Pinia: Gerenciamento de Estado no Vue 3

Apresentação sobre o Pinia, o gerenciador de estado oficial do Vue 3, explorando conceitos fundamentais e aplicações práticas.

Professor: F.Laurindo Costa jr

O que é o Pinia?

Definição Oficial

Pinia é a **loja oficial do Vue 3** para gerenciamento de estado. Funciona como uma caixa central onde guardamos e manipulamos estados que várias telas precisam acessar.

Principais Funcionalidades

- Compartilhar estado global entre componentes
- Organizar dados e regras de negócio
- Centralizar chamadas de API
- Evitar "perfuração de adereços" (prop drilling)

Conceito Simplificado

"Pinia é uma caixa central onde guardamos e manipulamos estados que várias telas precisam acessar."

Estrutura de uma Store

1. State

- Definido com `ref()` ou `computed()`
- Guarda os dados da aplicação

2. Actions

- Funções assíncronas que manipulam o estado
- Contêm a lógica de negócio

3. Getters

- Expõe para os componentes quais estados e ações estão disponíveis

Como definimos uma Store?

```
defineStore('id-do-store', () => { ... })
```

Por exemplo: `'consultas'` é o **ID único**

```
export const useConsultaStore =
defineStore('consultas', () => {

export const useMedicoStore =
defineStore('medicos', () => {

export const usePacienteStore =
defineStore('pacientes', () => {
```

Você consegue acessar a store em qualquer componente:

```
const consultaStore =
useConsultaStore()
```

State no Store

Exemplos Práticos

Consultas

```
const consultas = ref<Consulta[]>([])
const editingConsulta = ref<Consulta | null>(null)
```

Médicos

```
const pacientes = ref<Paciente[]>([])
```

Pacientes

```
const medicos = ref<Medico[]>([])
```

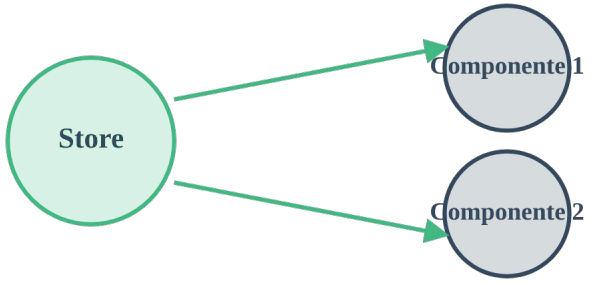
Isso é reatividade global funcionando no seu sistema.

Por que usar ref()?

O estado precisa ser reativo: quando a loja muda, a tela é atualizada automaticamente.

> A reatividade é o coração do Pinia. Ela garante que sua interface de usuário sempre esteja sincronizada com os dados.

Como a reatividade funciona



Actions - Lógica de Negócio + API

Actions são funções dentro do store usadas para gerenciar lógica assíncrona e interações com a API.

Propósito das Actions:

- Buscar dados
- Cadastrar novos registros
- Editar registros existentes
- Atualizar estado global

Lembre-se: Actions são o lugar certo para colocar lógica assíncrona. Componentes só chamam as actions.

Exemplos de Actions:

Buscar Consultas

```
const fetchConsultas = async () => {
consultas.value = awaitgetConsultas()
}
```

Adicionar Consulta

```
const addConsulta = async (consultaData) =>
const nova =
awaitcreateConsulta(consultaData)
consultas.value.push(nova)
}
```

Atualizar Consulta

```
const updateConsulta = async (id, data) =>
const atualizado =
awaitupdateConsultaService(id, data)
const i = consultas.value.findIndex(c => c.id
=== id)
consultas.value[i] = atualizado
}
```

Getters — Valores derivados

Getters computed properties dentro da store.

Propósito das Getters:

Esses métodos não armazenam dados, apenas derivam informações do state.

evita lógica repetida em várias telas.

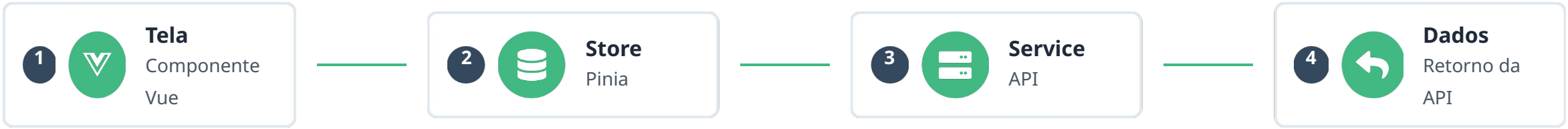
Exemplos de Getters:

```
const getMedicoNome = (id: string) => {
  const medico = medicos.value.find(m =>
m.id === id)
  return medico ? `${medico.nome}
(${medico.especialidade})` :
'Desconhecido'
}
```

```
const getPacienteNome = (id: string) => {
  const paciente = pacientes.value.find(p
=> p.id === id)
  return paciente ? paciente.nome :
'Desconhecido'
}
```

As funções são getters na prática, apenas não declarados como computed() porque não existe dependência reativa que precise ser recalculada.

Fluxo Completo da Aplicação



Fluxo de Requisição

- Componente chama método da store
- Store executa lógica de negócios
- Store chama serviço API
- API retorna dados para store

Fluxo de Resposta

- Store atualiza estados reativos
- Reatividade dispara atualização automática
- Componente recebe dados atualizados
- Tela é renderizada com novos dados

Quando o usuário clica em “Editar”:

```
consultaStore.setEditingConsulta(consulta)
```

O formulário usa `watch()` para reagir:

```
watch(editingConsulta, (nova) => {  if (nova)
 preencherFormulario(nova)
})
```

Exemplo Prático: Agendar Consulta

ConsultaForm.vue

consultaStore

API Service

1 Usuário clica em "Salvar"

```
<form @submit="onSubmit">
  <buttontype="submit">Salvar</button>
</form>
```

2 Componente chama store

```
const onSubmit = async () => {
  awaitconsultaStore.addConsulta(form.value);
  // Limpa o form
}
```

3 Store atualiza a lista

```
const addConsulta = async (consultaData) => {
  const nova =
awaitcreateConsulta(consultaData);
  // Atualiza a lista reativa
  consultas.value.push(nova);
}
```

Pontos importantes:

- A lista de consultas é atualizada automaticamente graças ao **ref()**
- O store gerencia a lógica de negócio e chamadas à API

Dúvidas Comuns

"Por que não usamos data() nos componentes?"

Porque o estado é global — várias telas precisam do mesmo dado. O **data()** cria propriedades locais a cada componente, não compartilháveis automaticamente.

"Posso alterar a loja direto sem usar ações?"

Tecnicamente sim, mas má prática. Manter o código desorganizado. Sempre use ações para garantir que as mudanças de estado sejam rastreáveis e reativas.

"O Pinia substitui adereços?"

Não. Ele substitui apenas parte da comunicação entre componentes. Props ainda são úteis para comunicação direta entre componentes parente-filho.

"Cada loja guarda uma tabela?"

No seu projeto, sim — é o padrão CRUD: pacientes.store.ts, medicos.store.ts, consultas.store.ts. Cada storegerencia uma entidade do banco de dados.

"Por que usar Pinia ao invés de Vuex?"

Pinia substitui o Vuex porque é mais simples, combina com Composition API, elimina mutations, funciona melhor com TypeScript, é a store oficial do Vue 3, e torna nossa arquitetura mais limpa e moderna.