O que vamos aprender

Hoje você aprenderá mais sobre testes e como testar seus scripts NodesJS. Você irá rever os conceitos de testes e os benefícios de escrever testes automatizados.
Para isso, você irá conhecer três ferramentas bastante populares para testes com NodeJS: Mocha , Chai e Sinon . Iremos nos aprofundar em cada uma dessas, aprendendo sobre suas funções e como utilizá-las.
Além disso, revisitaremos conceitos importantes como asserts , testes unitários e TDD .

Você será capaz de:

- Entender o que são e para quê servem testes automatizados.
- Escrever códigos para NodeJS aplicando o conceito de TDD.
- Escrever testes utilizando ferramentas populares;
- Estruturar cenários de testes de acordo com os requisitos;
- Criar *mocks* de funções para testes.

Por que isso é importante?

Testes são parte do dia-a-dia de pessoas desenvolvedoras. Ao longo do desenvolvimento de novas funcionalidades ou de correções de softwares, precisamos testar se nossas implementações estão se comportando como o esperado e realizando aquilo que se propõe da maneira correta.
Através dos testes, conseguimos assegurar a qualidade dessas implementações, assim como, garantir que alterações no código não irão quebrar ou afetar o funcionamento de outras partes do sistema, reduzindo as chances de gerar novos bugs ou efeitos inesperados.
Dessa forma, como exemplos, podemos citar que a agilidade, a produtividade, a performance e a qualidade do processo de desenvolvimento e do software em si, estão diretamente relacionadas à maneira como testamos nossos códigos.
E é por esses motivos que diversas empresas adotam em seu fluxo de desenvolvimento etapas para escrita de testes, sendo consideradas como parte fundamental das entregas. Sendo assim, saber escrever testes e porquê escrevê-los, além de agregar todos os benefícios dessa mentalidade ao seu dia-a-dia, também irá te ajudar à integrar times que possuem essa prática.
E, como já mencionamos em outros conteúdos, ao final desse curso você irá se formar com uma mentalidade orientada a testes, o que tem um grande valor no dia-a-dia de quem desenvolve!

Testes não são de outro mundo!

Antes de falarmos de testes automatizados e das ferramentas que podemos utilizar para testar nossos códigos em NodeJS, vamos refletir um pouco sobre algumas das experiências que já tivemos até aqui.
Você aprendeu diversos conceitos e, para fixá-los, colocou "a mão na massa" várias vezes, realizando exercícios e projetos onde era necessário desenvolver soluções para atender os requisitos propostos.
Durante esse processo, você deve lembrar, era comum executar o mesmo código várias e várias vezes para ver se seu comportamento era o mesmo que esperávamos. Muitas vezes realizamos o

mesmo teste alterando os dados de entrada *(input)* para garantir que a saída *(output)* era condizente com aquilo que foi codificado. E aposto que muitas das vezes o resultado não era positivo: havia faltado um *if* , as vezes precisava de mais um parâmetro ou até mesmo um retorno não tratado como deveria.
Você deve estar se perguntando onde iremos chegar com essa conversa toda né?! E a resposta é simples: Esse processo, que fazemos praticamente de forma natural, nada mais é do que testar . Ou seja, naturalmente testamos nosso código enquanto o escrevemos e esse é justamente o fazer testes .

Testes!

Por que testar?

Existem diversos tipos de testes, cada um com suas características e objetivos. O processo que acabamos de mencionar no tópico anterior, pode ser chamado de "testes manuais". Nesses testes re-executamos o código algumas vezes buscando validar se o comportamento que queremos está sendo realizado corretamente e também alteramos os parâmetros de entrada para tentarmos garantir que tal funcionamento se mantém mesmo com essas variações.
Vamos a um exemplo prático, imagine que queremos criar uma função que receba a média das notas de uma pessoa e responda se ela foi aprovada ou não segundo a seguinte regra:

Média	**Situação**
Menor que 7	Reprovado
Igual ou maior que 7	Aprovado

O primeiro passo que precisamos dar é pensar na estrutura da nossa função:

- Quantos e quais parâmetros ela irá esperar?
- Qual tipo de resposta ela irá retornar?

No nosso caso nossa função deverá receber um parâmetro "media" e responder com "reprovado" ou "aprovado".
Tendo em mente esses questionamentos poderíamos simplesmente já partir para a implementação e chegar ao seguinte código:
examples/calculaSituacao.js

Copiar
```
function calculaSituacao(media) {
  if (media > 7) {
    return 'aprovado';
  }

  return 'reprovado';
}

module.exports = calculaSituacao;
```

Simples né? Agora vamos testar essa função de acordo com os comportamentos que ela deveria ter segundo a proposta, nesse caso precisamos garantir que:

- Se passado um valor menor que 7 , por exemplo 4 , a resposta deve ser "reprovado" ;
- Se passado um valor maior que 7 , por exemplo 9 , a resposta ser "aprovado" ;
- E, não podemos esquecer do "OU", sendo assim, se passado 7 , a resposta deve ser "aprovado" ;

Para validar esses cenários que pensamos podemos escrever algumas chamadas a nossa função:

Copiar
```
const calculaSituacao = require('./examples/calculaSituacao');

console.log(calculaSituacao(4));
// console: reprovado
```

Para ficar mais simples, poderíamos adicionar algumas mensagens para nos ajudar e também já verificar se a resposta dada é aquela que esperamos:

Copiar
```
const calculaSituacao = require('./examples/calculaSituacao');

console.log('Quando a média for menor que 7, retorna "reprovado":');

const respostaCenario1 = calculaSituacao(4);
if (respostaCenario1 === 'reprovado') {
  console.log(`Ok 🚀`);
} else {
  console.error('Resposta não esperada 🚨');
}
// console:
// Quando a média for menor que 7, retorna "reprovado":
// Ok 🚀

console.log('Quando a média for maior que 7, retorna "aprovado":');

const respostaCenario2 = calculaSituacao(9);
if (respostaCenario2 === 'aprovado') {
  console.log(`Ok 🚀`);
```

```
} else {
  console.error('Resposta não esperada 🚨');
}
// console:
// Quando a média for maior que 7, retorna "aprovado":
// Ok 🚀

console.log('Quando a média for igual a 7, retorna "aprovado":');

const respostaCenario3 = calculaSituacao(7);
if (respostaCenario3 === 'aprovado') {
  console.log(`Ok 🚀`);
} else {
  console.error('Resposta não esperada 🚨');
}
// console:
// Quando a média for igual a 7, retorna "aprovado":
// Resposta não esperada 🚨
```

Temos um bug aqui! 🐞
De propósito, deixamos um comportamento falho para simular uma situação normal do dia-a-dia. Nesse caso pode ser um detalhe simples em uma função simples, mas em sistemas mais onde temos diversos pontos diferentes interligados e várias pessoas trabalhando no mesmo código, um cenário de falha é ainda maior.
O que poderíamos fazer em uma situação dessas é implementar a correção e chamar as funções novamente, garantindo que dessa vez todos os cenários estão cobertos inclusive aqueles que já estavam funcionando antes da correção.
Porém, como vimos na prática, testar manualmente nosso projeto pode ser uma tarefa árdua e repetitiva. Como pessoas desenvolvedoras, capazes de construir soluções para tornar processos mais eficientes e rápidos, menos repetitivos e menos sujeitos a erros humanos, por que não automatizamos esse processo também, colhendo dessas e outras vantagens?
É o que veremos a seguir!

Aperte os cintos e bora testar!

Testes automatizados

Ferramentas

Automatizar testes é uma necessidade tão presente no dia-a-dia dos times de desenvolvimento que é assunto constante de discussões e evoluções.
Hoje, já é um assunto amplamente difundido e é possível encontrar diversos tipos, técnicas, implementações e ferramentas diferentes. Essa base sólida sobre o assunto nos ajuda bastante, já que temos diversas ferramentas já consolidadas prontas para serem utilizadas.
Já vimos algumas outras ferramentas desse tipo em conteúdos anteriores, como o Jest e o assert .
Para implementar testes no back-end iremos utilizar a dupla mocha e chai . Apesar de serem duas ferramentas diferentes, elas se completam.
⚠️ Importante: Conforme dito, existem diversas ferramentas disponíveis para teste, e inclusive é possível utilizar o próprio Jest, que vimos em conteúdos anteriores, para testes no back-end também. Porém, como o objetivo é desenvolver uma mentalidade de testes independente das ferramentas, utilizaremos essa stack específica, mas, os conceitos são os mesmos.
Para utilizarmos essas ferramentas precisamos primeiro fazer a instalação, repare que utilizaremos a flag -D . Esses módulos só serão utilizados em fase de desenvolvimento e não serão utilizados para executar nossa aplicação quando ela for publicada. Dessa forma, evitamos instalar pacotes desnecessários em nossa versão de produção.

Copiar
```
npm install -D mocha chai
```

Feita a instalação já podemos importá-las em um arquivo .js e escrever nossos testes.

Tipos de teste

Uma coisa importante para se ter em mente na hora de produzir é o escopo e a interação dos testes. Para isso, existem algumas divisões arbitrárias que nos ajudam a pensar uma ordem de desenvolvimento de testes, sendo as mais comuns:

- Testes unitários : Trabalham presumindo um escopo limitado a um pequeno fragmento do seu código com interação mínima entre recursos externos. Ex: Uma função com um fim específico, como uma função que soma dois números:

Copiar
```
// ./funcoes/calculo/soma.js
// Aqui podemos escrever testes pensando somente o comportamento esperado para função `soma`

const soma = (valorA, valorB) => valorA + valorB;

module.exports = soma;
```

- Testes de integração : Trabalham presumindo a junção de múltiplos escopos (que tecnicamente devem possuir, cada um, seus próprios testes) com interações entre eles. Ex: Uma função de calculadora que usa funções menores que eu posso testar isoladamente/ de forma unitária:

Copiar
```
// ./funcoes/calculadora.js
// Aqui podemos escrever testes pensando o comportamento da função `calculadora` que presume o bom comportamento das funções que integram ela: `soma`, `subtracao`, `multiplicacao`, `divisao`

const { soma, subtracao, multiplicacao, divisao } = require("./calculo");
```

```
const calculadora = (valorA, valorB, operacao) => {
  switch(operacao) {
    case "soma":
      soma(valorA, valorB);
      break;
    case "subtracao":
      subtracao(valorA, valorB);
      break;
    case "multiplicacao":
      multiplicacao(valorA, valorB);
      break;
    case "divisao":
      divisao(valorA, valorB);
      break;
    default:
      break;
  }
};

module.exports = calculadora;

// Esse contexto fica mais evidente, quando temos operações mais complexas nos nossos testes, como operações que envolvem leitura de arquivos e consultas no banco de dados para composição de informações
```

- Testes de Ponto-a-ponto : Chamados também de Fim-a-fim *(End-to-End; E2E)* , esses testes pressupõe um fluxo de interação completo com a aplicação, de um ponto a outro: Aqui, poderíamos pensar uma API que utiliza nossa calculadora (assim como diversas outras funções mais complexas) na hora de realizar uma operação de venda de produtos. Esse teste é o mais completo pois pressupõe que todos os demais testes estão ou serão desenvolvidos (Pensando na prática do TDD que veremos mais adiante).
- Um exemplo prático disso, são os avaliadores de projetos de front-end: Eles pressupõe que toda cadeia de recursos deva estar funcionando para correta renderização das páginas. O que é avaliado com interações de um ponto a outra dessa aplicação (Que são os requisitos , na prática).

Evidentemente isso pode variar a depender do contexto e da forma como os grupos trabalham, mas no geral, existe sempre uma relação de escopo/interação que é definida durante o desenvolvimento de testes e quanto maior o número de escopos diferentes e situações de interação prevista dentro desses escopos, maior a coesão e a confiabilidade do seu projeto.
Aqui, vamos trabalhar com testes unitários pois são mais simples e com a prática, esse padrão tornará testes complexos mais fáceis de entender.

Escrevendo testes

Para exemplificar o processo de escrita de código vamos retomar o exemplo com a função *calculaSituacao* , que vimos anteriormente.

O primeiro passo é compreender, através dos requisitos, a estrutura que desejamos ter e os comportamentos esperados. Já desenvolvemos esses pensamentos anteriormente, retomando-os temos:

- Sobre a estrutura:
  - Nossa função deverá receber um parâmetro "media";
  - Responder com "reprovado" ou "aprovado".
- Sobre os comportamentos esperados:
  1 - Se passado um valor menor que 7 , por exemplo 4 , a resposta deve ser "reprovado" ; 2 - Se passado um valor maior que 7 , por exemplo 9 , a resposta ser "aprovado" ; 3 - E, não podemos esquecer do "OU", sendo assim, se passado 7 , a resposta deve ser "aprovado" ;

Essa estrutura é tudo o que precisamos para escrever nossos testes, antes mesmo de falarmos sobre código.

Estruturando testes com o Mocha

O mocha é um *framework* de testes para JS, isso significa que ele nos ajuda a arquitetar os nossos testes, nos fornecendo a estrutura e interface para escrevermos os nossos testes.
Vamos começar pelos comportamentos. Da mesma forma como descrevemos os comportamentos acima, temos que fazê-lo nos testes para dizermos o que estamos testando naquela caso específico. Para isso, o mocha nos fornece duas palavras reservadas o *describe* e o *it* .
O *describe* nos permite adicionar uma descrição para um teste específico ou um grupo de testes. Já o *it* nos permite sinalizar exatamente o cenário de teste que estamos testando naquele ponto.
Relembrando os testes que escrevemos "na mão", o mocha substitui aqueles logs que utilizamos para descrever cada teste:

Copiar
```
console.log('Quando a média for maior que 7, retorna "aprovado":');
```

Bora ver na prática como podemos fazer isso com a ajuda do mocha . Esse mesmo cenário *1* , utilizando describe para descrever o cenário ficaria assim:

Copiar
```
describe('Quando a média for menor que 7', () => {
  //
});
```

Perceba que o describe aceita dois parâmetros: o primeiro é a descrição e o segundo uma função para executar o cenário de teste. Outro ponto de atenção é que não é necessário importar o mocha em nosso arquivo, suas palavras reservadas serão interpretadas quando executamos o testes, mas veremos mais adiante como fazê-lo.
Descrito nosso comportamento, vamos adicionar o que será testado de fato, ou seja, o que é esperado. Para isso, temos o it :

Copiar
```
describe('Quando a média for menor que 7', () => {
  it('retorna "reprovado"', () => {
    //
  });
});
```

A sintaxe do it é bem semelhante à do describe : ela aceita uma string, qual o comportamento a ser testado, e uma função que executa os testes de fato.

Aferindo testes com o Chai

O chai nos ajudará com as asserções, ou seja, ele nos fornece maneiras de dizermos o que queremos testar e então ele validará se o resultado condiz com o esperado.
Até aqui não estamos testando nada de fato, apenas descrevemos o teste. Para de fato testar nossa função precisamos chamá-la passando o input desejado e então validar se a resposta é aquela que esperamos.
Sem as ferramentas de testes fizemos essa verificação utilizando alguns ifs , o que é bem trabalhoso:

Copiar

```
const respostaCenario1 = calculaSituacao(4);
if (respostaCenario1 === 'reprovado') {
  console.log(`Ok 🚀`);
} else {
  console.error('Resposta não esperada 🚨');
}
```

Essa validação é o que chamamos de *assertion* , "asserção" ou, em alguns casos, "afirmação" . Para nos ajudar com essa tarefa temos o chai , que nos fornece diversos tipos de validações diferentes.
Usaremos a interface expect do chai em nossos exemplos, que significa de fato o que é esperado para determinada variável:

Copiar

```
const resposta = calculaSituacao(4);

expect(resposta).equals('reprovado');
```

No código acima, estamos chamando nossa função e, logo em seguida, afirmamos que seu retorno, armazenado na constante resposta , deve ser *igual a* (equals) 4 .
Muito mais legível e simples!
Vamos ver como fica nosso cenário de teste inteiro com mocha + chai :
tests/calculaSituacao.js

Copiar

```
const { expect } = require('chai');

const calculaSituacao = require('../examples/calculaSituacao');

describe('Quando a média for menor que 7', () => {
  it('retorna "reprovado"', () => {
    const resposta = calculaSituacao(4);

    expect(resposta).equals('reprovado');
  });
});
```

Pronto, nosso primeiro cenário de teste está escrito. Perceba como o chai nos fornece uma função pronta, equals que irá comparar se o valor "esperado" na *resposta* é igual ao passado para ele, ou seja, igual a "reprovado".
A asserção equals é uma das diversas asserções disponíveis no chai. A documentação completa pode ser encontrada na documentação oficial do chai .
Para tornar nosso teste ainda mais legível e elegante, o chai nos fornece outros *getters* encadeáveis que possuem um papel puramente estético. Por exemplo o to e o be , que nos permite escrever nossa *assertion* da seguinte maneira:

tests/calculaSituacao.js

Copiar

```
const { expect } = require('chai');

const calculaSituacao = require('../examples/calculaSituacao');

describe('Quando a média for menor que 7', () => {
  it('retorna "reprovado"', () => {
    const resposta = calculaSituacao(4);

    expect(resposta).to.be.equals('reprovado');
  });
});
```

Perceba que o `to` e o `be` não alteraram em nada a validação realizada, porém, a leitura fica muito mais fluída e natural, é como se estivéssemos dito que nosso teste "espera a resposta ser igual a "reprovado".
Podemos encontrar um pouco mais sobre esse getters na documentação oficial do *chai* , em language chains .

Executando o teste

Antes de começarmos, precisamos criar nosso pacote node para incluirmos os scripts necessários em `package.json` :

Copiar

```
npm init # Iniciando o npm
```

Teste escrito, vamos ver como executá-lo. Como dito antes, o `mocha` é o responsável por executar nossos testes. Ele entenderá as palavras reservadas `describe` e `it` , assim como a estrutura do nosso teste.
Poderíamos tê-lo instalado de maneira global (`npm install -g mocha`) em nossa máquina, e bastaria chamá-lo diretamente em nosso terminal passando o arquivo do teste (`mocha tests/calculaSituacao.js`). Entretanto, faremos da maneira mais recomendada e elegante: utilizaremos o pacote que já temos instalado. Para isso, vamos adicionar um novo script ao nosso `package.json` , que chama o `mocha` e informa um arquivo ou diretório de testes:
package.json

Copiar

```
{
// ...
  "scripts": {
    "start": "node index.js",
    "test": "mocha tests"
  },
// ...
}
```

Dessa forma, não precisamos instalar nada globalmente, e para executar nosso teste basta rodar em nosso terminal o script `test` , que irá executar =o comando `mocha tests` utilizando o módulo instalado:

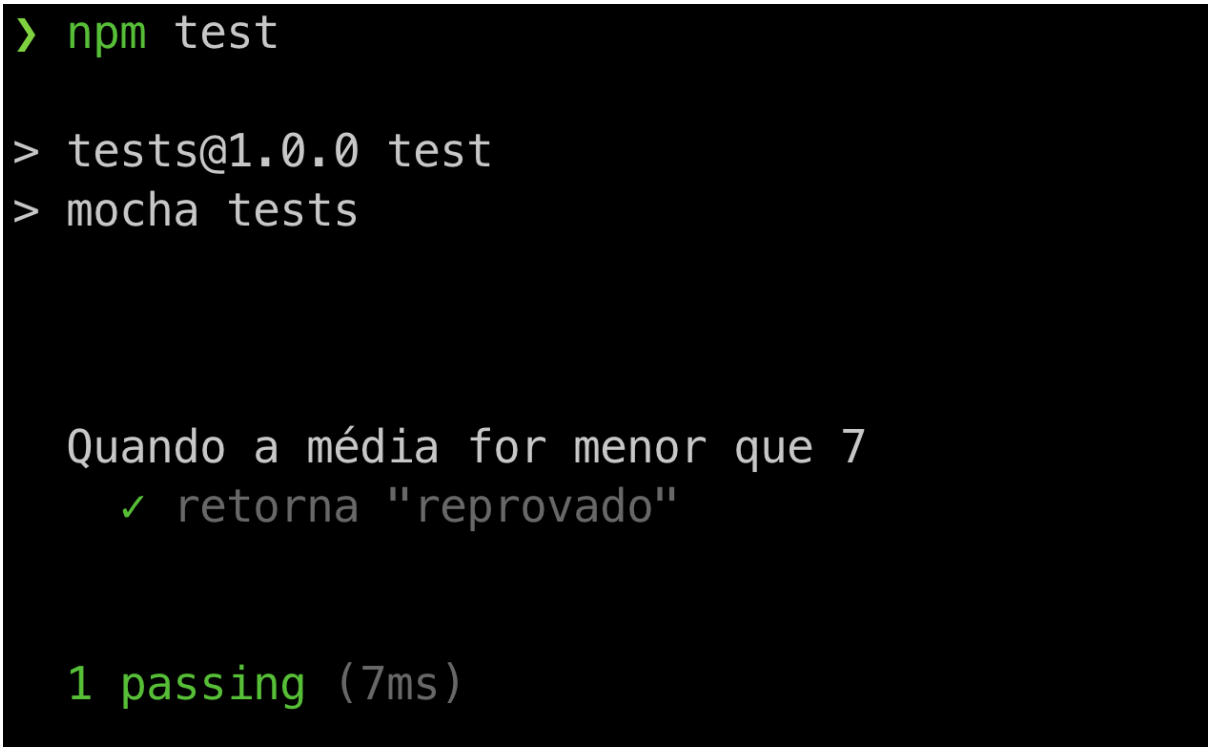
Copiar

```
npm run test
```

Ou simplesmente

Copiar
```
npm test
```

E teremos um output parecido com:



Testando todos os cenários

Adicionando os demais comportamentos, temos:
tests/calculaSituacao.js

Copiar
```
const { expect } = require('chai');

const calculaSituacao = require('../examples/calculaSituacao');

describe('Quando a média for menor que 7', () => {
  it('retorna "reprovado"', () => {
    const resposta = calculaSituacao(4);

    expect(resposta).to.be.equals('reprovado');
  });
});

describe('Quando a média for maior que 7', () => {
  it('retorna "aprovado"', () => {
    const resposta = calculaSituacao(9);

    expect(resposta).to.be.equals('aprovado');
```

```
  });
});

describe('Quando a média for igual a 7', () => {
  it('retorna "aprovado"', () => {
    const resposta = calculaSituacao(7);

    expect(resposta).to.be.equals('aprovado');
  });
});
```

Pronto, vamos executá-lo e ver o resultado.
Lembre-se que deixamos um cenário quebrado de proposito para corrigirmos.

```
❯ npm test

> tests@1.0.0 test
> mocha tests



  Quando a média for menor que 7
    ✓ retorna "reprovado"

  Quando a média for maior que 7
    ✓ retorna  "aprovado"

  Quando a média for igual a 7
    1) retorna "reprovado"


  2 passing (10ms)
  1 failing

  1) Quando a média for igual a 7
       retorna "reprovado":

      AssertionError: expected 'reprovado' to equal 'aprovado'
      + expected - actual

      -reprovado
      +aprovado

      at Context.<anonymous> (tests/calculaSituacao.js:25:32)
      at processImmediate (internal/timers.js:458:21)
```

Nosso teste agora está validando com sucesso os cenários esperados. Podemos então, aplicar a correção que falta em nosso código e então simplesmente rodar `npm test` para garantir tanto que o bug foi corrigido, quanto que os outros cenários continuaram funcionando após a correção.

Bora testar!

TDD - Transformando requisitos em testes

Agora que já vimos como utilizar ferramentas para nos ajudar na escrita de testes, vamos novamente refletir sobre o que fizemos até aqui.
No exemplo acima, começamos pela implementação do código, depois escrevemos os testes para validá-lo e, então, descobrimos que havia um cenário que não estava funcionando como esperado e que precisava de ajustes.
Perceba que tivemos uma espécie de "retrabalho" com a implementação, pois primeiro escrevemos uma primeira versão do código para depois identificar o erro e então corrigí-lo.
E se formos pelo caminho contrário? Se antes de tentarmos implementar o código já começarmos traduzindo as especificações em testes e então já desenvolver pensando neles?
Pensando dessa forma que surgiu o conceito de TDD (Test Driven Development), em tradução livre, *Desenvolvimento Orientado a Testes* . Esse metodologia é bastante difundida e pode trazer muitos benefícios para o desenvolvimento.
A prática do TDD em começar a escrever os testes que traduzem e validam os comportamentos esperados para aquele código antes de começar a implementação.
A ideia principal é começarmos escrever o código já pensando no que está sendo testado, ou seja, já teremos em mente quais os cenários que precisamos cobrir e também como nosso código precisa estar estruturado para que possamos testá-lo, já que códigos menos estruturados normalmente são mais difíceis de serem testados.
Dessa forma, pensando em passos para o TDD, podemos pensar da seguinte maneira:

1. Antes de qualquer coisa, precisamos interpretar os requisitos, pensando nos comportamentos que iremos implementar e a na estrutura do código: se será uma função, um módulo, quais os inputs, os outputs, etc..
2. Tendo isso em mente, começamos a escrever os testes, ou seja, criamos a estrutura de describes e its que vimos.
3. Depois, escrevemos as asserções. Perceba que antes mesmo de ter qualquer código, já iremos criar chamadas a esse código, o que significa que nossos testes irão falhar. Não se preocupe, pois essa é exatamente a ideia nesse momento.
4. Agora que já temos os testes criados, vamos a implementação do nosso código. A ideia é escrever os códigos pensando nos testes e, conforme vamos cobrindo os cenários, nossos testes que antes quebravam começam a passar.

Se precisar fazer algum ajuste nos testes em algum momento, não se preocupe! Isso é perfeitamente normal, visto que estamos escrevendo testes para código que ainda não existe, e um detalhe ou outro pode *escapulir* à mente.

Um pouco mais de testes

Até agora, você viu como transformar requisitos em testes, como escrevê-los com ajuda do mocha e do chai e o que é TDD. Vamos fazer mais um exemplo juntos utilizando tudo isso:
Escreveremos uma função que lê o conteúdo de um arquivo. Essa função:

- Receberá um parâmetro com o nome do arquivo a ser lido. Esse arquivo deverá estar na pasta io-files ;
- Caso o arquivo solicitado exista, responderá uma *string* com o conteúdo do arquivo;
- Caso o arquivo solicitado não exista, deverá responder null .

Seguindo o TDD, vamos começar estruturando os testes com o mocha e com o chai . Antes de mais nada, vamos criar um novo diretório raiz para receber o nosso pacote node e instalar nossas ferramentas de testes:

Copiar
```
mkdir examples2

cd examples2

mkdir io-test && cd io-test # Criando e entrando no diretório do nosso projeto

npm init # Iniciando o npm

npm install --save-dev mocha chai # Instalando as ferramentas de testes
```

Agora basta adicionar o seguinte script em seu package.json :
io-test/package.json

Copiar
```
{
  //
  "scripts": {
    "start": "node index.js",
    "test": "mocha test.js"
  },
  //
}
```

Mocha

Feito isso, vamos escrever nosso arquivo test.js . Começaremos estruturando os requisitos em forma de testes com o mocha :
io-test/test.js

Copiar

```
describe('leArquivo', () => {
  describe('Quando o arquivo existe', () => {
    describe('a resposta', () => {
      it('é uma string', () => {
        //
      });

      it('é igual ao conteúdo do arquivo', () => {
        //
      });
    });
  });

  describe('Quando o arquivo não existe', () => {
    describe('a resposta', () => {
      it('é igual a "null"', () => {
        //
      });
    });
  });
});
```

Chai

Em seguida vamos adicionar as asserções com o chai:
io-test/test.js

Copiar

```
const { expect } = require('chai');

const leArquivo = require('./leArquivo');

const CONTEUDO_DO_ARQUIVO = 'VQV com TDD';

describe('leArquivo', () => {
  describe('Quando o arquivo existe', () => {
    describe('a resposta', () => {
      const resposta = leArquivo('arquivo.txt');

      it('é uma string', () => {
        expect(resposta).to.be.a('string');
      });

      it('é igual ao conteúdo do arquivo', () => {
        expect(resposta).to.be.equals(CONTEUDO_DO_ARQUIVO);
      });
    });
  });

  describe('Quando o arquivo não existe', () => {
    it('a resposta é igual a "null"', () => {
      const resposta = leArquivo('arquivo_que_nao_existe.txt');
```

```
    expect(resposta).to.be.equal(null);
  });
 });
});
```

Aqui utilizamos uma nova asserção do chai , o a , que validará o "tipo" daquele retorno. Como se tivéssemos escrito: "espera a resposta ser uma string" (ou *expect response to be a string*).
Para que o teste seja executado, precisamos criar o arquivo que irá conter a função. Vamos começar com uma função vazia apenas para conseguir importá-la no arquivo de teste:
io-test/leArquivo.js

Copiar

```
module.exports = () => {
  //
}
```

Agora vamos rodar o teste e ver o resultado:

Copiar

```
npm test # ou npm run test
```

Teremos a seguinte saída em nosso console:

Implementação

io-test/leArquivo.js

Copiar

```
const fs = require('fs');

function leArquivo(nomeDoArquivo) {
 try {
  const conteudoDoArquivo = fs.readFileSync(nomeDoArquivo, 'utf8');

  return conteudoDoArquivo;
 } catch (err) {
  return null;
 }
}

module.exports = leArquivo;
```

Após a implementação desse código um dos testes já passa ser executado com sucesso:

Isolando nossos testes

Antes de continuar, precisamos ter atenção a um ponto: nossos testes não devem realizar operações de IO (*input* / *output*), ou seja, não devem acessar nem o disco, nem a rede.
Quando criamos aplicações de frontend, estamos na maior parte do tempo, manipulando o DOM. Quando falamos de javascript no backend com NodeJS, constantemente estamos realizando operações com IO, ou seja, nossa aplicação se comunica com o sistema de arquivos ou com a rede.

Exemplos dessas comunicações são a leitura e escrita de arquivos, chamadas a APIs ou consultas em um banco de dados.
Sendo assim, ao escrever testes, será muito comum precisarmos testar códigos que fazem esse tipo de operação de integração , o que pode adicionar complexidade aos nossos testes.
Vejamos o exemplo que estamos construíndo: para garantir nossos cenários, precisaríamos, além de criar o teste e realizar a chamada à nossa função `leArquivo` , preparar um arquivo para ser lido com o conteúdo que esperamos ler.
Pode parecer simples, mas por exemplo, para testar uma função que acessa um banco de dados, precisaríamos disponibilizar uma instância desse banco de dados para que nossos testes se conectassem, e precisaríamos garantir que existissem registros com as diversas situações que nossos testes precisassem testar. Além disso, após a execução dos nossos testes, tais registros provavelmente teriam sido alterados e teríamos que garantir que voltassem ao estado inicial para podermos executar nosso teste novamente.
Resumindo: criar testes para códigos que executem operações de IO nos dá diversas complexidades.

E agora?
Dessa forma, o ideal é não permitir que nosso código realize essas operações de IO de fato, mas apenas simular que elas estão sendo realizadas. Dessa forma, isolamos o IO de nossos testes, garantindo que um banco de dados inconsistente ou um arquivo faltando na hora de executar os testes não faça com que tudo vá por água abaixo
Para isso existe o conceito de `Test Doubles` , que são objetos que fingem ser o outro objeto para fins de testes.
Com esses objetos, podemos simular, por exemplo, as funções do módulo `fs` . Nosso código irá pensar que está chamando as funções do `fs` , porém, estará chamando as nossas funções, que se comportarão da maneira que queremos, mas sem a necessidade de escrever, ler ou ter dependência de arquivo reais.
Para nos ajudar com esse tipo de coisa, usaremos uma ferramenta chamada `sinon` que veremos a seguir.

```
> mocha test.js



  Executa a função "leArquivo"
    Quando o arquivo existe
      a resposta
        1) é uma string
        2) é igual ao conteúdo do arquivo
    Quando o arquivo não existe
      a resposta
        ✓ é igual a "null"


  1 passing (9ms)
  2 failing

  1) Executa a função "leArquivo"
       Quando o arquivo existe
         a resposta
           é uma string:
     AssertionError: expected null to be a string
      at Context.it (test.js:24:40)

  2) Executa a função "leArquivo"
       Quando o arquivo existe
         a resposta
           é igual ao conteúdo do arquivo:
     AssertionError: expected null to equal 'VQM com TDD'
      at Context.it (test.js:30:40)



npm ERR! Test failed.  See above for more details.
```

```
> mocha test.js



  Executa a função "leArquivo"
    Quando o arquivo existe
      a resposta
        1) é uma string
        2) é igual ao conteúdo do arquivo
    Quando o arquivo não existe
      a resposta
        3) é igual a "null"


  0 passing (10ms)
  3 failing

  1) Executa a função "leArquivo"
       Quando o arquivo existe
         a resposta
           é uma string:
     AssertionError: expected undefined to be a string
      at Context.it (test.js:24:40)

  2) Executa a função "leArquivo"
       Quando o arquivo existe
         a resposta
           é igual ao conteúdo do arquivo:
     AssertionError: expected undefined to equal 'VQM com TDD'
      at Context.it (test.js:30:40)

  3) Executa a função "leArquivo"
       Quando o arquivo não existe
         a resposta
           é igual a "null":
     AssertionError: expected undefined to equal null
      at Context.it (test.js:50:40)



npm ERR! Test failed.  See above for more details.
```

Sinon

O Sinon é uma ferramenta que fornece funções para diversos tipos dos Test Doubles ou, numa tradução livre, Dublês de Testes (remetendo aos dublês de filmes).
No momento focaremos em um tipo de Test Double, o stub . *Stubs* são objetos que podemos utilizar para simular interações com dependências externas ao que estamos testando de fato (na literatura, é comum referir-se ao sistema sendo testado como *SUT* , que significa System under Test).
Primeiro, vamos fazer a instalação do Sinon:

Copiar
```
npm install --save-dev sinon
```

Agora vamos ver na prática como podemos criar um stub para a função de leitura do fs :

Copiar
```
const fs = require('fs');
const sinon = require('sinon');

sinon.stub(fs, 'readFileSync')
  .returns('Valor a ser retornado');
```

Perceba que precisamos importar o módulo fs e, então, falamos para o sinon criar um stub para a função readFileSync que retornará 'Valor a ser retornado' , conforme especificamos na chamada para returns .

Sinon

O Sinon é uma ferramenta que fornece funções para diversos tipos dos Test Doubles ou, numa tradução livre, Dublês de Testes (remetendo aos dublês de filmes).
No momento focaremos em um tipo de Test Double, o stub . *Stubs* são objetos que podemos utilizar para simular interações com dependências externas ao que estamos testando de fato (na literatura, é comum referir-se ao sistema sendo testado como *SUT* , que significa System under Test).
Primeiro, vamos fazer a instalação do Sinon:

Copiar
```
npm install --save-dev sinon
```

Agora vamos ver na prática como podemos criar um stub para a função de leitura do fs :

Copiar
```
const fs = require('fs');
const sinon = require('sinon');

sinon.stub(fs, 'readFileSync')
  .returns('Valor a ser retornado');
```

Perceba que precisamos importar o módulo fs e, então, falamos para o sinon criar um stub para a função readFileSync que retornará 'Valor a ser retornado' , conforme especificamos na chamada para returns .

Exercícios

Hora de pôr a mão na massa!
back-end

Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma branch, com arquivos e commits específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. $ cd ~/trybe-exercicios
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

$ git checkout main

6. $ git pull
7. A partir da main, crie uma branch com o nome exercicios/26.3 *(bloco 26, dia 3)*
8. Copiar
9. $ git checkout -b exercicios/26.3
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

$ mkdir back-end

12. $ cd back-end
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

$ mkdir bloco-26-introducao-ao-desenvolvimento-web-com-nodejs

15. $ cd bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

$ mkdir dia-3-testes-com-nodejs

18. $ cd dia-3-testes-com-nodejs
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório ~/trybe-exercicios/back-end/block-26-introducao-ao-desenvolvimento-web-com-nodejs/dia-3-testes-com-nodejs. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

$ git status

```
On branch exercicios/26.3
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`
26. Você pode visualizar o log de todos os commits já feitos naquela branch com `git log`.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.3)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e coloquei pra quando clicar
aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300
```

28. `    Resolvendo o exercício 1 usando eventListener`
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch `exercicios/26.3` não vai existir no repositório remoto, então precisamos configurar o `remote` utilizando a opção `--set-upstream` (ou `-u`, que é a forma abreviada).
30. Copiar

31. `$ git push -u origin exercicios/26.3`
32. Após realizar o passo 9, podemos abrir a Pull Request a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com -u anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch exercicios/26.3, e disponíveis no repositório remoto do GitHub. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊👊

Agora a prática

Exercício 1 : Estruture os testes utilizando mocha e chai para um função que irá dizer se um número é "positivo", "negativo" ou "neutro":

- Essa função irá receber um número como parâmetro e retornar uma string como resposta;
- Quando o número passado for maior que 0 deverá retornar "positivo", quando for menor que 0 deverá retornar "negativo" e quando igual a 0 deverá retornar "neutro";

1. Descreva todos os cenário de teste utilizando describes ;
2. Descreva todos os testes que serão feitos utilizando its ;
3. Crie as asserções validando se os retornos de cada cenário tem o tipo e o valor esperado.

Exercício 2 : Implemente a função apresentada no exercício 1, garantindo que ela irá passar em todos os testes que você escreveu.
DICA : Lembre-se de adicionar o script de test no package.json e de instalar as dependências.
Exercício 3 Adicione à função um tratamento para casos em que o parâmetro informado não seja do tipo Number .

1. Adicione o cenário em seu arquivo de testes, passando um valor de tipo diferente a Number para a função;
2. Adicione uma asserção para esperar que o valor retornado para esse caso seja igual uma string "o valor deve ser um número";
3. Implemente em sua função tal validação para que o teste passe.

Exercício 4 Crie testes para uma função que escreverá um conteúdo em um arquivo específico.

- Essa função deverá receber dois parâmetros: o nome do arquivo e o conteúdo desse arquivo.
- Após concluir a escrita do arquivo ela deverá retornar um `ok` .

1. Descreva todos os cenários de teste utilizando `describes` ;
2. Descreva todos os testes que serão feitos utilizando `its` ;
3. Crie as asserções validando se o retorno da função possui o valor e tipo esperado.

Exercício 5 Implemente a função descrita no exercício 4.

1. Crie a função descrita no exercício 4 utilizando o módulo `fs` do node.
2. Adapte os testes adicionando `stub` ao módulo utilizado do `fs` , isolando assim o teste.
3. Garanta que todos os testes escritos no exercício 4 irão passar com sucesso.

Gabarito dos exercícios

A seguir encontra-se uma sugestão de solução para os exercícios propostos.

Exercício 1

Estruture os testes utilizando `mocha` e `chai` para um função que irá dizer se um número é "positivo", "negativo" ou "neutro":

- Essa função irá receber um número como parâmetro e retornar uma string como resposta;
- Quando o número passado for maior que 0 deverá retornar "positivo", quando for menor que 0 deverá retornar "negativo" e quando igual a 0 deverá retornar "neutro";

1. Descreva todos os cenários de teste utilizando `describes` ;
2. Descreva todos os testes que serão feitos utilizando `its` ;
3. Crie as asserções validando se os retornos de cada cenário tem o tipo e o valor esperado.

Resolução

Copiar

```
const { expect } = require('chai');

const numNaturalFn = require('./numerosNaturais');

describe('Executa a função numNaturalFn', () => {
  describe('quando o número for maior que 0', () => {
    describe('a resposta', () => {
      it('é uma "string"', () => {
        const resposta = numNaturalFn(10);

        expect(resposta).to.be.a('string');
      });

      it('é igual a "positivo"', () => {
        const resposta = numNaturalFn(10);
```

```
      expect(resposta).to.be.equals('positivo');
    });
  });
});

describe('quando o número for menor que 0', () => {
  describe('a resposta', () => {
    it('é uma "string"', () => {
      const resposta = numNaturalFn(-10);

      expect(resposta).to.be.a('string');
    });

    it('é igual a "negativo"', () => {
      const resposta = numNaturalFn(-10);

      expect(resposta).to.be.equals('negativo');
    });
  });
});

describe('quando o número for igual a 0', () => {
  describe('a resposta', () => {
    it('é uma "string"', () => {
      const resposta = numNaturalFn(0);

      expect(resposta).to.be.a('string');
    });

    it('é igual a "neutro"', () => {
      const resposta = numNaturalFn(0);

      expect(resposta).to.be.equals('neutro');
    });
  });
});
```

Exercício 2

Implemente a função apresentada no exercício 1, garantindo que ela irá passar em todos os testes que você escreveu.
DICA : Lembre-se de adicionar o script de test no package.json e de instalar as dependências.

Resolução

Copiar

```
module.exports = (numero) => {
  if (numero > 0) {
    return 'positivo';
```

```
  }

  if (numero < 0) {
    return 'negativo';
  }

  return 'neutro';
};
```

Exercício 3

Adicione à função um tratamento para casos em que o parâmetro informado não seja do tipo `number` .

1. Adicione o cenário em seu arquivo de testes, passando um valor de tipo diferente a `number` para a função;
2. Adicione uma asserção para esperar que o valor retornado para esse caso seja igual uma string "o valor deve ser um número";
3. Implemente em sua função tal validação para que o teste passe.

Resolução

Copiar

```
// demais casos de teste

describe('quando o parâmetro passado não é um número', () => {
  describe('a resposta', () => {
    it('é uma "string"', () => {
      const resposta = numNaturalFn('AAAA');

      expect(resposta).to.be.a('string');
    });

    it('é igual a "o parâmetro deve ser um número"', () => {
      const resposta = numNaturalFn('AAAA');

      expect(resposta).to.be.equals('o parâmetro deve ser um número');
    });
  });
});
```

Copiar

```
module.exports = (numero) => {
  if (typeof numero !== 'number') {
    return 'o parâmetro deve ser um número';
  }

  if (numero > 0) {
    return 'positivo';
  }

  if (numero < 0) {
```

```
    return 'negativo';
  }

  return 'neutro';
};
```

Exercício 4

Crie testes para uma função que escreverá um conteúdo em um arquivo específico.

- Essa função deverá receber dois parâmetros: o nome do arquivo e o conteúdo desse arquivo.
- Após concluir a escrita do arquivo ela deverá retornar um `ok` .

1. Descreva todos os cenários de teste utilizando `describes` ;
2. Descreva todos os testes que serão feitos utilizando `its` ;
3. Crie as asserções validando se o retorno da função possui o valor e tipo esperado.

Resolução

Copiar

```
const fs = require('fs');
const { expect } = require('chai');

const escrevaArquivo = require('./escrevaArquivo');

describe('Executa a função escrevaArquivo', () => {
  describe('a resposta', () => {
    it('é uma "string"', () => {
      const resposta = escrevaArquivo('arquivo.txt', '#vqv conteúdo');

      expect(resposta).to.be.a('string');
    });

    it('é igual a "ok"', () => {
      const resposta = escrevaArquivo('arquivo.txt', '#vqv conteúdo');

      expect(resposta).to.be.equals('ok');
    });
  });
});
```

Exercício 5

implemente a função descrita no exercício 4.

1. Crie a função descrita no exercício 4 utilizando o módulo `fs` do Node.
2. Adapte os testes adicionando `stub` ao módulo utilizado do `fs` , isolando assim o teste.
3. Garanta que todos os testes escritos no exercício 4 irão passar com sucesso.

Resolução

Copiar

```
const fs = require('fs');
const sinon = require('sinon');
const { expect } = require('chai');

const escrevaArquivo = require('./escrevaArquivo');

describe('Executa a função escrevaArquivo', () => {
  before(() => {
    sinon.stub(fs, 'writeFileSync');
  });

  after(() => {
    fs.writeFileSync.restore();
  });

  describe('a resposta', () => {
    it('é uma "string"', () => {
      const resposta = escrevaArquivo('arquivo.txt', '#vqv conteúdo');

      expect(resposta).to.be.a('string');
    });

    it('é igual a "ok"', () => {
      const resposta = escrevaArquivo('arquivo.txt', '#vqv conteúdo');

      expect(resposta).to.be.equals('ok');
    });
  });
});
```

Copiar

```
const fs = require('fs');

module.exports = (nomeDoArquivo, conteudoDoArquivo) => {
  fs.writeFileSync(`${__dirname}/${nomeDoArquivo}`, conteudoDoArquivo);

  return 'ok';
};
```