Escolha 2 dos 4 Exercícios Práticos abaixo para Implementação de Testes Unitários a partir dos modelos em: https://red-partridge-657937.hostingersite.com/testes/

Exercício 1: Testando um Componente de Lista de Tarefas

Objetivo: Criar e testar um componente TodoList em React que permita adicionar, marcar como concluída e excluir tarefas.

Requisitos:

- Crie um novo projeto React usando Vite:

```
npm create vite@latest todo-testing-app -- --template react-ts
cd todo-testing-app
npm install
```

- Instale as dependências de teste:

```
npm install -D vitest @testing-library/react @testing-library/jest-dom @testing-library/user-event jsdom
```

- Configure o Vitest no arquivo `vite.config.ts`:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './src/test/setup.ts',
  },
});
```

- Crie um arquivo `src/test/setup.ts`:

```
import '@testing-library/jest-dom';
```

Tarefa:

- **Implemente o componente `TodoList` em** `src/components/TodoList.tsx` com:
  - Estado para armazenar as tarefas
  - Input para adicionar novas tarefas
  - Exibição de uma lista de tarefas
  - Opções para marcar uma tarefa como concluída ou excluí-la

- **Escreva os seguintes testes em** `src/components/TodoList.test.tsx`:
    - Teste se uma tarefa pode ser adicionada quando o usuário digita no input e pressiona Enter
    - Teste se uma tarefa pode ser marcada como concluída
    - Teste se uma tarefa pode ser removida
    - Teste se o componente exibe uma mensagem quando não há tarefas

Dicas:

- Use o hook `useState` para gerenciar o estado das tarefas
- Cada tarefa deve ter: id, título e status de conclusão
- Use dados de teste identificáveis com atributos `data-testid`
- Simule eventos de usuário com `userEvent` da biblioteca Testing Library

Exercício 2: Testando um Hook de Fetchíng de Dados

Objetivo: Criar e testar um hook personalizado `useFetchData` que faça requisições à API JSONPlaceholder e gerencie estados de loading, erro e dados.

Requisitos:

- Use o mesmo projeto ou crie um novo:

```
npm create vite@latest fetch-testing-app -- --template react-ts
cd fetch-testing-app
npm install
npm install -D vitest @testing-library/react @testing-library/jest-dom @testing-library/user-event jsdom
```

-

Tarefa:

- Implemente o hook `useFetchData` em `src/hooks/useFetchData.ts`:

```
import { useState, useEffect } from 'react';

export function useFetchData<T>(url: string) {
  // Implemente os estados para dados, loading e erro
  // Implemente o fetchData com useEffect
  // Retorne os estados e uma função para recarregar os dados
}
```

- Escreva os seguintes testes em `src/hooks/useFetchData.test.ts`:
    - Teste se o estado inicial é correto (loading: true, data: null, error: null)
    - Teste se os dados são carregados com sucesso
    - Teste se o erro é tratado corretamente
    - Teste se a função de recarregamento funciona

Dicas:

- Use a JSONPlaceholder API: https://jsonplaceholder.typicode.com/todos
- Mock a função `fetch` global para simular diferentes respostas
- Use a função `renderHook` da biblioteca Testing Library para testar o hook
- Teste diferentes cenários: sucesso, erro de rede, erro de resposta

Exercício 3: Testando um Formulário de Cadastro

Objetivo: Implementar e testar um formulário de cadastro de usuário com validações e chamada à API.

Requisitos:

- Use o mesmo projeto ou crie um novo:

```
npm create vite@latest form-testing-app -- --template react-ts
cd form-testing-app
npm install
npm install -D vitest @testing-library/react @testing-library/jest-dom @testing-library/user-event jsdom
```

Tarefa:

- Crie um componente `RegistrationForm` em `src/components/RegistrationForm.tsx` com:
    - Campos para nome, e-mail, senha e confirmação de senha
    - Validações para cada campo
    - Estado para manipular os valores do formulário
    - Manipulador de envio que faz uma chamada API simulada
- Escreva os seguintes testes em `src/components/RegistrationForm.test.tsx`:
    - Teste se o formulário é renderizado corretamente
    - Teste as validações de cada campo:

        - Nome: obrigatório, mínimo 3 caracteres
        - E-mail: formato válido
        - Senha: mínimo 6 caracteres, uma letra maiúscula, um número
        - Confirmação: deve ser igual à senha
    - Teste se o botão de envio fica desativado quando o formulário é inválido
    - Teste se o envio chama a API com os dados corretos
    - Teste o comportamento do formulário durante o envio (loading) e após o envio (sucesso/erro)

Dicas:

- Use o padrão de Controlled Components para os inputs do formulário
- Crie uma função de validação separada para cada campo
- Use o atributo `disabled` no botão de submit quando o formulário é inválido
- Mock a função de envio para testar diferentes cenários de resposta

Exercício 4: Testando uma Aplicação Angular de Gerenciamento de Produtos

Objetivo: Criar uma aplicação Angular com componentes, serviços e testes unitários para um gerenciador de produtos.

Requisitos:

- Crie um novo projeto Angular:

```
ng new product-management
cd product-management
ng add @angular-eslint/schematics
npm install -D jest @types/jest jest-preset-angular
```

- Configure o Jest para Angular conforme a documentação.

Tarefa:

- Crie um modelo `Product` em `src/app/models/product.ts`:

```
export interface Product {
  id: number;
  name: string;
  price: number;
  category: string;
  inStock: boolean;
}
```

- Crie um serviço `ProductService` em `src/app/services/product.service.ts`:
    - Métodos para obter a lista de produtos
    - Método para adicionar um produto
    - Método para atualizar um produto
    - Método para excluir um produto
- Crie um componente `ProductList` que utilize o serviço para exibir produtos.
- Crie um componente `ProductForm` para adicionar/editar produtos.
- Escreva os seguintes testes:
    - Teste do serviço `ProductService` em `src/app/services/product.service.spec.ts`:
        - Teste se o método de obtenção retorna a lista correta
        - Teste se o método de adição funciona corretamente
        - Teste se o método de atualização funciona corretamente
        - Teste se o método de exclusão funciona corretamente
    - Teste do componente `ProductList` em `src/app/components/product-list/product-list.component.spec.ts`:
        - Teste se a lista é exibida corretamente
        - Teste se a mensagem "Nenhum produto" é exibida quando a lista está vazia
        - Teste se o botão de exclusão chama o método correto
    - Teste do componente `ProductForm` em `src/app/components/product-form/product-form.component.spec.ts`:
        - Teste se o formulário é renderizado corretamente
        - Teste as validações do formulário
        - Teste se o envio do formulário chama o método correto do serviço

Dicas:

- Use o HttpClientTestingModule para mock de chamadas HTTP
- Use o TestBed para configurar e injetar serviços
- Use o ComponentFixture para testar componentes
- Organize os testes com `describe` e `it` para facilitar a leitura

Instruções Gerais para os Exercícios:

- **Configuração do Ambiente:**
    - Siga as instruções de configuração específicas para cada exercício
    - Verifique se as ferramentas de teste estão funcionando com um teste simples
- **Desenvolvimento Orientado a Testes (TDD):**
    - Tente seguir o ciclo TDD: escreva o teste, veja-o falhar, implemente o código, veja o teste passar
    - Para cada funcionalidade, comece escrevendo o teste antes da implementação
- **Execução dos Testes:**
    - Para React/Vitest: `npm run test`
    - Para Angular/Jest: `npm test`
- **Validação:**
    - Todos os testes devem passar
    - A cobertura de código deve ser alta (pelo menos 80%)
    - Os testes devem ser legíveis e manutenível
- **Recursos Adicionais:**
    - Consulte a documentação das ferramentas quando necessário:
        - Vitest
        - Testing Library
        - Jest
        - Angular Testing