Vue.js (State Management)

Interface Pessoa-Máquina - 25/26 - LEI / UM

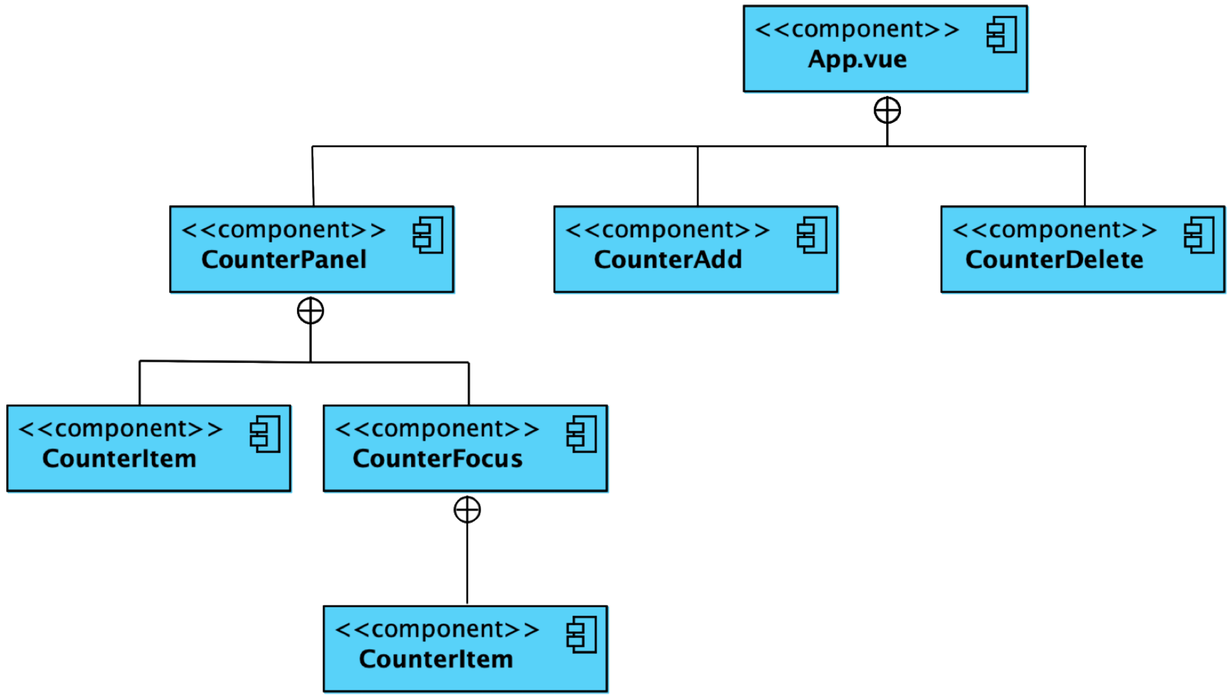
Hugo Pacheco
hpacheco@di.uminho.pt

State Management em Vue

- Até agora, algumas limitações a gerir estado para hierarquias de components
  - 💡 Colocar tudo no root component (por causa de unidirectional data flow)
    - 💡 Root component propaga para descendentes
      - ❗ `Props` têm que ser passadas explicitamente de pai para filho ao longo da hierarquia
    - 💡 Root component regista callbacks em eventos dos filhos
      - ❗ Não existe bubbling, cada componente na hierarquia tem que registar um event handler e re-emitir evento para o seu pai
    - ⛔ Passthrough components ⇒ código repetido

Exemplo (Counters)

- Relembrar um exemplo de components
  - `App.vue` guarda estado dos vários contadores
  - Cadeia de `props` / event handlers pra sub-components



Provide / Inject

- Uma alternativa a `props`, mesmo unidirectional data flow
- Componente **“provides”** parte do seu estado apenas a descendentes

```
const name = ref(…);
provide('name', name);
```

Objeto a registar

  - Componente **“injects“** estado necessariamente de um ascendente

```
const name = inject(‘name');
```

Nome do registo

  - Passado implicitamente pela hierarquia (ao contrário de `props`)
  - Podemos passar qualquer objeto, inclusive a própria `ref` (tal como `props`)

Exemplo (Counters + Provide/Inject)

- Substituir `props` de contadores por `provide`
  - Remover passagem de `props` como atributos
  - Registar `provide` nos producers
  - Registar `inject` nos consumers
  - Para cada contador isolado
    - Passar apenas o id do contador como `prop`
    - Contador como computed property sobre a injected `ref` de contadores

Provide / Inject: bidirecional?

- Uma variante implícita de unidirectional data flow entre components
  - ✚ Evitar passthrough component logic de pais para filhos
- Mas o que acontece no sentido inverso?
  - Event handlers encadeados exatamente como antes…
    - ➖ Continuamos a ter passthrough component logic de filhos para pais
  - JS é uma linguagem funcional ⇒ higher-order!
    - 💡 Podemos passar qualquer objeto, inclusive funções com `props` / `provide`
    - ✚ Passar event handlers com `props` / `provide`

Exemplo (Counters + Provide/Inject + Events)

- Remover events para incrementar, adicionar e remover contadores
- Trocar painel de mensagens de `CounterPanel` para `App`
  - Root component `App` “provides” função para atualizar painel de mensagens
  - Lógica definida nos descendentes (`CounterItem`, `CounterAdd`, `CounterDelete`)
    - Mensagens quando contadores são incrementados, adicionados ou removidos
    - Alteram diretamente o estado dos contadores

Pinia

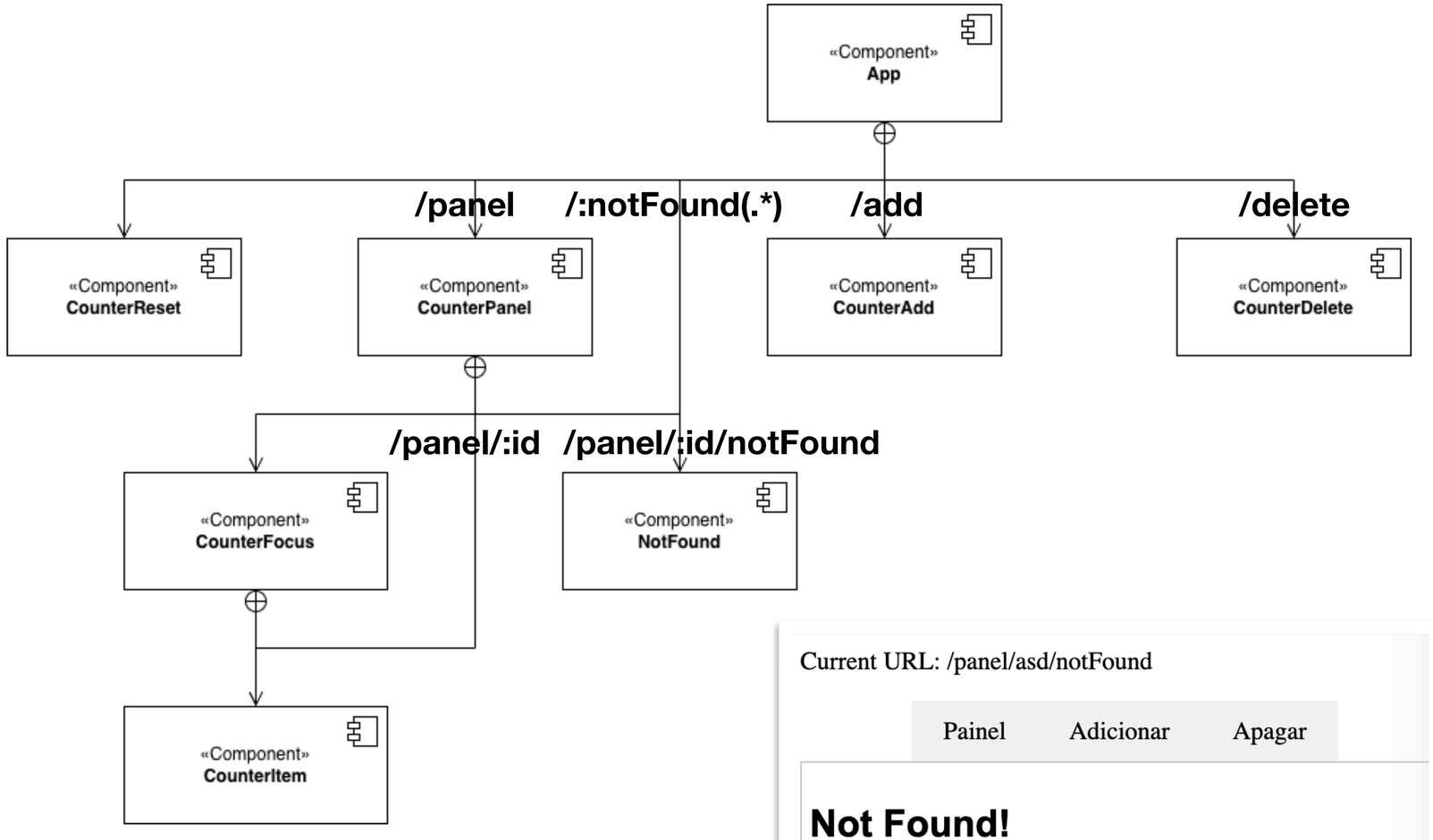
- Biblioteca oficial para **gestão de estado global** em Vue

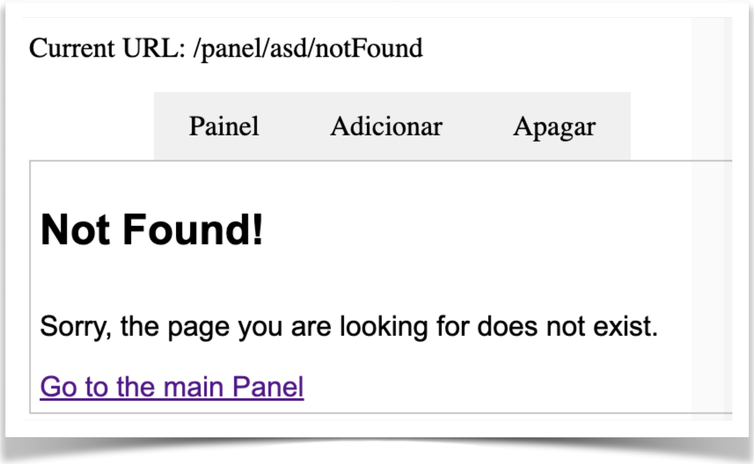
  ```
  npm install pinia
  ```

- Reativa: Gestão centralizada e reativa do estado
- Modular:
  - Permite criar e compor multiplas stores, separadas dos componentes
  - Estruturação de **stores** em
    - **state** (cf. `refs` em components)
    - **getters** sobre o estado (cf. `computed properties` em componentes)
    - **actions** para atualizar o estado (cf. `methods` em componentes)

Exemplo (Counters + NotFound)

- Utilização de routes
  - Definidas no `main.js`
- ! Um problema
  - ! Routes que dependem do estado exigem estado global
  - 💡 Solução ad-hoc
    - 💡 Definir estado Vue em ficheiro JS global: `state/counter.js`





Setup

- Tal como o router, as stores Pinia são definidas fora dos componentes
- No ficheiro `main.js`
  - Acrescentar uma nova dependência

```
import { createPinia } from ‘pinia';
```

- Declarar as stores Pinia o router

```
const pinia = createPinia();
```

- Ao montar a aplicação, acrescentar as stores Pinia

```
app.use(pinia);
```

? Store abstrata? Não dissemos o que contém!

Setup

- Num novo (não obrigatoriamente) ficheiro JS, definir a store

```
import { defineStore } from 'pinia';

import { ref } from 'vue';

export const useStore = defineStore('store', () => {
  const contadores = ref(…);
  const msg = ref(..);
  return {
    contadores,
    msg,
  };
});
```

ID único para o Pinia

Store pode conter múltiplas refs

- Não precisamos de importar a store no `main.js`

Setup

- Utilizar a store em cada componente que necessite

```
import { useStore } from './state/store.js';
const store = useStore();
```

  - Intuição: como se fosse uma variável global
    - Permite ler, e.g.:

```
console.log(store.contadores);
```

    - E escrever, e.g.:

```
store.contadores.push(…);
```

  - Behind the scenes, a store é reativa (ao contrário de variáveis globais JS tradicionais)
  - Não é necessário declarar provide/inject

Exemplo (Counters + NotFound + Pinia)

- Original: Vue router, sem provide/inject
- Plano: Migrar props e custom events para Pinia
- Com Vue router e Pinia
  - Continuamos a usar algumas props
  - Não estamos a utilizar eventos de todo

Pinia

- Boa prática em Pinia é não aceder ao estado diretamente nos componentes
  - Utilizar getters + actions
    - Retirar lógica dos componentes
    - Mais verboso
    - Potencial de code bloating na store
- Podemos declarar múltiplas stores
  - Recomendado que em módulos JS diferentes
  - Uma forma de namespacing
  - Provavelmente desnecessário para a dimensão do projeto prático de IPM

Exemplo (Counters + NotFound + Pinia)

- Podemos passar praticamente toda a lógica dos components para a store
  - Getters: computed properties ou function

```
const grandTotal = computed(()
=> … );

function
counterNameExists(name)
{ return …; }
```

  - Actions: function (potencialmente `async`)

```
function resetCounters() { … }

async function syncDB() { … }
```

- Nesta versão:
  - Não estamos a utilizar props
  - Não estamos a utilizar eventos