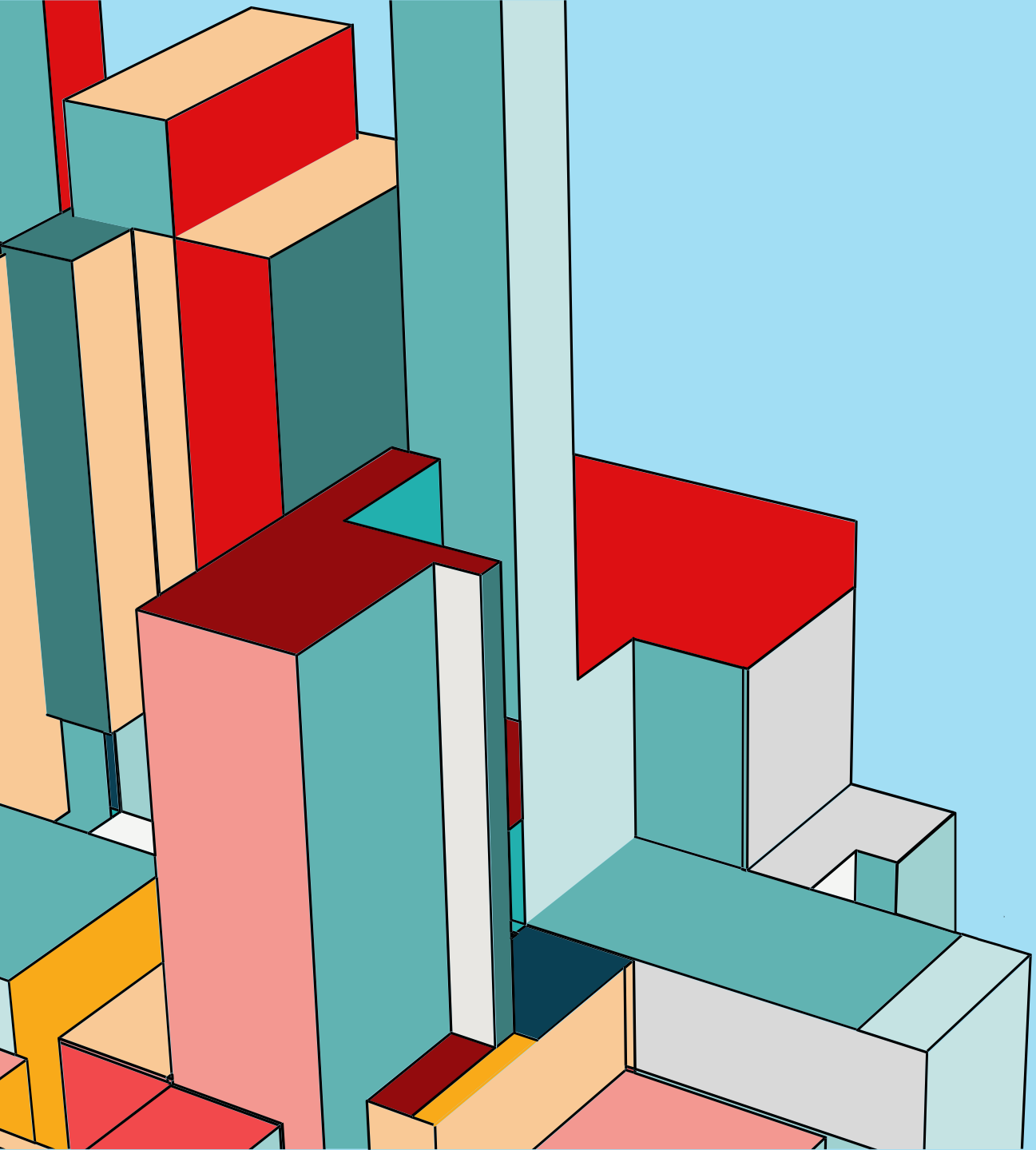


UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

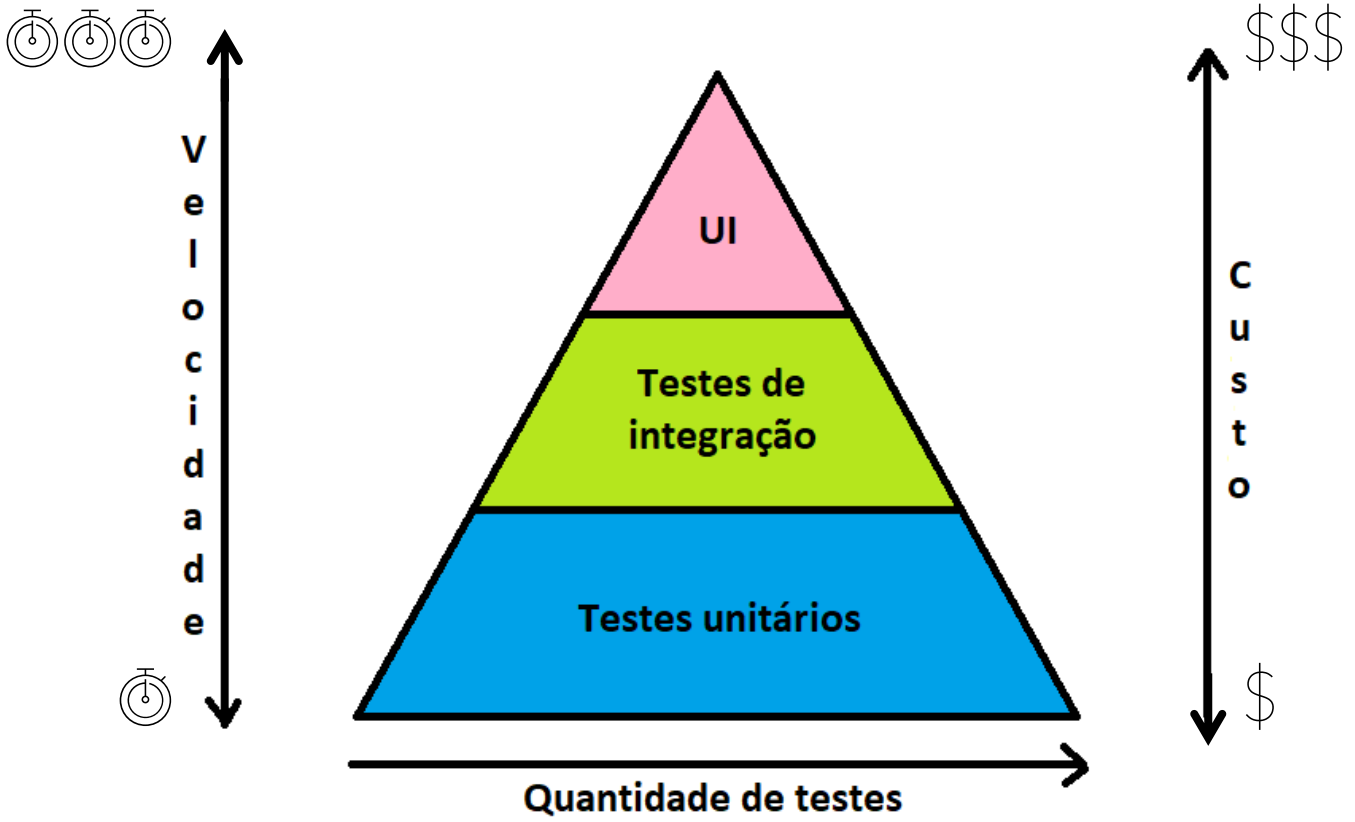
TESTE DE SOFTWARE

TESTES DE INTEGRAÇÃO - BACKEND

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

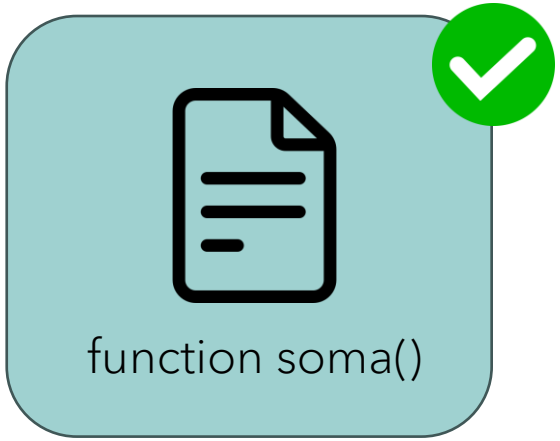
TESTES DE INTEGRAÇÃO

- Testes de integração juntam e testam **diversos componentes ou módulos da aplicação** para avaliar como **funcionam em conjunto**.
- Os testes de integração buscam garantir que essas partes reunidas consigam se comunicar e interagir com sucesso.
- Enquanto os testes unitários avaliam unidades isoladas, **os de integração validam fluxos completos**, por exemplo, uma requisição chegando a uma rota, passando pelo serviço e chegando ao banco.

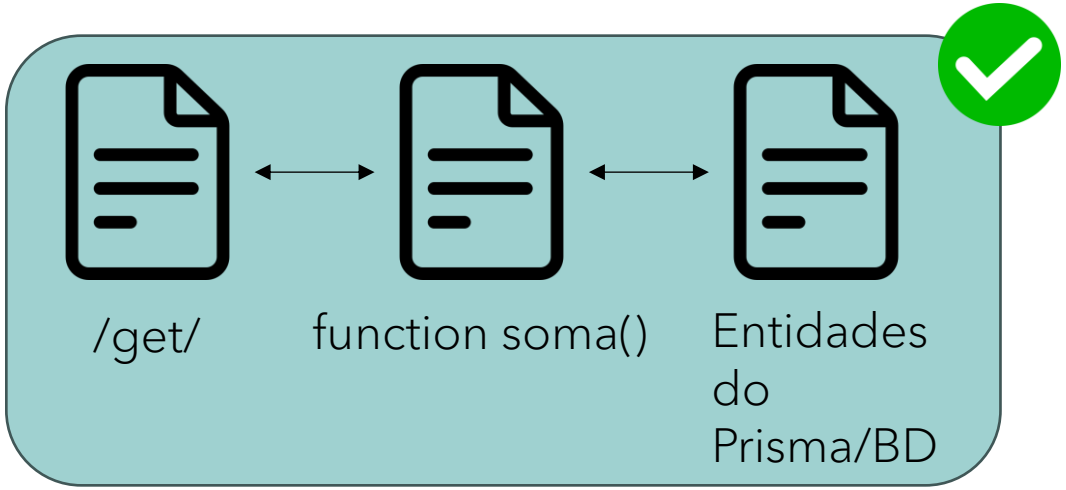


DIFERENÇA ENTRE UNITÁRIO E INTEGRAÇÃO

Um teste unitário testa o "**como**", verificando se uma função retorna o resultado esperado isoladamente.

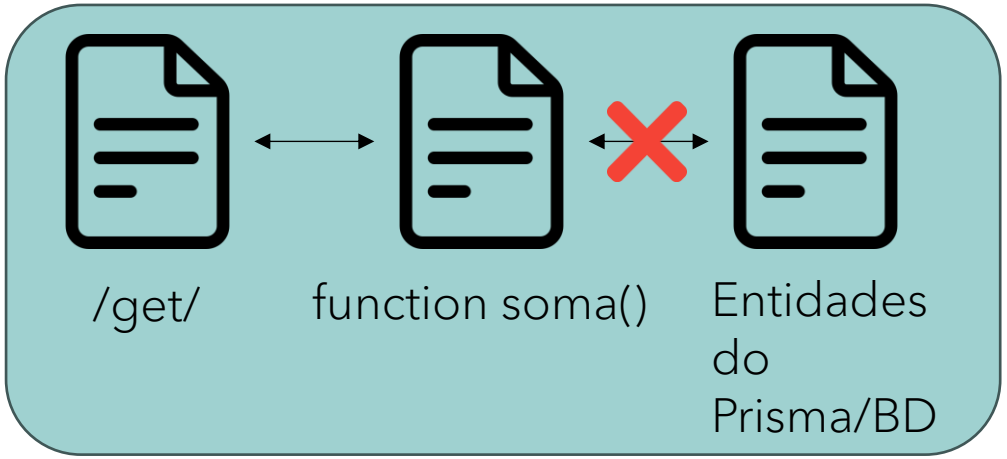


Um teste de integração testa o "**fluxo**", verificando se múltiplos **componentes** trabalham juntos corretamente.



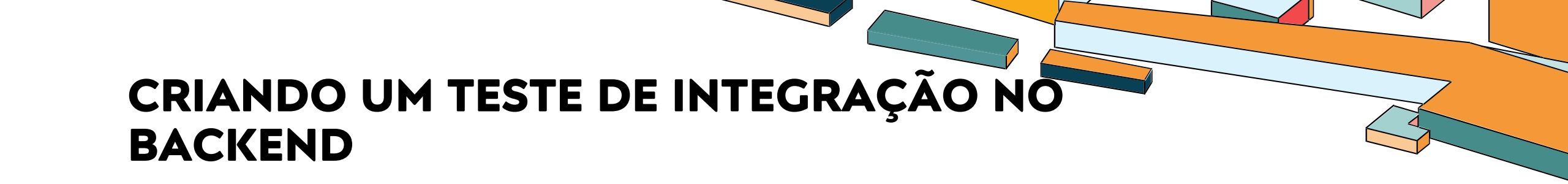
IMPORTÂNCIA DOS TESTES DE INTEGRAÇÃO

- Eles simulam o uso real do sistema sem precisar de um ambiente completo.
- Ajudam a detectar falhas de comunicação entre módulos, problemas de validação e incompatibilidade de dados entre camadas.



PRINCIPAIS CARACTERÍSTICAS

- Executam caminhos reais do código
- Usam dados reais ou simulações controladas
- Validam entradas, saídas e efeitos colaterais
- Não dependem de servidor ou navegador reais
- Garantem que módulos interajam de forma previsível



CRIANDO UM TESTE DE INTEGRAÇÃO NO BACKEND

O objetivo da prática será testar as rotas da API para usuários (`/api/users`) usando **Vitest**, **Supertest** e **Prisma**

Sendo que:

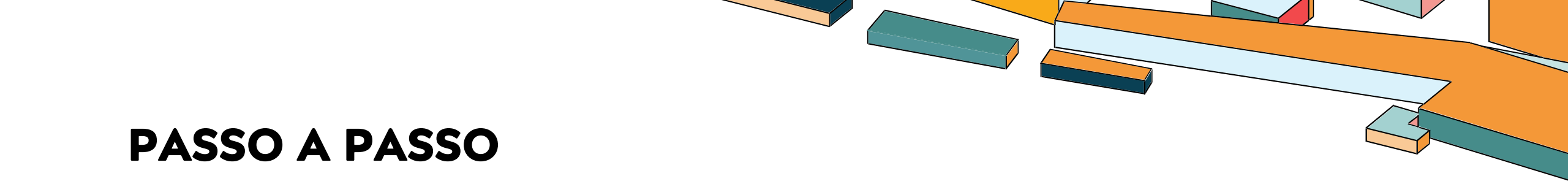
- **Vitest** é o framework executor dos testes.
- **Supertest** permite fazer chamadas HTTP diretamente ao objeto `app` do `express` dentro do processo, evitando `app.listen` e portas reais, mas exercitando middlewares, validações e roteamento.
- **Prisma** fornece acesso ao banco, permitindo limpar e recriar dados antes de cada teste.
- Juntos, eles permitem testar **rota→validação→serviço→banco** como acontece em produção, só que em ambiente controlado.

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:

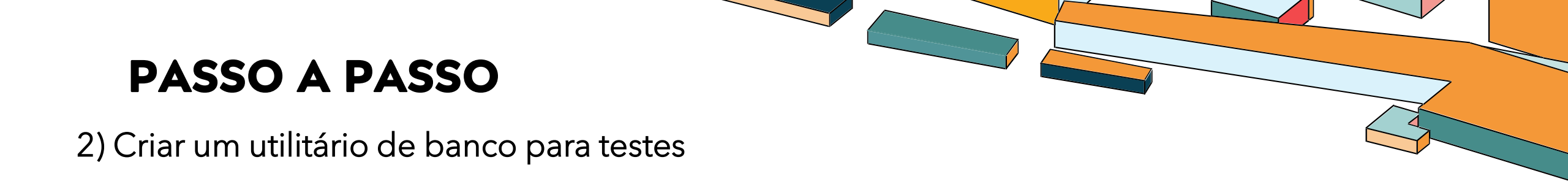
```
if (process.env.NODE_ENV !== 'test' && !process.env.VITEST) {
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`)
    console.log(`Health check: http://localhost:${PORT}/health`)
    console.log(`API docs: http://localhost:${PORT}/api`)
  })
}

export { prisma }
export default app
```



PASSO A PASSO

1) Instalar dependências de teste (backend):

```
docker-compose exec backend npm i -D supertest @types/supertest
```



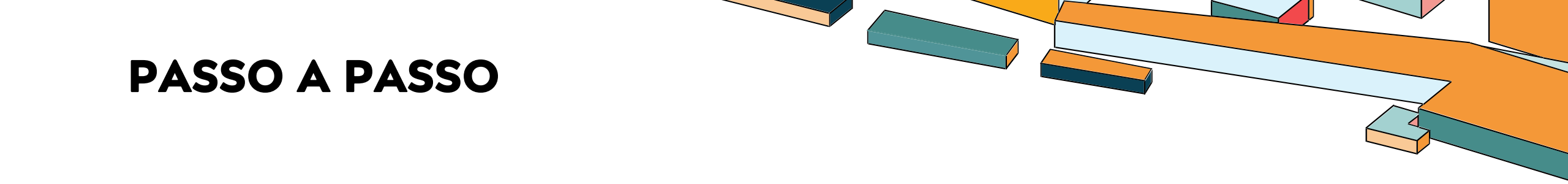
PASSO A PASSO

2) Criar um utilitário de banco para testes

Crie uma pasta chamada integration dentro de tests (do backend)

Crie um arquivo de nome testDb.ts

```
import { PrismaClient } from '@prisma/client'
export const prisma = new PrismaClient()

export async function resetDb() {
  await prisma.task.deleteMany()
  await prisma.category.deleteMany()
  await prisma.user.deleteMany()
}
export async function seedMinimal() {
  const user = await prisma.user.create({ data: { name: 'Ana', email: 'ana@ex.com' } })
  const category = await prisma.category.create({ data: { name: 'Work' } })
  return { user, category }
}
```



PASSO A PASSO

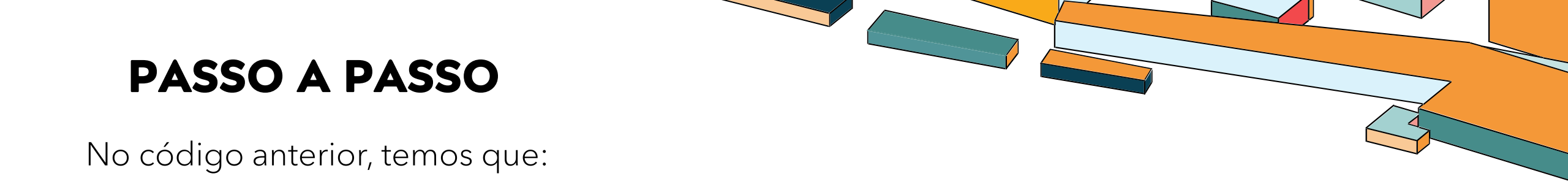
O arquivo anterior cria funções utilitárias para que possamos remove registros das tabelas envolvidas (função `resetDb()`) e cria dados básicos quando necessário (por exemplo, um usuário e uma categoria) através da função `seedMinimal()` . Em testes de integração, consistência é tudo, por isso precisamos de um ambiente controlado.

PASSO A PASSO

3) Teste de integração (criar e listar usuários)

Crie dentro da pasta integration um arquivo de nome users.int.test.ts

```
import { describe, it, beforeAll, afterAll, beforeEach, expect } from 'vitest'
import request from 'supertest'
import app, { prisma as appPrisma } from '../../src/index'
import { prisma, resetDb } from './testDb'

describe('Users API', () => {
  afterAll(async () => {
    await prisma.$disconnect()
    await appPrisma.$disconnect()
  })
  beforeEach(async () => {
    await resetDb()
  })
  it('POST /api/users cria usuário válido', async () => {
    const res = await request(app)
      .post('/api/users')
      .send({ name: 'Ana', email: 'ana@ex.com' })
    expect(res.status).toBe(201)
    expect(res.body.data).toMatchObject({ name: 'Ana', email: 'ana@ex.com' })
  })
  it('GET /api/users lista usuários', async () => {
    await prisma.user.create({ data: { name: 'Ana', email: 'ana@ex.com' } })
    const res = await request(app).get('/api/users')
    expect(res.status).toBe(200)
    expect(Array.isArray(res.body.data)).toBe(true)
    expect(res.body.data.some((u: any) => u.email === 'ana@ex.com')).toBe(true)
  })
})
```



PASSO A PASSO

No código anterior, temos que:

`vitest` fornece `describe, it, expect`, além de ganchos como `beforeEach()` e `afterAll()` para preparar e limpar o cenário.

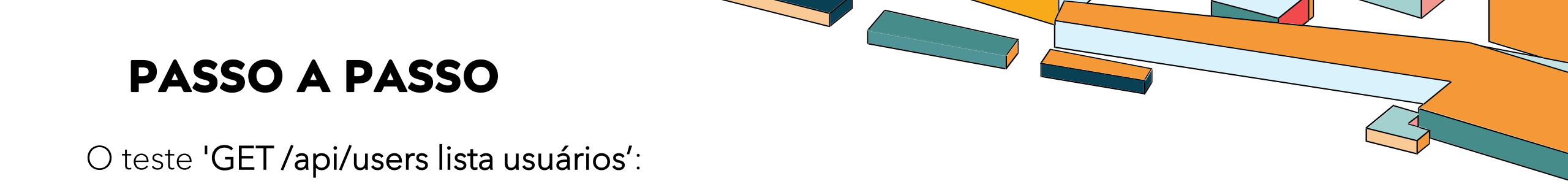
`supertest` expõe `request(app)` para enviar `get, post, put, delete` e inspecionar `status` e `body`.

`prisma` oferece `prisma.model.create, findMany e deleteMany` e o `.$disconnect()` para encerrar conexões;

O helper `resetDb()` é usado para limpar estado entre execuções e evitar interferência entre casos.

O teste 'POST /api/users cria usuário válido':

- Ao chamar `request(app).post('/api/users').send({ name, email })`, o `supertest` envia uma requisição real ao `express`. O teste valida que o status retorna 201 e que o corpo contém os campos esperados.
- Isso cobre o encadeamento controller→validação→serviço→prisma. Se a rota, o `schema` ou o repositório quebrarem, o teste falha, revelando falhas de integração.



PASSO A PASSO

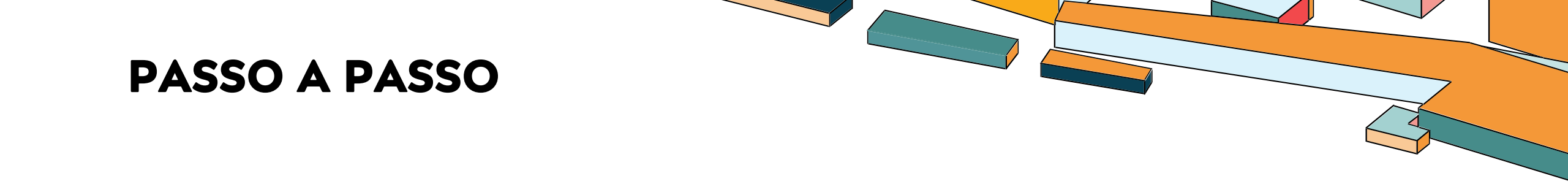
O teste 'GET /api/users lista usuários':

- Cria um usuário diretamente com `prisma`
- Chama `request(app).get('/api/users')`
- Espera um status 200 e que a lista inclua o e-mail inserido
- Isso confirma se o caminho rota→serviço→consulta ao banco→serialização está íntegro.

A importância do `afterAll()`:

Ao final do teste, a função `afterAll()` é executada, chamando `prisma.$disconnect()` e, se o `app` exportar uma instância de prisma ou abrir algo adicional, também é desconectado. Isso evita "open handles" no vitest, garante velocidade em execuções repetidas e estabilidade no CI.

PASSO A PASSO

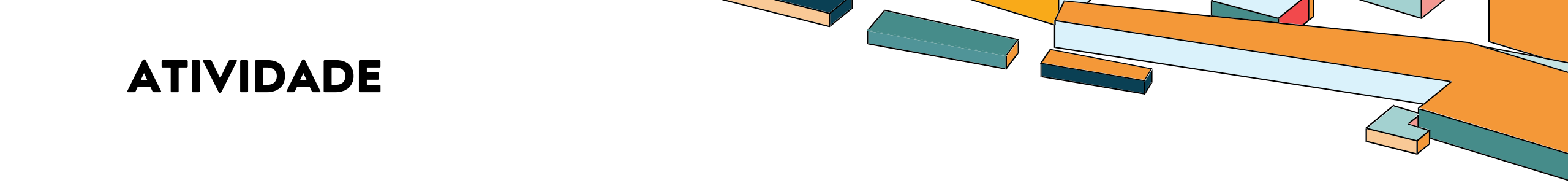


4) Executar

Rodar: docker-compose exec backend npm test

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

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



ATIVIDADE

- Crie os testes para atualizar e excluir Usuários.
- Crie os teste de CRUD para todas as rotas de Tarefas (/api/tasks/)