Objetivos:

I. Mock, spy e stub;
II. Mock de classe;
III. Mock de objeto;
IV. Mock de função;
V. Exemplo de uso de mock.

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

a) Crie uma pasta de nome aula (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 `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo tsconfig.json);
d) No terminal, execute o comando `npm i -D jest ts-jest @types/jest` para instalar os pacotes necessários para rodar o jest;
e) Crie as pastas `src` e `test` no projeto assim como é mostrado ao lado;
f) Coloque as seguintes propriedades no arquivo package.json para configurar os testes:

Estrutura de pastas e arquivos do projeto:


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

i. Mock, spy e stub

O principal objetivo do mock nos testes unitários é isolar o código que está sendo testado de suas dependências. Isso nos permite testar o código em si, sem se preocupar com o comportamento de outras partes do sistema.

Benefícios do uso de mocks:

- Testes mais rápidos: mocks podem ser muito mais rápidos do que as dependências reais, o que pode reduzir significativamente o tempo de execução dos testes;
- Testes mais precisos: mocks podem ser configurados para simular comportamentos específicos, o que permite testar cenários específicos que podem ser difíceis ou impossíveis de reproduzir com as dependências reais;

- Testes mais confiáveis: mocks podem ajudar a evitar "*testes flaky*" - são testes que apresentam resultados inconsistentes, ou seja, passam ou falham aleatoriamente, mesmo quando o código e o ambiente de teste não mudaram.

Exemplos de uso de mocks:

- Testar uma função que utiliza uma chamada de um BD. Podemos usar um mock para simular o BD e verificar se a função está fazendo as chamadas corretas;
- Testar uma função que usa uma API externa. Podemos usar um mock para simular a API e verificar se a função está processando os dados corretamente.

Testes usando simulação envolvem:

a) **Mock:** é uma implementação simulada de uma função ou método que podemos controlar durante o teste. Podemos definir o comportamento esperado do mock e verificar se ele foi chamado com os argumentos corretos.

Considere a classe `Operacao` e a função `somarArray` como exemplos.

src/Operacao.ts

```
export default class Operacao {
  somar(a: number, b: number) {
    return a + b;
  }

  incrementar(nro: number){
    return nro + 1;
  }
}
```

src/array.ts

```
import Operacao from "./Operacao";

export default function somarArray(vet:number[]){
    const operacao = new Operacao();
    let r = 0;
    for(let i = 0; i < vet.length; i++){
        r = operacao.somar(r,vet[i]);
    }
    return r;
}
```

O código a seguir cria um mock do método `somar` da classe `Operacao`. Desta forma, o teste da função `somarArray` não utilizará o método real implementado no módulo src/Operacao, mas a implementação simulada criada usando `jest.mock`.

test/mock.test.ts

```
import somarArray from "../src/array";

jest.mock("../src/Operacao", () => {
  const mOperacao = {
    somar: jest.fn().mockImplementation( (a, b) => a + b )
  };
  return jest.fn( () => mOperacao );
});

it("Somar array", () => {
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(10);
});
```

Características de um mock:

- Um mock é um objeto falso que simula o comportamento de uma dependência real;

- Mocks podem ser configurados para esperar chamadas específicas e retornar valores específicos;
- Mocks são usados para testar se o código está interagindo com suas dependências da maneira correta.

b) **Spy:** é um objeto que observa as chamadas feitas a um objeto real. Spies podem ser usados para verificar a quantidade de vezes que uma função/método foi chamada e quais os argumentos usados nessas chamadas.

No teste a seguir usamos `jest.spyOn` para criar um spy no método `somar` da classe `Operacao`. Em seguida, estamos chamando a função `somarArray` e verificando se o método `somar` foi chamado 4 vezes e com os argumentos esperados.

test/spy.test.ts

```
import somarArray from "../src/array";
import Operacao from "../src/Operacao";

// Cria um spy para o método somar da classe Operacao
const spySomar = jest.spyOn(Operacao.prototype, "somar");

beforeEach(() => {
  // Limpa o mock do método somar antes de cada teste.
  // Faz diferença ao contar a quantidade de vezes que o método foi chamado
  spySomar.mockClear();
});

it("Spy no método somar", () => {
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(10);

  // Verifica se o método somar foi chamado corretamente
  // Espera-se que o método seja chamado 4 vezes
  expect(spySomar).toHaveBeenCalledTimes(4);
  // Espera-se que o método tenha sido chamado com os seguintes argumentos
  expect(spySomar).toHaveBeenCalledWith(0, 1);
  expect(spySomar).toHaveBeenCalledWith(1, 2);
  expect(spySomar).toHaveBeenCalledWith(3, 3);
  expect(spySomar).toHaveBeenCalledWith(6, 4);
});
```

c) **Stub:** é um objeto falso que fornece uma implementação simples para uma função/método. Stubs são usados para evitar a necessidade de interagir com dependências reais durante os testes, e são usados para focarmos no teste do código em si, sem nos preocuparmos com o comportamento de suas dependências.

Neste teste estamos substituindo a implementação real do método `somar` da classe `Operacao` por duas implementações simuladas que retornam 20 e 42, respectivamente. Isso nos permite testar o resultado da função `somarArray` quando o método `somar` retorna um valor conhecido.

test/stub.test.ts

```
import somarArray from "../src/array";
import Operacao from "../src/Operacao";
```

```
it("Stub e spy no método somar", () => {
  // Stub do método somar que sempre retorna 20
  jest.spyOn(Operacao.prototype, "somar").mockReturnValue(20);
  const r = somarArray([1, 2, 3, 4]);
  // Spy para verificar se o método somar foi chamado 4 vezes
  expect(Operacao.prototype.somar).toHaveBeenCalledTimes(4);
  // Spy para verificar se o método somar foi com os argumentos esperados
  expect(Operacao.prototype.somar).toHaveBeenCalledWith(0, 1);
  expect(Operacao.prototype.somar).toHaveBeenCalledWith(20, 2);
  expect(Operacao.prototype.somar).toHaveBeenCalledWith(20, 3);
  expect(Operacao.prototype.somar).toHaveBeenCalledWith(20, 4);
  // Asserção para verificar se o resultado é igual ao valor simulado
  expect(r).toBe(20);
});

it("Stub no método somar", () => {
  jest.spyOn(Operacao.prototype, "somar").mockReturnValue(42);
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(42);
});
```

Diferenças entre mock e stub:

- Os stubs são parecidos com os mocks, mas geralmente são mais simples;
- Usamos mocks quando precisamos verificar como o código em teste interage com suas dependências. Isso é especialmente importante para interações complexas com comportamento esperado bem definido. Implementação:
  - No Jest, ao usar o método mockImplementation, estamos criando um mock, pois estamos definindo uma implementação específica para a função/método que está sendo mockado, permitindo controlar seu comportamento durante os testes.
- Usamos stubs quando simplesmente precisamos controlar o resultado de uma chamada de função/método e não nos importamos com as interações específicas. Isso pode ser útil para isolar o código em teste de dependências externas durante o desenvolvimento. Implementação:
  - No Jest, ao usar o método mockReturnValue, estamos criando um stub, pois estamos apenas definindo o valor de retorno fixo para a função/método mockado, sem especificar sua implementação detalhada. Isso é útil quando precisamos controlar o resultado de uma chamada de função/método, sem nos preocuparmos com a lógica interna ou as interações detalhadas durante o teste.

Analogia: um stub funciona como uma resposta pré-gravada em uma caixa postal, um stub simplesmente fornece uma saída fixa sem qualquer participação ativa ou verificação da interação. Já um mock funciona como um ator em uma peça teatral, participa ativamente da interação, seguindo um script (comportamento esperado) e nos permitindo verificar o seu desempenho.

ii. Mock de classe

No código a seguir o módulo Operacao exporta uma classe, desta forma, estamos usando um mock para substituir a implementação real do método somar da classe Operacao com uma implementação simulada fornecida por `jest.fn().mockImplementation()`. Isso nos permite controlar o comportamento do método somar durante o teste.

src/Operacao.ts

```
export default class Operacao {
  somar(a: number, b: number) {
    return a + b;
  }

  incrementar(nro: number){
    return nro + 1;
  }
}
```

src/array.ts

```
import Operacao from "./Operacao";

export default function somarArray(vet:number[]){
    const operacao = new Operacao();
    let r = 0;
    for(let i = 0; i < vet.length; i++){
        r = operacao.somar(r,vet[i]);
    }
    return r;
}
```

Criamos um mock da classe Operacao para a função somarArray não utilizar a implementação real do método somar. Desta forma a instrução `r = operacao.somar(r,vet[i])` utilizará a operação implementada em `(a, b) => a + b`. Para isso utilizamos a função `mock` do jest para simular o módulo Operacao.

test/array.test.ts

```
import somarArray from "../src/array";

jest.mock("../src/Operacao", () => {
  const mOperacao = {
    somar: jest.fn().mockImplementation( (a, b) => a + b )
  };
  return jest.fn( () => mOperacao );
});

it("Somar array", () => {
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(10);
});
```

Explicação do código que faz o mock:

- `jest.mock("../src/Operacao", () => { ... })`: indica ao Jest para substituir o módulo Operacao por essa implementação de mock durante os testes. O 1º argumento `"../src/Operacao"` é o caminho para o módulo que queremos mockar. O 2º argumento é uma função que retorna o mock do módulo;
- `const mOperacao = { ... }`: a variável mOperacao recebe as propriedades que queremos mockar da classe Operacao;
- `somar: jest.fn().mockImplementation( (a, b) => a + b )`: usamos para definir o comportamento do método somar do mock da classe Operacao. Estamos usando `jest.fn()` para criar uma função simulada (um "spy") para o método somar, e usamos `mockImplementation` para fornecer uma implementação para essa função simulada;

- `return jest.fn( () => mOperacao )`: significa que, sempre que o módulo Operacao for importado em um teste, será retornado o objeto `mOperacao`, que contém o método somar mockado;
- A implementação do módulo Operacao não incluiu o método `incrementar` por ele não ser necessário no nosso teste.

No código de teste anterior não conseguimos utilizar spies para verificar se o método `somar` foi chamado corretamente. A versão a seguir é capaz de "espionar" o objeto que está na variável `mOperacao`.

test/array.test.ts

```
import somarArray from "../src/array";

const mOperacao = {
  somar: jest.fn().mockImplementation((a, b) => a + b),
};

// Mock do módulo
jest.mock("../src/Operacao", () => ({
  __esModule: true,
  default: jest.fn(() => mOperacao),
}));

it("Somar array", () => {
  const r = somarArray([1, 2, 3, 4]);

  // Verifica se o método somar foi chamado 4 vezes e com números
  expect(mOperacao.somar).toHaveBeenCalledTimes(4);
  expect(mOperacao.somar).toHaveBeenCalledWith(expect.any(Number), expect.any(Number));

  // Verifica o resultado da função somarArray
  expect(r).toBe(10);
});
```

Explicação do código que faz o mock do módulo:

- `jest.mock("../src/Operacao", ...)`: indicamos o caminho do módulo que queremos substituir pelo mock. Aqui substituímos o módulo Operacao;
- `() => ({...})`: função call-back que define o comportamento do mock. Dentro desta função, estamos retornando um objeto que representa o mock do módulo Operacao;
- `__esModule: true`: esta propriedade indica que o módulo é um módulo ES6 (6ª edição do padrão ECMAScript, também conhecida por ECMAScript 2015, que é a especificação padrão para a linguagem JS);
- `default: jest.fn(() => mOperacao)`: define o comportamento do mock para a exportação padrão do módulo. Estamos substituindo a exportação padrão por uma função simulada (mock) que retorna a variável `mOperacao`. Isso significa que sempre que o módulo Operacao for importado, ele retornará o objeto `mOperacao`.
- Isso nos permitiu controlar o comportamento das funções do módulo Operacao durante os testes e chamar o método `toHaveBeenCalledWith` para espionar a chamada do método somar.

iii. Mock de objeto

O código a seguir é semelhante ao anterior, a diferença consiste no fato de o módulo Operacao exportar um objeto ao invés de exportar a classe.

src/Operacao.ts – exporta um objeto da classe

```
class Operacao {
  somar(a: number, b: number) {
    return a + b;
  }

  incrementar(nro: number){
    return nro + 1;
  }
}

export default new Operacao();
```

src/array.ts – importa o objeto da classe Operacao

```
import operacao from "./Operacao";

export default function somarArray(vet:number[]){
    let r = 0;
    for(let i = 0; i < vet.length; i++){
        r = operacao.somar(r,vet[i]);
    }
    return r;
}
```

O código a seguir é usado para fazer o mock e spy do método somar do objeto exportado pelo módulo Operacao.

A função call-back retorna um objeto com a propriedade somar. Usamos a função jest.mock para criar um objeto que simula as operações do tipo Operacao. Observe que a função call-back passada como parâmetro para jest.mock retorna o objeto simulado:

```
{ somar: jest.fn().mockImplementation((a, b) => a + b) }
```

test/array.test.ts

```
import somarArray from "../src/array";
import operacao from "../src/Operacao";

jest.mock("../src/Operacao", () => {
  return {
    somar: jest.fn().mockImplementation((a, b) => a + b),
  };
});

it("Somar array", () => {
  // Cria um spy para o método somar da classe Operacao
  const spySomar = jest.spyOn(operacao, "somar");
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(10);
  // Verifica se o método somar foi chamado 4 vezes e com números
  expect(spySomar).toHaveBeenCalledTimes(4);
  expect(spySomar).toHaveBeenCalledWith(expect.any(Number), expect.any(Number));
});
```

iv. Mock de função

Neste exemplo o módulo Operacao exporta as funções. Desta forma, o código de teste precisou fazer o mock da função somar.

src/Operacao.ts – exporta funções

```ts
export function somar(a: number, b: number){
    return a + b;
}

export function incrementar(nro: number){
    return nro + 1;
}
```

src/array.ts – importa a função somar do módulo Operacao

```ts
import { somar } from "./Operacao";

export default function somarArray(vet: number[]){
  let r = 0;
  for (let i = 0; i < vet.length; i++) {
    r = somar(r, vet[i]);
  }
  return r;
}
```

test/array.test.ts

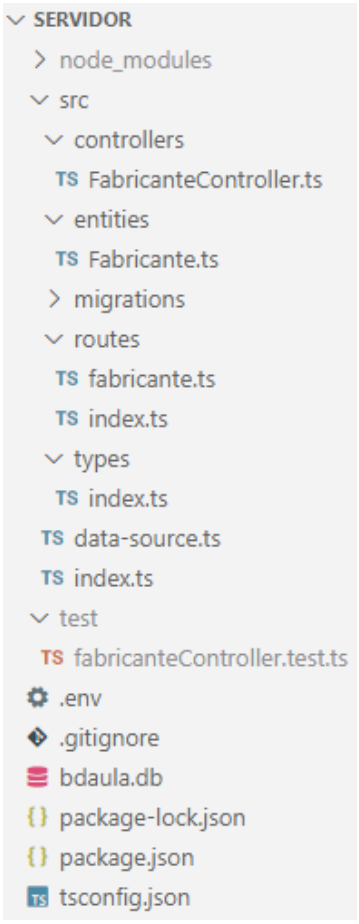
```ts
import somarArray from "../src/array";
import * as Operacao from "../src/Operacao";

jest.mock("../src/Operacao", () => {
  return {
    somar: jest.fn().mockImplementation((a, b) => a + b),
  };
});

it("Somar array", () => {
  // Cria um spy para o método somar da classe Operacao
  const spySomar = jest.spyOn(Operacao, "somar");
  const r = somarArray([1, 2, 3, 4]);
  expect(r).toBe(10);
  // Verifica se o método somar foi chamado 4 vezes e com números
  expect(spySomar).toHaveBeenCalledTimes(4);
  expect(spySomar).toHaveBeenCalledWith(expect.any(Number), expect.any(Number));
});
```

v. Exemplo de uso de mock

A aplicação disponível em https://github.com/arleysouza/mock possui a estrutura mostrada ao lado.



O objetivo é fazermos os testes dos métodos da classe FabricanteController fazendo o mock do objeto DataSource (exportado pelo módulo src/data-source.ts), ou seja, não usaremos o BD da aplicação que está no arquivo bdaula.db.

No código de teste a seguir usamos o método `jest.mock` para criar um mock do objeto `DataSource` exportado pelo módulo `data-source`.

A função call-back passada como parâmetro para o método `jest.mock` retorna um objeto com a propriedade `manager` contendo um objeto com as propriedades `save`, `find`, `findOneBy` e `delete`. Cada uma dessas propriedades possui um mock criado usando jest.fn() e configuradas com diferentes comportamentos. Tome como exemplo as implementações do método `save`. Ele possui 6 implementações codificadas usando o método `mockImplementationOnce`. Cada implementação dessa será chamada uma vez nos testes e na sequência que elas aparecem, ou seja, devemos mudar a ordem dos mocks se mudarmos a ordem dos testes.

```
save: jest.fn()
  .mockImplementationOnce(() => {
    return { id: 1, nome: "um" };
  })

  .mockImplementationOnce(() => {
    throw new Error("SQLITE_CONSTRAINT: UNIQUE constraint failed: fabricantes.nome");
  })
  .mockImplementationOnce(() => {
    throw new Error("Qualquer");
  })
  .mockImplementationOnce(() => {
    return { id: 1, nome: "dois" };
  })
  .mockImplementationOnce(() => {
    throw new Error("SQLITE_CONSTRAINT: UNIQUE constraint failed: fabricantes.nome");
  })
  .mockImplementationOnce(() => {
    throw new Error("Qualquer");
  }),
```

Arquivo: test/fabricanteController.test.ts

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

jest.mock("../src/data-source", () => {
  return {
```

```
    manager: {
      save: jest
        .fn()
        .mockImplementationOnce(() => {
          return { id: 1, nome: "um" };
        })
        .mockImplementationOnce(() => {
          throw new Error(
            "SQLITE_CONSTRAINT: UNIQUE constraint failed: fabricantes.nome"
          );
        })
        .mockImplementationOnce(() => {
          throw new Error("Qualquer");
        })
        .mockImplementationOnce(() => {
          return { id: 1, nome: "dois" };
        })
        .mockImplementationOnce(() => {
          throw new Error(
            "SQLITE_CONSTRAINT: UNIQUE constraint failed: fabricantes.nome"
          );
        })
        .mockImplementationOnce(() => {
          throw new Error("Qualquer");
        }),
      find: jest
        .fn()
        .mockImplementationOnce(() => {
          return [
            { id: 1, nome: "um" },
            { id: 2, nome: "dois" },
          ];
        })
        .mockImplementationOnce(() => {
          throw new Error(
            "SQLITE_CONSTRAINT: UNIQUE constraint failed: fabricantes.nome"
          );
        }),
      findOneBy: jest
        .fn()
        .mockImplementationOnce(() => null)
        .mockImplementation(() => {
          return { id: 1, nome: "um" };
        }),
      delete: jest
        .fn()
        .mockImplementationOnce(() => {
          return { affected: 1 };
        })
```

```
        .mockImplementationOnce(() => {
          return { affected: 0 };
        })
        .mockImplementationOnce(() => {
          throw new Error("Qualquer");
        }),
    },
  };
});

let controller: FabricanteController;
beforeAll(() => {
  controller = new FabricanteController();
});

describe("FabricanteController - save", () => {
  it("Salvar fabricante sem o nome", async () => {
    const req = { body: {} } as Request;
    const res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
    } as unknown as Response;

    await controller.save(req, res);

    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ erro: "Forneça o nome" });
  });

  it("Salvar fabricante no BD", async () => {
    const req = { body: { nome: "um" } } as Request;
    const res = {
      json: jest.fn(),
    } as unknown as Response;

    await controller.save(req, res);

    expect(res.json).toHaveBeenCalledWith({ id: 1, nome: "um" });
  });

  it("Salvar fabricante repetido no BD", async () => {
    const req = { body: { nome: "um" } } as Request;
    const res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
    } as unknown as Response;

    await controller.save(req, res);
    expect(res.status).toHaveBeenCalledWith(409);
```

```
    expect(res.json).toHaveBeenCalledWith({ erro: "Nome repetido" });
  });

  it("Erro genérico ao salvar o fabricante no BD", async () => {
    const req = { body: { nome: "um" } } as Request;
    const res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
    } as unknown as Response;

    await controller.save(req, res);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ erro: "Qualquer" });
  });
});

describe("FabricanteController - update", () => {
  it("Atualizar sem fornecer o ID", async() => {
    const req = { body: {} } as Request;
    const res = { json: jest.fn(), status: jest.fn().mockReturnThis() } as unknown as Response;

    await controller.update(req, res);
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ erro: "Forneça o identificador" });
  });

  it("Atualizar sem fornecer o nome", async() => {
    const req = { body: {id:1} } as Request;
    const res = { json: jest.fn(), status: jest.fn().mockReturnThis() } as unknown as Response;

    await controller.update(req, res);
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ erro: "Forneça o nome" });
  });

  it("Atualizar com ID não localizado", async() => {
    const req = { body: {id:1, nome:"marca"} } as Request;
    const res = { json: jest.fn(), status: jest.fn().mockReturnThis() } as unknown as Response;

    await controller.update(req, res);
    expect(res.status).toHaveBeenCalledWith(409);
    expect(res.json).toHaveBeenCalledWith({ erro: "Registro não localizado" });
  });

  it("Atualizar fabricante", async() => {
    const req = { body: {id:1, nome:"dois"} } as Request;
    const res = { json: jest.fn() } as unknown as Response;

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

```
    expect(res.json).toHaveBeenCalledWith({id:1, nome:"dois"});
  });

  it("Atualizar fabricante repetido no BD", async() => {
    const req = { body: {id:1, nome:"um"} } as Request;
    const res = { json: jest.fn(), status: jest.fn().mockReturnThis() } as unknown as Response;

    await controller.update(req, res);
    expect(res.status).toHaveBeenCalledWith(409);
    expect(res.json).toHaveBeenCalledWith({ erro: "Nome repetido" });
  });

  it("Erro genérico ao atualizar o fabricante", async() => {
    const req = { body: {id:1, nome:"um"} } as Request;
    const res = { json: jest.fn(), status: jest.fn().mockReturnThis() } as unknown as Response;

    await controller.update(req, res);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ erro: "Qualquer" });
  });
});

describe("FabricanteController - list", () => {
  it("Lista de fabricantes", async () => {
    const req = {} as Request;
    const res = { json: jest.fn() } as unknown as Response;

    await controller.list(req, res);

    expect(res.json).toHaveBeenCalledWith([
      { id: 1, nome: "um" },
      { id: 2, nome: "dois" },
    ]);
  });

  it("Erro ao listar os fabricantes", async () => {
    const req = {} as Request;
    const res = {
      json: jest.fn(),
      status: jest.fn().mockReturnThis(),
    } as unknown as Response;

    await controller.list(req, res);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ erro: "Problemas com o BD" });
  });
});

describe("FabricanteController - remove", () => {
```

```
  it("Remove fabricante", async () => {
    const req = { body: { id: 1 } } as Request;
    const res = { json: jest.fn() } as unknown as Response;

    // Chamada do método list
    await controller.remove(req, res);

    expect(res.json).toHaveBeenCalledWith({ message: "Registro excluído" });
  });

  it("Fabricante não localizado", async () => {
    const req = { body: { id: 1 } } as Request;
    const res = {
      json: jest.fn(),
      status: jest.fn().mockReturnThis(),
    } as unknown as Response;

    // Chamada do método list
    await controller.remove(req, res);
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ message: "Registro não localizado" });
  });

  it("Erro ao remover fabricante", async () => {
    const req = { body: { id: 1 } } as Request;
    const res = {
      json: jest.fn(),
      status: jest.fn().mockReturnThis(),
    } as unknown as Response;

    // Chamada do método list
    await controller.remove(req, res);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ erro: "Qualquer" });
  });
});
```