Framework de testes

Introdução ao Vitest

Framework oficial de testes do ecossistema Vite

Professor: F.Laurindo Costa jr

O que é o Vitest?

Definição

Vitest é o framework oficial de testes do ecossistema Vite, criado pela mesma equipe do Vue/Vite.

Ele substitui Jest em projetos modernos, oferecendo uma experiência de teste mais rápida e integrada ao ecossistema Vue e Vite.

Comparação com Jest

Vitest	Jest
Integração com Vite	Configuração complexa
Configuração mínima	Menos integrado ao Vue 3
Suporte ESM	Requer adaptadores para ESM
Performance superior	Performance lenta

Principais Vantagens

Mais rápido
Utiliza a arquitetura do Vite com HMR, resultando em execução instantânea.

Integrado ao Vite
Funciona perfeitamente com o ecossistema Vite sem configuração extra.

Configuração simplificada
Defaults inteligentes para projetos Vue/Vite.

Por que usar Vitest no nosso projeto?

Aplicação de Agendamentos

No contexto da nossa aplicação, o Vitest oferece benefícios específicos que melhoram a qualidade e confiabilidade do código.

> *"Vitest + Pinia = ambiente perfeito para testes em Vue 3."*

Benefícios específicos para o nosso projeto:

Testar a lógica das Stores (Pinia)
Valida regras de negócio sem dependência de backend.

Garantir que CRUD funciona sem backend
Mockamos services para testar operações CRUD.

Garantir que mudanças não quebrem o fluxo
Protector contra regressões em novas funcionalidades.

Como o Vitest funciona?

O Vitest trabalha com três conceitos principais que formam o coração de qualquer suite de testes:

describe()

Agrupa testes relacionados

```
describe("Consulta Store", () =>
{ ... })
```

`describe()` é usado para **organizar** e **agrupar testes relacionados**.

Criar sessões lógicas (“Consultas Store”, “Modal Component”, etc.)

Facilitar leitura dos testes

Ajudar a manter o código limpo

Isolar testes com `beforeEach()` e `afterEach()`

it() ou test()

Define um caso de teste

```
it("deve adicionar uma consulta",
() => { ... })
```

`it()` (ou `test()`) define **um teste específico**, com **um único objetivo**.

Representa um comportamento esperado

Explica *o que* o sistema deve fazer

É executado individualmente

expect()

Faz asserções (o que esperamos)

```
expect(store.consultas.length).toBe(1)
```

`expect()` **é onde verificamos se o resultado do código corresponde ao esperado.**

Fazer validações (“asserções”)

Detectar erros

Garantir que seu sistema está funcionando corretamente

Testando Stores - Configuração Inicial

1. Criar Pinia de teste

Inicie cada teste com um estado limpo para evitar interferências.

</> use setActivePinia(createPinia())

2. Mockar o Service

Substitua o módulo real pelo mock para evitar chamadas à API.

</> vi.mock('@/services/consultas.service')

Aprender

- Como configurar ambiente de teste isolado
- Como substituir dependências por mocks

Exemplo

```
import { setActivePinia, createPinia } from 'pinia'import {
useConsultaStore } from
'@/stores/consultas.store'vi.mock('@/services/consultas.service', () =>
({
  getConsultas:
vi.fn().mockResolvedValue([]),
  createConsulta:
vi.fn().mockResolvedValue({ id: '1', motivo: 'Teste' })
}))

beforeEach(() => {
  setActivePinia(createPinia())
})
```

O mock substitui chamadas à API, permitindo testes previsíveis.

Por que o Vitest usa vi?

O “vi” é o objeto global responsável por:

- criar mocks
- criar spies
- criar funções Simuladas
- limpar mocks
- redefinir módulos
- simular timers
- simular módulos inteiros

Mockar um service

```
vi.mock('@/services/consultas.service', () => ({
  getConsultas: vi.fn().mockResolvedValue([]),
  createConsulta: vi.fn().mockResolvedValue({ id:
"1", motivo: "Teste" })
}));
```

Criar uma função simulada

```
const fn = vi.fn()
fn()
expect(fn).toHaveBeenCalled()
```

✔ **Espionar método de objeto**

```
vi.spyOn(store, "addConsulta")
```

✔ **Resetar mocks entre testes**

```
vi.clearAllMocks()
```

Testando Componentes Vue

Usando @vue/test-utils

O Vitest usa @vue/test-utils para testes de componentes Vue, permitindo montar e interagir com componentes isolados.

Testando BaseModal.vue

```
// BaseModal.spec.ts
import { mount } from "@vue/test-utils"
import BaseModal from "@/components/BaseModal.vue"

it("renderiza conteúdo corretamente", () => {
  const wrapper = mount(BaseModal, {
    slots: {
      default: "<p>Conteúdo do modal</p>"
    }
  })

  expect(wrapper.html()).toContain("Conteúdo do modal")
})
```

O que estamos testando:

- Renderização do conteúdo passado via slot
- Verificação do HTML gerado pelo componente

Testando BaseToast.vue

```
// BaseToast.spec.ts
import { mount } from "@vue/test-utils"
import BaseToast from "@/components/BaseToast.vue"

it("mostra mensagem passada via prop", () => {
  const wrapper = mount(BaseToast, {
    props: { message: "Sucesso!" }
  })

  expect(wrapper.text()).toContain("Sucesso!")
})
```

O que estamos testando:

- Renderização da mensagem passada via prop
- Verificação do texto exibido pelo componente

Organização e Boas Práticas

Estrutura de Pastas

A melhor organização coloca os testes **ao lado do arquivo testado ou numa pasta _test**, facilitando a manutenção.

```
src/
  stores/
    consultas.store.ts
    consultas.store.spec.ts
  components/
    BaseModal.vue
    BaseModal.spec.ts
  services/
    consultas.service.ts
    consultas.service.spec.ts
```

Essa abordagem facilita a manutenção e segue o padrão internacional "teste ao lado do arquivo testado".

Erros Comuns

Esquecer do await
Teste passa "falso positivo". Sempre usar await store.action()

Não limpar o estado entre testes
Usar beforeEach(() => setActivePinia(createPinia()))

Mock mal configurado
Módulos importados devem ser mockados antes do uso

Testar API real
Testes devem ser previsíveis → somente mocks

Resumo

Principais pontos sobre Vitest

Framework oficial do Vite

Criado pela mesma equipe do Vue/Vite, substituindo o Jest em projetos modernos.

Testes de stores e components

Permite testar a lógica das stores com Pinia de teste e components com @vue/test-utils.

Desempenho e simplicidade

Mais rápido e simples que Jest, com configuração automática e API intuitiva.

Mocking nativo

Permite mockar serviços reais sem precisar de backend, garantindo que o CRUD funcione sem regressões.

Integração perfeita

Funciona perfeitamente com Vue 3, Pinia e Components SFC, com suporte nativo a ESM.