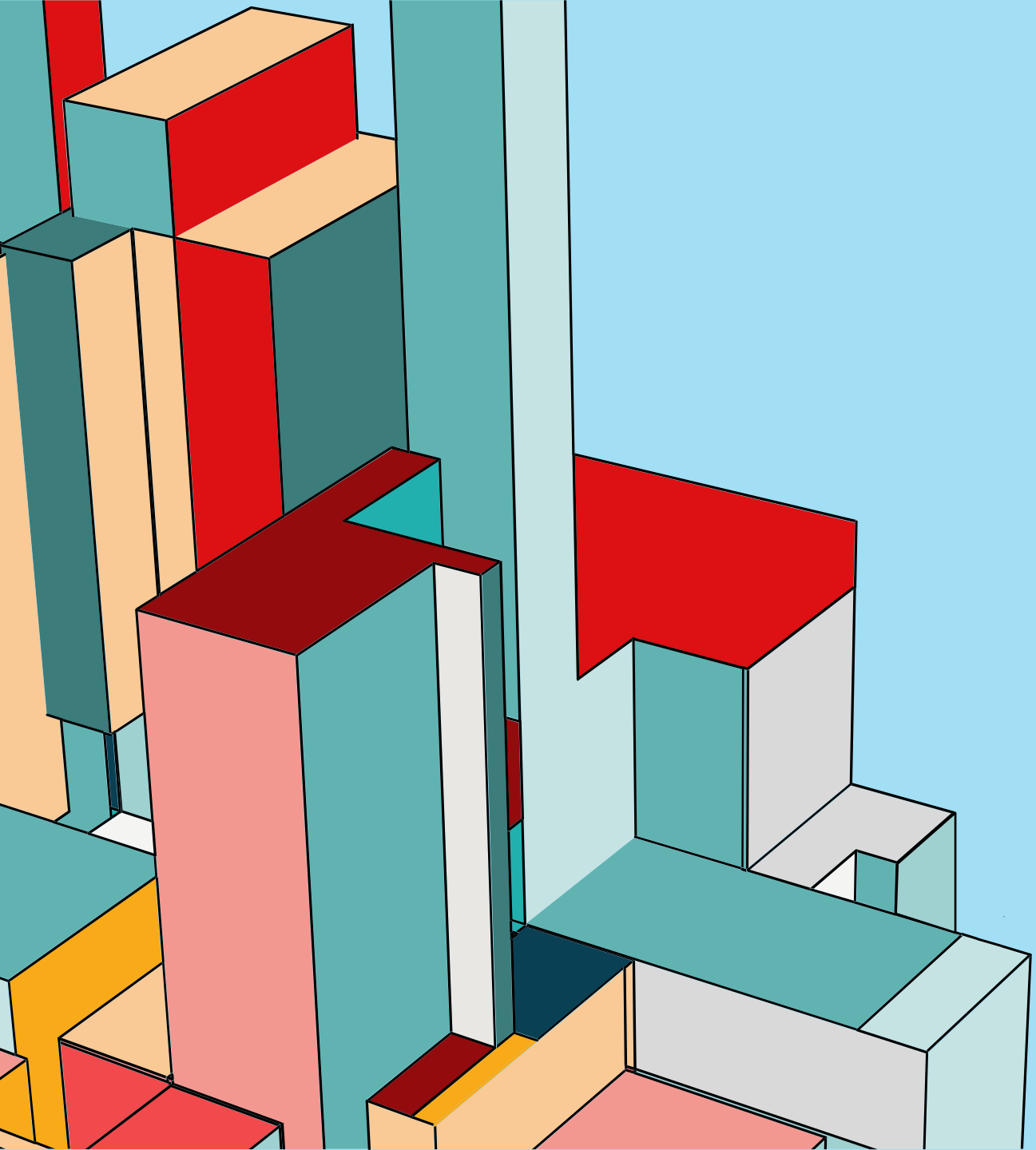
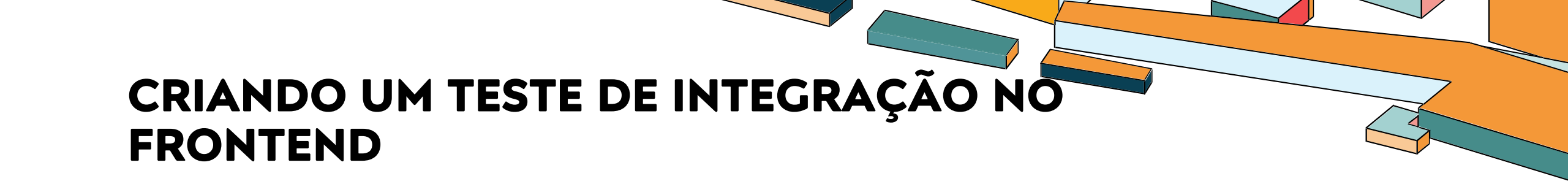


UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

TESTE DE SOFTWARE

TESTES DE INTEGRAÇÃO - FRONTEND

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



CRIANDO UM TESTE DE INTEGRAÇÃO NO FRONTEND

O objetivo da prática será simular a resposta do backend e verificar se o componente `Users` reage corretamente, usando **Vitest**, **Testing Library** e **MSW**

Sendo que:

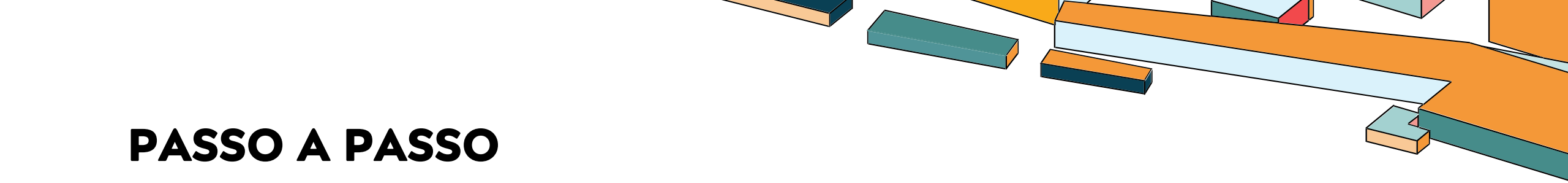
- **Vitest** é o framework executor dos testes.
- **Testing Library** foca no comportamento do usuário e no que aparece na tela, incentivando testes menos acoplados à implementação.
- **MSW (Mock Service Worker)** Simula a api interceptando `fetch/axios` e respondendo com dados “falsos”, preservando o contrato http.
- Juntos, eles permitem testar a integração do componente com a camada de HTTP sem bater no backend.

PASSO A PASSO - AJUSTE

O `supertest` ao importar backend/src/index.ts nos testes, vai executar o app.listen e tentar abrir a porta que já está ocupada pelo server em dev (docker).

Vamos condicionar o `listen` para não rodar em testes. O Vitest define **process.env.VITEST** automaticamente. Edite **backend/src/index.ts** e troque o trecho do **listen** (antes do fim do arquivo) por:

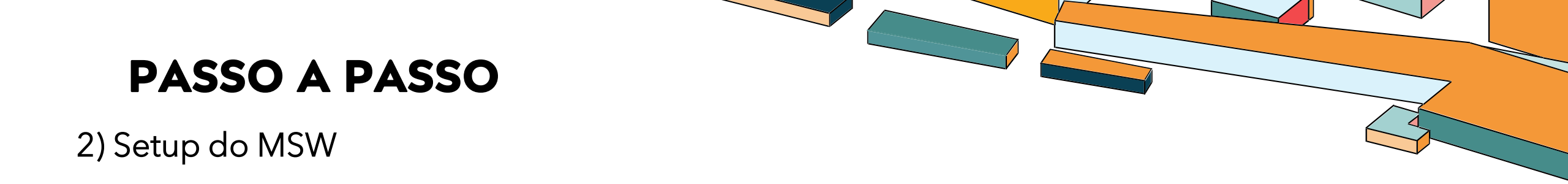
```
if (process.env.NODE_ENV !== 'test' && !process.env.VITEST) {
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`)
    console.log(`Health check: http://localhost:${PORT}/health`)
    console.log(`API docs: http://localhost:${PORT}/api`)
  })
}

export { prisma }
export default app
```



PASSO A PASSO

1) Instalar dependências de teste (frontend):

```
docker-compose exec frontend npm i -D msw
```



PASSO A PASSO

2) Setup do MSW

Dentro do arquivo frontend/tests/setup.ts vamos adicionar o seguinte código

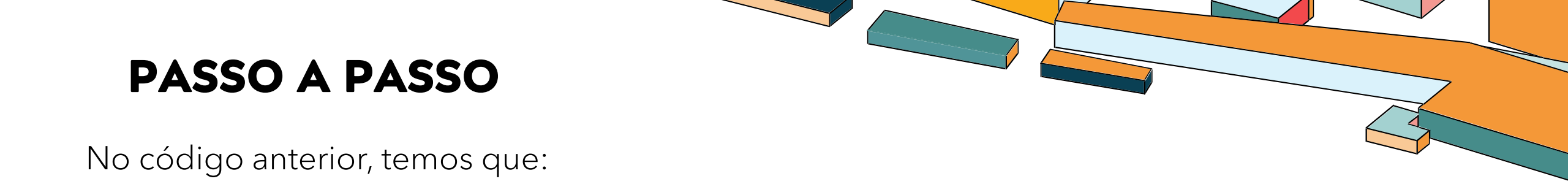
```
import '@testing-library/jest-dom' //essa linha já existe
import { setupServer } from 'msw/node'
import { http, HttpResponse } from 'msw'

export const server = setupServer()
beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

// helper para rotas da API local (usa wildcard de host)
export const apiGet = (path: string, resolver: Parameters<typeof http.get>[1]) =>
  http.get(`*/api${path}`, resolver)

// helper para responder JSON
export const json = (body: any, init?: ResponseInit) => HttpResponse.json(body, init)
```

PASSO A PASSO

3) Teste de integração (carregar lista de usuários via API)

Crie dentro da pasta integration um arquivo de nome Users.load.int.test.tsx

```
// frontend/tests/integration/Users.load.int.test.tsx
import { render, screen, waitFor } from '@testing-library/react'
import Users from '../../src/components/Users'
import { server, apiGet, json } from '../setup'

describe('Users integration - carga de lista', () => {
  it('renderiza usuários retornados pela API', async () => {
    server.use(
      apiGet('/users', (_req) =>
        json({
          data: [
            { id: '1', name: 'Ana', email: 'ana@ex.com', createdAt: new Date().toISOString(), tasks: [] },
          ]
        })
      )
    )

    render(<Users />)

    await waitFor(() => {
      expect(screen.getByText('Ana')).toBeInTheDocument()
      expect(screen.getByText('ana@ex.com')).toBeInTheDocument()
    })
  })
})
```



PASSO A PASSO

No código anterior, temos que:

`vitest` fornece `describe, it, expect`

`testing library` expõe `render`, seletores como `screen.getByText` e utilitários assíncronos como `waitFor` quando há chamadas assíncronas.

`msw` cria um servidor de mocks via `setupServer()`, `com server.listen()` antes dos testes, `server.resetHandlers()` após cada um e `server.close()` ao final.

O helper `apiGet()` cria o `handler http.get/...` e, nos testes, ativamos esse handler específico com `server.use(...)`.

O teste 'renderiza usuários retornados pela API':

- Usa o handler do MSW para retornar uma lista simulada de usuários.
- O componente Users consome a rota e exibe os dados.
- O teste verifica se os nomes aparecem na tela.

PASSO A PASSO

4) Teste de integração (erro de API exibindo mensagem de erro)

Crie dentro da pasta integration um arquivo de nome Users.error.int.test.tsx

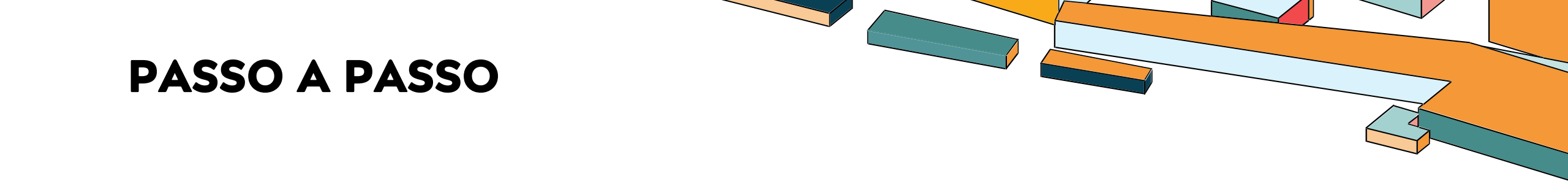
```
import { render, screen, waitFor } from "@testing-library/react";
import Users from "../../src/components/Users";
import { server, apiGet } from "../setup";

describe("Users integrations - falhas da API", () => {
    it("mostra mensagem de erro quando a API falha", async () => {
        server.use(
            apiGet('/users', () => HttpResponse.error())
        )
        render(<Users />);
        await waitFor(() => {
            expect(
                screen.getByText(/Erro ao carregar usuários/i)
            ).toBeInTheDocument();
        });
    });
});
```

PASSO A PASSO

O teste 'mostra mensagem de erro quando a API falha':

- Garante integração em cenários ruins.
- Ao simular `HttpResponse.error()`, o teste verifica se o componente exibe a mensagem de erro e evita estados quebrados.

PASSO A PASSO

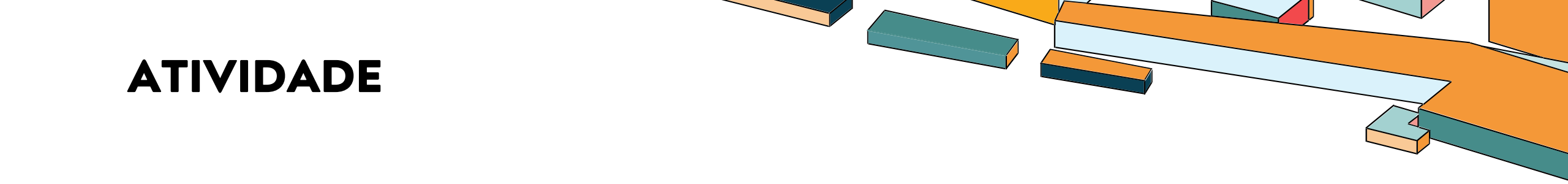


5) Executar

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



ATIVIDADE

- Crie um teste para um componente que consome a mesma rota (Categoria ou Tarefas), criando **`handler`** de sucesso e **`handler`** de erro, garantindo renderização e fallback de falha.