Testes Unitários com Jasmine

BDD em Javascript

Aprenda desenvolvimento guiado por comportamentos com esse poderoso framework

Kazale.com

Testes Unitários com Jasmine

BDD em Javascript

Márcio Casale de Souza
Kazale.com

Para você e a todas as pessoas que de alguma forma acompanham o meu trabalho e me motivam e incentivam a seguir em frente compartilhando conhecimento com as demais pessoas

Márcio Casale de Souza

Deseja acelerar o seu aprendizado e aprender testes unitários com Javascript rapidamente?

Com o curso de Testes unitários com Javascript será possível!

udemy

Veja as vantagens:

- Você se tornará um profissional completo em teste de software
- Aprenderá a criar testes unitários com Jasmine
- Aprenderá automatizar testes com Karma
- Versionamento de código fonte com Git/GitHub
- Integração contínua com Travis CI

Veja suas garantias:

- Curso disponível na plataforma Udemy
- Mais de 9 horas de aula em vídeo
- Assista as aulas de seu computador ou smartfone
- Garantia de satisfação ou seu dinheiro de volta
- Certificado de conclusão de curso emitido pela Udemy
- Acesso total vitalício ao conteúdo do curso

Acesse através do link abaixo para ganhar um desconto de 30%!

Curso de Testes Unitários com Javascript

Autor

Sou o Márcio Casale de Souza, formado em Sistemas de Informação e Pós Graduado em Desenvolvimento de Sistemas Corporativos em Java.

Possuo vasta experiência em desenvolvimento de sistemas, no qual trabalho profissionalmente a mais de oito anos.

Sou entusiasta de novas tecnologias, e gosto de compartilhar e ensinar o que tenho aprendido ao longo dos anos.

Website: http://kazale.com

Email: contato@kazale.com

LinkedIn: https://ie.linkedin.com/in/m4rciosouza

GitHub: https://github.com/m4rciosouza

Facebook: https://www.facebook.com/Kazaleit

YouTube: https://www.youtube.com/c/MarcioSouzaKazale

Prefácio

O objetivo deste livro é servir como uma referência das funcionalidades do framework Javascript Jasmine, utilizado para criação de testes unitários.

O Jasmine é um framework BDD, ou seja, para desenvolvimento guiado por testes, e aqui serão apresentadas todas as suas funcionalidades através de sua descrição e exemplos práticos.

Introdução

Com a evolução e complexidade dos softwares a serem desenvolvidos, cada vez mais é importante ter uma boa cobertura de testes no código desenvolvido, para evitar bugs e problemas futuros ao adicionar novas funcionalidades ou mesmo alterar um código.

O Jasmine é um framework utilizado para criação de testes em Javascript.

Ele utiliza os conceitos do BDD (Behavior Driven Development), que seriam testes guiados por comportamento, que permite a criação de testes intuitivos e de fácil compreensão.

Suas vantagens incluem ser rápido, e não possuir dependências externas, incluindo por padrão tudo o que é necessário para testar uma aplicação.

Possui a capacidade de executar os testes diretamente no navegador, ou por linha de comando no terminal, além de ser de fácil instalação e configuração.

Mesmo sendo um framework Javascript, ele pode ser utilizado em projetos Ruby ou Python, além de se integrar perfeitamente com NodeJS.

Sua página oficial é https://jasmine.github.io.

A seguir serão estudados todos os componentes e recursos do Jasmine 2.5.2, que é a última disponível no momento da escrita deste livro.

Sumário

1. Distribuição Standalone

A distribuição standalone permite começar um projeto Jasmine rapidamente, utilizando um formato de projeto configurado e pronto para uso.

É a versão a ser utilizada para aprender os recursos do Jasmine

Possui o arquivo ‘SpecRunner.html’, responsável por executar os testes no navegador, e exibir o resultado do teste em forma de relatório

Possui tudo o que é preciso para aprender sobre o Jasmine, para posteriormente incorporar o framework em projetos mais complexos e automatizados

Página de download: https://github.com/jasmine/jasmine/releases

Sua instalação é bastante simples, basta fazer o download na url acima, descompactar o arquivo, e executar o arquivo ‘SpecRunner.html’ para executar os testes.

2. Suítes

- ❏ Suítes de testes servem para definir o escopo do que está sendo testado
- ❏ Uma aplicação é composta por diversas suítes de testes
- ❏ Exemplos de suítes seriam: Cadastro de Clientes, Operações Matemáticas,...
- ❏ No Jasmine, a suíte é uma função global Javascript chamada ‘describe’, que possui dois parâmetros, que seriam sua descrição e os testes (specs)

Exemplo:

```
describe("Operação de Adição", function() {
});
```

3. Testes (Specs)

- ❏ Specs são os testes que validam uma suíte de testes
- ❏ Assim como as suítes, ela é uma função global Javascript chamada ‘it’, que contém dois parâmetros, uma descrição e uma função, respectivamente
- ❏ Dentro do segundo parâmetro, é onde adicionamos as verificações (expectations)

Exemplo:

```
it("deve garantir que 1 + 9 = 10", function() {
});
```

4. Verificações (Expectations)

- ❏ Verificações servem para validar um resultado de um teste
- ❏ O Jasmine possui uma função global Javascript chamada ‘expect’, que recebe um parâmetro como argumento, que é o resultado a ser verificado
- ❏ O ‘expect’ deve ser utilizado em conjunto com uma comparação (Matcher), que conterá o valor a ser comparado
- ❏ Uma Spec poderá conter uma ou mais verificações
- ❏ Uma boa prática é sempre manter as verificações no final da função

---

Exemplo:

```
expect(Calculadora.adicionar(1, 9)).toBe(10);
```

5. Comparações (Matchers)

- ❏ Comparações (Matchers) são funções que retornam um valor booleano para ser verificado através de uma expectation (verificação)
- ❏ O Jasmine contém uma série de matchers implementados por padrão
- ❏ É possível criar seu próprio matcher
- ❏ Todo matcher pode ser negado através da palavra chave ‘not’, inserida entre uma expectation e um matcher

---

A seguir serão apresentadas todas comparações (matchers):

5.1. toBe

- ❏ Realiza a comparação com o operador '===', que compara o valor e também o tipo do objeto
- ❏ Deve ser utilizado para comparar valores de forma mais efetiva pelo fato de também verificar o tipo do objeto

---

Exemplo:

```
describe("Suíte de testes do tópico 5.1", function() {

        var valorBooleano = true;
        var valorBooleanoCopia = valorBooleano;
        var valorBooleanoTexto = "true";
        var obj1 = { 'valor': valorBooleano };
        var obj2 = { 'valor': valorBooleano };

        it("deve validar o uso do matcher 'toBe'", function() {
                expect(valorBooleano).toBe(true);
                expect(valorBooleanoCopia).toBe(valorBooleano);
                expect(valorBooleano).not.toBe(valorBooleanoTexto);
                expect(valorBooleanoTexto).toBe("true");
                expect(obj1).not.toBe(obj2);
        });

});
```

5.2. toEqual

- ❏ Realiza a comparação de dois elementos de modo muito similar ao ‘toBe’
- ❏ A única diferença em relação ao ‘tbBe’ é que ele não compara o tipo do objeto, somente seu valor
- ❏ É recomendado seu uso para comparação de valores literais

---

Exemplo:

```
describe("Suíte de testes do tópico 5.2", function() {

        var valorBooleano = true;
        var valorBooleano2 = true;
        var valorBooleanoCopia = valorBooleano;
        var valorBooleanoTexto = "true";
        var obj1 = { 'valor': valorBooleano };
        var obj2 = { 'valor': valorBooleano };

        it("deve validar o uso do matcher 'toEqual'", function() {
                expect(valorBooleano).toEqual(true);
                expect(valorBooleano).toEqual(valorBooleano2);
                expect(valorBooleanoCopia).toEqual(valorBooleano);
                expect(valorBooleano).not.toEqual(valorBooleanoTexto);
                expect(valorBooleanoTexto).toEqual("true");
                expect(obj1).toEqual(obj2);
        });

});
```

5.3. toMatch

- ❏ Realiza a comparação utilizando expressões regulares
- ❏ Caso seja passada uma string como parâmetro, o comparador tentará encontrar o texto passado dentro do valor a ser comparado

---

Exemplo:

```
describe("Suíte de testes do tópico 5.3", function() {

        var textoComparar = "Curso de testes com Jasmine";

        it("deve validar o uso do matcher 'toMatch'", function() {
                expect(textoComparar).toMatch(/Jasmine/);
                expect(textoComparar).toMatch("Jasmine");
                expect(textoComparar).toMatch(/jasmine/i);
                expect(textoComparar).not.toMatch(/Javascript/);
                 expect("14/12/2016").toMatch(/^\d{2}\/\d{2}\/\d{4}$/);
        });

});
```

5.4. toBeDefined

- ❏ Realiza a comparação de um objeto como não sendo ‘undefined’
- ❏ Prefira usar ‘toBeUndefined’ ao invés de ‘not.toBeDefined’ para deixar o código de mais fácil compreensão

---

Exemplo:

```
describe("Suíte de testes do tópico 5.4", function() {

	var numero = 10;
	var texto;
	var obj = { 'valor': 10 };

	it("deve validar o uso do matcher 'toBeDefined'", function() {
		expect(numero).toBeDefined();
		expect(texto).not.toBeDefined();
		expect(obj.valor).toBeDefined();
		expect(obj.mensagem).not.toBeDefined();
	});

});
```

5.5. toBeUndefined

- ❏ Realiza a comparação de um objeto como sendo ‘undefined’
- ❏ Prefira usar ‘toBeDefined’ ao invés de ‘not.toBeUndefined’ para deixar o código de mais fácil compreensão

---

Exemplo:

```
describe("Suíte de testes do tópico 5.5", function() {

        var numero = 10;
        var texto;
        var obj = { 'valor': 10 };

        it("deve validar o uso do matcher 'toBeUnDefined'", function() {
                expect(texto).toBeUndefined();
                expect(numero).not.toBeUndefined();
                expect(obj.mensagem).toBeUndefined();
                expect(obj.valor).not.toBeUndefined();
        });

});
```

5.6. toBeNull

- ❏ Realiza a comparação de um objeto como sendo ‘null’
- ❏ Usamos ‘null’ para dizer que uma variável não possui um valor
- ❏ O valor ‘null’ se diferencia de ‘undefined’ pelo fato de ‘null’ ser um tipo e ‘undefined’ ser uma variável ainda não definida

---

Exemplo:

```
describe("Suíte de testes do tópico 5.6", function() {

        var objeto = null;
        var texto;
        var numero = 10;

        it("deve validar o uso do matcher 'toBeNull'", function() {
                expect(objeto).toBeNull();
                expect(numero).not.toBeNull();
                expect(texto).not.toBeNull();
                expect(objeto).not.toEqual(texto);
        });

});
```

5.7. toBeTruthy

- ❏ Realiza uma comparação dizendo se uma variável ou objeto possui um valor válido
- ❏ Um valor será considerado válido caso ele possua um valor diferente de ‘false’, ‘0’, ‘’, ‘undefined’, ‘null’, ou ‘NaN’
- ❏ Deve ser utilizado quando a verificação abordar valores inválidos distintos, baseados nas opções citadas acima
- ❏ Prefira usar ‘toBeFalsy’ ao invés de ‘not.toBeTruthy’ para deixar o código mais fácil de compreender

---

Exemplo:

```
describe("Suíte de testes do tópico 5.7", function() {

        var objeto = { 'valor': 123 };
        var texto;
        var numero = 10;

        it("deve validar o uso do matcher 'toBeTruthy'", function() {
                expect(objeto).toBeTruthy();
                expect(numero).toBeTruthy();
                expect(texto).not.toBeTruthy();
        });

});
```

5.8. toBeFalsy

- ❏ Realiza uma comparação dizendo se uma variável ou objeto possui um valor inválido
- ❏ Um valor será considerado inválido caso seja ‘false’, ‘0’, ‘’, ‘undefined’, ‘null’, ou ‘NaN’
- ❏ Deve ser utilizado quando a verificação abordar valores inválidos distintos, baseados nas opções citadas acima
- ❏ Prefira usar ‘toBeTruthy’ ao invés de ‘not.toBeFalsy’, para deixar o código de mais fácil compreensão

---

Exemplo:

```
describe("Suíte de testes do tópico 5.8", function() {

        var numero = 10;

        it("deve validar o uso do matcher 'toBeFalsy'", function() {
                expect(false).toBeFalsy();
                expect("").toBeFalsy();
                expect(0).toBeFalsy();
                expect(undefined).toBeFalsy();
                expect(null).toBeFalsy();
                expect(NaN).toBeFalsy();
                expect(numero).not.toBeFalsy();
                expect("false").not.toBeFalsy();
        });

});
```

5.9. toContain

- ❏ Realiza a busca por determinado item em uma array
- ❏ Também pode ser utilizado para buscar uma substring dentro de uma string
- ❏ Não suporta busca por expressões regulares

Exemplo:

```
describe("Suíte de testes do tópico 5.9", function() {

	var nomes = ["Fulano", "Ciclano", "Beltrano"];
	var nomesTexto = "Fulano Ciclano Beltrano";

	it("deve validar o uso do matcher 'toContain'", function() {
		expect(nomes).toContain("Ciclano");
		expect(nomesTexto).toContain("Fulano");
		expect(nomesTexto).toContain("Bel");
		expect(nomes).not.toContain("Maria");
		expect(nomes).not.toContain("ciclano");
	});

});
```

5.10. toBeLessThan

- ❏ Compara se um valor numérico é menor do que outro valor
- ❏ Realiza a conversão para valor numérico antes da comparação, podendo o valor ser passado em formato texto
- ❏ Prefira usar ‘toBeGreaterThan’ ao invés de ‘not.toBeLessThan’, para deixar o código de mais fácil compreensão
- ❏ Para valores iguais utilize o ‘toEqual’

Exemplo:

```
describe("Suíte de testes do tópico 5.10", function() {

        const PI = 3.1415;
        var numero = 2;

        it("deve validar o uso do matcher 'toBeLessThan'", function() {
                expect(numero).toBeLessThan(PI);
                expect("1.2").toBeLessThan(PI);
                expect(5).not.toBeLessThan(PI);
                 expect(PI).not.toBeLessThan(PI);
        });

});
```

5.11. toBeGreaterThan

- ❏ Compara se um valor numérico é maior do que outro valor
- ❏ Realiza a conversão para valor numérico antes da comparação, podendo o valor ser passado em formato texto
- ❏ Prefira usar “toBeLessThan” ao invés de “not.toBeGreaterThan” para deixar o código de mais fácil compreensão
- ❏ Para valores iguais utilize o “toEqual”

Exemplo:

```
describe("Suíte de testes do tópico 5.11", function() {

        const PI = 3.1415;
        var numero = 4;

        it("deve validar o uso do matcher 'toBeGreaterThan'", function() {
                expect(numero).toBeGreaterThan(PI);
                expect("3.2").toBeGreaterThan(PI);
                expect(2).not.toBeGreaterThan(PI);
                expect(PI).not.toBeGreaterThan(PI);
        });

});
```

5.12. toThrow

- ❏ Verifica se uma exceção é lançada por um método
- ❏ Não realiza a validação em detalhe o tipo da exceção lançada, apenas certifica que um erro ocorreu na execução da função
- ❏ Deve ser utilizada quando deseja apenas certificar que um erro ocorreu, sem se preocupar com detalhes como tipo ou mensagem de erro

---

Exemplo:

```
describe("Suíte de testes do tópico 5.12", function() {

	var comErro = function() {
		return numero * 10;
	};

	var semErro = function(numero) {
		return numero * 10;
	};

	it("deve validar o uso do matcher 'toThrow'", function() {
		expect(comErro).toThrow();
		expect(semErro).not.toThrow();
	});

});
```

5.13. toThrowError

- ❏ Verifica se uma exceção é lançada por um método
- ❏ Valida o tipo da exceção lançada
- ❏ Valida a mensagem de erro contida na exceção
- ❏ Suporta expressão regular na validação da mensagem de erro da exceção
- ❏ Deve ser utilizada para maior controle do erro lançado

Exemplo:

```
describe("Suíte de testes do tópico 5.13", function() {

	var calcularDobro = function(numero) {
		if (numero <= 0) {
			throw new TypeError("O número deve ser maior que 0.");
		}
		return numero * numero;
	};

	it("deve validar o uso do matcher 'toThrowError'", function() {
		expect(function() { calcularDobro(0) })
			.toThrowError();
		expect(function() { calcularDobro(0) })
			.toThrowError("O número deve ser maior que 0.");
		expect(function() { calcularDobro(0) })
			.toThrowError(/maior que 0/);
		expect(function() { calcularDobro(0) })
			.toThrowError(TypeError);
		expect(function() { calcularDobro(0) })
			.toThrowError(TypeError,
				"O número deve ser maior que 0.");
		expect(calcularDobro).not.toThrowError();
	});
});
```

6. Falha manual (Fail)

- ❏ Falha manual permite interromper um teste lançando um erro
- ❏ O Jasmine possui a função “fail” para falhar manualmente um teste
- ❏ Utilizamos a falha manual para certificar que uma operação não desejada não seja executada

---

Exemplo:

```
describe("Testa a função 'fail' de falha manual", function() {

  var operacao = function(deveExecutar, callBack) {
    if (deveExecutar) {
      callBack();
    }
  };

  it("não deve executar a função de callBack", function() {
    operacao(false, function() {
      fail("Função de callback foi executada");
    });
  });
});
```

7. Executando código antes e depois do teste

- ❏ O Jasmine permite executar códigos antes e depois de um teste / suíte com o uso de funções especiais
- ❏ Com o uso dessas funções repetições de códigos podem ser evitadas
- ❏ Permite executar um mesmo código antes e/ou depois de cada suíte
- ❏ Permite executar um código antes e/ou depois de cada teste

---

A seguir serão apresentadas todas as operações de execução ‘beforeEach’, ‘afterEach’, ‘beforeAll’, ‘afterAll’:

7.1. beforeEach

- ❏ Função Javascript global do Jasmine que é executada antes de cada teste
- ❏ Por ser executada antes de cada teste, serve para inicializar ou reiniciar um status
- ❏ Pode também executar uma ação antes de cada teste

Exemplo:

```
describe("Suíte de testes do tópico 7.1", function() {

  var contador = 0;

  beforeEach(function() {
    contador++;
  });

  it("deve exibir o contador com valor 1", function() {
    expect(contador).toEqual(1);
  });

  it("deve exibir o contador com valor 2", function() {
    expect(contador).toEqual(2);
  });
});
```

7.2. afterEach

- ❏ Função Javascript global do Jasmine que é executada depois de cada teste
- ❏ Por ser executada depois de cada teste, serve para reiniciar um status
- ❏ Pode também executar uma ação depois de cada teste

Exemplo:

```
describe("Suíte de testes do tópico 7.2", function() {

  var contador = 0;

  beforeEach(function() {
    contador++;
  });

  afterEach(function() {
    contador = 0;
  });

  it("deve exibir o contador com valor 1", function() {
    expect(contador).toEqual(1);
  });

  it("deve continuar exibindo o contador com valor 1", function() {
    expect(contador).toEqual(1);
  });
});
```

7.3. beforeAll

- ❏ Função Javascript global do Jasmine que é executada uma única vez antes da execução dos testes
- ❏ Por ser executada antes de todos os testes, serve para inicializar um status, criar objetos

---

Exemplo:

```
describe("Suíte de testes do tópico 7.3", function() {

  var contador;

  beforeAll(function() {
    contador = 10;
  });

  beforeEach(function() {
    contador++;
  });

  it("deve exibir o contador com valor 11", function() {
    expect(contador).toEqual(11);
  });

  it("deve exibir o contador com valor 12", function() {
    expect(contador).toEqual(12);
  });
});
```

7.4. afterAll

- ❏ Função Javascript global do Jasmine que é executada uma única vez depois da execução dos testes
- ❏ Por ser executada depois de todos os testes, serve para limpar algum status global

---

Exemplo:

```
describe("Suíte de testes do tópico 7.4", function() {

  var contador;

  beforeAll(function() {
    contador = 10;
  });

  afterAll(function() {
    contador = 0;
  });

  it("deve exibir o contador com valor 10", function() {
    expect(contador).toEqual(10);
  });

  it("deve manter o contador com valor 10", function() {
    expect(contador).toEqual(10);
  });
});
```

8. Aninhando suítes

- ❏ Suítes podem ser aninhadas e conter outras suítes dentro delas
- ❏ As funções especiais como o “beforeEach” ou “afterAll” serão executadas antes e depois de todos os testes, em ordem
- ❏ Tome cuidado ao aninhar suítes para não tornar o teste complexo e de difícil compreensão

---

Exemplo:

```
describe("Suíte de testes do tópico 8", function() {
  var contadorExterno = 0;
  beforeEach(function() {
    contadorExterno++;
  });
  it("deve ter incrementado o contador externo para 1", function() {
    expect(contadorExterno).toEqual(1);
  });

  describe("Suíte aninhada à anterior", function() {
    var contadorInterno = 1;
    beforeEach(function() {
      contadorInterno++;
    });

    it("deve conter o valor '2' para ambos contadores", function() {
      expect(contadorInterno).toEqual(contadorExterno);
    });
  });
});
```

9. Desabilitando suítes

- ❏ Uma suíte pode ser desabilitada a qualquer momento renomeando a função “describe” para “xdescribe”
- ❏ Todos os testes contidos dentro da suíte desabilitada serão ignorados e não serão exibidos no relatório de execução

---

Exemplo:

```
xdescribe("Suíte desabilitada devido ao uso do 'xdescribe'", function() {
  var contador = 0;

  beforeEach(function() {
    contador += 1;
  });

  it("deve garantir que o contador foi incrementado em 1", function() {
    expect(contador).toEqual(1);
  });
});
```

10. Desabilitando testes

- ❏ Assim como suítes, os testes também podem ser desabilitados ao renomear a função “it” por “xit”
- ❏ Os testes também serão considerados inativos caso só possuam a descrição como argumento
- ❏ Por último é possível usar a função “pending” dentro do teste para inativá-lo

---

Exemplo:

```
xit("teste desabilitado devido ao uso da função 'it'", function() {
  expect(true).toBe(false);
});

it("teste desabilitado por não possuir uma função definida");

it("teste desabilitado devido ao uso da função 'pending'", function() {
  expect(true).toBe(false);
  pending('teste ainda pendente...');
});
```

11. Spies

- ❏ Spies são objetos falsos utilizados quando queremos manipular algum retorno que não faça parte do teste em si
- ❏ Spies são utilizados para isolar somente o bloco de código que estamos testando
- ❏ Spies somente poderão ser criados dentro dos blocos “describe” e “it”
- ❏ Spies são removidos ao término da execução da suíte

A seguir serão apresentadas todas as operações associadas ao uso de Spies:

11.1. spyOn

- ❏ spyOn serve para criar um mock (objeto falso) a ser utilizado nos testes
- ❏ Um objeto spy contém uma série de atributos que serão estudados ao longo do capítulo
- ❏ A função spyOn recebe como parâmetros o nome do objeto e do método a serem utilizados como mock

---

Exemplo:

```
describe("Suíte de testes do tópico 11.1", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });
});
```

11.2. toHaveBeenCalled

- ❏ toHaveBeenCalled serve para informar se um método do spy foi executado ao menos uma vez
- ❏ toHaveBeenCalled não possui parâmetros

Exemplo:

```
describe("Suíte de testes do tópico 11.2", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'toHaveBeenCalled'", function() {
    calculadora.somar(1, 1);
    expect(calculadora.somar).toHaveBeenCalled();
  });
});
```

11.3. toHaveBeenCalledTimes

- ❏ toHaveBeenCalledTimes serve para verificar quantas vezes um método do spy foi chamado
- ❏ toHaveBeenCalledTimes recebe como parâmetro o número de execuções a ser verificado

---

Exemplo:

```
describe("Suíte de testes do tópico 11.3", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'toHaveBeenCalledTimes'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 2);
    expect(calculadora.somar).toHaveBeenCalledTimes(2);
  });
});
```

11.4. toHaveBeenCalledWith

- ❏ toHaveBeenCalledWith serve para verificar com quais parâmetros um método do spy foi chamado
- ❏ toHaveBeenCalledWith recebe como parâmetro os valores da chamada do método do spy, separados por vírgula

---

Exemplo:

```
describe("Suíte de testes do tópico 11.4", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'toHaveBeenCalledWith'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 2);
    expect(calculadora.somar).toHaveBeenCalledWith(1, 1);
    expect(calculadora.somar).toHaveBeenCalledWith(2, 2);
  });
});
```

11.5. and.callThrough

- ❏ and.callThrough serve para informar ao spy que o método original deve ser executado
- ❏ and.callThrough deve ser aplicado ao objeto spy
- ❏ Nesse caso o método original será executado, e todos os recursos do spy serão mantidos e estarão disponíveis para verificação

---

Exemplo:

```
describe("Suíte de testes do tópico 11.5", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    },
    subtrair: function(n1, n2) {
          return n1 - n2;
    }
  };
  beforeEach(function() {
    spyOn(calculadora, "somar").and.callThrough();
    spyOn(calculadora, "subtrair");
  });

  it("deve validar o uso do 'and.callThrough'", function() {
    expect(calculadora.somar(1, 1)).toEqual(2);
    expect(calculadora.subtrair(2, 2)).toBeUndefined();
    expect(calculadora.somar).toHaveBeenCalledTimes(1);
  });
});
```

11.6. and.returnValue

- ❏ and.returnValue serve para informar ao spy o valor de retorno de determinado método
- ❏ and.returnValue deve ser aplicado ao objeto spy

Exemplo:

```
describe("Suíte de testes do tópico 11.6", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar").and.returnValue(10);
  });

  it("deve validar o uso do 'and.returnValue'", function() {
    expect(calculadora.somar(5, 2)).toEqual(10);
    expect(calculadora.somar).toHaveBeenCalled();
  });
});
```

11.7. and.returnValues

- ❏ and.returnValues serve para informar ao spy quais os valores a serem retornados por chamada
- ❏ and.returnValues aceita como parâmetro um ou mais valores, separados por vírgula
- ❏ Se o número de chamadas for maior do que o de valores a serem retornados, será retornado “undefined”
- ❏ and.returnValues deve ser aplicado ao objeto spy

Exemplo:

```
describe("Suíte de testes do tópico 11.7", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar").and.returnValues(10, 20);
  });

  it("deve validar o uso do 'and.returnValues'", function() {
    expect(calculadora.somar(5, 2)).toEqual(10);
    expect(calculadora.somar(5, 2)).toEqual(20);
    expect(calculadora.somar(5, 2)).toBeUndefined();
    expect(calculadora.somar).toHaveBeenCalledTimes(3);
  });
});
```

11.8. and.callFake

- ❏ and.callFake serve para definir uma nova implementação para um método de um spy
- ❏ and.callFake deve ser aplicado ao objeto spy
- ❏ and.callFake recebe como parâmetro uma função com a nova implementação a ser executada quando o método for chamado

---

Exemplo:

```
describe("Suíte de testes do tópico 11.8", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar").and.callFake(
          function(n1, n2) {
                return n1 * n2;
          });
  });

  it("deve validar o uso do 'and.callFake'", function() {
    expect(calculadora.somar(5, 2)).toEqual(10);
    expect(calculadora.somar).toHaveBeenCalled();
  });
});
```

11.9. and.throwError

- ❏ and.throwError serve para informar ao spy que determinado método deve lançar um erro ao ser executado
- ❏ and.throwError deve ser aplicado ao objeto spy
- ❏ and.throwError recebe como parâmetro uma string contendo a mensagem de erro a ser lançada

---

Exemplo:

```
describe("Suíte de testes do tópico 11.9", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar").and.throwError(
          "valores inválidos");
  });

  it("deve validar o uso do 'and.throwError'", function() {
    expect(function() { calculadora.somar(5, 2); }).
      toThrowError("valores inválidos");
    expect(calculadora.somar).toHaveBeenCalled();
  });
});
```

11.10. calls.any

- ❏ Todo spy possui um atributo “calls” com informações sobre suas chamadas
- ❏ O “calls.any” serve para indicar se o método do spy foi chamada ao menos uma vez
- ❏ Ele não recebe parâmetros e retorna um valor booleano se ocorreu ou não uma chamada ao método do spy

---

Exemplo:

```
describe("Suíte de testes do tópico 11.10", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.any'", function() {
    expect(calculadora.somar.calls.any()).toBeFalsy();
    calculadora.somar(1, 1);
    expect(calculadora.somar.calls.any()).toBeTruthy();
  });
});
```

11.11. calls.count

- ❏ O “calls.count” armazena e retorna o número de vezes que um método do spy foi chamado
- ❏ Ele não possui parâmetros e retorna o número de chamadas do método

---

Exemplo:

```
describe("Suíte de testes do tópico 11.11", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.count'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 2);
    expect(calculadora.somar.calls.count()).toEqual(2);
  });
});
```

11.12. calls.argsFor

- ❏ O “calls.argsFor” armazena e retorna uma lista (array) contendo os parâmetros utilizados em cada chamada do método de um spy
- ❏ Ele recebe como parâmetro o índice da chamada a ser retornada
- ❏ É bastante útil para validar se um método foi chamado com os parâmetros corretos

---

Exemplo:

```
describe("Suíte de testes do tópico 11.12", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.argsFor'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 2);
    expect(calculadora.somar.calls.argsFor(0)).toEqual([1, 1]);
    expect(calculadora.somar.calls.argsFor(1)).toEqual([2, 2]);
  });
});
```

11.13. calls.allArgs

- ❏ O “calls.allArgs” retorna uma lista com todos os argumentos de todas as chamadas aos métodos de um spy
- ❏ Ele não recebe nenhum argumento como parâmetro
- ❏ Prefira utilizar o “calls.argsFor” quando precisar verificar um item em específico

---

Exemplo:

```
describe("Suíte de testes do tópico 11.13", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.allArgs'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 2);
    expect(calculadora.somar.calls.allArgs())
      .toEqual([[1, 1], [2, 2]]);
  });
});
```

11.14. calls.all

- ❏ O “calls.all” contém e retorna todas as informações de chamadas de um método do spy
- ❏ As informações armazenadas são o tipo de objeto (object), os parâmetros de chamada (args), e os valores de retorno (returnValue)
- ❏ Os itens acima são agrupados em uma lista, e são referenciados por número de chamada

---

Exemplo:

```
describe("Suíte de testes do tópico 11.14", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.all'", function() {
    calculadora.somar(1, 1);
    var retorno = calculadora.somar.calls.all();
    expect(retorno[0].object).toEqual(calculadora);
    expect(retorno[0].args).toEqual([1, 1]);
    expect(retorno[0].returnValue).toBeUndefined();
  });
});
```

11.15. calls.mostRecent

- ❏ O “calls.mostRecent” retorna os dados da última chamada do método do spy
- ❏ Seria o mesmo que acessar o último elemento (quantidade de itens - 1) da lista contida em “calls.all”

Exemplo:

```
describe("Suíte de testes do tópico 11.15", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.mostRecent'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 3);
    var retorno = calculadora.somar.calls.mostRecent();
    expect(retorno.object).toEqual(calculadora);
    expect(retorno.args).toEqual([2, 3]);
    expect(retorno.returnValue).toBeUndefined();
  });
});
```

11.16. calls.first

- ❏ O “calls.first” retorna os dados da primeira chamada do método do spy
- ❏ Seria o mesmo que acessar o primeiro elemento (posição 0) da lista contida em “calls.all”

Exemplo:

```
describe("Suíte de testes do tópico 11.16", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.first'", function() {
    calculadora.somar(1, 1);
    calculadora.somar(2, 3);
    var retorno = calculadora.somar.calls.first();
    expect(retorno.object).toEqual(calculadora);
    expect(retorno.args).toEqual([1, 1]);
    expect(retorno.returnValue).toBeUndefined();
  });
});
```

11.17. calls.reset

- ❏ O “calls.reset” serve para limpar a lista com os dados das chamadas dos métodos de um spy
- ❏ Pode ser útil quando tiver suítes aninhadas ou mesmo precisar restaurar o valor padrão das chamadas

---

Exemplo:

```
describe("Suíte de testes do tópico 11.17", function() {

  var calculadora = {
    somar: function(n1, n2) {
      return n1 + n2;
    }
  };

  beforeEach(function() {
    spyOn(calculadora, "somar");
  });

  it("deve validar o uso do 'calls.reset'", function() {
    calculadora.somar(1, 1);
    expect(calculadora.somar.calls.any()).toBeTruthy();
    calculadora.somar.calls.reset();
    expect(calculadora.somar.calls.any()).toBeFalsy();
  });
});
```

11.18. createSpy

- ❏ createSpy é uma função global Javascript do Jasmine
- ❏ Serve para criar um spy do zero
- ❏ Ele possui todos os atributos de um objeto spy comum
- ❏ Recebe como parâmetro o nome da função a ser criado o spy
- ❏ Deve ser utilizado quando precisa de um objeto que não se tem acesso direto a ele
- ❏ createSpy possui a limitação de não permitir implementar o método declarado, assim como somente permite a criação de um método

---

Exemplo:

```
describe("Suíte de testes do tópico 11.18", function() {

  var somar = jasmine.createSpy("somar");

  it("deve validar o uso do 'createSpy'", function() {
    somar(1, 2);
    expect(somar).toHaveBeenCalled();
    expect(somar).toHaveBeenCalledWith(1, 2);
    expect(somar.calls.mostRecent().args[0]).toEqual(1);
  });
});
```

11.19. createSpyObj

- ❏ createSpyObj é uma função global Javascript do Jasmine
- ❏ Serve para criar um objeto spy do zero
- ❏ Ele possui todos os atributos de um objeto spy comum
- ❏ Recebe como parâmetro o nome do objeto a ser criado, assim como seus métodos em formato de array
- ❏ Deve ser utilizado quando precisa de um objeto que não se tem acesso direto a ele
- ❏ createSpyObj permite a utilização de todos os elementos “and.” estudados anteriormente

---

Exemplo:

```
describe("Suíte de testes do tópico 11.19", function() {

  var calculadora = jasmine.createSpyObj("calculadora",
        ["somar", "subtrair"]);

  calculadora.somar.and.returnValue(10);

  it("deve validar o uso do 'createSpyObj'", function() {
    expect(calculadora.somar(1, 2)).toEqual(10);
    expect(calculadora.somar).toHaveBeenCalled();
    expect(calculadora.somar).toHaveBeenCalledWith(1, 2);
    expect(calculadora.somar.calls.mostRecent().args[0]).toEqual(1);
  });
});
```

12. Objeto “jasmine”

- ❏ O objeto “jasmine” possui 5 comparadores genéricos para utilização
- ❏ Os comparadores podem retornar diferentes valores, como valores booleanos, objetos,..

A seguir serão apresentadas todas as operações associadas ao uso do objeto ‘jasmine’:

12.1. jasmine.any

- ❏ O “jasmine.any” serve para verificar se um valor é de um determinado tipo
- ❏ Pode ser usado para comparar valores reais ou valores utilizado em spies
- ❏ Em spies é muito útil quando deseja verificar se um método foi chamado com um argumento de determinado tipo, sem se importar com o seu valor real

---

Exemplo:

```
describe("Suíte de testes do tópico 12.1", function() {

  var somar;

  beforeEach(function() {
        somar = jasmine.createSpy("somar");
  });

  it("deve validar o uso do 'jasmine.any'", function() {
    somar(1, 9);
    expect(somar).toHaveBeenCalledWith(
      jasmine.any(Number), jasmine.any(Number));
    expect({}).toEqual(jasmine.any(Object));
    expect("Texto").toEqual(jasmine.any(String));
  });
});
```

12.2. jasmine.anything

- O “jasmine.anything” serve para verificar se um objeto ou variável é diferente de “null” ou “undefined”
- Ele é muito similar ao “jasmine.any”, com a diferença de que ele não verifica o tipo do objeto ou variável
- Pode ser utilizado em spies para certificar que um valor foi passado como parâmetro

---

Exemplo:

```
describe("Suíte de testes do tópico 12.2", function() {

  var somar;

  beforeEach(function() {
        somar = jasmine.createSpy("somar");
  });

  it("deve validar o uso do 'jasmine.anything'", function() {
        somar(1, 9);
        expect(somar).toHaveBeenCalledWith(
              jasmine.anything(), jasmine.anything());
    expect({}).toEqual(jasmine.anything());
  });
});
```

12.3. jasmine.objectContaining

- ❏ O “jasmine.objectContaining” serve para verificar se determinada entrada (chave ou valor) existem em um objeto
- ❏ Ele recebe como parâmetro o bloco a ser verificado em um objeto

---

Exemplo:

```
describe("Suíte de testes do tópico 12.3", function() {

  var carro;

  beforeEach(function() {
          carro = {
      'movido': 'gasolina',
      'ano': 2016
    };
  });

  it("deve validar o uso do 'jasmine.objectContaining'", function() {
    expect(carro).toEqual(jasmine.objectContaining(
      { 'movido': 'gasolina' }));
    expect(carro).toEqual(jasmine.objectContaining(
      { 'ano': 2016 }));
    expect(carro).not.toEqual(jasmine.objectContaining(
      { 'portas': 5 }));
  });
});
```

12.4. jasmine.arrayContaining

- ❏ O “jasmine.arrayContaining” serve para verificar se determinados valores existem em uma array
- ❏ Ele recebe como parâmetro uma array com o valores a serem verificados

---

Exemplo:

```
describe("Suíte de testes do tópico 12.4", function() {

  var numeros;

  beforeEach(function() {
    numeros  = [1, 2, 3, 5, 7, 11];
  });

  it("deve validar o uso do 'jasmine.arrayContaining'", function() {
    expect(numeros).toEqual(jasmine.arrayContaining([1, 3, 5]));
    expect(numeros).toEqual(jasmine.arrayContaining([11]));
    expect(numeros).not.toEqual(jasmine.arrayContaining([1, 4]));
  });
});
```

12.5. jasmine.stringMatching

- ❏ O “jasmine.stringMatching” serve para verificar por uma porção de texto dentro de uma string
- ❏ Ele suporta o uso de expressões regulares
- ❏ Também pode ser utilizado com spies
- ❏ Recebe como parâmetro a porção de texto ou expressão regular utilizada no teste

---

Exemplo:

```
describe("Suíte de testes do tópico 12.5", function() {

  var exibirTexto;

  beforeEach(function() {
        exibirTexto = jasmine.createSpy("exibirTexto");
  });

  it("deve validar o uso do 'jasmine.stringMatching'", function() {
    exibirTexto("Fulano de Tal");
    expect(exibirTexto).toHaveBeenCalledWith(
      jasmine.stringMatching("Fulano"));
    expect({ 'tipo': 'gasolina' }).toEqual(
      { 'tipo': jasmine.stringMatching('gasolina') });
    expect("Fulano de Tal").toEqual(jasmine.stringMatching(/^fulano/i));
  });
});
```

13. Jasmine Clock

- ❏ Serve para tornar síncrono as chamadas do “setTimeout” e “setInterval”
- ❏ Deve ser instalado antes da chamada com “jasmine.clock().install”
- ❏ Deve ser removido ao término do teste / suíte com “jasmine.clock().uninstall”
- ❏ Executa a operação de chamada com “jasmine.clock().tick”, recebendo como parâmetro o número de milissegundos

---

Exemplo:

```
describe("Suíte de testes do tópico 13", function() {

  var somar;

  beforeEach(function() {
        somar = jasmine.createSpy("somar");
    jasmine.clock().install();
  });

  afterEach(function() {
    jasmine.clock().uninstall();
  });

  it("deve validar a chamada do 'setTimeout' síncronamente", function() {
    setTimeout(function() { somar(); }, 200);
    jasmine.clock().tick(100);
    expect(somar).not.toHaveBeenCalled();
    jasmine.clock().tick(200);
    expect(somar).toHaveBeenCalled();

  });

  it("deve validar a chamada do 'setInterval' síncronamente", function() {
    setInterval(function() { somar(); }, 100);
    jasmine.clock().tick(100);
    expect(somar).toHaveBeenCalled();
  });
});
```

14. Criando um comparador personalizado

- ❏ Jasmine permite a criação de um comparador próprio
- ❏ Um objeto com uma função que recebe dois argumentos deve ser criada
- ❏ Os argumentos recebidos são “util” e “customEqualityTesters”
- ❏ Uma função “compare” deve ser definida, e ela deverá retornar um objeto com a propriedade booleana “pass”
- ❏ Uma propriedade “message” contendo a mensagem de erro pode ser adicionada ao objeto de retorno para detalhar a causa da falha
- ❏ O novo comparador deve ser registrado antes do teste através do objeto “jasmine.addMatchers”

Exemplo de comparador:

```
var meuMatcher = {

 toBeValidEmail: function(util, customEqualityTesters) {
  var emailRegex = /\S+@\S+\.\S+/;

  return {
   compare: function(actual, expected) {
    var result = {};
    if (expected === undefined) {
     result.pass = emailRegex.test(actual);
    } else {
     result.pass = expected.test(expected);
    }
    if (result.pass) {
     result.message = actual + " é um e-mail válido";
    } else {
     result.message = "Esperado que " + actual +
      " seja um e-mail válido";
    }
    return result;
   }
```

```
    }
  }
};
```

Exemplo da utilização do comparador:

```
describe("Suíte de testes do tópico 14", function() {

  beforeEach(function() {
    jasmine.addMatchers(meuMatcher);
  });

  it("deve testar o uso do matcher 'toBeValidEmail'", function() {
    expect("email@email.com").toBeValidEmail();
    expect("email@email.com").toBeValidEmail(/\S+@\S+\.\S+/);
    expect("email").not.toBeValidEmail();
  });
});
```

15. Conclusão

Conforme apresentado, todas as principais referências do framework Jasmine foram descritas, assim como um código de exemplo foi adicionado para facilitar o entendimento e funcionamento de cada item.

O conteúdo apresentado serve como base e referência para o Curso de Testes Unitários com Javascript, onde todo o conteúdo aqui apresentado é detalhado e explicado em vídeos, assim como uma série de outros recursos são apresentados, dentre eles:

- ❏ BDD - Desenvolvimento Guiado por Comportamentos
- ❏ Versionamento de código fonte com Git e GitHub
- ❏ Integração Contínua com Travis CI
- ❏ Processo de desenvolvimento completo utilizando testes como base

Como próximos passos recomendo você conferir esse curso, e se tornar um profissional completo e diferenciado quanto a boas práticas em desenvolvimento orientado a testes.

Também não deixe de conferir o curso gratuito de testes unitários com Angular 2, onde você utilizará os conceitos aqui aprendidos para a criação de testes unitários no poderoso framework Angular 2.

Boa sorte e espero te ver em breve!

Deseja acelerar o seu aprendizado e aprender testes unitários com Javascript rapidamente?

Com o curso de Testes unitários com Javascript será possível!

udemy

Veja as vantagens:

- Você se tornará um profissional completo em teste de software
- Aprenderá a criar testes unitários com Jasmine
- Aprenderá automatizar testes com Karma
- Versionamento de código fonte com Git/GitHub
- Integração contínua com Travis CI

Veja suas garantias:

- Curso disponível na plataforma Udemy
- Mais de 9 horas de aula em vídeo
- Assista as aulas de seu computador ou smartfone
- Garantia de satisfação ou seu dinheiro de volta
- Certificado de conclusão de curso emitido pela Udemy
- Acesso total vitalício ao conteúdo do curso

Acesse através do link abaixo para ganhar um desconto de 30%!

Curso de Testes Unitários com Javascript