**TDD com a biblioteca Tape em ambiente Node.js**

O TDD (Test Driven Development ou Desenvolvimento Orientado à Testes) é uma técnica criada por Kent Beck, um famoso programador e autor de livros, que permite testar um software antes de tê-lo pronto. Com a aplicação do TDD, os testes são escritos antes do código de produção e isso permite atingir feitos como validar o problema e verificar cenários positivos e negativos. Para praticar o TDD é necessário seguir o fluxo da técnica que é o seguinte: (1) escrever primeiro o teste que irá falhar, (2) em seguida implementar o código de produção que permita o teste passar, (3) refatorar o código de produção implementando a lógica da regra de negócio da aplicação.

No nosso cenário, faremos a aplicação apenas de testes unitários, que são testes de pequenas unidades de código, normalmente classes, as quais seus métodos são testados isoladamente para verificar se o retorno está conforme o esperado. Vale aqui ressaltar que existem outros tipos de testes que não cabem no escopo deste trabalho.

Utilizaremos a linguagem de programação Javascript, o ambiente de desenvolvimento Node.js, a biblioteca de testes Tape e o editor de código VScode. Para seguir os passos deste tutorial, é necessário que já estejam instalados na máquina o Node.js e o VScode. O desenvolvimento será algo trivial, apenas com a finalidade de apresentar o fluxo do TDD.

Com a pasta do projeto criada, e já aberta no vscode, executaremos no terminal integrado da ferramenta os comandos, sem as aspas;

"*npm init -y* ", para criar o arquivo *package.json* na raiz do projeto;

*"npm install --save-dev tape tap-spec",* para instalar a Tape e Tap-spec como bibliotecas de desenvolvimento. O Tap-spec server para estilizar a saída dos testes.

E editaremos o arquivo *package.json*, na seção de scripts.

"scripts": {

    "test": "tape src/tests/\*\*/\*-test.js | tap-spec"

  },

Ao final dos passos acima, o arquivo *package.json* deve ficar assim:

{

  "name": "tape",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "tape src/tests/\*\*/\*-test.js | tap-spec"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "devDependencies": {

    "tap-spec": "^5.0.0",

    "tape": "^5.6.1"

  }

}

Criando a estrutura de pastas.

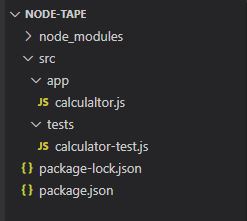
"*mkdir src"* , para criar a pasta *src* do projeto;

*"cd src"*, para acessar a pasta *src;*

*"mkdir app"*, criando dentro de *src* a pasta *app;*

*"mkdir tests"*, criando dentro de *src* a pasta *tests;*

Dentro da pasta *app,* criaremos o arquivo *calculator.js*, e dentro da pasta *tests*, criaremos o arquivo *calculator-test.js*. Segue abaixo imagem da estrutura de pastas e arquivos.



Neste ponto, começaremos escrever primeiro nosso caso de teste numero um, e devemos fazer assim porque esse é o fluxo do TDD.

No arquivo *tests/calculator-test.js*, faremos a importação da biblioteca *tape* e do arquivo *app/calculator.js* que contém a classe *Calculator, a qual* terá os seus métodos testados, e escreveremos nosso primeiro caso de teste, que será a soma de dois números.

// tests/calculator-test.js

const test = require('tape');

const calc = require('./../app/calculator');

test('#Calculator - add - should return the sum of two numbers', async (t) => {

  const result = await calc.add(5,8);

  const expected = 13;

  t.assert(result === expected, 'Calculator add ok.');

  t.end();

});

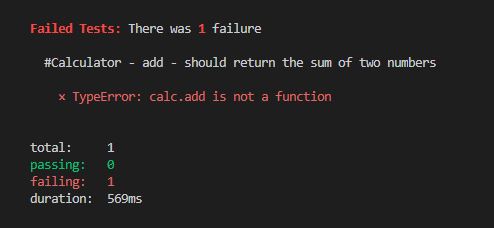
// app/calculator.js

class Calculator {

}

module.exports = Calculator;

Executaremos no terminal o comando "*npm test*", e o nosso primeiro teste deverá falhar, porque o método *add* ainda não existe na classe *Calculator.*



Então, a partir de agora vamos para o segundo passo do TDD, que é implementar uma lógica de forma que o teste passe. A classe *Calculator* deve ficar assim*:*

// app/calculator.js

class Calculator {

  static async add(x, y) {

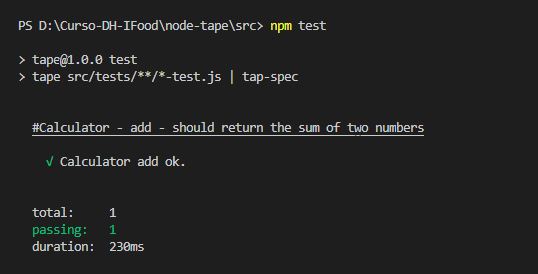
    return x + y;

  }

}

module.exports = Calculator;

Feito isso, rodaremos novamente a chamada aos testes, com o comando "*npm test*", e agora o nosso teste deve passar. O resultado segue na imagem abaixo.



Agora que o nosso teste passou, o terceiro passo do TDD seria refatorar o método *add* da classe *Calculator* de acordo com as regras de negócio da aplicação. Como nossa aplicação é bem trivial, aqui não faremos este passo e adicionaremos um segundo caso de teste, que será a divisão.

// tests/calculator-test.js

...

test('#Calculator - Share - should return the value of dividing two numbers', async(t) => {

  const result = await calc.share(24,6);

  const expected = 4;

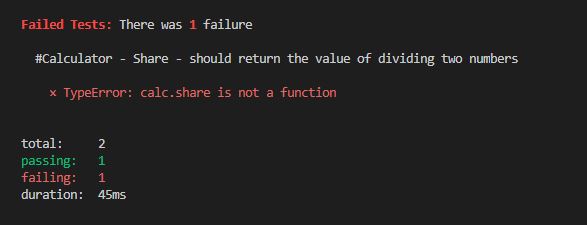
  t.assert(result === expected, 'Calculator share ok.');

  t.end();

});

...

Ao rodar novamente o comando "*npm test*", esse novo teste deve falhar, porque não existe implementação do método *share* na classe Calculator. Veja o resultado:



Então passamos para a implementação do método *share* na classe Calculator, que deve ficar como abaixo:

// app/calculator.js

class Calculator {

  static async add(x, y) {

    return x + y;

  }

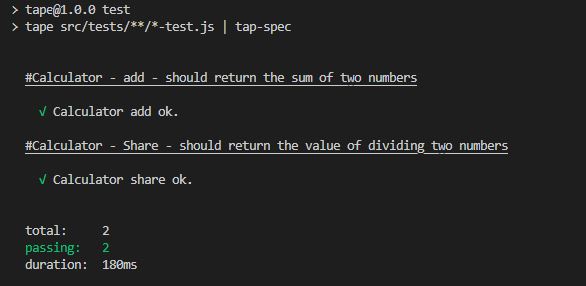
  static async share(x, y) {

    return x / y;

  }

}

Agora rodando mais um vez o comando "*npm test*"*,* nosso segundo caso de teste também passa, e sem afetar o primeiro caso. Veja o resultado:



Nosso segundo caso de teste passou, mas o método *share* é um pouco mais complexo e como regra de negócio, deve verificar se o segundo parâmetro é igual a zero, portanto, devemos refatorar o método, executando assim o terceiro passo do TDD. Para isso, criaremos um novo caso de teste, que verificará o retorno do método *share* quando o segundo parametro passado for igual a zero.

// tests/calculator-test.js

...

test(`#Calculator - share - should return a message

  stating that the second parameter cannot be zero`, async(t) => {

  const result = await calc.share(51,0);

  const expected = 'second parameter cannot be zero';

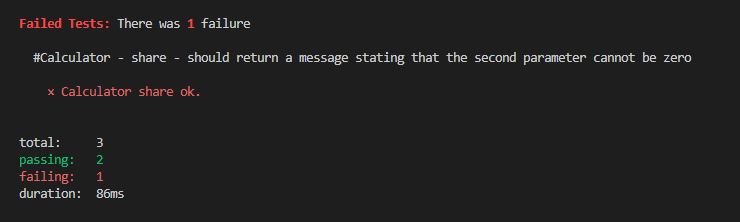
  t.assert(result === expected, 'Calculator share ok.');

  t.end();

});

...

Rodando mais uma vez o comando "*npm test*"*,* nosso novo caso de teste deve falhar, sem afetar os demais casos. Veja abaixo:



Então agora vamos a refatoração do método *share* na classe *Calculator,* onde adicionaremos uma condicional para verificar se o segundo parametro é igual a zero, e se for, retornar uma mensagem.

// app/calculator.js

class Calculator {

  static async add(x, y) {

    return x + y;

  }

  static async share(x, y) {

    if(y === 0) {

      return 'second parameter cannot be zero';

    }

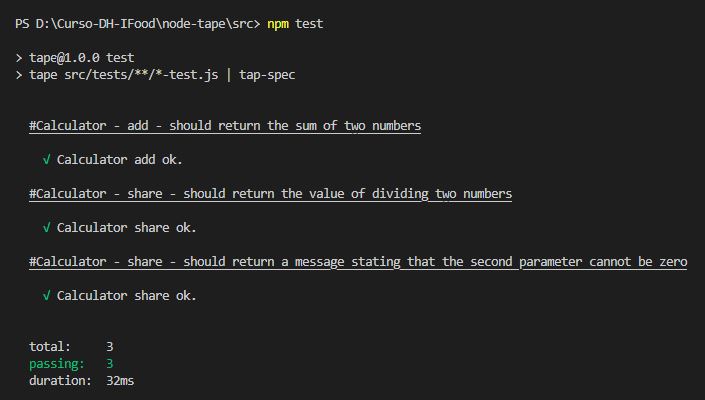
    return x / y;

  }

}

module.exports = Calculator;

Terminada a refatoração, ao executar mais uma vez o comando "*npm test*", nossos três casos de teste escritos até o momento devem passar. Veja o resultado:



Para finalizar o nosso tutorial da calculadora simples usando TDD, escreveremos mais dois casos de teste, o de multiplicação e o de subtração.

O caso de teste da multiplicação

// tests/calculator-test.js

...

test(`#Calculator - multiply - should return the value of multiplying two numbers`, async(t) => {

  const result = await calc.multiply(2,3);

  const expected = 6;

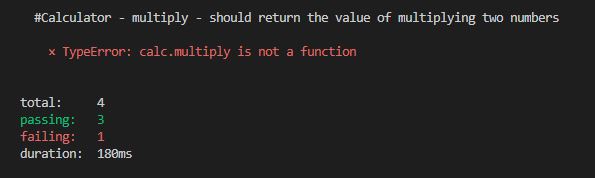
  t.assert(result === expected, 'Calculator multiply ok.');

  t.end();

});

...

O resultado da falha ao rodar o comando "*npm test*".



A implementação do método *multiply* na classe *Calculator* .

// app/calculator.js

class Calculator {

  static async add(x, y) {

    return x + y;

  }

  static async share(x, y) {

    if(y === 0) {

      return 'second parameter cannot be zero';

    }

    return x / y;

  }

  static async multiply(x, y) {

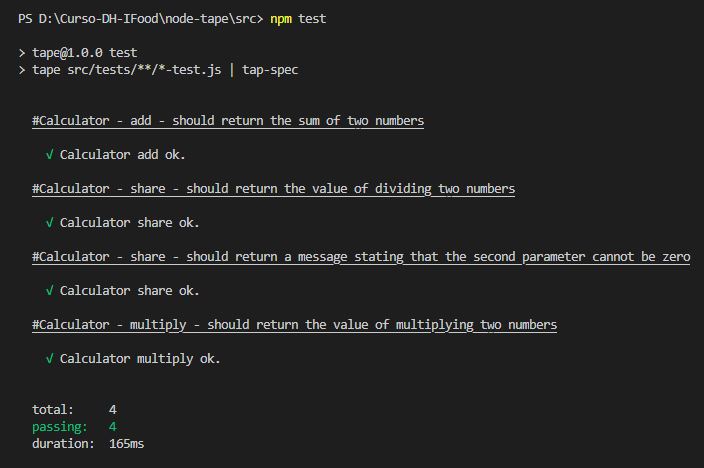
    return x \* y;

  }

}

module.exports = Calculator;

O resultado de todos os testes passando.



E agora por fim o caso de teste do método *subtract.*

// tests/calculator-test.js

...

test(`#Calculator - subtract - should return the value of subtraction of two numbers`, async(t) => {

  const result = await calc.subtract(9,4);

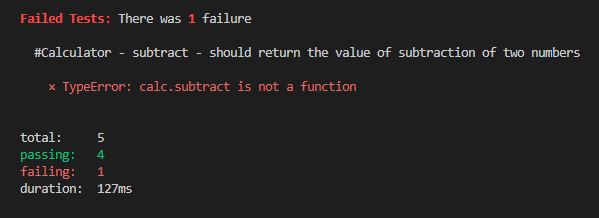
  const expected = 5;

  t.assert(result === expected, `Calculator subtract ok.`);

  t.end();

});

A falha do caso de teste acima, porque o método *subtract* ainda não existe na classe *Calculator*.



A implementação do método *subtract* na classe *Calculator.*

// app/calculator.js

class Calculator {

  static async add(x, y) {

    return x + y;

  }

  static async share(x, y) {

    if(y === 0) {

      return 'second parameter cannot be zero';

    }

    return x / y;

  }

  static async multiply(x, y) {

    return x \* y;

  }

  static async subtract(x, y) {

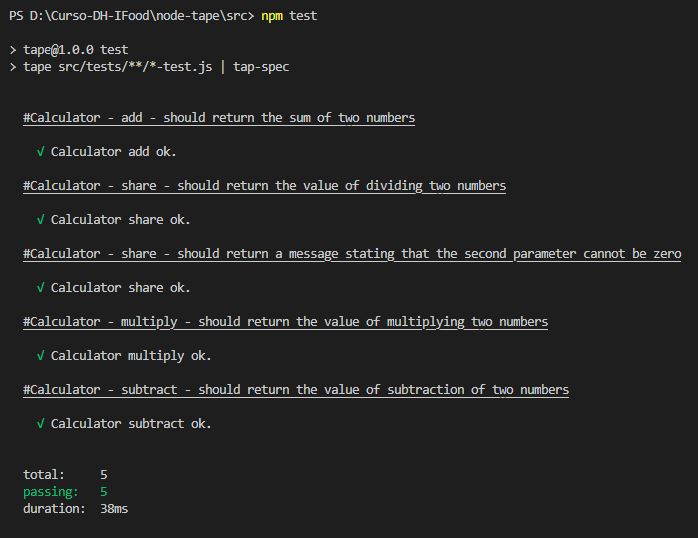
    return x - y;

  }

}

module.exports = Calculator;

E o resultado na imagem abaixo mostrando todos os testes passando.



Este foi um tutorial bastante simples, apresentando o conceito de forma bem superficial e o fluxo de trabalho da técnica TDD. Este projeto esta disponível em: https://github.com/cristianoidelfonso/node-tape