

Aplicações do *framework Jest* e boas práticas de *unit testing* em *Javascript*

◀ Mateus Mendonça Dias Rezende

Por quê *Jest*?



- Simplicidade
- Excelente documentação
- Totalmente *Open-source*
- Suporta diversas tecnologias *Javascript*:
 - ***Back-end Node.js***
 - **Bibliotecas de componentes *Front-end* (Angular, Vue, React)**
 - ***Runtime* em navegadores (ECMAScript)**
 - **Suporte para *Typescript***

Jest agora faz parte da *OpenJS*



Desde o dia 11/05/22, o projeto Jest deixou de fazer parte da Meta e se tornou propriedade da fundação OpenJS, como projeto de Impacto.

Como são os testes usando *Jest*?

- A forma mais comum de declarar um teste é através de *callbacks*, da forma:

```
test("Nome do teste", () => {  
  // Validações aqui  
})
```

- Dentro das chamadas de função, podem ser feitas várias verificações utilizando todos os recursos da linguagem: *loops*, uso de bibliotecas de terceiros, etc.
- O escopo das funções anônimas garante que os testes serão isolados entre si.

Matchers comuns

- Assim como no *JUnit*, existem diversas verificações que definem o valor de retorno do teste:

Por exemplo: *expect.toBe* checa por uma correspondência exata:

```
test("2 * 5 deveria ser 10", () => {  
    expect(2 * 5)  
        .toBe(10)  
})
```

- **expect:**

- o `toEqual(objeto)`, para correspondências recursivas
- o `toBeNull()`
- o `toBeUndefined()`
- o `toBeGreaterThan(número)`
- o `toBeGreaterThanOrEqual(número)`
- o `toBeLessThan(número)`
- o `toBeLessThanOrEqual(número)`
- o `toBeCloseTo(número, número_de_dígitos)`, para valores próximos e ponto flutuante
- o `toMatch(regex)`, para Strings
- o `toContain(item)`, para objetos iteráveis (arrays, conjuntos, etc.)
- o `toThrow()`, para entradas que devem retornar algum erro
- o `toThrow(exception)`, para entradas que devem retornar um erro específico
- o [...] e vários outros testes

Integração com Editores de texto

- O Jest, por estar disponível como um pacote *Node.js*, pode ser utilizado diretamente pelo terminal, ou pelo próprio editor de texto ou IDE:

```
PS C:\Users\Mateus Rezende\College\jest> npm run test

> test
> jest --coverage

PASS tests/fatorial.test.ts
PASS tests/grausParaRadianos.test.ts
PASS tests/cos.test.ts
PASS tests/taylor.test.ts
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.23	85.71	100	100	
functions.ts	95.23	85.71	100	100	36

```
Test Suites: 4 passed, 4 total
Tests: 16 passed, 16 total
Snapshots: 0 total
Time: 5.198 s
Ran all test suites.
```

```
jest (auto-run-watch)
├─ tests
│  └─ cos.test.ts
│     ├── Cosseno de 0 rad deveria ser 1
│     ├── Cosseno de 0 graus deveria ser 1
│     ├── Cosseno de 60 graus deveria ser próximo de 0.5
│     └─ Cosseno de pi rad deveria ser próximo de -1
├─ fatorial.test.ts
│  ├── Fatorial de 0 deveria ser igual a 1
│  ├── Fatorial de 1 deveria ser igual a 1
│  ├── Fatorial de 2 deveria ser igual a 2
│  ├── Fatorial de 3 deveria ser igual a 6
│  └─ Fatorial de 5 deveria ser igual a 5 * 4 * 3 * 2 * 1
├─ grausParaRadianos.test.ts
│  ├── 0 graus deveria ser igual a 0 radianos
│  ├── 180 graus deveria ser igual a pi radianos
│  └─ 270 graus deveria ser igual a 3pi/2 radianos
└─ taylor.test.ts
   ├── O primeiro termo (0) do polinômio deveria ser 1, para qualquer ângulo
   ├── Todos os termos após o primeiro devem ser 0, caso o ângulo seja 0
   ├── O segundo termo (1) do polinômio deveria ser -1 * ângulo²/2
   └─ O terceiro termo (2) do polinômio deveria ser ângulo**4/(4*3*2*1)
```




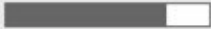




A controvérsia do *Test coverage* 100%

- Além de ser fácil de configurar, em alguns casos dispensando qualquer configuração, o *Jest* gera um relatório da taxa de código testada no projeto atual:

All files

80.49% Statements 524/651 55.75% Branches 97/174 67.76% Functions 124/183 79.02% Lines 437/553

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File ▲		Statements ▾		Branches ▾		Functions ▾		Lines ▾	
src		100%	15/15	100%	0/0	100%	1/1	100%	15/15
src/app		100%	36/36	100%	0/0	100%	8/8	100%	27/27
src/app/auth		60.8%	76/125	25%	9/36	37.14%	13/35	58.77%	67/114
src/app/common		79.55%	70/88	26.92%	7/26	66.67%	12/18	78.21%	61/78
src/app/home		100%	14/14	100%	0/0	100%	5/5	100%	11/11
src/app/inventory		100%	6/6	100%	0/0	100%	3/3	100%	4/4
src/app/inventory/categories		100%	6/6	100%	0/0	100%	3/3	100%	4/4

Vantagens da metodologia

- Toda linha de código é executada, ao menos uma vez
- O código tem uma consistência desejável quando se trata de alterações de código já existente
- É requisito em algumas equipes de desenvolvimento

Desvantagens

- Sensação de falsa segurança, visto que testes unitários não garantem que o código testado funciona, nem que é eficiente e muito menos que faz sentido
- Mais tempo gasto escrevendo testes redundantes/desnecessários só para completar 100%
- Ter **bons** testes unitários concentrados em funcionalidades mais complexas é melhor do que testes simples em todo o projeto

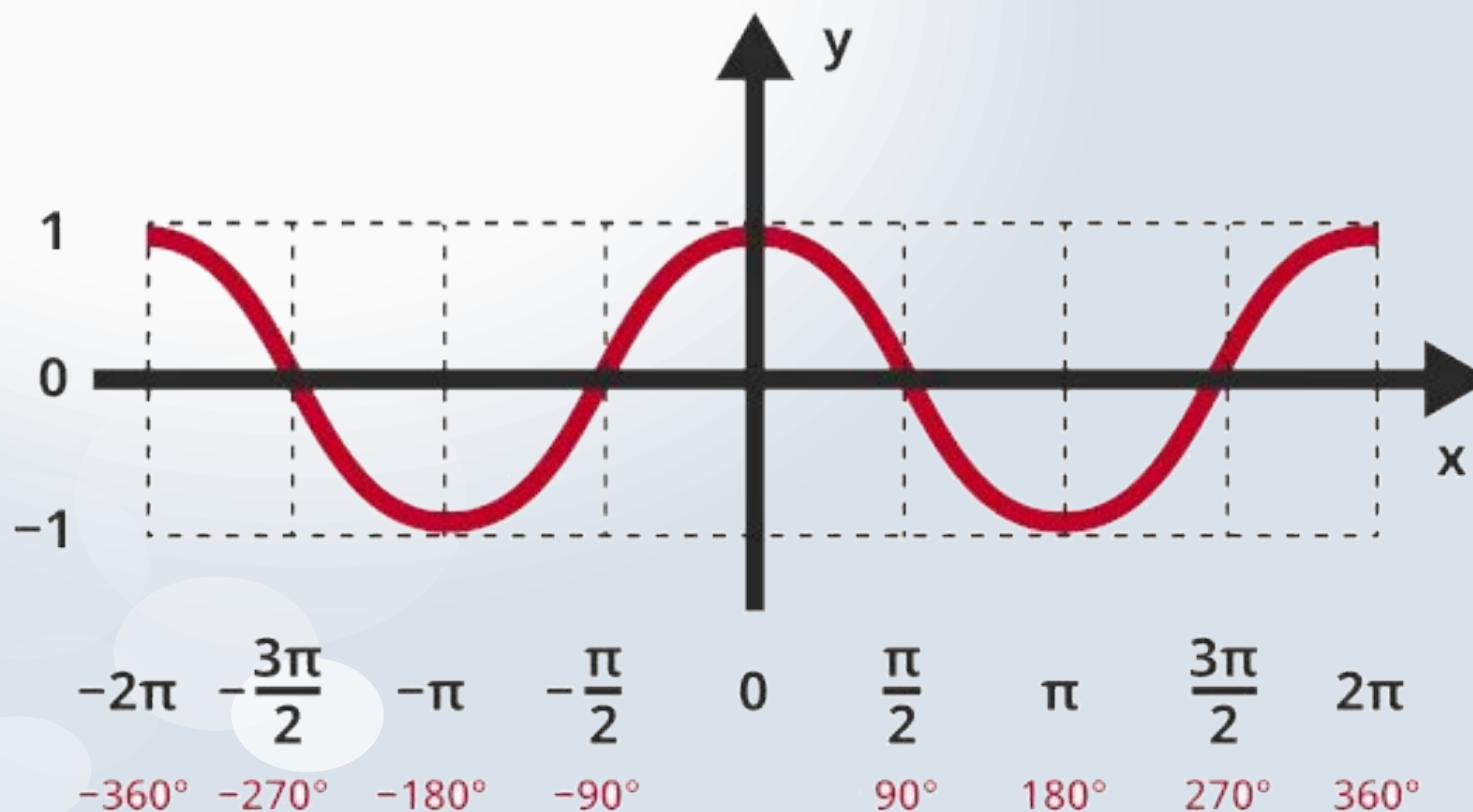
Afinal, o que são bons testes unitários?

- Bons testes unitários devem:
 - o Ser isolados (funções puras, portanto, sem efeitos colaterais);
 - o Ser determinísticos (sempre retornam o mesmo resultado, portanto não podem conter valores pseudoaleatórios ou que dependam da data, acesso a disco, requisições *HTTP*, etc.);
 - o Ignorar a implementação das funções/métodos/classes que testam, separando-a da tarefa que estes devem realizar;
 - o Ser específicos (testam um único caso ou um conjunto de casos semelhantes);
 - o Incluir casos triviais e valores inesperados (*null*, *undefined*, tipos não permitidos, etc.);
 - o Ter nomes intuitivos, que implicam qual caso é testado;

Caso de uso: função de estimativa do cosseno

```
cos(ângulo: number, graus?: boolean, erro?:  
number): number
```

Ângulo considerado no cálculo



Caso de uso: função de estimativa do cosseno

- Dada a função cosseno, independente de qual seja a implementação, podemos fazer as seguintes verificações:

```
test("Cosseno de 0 rad deveria ser 1", () => {  
  expect(cos(0)).toBe(1)  
})  
  
test("Cosseno de 0 graus deveria ser 1", () => {  
  expect(cos(0, true)).toBe(1)  
})  
  
test("Cosseno de 60 graus deveria ser próximo de 0.5", () => {  
  expect(cos(60, true)).toBeCloseTo(0.5, 4)  
})  
  
test("Cosseno de pi rad deveria ser próximo de -1", () => {  
  expect(cos(Math.PI)).toBeCloseTo(-1, 4)  
})
```

Caso de uso: função de estimativa do cosseno

$$\begin{aligned}\cos(x) &= \frac{d}{dx} \sin(x) \\ &= \frac{d}{dx} \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \\ &= \frac{d}{dx} \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \right) \\ &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \\ &= \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!}\end{aligned}$$

```
/**
 *
 * @param x Valor do ângulo
 * @param n Termo do polinômio
 * @returns N-ésimo termo do polinômio de Taylor da
 * função cos(x) em torno do ponto x = 0
 */
export const taylorCosseno = (x: number, n: number) => {
  return (-1)**(n) * x**(2*n) / fatorial(2*n);
}
```

Futuro do *Jest*

- Com uma comunidade ativa, o *Jest* tende a permanecer como uma confiável alternativa de *framework* de testes unitários em *Javascript*.
- O *framework* tende a dar suporte às soluções que venham a surgir, como já acontece nas integrações com bibliotecas *UI* e no suporte a *Babel/Typescript*.
- Por já ser utilizado em diversas aplicações grandes que estão em produção, vagas que buscam desenvolvedores que tenham familiaridade com *Jest* tende a aumentar. Além disso, esta demanda já existe, tanto tendo tal habilidade como requisito quanto como atributo desejável.

Referências

- *Jest Team. Jest Docs: Getting Started*. 28.1. 2022. Disponível em: <<https://jestjs.io/docs/getting-started>>. Acesso em: 08 jun. 2022.
- Pedro Lopes. Sobre Desenvolvimentos em Séries de Potências, Séries de Taylor e Fórmula de Taylor. Única. 2006. Disponível em: <<https://www.math.tecnico.ulisboa.pt/~pelopes/taylor05.pdf>>. Acesso em: 10 jun. 2022.