil quando existem diversos `describes` para diferentes funcionalidades da aplicação, e precisam-se executar somente os testes de uma determinada funcionalidade.

Um exemplo real é quando você acabou de criar um novo arquivo de testes, que você deseja testar, mas sem a necessidade de perder tempo executando todos os outros. Tal necessidade é atingida da seguinte maneira:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

fdescribe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();
```

```
  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

Perceba que a definição do `describe` é modificada por `fdescribe`. Ao definir um `describe` como `fdescribe`, quando o Protractor é executado, ele executa somente os `fdescribe` e desconsidera todos os outros **describes**.

14.3 DEPURANDO TESTES

Muitas vezes quando escrevemos testes ou os executamos, é necessário de alguma forma depurá-los, principalmente quando eles estão falhando e não sabemos exatamente qual a causa. O Protractor dispõe da possibilidade de pausar um teste durante sua execução para facilitar na depuração.

Vejamos como pausar um teste durante sua execução, conforme o exemplo:

```
it('add new item in the todo list', function() {
  var text = 'Create new test without page object';

  todoMvc.visit();

  todoMvc.addItemOnTodoList(text);
  browser.pause();
```

```
  expect(todoMvc.listOfItems.getText()).toContain(text);
});
```

Veja que há um código `browser.pause();` após o passo em que um item é adicionado na lista de TODOs. Quando o Protractor executa os testes e encontra o código de pausa, o seguinte é exibido no console e o navegador fica aberto exatamente neste ponto:

```
[19:00:09] I/protractor - Encountered browser.pause(). Attaching d
ebugger...
[19:00:09] I/protractor -
[19:00:09] I/protractor - ------- WebDriver Debugger -------
[19:00:09] I/protractor - Starting WebDriver debugger in a child p
rocess. Pause is still beta, please report issues at github.com/an
gular/protractor
[19:00:09] I/protractor -
[19:00:09] I/protractor - press c to continue to the next webdrive
r command
[19:00:09] I/protractor - press ^D to detach debugger and resume c
ode execution
[19:00:09] I/protractor - type "repl" to enter interactive mode
[19:00:09] I/protractor - type "exit" to break out of interactive
mode
```

Com isso, é possível analisar a aplicação onde o navegador parou para entender algum problema específico, como por que um elemento não está sendo usado, por exemplo. Neste caso, pode-se descobrir que o seletor utilizado para a escolha do elemento com o qual se desejava interagir não era adequado, visto que mais de um elemento possui o mesmo seletor, digamos.

14.4 TESTANDO APLICAÇÕES NÃO ANGULARJS

Apesar de o Protractor ser o framework oficial para a criação e execução de testes e2e para aplicações AngularJS, com apenas uma linha de código é possível usá-lo para a criação e execução de testes para qualquer tipo de aplicação web.

Visto que criar testes com o Protractor é fácil, projetos não AngularJS também podem utilizá-lo. Neles todos os localizadores, exceto os específicos de aplicações AngularJS (como `by.binding`, `by.model` e `by.repeater`), estarão disponíveis.

Veja a alteração necessária no arquivo de configurações do **Protractor** para sua utilização para testar aplicações não AngularJS:

```
onPrepare() {
  browser.ignoreSynchronization = true;
},
```

Conforme visto no capítulo *12. Configurações avançadas*, diversas configurações podem ser feitas no momento da preparação dos testes, antes de sua execução. Uma delas é a configuração `browser.ignoreSynchronization = true;`.

Por padrão, o valor desta configuração é `false`. Isso significa que sempre que o Protractor for executado, ele vai entender que a aplicação em teste é uma aplicação AngularJS.

Ao setar tal configuração com valor `true`, no momento da execução dos testes, o Protractor não procurará por questões específicas do AngularJS, habilitando-o para criação e execução de testes de qualquer tipo de aplicação web, como Ember, Drupal etc.

14.5 DICAS PARA DEMONSTRAÇÕES

Testes automatizados geralmente são executados de forma muito rápida quando comparados a um humano executando os mesmos casos de teste. Este é um dos benefícios de sua utilização. Porém, testes automatizados podem ser usados para outras questões, como documentação sobre o comportamento da aplicação em diferentes situações, ou mesmo para demonstração de funcionalidades.

Digamos que estamos desenvolvendo uma aplicação para um cliente externo e, a cada final de semana (sextas-feiras), as novas funcionalidades são demonstradas a tal cliente. Uma forma de demonstrar tal funcionalidade ao cliente pode ser a execução dos testes e2e, visto que eles são exatamente criados pensando no uso da aplicação da perspectiva do usuário final. Entretanto, visto que tais testes são executados tão rápido, pode ser difícil de o cliente entender o que cada testes está realmente fazendo.

O Protractor dispõe da funcionalidade de **dormir** (*sleep*), que define um tempo de espera entre um passo e outro durante a execução dos testes. A utilização de *sleeps* não é recomendada para a execução dos testes na sua forma natural, pois torna-os mais lentos e nem sempre confiáveis. Porém, para demonstrações, tal funcionalidade pode ser muito útil.

Veja a seguir um exemplo de um caso de teste, no qual um tempo de espera de 5 segundos é adicionado entre cada passo do teste, para que o cliente possa vê-lo sendo executado de forma mais "pausada", e possa entender o comportamento da aplicação passo a passo:

```
it('add new item in the todo list', function() {
  var text = 'Create new test without page object';

  todoMvc.visit();
  browser.sleep(5000);

  todoMvc.addItemOnTodoList(text);
  browser.sleep(5000);

  expect(todoMvc.listOfItems.getText()).toContain(text);
});
```

A função `sleep` recebe como argumento um valor em milissegundos, e o próximo passo do teste é executado quando tal espera é atingida.

Observação

Lembrando de que o uso de *sleep times* não é recomendado a não ser em questões relacionadas a demonstrações do uso da aplicação para clientes. Ou seja, evite utilizar *sleep times* nos testes em qualquer outra situação.

14.6 SOBRESCREVENDO CONFIGURAÇÕES VIA LINHA DE COMANDO

Em alguns momentos, pode ser interessante sobrescrever as configurações definidas no arquivo de configurações do Protractor no momento de sua execução. Temos como exemplo: para executar o mesmo teste em um navegador diferente do configurado, para executar o mesmo teste contra uma URL base diferente, para executar um arquivo de teste específico, para rodar os testes utilizando o webdriver do próprio navegador, dentre outras.

Com o Protractor, é possível sobrescrever as configurações direto na linha de comando. Vejamos adiante.

Executando os testes em um navegador diferente do configurado

Digamos que o navegador definido no arquivo `protractor.conf.js` seja o Chrome. Porém, no momento da execução dos testes, é necessário rodar os mesmos testes contra o navegador Firefox.

No console, digamos que você já esteja no diretório onde se encontra o arquivo de configurações. Então, executa o seguinte comando:

```
protractor --browserName firefox
```

Ao passar ao `protractor` o argumento `--browserName firefox`, tal *capability* é sobrescrita no arquivo de configuração com o valor definido direto na linha de comando. Neste caso, a *capability* `browserName` recebe o valor `firefox`.

Executando os testes em uma URL base diferente

Da mesma forma, a URL base pode ser sobrescrita. Tal abordagem pode ser útil para a execução dos mesmos testes em diferentes ambientes de testes, como ambiente de QA, ambiente de UAT (*user acceptance testing*), ou mesmo em ambiente de produção. Veja:

```
protractor --baseUrl http://qa-sample.io
```

Com esse comando, a URL base é sobrescrita para executar os testes em ambiente de QA, por exemplo.

```
protractor --baseUrl http://uat-sample.io
```

Com este, a URL base é sobrescrita para executar os testes em ambiente de UAT.

```
protractor --baseUrl http://sample.io
```

Já com esse comando, a URL base é sobrescrita para executar os testes em ambiente de produção, por exemplo.

Executando somente os testes de um arquivo específico

Caso exista a necessidade da execução de um arquivo específico a partir do diretório de `specs`, por exemplo, basta executar o seguinte comando:

```
protractor --specs specs/specific-test.spec.js
```

Ao executá-lo, a configuração do array de specs é sobrescrita

pelo arquivo definido direto na linha de comando, e então somente tal arquivo é executado.

Executando os testes utilizando o webdriver do próprio navegador

Normalmente, o Protractor é configuradao para a utilização de um servidor do Selenium para a execução dos testes. Tal abordagem é bastante usada quando executamos testes em servidores de integração contínua, por exemplo.

Porém, em alguns momentos, o desenvolvedor (ou testador) pode querer executar os testes direto em seu computador. Caso ele utilize os navegadores Chrome ou Firefox, tais testes podem ser executados sem a necessidade de um servidor do Selenium rodando, já que, com o Protractor, é possível usar o webdriver destes próprios navegadores.

Para sobrescrever tal configuração, execute o seguinte comando:

```
protractor --directConnect true
```

Ao executar esse comando, mesmo que esteja na configuração no arquivo `protractor.conf.js` um `seleniumAddress` , tal valor será desconsiderado e o webdriver do próprio navegador será utilizado.

> Lembre-se de que só é possível utilizar o webdriver dos navegadores Chrome e Firefox.

Utilizar as dicas aqui mencionadas pode ajudar em diferentes situações quando escrevemos ou executamos testes e2e

automatizados. Portanto, espero que sejam úteis!

Capítulo 15

INDO ALÉM

Durante o aprendizado de testes e2e automatizados, vimos a importância do uso de boas práticas e do uso de padrões de projeto, como Page Objects, para ajudar na manutenabilidade e legibilidade dos testes. Também vimos o uso de *helpers*, com funções genéricas, para facilitar a escrita de testes. Além disso, conhecemos alguns node modules que podem ser usados para complementar a escrita dos testes, e como fazer ações e verificações durante os testes.

Também conhecemos diferentes técnicas que podem ser utilizadas em conjunto aos testes e2e, tais como testes de revisão visual, testes na nuvem, testes para *mobile* e como integrar algumas dessas rotinas no processo de integração contínua. Tudo isso ajudando a prover feedback rápido sempre que mudanças ocorrerem na aplicação em teste.

Por fim, foram apresentadas questões como a escritas de testes e2e usando ECMAScript 2015, algumas configurações avançadas e o processo criativo que envolve a escrita de testes automatizados. Por último, vimos algumas dicas para ajudar em diferentes momentos quando realizamos atividades relacionadas a teste de software.

Todo o conteúdo abordado durante o livro pode evoluir ainda mais com a prática. Portanto, minha sugestão é que você comece agora mesmo a implementar testes e2e automatizados em seus projetos de software pessoais, em seu trabalho, ou em trabalhos acadêmicos. Isso porque, como profissionais de desenvolvimento de

software, todos os dias temos de vislumbrar desenvolver softwares melhores, e a prática é o que levará a perfeição.

Walmyr Lima e Silva Filho

http://walmyr-filho.com