Aula 2 — Vue 3 com `<script setup>` (CLI) + Todo List Progressivo

Disciplina de Frameworks Modernos para Desenvolvimento de Sistemas

Sumário

1 Instalação do ambiente (Node.js ou Bun)

Para rodar projetos Vue 3 precisamos de um **runtime JavaScript** atualizado. Você pode escolher entre **Node.js** ou **Bun**.

Opção 1: Node.js

Baixe e instale pelo site oficial: https://nodejs.org/pt (versão LTS recomendada). Após instalar, verifique no terminal:

```
node -v
npm -v
```

Listing 1: Verificando Node.js e npm

Opção 2: Bun

O Bun é um runtime moderno que substitui `node` + `npm` de forma mais rápida. Site oficial: https://bun.sh

Instalação (Linux/Mac):

```
curl -fsSL https://bun.sh/install | bash
```

Listing 2: Instalando o Bun

Após instalar, verifique:

```
bun -v
```

Listing 3: Verificando Bun

2 Criando um projeto Vue 3

O guia oficial recomenda o `create-vue`.

Usando npm

```
npm create vue@latest
```

Usando bun

```
bun create vue@latest
```

Siga o assistente para escolher nome do projeto e recursos (Router, Pinia etc).

Instalar dependências

```
cd todo-list-vue3

# com npm
npm install

# ou com bun (mais r pido)
bun install
```

Rodar o servidor de desenvolvimento

```
# com npm
npm run dev

# com bun
bun run dev
```

A aplicação abrirá em `http://localhost:5173` (porta pode variar).

3 Estrutura de pastas do projeto

Estrutura típica criada pelo `create-vue` (com Vite):

```
todo-list-vue3/
       node_modules/       # depend ncias
       public/             # est ticos
       src/                # c digo-fonte
            assets/          # imagens, estilos
            components/      # componentes Vue
            App.vue          # componente raiz
            main.js          # ponto de entrada
       index.html          # HTML principal
       package.json        # depend ncias e scripts
       vite.config.js      # config do Vite
```

Listing 4: Estrutura de diretórios

Resumo:

- **src/**: onde programamos.
- **App.vue**: componente raiz.
- **main.js**: inicializa o Vue.
- **index.html**: casca HTML (Vue entra no #app).

4 Todo List com Vue 3 + `<script setup>`

Objetivo: evoluir um Todo List único para apresentar os conceitos: `ref`, `computed`, `onMounted`, `v-if`, `v-for`, `v-on`, `v-bind`, `v-model`. **Padrão:** usar **arrow functions** e `<script setup>` (sem `setup()`).

4.1 Bootstrap (main.js)

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Listing 5: src/main.js

4.2 Componente raiz (App.vue)

Explicação curta: Em `<script setup>`, usamos diretamente `ref`, `computed` e `onMounted`. Valores e funções ficam disponíveis no template sem `return`.

Template (estrutura inicial)

```
<template>
  <div class="container">
    <h1>Todo List</h1>

    <div class="input-row">
      <input
        type="text"
        v-model="newTask"
        placeholder="Digite uma tarefa e pressione Enter"
        @keyup.enter="addTask"
        aria-label="Nova tarefa"
      />
      <button @click="addTask" aria-label="Adicionar tarefa">
        Adicionar</button>
```

```
    </div>

    <p v-if="tasks.length === 0">Nenhuma tarefa adicionada.</
       p>

    <ul v-else class="list">
      <li v-for="(t, i) in tasks" :key="t.id" class="item">
        <span
          class="text"
          :style="{
            textDecoration: t.done ? 'line-through' : 'none',
            opacity: t.done ? 0.6 : 1
          }"
          @click="toggle(i)"
          role="button"
        >
          {{ t.done ? '   ' : '   ' }} {{ t.text }}
        </span>
        <button class="small" @click="toggle(i)">{{ t.done ?
           'Desfazer' : 'Concluir' }}</button>
        <button class="small danger" @click="removeTask(i)">
           Remover</button>
      </li>
    </ul>

    <div class="summary" v-if="tasks.length > 0">
      <strong>Total:</strong> {{ tasks.length }} |
      <strong>Conclu das:</strong> {{ doneCount }} |
      <strong>Pendentes:</strong> {{ pendingCount }}
    </div>
  </div>
</template>
```

Listing 6: src/App.vue — Template

Script (<script setup>) — estado, handlers e persistência

```
<script setup>
import { ref, computed, onMounted } from 'vue'

// estado reativo
const newTask = ref('')
const tasks   = ref([]) // { id:number, text:string, done:
   boolean }

// gerador simples de IDs (did tico)
let nextId = 1
const newId = () => nextId++

```

```
// handlers com arrow functions
const addTask = () => {
  const v = newTask.value.trim()
  if (!v) return
  tasks.value.push({ id: newId(), text: v, done: false })
  newTask.value = ''
}

const toggle = (index) => {
  tasks.value[index].done = !tasks.value[index].done
}

const removeTask = (index) => {
  tasks.value.splice(index, 1)
}

// computeds (derivados)
const doneCount    = computed(() => tasks.value.filter(t => t.done).length)
const pendingCount = computed(() => tasks.value.length - doneCount.value)

// ciclo de vida
onMounted(() => {
  const saved = localStorage.getItem('tasks')
  if (saved) {
    const arr = JSON.parse(saved)
    tasks.value = Array.isArray(arr) ? arr : []
    // evita colis o de id ao reabrir
    const maxId = tasks.value.reduce((m, t) => Math.max(m, t.id || 0), 0)
    nextId = maxId + 1
  }
})

// persist ncia autom tica simples (sem watchers avan ados)
const save = () => localStorage.setItem('tasks', JSON.stringify(tasks.value))

// monkey-patch did tico de push/splice para salvar ap s mudan as
const _push = tasks.value.push.bind(tasks.value)
tasks.value.push = (...args) => { const r = _push(...args); save(); return r }

const _splice = tasks.value.splice.bind(tasks.value)
tasks.value.splice = (...args) => { const r = _splice(...args); save(); return r }
```

```
53 </script>
```

Listing 7: src/App.vue — Script com ¡script setup¿

Estilos (opcional)

```
1  <style>
2    .container { max-width: 720px; margin: 2rem auto; padding:
       1rem; }
3    .input-row { display: flex; gap: .5rem; margin-bottom: 1rem
       ; }
4    .list { list-style: none; margin: 0; padding: 0; }
5    .item { display: flex; align-items: center; gap: .5rem;
       padding: .25rem 0; }
6    .text { cursor: pointer; flex: 1; }
7    .small { font-size: .85rem; }
8    .danger { color: #a00; }
9    .summary { margin-top: .75rem; }
10 </style>
```

Listing 8: src/App.vue — Style

5 Conceitos vistos

5.1 Reatividade: `ref`

Resumo: `ref` cria valores reativos. Acesse/mude com `.value`. Usamos `newTask` e `tasks`.

5.2 Diretivas: `v-if` / `v-else`

Resumo: renderização condicional (mensagem de lista vazia vs. lista de tarefas).

5.3 Listas: `v-for` + `:key`

Resumo: `v-for` para iterar, `:key="t.id"` para identidade estável.

5.4 Eventos: `v-on` / `@`

Resumo: `@click`, `@keyup.enter` disparam handlers (arrow functions: `addTask`, `toggle`, `removeTask`).

5.5 Atributos/estilos: `v-bind / :`

Resumo: `:style` reativo para riscar concluidas e alterar opacidade.

5.6 Formulários: `v-model`

Resumo: binding bidirecional no `input` de nova tarefa.

5.7 Derivados: `computed`

Resumo: `doneCount` e `pendingCount` calculados reativamente.

5.8 Ciclo de vida: `onMounted`

Resumo: carregar do `localStorage` quando o componente montar.

6 Ciclo de Vida dos Componentes no Vue 3

No Vue 3, cada componente passa por uma série de etapas chamadas **ciclo de vida**. O ciclo de vida define quando e em que ordem o Vue executa determinadas funções ou disponibiliza recursos [1].

Visão geral

O fluxo básico é:

1. Inicialização do componente (`setup`, reatividade, props).
2. Montagem do componente no DOM.
3. Atualizações quando dados reativos mudam.
4. Desmontagem (remoção) do componente.

Principais etapas e hooks

- **setup() (Composition API)** — executado antes da criação do componente, define estado reativo e funções.
- **onBeforeMount** — chamado antes do Vue inserir o componente no DOM.
- **onMounted** — chamado após o componente ser inserido no DOM. Ideal para chamadas a APIs ou manipulação direta de elementos.

- **onBeforeUpdate** — executado antes de o DOM ser atualizado devido a mudanças reativas.
- **onUpdated** — executado após o DOM ser atualizado.
- **onBeforeUnmount** — chamado antes do componente ser removido.
- **onUnmounted** — chamado após o componente ter sido removido do DOM.

Exemplo prático

```
<script setup>
import { ref, onMounted, onBeforeUnmount } from 'vue'

const count = ref(0)

onMounted(() => {
  console.log('Componente montado no DOM')
})

onBeforeUnmount(() => {
  console.log('Componente prestes a ser desmontado')
})
</script>
```

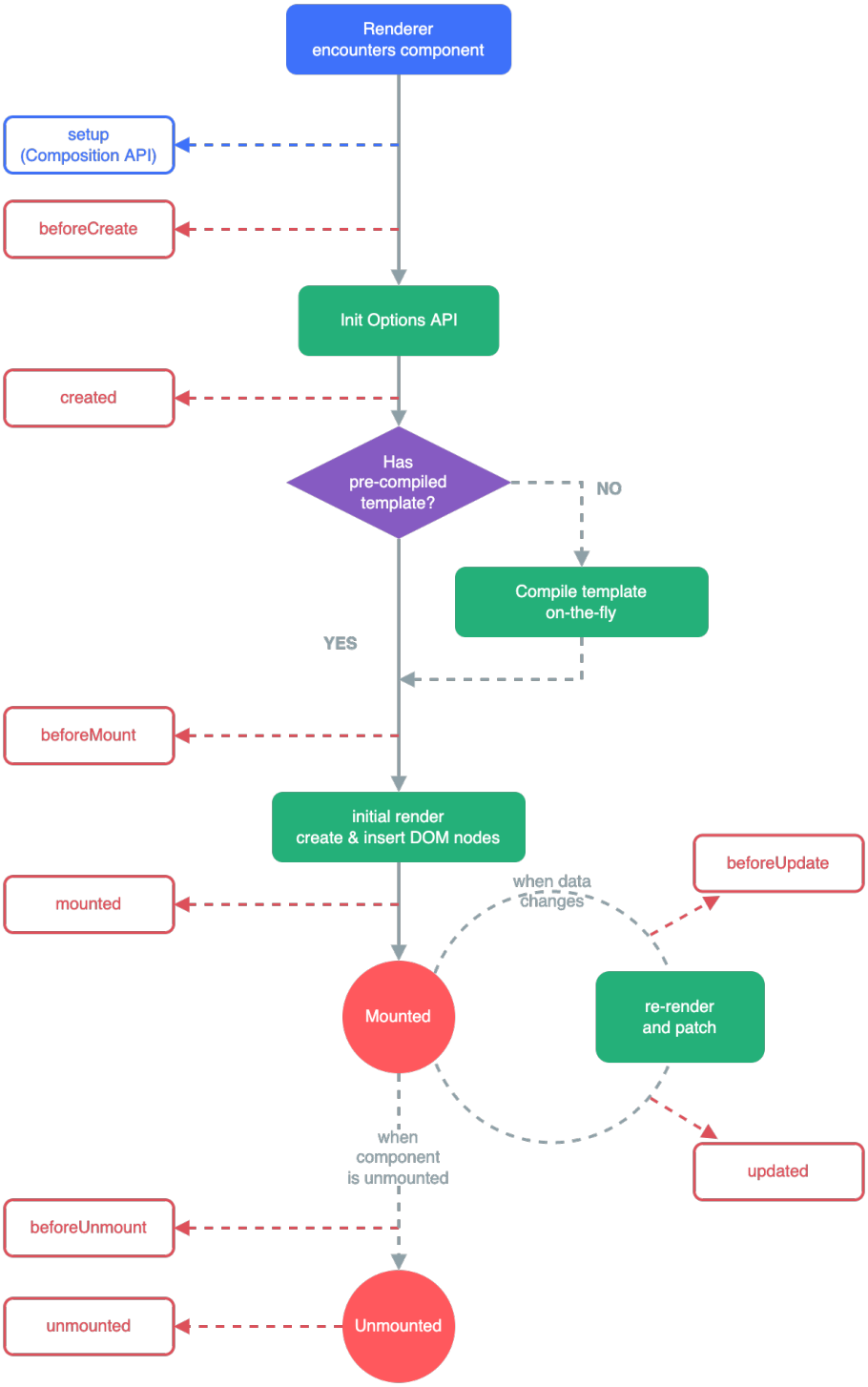
Listing 9: Uso dos hooks de ciclo de vida

Resumo

Esses hooks permitem que você **reaja a momentos específicos do ciclo de vida**, como inicialização, montagem, atualização e desmontagem. Na **Composition API**, usamos funções como `onMounted` e `onUnmounted`, enquanto na Options API existiam os métodos `mounted`, `beforeUnmount`, etc.

Apêndice A — Ciclo de Vida no Vue 3

A imagem a seguir (retirada da documentação oficial) resume graficamente o ciclo de vida:


Renderer
encounters component
setup
(Composition API)
beforeCreate
Init Options API
created
Has
pre-compiled
template?
NO
Compile template
on-the-fly
YES
beforeMount
initial render
create & insert DOM nodes
mounted
beforeUpdate
when data
changes
Mounted
re-render
and patch
updated
when
component
is unmounted
beforeUnmount
unmounted
Unmounted

Apêndice B - Desafio da Aula

Implemente no seu Todo List:

1. **Filtros reativos**: botões `Todos`, `Pendentes`, `Concluídos` usando `computed` para a lista filtrada.
2. **Edição inline**: clique no texto para editar; `@keyup.enter` confirma, `@keyup.esc` cancela; não perca o foco de teclado.
3. **Acessibilidade**: use `aria-label`, `role` adequado e navegação por teclado.
4. **Persistência robusta**: extraia `load/save` para util próprio e garanta integridade ao editar/remover.

Referências

[1] Vue.js. *Lifecycle Hooks*. Disponível em: `https://vuejs.org/guide/essentials/lifecycle.html`. Acesso em: ago. 2025.