Sumário

[1. TESTES END-TO-END (E2E) 2](#_Toc14267983)

[1.1. Ferramentas necessárias: 2](#_Toc14267984)

[1.2. Preparando o ambiente 2](#_Toc14267985)

[1.2.1. Parte 1: Programas principais e estrutura de pastas 2](#_Toc14267986)

[1.2.2. Parte 2: Instalação/atualização e configuração 3](#_Toc14267987)

[1.2.3. Parte 3: **tsconfig.json** 4](#_Toc14267988)

[1.2.4. Parte 4: **conf.ts** 4](#_Toc14267989)

[1.2.5. Alguns scripts úteis 6](#_Toc14267990)

[1.3. Criando nossa primeira suíte de testes 7](#_Toc14267991)

[1.3.1. Escrevendo o teste: 7](#_Toc14267992)

[describe 8](#_Toc14267993)

[it 8](#_Toc14267994)

[expect 8](#_Toc14267995)

[browser 8](#_Toc14267996)

[element(by.[]) 9](#_Toc14267997)

[isPresent() 9](#_Toc14267998)

[1.3.2. Executando o teste 9](#_Toc14267999)

[SUGESTÃO: Performando a execução: 9](#_Toc14268000)

[1.3.3. Adicionando mais cenários: 10](#_Toc14268001)

[1.4. Boas práticas em escrita de códigos de testes 12](#_Toc14268002)

[1.4.1. Parte 1: Refatorando o código 12](#_Toc14268003)

[Limpando os specs e adicionando páginas 12](#_Toc14268004)

[Criando funções genéricas - Parte 1 15](#_Toc14268005)

[Criando funções genéricas – Parte 2 16](#_Toc14268006)

[1.4.2. Utilizando parâmetros 19](#_Toc14268007)

# TESTES END-TO-END (E2E)

Realização de testes de aceitação automatizados passo a passo utilizando Typescript + Protractor + Jasmine.

## Ferramentas necessárias:

Para o desenvolvimento dos testes e2e, serão utilizadas as seguintes ferramentas:

* Interpretador de código: [NodeJS](https://nodejs.org/en/)
* Editor de texto/IDE de desenvolvimento: [Visual Studio Code](https://code.visualstudio.com/);
* Linguagem de programação: [Typescript](https://www.typescriptlang.org/);
* Frameworks de testes: [Protractor](https://www.protractortest.org/) / [Jasmine](https://jasmine.github.io/) / [Selenium](https://www.seleniumhq.org/)
* Framework de geração de relatórios: [Allure Reports](http://allure.qatools.ru/)
* Navegador: [Chrome](https://www.google.com/intl/pt-BR/chrome/)

## Preparando o ambiente

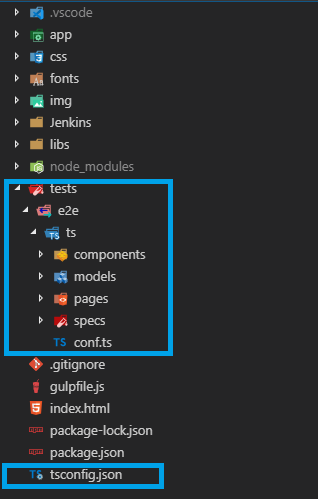
### Parte 1: Programas principais e estrutura de pastas

Para preparar o ambiente, siga as etapas abaixo:

1. Instalação do [NodeJS](https://nodejs.org/en/): a instalação do Node vai permitir você interpretar os códigos em Javascript e executá-los a partir do comando **node**. Também com a instalação do NodeJS, o comando permitirá acesso ao gerenciador de pacotes do node com o comando [npm](https://www.npmjs.com/).
2. Instalação do [Visual Studio Code](https://code.visualstudio.com/): Basta baixar o executável e seguir as etapas de instalação.
3. Determinar a estrutura do projeto de testes: Para os testes e2e, a estrutura de diretórios ficará da seguinte forma:

O projeto acima representa uma estrutura de um projeto front-end, por isso abrigaremos nossos testes e2e onde tiverem as estruturas de front-end.

Na raiz, teremos um arquivo de configuração geral para todo o código dentro desse projeto desenvolvido em Typescript, que é o **tsconfig.json**!



O diretório que utilizaremos será o “tests” é o responsável por abrigar todo o tipo de teste acoplado àquela aplicação.

O subdiretório indica o tipo específico do teste que está sendo utilizado. Para o nosso caso, este será chamado de “e2e”.

Os diretórios seguintes para testes e2e vão seguir o seguinte padrão: components, models, pages e specs. Seguido destes, teremos um arquivo de configuração, mas apenas para os testes automatizados, que é o **conf.ts**.

Obs.: em algumas outras referências, pode-se encontrar esse arquivo como *protractor.conf.ts*, mas será adotado o acima.

Para criar as estruturas acima, vá para a pasta raiz do projeto e basta reproduzir os seguintes comandos:

|  |
| --- |
| > cd diretorio\_do\_projeto  > touch ./tsconfig.json  > mkdir tests/ tests/e2e tests/e2e /ts/ tests/e2e/ts/componentes tests/e2e/ts/models tests/e2e/ts/pages tests/e2e/ts/specs  > cd tests/e2e/ts  > touch conf.ts |

### Parte 2: Instalação/atualização e configuração

Para seguir, basta realizar os seguintes comandos:

|  |
| --- |
| > npm install -g typescript |

Ao instalar o typescript, ele já vem com o Selenium, que é um gerenciador de drivers para nosso teste. No caso, o Selenium irá gerenciar nosso navegador utilizado para testes. Para configurar nosso driver, basta executar o seguinte comando:

|  |
| --- |
| > webdriver-manager update –-standalone=false –-gecko=false |

Para os pacotes restantes, os comandos abaixo irão baixar os repositórios com a versão mais atualizada de cada pacote.

|  |
| --- |
| > npm install --save-dev protractor jasmine jasmine-allure2-reporter protractor-jasmine2-html-reporter jasmine-spec-reporter allure-commandline allure2-js-commons  > npm install --save-dev @types/protractor @types/jasmine @types/jasminewd2 @types/node |

### Parte 3: **tsconfig.json**

Sugiro começar com um arquivo semelhante ao modelo abaixo. Segue um exemplo de arquivo tsconfig.json com algumas das principais opções comentadas.

{

"compilerOptions": {

"target": "es6", /\*Especifica qual versão alvo do ECMAScript para compilação\*/

"module": "commonjs", /\*Especifica geração de código do módulo\*/

"moduleResolution": "node", /\*Especifica estratégia de resolução do código (no caso, utilizando o node)\*/

"sourceMap": false, /\*Gera o arquivo .map\*/

"declaration": false, /\*Gera um sourcemap para cada arquivo .d.ts correspondente\*/

"removeComments": true, /\*Códigos compilados não carregam os comentários dos arquivos .ts\*/

"noImplicitAny": true, /\*Tipo any não pode ficar implícito na declaração de qualquer coisa\*/

"noEmitOnError": true, /\*Não gerará arquivo .js para o arquivo .ts que tiver erro de compilação\*/

"outDir": "tests/e2e/js", /\*Diretório destino do\*/

"strictNullChecks": true,

"experimentalDecorators": true,

"noImplicitThis": true, /\* Gera erro ao ter uma expressão this com tipo any implícito\*/

"types": [

"jasmine",

"jasminewd2",

"node"

], /\*Arquivos de declaração do tipo @type também serão compilados\*/

"lib": [

"dom",

"es2018"

] /\*Especifica arquivos de bibliotecas para incluir na compilação\*/

},

"exclude": [

"node\_modules"

], /\*Exclui os diretórios especificados na compilação\*/

"include": [

"tests/e2e/ts/\*\*/\*"

] /\*Inclui os diretórios especificados na compilação\*/

}

### Parte 4: **conf.ts**

As configurações abaixo criam as condições necessárias para execução dos testes:

* *config:* essa variável carrega várias propriedades de configuração dos nossos frameworks.

1. *seleniumAdress:* configura endereço default para testes utilizando selenium webdriver;
2. *framework:* Especifica o framework que realizará os testes;
3. *suites:* aqui é onde serão feitas as referências das specs que serão executadas ao rodarem os nossos testes. OBSERVAÇÃO IMPORTANTE: Note que, apesar de escrevermos nossos códigos em TS, as referências são feitas aos arquivos .js, pois eles serão interpretados pelo Protractor/Jasmine.
4. *baseURL:* Trata da URL que o teste buscará por padrão quando for inicializado. Se não tiver um **browser.get** na sequência, os testes começarão no site que estiver referenciado nessa propriedade.
5. *params:* Propriedade muito útil para utilizar valores que serão reaproveitados durante todo o teste ou que podem ser modificados (via linha de comando), porém inicialmente representam um valor default caso nenhum parâmetro seja passado na linha de comando.
6. *[multi]Capabilities:* estabelece propriedades de qual navegador será utilizado para os testes e como se comportará. O termo “capabilities” pode ser usado sem o “multi”, pois quando o Multi é usado é quando queremos configurar mais de um navegador diferente para o teste.
7. *jasmineNodeOpts:* estabelece alguns parâmetros de execução para o Jasmine.
8. *onPrepare:* estabelece regras importantes para nosso relatório de teste, tanto no exibido no console quanto para o relatório estilizado que será gerado posteriormente com o allure reports.

*./tests/e2e/ts/conf.ts*

import { Config } from 'protractor';

import { SpecReporter } from 'jasmine-spec-reporter';

export const config: Config = {

seleniumAddress: 'http://localhost:4444/wd/hub',

framework: 'jasmine',

directConnect: true,

suites: {

e2e: [

'./specs/cenario01.spec.js',

'./specs/cenario02.spec.js'

],

smoke: []

},

// Define a URL base da aplicação

baseURL: '',

// Define alguns parâmetros comuns aos testes para evitar repetições desnecessárias

params: {

p1: {

subp1: "test1”,

subp2: "test2”

}

},

multiCapabilities: [

{

'browserName': 'chrome',

'chromeOptions': {

// --start-maximized: inicia o chrome maximizado.

args: [

//'--no-sandbox',

'--test-type=browser',

'--start-maximized'

],

prefs: {

/\*'download': {

'prompt\_for\_download': false,

'directory\_upgrade': true,

'default\_directory': downloadsPath

},\*/ 'profile.managed\_default\_content\_settings.notifications': 1

}

},

}

],

jasmineNodeOpts: {

defaultTimeoutInterval: 120000,

showColors: true,

realTimeFailure: true,

print: function () {

}

},

onPrepare: () => {

jasmine.getEnv().addReporter(new SpecReporter({

spec: {

displayStacktrace: true,

displayErrorMessages: true,

displayFailed: true,

displayDuration: true

},

summary: {

displayStacktrace: false,

displayErrorMessages: true,

displaySuccessful: true,

displayFailed: true,

displayDuration: true

},

colors: {

enabled: true

}

}));

}

};

### Alguns scripts úteis

Tudo configurado, mas antes, para facilitar no desenvolvimento do código e execução dos testes, vou dar umas sugestões de scripts para colocar no package.json:

*./package.json*

"scripts": {

"compile": "tsc",

"watch": "tsc -w",

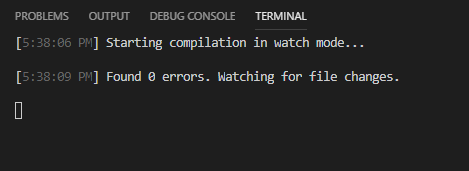
"start": "npm run watch"

},

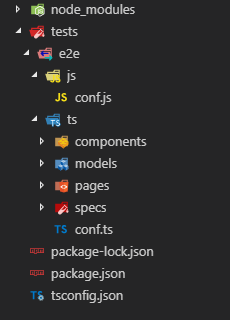
Posteriormente novos commandos serão incluídos, mas esses já nos bastam por hora.

* compile: ao digitar **npm run compile**, o compilador typescript vai compilar o código presente baseado nas regras definidas no tsconfig.json.
* watch: recurso muito interessante e útil, pois o watch atualiza e recompila o código a cada alteração feita nele, inclusive apontando os erros de compilação em tempo real
* start: nativo do npm, apenas foi sintetizado o watch nele. Basta executar **npm start**.

Até aqui, ao executar o npm start nosso código deve estar compilando sem mais problemas:



Perceba que foi gerada um diretório tests/e2e/js, que possui os arquivos .ts compilados. Como só foi criado o conf.ts, apenas ele foi compilado. Lembra da configuração do outDir do tsconfig.json? Então, os compilados foram parar lá.

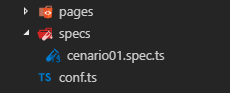


## Criando nossa primeira suíte de testes

### Escrevendo o teste:

Após tudo configurado até essa etapa, sugiro continuar desenvolvendo com um console rodando o npm start configurado anteriormente.

Dentro do diretório specs, crie um novo arquivo chamado cenario01.spec.ts.



Logo em seguida, faça o mapeamento desse cenário no nosso conf.ts (lembrando de fazer a referência ao arquivo .js gerado após a compilação).

*./tests/e2e/ts/conf.ts*

suites: {

e2e: [

'./cenarios/cenario01.spec.js',

]

}

Se estiver no VS Code, para começar a escrever trechos de validação, basta digitar “describe” que aparecerá para auto completar já com a sintaxe do Jasmine. Pode escolher tanto a opção t-describe como a t-describe-it.

Para esse exemplo vamos acessar o site de exemplo <http://juliemr.github.io/protractor-demo/>.

Coloque o código abaixo em cenario01.spec.ts:

*./tests/e2e/ts/specs/cenario01.spec.ts*

import { browser, element, by } from "protractor";

describe(' Teste cenário 01', () => {

it('Verifica se o primeiro campo da soma existe na página', () => {

browser.get('http://juliemr.github.io/protractor-demo/');

var primeiraParcela = element(by.model('first'));

expect(primeiraParcela.isPresent()).toBe(true);

});

});

Esse código simplesmente vai unir a sintaxe do Jasmine com a do Protractor e compor um teste de validação simples de um elemento de página. Ao inserir e salvar, como o “watch” está rodando, esse código deve compilar sem problemas e já deve gerar um novo arquivo .js compilado: *./tests/e2e/js/specs/cenario01.spec.js*

Explicando o código:

* Sintaxe do Jasmine

#### describe

Ele cria a descrição base daquele caso de teste. Um arquivo pode ter quantos describes quiser, mas a ideia é que ele seja uma descrição principal e macro do teste/cenário que está sendo testado.

#### it

O it é responsável por descrever um “sub-cenário” ou uma descrição mais detalhada de uma funcionalidade específica que estamos testando dentro do cenário. Pequenas validações podem ser feitas para testar um cenário macro do describe, como validação de elementos, validação de operações etc.

É IMPORTANTÍSSIMO TER O it MUITO BEM DETALHADO, POIS ELE GUIA A LEITURA DE EXECUÇÃO DE TESTES DOS NOSSOS CENÁRIOS!

#### expect

É o trecho que cria os critérios de validação dos testes. A partir de uma promise passada, você pode validar se as condições seguidas no teste serão satisfeitas. Ex.: é possível validar se um elemento está presente (como no exemplo acima), validar URL, validar número de elementos, validar outros elementos de tela existentes quando ativos/inativos etc.

Esses são os mais básicos. Para mais detalhes e outros recursos, veja a documentação do [Jasmine](https://jasmine.github.io/tutorials/your_first_suite).

* Sintaxe do Protractor

#### browser

O browser trabalha em conjunto com o Selenium Webdriver e tem suporte a várias implementações de apoio a diversos navegadores. Existem vários métodos auxiliares para browser como “get”, “wait”, “driver” etc, que podem ser utilizados para os testes. O browser.get no caso, acessa a página que queremos.

#### element(by.[])

Element é o recurso utilizado para acessar um determinado elemento da página de testes e que contém a [API By](http://www.protractortest.org/#/api?view=ProtractorBy). Existem várias formas de acessá-los. Pode ser por: id, model, css, tagName, className e outros. Basta ver sobre no link acima.

É ALTAMENTE ACONSELHÁVEL QUE, PARA FACILITAR A MANIPULAÇÃO DOS ELEMENTOS, ELES CONTENHAM ID EM SEU HTML! A própria documentação do Protractor sugere essa adoção.

#### isPresent()

É uma promise que retorna um valor booleano caso o elemento exista. Apenas uma das inúmeras funções de auxílio que o Protractor disponibiliza para lidar com elementos de página e validá-los.

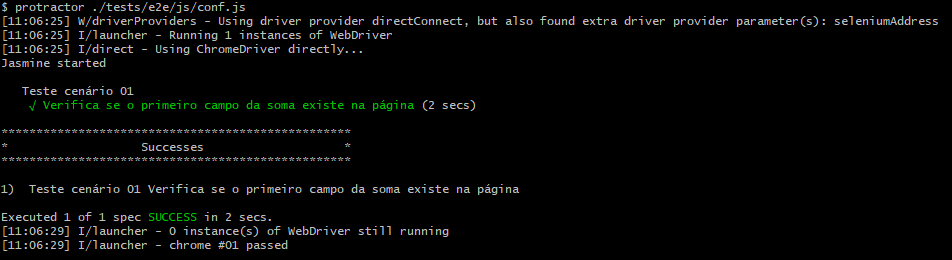
Esses são apenas os recursos utilizados NESTE teste. Para mais detalhes, veja a documentação do [Protractor](https://www.protractortest.org/#/locators).

### Executando o teste

Código compilado, basta abrir outro console (o principal já está rodando o watch), ir para o diretório raiz da aplicação (onde contém o package.json, tsconfig.json) e executar:

|  |
| --- |
| > protractor ./tests/e2e/js/conf.js |

Após digitar o comando, o browser irá abrir na página que o browser.get() estava buscando e logo em seguida, teremos o seguinte no console:



Ou seja, sucesso!

Repare que esse relatório que é exibido no console tudo está configurado conforme definido na prorpiedade onPrepare() do conf.ts.

#### SUGESTÃO: Performando a execução:

Para facilitar, eu gosto de configurar um comando de execução específico para os testes sem ter que passar esse caminho todo do arquivo. Pode ser feito, assim como o “npm start” um novo script no package.json. Vamos colocar com o nome de “e2e” e simplesmente vamos copiar o código de execução e colocar lá. Nossa parte de scripts do package.json deverá ficar assim:

*./package.json*

"scripts": {

"compile": "tsc",

"watch": "tsc -w",

"start": "npm run watch",

"e2e": "protractor ./tests/e2e/js/conf.js"

}

E ao executar o comando

|  |
| --- |
| > npm run e2e |

o código executará normalmente como no comando anterior!

### Adicionando mais cenários:

Vamos seguir a seguinte abordagem:

Teremos dois cenários, um validaremos os elementos da página (Cenário 01) e no outro, validaremos as operações da página (Cenário 02). Para isso, no nosso cenario01.spec.ts, vamos adicionar mais cenários. Ficará dessa forma:

*./tests/e2e/ts/specs/cenario01.spec.ts*

import { browser, element, by } from "protractor";

describe(' Teste cenário 01 - Validando elementos', () => {

/\* Aqui foi feita a primeira refatoração no código: para não ter que repetir \*/

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

/\*Observe que browser.getTitle() pega a propriedade title do header da página \*/

it("Valida nome do título", () => {

var titulo = element(by.tagName('h3'));

expect(titulo).toEqual("Super Calculator");

});

/\*\* Aqui eu sintetizei dois em um: validando os dois campos de parcela\*/

it("Verifica se os campos de parcela da soma existem na página", () => {

var primeiraParcela = element(by.model('first'));

var segundaParcela = element(by.model('second'));

expect(primeiraParcela.isPresent()).toBe(true);

expect(segundaParcela.isPresent()).toBe(true);

});

it("Valida se o botão 'Go' está presente", () => {

var gobutton = element(by.id('gobutton'));

expect(gobutton.isEnabled()).toBe(true);

});

});

E para o Cenário 02 vamos criar um novo arquivo, o cenario02.spec.ts dentro do mesmo diretório e ficará assim:

*./tests/e2e/ts/specs/cenario02.spec.ts*

import { element, by, browser } from "protractor";

describe('Teste cenário 02 - Funcionalidades gerais', () => {

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it('Soma dois números (no caso, 1 e 3)', () => {

var primeiraParcela = element(by.model('first'));

var segundaParcela = element(by.model('second'));

var gobutton = element(by.id('gobutton'));

/\* Usa sendKeys para passar para o elemento o valor do parâmetro \*/

primeiraParcela.sendKeys(1);

segundaParcela.sendKeys(3);

gobutton.click(); // Clica no botão Go! Note que ele só vai somar porque o valor default dele é o da operação de soma

/\* Busca o valor do resultado e valida com o texto esperado\*/

var resultado = element(by.binding('latest'));

expect(resultado.getText()).toEqual('4');

});

/\*\* Como fizemos só uma soma, essa soma deve ser incluída no histórico \*/

it('Valida inclusão no histórico', () => {

/\* perceba que aqui temos o element.all, significa que a variável historico carrega varios elementos (com as mesmas características OU NÃO)\*/

var historico = element.all(by.repeater('result in memory'));

/\* Como o elemento acima representa os elementos do histórico, vou comparar após a primeira soma se aparece UM elemento apenas \*/

expect(historico.count()).toEqual(1);

});

});

Lembre-se de adicionar o novo spec no mapeamento do conf.ts:

*./tests/e2e/ts/conf.ts*

suites: {

e2e: [

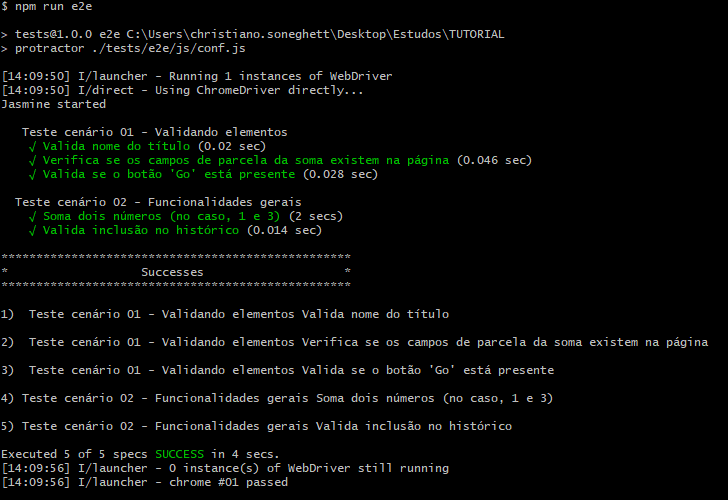
'./specs/cenario01.spec.js',

'./specs/cenario02.spec.js'

]

},

Até esse momento, nossos testes devem estar rodando muito bem! Segue um exemplo de como deve estar no console nessa altura:



## Boas práticas em escrita de códigos de testes

Até o momento, já escrevemos algumas coisas que nos permitiram começar os primeiros testes automatizados. Algumas coisas inclusive já dentro de um padrão de implementação, como por exemplo, a configuração de um arquivo conf.ts (poderíamos muito bem criar um cenário .spec.ts separado e executá-lo independente de arquivo conf.ts. Porém o arquivo conf é uma parte do que chamamos de boa estruturação de códigos. Todos as empresas e até mesmo desenvolvedores adotam certos padrões para desenvolvimento de suas aplicações, seja para front-end, back-end ou códigos de teste.

Também utilizei já um padrão de nomenclatura para nossos cenários, chamando os arquivos com o [nomeDoCenario].spec.ts e inclusive separei os cenários, diferenciando cada um do que eles faziam. Mas temos muitas outras coisas para melhorar!

Para esse código, estou adotando sugestões da própria documentação de padrão de codificação, além de características que eu já vi em muitos outros tutoriais. Muitos desenvolvedores costumam variar, mas não foge muito desse padrão não.

### Parte 1: Refatorando o código

Se observar bem no nosso código, eu expliquei anteriormente alguns recursos já foram colocados de uma forma que facilitam a organização e compreensão do mesmo. Mas sempre dá pra melhorar, certo?

O que já temos?

* Esquema de diretórios;
* Arquivos de configuração (tsconfig.ts, conf.ts)
* Nomenclatura dos arquivos de cenários ( sufixo *“.spec.ts*”)

O que falta? Muita coisa!

#### Limpando os specs e adicionando páginas

Se observar nosso código, nós declaramos as variáveis internamente dentro de nossos ‘its’. E em alguns lugares ela inclusive se repetiu! Desnecessário, certo?

E se separássemos uma página inteira para colocarmos todos os elementos que vamos utilizar? O código ficaria bem mais limpo, né? É isso que será feito.

Primeiro, crie um arquivo com o nome de inicio.page.ts dentro de ./tests/e2e/ts/pages. Em seguida, crie uma classe, pode chamar de InicioPage. O arquivo ficará dessa forma:

*./tests/e2e/ts/pages/inicio.page.ts*

class InicioPage {

};

Copie agora todas as referências de elementos dos specs e coloque dentro dessa classe, considerando todos como atributos da mesma. Fazendo os imports necessários, ficará assim:

*./tests/e2e/ts/pages/inicio.page.ts*

import { element, by, browser } from "protractor";

export class InicioPage {

titulo = element(by.tagName('h3'));

primeiraParcela = element(by.model('first'));

segundaParcela = element(by.model('second'));

historico = element.all(by.repeater('result in memory'));

resultado = element(by.binding('latest'));

gobutton = element(by.id('gobutton'));

};

Observação: estamos escrevendo um código em Typescript, ou seja, a política de modificadores de acesso (public, private, protected...) também existe nessa linguagem. Por padrão, os atributos da classe (como declarados acima) já vem implicitamente com o modificador public!

Em seguida, apagaremos as referências dos nossos specs, essas mesmas que colocamos aí. Nosso código ficará assim, ocasionando em **vários erros de compilação**, pois apagamos as referências dos elementos (o watch está rodando aí, né?):

*./tests/e2e/ts/specs/cenario01.spec.ts*

import { browser, element, by } from "protractor";

describe(' Teste cenário 01 - Validando elementos', () => {

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it("Valida nome do título", () => {

expect(titulo).toEqual("Super Calculator");

});

it("Verifica se os campos de parcela da soma existem na página", () => {

expect(primeiraParcela.isPresent()).toBe(true);

expect(segundaParcela.isPresent()).toBe(true);

});

it("Valida se o botão 'Go' está presente", () => {

expect(gobutton.isEnabled()).toBe(true);

});

});

*./tests/e2e/ts/specs/cenario02.spec.ts*

import { element, by, browser } from "protractor";

describe('Teste cenário 02 - Funcionalidades gerais', () => {

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it('Soma dois números (no caso, 1 e 3)', () => {

primeiraParcela.sendKeys(1);

segundaParcela.sendKeys(3);

gobutton.click();

expect(resultado.getText()).toEqual('4');

});

it('Valida inclusão no histórico', () => {

expect(historico.count()).toEqual(1);

});

});

Para os specs reconhecerem nossos elementos de página, vamos exportar a classe e importá-la nos specs. As referências antigas que estão nas specs, vamos acessá-las a partir de uma nova instância da classe da página.

Fazendo o processo acima, os cenários ficarão da seguinte forma:

*./tests/e2e/ts/specs/cenario01.spec.ts*

import { browser } from "protractor";

import { InicioPage } from "../pages/inicio.page";

describe(' Teste cenário 01 - Validando elementos', () => {

var Inicio: InicioPage = new InicioPage(); //Declaramos uma instância da nossa página aqui!

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it("Valida nome do título", () => {

expect(Inicio.titulo.getText()).).toEqual("Super Calculator");

});

it("Verifica se os campos de parcela da soma existem na página", () => {

expect(Inicio.primeiraParcela.isPresent()).toBe(true);

expect(Inicio.segundaParcela.isPresent()).toBe(true);

});

it("Valida se o botão 'Go' está presente", () => {

expect(Inicio.gobutton.isEnabled()).toBe(true);

});

});

*./tests/e2e/ts/specs/cenario02.spec.ts*

import { browser } from "protractor";

import { InicioPage } from "../pages/inicio.page";

describe('Teste cenário 02 - Funcionalidades gerais', () => {

var Inicio: InicioPage = new InicioPage(); //Declaramos uma instância da nossa página aqui!

beforeAll(() => {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it('Soma dois números (no caso, 1 e 3)', () => {

Inicio.primeiraParcela.sendKeys(1);

Inicio.segundaParcela.sendKeys(3);

Inicio.gobutton.click();

expect(Inicio.resultado.getText()).toEqual('4');

});

it('Valida inclusão no histórico', () => {

expect(Inicio.historico.count()).toEqual(1);

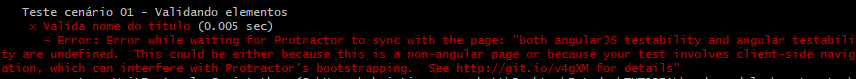
});

});

##### Erro encontrado!

E os problemas de compilação se encerraram! Vamos tentar executar o teste com npm run e2e.

Deu problema aí? Pois é. Deve ter encontrado o seguinte erro no console:



É apenas um trecho, pois esse erro se estendeu para os outros cenários. Quando tentamos mudar a abordagem de manipulação dos elementos, o interpretador de alguma forma achou que estávamos testando uma página obrigatoriamente construída com Angular. Porém não é o caso. Para resolver esse problema, [o próprio Protractor, em sua documentação, explica como lidar com páginas simples (com bootstrap manual) sem Angular](http://www.protractortest.org/#/system-setup). Para isso, vamos corrigir nossos arquivos, baseada na referência do Protractor: onde é utilizado browser.[método], será substituído por browser.driver.[método]. Como foi utilizado apenas para acessar a URL, a substituição será feita apenas nessa parte.

Ficará assim:

*./tests/e2e/ts/specs/cenario01.spec.ts e ./tests/e2e/ts/specs/cenario02.spec.ts*

beforeAll(() => {

browser.driver.get('http://juliemr.github.io/protractor-demo/');

});

E vamos tentar novamente executar o comando npm run e2e. Agora o teste irá rodar com sucesso!

#### Criando funções genéricas - Parte 1

Neste exemplo, temos em uma validação em que está sendo feita uma soma no Cenário 02. E se, na nossa page fosse criada uma função para realizar essa soma? Ao invés de repetir esse código em outros ‘its’, bastaria propagar a função “soma” criada.

Podemos fazer assim:

*./tests/e2e/ts/pages/inicio.spec.ts*

export class InicioPage {

titulo = element(by.tagName('h3'));

primeiraParcela = element(by.model('first'));

segundaParcela = element(by.model('second'));

historico = element.all(by.repeater('result in memory'));

resultado = element(by.binding('latest'));

gobutton = element(by.id('gobutton'));

soma(x: number, y: number): void {

this.primeiraParcela.sendKeys(x);

this.segundaParcela.sendKeys(y);

this.gobutton.click();

}

};

E modificar o Cenário 02 assim:

*./tests/e2e/ts/specs/cenario02.spec.ts*

//Código acima omitido

it('Soma dois números (no caso, 1 e 3)', () => {

Inicio.soma(1,3);

expect(Inicio.resultado.getText()).toEqual('4');

});

//Código abaixo omitido

Se executar, verá que conseguimos sintetizar um bloco de código (que poderia ser repetido em outras validações) em uma função só.

#### Criando funções genéricas – Parte 2

Até então, tudo bem, o código cada vez mais modularizado. Mas e se for necessário testar outras operações além da operação de soma? Perceba que no site de exemplo existe um combobox de operações. Podemos fazer uma função genérica para selecionar uma opção desse combobox. Porém como é uma função que pode ser aproveitada para diversos cenários, será criado um arquivo dentro de ‘components’ chamado utils.component.ts e já vamos criá-lo como uma classe a ser exportada.

*./tests/e2e/ts/components/utils.component.ts*

export class Utils {

}

E será criado um método estático capaz de selecionar a operação do combobox que quisermos. Como faremos isso? Temos duas formas:

1ª – Por índice (podemos conhecer ou não qual operação corresponde a cada índice)

2ª – Por correspondência de texto: comparando o elemento correspondente a operação esperada

##### Por índice:

*./tests/e2e/ts/components/utils.component.ts*

import { ElementArrayFinder, element, by, ElementFinder } from "protractor";

export class Utils {

static escolherOperacaoPorIndice(combobox: ElementFinder, index = 0) {

combobox.click();

var opcoes: ElementArrayFinder = element.all(by.tagName('option'));

opcoes.get(index).click();

}

}

Perceba que no código acima, os parâmetros da nossa função são: **combobox** que é um elemento tipo ElementFinder. Para acessar os outros elementos dentro dele, temos que clicar nele antes; e **index**, que é o índice do combobox que queremos acessar. Por default é 0.

Agora basta testar criando um outro ‘it’ no Cenário 02. Segue um exemplo:

*./tests/e2e/ts/specs/cenario02.spec.ts*

//Código acima omitido

it('Escolhe a terceira operação (índice 2)', () => {

var combobox = element(by.model('operator'));

Utils.escolherOperacaoPorIndice(combobox, 2);

});

});

Como o método é estático (static), pode-se acessá-lo sem instância da classe.

O teste executará normalmente, porém não temos como avaliar a operação que estamos utilizando por índice (a menos que a gente já soubesse qual que ela representa). **ASSUMINDO QUE JÁ SE SABE QUE A TERCEIRA OPERAÇÃO É O MÓDULO**, vamos validar nosso cenário:

*./tests/e2e/ts/specs/cenario02.spec.ts*

//Código acima omitido

it('Calcula o módulo entre 5 e 2', () => {

var combobox = element(by.model('operator'));

Utils.escolherOperacaoPorIndice(combobox, 2);

Inicio.primeiraParcela.sendKeys(5);

Inicio.segundaParcela.sendKeys(2);

Inicio.gobutton.click();

expect(Inicio.resultado.getText()).toEqual('1');

});

});

Mas perceba que voltamos a reproduzir código que possivelmente será repetido posteriormente. Como a obtenção por índice é bem “obscura” se partirmos do princípio que não sabemos exatamente o que contém cada índice (sabemos que ele tem operações matemáticas, mas não onde está cada um exatamente). Para melhorar um pouco essa situação, vamos para a segunda abordagem:

##### Por correspondência de texto:

Nessa opção já será usado um recurso novo do Typescript para não depender da manipulação das famigeradas promises: async/await

Inclua o trecho abaixo na classe Utils:

*./tests/e2e/ts/components/utils.component.ts*

static async escolherOperacaoPorTexto(combobox: ElementFinder, texto: string) {

combobox.click();

var opcoes: ElementFinder[] = await element.all(by.tagName('option'));

for (var opcao of opcoes) {

var s = await opcao.getText();

if (s === texto)

opcao.click();

}

}

Perceba: dessa vez estamos trabalhando com um recurso maravilhoso, que é capaz de salvar resultados de promises em variáveis sem ter que utilizar inúmeros call-backs. A variável opções guarda em um array de ElementFinder cada operação do combobox. Ao fazer um for nas opções, buscamos o texto de operação de cada uma delas e comparamos com o parâmetro **texto** passado na função.

Tendo nossa função genérica para acessar um combobox QUALQUER por texto. Agora para tratar de coisas específicas dos testes, vamos utilizar as pages. A page Inicio que foi criada será reformulada: será removido o método “soma” e incluído um novo para generalizar nossas operações:

*./tests/e2e/ts/pages/inicio.page.ts*

import { element, by } from "protractor";

import { Utils } from "../components/utils.component";

export class InicioPage {

titulo = element(by.tagName('h3'));

operacoesCombobox = element(by.model('operator')); // Incluído o mapeamento do combobox de operacoes

primeiraParcela = element(by.model('first'));

segundaParcela = element(by.model('second'));

historico = element.all(by.repeater('result in memory'));

resultado = element(by.binding('latest'));

gobutton = element(by.id('gobutton'));

// Método ‘soma’ removido e incluído o ‘realizaOperacao’

realizaOperacao(operacao: string, x: number, y: number) {

Utils.escolherOperacaoPorTexto(this.operacoesCombobox, operacao);

this.primeiraParcela.sendKeys(x);

this.segundaParcela.sendKeys(y);

this.gobutton.click();

};

};

Nesse momento nosso compilador vai apresentar um erro no cenario02.spec.ts. Dizendo que o método ‘soma’ usado no primeiro ‘it’ não existe mais. Substitua ‘soma’ por ‘realizaOperacao’ e inclua a operação correspondente a desejada, no caso, ‘+’. Assim temos:

*./tests/e2e/ts/specs/cenario02.spec.ts*

it('Soma dois números (no caso, 1 e 3)', () => {

Inicio.realizaOperacao('+', 1, 3);

expect(Inicio.resultado.getText()).toEqual('4');

});

Nesse momento o compilador parou de dar erro. Agora vamos reformular o código de validação do último it do cenário02. Como estávamos validando módulo, agora será tudo sintetizado na nova função criada. O Cenário 02 completo, fazendo as modificações necessárias ficará assim:

*./tests/e2e/ts/specs/cenario02.spec.ts*

import { browser } from "protractor";

import { InicioPage } from "../pages/inicio.page";

describe('Teste cenário 02 - Funcionalidades gerais', () => {

var Inicio: InicioPage = new InicioPage(); //Declaramos uma instância da nossa página aqui!

beforeAll(() => {

browser.driver.get('http://juliemr.github.io/protractor-demo/');

});

it('Soma dois números (no caso, 1 e 3)', () => {

Inicio.realizaOperacao('+', 1, 3);

expect(Inicio.resultado.getText()).toEqual('4');

});

it('Valida inclusão no histórico', () => {

expect(Inicio.historico.count()).toEqual(1);

});

it('Realiza o módulo de dois números (5 e 2)', () => {

Inicio.realizaOperacao('%', 5, 2);

expect(Inicio.resultado.getText()).toEqual('1');

});

});

**Exercício: faça um ‘it’ com cada operação.**

### Utilizando parâmetros

Existe uma coisa redundante no código. Temos o conf.ts que possui informações que podem ser buscadas nos outros arquivos e carregadas na medida que os testes estiverem sendo executados. A primeira coisa é a URL de acesso.

No arquivo conf.ts existe um parâmetro do config que se chama baseUrl, que pode ser incluída uma URL padrão que contenha acesso à pagina usada para teste.

[TO BE CONTINUED...]