Revisão da Prova I – Desenvolvimento Web III – Prof. Arley

Veja o vídeo se tiver dúvidas nos exercícios: https://youtu.be/I6LanEBOYNY

1 – Qual é a diferença básica entre teste de integração e teste unitário?

R: A diferença básica está no escopo e no foco dos testes:

- Teste Unitário: foca na verificação do funcionamento de unidades individuais de código, como funções, métodos ou classes. É realizado de forma isolada, sem dependências externas;
- Teste de integração: foca na verificação da interação e integração entre diferentes componentes, módulos ou sistemas. Envolve a execução de vários componentes ou módulos em conjunto.

2 – Por que não usamos mocks ou stubs nos testes de integração?

R: Evitamos o uso de mocks ou stubs porque o objetivo principal é verificar a integração real entre os componentes do sistema, incluindo suas dependências externas.

3 – Por que usamos mocks ou stubs nos testes unitários?

R: Porque o objetivo é testar o código isoladamente, independente de componentes externos.

4 – O que é a instrução `expect(somar(2, 3)).toBe(5)` no teste a seguir?

```
function somar(a: number, b: number):number {
  return a + b;
}
test("soma com valores positivos", () => {
  expect(somar(2, 3)).toBe(5);
});
```

R: A instrução é uma asserção. Nesse contexto, a função expect() é usada para afirmar ou verificar um comportamento esperado do código sendo testado. O método .toBe(5) é um matcher, que verifica se o resultado retornado pela função somar(2, 3) é exatamente igual a 5. Portanto, a combinação dessas duas partes constitui uma asserção, onde estamos afirmando que o resultado da função somar(2, 3) deve ser igual a 5.

5 – A cobertura de testes unitários é uma métrica que indica a porcentagem de código-fonte que foi exercida pelos testes durante a execução de um conjunto de testes. Nesse contexto, a cobertura de testes da função somar refere-se à extensão em que essa função é testada cobrindo as possibilidades de respostas. Codificar a cobertura de testes da função somar.

```
async function somar(a: any, b: any): Promise<number|never> {
  if (typeof a == "number" && typeof b == "number") {
    return a + b;
  }
  throw new Error("Parâmetros inválidos");
}
```

6 – Codificar os testes da função operar.

Revisão da Prova I – Desenvolvimento Web III – Prof. Arley

```
function multiplicar(a:number, b:number): number {
  return a * b;
}

function operar(nros: number[], f: Function) {
  let r = 1;
  for( let i = 0; i < nros.length; i++ ){
    r = f(r,nros[i]);
  }
  return r;
}
```

7 – 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. Adicione um teste no Exercício 6 para verificar se a função multiplicar está sendo chamada a quantidade esperada de vezes.

Dicas:

- Use jest.fn() para criar um mock da função multiplicar;
- Use o matcher toHaveBeenCalledTimes para verificar a quantidade de vezes que o mock foi chamado.

8 – No código do Exercício 6 a função multiplicar é chamada várias vezes com diferentes argumentos dentro do loop da função operar. Adicione um teste no Exercício 6 para verificar se a função multiplicar está sendo chamada com os argumentos corretos.

Dicas:

- Use o matcher toHaveBeenNthCalledWith para verificar se a função multiplicar está sendo chamada na enésima vez com os argumentos esperados;
- O matcher toHaveBeenCalledWith não funciona da maneira esperada neste caso pelo fato de a função multiplicar ser chamada várias vezes.

9 – Codificar a cobertura de testes da função saudacao. Como a função saudacao não possui retorno, então será necessário verificar se a função exibir foi chamada para testar o funcionamento da função saudacao.

```
function exibir(msg: String): void {
  console.log(msg);
}

async function saudacao(nome: String, f: Function) {
  if (nome && nome.length > 0) {
    f(`Boa noite ${nome}`);
  }
}
```

Revisão da Prova I – Desenvolvimento Web III – Prof. Arley

10 – Podemos utilizar a função jest.mock() para substituir a implementação de um módulo por uma versão simulada. Codificar o teste da função operar utilizando um mock do método multiplicar da classe Operacao.

Dicas:

- Use jest.mock("caminho do módulo") para criar um mock para o módulo src/Operacao;
- Use jest.fn().mockImplementation para criar uma versão simulada do método multiplicar.

Arquivo: src/Operacao.ts

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

  multiplicar(a: number, b: number) {
    console.log("aqui", a, b);
    return a + b;
  }
}
```

Arquivo: test/operacao.test.ts

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

function operar(nros: number[]) {
  const operacao = new Operacao();
  let r = 1;
  for (let i = 0; i < nros.length; i++) {
    r = operacao.multiplicar(r, nros[i]);
  }
  return r;
}
```

11 – Adicionar os seguintes testes no código do Exercício 10:

- Adicione um teste para verificar se o método multiplicar está sendo chamado a quantidade certa de vezes;
- Adicione um teste para verificar se o método multiplicar está sendo chamado com os argumentos corretos.

Dica:

- Para utilizar spies no mock do módulo o mock do módulo precisa ter as propriedades __esModule e default.

12 – 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. Repita os mesmos testes do Exercício 11 no código a seguir, observe que o módulo Operacao exporta um objeto.

Dicas:

- Use jest.mock("caminho do módulo") para criar um mock para o módulo src/Operacao;
- Use jest.spyOn para criar um spy no método multiplicar do objeto Operacao.

Arquivo: src/Operacao.ts

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

  multiplicar(a: number, b: number) {
    console.log("aqui", a, b);
    return a + b;
  }
}
```

Arquivo: test/operacao.test.ts

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

function operar(nros: number[]) {
  let r = 1;
  for (let i = 0; i < nros.length; i++) {
    r = operacao.multiplicar(r, nros[i]);
  }
  return r;
}
```

`export default new Operacao();`	

13 – O Supertest é uma biblioteca usada para testar requisições HTTP em conjunto com frameworks de testes como o Jest. Codifique os testes das rotas do arquivo a seguir.

Arquivo: src/index.ts

```
import express from "express";

const app = express();
app.use(express.json());
app.listen(3010, () => console.log(`Rodando...`));

export default app;

app.get("/somar", (req,res) => {
    const {x,y} = req.body;
    const r = x + y;
    res.json({r});
});

app.post("/subtrair/:x/:y", (req,res) => {
    const {x,y} = req.params;
    const r = parseInt(x) - parseInt(y);
    res.json({r});
});
```