

Teste Unitários e de Integração com Jest - Maiself

Introdução

Com o passar o tempo a tecnologia vem avançando, com isso a produção de softwares em larga escala também. Cada vez mais a demanda por softwares aumentam, cada vez mais os softwares vem se tornando mais inteligentes. Se formos comparar um “*software bem antigo*” a um que tenhamos atualmente, podemos notar algumas coisas, vamos citar o software que compus a *Calculadora de Pascal*, para hoje podemos o considerar simples, sem muitas complicações, até aí tudo bem, mas como Pascal constatou que sua calculadora estava correta? Como ele concluiu que ela funcionava corretamente? A resposta parece ser meio lógica, testando! Mas quando paramos para analisar, o ato/atividade de testar é complexo, hoje a Engenharia de Software prevê ao mínimo dois tipos de testes os de **Caixa Branca** e os de **Caixa Preta**, mas em que se diferem? Resumidamente: O de **Caixa Branca** basicamente é os testes feitos a nível de código, de desenvolvimento mesmo, onde o testador tem conhecimento da linguagem/tecnologia utilizada e vai testando as partes/módulos das implementações a fim de buscar falhas de implementações incorretas, faltosas e etc. O de **Caixa Preta** é o inverso, para o mesmo não necessita ter conhecimento técnico sobre o projeto, linguagem ou tecnologia, mas sim, dos requisitos e necessidades do usuário, assim executando ao invés de um teste de código um teste de sistema. Muito provavelmente Pascal fez os testes de sua calculadora a mão mesmo, imaginava/calculava uma operação, a repassava a calculadora e ficava a observar se o resultado dela era igual ao que ele já sabia/tinha calculado. Todos testes manuais e demorados.

Hoje os softwares são muito maiores e complexos que a *Calculadora de Pascal*, nos referimos desta forma quando nos tratamos a respeito de robustez e complexidade, assim testa de forma manual como *Pascal* fez, se torna inviável e muito mais custoso, um projeto que seja testado dessa forma, irá elevar seu valor final muitas vezes a cima do comum/normal/esperado que fosse, mas como testar softwares hoje? Existe inúmeras formas, mas uma das mais eficientes e populares são os testes automatizados, e como funcionam? Basicamente podem ser escritos na mesma linguagem utilizada pelo projeto, pois a linguagem mesmo terá sua biblioteca para tal, assim os testes são escritos de forma automatizada em *scripts* de testes e executados sob as implementações. Nesse caso estamos nos referindo aos testes de **Caixa Branca**.

Descrição

Como a **Maiself**, optou por desenvolver a plataforma em questão imersa no mundo *Javascript/Typescript*, o mecanismo/ferramenta/biblioteca de testes pelo qual optamos de fazer o uso para implementar os testes automatizados e testar de forma automatizada nossa aplicação, foi o *Jest*. Como o *Jest* mesmo descreve em sua [Home Page](#) “*Jest é um poderoso Framework de Testes em Javascript com um foco na simplicidade*”, mas onde pode-se usar *Jest* para testes? O mesmo pode ser usado com: *Babel*, *Typescript*, *Node*, *React*, *Angular*, *Vue* entre muitos outros. O *Jest* é um Framework ou biblioteca como preferir chamar, implementado/produzido em *Javascript*, o mesmo promove ou dar menção ao uso tanto para testes unitários quando de integração, de maneira geral o seu uso é bem simples e prático, se formos comparar de forma bem esdruxula o mesmo chega a se parecer bastante com o *JUnit* do *Java*.

Entenda de forma rápida o funcionamento do Jest

Como forma de ilustração básica do que é o *Jest*, implementamos um calculadora simples em *Node* e uma bateria de testes para a mesma. Veja:

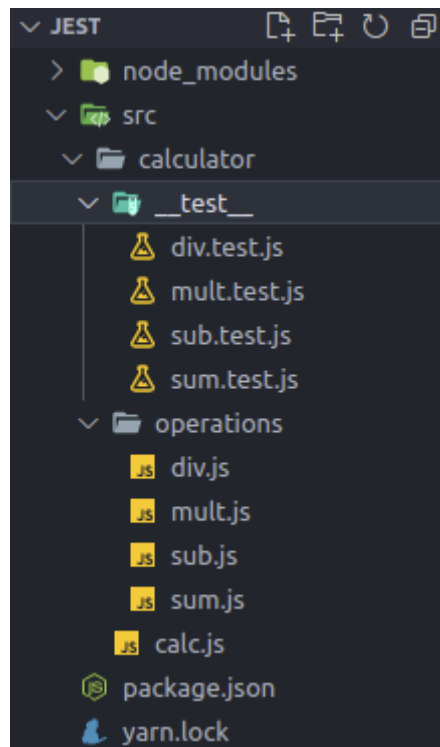


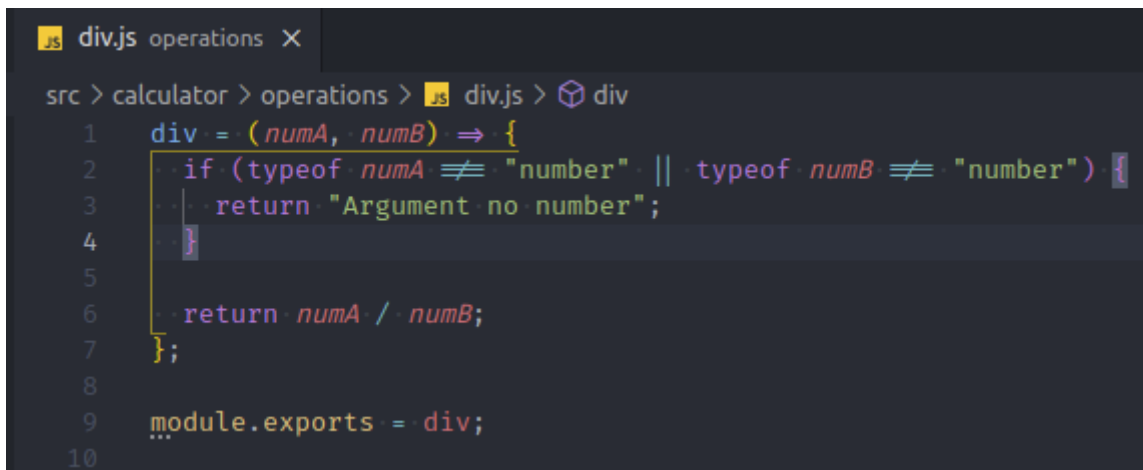
Imagem 01: Arquitetura do projeto de exemplificação dos teste em Jest

Essa é a arquitetura básica de nossa calculadora, a *main* é o arquivo designado como "*calc.js*", as funções da calculadora estão implementadas no subdiretório *operations* da calculadora e dentro do mesmo, estão dispostos os arquivos: *div.js*, *mult.js*, *sub.js* e *sum.js*, que dizem respeito a implementação da divisão, multiplicação, subtração e soma respectivamente.

```
src > calculator > calc.js > ...
1  const sum = require("./operations/sum");
2  const div = require("./operations/div");
3  const mult = require("./operations/mult");
4  const sub = require("./operations/sub");
5
6  console.log("sum => " + sum(1, 1));
7  console.log("div => " + div(1, 1));
8  console.log("multi => " + mult(1, 1));
9  console.log("sub => " + sub(1, 1));
10
```

Imagem 02: Arquivo *main* da calculadora o *calc.js*

Aqui podemos ver o arquivo *main* o *calc.js* da calculadora, um arquivo bem simples apenas para termos a certeza que as funções foram implementadas, o arquivo faz a requisição a cada módulo da calculadora, pois os mesmos foram implementados separadamente.



```

src > calculator > operations > JS div.js > div
1  div = (numA, numB) => {
2    if (typeof numA !== "number" || typeof numB !== "number") {
3      return "Argument no number";
4    }
5
6    return numA / numB;
7  };
8
9  module.exports = div;
10

```

Imagem 03: Arquivo/Módulo de divisão da calculadora o *div.js*

Aqui temos a implementação do módulo de divisão da calculadora, um módulo bem simples, onde o mesmo apenas receber dois parâmetros, se os dois ou um dos mesmos não for um número ele irá retornar uma mensagem de erro, caso o contrário, ele irá retornar o resultado da operação.



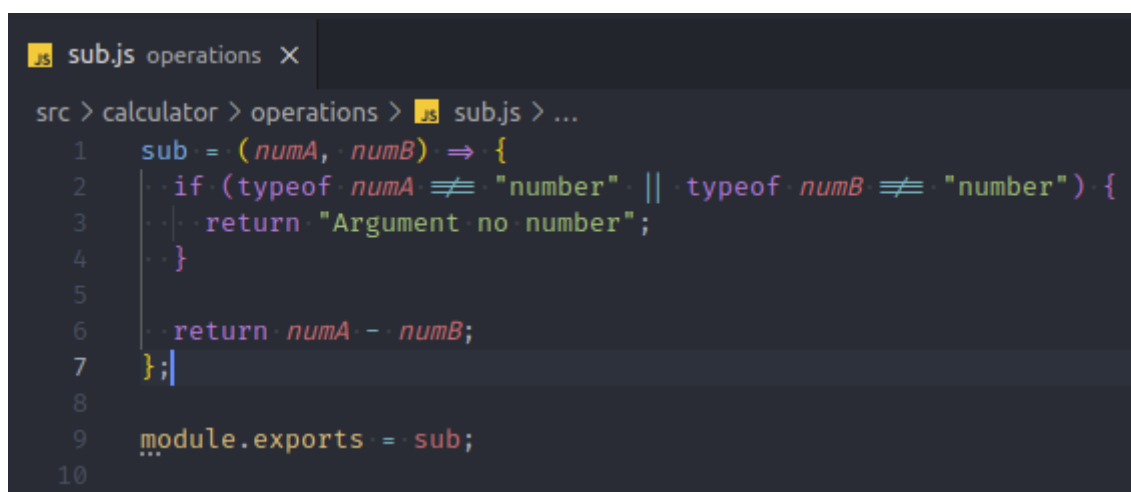
```

src > calculator > operations > JS mult.js > mult
1  mult = (numA, numB) => {
2    if (typeof numA !== "number" || typeof numB !== "number") {
3      return "Argument no number";
4    }
5
6    return numA * numB;
7  };
8
9  module.exports = mult;
10

```

Imagem 04: Arquivo/Módulo de multiplicação da calculadora o *mult.js*

Aqui temos a implementação do módulo de multiplicação da calculadora, um módulo bem simples, onde o mesmo apenas receber dois parâmetros, se os dois ou um dos mesmos não for um número ele irá retornar uma mensagem de erro, caso o contrário, ele irá retornar o resultado da operação.



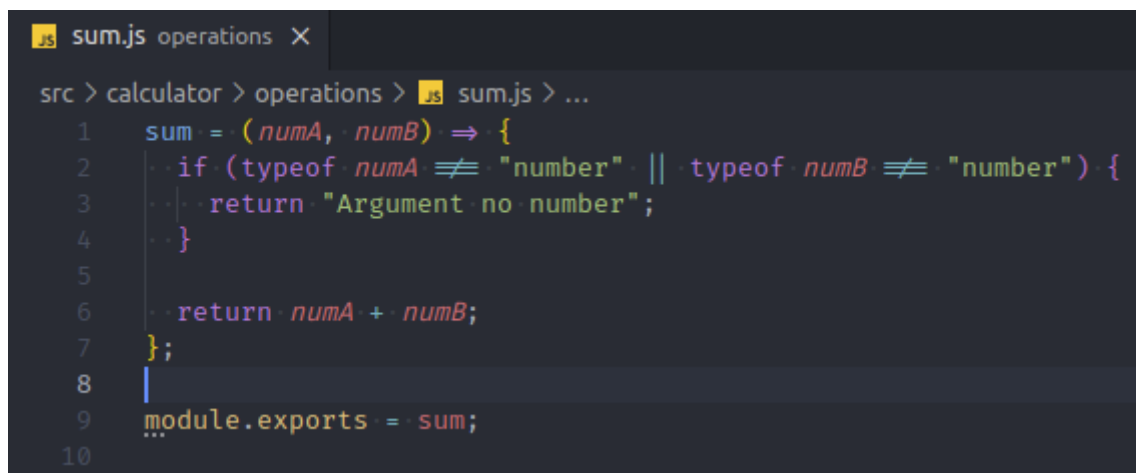
```

src > calculator > operations > JS sub.js > ...
1  sub = (numA, numB) => {
2    if (typeof numA !== "number" || typeof numB !== "number") {
3      return "Argument no number";
4    }
5
6    return numA - numB;
7  };
8
9  module.exports = sub;
10

```

Imagem 05: Arquivo/Módulo de subtração da calculadora o *sub.js*

Aqui temos a implementação do módulo de subtração da calculadora, um módulo bem simples, onde o mesmo apenas receber dois parâmetros, se os dois ou um dos mesmos não for um número ele irá retornar uma mensagem de erro, caso o contrário, ele irá retornar o resultado da operação.



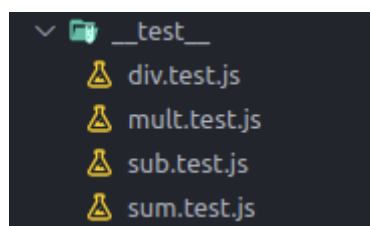
```
src > calculator > operations > JS sum.js > ...
1  sum = (numA, numB) => {
2    if (typeof numA !== "number" || typeof numB !== "number") {
3      return "Argument no number";
4    }
5
6    return numA + numB;
7  };
8
9  module.exports = sum;
10
```

Imagem 06: Arquivo/Módulo de soma da calculadora o *sum.js*

Aqui temos a implementação do módulo de soma da calculadora, um módulo bem simples, onde o mesmo apenas receber dois parâmetros, se os dois ou um dos mesmos não for um número ele irá retornar uma mensagem de erro, caso o contrário, ele irá retornar o resultado da operação.

Ambos os módulos são muito semelhantes, apenas com a diferença do retorno de implementação da operação no qual ele reflete mesmo. Mas como testar essa nossa calculadora mesmo? Diferente de *Pascal*, que fez testes manuais, nos iremos automatizar nossos testes! Como? Com *Jest*!

Dentro da estrutura desse nosso projeto da calculadora foi mostrado um diretório chamado “__test__”, essa nomenclatura já vem dos padrões de uso do *Jest* no *Node*, é nesse local que está contido nossa bateria de testes, ou como chama o *Jest*, nossa *Suite*, veja:



```
__test__
├── div.test.js
├── mult.test.js
├── sub.test.js
└── sum.test.js
```

Imagem 07: Diretório referente aos scripts de teste

Aqui estão descritos as rotinas de testes para cada uma das quatro operações de nossa calculadora, assim para cada um dos módulos desse nosso projeto. Iremos exibir a baixo de forma sequencial os quatro módulos de testes, veja:

```

div.test.js __test__ x
src > calculator > __test__ > div.test.js > ...
1  const div = require("../operations/div");
2
3  test("dividir 1 / 3 dever retornar 0.3333333333333333", () => {
4    expect(div(1, 3)).toBe(0.3333333333333333);
5  });
6
7  test("dividir 0 / 1 dever retornar 0", () => {
8    expect(div(0, 1)).toBe(0);
9  });
10
11 test("dividir -1 / 1 dever retornar -1", () => {
12   expect(div(-1, 1)).toBe(-1);
13 });
14
15 test("dividir -2 / 1 dever retornar -2", () => {
16   expect(div(-2, 1)).toBe(-2);
17 });
18
19 test("dividir a / 1 dever retornar Argument no number", () => {
20   expect(div("a", 1)).toBe("Argument no number");
21 });
22
23 test("dividir 10 / b dever retornar Argument no number", () => {
24   expect(div(10, "b")).toBe("Argument no number");
25 });
26
27 test("dividir a / b dever retornar Argument no number", () => {
28   expect(div("a", "b")).toBe("Argument no number");
29 });
30
31 test("dividir 10.2 / 0.8 dever retornar 12.749999999999998", () => {
32   expect(div(10.2, 0.8)).toBe(12.749999999999998);
33 });
34
35 test("dividir 0.00001 / 0.00009 dever retornar 0.1111111111111112", () => {
36   expect(div(0.00001, 0.00009)).toBe(0.1111111111111112);
37 });
38
39 test("dividir 0.00001 / 0.99999 dever retornar 0.00001000010000100001", () => {
40   expect(div(0.00001, 0.99999)).toBe(0.00001000010000100001);
41 });
42

```

Imagem 08: Arquivo/Módulo de testes referente ao módulo divisão da calculadora o *div.test.js*

```

mult.test.js __test__ x
src > calculator > __test__ > mult.test.js > ...
1  const mult = require("../operations/mult");
2
3  test("multiplicar 1 * 3 dever retornar 3", () => {
4    expect(mult(1, 3)).toBe(3);
5  });
6
7  test("multiplicar 0 * 1 dever retornar 0", () => {
8    expect(mult(0, 1)).toBe(0);
9  });
10
11 test("multiplicar -1 * 1 dever retornar -1", () => {
12   expect(mult(-1, 1)).toBe(-1);
13 });
14
15 test("multiplicar -2 * 1 dever retornar -2", () => {
16   expect(mult(-2, 1)).toBe(-2);
17 });
18
19 test("multiplicar a * 1 dever retornar Argument no number", () => {
20   expect(mult("a", 1)).toBe("Argument no number");
21 });
22
23 test("multiplicar 10 * b dever retornar Argument no number", () => {
24   expect(mult(10, "b")).toBe("Argument no number");
25 });
26
27 test("multiplicar a * b dever retornar Argument no number", () => {
28   expect(mult("a", "b")).toBe("Argument no number");
29 });
30
31 test("multiplicar 10.2 * 0.8 dever retornar 8.16", () => {
32   expect(mult(10.2, 0.8)).toBe(8.16);
33 });
34
35 test("multiplicar 0.00001 * 0.00009 dever retornar 9.000000000000001e-10", () => {
36   expect(mult(0.00001, 0.00009)).toBe(9.000000000000001e-10);
37 });
38
39 test("multiplicar 0.00001 * 0.99999 dever retornar 0.0000099999", () => {
40   expect(mult(0.00001, 0.99999)).toBe(0.0000099999);
41 });
42

```

Imagem 09: Arquivo/Módulo de testes referente ao módulo multiplicação da calculadora o mult.test.js

sub.test.js __test__ X

src > calculator > __test__ > sub.test.js > test("subtrair -1 - 1 dever retornar -2") callback

```
1  const sub = require("../operations/sub");
2
3  test("subtrair 1 - 3 dever retornar -2", () => {
4    expect(sub(1, 3)).toBe(-2);
5  });
6
7  test("subtrair 0 - 1 dever retornar -1", () => {
8    expect(sub(0, 1)).toBe(-1);
9  });
10
11 test("subtrair -1 - 1 dever retornar -2", () => {
12   expect(sub(-1, 1)).toBe(-2);
13 });
14
15 test("subtrair -2 - 1 dever retornar -3", () => {
16   expect(sub(-2, 1)).toBe(-3);
17 });
18
19 test("subtrair a - 1 dever retornar Argument no number", () => {
20   expect(sub("a", 1)).toBe("Argument no number");
21 });
22
23 test("subtrair 10 - b dever retornar Argument no number", () => {
24   expect(sub(10, "b")).toBe("Argument no number");
25 });
26
27 test("subtrair a - b dever retornar Argument no number", () => {
28   expect(sub("a", "b")).toBe("Argument no number");
29 });
30
31 test("subtrair 10.2 - 0.8 dever retornar 9.399999999999999", () => {
32   expect(sub(10.2, 0.8)).toBe(9.399999999999999);
33 });
34
35 test("subtrair 0.00001 - 0.00009 dever retornar -0.00008", () => {
36   expect(sub(0.00001, 0.00009)).toBe(-0.00008);
37 });
38
39 test("subtrair 0.00001 - 0.99999 dever retornar -0.9999800000000001", () => {
40   expect(sub(0.00001, 0.99999)).toBe(-0.9999800000000001);
41 });
42
```

Imagem 10: Arquivo/Módulo de testes referente ao módulo subtração da calculadora o *sub.test.js*

```
sum.test.js __test__ X
src > calculator > __test__ > sum.test.js > test("somar a + 1 dever retornar Argument no number")
1  const sum = require("../operations/sum");
2
3  test("somar 1 + 3 dever retornar 4", () => {
4    expect(sum(1, 3)).toBe(4);
5  });
6
7  test("somar 0 + 1 dever retornar 1", () => {
8    expect(sum(0, 1)).toBe(1);
9  });
10
11 test("somar -1 + 1 dever retornar 0", () => {
12   expect(sum(-1, 1)).toBe(0);
13 });
14
15 test("somar -2 + 1 dever retornar -1", () => {
16   expect(sum(-2, 1)).toBe(-1);
17 });
18
19 test("somar a + 1 dever retornar Argument no number", () => {
20   expect(sum("a", 1)).toBe("Argument no number");
21 });
22
23 test("somar 10 + b dever retornar Argument no number", () => {
24   expect(sum(10, "b")).toBe("Argument no number");
25 });
26
27 test("somar a + b dever retornar Argument no number", () => {
28   expect(sum("a", "b")).toBe("Argument no number");
29 });
30
31 test("somar 10.2 + 0.8 dever retornar 11.0", () => {
32   expect(sum(10.2, 0.8)).toBe(11.0);
33 });
34
35 test("somar 0.00001 + 0.00009 dever retornar 0.0001", () => {
36   expect(sum(0.00001, 0.00009)).toBe(0.0001);
37 });
38
39 test("somar 0.00001 + 0.99999 dever retornar 1.0000", () => {
40   expect(sum(0.00001, 0.99999)).toBe(1.0);
41 });
42
```

Imagem 11: Arquivo/Módulo de testes referente ao módulo soma da calculadora o *sum.test.js*

No início de cada módulo de testes, fazemos a requisição do uso ao módulo que queremos testar, isso é feito em toda linha 1 de cada módulo como descrito nas imagens acima, todo módulo/operação da calculadora passou por 10 testes, dos mais variados afim de buscar quebrar a implementação da calculadora. Todo teste tem sua descrição, nesse nosso caso adotamos o padrão na frase do teste ao início a operação/módulo no qual se testa, em seguida os dois valores repassados ao módulo para o teste e ao fim o retorno

esperado com base em cálculos matemáticos previamente produzidos. Veja a explicação detalhada de um teste:

```
test("somar 1 + 3 dever retornar 4", () => {  
  expect(sum(1, 3)).toBe(4);  
});
```

Imagem 12: Exemplificação de um teste de nossa calculadora, referente ao módulo de soma

Aqui destacamos as cores para podermos entender cada argumento, sublinhado em vermelho temos o argumento "test", que é a notação que define um fluxo de teste, tudo que estiver dentro do mesmo faz parte da dada bateria de testes, sublinhado em azul escuro, temos a descrição do "narrada" do teste, no nosso caso adotado o padrão é a operação seguida dos valores da mesma seguida o resultado que se espera da mesma, em verde claro, o argumento "expect" que assim como "test", faz parte da sintaxe do Jest, no qual o mesmo é uma função que dentro se seu parâmetro a funcionalidade que desejamos testes, nesse nosso caso o que está sublinhado em amarelo, é a chamada ao módulo "sum" passando a ele os dois parâmetros para a operação, nesse caso os dois valores (1 e 3) que está sublinhado em rosa, sublinha de azul ciano e o argumento "toBe", que também faz parte da sintaxe do Jest, ele basicamente quer dizer: "expect execute essa função passada para você, mas eu espero isso que você me retorno resultado tal", esse resultado tal é o que está dentro da função "toBe" nesse nosso caso o valor 4, que está sublinhado em laranja. E isso basicamente é um teste do Jest, lógico que o mesmo tem muitos outros argumentos e descritivas, mas esse que foi apresentado é o que nos abatesse a nível de calculadora como essa que implementamos.

Mas o que acontece ao rodarmos os testes, veja:

```
eric in Arquivos/Programacao/Jest  
> yarn test  
yarn run v1.22.5  
$ jest  
PASS src/calculator/__test__/sub.test.js  
PASS src/calculator/__test__/mult.test.js  
PASS src/calculator/__test__/sum.test.js  
PASS src/calculator/__test__/div.test.js  
  
Test Suites: 4 passed, 4 total  
Tests: 40 passed, 40 total  
Snapshots: 0 total  
Time: 1.473 s  
Ran all test suites.  
Done in 7.43s.
```

Imagem 13: Exibição da execução de testes do Jest

Aqui temos o resumo de execução dos nossos testes, temos um total de 4 módulos de testes ou *Suites* como chama o Jest, totalizando 40 testes em nossa calculadora, 10 por *Suite*, nesse caso todos os testes passaram com sucesso (objetivo de todo programador/testador de **Caixa Branca**). Mas, vamos fazer um teste falhar, para vermos como o Jest nos mostra de forma amigável o erro, e isso é muito importante, pois quanto mais amigável e entendível é uma mensagem de erro, melhor e mais fácil é de achar o ponto do erro e o refatorar. Iremos fazer o teste exibido na imagem 12, falhar, iremos o reescrever para que seu "toBe" seja 5 ao invés de 4, e assim o teste é para falhar. Veja a execução dos testes com essa modificação:

```

eric in Arquivos/Programacao/Jest
> yarn test
yarn run v1.22.5
$ jest
FAIL src/calculator/__test__/sum.test.js
  ● somar 1 + 3 dever retornar 4

    expect(received).toBe(expected) // Object.is equality

    Expected: 5
    Received: 4

   2 |
   3 |   test("somar 1 + 3 dever retornar 4", () => {
>  4 |     expect(sum(1, 3)).toBe(5);
      |                       ^
   5 |   });
   6 |
   7 |   test("somar 0 + 1 dever retornar 1", () => {

at Object.<anonymous> (src/calculator/__test__/sum.test.js:4:21)

PASS src/calculator/__test__/sub.test.js
PASS src/calculator/__test__/div.test.js
PASS src/calculator/__test__/mult.test.js

Test Suites: 1 failed, 3 passed, 4 total
Tests:       1 failed, 39 passed, 40 total
Snapshots:   0 total
Time:        0.451 s, estimated 1 s
Ran all test suites.

```

Imagem 14: Exibição da execução de testes do *Jest*, com um teste do módulo de divisão falhando

O *Jest* exibe que a *Suite* referente aos testes do módulo de soma falhou “*FAIL*” em vermelho, abaixo do mesmo ele mostra a descrição do teste em vermelho, essa descrição do teste que falhou diz “somar 1 + 3 deve retornar 4” com texto em vermelho indicando que esse teste da *Suite* de testes de soma falhou, mais abaixo podemos ver que o *Jest* nós diz que, estávamos esperando o valor 5, e a implementação de nossa soma retornou que 1 + 3 é 4, assim o teste falhou, e logo abaixo o *Jest* mostra o teste escrito que falhou, mais abaixo ele mostra que as demais *Suites* tiveram seus testes passados com sucesso, só final ele mostra que das 4 *Suites*, 1 falhou e 3 passaram de um total de 4 *Suites* (lembrando que para o *Jest* uma *Suite* só passa quando todos os testes dela passam, basta 1 não passar para a *Suite* falhar, como podemos ver por esse nosso exemplo), logo mais abaixo podemos ver que 1 teste falhou e 39 passam de um total de 40 testes. E esse foi nosso overview sobre o *Jest*, a ferramenta de teste que o **Maiself** fará uso.