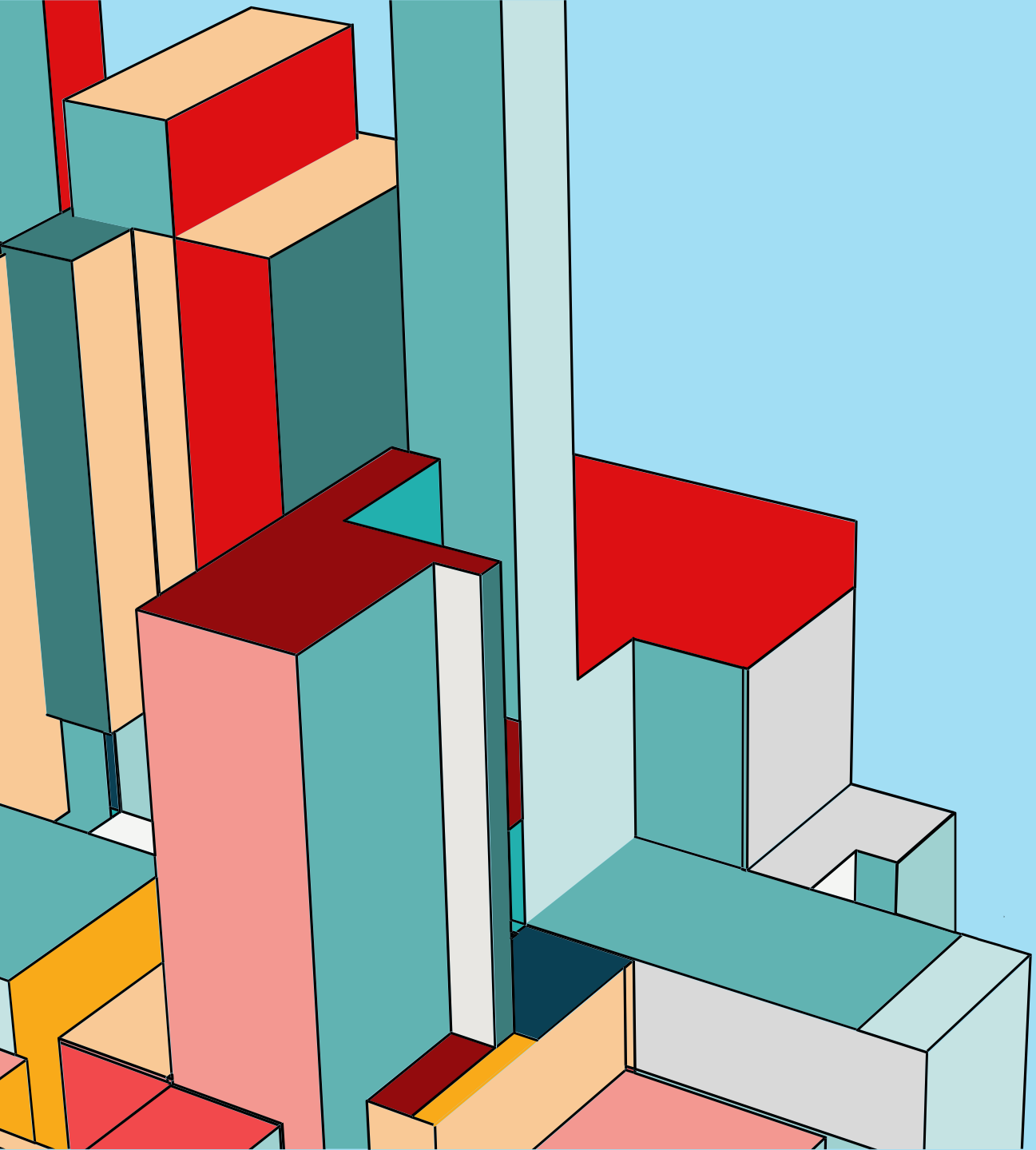

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

TESTE DE SOFTWARE

TESTES UNITÁRIOS - FRONTEND

Prof. Flávio Belizário da Silva Mota
Universidade do Vale do Sapucaí – UNIVAS
Sistemas de Informação

PASSO A PASSO: TESTANDO O COMPONENTE USERS – ANTES UM AJUSTE

```
// No arquivo Users.tsx -> dentro do formulário - linhas 116 a 128
<div className="form-group">
  <label htmlFor="name">Nome:</label>
  <input
    id="name"
    type="text"
    value={formData.name}
    onChange={(e) => setFormData({ ...formData, name: e.target.value })}
    required
  />
</div>
<div className="form-group">
  <label htmlFor="email">Email:</label>
  <input
    id="email"
    type="email"
    value={formData.email}
    onChange={(e) => setFormData({ ...formData, email: e.target.value })}
    required
  />
</div>
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

- Objetivo: garantir que, ao clicar em “Adicionar Usuário”, o formulário com “Nome” e “Email” apareça.

1) Instalar dependências de teste (frontend):

```
docker-compose exec frontend npm i -D vitest @testing-library/react @testing-library/jest-dom @testing-library/user-event jsdom
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

Esses pacotes permitem que você teste componentes React simulando o comportamento real do navegador e do usuário.

@testing-library/react: é a biblioteca usada para *renderizar* componentes React em um ambiente de teste.
Ela prioriza testes baseados em comportamento do usuário (por exemplo, encontrar elementos pelo texto ou rótulo).

@testing-library/jest-dom: adiciona *matchers* ao Vitest, como toBeInTheDocument(), toHaveTextContent() e outros, para deixar asserções mais expressivas.

@testing-library/user-event: simula ações reais do usuário (clicar, digitar, pressionar teclas).
É mais fiel do que usar fireEvent.

jsdom: cria um *ambiente DOM virtual*, simulando o navegador dentro do Node.js.
Sem ele, o React não conseguiria renderizar nem manipular elementos HTML em teste.

PASSO A PASSO: TESTANDO O COMPONENTE USERS

2) Adicionar scripts de teste no frontend/package.json

```
{
  "scripts": {
    "test": "vitest run",
    "test:watch": "vitest",
    "test:coverage": "vitest run --coverage"
  }
}
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

3) Configurar Vitest (ambiente jsdom)

Crie frontend/vitest.config.ts:

```
import { defineConfig } from 'vitest/config'


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

PASSO A PASSO: TESTANDO O COMPONENTE USERS

O código anterior é onde definimos o ambiente e o comportamento padrão dos testes.

environment: 'jsdom': garante que cada teste rode em um ambiente de navegador simulado (HTML, eventos, etc.).

setupFiles: arquivos executados antes dos testes (ótimo para importar utilitários globais ou configurar mocks).

globals: true: permite usar funções globais como describe, it, expect sem precisar importá-las em cada arquivo.

Precisamos ainda criar o arquivo `frontend/tests/setup.ts`

E nele importar o Jest DOM:

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

PASSO A PASSO: TESTANDO O COMPONENTE USERS

O componente sendo testado (Users.tsx)

Aqui, o botão “Adicionar Usuário” altera o estado showForm para true, o que faz o formulário ser renderizado.
O teste precisa verificar **esse comportamento interativo,** ou seja, que clicar no botão faz o formulário aparecer.

```tsx
<button className="btn" onClick={() => setShowForm(true)}>
  Adicionar Usuário
</button>

{showForm && (
  <form onSubmit={handleSubmit}>
    <div className="form-group">
      <label>Nome:</label>
      <input ... />
    </div>
    <div className="form-group">
      <label>Email:</label>
      <input ... />
    </div>
  </form>
)}
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

5) Implementar o teste (UI + interação)

Crie `frontend/tests/unit/Users.ui.test.tsx`

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { vi } from 'vitest'
import Users from '../../src/components/Users'
import { api } from '../../src/services/api'

describe('Users - abrir formulário de criação', () => {
  beforeEach(() => {
    vi.restoreAllMocks()
  })
  it('mostra o formulário ao clicar em "Adicionar Usuário"', async () => {
    vi.spyOn(api, 'get').mockResolvedValue({ data: { data: [] } } as any)
    render(<Users />)
    // espera o botão ficar disponível após o loading sumir
    const addButton = await screen.findByRole('button', { name: /Adicionar Usuário/i })
    await userEvent.click(addButton)
    expect(screen.getByLabelText(/Nome:/i)).toBeInTheDocument()
    expect(screen.getByLabelText(/Email:/i)).toBeInTheDocument()
  })
})
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

render(<Users />): monta o componente em um ambiente de teste (DOM virtual).

screen: é uma interface global para buscar elementos renderizados (por papel, texto, rótulo, etc.).

userEvent.click(...): simula um clique real do usuário.

vi.spyOn(api, 'get'): intercepta a função api.get (geralmente uma requisição HTTP).
O mockResolvedValue faz com que ela retorne uma resposta falsa e *não chame a API de verdade*.

O que acontece no teste:

- O componente é renderizado.
- O teste procura o botão “Adicionar Usuário” por papel (button) e nome visível.
- O clique é simulado.
- Após o clique, o formulário é exibido.
- O teste confirma que os campos com os rótulos “Nome” e “Email” estão presentes no DOM.

Assim, validamos **comportamento e acessibilidade**, não apenas estrutura HTML.

PASSO A PASSO: TESTANDO O COMPONENTE USERS

6) Executar

Rodar: docker-compose exec frontend npm test

Modo watch: docker-compose exec frontend npm run test:watch

Cobertura: docker-compose exec frontend npm run test:coverage

PASSO A PASSO: TESTANDO O COMPONENTE USERS

7) Cenário de erro

Crie `frontend/tests/unit/Users.error.test.tsx`

```
import { render, screen, waitFor } from '@testing-library/react'
import { vi } from 'vitest'
import Users from '../../src/components/Users'
import { api } from '../../src/services/api'

describe('Users (falha de carregamento)', () => {
  it('exibe mensagem de erro quando a API falha', async () => {
    vi.spyOn(api, 'get').mockRejectedValue(new Error('Network error'))

    render(<Users />)
    await waitFor(() => {
      expect(screen.getByText(/Erro ao carregar usuários/i)).toBeInTheDocument()
    })
  })
})
```

PASSO A PASSO: TESTANDO O COMPONENTE USERS

No código anterior testamos um cenário negativo: quando a API retorna erro.

mockRejectedValue: simula uma *falha na promessa*, gerando erro de rede.

waitFor() : aguarda o componente atualizar o DOM de forma assíncrona (por exemplo, exibir uma mensagem de erro após o catch).

getByText() : verifica se a mensagem de erro realmente apareceu.