Testes Unitários com Jest

Métodos de Desenvolvimento de Software - Unidade 02

Equipe Meio a Meio

RESUMO:

Testes Unitários:

- O teste unitário consiste em verificar o comportamento das menores unidades em sua aplicação.
- Funcionalmente, pode ser um conjunto de classes intimamente relacionadas. Como um "Cervo" e suas classes de suporte "Cabeça", "Rabo" e "Movimento".
- Os testes unitários precisam funcionar isoladamente porque precisam funcionar rapidamente.
- Todo o conjunto de testes unitários de uma aplicação precisa funcionar em minutos, de preferência em segundos. Você verá o porquê mais tarde.]
- Se seu código de teste (ou as bibliotecas que utiliza) fizer E/S ou acessar qualquer coisa fora de seu processo, não é um teste unitário, e sim um teste de integração.
- Benefícios : Economizar tempo, se você for testando parte por parte do código, fica mais difícil ter bugs, consequentemente menos coisa pra arrumar posteriormente no código.
- Para melhores práticas durante a realização de testes unitários, faça com que o teste só dê errado se você quiser que ele dê errado e só dê certo se você quiser que ele dê certo caso contrário não irá dar certo.

Vamos começar por escrever um teste para uma função hipotética que soma dois números. Primeiro, crie um arquivo `sum.js`:

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Em seguida, crie um arquivo chamado `sum.test.js`. Este irá conter o nosso teste real:

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Adicione a seguinte seção ao seu `package.json`:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Por fim, execute `yarn test` ou `npm run test` e o Jest irá imprimir a segunte mensagem:

```
PASS  ./sum.test.js
✓ adds 1 + 2 to equal 3 (5ms)
```

-

Matchers:

- São comparativos para se testar valores diferentes
- Você sempre deve usar o "matcher" que corresponde mais precisamente para o que você deseja que seu código faça.

    - Comuns - A maneira mais simples para testar um valor

    Ex: toBe = testa a igualdade exata entre dois objetos;
    toEqual = recursivamente verifica cada campo de um objeto ou array;
    not.toBe = oposto ao toBe - o metodo ".not" testa o oposto de qualquer matcher

    - Verdade - Quando se precisa distinguir as definições, sem querer trata-las de forma diferente

    Ex: toBeNull = corresponde a apenas null;
    toBeUndefined = corresponde a apenas undefined;
    toBeDefined = é o oposto de toBeUndefined;
    toBeTruthy = combina com qualquer coisa que uma instrução if trata como verdadeiro;
    toBeFalsy = combina com qualquer coisa que uma instrução if trata como falso;

    - Números - A maioria das formas de comparar números têm "matcher" equivalentes.

    obs: toBe e toEqual são equivalentes para números, mas para igualdade de ponto flutuante, use toBeCloseTo em vez de toEqual, porque você não quer um teste dependa de um pequeno erro de arredondamento.

    Ex: toBeGreaterThan = maior que;
    toBeGreaterThanOrEqual = maior ou igual que;
    toBeLessThan = menor que;
    toBeLessThanOrEqual = menor ou igual que;

    - Strings - toMatch = verifica strings contra expressões regulares;

    - Arrays e Iteráveis - toContain = verifica se um array ou iterável contém um item específico;

    - Exceções - toThrow = testa se uma determinada função lança um erro quando é chamada;

Códigos Assíncronos:

- Operações assíncronas: Operações executadas "por baixo dos panos", ou seja, permitem que o código principal possa continuar sendo executado sem precisar esperar que elas terminem;

- Quando você tiver o código que é executado de forma assíncrona, Jest precisa saber quando o código que está testando for concluído, antes que possa passar para outro teste.
- Todos os tópicos descrevem o mesmo trecho de código só que de maneiras diferentes de se escrevê-los: async/await e callbacks são formas alternativas as promises, ja .resolves e .rejects são formas diferentes do then de escrever promises.
- Ordem: Promises > .resolves/.rejects > Async/await > Callbacks

- Promise:

O que é uma promise? Como ja dito, uma promessa que garante que o seu código assíncrono vai -retornar alguma coisa no final - seja sucesso, que está pendente ou uma falha. Quando relacionado ao Jest, retorne uma promise do seu teste, e o Jest vai esperar essa promise ser resolvida. Se a promessa for rejeitada, o teste irá falhar.

- .resolves/.rejects:

São matchers de Promises alternativos ao then com o mesmo intuito básico, sendo o .resolves para quando o Jest aguardar a promessa se resolver (falhando imediatamente se for rejeitada); e o .rejects, funcionando de forma analogica ao resolves, para quando o Jest aguardar que a promessa seja rejeitada (falando imediatamente se for cumprida).

- Async/Await:

São alternativas práticas e simples da lógica das Promises nos testes. Sendo que, com elas, basta declarar "async" na frente da função passada para teste e await quando a execução da função for esperada.

PODEM SER COMBINADOS COM OS RESOLVES E REJECTS!!!

- Callbacks:

O que é uma callback? Também conhecida como função de retorno, é uma função ou URL que é executada quando algum evento acontece ou depois que algum código chega ao estado desejado. Já relacionado ao Jest, é usada quando não se opta por uma Promise numa função assíncrona (usar exemplo de fetchData como na descrição - no lugar de retornar uma Promise, como then, usar callback()) - Continuar explicaçao do fetchData -> problema das callbacks: se apenas chama-la dentro do teste, ele vai finalizar assim que fetchData for completa. Pra corrigir isso, use a callback"done", porque ai o Jest vai ficar esperando ela aparecer antes de terminar o teste - se o done nao for chamado, o teste ira falhar com timeout

Mocks:

- Em português: função de simulação
- São objetos que simulam o comportamento de objetos reais de forma controlada.

- Permitem que você teste os links entre códigos, apagando a implementação real de uma função, capturando chamadas para a função
- Existem duas maneiras de simular funções: Seja criando uma função simulada para usar no código de teste, ou escrevendo uma simulação manual para sobrescrever uma dependência de módulo.
- Todas as funções de simulação (.mock) possuem esta especial propriedade .mock, que é onde os dados sobre como a função a qual foi chamada são mantidos.
- Funções mock também podem ser usadas para injetar valores de teste no código durante um teste

Passo a passo: Configuração de ambiente

Utilizando o gerenciador de pacotes da sua preferência (NPM ou YARN), execute os comandos abaixo:

No exemplo foi utilizado NPM

Passo 1: Instale a library do Jest: npm install --save-dev jest

Passo 2: Adicione no package.json do projeto o script:

```
"scripts": {
        "test": "jest"
}
```

Passo 3: Escrevendo o teste

- Crie um arquivo ".js" com o nome de sua preferência para implementar as funcionalidades.
- Crie um novo arquivo ".js" para escrever os seus testes unitários.
    - Obs.: Para testes assíncronos, instale a library ... (perguntar o nome ao Samuel) - É feita uma requisição com uma API externa

Passo 4: Execute o teste: npm run test

Construindo um teste unitário

Construa o seu método de teste em etapas:

1. **ARRANGE:** É a etapa em que você prepara o ambiente para o caso de teste. Exemplo: mockar serviços externos e criar objetos.
2. **ACT:** Esse é o momento onde será executada a ação principal do caso que está sendo testado.
3. **ASSERT:** Onde é feita a validação do retorno do act com o que esperamos.

Dicas para escrever bons testes

1. Dê nomes aos seus métodos de teste que o ajudem a entender os requisitos do código que você está testando.
2. Certifique-se de que um teste só tenha sucesso porque o código que ele testa está correto.
3. Evite tornar os testes dependentes de seu ambiente de testes.
4. Um teste só deve validar somente um caso de teste.
5. Sempre que ocorrer um comportamento inesperado, faça testes e refatore!
6. Teste casos de falha para saber como a aplicação se comporta.
7. Evite repetições!
8. É importante que todos os métodos do projeto sigam o mesmo padrão, isso gera consistência.
9. Code Review, assim como as demais implementações, os testes também devem passar pelo code review do time.

PERGUNTAS:

1. O que são Testes Unitários?

   Resposta: São mecanismos de validação das menores unidades de uma aplicação (aceita respostas semelhantes).

2. Cite um dos benefícios de fazer Testes Unitários.

   Resposta: Feedback instantâneo da implementação; Diz o quão testável é seu código; Aumenta a produtividade da equipe.

3. Qual é o símbolo do Jest?

   Resposta: Um sapato de bobo da corte.

4. Qual é o Matcher utilizado para encontrar expressões regulares em strings?

   Resposta: toMatch.

5. O que são Mocks?

   Resposta: Mecanismos para simular a chamada de endpoints e módulos (aceita respostas semelhantes).

6. O que faz o beforeAll?

   Resposta: Configura o ambiente de testes antes de executá-los.

7. O que o Matcher toEqual faz?

   Resposta: Verifica se uma expressão é exatamente igual a outra.

8. O que o Matcher toContain faz?

   Resposta: Verifica se o elemento está presente no array.

9. Quais são as assinaturas de uma função assíncrona?

   Resposta: Async/Await.

10. Cite dois matchers de comparação de números.

Resposta: toBeGreaterThan; toBeGreaterThanOrEqual; toBeLessThan; toBeLessThanOrEqual..