

Objetivos:

- I. Teste unitário;
- II. Teste unitário usando Jest;
- III. Matchers e modificadores no Jest;
- IV. Grupo de testes;
- V. Hooks de ciclo de vida do teste;
- VI. Teste de função objetivo de requisição HTTP;
- VII. Teste de CRUD no MongoDB.

Siga as instruções para criar o projeto para reproduzir os exemplos:

- a) Crie uma pasta de nome **servidor** (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
- b) No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node, arquivo `package.json`;
- c) No terminal, execute o comando `npm i express dotenv mongoose` para instalar os pacotes. O mongoose é uma biblioteca usada para fazer o acesso ao MongoDB (<https://www.npmjs.com/package/mongoose>);
- d) No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém as definições de tipos do pacote express. Quando usamos um pacote é preciso ter acesso às declarações de tipo do pacote para que o TS saiba quais tipos de dados esperar do framework;
- e) No terminal, execute o comando `npm i -D ts-node ts-node-dev typescript` para instalar os pacotes `ts-node`, `ts-node-dev` e `typescript` como dependências de desenvolvimento;
- f) No terminal, execute o comando `npm i -D jest ts-jest @types/jest` para instalar o framework de teste;
- g) No terminal, execute o comando `npm i -D mongodb-memory-server` para instalar o pacote usado para criar um servidor MongoDB em memória (temporário);
- h) No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo `tsconfig.json`);
- i) Crie o arquivo `.gitignore` na raiz do projeto e coloque a linha para ignorar a pasta `node_modules`;
- j) Crie o arquivo `.env` na raiz do projeto e coloque a seguinte variável de ambiente:
`PORT = 3001`

Estrutura de pastas e arquivos do projeto:

```
▼ SERVER
  > node_modules
  ▼ src
    ▼ controllers
      TS CarController.ts
      TS OperacaoController.ts
    ▼ models
      TS connection.ts
      TS index.ts
    ▼ routes
      TS car.ts
      TS index.ts
      TS operacao.ts
      TS index.ts
  ▼ test
    TS Car.test.ts
    TS Operacao.test.ts
  .env
  .gitignore
  package-lock.json
  package.json
  tsconfig.json
```

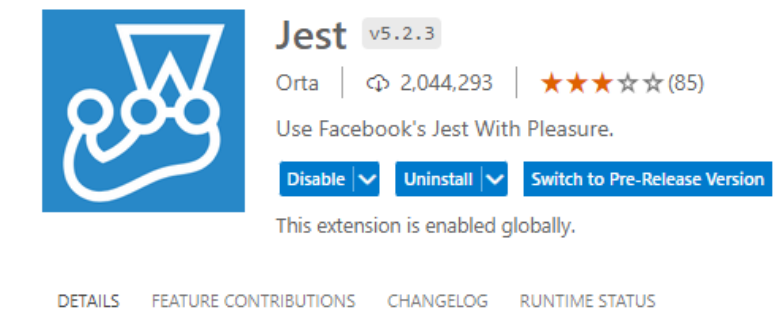
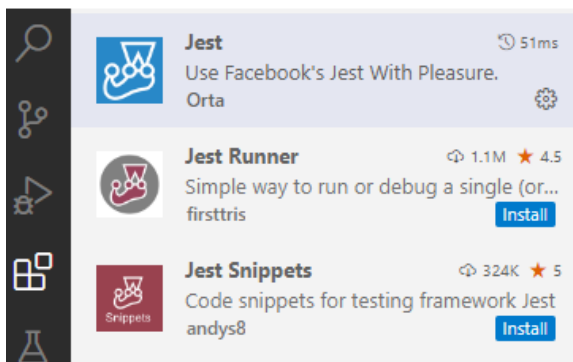
k) Coloque as seguintes propriedades no arquivo package.json:

```
"scripts": {
  "start": "ts-node ./src",
  "dev": "ts-node-dev ./src" ,
  "test": "jest"
},
"jest": {
  "preset": "ts-jest",
  "testEnvironment": "node",
  "testMatch": [
    "<rootDir>/test/**/*.test.ts"
  ]
},
```

A propriedade **test**, da scripts, usa o jest para rodar os arquivos de testes e a propriedade **jest** é usada para configurar o Jest:

- **preset**: especifica um conjunto predefinido de configurações do Jest para trabalhar com TS. No caso, o ts-jest é um preset que configura automaticamente o Jest para lidar com arquivos TS, incluindo a transpilação para JS antes da execução dos testes;
- **testEnvironment**: define o ambiente em que os testes serão executados. Isso é apropriado para testes de lógica de servidor ou código que depende de APIs específicas do Node.js;
- **testMatch**: define padrões que o Jest deve considerar como nomes de arquivos que contêm testes. Neste caso, `<rootDir>/test/**/*.test.ts` significa que o Jest procurará por arquivos dentro da pasta **test** que tenham a extensão **.test.ts** e os considerará como testes a serem executados.

Para auxiliar na execução dos testes no VS Code recomenda-se instalar a extensão vscode-jest ou alguma outra de sua preferência.



Resultado do teste usando a extensão vscode-jest:

Resultado do teste usando npm run test:

Na aba TEST RESULTS também é possível ver o resultado dos testes

```

> Test run started at 2/8/2024, 9:55:02 AM <
PASS test/controllers/Person.test.ts
PASS test/controllers/Car.test.ts
PASS test/controllers/CarByPerson.test.ts
> Test run finished at 2/8/2024, 9:55:06 AM <

Test Suites: 3 passed, 3 total
Tests:       35 passed, 35 total
Snapshots:   0 total
Time:        4.659 s, estimated 5 s
Ran all test suites.
  
```

Para ver no terminal o resultado dos testes

```

PS D:\pessoal\servidor> npm run test
> servidor@1.0.0 test
> jest

PASS test/controllers/Person.test.ts
PASS test/controllers/Car.test.ts
PASS test/controllers/CarByPerson.test.ts

Test Suites: 3 passed, 3 total
Tests:       35 passed, 35 total
Snapshots:   0 total
Time:        3.376 s, estimated 4 s
Ran all test suites.
  
```

i. Teste unitário

O teste unitário ou teste de unidade é uma técnica que consiste na verificação da menor unidade testável de um programa. No caso do TS, a menor unidade é uma função ou método.

O teste unitário é de responsabilidade do próprio programador durante a codificação, como exemplo, após codificar uma classe o programador deve codificar os testes dos métodos. Esta técnica permite que o teste encontre erros mais facilmente, visto que cada classe é testada após ser codificada, em vez de fazer o teste após codificar todo o programa.

A prática de escrever testes unitários ajuda a identificar e corrigir rapidamente problemas em pequenas partes do código, contribuindo para a confiabilidade e manutenibilidade do sistema como um todo.

O principal objetivo dos testes unitários é garantir que cada unidade de código funcione conforme o esperado, isoladamente do restante do sistema, ou seja, testa-se cada função ou método isoladamente.

O teste unitário é considerado o primeiro de uma cadeia de testes a qual um software pode ser submetido, pois a ideia é testar do menor módulo (método/função) até o maior (todas as funcionalidades da aplicação).

O teste de uma unidade pode ser escrito **antes** ou **após** a codificação do método/função.

Um bom teste é aquele que tem elevada probabilidade de revelar um erro, e um teste para ser bem-sucedido deve revelar erros ainda não revelados. Os testes têm de ser sempre documentados e reproduzíveis. Além disso, deve apresentar características tais como:

- Ser operável: significa que deve ser fácil de executar e não depender de configurações complexas ou pré-requisitos difíceis de atender;
- Ser observável: permitindo que os resultados e o comportamento do código sejam facilmente monitorados. Isso facilita a identificação de problemas e a compreensão do que está acontecendo durante a execução do teste;
- Controlável: capacidade de isolar e testar unidades específicas sem depender fortemente de outros componentes;
- Decomposto: significa que podem ser divididos em partes menores e independentes. Isso facilita a manutenção, a identificação de falhas específicas e a compreensão dos testes como um todo;





- Simplicidade: a simplicidade é essencial em testes. Testes simples são mais fáceis de entender, manter e executar. Evitar complexidade desnecessária contribui para a eficiência do processo de teste;
- Estabilidade: testes devem ser estáveis, ou seja, consistentes em suas execuções. Testes que produzem resultados flutuantes ou inconsistentes podem levar a confusão e dificultar a identificação de problemas reais;
- Compreensível: testes devem ser compreensíveis, permitindo que desenvolvedores e membros da equipe entendam facilmente o propósito do teste, as condições de entrada e o que está sendo verificado. Isso facilita a manutenção e a colaboração na equipe.

ii. Teste unitário usando Jest

Os testes unitários são geralmente escritos usando frameworks de teste, como Jest, Mocha ou Jasmine. Aqui utilizaremos o framework Jest.

Um teste unitário consiste em **enviar uma mensagem** para a unidade a ser testada e verificar se tem o retorno previsto. No TS, o termo **enviar uma mensagem** significa chamar o método/função - passando os parâmetros - e verificar se a resposta confere com o previsto.

A seguir tem-se a função **somar** sendo testada em três situações distintas e, para cada situação de teste, o resultado da chamada da função está sendo comparado com o valor esperado.

Código de teste	Resultado do teste
<pre>// Código a ser testado function somar(a: number, b: number): number { return a + b; } // Três testes unitários da função somar test("soma com valores positivos", () => { // Espera-se que o resultado de somar(2, 3) seja 5 expect(somar(2, 3)).toBe(5); }); test("soma com valores negativos", () => { expect(somar(-2, -3)).toBe(-5); }); test("soma com zeros", () => { expect(somar(0, 0)).toBe(0); });</pre>	<div>  Test run <ul style="list-style-type: none">  teste de soma com valores positivos  teste de soma com valores negativos  teste de soma com zeros </div>

Estrutura de um teste:

```
test("descrição do teste", () => {
  // Aqui dentro precisa ter a chamada da função expect
});
```

- test: função global fornecida pelo Jest. Ela recebe:
 - 1º argumento: string com a **descrição do teste** a ser exibida ao rodar os testes;
 - 2º argumento: função call-back, no corpo dessa função mantemos as instruções que definem o teste.

- `expect`: é usada toda vez que queremos testar um valor. Ela é usada junto com uma função `matcher` para verificar algo sobre um valor. Como exemplo:

```
expect(somar(2, 3)).toBe(5);
```

A expectativa é definida usando `expect(somar(2, 3))`, onde `somar(2, 3)` é a chamada da função que estamos testando. Desta forma, temos `expect(5)` após processar `somar(2, 3)`.

A função `toBe(5)` é um `matcher` que verifica se o valor é igual ao argumento da função `expect`, ou seja, o resultado do teste será positivo se os valores forem iguais.

Embora tecnicamente seja possível colocar várias expectativas dentro de um único teste, geralmente não é considerado uma boa prática por algumas razões:

```
test("teste de soma", () => {  
  expect(somar(2, 3)).toBe(5);  
  expect(somar(-2, -3)).toBe(-5);  
  expect(somar(0, 0)).toBe(0);  
});
```

- Clareza do teste: um teste deve ser claro e focado em **uma única condição ou comportamento**. Quando há várias expectativas em um teste, pode ser difícil determinar qual expectativa falhou se o teste não passar. A clareza do teste é fundamental para a identificação rápida e correção de problemas;
- Facilidade de manutenção: testes devem ser fáceis de manter e entender. Se um teste contiver muitas expectativas, ele pode se tornar complexo e difícil de modificar ou adaptar conforme o código evolui;
- Teste de contrato único: cada teste deve verificar um contrato específico ou comportamento da função ou componente que está sendo testado. Múltiplas expectativas podem indicar que o teste está tentando verificar coisas diferentes, o que pode tornar a finalidade do teste menos clara;
- Isolamento de falhas: ao dividir testes em casos individuais, é mais fácil isolar falhas. Se um teste com várias expectativas falhar, podemos não saber imediatamente qual expectativa causou o problema.

É uma prática recomendada escrever testes mais granulares, cada um focado em um aspecto específico do comportamento da função/método. Isso melhora a organização, a manutenção e a legibilidade dos testes. Se desejamos testar diferentes cenários, então devemos criar testes separados para cada um deles.

A abrangência dos testes é importante para garantir que o código seja robusto e confiável em várias situações. No exemplo anterior a função `somar` foi testada em apenas três situações, pelo fato de julgarmos que essas situações seriam suficientes para garantir a qualidade da função.

Antes de codificar os testes é importante saber quais funções e métodos serão testados e em quais condições, pois não existe a necessidade de testar todas as funções e métodos. Considere os seguintes pontos antes de codificar os testes:

- A principal regra para saber o que testar é:
 - Imaginar as condições de contorno e testar nesses limites, por exemplo, um método que válida se a idade é “de menor”. Bastaria testar os parâmetros 17, 18 e 19, ou seja, não precisa testar outros valores;

- Comece pelos mais simples e deixe os testes complexos para o final;
- Use apenas dados suficientes, ou seja, não teste 10 condições, se apenas 3 forem o suficiente;
- Não teste funções e métodos triviais.

Algumas considerações relacionadas à quantidade de testes:

- Cobertura funcional: garantir que diferentes aspectos funcionais da função ou método sejam testados é crucial. Isso inclui testar diferentes caminhos de execução, manipulação de entradas e tratamento de saídas;
- Cobertura de casos de borda: verificar o comportamento da função em situações extremas ou limites. Por exemplo, testar valores nulos, valores máximos ou mínimos permitidos;
- Cobertura de erros: verificar como a função lida com condições de erro é fundamental. Isso inclui entradas inválidas, lançamento de exceções e comportamento em situações inesperadas;
- Repetição de testes: repetir testes com diferentes conjuntos de dados ou contextos pode ajudar a identificar possíveis problemas de desempenho ou situações não consideradas anteriormente.

No Jest, as funções `test` e `it` são equivalentes e servem ao mesmo propósito: definir um caso de teste. A principal diferença é puramente semântica, sendo apenas uma questão de preferência de escrita.

Por exemplo, os dois casos de teste a seguir são equivalentes:

```
test("soma com valores positivos", () => {
  expect(somar(2, 3)).toBe(5);
});
```

```
test("soma com zeros", () => {
  expect(somar(0, 0)).toBe(0);
});
```

```
it("soma com valores positivos", () => {
  expect(somar(2, 3)).toBe(5);
});
```

```
it("soma com zeros", () => {
  expect(somar(0, 0)).toBe(0);
});
```

iii. Matchers e modificadores no Jest

Os matchers são funções fornecidas pelo framework para fazer afirmações (asserções) nos testes. Matchers são funções que encadeamos à expectativa para verificar se o valor atende a uma determinada condição (<https://jestjs.io/pt-BR/docs/expect>). Quando Jest é executado, ele rastreia todos os matchers que falharam para imprimir as mensagens de erro de uma forma agradável.

Exemplos de funções usadas como matchers no Jest:

- `toBe`: usado para comparar valores primitivos. Verifica se dois valores são estritamente iguais (usando `===`):

```
expect("5").toBe("5");
```

- `toBeCloseTo`: usado para comparar valores aproximados.

Em muitas linguagens de programação, incluindo JavaScript, os números de ponto flutuante são representados em binário e podem não ter uma precisão exata para certos valores. No caso específico de $0.2 + 0.1$, o resultado é aproximadamente `0.30000000000000004`, não exatamente `0.3`. Ao usar o matcher `toBe`, ele verifica a igualdade estrita (`===`), o que leva à falha do teste:

```
expect(0.2 + 0.1).toBe(0.3);
```

O matcher `toBeCloseTo` permite uma pequena imprecisão e é mais apropriado para comparações de números de ponto flutuante. A precisão padrão é de 4 casas decimais, mas podemos especificar a precisão desejada, se necessário:

```
expect(0.2 + 0.1).toBeCloseTo(0.3);
expect(4/3).toBeCloseTo(1.3, 1); // testa considerando 1 casa decimal
```

- `toEqual`: verifica se dois objetos têm as mesmas propriedades e valores. Ele faz uma comparação profunda entre todas as propriedades dos objetos aninhados:

```
expect({nome: "Ana", carro: {modelo: "Uno"}}).toEqual({carro: {modelo: "Uno"}, nome: "Ana"});
```

- Verificam intervalos:

```
expect(5).toBeLessThan(10);
expect(10).toBeGreaterThanOrEqual(10);
```

- Verificam valores específicos:

```
expect(0/0).toBeNaN(); // 0/0 é NaN
expect(null).toBeNull();
expect(undefined).toBeUndefined();
expect(true).toBeTruthy();
expect(false).toBeFalsy();
```

- `toHaveLength`: verifica se o valor da propriedade `length` do objeto é igual ao valor fornecido:

```
expect(["a", "b", "c"]).toHaveLength(3); // o array possui 3 elementos
expect("abcde").toHaveLength(5); // a string possui 5 elementos
```

- `toMatch`: verifica se uma string corresponde a um padrão de expressão regular:

```
expect("Teste unitário com Jest").toMatch(/com/);
// O flag i na expressão regular indica que a correspondência
// deve ser feita de forma insensível a maiúsculas e minúsculas
expect("Teste unitário com Jest").toMatch(/Com/i);
```

- `toThrow`: verifica se uma função/método lança exceção:

```
function incrementar(nro:number):number | never {
  if( nro < 0 ){
    throw new Error("Número negativo");
  }
  return nro + 1;
}
```

```
test("teste de exceção", () => {
  expect(
    () => incrementar(-1)
  ).toThrow("Número negativo");
});
```

```
test("teste de sucesso", () => {
  expect(incrementar(2)).toBe(3);
});
```

Observação: precisamos colocar a chamada da função incrementar(-1) dentro de uma função, caso contrário, o erro não será “pego” e a asserção falhará. Nesse exemplo, usamos uma arrow function, mas poderia ter sido usada uma função anônima.

Para obter outros matchers consulte <https://jestjs.io/pt-BR/docs/expect>.

Modificadores: além dos matchers, podemos encadear propriedades adicionais para modificar ou verificar mais propriedades do valor. Existem apenas três modificadores:

- not: nega a afirmação feita pelo matcher subsequente. Neste exemplo, not atua como um modificador que nega a condição do matcher toBe, indicando que esperamos que o valor não seja estritamente igual a "5":

```
expect(5).not.toBe("5");
```

- resolves e reject: é usado para testar funções assíncronas e verificar se uma promessa (Promise) é resolvida ou rejeitada (lança uma exceção).

As propriedades resolves e rejects são usadas para decodificar o valor de uma promessa cumprida ou rejeitada, antes de qualquer matcher ser encadeado.

Como estamos testando promises, então o teste é assíncrono. Por este motivo, temos de usar `await` na instrução de teste e, por consequência, a função call-back precisa ser `async`:

```
async function calc(nro: number) : Promise<number> {  
  if (nro < 0) {  
    throw new Error("Número negativo");  
  }  
  return nro * 2;  
}  
  
// A função callback precisa ser async, visto que precisamos esperar  
// a promise ser resolvida/rejeitada  
test('dobro do número para valores positivos', async () => {  
  // Só obteremos sucesso no teste se a promise for cumprida  
  await expect(calc(5)).resolves.toBe(10);  
});  
  
test('lança uma exceção para número negativo', async () => {  
  // Usamos rejects para testar se a função lança uma exceção.  
  // Só obteremos sucesso no teste se a promise for rejeitada  
  await expect(calc(-5)).rejects.toThrow("Número negativo");  
});
```

iv. Grupo de testes

A função describe é usada para agrupar hierarquicamente os testes em blocos, tornando a organização mais clara.

Estrutura da função describe:

```
describe("descrição do bloco", () => {  
  // Aqui dentro precisa ter as chamadas das funções test e describe  
});
```


No exemplo a seguir o teste com a descrição `testa undefined` não está dentro de nenhum bloco, já o bloco com a descrição `teste de intervalos` está aninhado dentro do bloco `teste com números`. A forma como os resultados são apresentados depende do ambiente de exibição, neste exemplo os resultados foram exibidos no terminal:

Código de teste:

```
describe("teste com números", () => {
  test("soma de float", () => {
    expect(5.2 + 4.5).toBeCloseTo(9.7);
  });

  describe("teste de intervalos", () => {
    test("menor", () => {
      expect(5).toBeLessThan(5.1);
    });

    test("maior", () => {
      expect(5.1).toBeGreaterThan(5);
    });
  });
});

describe("texto", () => {
  test("soma de float", () => {
    expect("a" + "b").toBe("ab");
  });

  test("busca", () => {
    expect("abcdef").toMatch(/D/i);
  });
});

test("testa undefined", () => {
  expect(undefined).toBeUndefined();
});
```

Resultado do teste no terminal:

```
PASS test/Exemplo.test.ts
  ✓ testa undefined
  teste com números
    ✓ soma de float (4 ms)
    teste de intervalos
      ✓ menor (1 ms)
      ✓ maior
  texto
    ✓ soma de float
    ✓ busca (1 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:  0 total
Time:        1.483 s, estimated 2 s
Ran all test suites.
```

Estrutura de pastas do projeto:

```

v SERVER
  > node_modules
  > src
  v test
    TS Exemplo.test.ts
  .env
  .gitignore
  package-lock.json
  package.json
  tsconfig.json
```

O uso do `describe` é especialmente útil quando temos muitos testes e desejamos organizá-los de forma hierárquica. Cada bloco `describe` serve como um nível na hierarquia de organização dos testes. Isso facilita a execução seletiva de testes específicos ou grupos de testes durante o desenvolvimento ou a execução de testes em um ambiente de integração contínua.

O arquivo é uma suíte de testes e a função `describe` também define uma suíte de testes dentro do arquivo. No exemplo anterior o arquivo `Exemplo.test.ts` define uma suíte de testes e a função `describe` pode ser considerada uma suíte dentro desse arquivo.

v. Hooks de ciclo de vida do teste

As funções `beforeAll`, `beforeEach`, `afterAll` e `afterEach` são hooks de ciclo de vida que permitem realizar ações específicas antes ou depois da execução dos testes. Essas funções são úteis para configurar e limpar o ambiente de teste, preparando-o para a execução dos testes:

- `beforeAll`: é executada uma vez antes de todos os testes na suíte. Ela é útil para ações que precisam ser realizadas apenas uma vez antes do início dos testes;
- `afterAll`: é executada uma vez após a conclusão de todos os testes na suíte. É útil para realizar ações de limpeza que devem ocorrer apenas uma vez após todos os testes;
- `beforeEach`: é executada antes de cada teste na suíte. Ela é usada para configurar o ambiente antes de cada teste, garantindo um estado inicial consistente;
- `afterEach`: é executada após a conclusão de cada teste na suíte. Ela é usada para limpar ou redefinir o ambiente após a execução de cada teste.

Observe no exemplo a seguir que as funções `beforeAll` e `afterAll` são, respectivamente, a 1ª e última função a serem executadas antes de todos os testes do arquivo (o arquivo define uma suíte). As funções `beforeAll` e `afterAll` definidas dentro da `describe` são executadas, respectivamente, antes e depois de todos os testes que existem dentro da suíte definida pela função `describe`.

Código para testar as funções `beforeAll` e `afterAll`:

```
beforeAll(() => {
  console.log("antes de tudo");
});

afterAll(() => {
  console.log("depois de tudo");
});

// a função describe define uma suíte de teste
describe("texto", () => {
  beforeAll(() => {
    console.log("antes de tudo - describe");
  });

  afterAll(() => {
    console.log("depois de tudo - describe");
  });

  test("soma de float", () => {
    console.log("soma de float");
    expect("a" + "b").toBe("ab");
  });

  test("busca", () => {
    console.log("busca");
    expect("abcdef").toMatch(/D/i);
  });
});
```

Resultado do teste no terminal:

```
> Test run started
console.log
  antes de tudo ← beforeAll
console.log
  antes de tudo - describe ← beforeAll da suíte
console.log
  soma de float ← teste dentro da suíte
console.log
  busca ← teste dentro da suíte
console.log
  depois de tudo - describe ← afterAll da suíte
console.log
  testa undefined ← teste fora da suíte
console.log
  depois de tudo ← afterAll
PASS test/Exemplo.test.ts
  ✓ testa undefined (2 ms)
  texto
    ✓ soma de float (6 ms)
    ✓ busca (1 ms)

Test Suites: 1 passed, 1 total
Tests:      3 passed, 3 total
Snapshots:  0 total
```

```
});

test("testa undefined", () => {
  console.log("testa undefined");
  expect(undefined).toBeUndefined();
});
```

Como exemplo, as funções `beforeAll` e `afterAll` foram definidas dentro da suíte definida pela função `describe`. Veja que as funções `beforeAll` e `afterAll` são executadas, respectivamente, antes e após cada um dos testes que existem dentro do bloco da função `describe`.

Código para testar as funções `beforeEach` e `afterEach`:

```
beforeAll(() => {
  console.log("antes de tudo");
});

afterAll(() => {
  console.log("depois de tudo");
});

// a função describe define uma suíte de teste
describe("texto", () => {
  beforeAll(() => {
    console.log("antes de tudo - describe");
  });

  afterAll(() => {
    console.log("depois de tudo - describe");
  });

  beforeEach(() => {
    console.log("antes de cada - describe");
  });

  afterEach(() => {
    console.log("depois de cada - describe");
  });

  test("soma de float", () => {
    console.log("soma de float");
    expect("a" + "b").toBe("ab");
  });

  test("busca", () => {
    console.log("busca");
    expect("abcdef").toMatch(/D/i);
  });
});
```

Resultado do teste no terminal:

```
> Test run started
console.log
  antes de tudo ← beforeAll
console.log
  antes de tudo - describe ← beforeAll da suíte
console.log
  antes de cada - describe } Para cada
                           } teste são
                           } executadas
                           } as funções:
                           } beforeEach
                           } test
                           } afterEach
console.log
  soma de float
console.log
  depois de cada - describe } Para cada
                           } teste são
                           } executadas
                           } as funções:
                           } beforeEach
                           } test
                           } afterEach
console.log
  antes de cada - describe }
console.log
  busca
console.log
  depois de cada - describe }
console.log
  depois de tudo - describe ← afterAll da suíte
console.log
  testa undefined ← teste fora da suíte
console.log
  depois de tudo ← afterAll

PASS test/Exemplo.test.ts
✓ testa undefined (1 ms)
texto
✓ soma de float (6 ms)
✓ busca (3 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
```

```
});

test("testa undefined", () => {
  console.log("testa undefined");
  expect(undefined).toBeUndefined();
});
```

Se a função call-back passada como parâmetro para `beforeAll` e `beforeEach` retornar uma promise, o Jest vai esperar até que a Promise ser resolvida para executar os testes.

vi. Teste de função objetivo de requisição HTTP

A função objetivo em uma requisição HTTP é aquela função responsável por processar a requisição e responder ao cliente. A função objetivo recebe os objetos Request e Response como argumento. Lembre-se que esses objetos são criados pelo servidor Express ao receber a requisição do cliente.

Qualquer teste de uma função envolve chamar a função passando parâmetros e conferir se a função retorna o valor esperado. Acontece que os argumentos de uma função objetivo são embutidos nas propriedades `body` e `params` do objeto Request e o resultado está nas propriedades `json` e `send` do objeto Response. Desta forma, teremos de simular os objetos Request e Response para testarmos os métodos `sum` e `power` da classe `OperacaoController`.

Código do arquivo `src/controllers/OperacaoController.ts`:

```
import { Request, Response } from "express";

export class OperacaoControlador {
  public async sum(req: Request, res: Response): Promise<Response> {
    let { x, y } = req.body;
    if (x && y) {
      x = parseInt(x);
      y = parseInt(y);
      if (isNaN(x) || isNaN(y)) {
        return res.json({ message: "Parâmetros não são números" });
      }
      return res.json({ r: x + y });
    }
    return res.json({ message: "Parâmetros inválidos" });
  }

  public async power(req: Request, res: Response): Promise<Response> {
    let { x, y }: any = req.params;
    x = parseInt(x);
    y = parseInt(y);
    if (isNaN(x) || isNaN(y)) {
      return res.send({ message: "Parâmetros não são números" });
    }
    return res.send({ r: x ** y });
  }
}
```

Estrutura do projeto:

```

✓ SERVIDOR
  > node_modules
  ✓ src
    ✓ controllers
      TS OperacaoController.ts
    ✓ routes
      TS index.ts
      TS operacao.ts
      TS index.ts
  ✓ test
    TS Operacao.test.ts
  .env
  .gitignore
  {} package-lock.json
  {} package.json
  TS tsconfig.json
```

```
}

export default new OperacaoControlador();
```

Os testes dos métodos `sum` e `power` precisam cobrir todos os caminhos de respostas dentro dos métodos. Por este motivo foram necessários 3 testes para o método `sum` e 2 testes para o método `power`.

Para construir um objeto do tipo Request usamos um JSON com a propriedade `body` (teste do método `sum`) e `params` (teste do método `power`). Nesses objetos JSON estão os dados que o cliente enviaria na requisição:

```
const req = { body: { x: 2, y: 3 } } as Request; // para testar o método sum
const req = { params: { x: "2", y: "3" } } as unknown as Request; // para testar o método power
```

`jest.fn()` é usado para criar uma função espiã que registra chamadas, argumentos e retornos. Aqui `jest.fn` está sendo usada para assistir as propriedades `json` e `send` do objeto Response.

```
const res = { json: jest.fn() } as unknown as Response; // para testar o método sum
const res = { send: jest.fn() } as unknown as Response; // para testar o método power
```

Para chamar as funções `sum` e `power` passamos os objetos que estão nas variáveis `req` e `res`:

```
await controlador.sum(req, res);
await controlador.power(req, res);
```

Para comparar o resultado esperado usamos as instruções a seguir. O método `sum` coloca o resultado na propriedade `json` e o método `power` coloca o resultado na propriedade `send`, então temos de verificar se o resultado esperado está nessas propriedades:

```
expect(res.json).toHaveBeenCalledWith(
  expect.objectContaining({ r: 5 })
);

expect(res.send).toHaveBeenCalledWith(
  expect.objectContaining({ r: 8 })
);
```

Código do arquivo `test/Operacao.test.ts`:

```
import { Request, Response } from "express";
import controlador from "../src/controllers/OperacaoController";

describe("somar", () => {
  test("sum - calcular a soma", async () => {
    // Objeto simulado para representar uma requisição HTTP
    const req = { body: { x: 2, y: 3 } } as Request;
    // Objeto simulado para representar uma resposta HTTP.
    // unknown é um tipo mais amplo que aceita qualquer valor,
    // sem fornecer muita informação sobre sua estrutura
    const res = { json: jest.fn() } as unknown as Response;
    // Chama o sum passando os objetos simulados Request e Response
    await controlador.sum(req, res);
    // O método toHaveBeenCalledWith é usado para verificar se
    // o método json foi chamado com o argumento { r: 5 }
```

```

    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ r: 5 })
    );
  });

test("sum - mensagem de parâmetros inválidos", async () => {
  const req = { body: {} } as Request;
  const res = { json: jest.fn() } as unknown as Response;

  await controlador.sum(req, res);

  expect(res.json).toHaveBeenCalledWith(
    expect.objectContaining({ message: "Parâmetros inválidos" })
  );
});

test("sum - mensagem de parâmetros não são números", async () => {
  const req = { body: { x: 2, y: "y" } } as Request;
  const res = { json: jest.fn() } as unknown as Response;

  await controlador.sum(req, res);

  expect(res.json).toHaveBeenCalledWith(
    expect.objectContaining({ message: "Parâmetros não são números" })
  );
});

describe("potência", () => {
  test("power - calcular a potência", async () => {
    // O método power recebe os dados como parâmetro
    const req = { params: { x: "2", y: "3" } } as unknown as Request;
    // O método power retorna os dados na propriedade send
    const res = { send: jest.fn() } as unknown as Response;

    await controlador.power(req, res);

    expect(res.send).toHaveBeenCalledWith(
      expect.objectContaining({ r: 8 })
    );
  });

  test("power - mensagem de parâmetros inválidos", async () => {
    const req = { params: { x: "a", y: 3 } } as unknown as Request;
    const res = { send: jest.fn() } as unknown as Response;

    await controlador.power(req, res);

    expect(res.send).toHaveBeenCalledWith(

```

```
    expect.objectContaining({ message: "Parâmetros não são números" })
  });
});
```

Resultado do teste:

```
PASS test/Operacao.test.ts
  somar
    ✓ sum - calcular a soma (5 ms)
    ✓ sum - mensagem de parâmetros inválidos
    ✓ sum - mensagem de parâmetros não são números
  potência
    ✓ power - calcular a potência
    ✓ power - mensagem de parâmetros inválidos
```

vii. Teste de CRUD no MongoDB

Para testar os métodos que fazem as operações de CRUD (Create, Read, Update e Delete) em um BD, recomenda-se usarmos uma instância temporária, também chamada, em memória. Essa instância temporária proporciona um ambiente de teste mais controlado, eficiente e independente, contribuindo para testes mais rápidos, confiáveis e portáteis.

A seguir tem-se o código a ser testado. Ele é usado para persistir os dados na coleção cars do MongoDB. O código para criar o esquema e modelo da coleção foi apresentado na aula anterior.

cars	
id:	INTEGER
model:	VARCHAR(15)

Código do arquivo src/controllers/CarController.ts:

```
import { Request, Response } from "express";
import { Car } from "../models";

export class CarController {
  public async create(req: Request, res: Response): Promise<Response> {
    const { model } = req.body;
    try {
      const document = new Car({ model });
      const resp = await document.save();
      return res.json(resp);
    } catch (error: any) {
      if (error.code === 11000 || error.code === 11001) {
        return res.json({ message: "Este modelo já está em uso" });
      } else if (error && error.errors["model"]) {
        return res.json({ message: error.errors["model"].message });
      }
      return res.json({ message: error.message });
    }
  }

  public async list(_: Request, res: Response): Promise<Response> {
    try {
      const objects = await Car.find().sort({ model: "asc" });
      return res.json(objects);
    }
  }
}
```

```

    } catch (error: any) {
      return res.json({ message: error.message });
    }
  }

  public async delete(req: Request, res: Response): Promise<Response> {
    const { id: _id } = req.body; // _id do registro a ser excluído
    try {
      const object = await Car.findByIdAndDelete(_id);
      if (object) {
        return res.json({ message: "Registro excluído com sucesso" });
      } else {
        return res.json({ message: "Registro inexistente" });
      }
    } catch (error: any) {
      return res.json({ message: error.message });
    }
  }

  public async update(req: Request, res: Response): Promise<Response> {
    const { id, model } = req.body;
    try {
      const document = await Car.findById(id);
      if (!document) {
        return res.json({ message: "Carro inexistente" });
      }
      document.model = model;
      const resp = await document.save();
      return res.json(resp);
    } catch (error: any) {
      if (error.code === 11000 || error.code === 11001) {
        return res.json({ message: "Este modelo já está em uso" });
      } else if (error && error.errors["model"]) {
        return res.json({ message: error.errors["model"].message });
      }
      return res.json({ message: error.message });
    }
  }
}

export default new CarController();

```

Para fazer os testes usamos o mongodb-memory-server (<https://github.com/nodkz/mongodb-memory-server>) para criar um servidor MongoDB em memória exclusivo para cada teste. Na função `beforeAll` criamos a instância do BD e na `afterAll` destruímos. O restante do código funciona normalmente, porém os métodos da classe `CarController` usarão a conexão com o MongoDB criada na função `beforeAll`, visto que o Mongoose utiliza a última conexão aberta com o MongoDB. Antes de cada teste limparemos a coleção `cars` para garantir que os testes não sejam contaminados.

Código do arquivo `test/Car.test.ts`:


```
import { Request, Response } from "express";
import carController from "../src/controllers/CarController";
import mongoose from "mongoose";
import { Car } from "../src/models";
import { MongoMemoryServer } from "mongodb-memory-server";

// Global para ser acessado dentro do beforeEach e afterEach
let mongoServer: MongoMemoryServer;

beforeAll(async () => {
  // Criar e inicializar uma instancia do MongoMemoryServer
  mongoServer = await MongoMemoryServer.create();
  // Obter a URI do servidor de memória
  const uri = mongoServer.getUri();
  await mongoose.connect(uri);
});

beforeEach(async () => {
  await Car.deleteMany({}); // Limpa a coleção cars
});

afterAll(async () => {
  await mongoose.disconnect();
  // Parar o servidor de memória
  await mongoServer.stop();
});

describe("CarController - create", () => {
  test("criar um carro com sucesso", async () => {
    // Fornece a propriedade model para criar um carro
    const req = { body: { model: "Uno" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.create(req, res);

    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ model: "Uno" })
    );
  });

  test("mensagem de modelo obrigatório", async () => {
    // Não está sendo fornecido o parâmetro model
    const req = { body: {} } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.create(req, res);

    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ message: "O modelo é obrigatório" })
    );
  });
});
```

```

    });
  });

test("mensagem de modelo ter no máximo 15 caracteres", async () => {
  const req = { body: { model: "1234567890123456" } } as Request;
  const res = { json: jest.fn() } as unknown as Response;

  await carController.create(req, res);

  expect(res.json).toHaveBeenCalledWith(
    expect.objectContaining({
      message: "O modelo pode ter no máximo 15 caracteres",
    })
  );
});

test("mensagem de modelo em uso ao carro com modelo duplicado", async () => {
  // Adiciona um documento na coleção para simular um modelo duplicado.
  // Car é um modelo do Mongoose. A partir do modelo é criado um documento e inserido na
  coleção do BD
  await Car.create({ model: "Pampa" });

  const req = { body: { model: "Pampa" } } as Request;
  const res = { json: jest.fn() } as unknown as Response;

  await carController.create(req, res);

  expect(res.json).toHaveBeenCalledWith({
    message: "Este modelo já está em uso",
  });
});

describe("CarController - list", () => {

  test("listar os carros com sucesso", async () => {
    // Adiciona 3 documentos na coleção
    await Car.create([
      { model: "Fusca" },
      { model: "Corcel" },
      { model: "Opala" },
    ]);

    const req = {} as Request;
    const res = { json: jest.fn() } as unknown as Response;
    // Chama o método list do carController
    await carController.list(req, res);
  });
});

```

```
// Verifica se a resposta do método json foi chamada com os objetos esperados, incluindo a
ordem
expect(res.json).toHaveBeenCalledWith([
  expect.objectContaining({ model: "Corcel" }),
  expect.objectContaining({ model: "Fusca" }),
  expect.objectContaining({ model: "Opala" }),
]);
});

test("retorna um array vazio quando a coleção não possui documentos", async () => {
  const req = {} as Request;
  const res = { json: jest.fn() } as unknown as Response;

  await carController.list(req, res);

  // Verifica se a resposta do método json foi chamada com um array vazio
  expect(res.json).toHaveBeenCalledWith([]);
});

describe("CarController - delete", () => {
  test("excluir documento com sucesso", async () => {
    // Adiciona um documento na coleção
    const document = await Car.create({ model: "Gol" });

    const req = { body: { id: document._id } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.delete(req, res);

    // Verifica se a resposta do método json foi chamada com a mensagem esperada
    expect(res.json).toHaveBeenCalledWith({
      message: "Registro excluído com sucesso",
    });
  });

  test("mensagem de registro inexistente ao excluir documento que não existe", async () => {
    // Objeto Request com um _id que não existe na coleção
    const req = { body: { id: "65ad9cd937249b0b77a4e343" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.delete(req, res);

    // Verifica se a resposta do método json foi chamada com a mensagem esperada
    expect(res.json).toHaveBeenCalledWith({ message: "Registro inexistente" });
  });
});
```

```
describe("CarController - update", () => {
  test("atualiza documento com sucesso", async () => {
    // Adiciona um documento na coleção
    const document = await Car.create({ model: "Gol" });

    const req = { body: { id: document._id, model: "Polo" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.update(req, res);

    // Verifica se a resposta do método json foi chamada com a mensagem esperada
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ model: "Polo" })
    );
  });

  test("mensagem de registro inexistente ao atualizar documento que não existe", async () => {
    // Objeto Request com um _id que não existe na coleção
    const req = { body: { id: "65ad9cd937249b0b77a4e343" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.update(req, res);

    // Verifica se a resposta do método json foi chamada com a mensagem esperada
    expect(res.json).toHaveBeenCalledWith({ message: "Carro inexistente" });
  });

  test("mensagem de modelo em uso ao atualizar um carro com modelo duplicado", async () => {
    // Adiciona 2 documentos na coleção
    await Car.create({ model: "Saveiro" });
    const document = await Car.create({ model: "D20" });

    const req = { body: { id: document._id, model: "Saveiro" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.update(req, res);

    expect(res.json).toHaveBeenCalledWith({
      message: "Este modelo já está em uso",
    });
  });

  test("mensagem de modelo obrigatório ao atualizar carro", async () => {
    const document = await Car.create({ model: "C10" });

    const req = { body: { id: document._id, model: "" } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await carController.update(req, res);
```

```
expect(res.json).toHaveBeenCalledWith({
  message: "O modelo é obrigatório",
});
});
});
```

Resultado do teste:

```
PASS test/Car.test.ts
  CarController - create
    ✓ criar um carro com sucesso (22 ms)
    ✓ mensagem de modelo obrigatório (1 ms)
    ✓ mensagem de modelo ter no máximo 15 caracteres
    ✓ mensagem de modelo em uso ao carro com modelo duplicado (8 ms)
  CarController - list
    ✓ listar os carros com sucesso (12 ms)
    ✓ retorna um array vazio quando a coleção não possui documentos (3 ms)
  CarController - delete
    ✓ excluir documento com sucesso (3 ms)
    ✓ mensagem de registro inexistente ao excluir documento que não existe (1 ms)
  CarController - update
    ✓ atualiza documento com sucesso (7 ms)
    ✓ mensagem de registro inexistente ao atualizar documento que não existe (2 ms)
    ✓ mensagem de modelo em uso ao atualizar um carro com modelo duplicado (5 ms)
    ✓ mensagem de modelo obrigatório ao atualizar carro (4 ms)
```

Vantagens do mongodb-memory-server em testes comparado com o uso de uma instância do MongoDB real:

- Isolamento do ambiente: o mongodb-memory-server cria um servidor MongoDB em memória exclusivo para cada execução de teste. Isso evita interferências entre testes, garantindo que cada teste tenha um ambiente de BD isolado;
- Velocidade: o servidor em memória é mais rápido do que um servidor MongoDB tradicional, uma vez que os dados são manipulados na RAM, sem a sobrecarga de E/S no disco;
- Ausência de dependência externa: o uso do mongodb-memory-server elimina a dependência de um servidor MongoDB externo. Isso facilita a execução dos testes em ambientes que podem não ter uma instância do MongoDB disponível ou podem exigir permissões específicas para acessar o BD;
- Fácil configuração: o mongodb-memory-server é fácil de configurar e não requer a configuração de uma instância do MongoDB separada para os testes. Isso simplifica o processo de configuração e execução dos testes, tornando-os mais portáteis e menos propensos a erros de configuração;
- Descarte automático dos dados: o servidor em memória é descartado automaticamente quando o processo de teste é encerrado, garantindo que não haja resíduos de dados entre execuções de testes;
- Não requer rede externa: a execução dos testes com mongodb-memory-server não requer uma conexão com a rede externa, tornando-os adequados para cenários onde a conectividade de rede pode ser um problema ou quando se deseja realizar testes offline.

Exercício

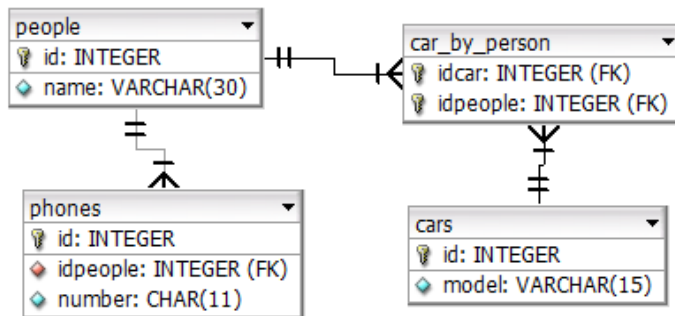
Veja os vídeos se tiver dúvidas nos exercícios:

Explicação da aula - <https://youtu.be/2ITOpjKGgIA>

Exercício - <https://youtu.be/w7zlvhh4SQU>

Exercício 1 – Na aula anterior foi criada uma aplicação servidora para persistir no MongoDB os dados representados no modelo a seguir. No exemplo anterior foi codificado os testes dos métodos da classe CarController.

No âmbito desse exercício, codifique os testes dos métodos das classes PersonController e CarByPersonController. Ao lado tem-se a estrutura esperada para o projeto.



```

v SERVERIOR
  > node_modules
  v src
    v controllers
      TS CarByPersonController.ts
      TS CarController.ts
      TS PersonController.ts
      TS PhoneController.ts
    v models
      TS connection.ts
      TS index.ts
    > routes
      TS index.ts
  v test
    v controllers
      TS Car.test.ts
      TS CarByPerson.test.ts
      TS Person.test.ts
  .env
  .gitignore
  {} package-lock.json
  {} package.json
  tsconfig.json
  
```