INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação

ENSINO MÉDIO INTEGRADO: INFORMÁTICA
Disciplina de Desenvolvimento Web
Aula 11: Overview - Angular | React | Svelte | Vue

Gil Eduardo de Andrade

	PLAYGROUND LINK
Angular	https://angular.dev/playground
React	https://reactplayground.vercel.app/
Svelte	https://svelte.dev/playground/
Vue	https://play.vuejs.org/

Conceitos Preliminares

Introdução

Este material tem como objetivo apresentar os principais conceitos e características dos quatro principais frameworks JavaScript voltados ao desenvolvimento web frontend: Angular, React, Svelte e Vue JS. Ambos são amplamente utilizados para a criação de interfaces de usuário (UI) e Single Page Applications (SPA), porém possuem abordagens e filosofias distintas.

---

1. Framework Angular (https://angular.dev/tutorials/learn-angular)

Angular é uma plataforma e framework de desenvolvimento web baseado em TypeScript, desenvolvido e mantido pelo Google. Ao contrário do Vue, que é considerado um framework progressivo, o Angular é opinativo (impõe uma estrutura rígida), mostrando-se uma solução completa para a construção de Single Page Applications (SPAs) do lado do cliente. Ele oferece uma estrutura robusta e padronizada que facilita o desenvolvimento de aplicações escaláveis em nível empresarial.

A principal diferença arquitetônica do Angular é sua dependência do TypeScript, uma linguagem que adiciona tipagem estática ao JavaScript. Isso proporciona ferramentas de desenvolvimento mais avançadas, como autocompletar inteligente e verificação de erros em tempo de compilação.

Principais Conceitos

A arquitetura do Angular é baseada em componentes organizados em módulos hierárquicos, suportados por um sistema de injeção de dependência e serviços.

Componentes

Os componentes são os blocos de construção fundamentais de qualquer aplicação Angular. Cada componente define uma classe que contém a lógica da aplicação e os dados associados a uma visualização HTML, que define o que será renderizado na tela. A relação entre a classe e o template HTML é estabelecida através de decoradores. Os componentes controlam uma parte da tela ("view") e consistem em:

- *Um Template (HTML).*
- *Uma Classe (TypeScript) para a lógica.*
- *Estilos (CSS).*

TypeScript - Componente / Exemplo A

```typescript
import { Component } from '@angular/core';
@Component({
      selector: 'app-hello',
      template: `<h1>Olá, {{ nome }}!</h1>`
})

export class HelloComponent {
      nome = 'Angular';
}
```

Decoradores

Os decoradores no Angular são funções que modificam as classes TypeScript, adicionando metadados essenciais para que o framework entenda como processá-las. O decorador *@Component*, por exemplo, informa ao Angular que a classe abaixo dele é um componente e fornece a configuração necessária, como o seletor HTML e o template correspondente.

DECORADOR	FINALIDADE	EXEMPLO DE USO
@Component	Define uma classe como um componente Angular e fornece seus metadados.	`@Component({ selector: 'app-root' })`
@Injectable	Marca uma classe como disponível para injeção de dependência.	`@Injectable({ providedIn: 'root' })`
@Input	Permite que dados sejam	`@Input() titulo: string;`

	passados de um componente pai para um filho.	
@Output	Permite que um componente filho emita eventos para um componente pai.	`@Output() evento = new EventEmitter();`

Data Binding

O Angular oferece mecanismos poderosos para sincronizar os dados entre a classe do componente e seu template HTML. O data binding no Angular pode ser unidirecional ou bidirecional.

- *Interpolação*: permite incorporar valores de propriedades no template HTML usando chaves duplas {{ }} .
- *Property Binding*: permite definir o valor de uma propriedade de um elemento HTML usando colchetes [ ].
- *Event Binding*: permite escutar eventos do DOM usando parênteses ( ).
- *Two-way Binding*: sincroniza dados bidirecionalmente usando a diretiva [(ngModel)].

HTML

```HTML
<!-- Exemplo de Data Binding no Angular -->

<!-- Interpolação -->
<p>{{ mensagem }}</p>

<!-- Property Binding -->
<img [src]="urlImagem">

<!-- Event Binding -->
<button (click)="salvar()">Salvar</button>

<!-- Two-way Binding -->
<input [(ngModel)]="nomeUsuario">

<!-- O uso dos colchetes [] indica que o dado vai do código para a tela, e os parênteses () indicam que o dado vai da tela para o código. Juntos, eles mantêm os dois lados sincronizados. -->
```

Diretivas
As diretivas são instruções que alteram a aparência ou o comportamento do DOM.

- **Estruturais (*ngIf, *ngFor)**: alteram o layout adicionando ou removendo elementos.
- **De Atributo (ngClass, ngStyle, ngModel)**: alteram a aparência de elementos existentes.

*TypeScript - Diretivas Estruturais (*ngIf)*

```TypeScript
<p *ngIf="isLoggedIn">Bem-vindo de volta, usuário!</p>

<div *ngIf="tasks.length > 0; else noTasks">
  Você tem tarefas pendentes.
</div>

<ng-template #noTasks>
  <p>Parabéns! Sua lista está vazia.</p>
</ng-template>

/* O Angular exige que o conteúdo do "else" esteja dentro de uma tag <ng-template>. Essa tag, por padrão, fica escondida e só é "renderizada" (desenhada na tela) se o *ngIf mandar. */
```

*TypeScript - Diretivas Estruturais (*ngFor)*

```TypeScript
<ul>
  <li *ngFor="let produto of produtos; let i = index">
    {{ i + 1 }} - {{ produto.nome }} (R$ {{ produto.preco }})
  </li>
</ul>
```

*TypeScript - Diretivas de Atributo (*ngClass)*

```TypeScript
<div [ngClass]="{ 'text-success': isValid, 'text-danger': !isValid }">
  Status da Validação
</div>
```

*TypeScript - Diretivas de Atributo (*ngStyle)*

```typescript
<div [ngStyle]="{
  'color': statusCor,
  'font-size.px': tamanhoFonte
}">
  Este texto muda de cor e tamanho.
</div>
```

*TypeScript - Diretivas de Atributo (*ngModel)*

```typescript
<input [(ngModel)]="nome" placeholder="Digite seu nome">
<p>Olá, {{ nome }}!</p>
```

OBS.: coração do Two-way Data Binding (vinculação de dados bidirecional) no Angular. Ele garante que, se o usuário digitar algo no input, a variável no TypeScript seja atualizada e, se o TypeScript alterar a variável, o input mude automaticamente na tela.

Injeção de Dependência (Dependency Injection)
A Injeção de Dependência (DI) é um padrão de design central no Angular. Em vez de instanciar diretamente as classes que precisam utilizar (como serviços para chamadas HTTP), os componentes declaram essas dependências em seus construtores. O framework Angular se encarrega de criar e fornecer as instâncias corretas dessas dependências quando o componente é instanciado.

TypeScript - Injeção de Dependência

```typescript
import { Injectable, Component } from '@angular/core';

// Serviço injetável
@Injectable({
      providedIn: 'root' // Torna o serviço disponível em toda a aplicação
})

export class LoggerService {
      log(mensagem: string) {
            console.log(mensagem);
      }
```

```
}

export class LoggerComponent {
      // O Angular injeta automaticamente o LoggerService
      constructor(private logger: LoggerService) {}
            registrarLog() {
            this.logger.log('Botão clicado!');
      }
}
```

Serviços (Services)
Os serviços no Angular são classes TypeScript utilizadas para encapsular lógica de negócios, acesso a dados (como requisições HTTP) ou funcionalidades que precisam ser compartilhadas entre múltiplos componentes. Eles promovem a separação de responsabilidades, mantendo os componentes focados apenas na lógica de apresentação e interação do usuário.

TypeScript - Service

```TypeScript
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root' // Torna o serviço disponível em toda a aplicação
})
export class TaskService {

  private tasks: string[] = [];

  addTask(task: Task) { }

  removeTask(id: string) { }
}
```

Sinais (Signals)
O Angular Signals é um sistema que rastreia detalhadamente como e onde o estado do seu aplicativo é usado, permitindo que o framework otimize as atualizações de renderização.
Um sinal é um invólucro em torno de um valor que pode notificar os consumidores interessados quando esse valor muda. Os sinais podem conter qualquer valor, desde tipos primitivos simples até estruturas de dados complexas.

Angular Signals - Exemplo Contador

TypeScript

```typescript
import { Component, signal, computed } from '@angular/core';
import {bootstrapApplication} from '@angular/platform-browser';

@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h1>Contador com Signals</h1>

    <p>Valor atual: <strong>{{ contador() }}</strong></p>
    <p>O dobro é: <strong>{{ dobro() }}</strong></p>

    <button (click)="incrementar()">Incrementar</button>
    <button (click)="resetar()">Resetar</button>
  `,
})
export class Playground {
  // 1. Criando um sinal com valor inicial 0
  contador = signal(0);

   // 2. Criando um valor computado (se atualiza sozinho quando o
contador muda)
  dobro = computed(() => this.contador() * 2);

  incrementar() {
    // 3. Para atualizar, usamos o método .update()
    this.contador.update(valor => valor + 1);
  }

  resetar() {
    // 4. Para definir um valor fixo, usamos o .set()
    this.contador.set(0);
  }
}

bootstrapApplication(Playground);
```

Angular - Exemplo Completo (Tarefas)

Estrutura de Pastas Adotada

- *src/app/models/*: Interfaces TypeScript.
- *src/app/services/*: Lógica de persistência (LocalStorage).
- *src/app/components/*: Componentes reutilizáveis (Formulário e Tabela).

Definindo os Tipos

Na pasta ***"models"*** vamos criar o arquivo **"*task.model.ts*"**, para definir o tipo tarefa, com os seguinte código:

TypeScript

```typescript
// src/app/models/task.model.ts

export interface Task {
  id: string;
  name: string;
  type: string;
  date: string;
}
```

Construindo a Lógica da Aplicação (Regras de Negócio)

Na pasta ***"services"*** vamos criar os arquivos **"*storage.service.ts*"** e **"*task.service.ts*"**, ler e escrever no LocalStorage. Utilizaremos o conceito de Signals, nova forma reativa do Angular para gerenciar estado de forma eficiente. Neste arquivo teremos o seguinte código:

TypeScript

```typescript
// src/app/services/storage.service.ts

import { Injectable, inject, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

@Injectable({
  providedIn: 'root'
```

```
})
export class StorageService {

  private platformId = inject(PLATFORM_ID);
  private isBrowser = isPlatformBrowser(this.platformId);

  getItem(key: string): string | null {
    if (this.isBrowser) {
      return localStorage.getItem(key);
    }
    return null; // Retorno seguro simulado para o ambiente do Servidor (SSR)
  }

  setItem(key: string, value: string): void {
    if (this.isBrowser) {
      localStorage.setItem(key, value);
    }
  }

  removeItem(key: string): void {
    if (this.isBrowser) {
      localStorage.removeItem(key);
    }
  }
}
```

TypeScript

```
// src/app/services/task.service.ts

import { Injectable, inject, signal, effect } from '@angular/core';
import { Task } from '../models/task.model';
import { StorageService } from './storage.service';

@Injectable({ providedIn: 'root' })
export class TaskService {

  private storage = inject(StorageService);
  // Signal que armazena a lista de atividades
```

```
  private tasksSignal = signal<Task[]>(this.loadFromStorage());

  // Exposição pública do Signal (apenas leitura)
  tasks = this.tasksSignal.asReadonly();

  constructor() {
    // Effect que salva automaticamente no LocalStorage sempre que o Signal mudar
    effect(() => {
      this.storage.setItem('tasks', JSON.stringify(this.tasksSignal()));
    });
  }

  private loadFromStorage(): Task[] {
      const data = this.storage.getItem('tasks');
      return data ? JSON.parse(data) : [];
  }

  addTask(task: Task) {
    this.tasksSignal.update(list => [...list, task]);
  }

  removeTask(id: string) {
    this.tasksSignal.update(list => list.filter(a => a.id !== id));
  }
}
```

Criando os Componentes
Na pasta ***"components"*** vamos criar os arquivos **"*task-form.component.ts*"** e **"*task-table.component.ts*"**, que representam, respectivamente, os componentes de formulário de cadastro de atividades e tabela de atividades cadastradas.

Para o ***componente formulário*** temos o seguinte código:

TypeScript

```
// src/app/components/task-form.component.ts

import { Component, inject } from '@angular/core';
import { NonNullableFormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { TaskService } from '../services/task.service';
```

```
@Component({
  selector: 'app-task-form',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: `
     <form [formGroup]="form" (ngSubmit)="submit()" class="bg-white p-6
rounded-lg shadow-md mb-6 border border-gray-200">
      <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
          <input formControlName="name" placeholder="Nome da atividade"
class="border p-2 rounded w-full" />
             <select formControlName="type" class="border p-2 rounded
w-full">
          <option value="">Selecione o tipo</option>
          <option value="Trabalho">Trabalho</option>
          <option value="Estudo">Estudo</option>
          <option value="Lazer">Lazer</option>
        </select>
           <input type="date" formControlName="date" class="border p-2
rounded w-full" />
      </div>
          <button type="submit" [disabled]="form.invalid" class="mt-4
bg-blue-600 text-white px-6 py-2 rounded hover:bg-blue-700
disabled:opacity-50">
        Cadastrar Atividade
      </button>
    </form>
  `
})
export class TaskFormComponent {

  private fb = inject(NonNullableFormBuilder);
  private taskService = inject(TaskService);

  form = this.fb.group({
    name: ['', [Validators.required]],
    type: ['', [Validators.required]],
    date: ['', [Validators.required]]
  });
```

```
  submit() {
    if (this.form.valid) {
      this.taskService.addTask({
        ...this.form.getRawValue(),
        id: crypto.randomUUID()
      });
      this.form.reset();
    }
  }
}
```

Para o ***componente tabela*** temos o seguinte código:

TypeScript

```
// src/app/components/task-table.component.ts

import { Component, inject } from '@angular/core';
import { TaskService } from '../services/task.service';

@Component({
  selector: 'app-task-table',
  standalone: true,
  template: `
    <div class="overflow-x-auto bg-white rounded-lg shadow-md border border-gray-200">
      <table class="w-full text-left border-collapse">
        <thead class="bg-gray-50">
          <tr>
            <th class="p-4 border-b">Nome</th>
            <th class="p-4 border-b">Tipo</th>
            <th class="p-4 border-b">Data</th>
            <th class="p-4 border-b">Ações</th>
          </tr>
        </thead>
        <tbody>
          @for (item of tasks(); track item.id) {
            <tr class="hover:bg-gray-50">
              <td class="p-4 border-b">{{ item.name }}</td>
```

```
            <td class="p-4 border-b">
              <span class="px-2 py-1 rounded text-xs bg-gray-200">
                {{ item.type }}
              </span>
            </td>
            <td class="p-4 border-b">{{ item.date }}</td>
            <td class="p-4 border-b">
              <button (click)="remove(item.id)" class="text-red-500 hover:underline">
                Excluir
              </button>
            </td>
          </tr>
        } @empty {
          <tr>
            <td colspan="4" class="p-8 text-center text-gray-500">
               Nenhuma atividade cadastrada.
            </td>
          </tr>
        }
      </tbody>
    </table>
  </div>
  `
})

export class TaskTableComponent {

  private taskService = inject(TaskService);
  tasks = this.taskService.tasks; // Referência direta ao Signal

  remove(id: string) {
    this.taskService.removeTask(id);
  }
}
```

Codificando a Página/Componente Principal (app.ts)
No arquivo ***"src/app/app.ts"*** teremos o seguinte código:

TypeScript

```typescript
// src/app/app.ts

import { Component } from '@angular/core';
import { TaskFormComponent } from '../components/task-form.component';
import { TaskTableComponent } from '../components/task-table.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [TaskFormComponent, TaskTableComponent],
  template: `
    <main class="min-h-screen bg-gray-100 p-8">
      <div class="max-w-4xl mx-auto">
        <h1 class="text-3xl font-bold text-gray-800 mb-8">
          Gerenciador de Atividades
        </h1>
        <app-task-form />
        <app-task-table />
      </div>
    </main>
  `
})
export class App {}
```

---

2. Biblioteca React (https://react.dev/learn)

O React é, tecnicamente, uma biblioteca para construção de interfaces, mas no ecossistema moderno funciona como um framework. Diferente do Angular e do Vue, que usam "templates" (HTML estendido), o React usa JSX, onde o HTML é escrito diretamente dentro do JavaScript.

Principais Conceitos

O React JS tem por objetivo facilitar a criação de UIs complexas e interativas, tornando o processo de desenvolvimento mais eficiente e escalável, através de um paradigma de componentes. A seguir são apresentados os principais conceitos da biblioteca React.

Componentes Funcionais

No React moderno, componentes são funções JavaScript que retornam elementos visualizáveis (JSX). Eles são a unidade básica de UI.

JSX (JavaScript XML)
É uma extensão de sintaxe que permite escrever algo muito parecido com HTML dentro do código JS. O React transforma isso em chamadas de funções que atualizam o DOM de forma eficiente.

JavaScript - Componente Funcional JSX

```javascript
import React, { useState, useEffect } from 'react';

function Welcome() {
  return (
    <div style={{ padding: '20px', fontFamily: 'Arial' }}>
      <h1>Olá Mundo!</h1>
    </div>
  );
}

export default Welcome;
```

Hooks (Estado e Efeito)
Os Hooks são funções que permitem "conectar" recursos de estado e ciclo de vida:

- ***useState***: gerencia dados que, ao mudarem, fazem o componente renderizar novamente.

- ***useEffect***: manipula com "efeitos colaterais", como chamadas a APIs ou inscrições em eventos.

JavaScript - Hooks: useState e useEffect

```typescript
import React, { useState, useEffect } from 'react';

export default function Counter() {
      // Define o estado "count" e a função que permite
      // modificá-lo "setCount" (Hooks)
      const [count, setCount] = useState(0);
      // Função invocada, como efeito colateral, quando um estado
      // do componente é modificado (Hooks)
```

```
      useEffect(() => {
            document.title = `Você clicou ${count} vezes`;
      });

      return (
            <div style={{ marginTop: '20px'}}>
                  <p>Você clicou {count} vezes</p>
                  <button onClick={() => setCount(count + 1)}>
                        Clique para incrementar
                  </button>
            </div>
      )
}
```

Props (Propriedades)
Assim como no Vue, as Props são a forma de passar dados de um componente pai para um componente filho. No React, os dados fluem em uma única direção (unidirecional).

JavaScript - props / Componente Funcional

```
TypeScript
function Welcome(props) {
      return (
            <div style={{ padding: '20px', fontFamily: 'Arial' }}>
                  <h1>Olá {props.name}!</h1>
            </div>
      )
}
export default Welcome;
```

React - Exemplo Completo (Tarefas)

Estrutura de Pastas Adotada

- *src/models/*: Interfaces TypeScript.
- *src/contexts/*: Onde reside o Estado Global.
- *src/lib/components/*: Componentes Visuais (Formulário e Tabela).

Definindo os Modelos de Dados (Tipos)
Na pasta ***"models"*** vamos criar os arquivos **"*task.model.ts*"** e ***"taskContextType.model.ts"***, para definir o tipo tarefa e o tipo do contexto da tarefa, com os seguinte código:

TypeScript
```
// src/models/task.model.ts

export interface Task {
  id: string;
  name: string;
  type: string;
  date: string;
}
```

TypeScript
```
// src/models/taskContextType.model.ts

import type { Task } from './task.model';

export interface TaskContextType {
  tasks: Task[];
  addTask: (task: Task) => void;
  removeTask: (id: string) => void;
}
```

Construindo a Lógica da Aplicação (Regras de Negócio)
No React, para evitar passar dados via ***"props"*** por todos os níveis, usamos o Contexto. Ele gerencia o estado e a persistência de forma centralizada. Na pasta ***"contexts"*** vamos criar o arquivo **"*TaskContext.tsx*"**, nele teremos o seguinte código:

TypeScript
```
// src/contexts/TaskContext.tsx

import React, { createContext, useContext, useState, useEffect } from 'react';
import type { Task } from '../models/task.model';
import type { TaskContextType } from '../models/taskContextType.model';
```

```tsx
const TaskContext = createContext<TaskContextType | undefined>(undefined);

export function TaskProvider({ children }: { children: React.ReactNode }) {
  // Inicialização preguiçosa (Lazy Initializer) para ler o LocalStorage apenas uma vez
  const [tasks, setTask] = useState<Task[]>(() => {
    const saved = localStorage.getItem('tasks');
    return saved ? JSON.parse(saved) : [];
  });

  // Efeito para persistir no LocalStorage sempre que o estado mudar
  useEffect(() => {
    localStorage.setItem('tasks', JSON.stringify(tasks));
  }, [tasks]);

  const addTask = (task: Task) => {
    setTask(prev => [...prev, task]);
  };

  const removeTask = (id: string) => {
    setTask(prev => prev.filter(a => a.id !== id));
  };

  return (
    <TaskContext.Provider value={{ tasks, addTask, removeTask }}>
      {children}
    </TaskContext.Provider>
  );
}

// Hook personalizado para facilitar o uso do contexto
export const useTask = () => {
  const context = useContext(TaskContext);
  if (!context) throw new Error('useTask deve ser usado dentro de um TaskProvider');
  return context;
};
```

Criando os Componentes

No React usamos "componentes controlados" onde o ***value*** e o ***onChange*** mantêm o estado sincronizado. Na pasta ***"components"*** vamos criar os arquivos ***"TaskForm.tsx"*** e ***"TaskTable.tsx"***, que representam, respectivamente, os

componentes de formulário de cadastro de atividades e tabela de atividades cadastradas.

Para o ***componente formulário***, temos o seguinte código:

TypeScript

```
// src/components/TaskForm.tsx

import React, { useState } from 'react';
import { useTask } from '../contexts/TaskContext';

export function TaskForm() {
  const { addTask } = useTask();
  const [formData, setFormData] = useState({ name: '', type: '', date: '' });

  const handleSubmit = (e: React.SubmitEvent) => {
    // impedir que o navegador execute o comportamento padrão, recarregar a página
    e.preventDefault();
    if (formData.name && formData.type && formData.date) {
      addTask({ ...formData, id: crypto.randomUUID() });
      setFormData({ name: '', type: '', date: '' }); // Reset
    }
  };

  return (
      <form onSubmit={handleSubmit} className="bg-white p-6 rounded-lg
shadow-md mb-6 border border-gray-200">
      <div className="grid grid-cols-1 md:grid-cols-3 gap-4">
        <input
          value={formData.name}
                    onChange={e => setFormData({...formData, name:
e.target.value})}
          placeholder="Nome da atividade"
          className="border p-2 rounded w-full" required
        />
        <select
          value={formData.type}
                    onChange={e => setFormData({...formData, type:
e.target.value})}
          className="border p-2 rounded w-full" required
        >
```

```
          <option value="">Selecione o tipo</option>
          <option value="Trabalho">Trabalho</option>
          <option value="Estudo">Estudo</option>
          <option value="Lazer">Lazer</option>
        </select>
        <input
          type="date"
          value={formData.date}
          onChange={e => setFormData({...formData, date: e.target.value})}
          className="border p-2 rounded w-full" required
        />
      </div>
       <button type="submit" className="mt-4 bg-cyan-600 text-white px-6 py-2 rounded hover:bg-cyan-700">
         Cadastrar Atividade
      </button>
    </form>
  );
}
```

Para o ***componente tabela***, temos o seguinte código:

TypeScript

```
// src/components/TaskTable.tsx

import type { Task } from '../models/task.model';
import { useTask } from '../contexts/TaskContext';

export function TaskTable() {

  const { tasks, removeTask } = useTask();

  return (
       <div  className="overflow-x-auto  bg-white  rounded-lg  shadow-md border border-gray-200">
      <table className="w-full text-left border-collapse">
        <thead className="bg-gray-50">
          <tr>
```

```
            <th className="p-4 border-b">Nome</th>
            <th className="p-4 border-b">Tipo</th>
            <th className="p-4 border-b">Data</th>
            <th className="p-4 border-b">Ações</th>
          </tr>
        </thead>
        <tbody>
          {tasks.length > 0 ? (
            tasks.map((item: Task) => (
              <tr key={item.id} className="hover:bg-gray-50">
                <td className="p-4 border-b">{item.name}</td>
                <td className="p-4 border-b">
                          <span  className="px-2  py-1  rounded  text-xs
bg-cyan-100 text-cyan-800">{item.type}</span>
                </td>
                <td className="p-4 border-b">{item.date}</td>
                <td className="p-4 border-b">
                          <button  onClick={()  =>  removeTask(item.id)}
className="text-red-500 hover:underline">
                    Excluir
                  </button>
                </td>
              </tr>
            ))
          ) : (
            <tr>
                          <td  colSpan={4}  className="p-8  text-center
text-gray-500">Nenhuma atividade cadastrada.</td>
            </tr>
          )}
        </tbody>
      </table>
    </div>
  );
}
```

Codificando a Página Principal (App.tsx)
No React, precisamos envolver a aplicação no nosso ***Provider*** para que o contexto funcione. Sendo assim, no arquivo ***"src/App.tsx"*** teremos o seguinte código:

TypeScript

```tsx
// src/App.tsx

import './App.css'
import { TaskProvider } from './contexts/TaskContext';
import { TaskForm } from './components/TaskForm';
import { TaskTable } from './components/TaskTable';

function App() {

  return (
    <TaskProvider>
      <main className="min-h-screen bg-slate-100 p-8">
        <div className="max-w-4xl mx-auto">
          <header className="mb-8">
            <h1 className="text-3xl font-bold text-slate-800">
               Gerenciador de Tarefas
            </h1>
          </header>

          <TaskForm />
          <TaskTable />
        </div>
      </main>
    </TaskProvider>
  );
}

export default App
```

---

3. Framework Svelte (https://svelte.dev/)

O Svelte é um framework que possui uma abordagem simplista para construção de interfaces de usuário. Ao invés de executar a maior parte do trabalho no navegador, como outros frameworks, o Svelte compila seu código em JavaScript simples e otimizado, durante o processo de build. Tal comportamento possibilita a construção

de aplicativos mais rápidos e eficientes, que apresentam um melhor desempenho e um menor tamanho de bundle.

Principais Conceitos

A seguir são apresentados os principais conceitos do Framework Svelte.

Reatividade

A reatividade é um dos pilares do Svelte, permitindo que você crie interfaces de usuário dinâmicas e responsivas.

Declarações reativas

Você pode usar a diretiva "**$:**" para criar declarações reativas que são automaticamente atualizadas quando seus valores dependentes mudam.

JavaScript - Reatividade (Declarações Reativas)

```
JavaScript

<script>

  let count = 0;

  // let dobro = count * 2;

  $: dobro = count * 2;

</script>

<button on:click={() => count++}>Contador: {count}</button>

<p>Dobro: {dobro}</p>
```

JavaScript - Reatividade (Declarações Reativas) - Utilizando $state

```
JavaScript

<script>

      // Criamos o estado reativo

      let novoItem = $state(0);

      let lista = $state([]);

      // Função para adicionar
```

```
      function adicionar() {
            if (novoItem != 0) {
                  // No Svelte 5, você pode dar um .push() direto!
                  lista.push(novoItem);
                  novoItem = 0; // Limpa o input
            }
      }
</script>

<h1>Valor: {novoItem}</h1>

<input type="number" bind:value={novoItem} />

<button onclick={adicionar}>Adicionar</button>

<p>Total de itens: {lista.length}</p>

{#each lista as item}
      <li>{item}</li>
{/each}
```

Atualizações automáticas

O Svelte atualiza automaticamente o DOM quando os valores das variáveis reagem às mudanças. A diretiva "***bind***" é usada como um atributo em elementos HTML, seguido de ***":"***. Por exemplo, ***"bind:value={myVariable}"*** vincula o valor de ***"myVariable"*** ao atributo value de um elemento HTML.

JavaScript - Atualizações Automáticas

JavaScript

```
<script>
  let nome = "Pequeno Gafanhoto";
</script>

<h1>Olá, {nome}!</h1>
```

```
<input type="text" bind:value={nome} />
```

Componentes

Os componentes são blocos de construção reutilizáveis que permitem que você organize seu código de forma modular e eficiente.

JavaScript - Componentes (MeuComponente)

JavaScript

```
<script>
  // Propriedade "nome" com valor padrão
  export let nome = "Visitante";
</script>

<main>
  <div>
    <h2>Olá, {nome}!</h2>
    <p>Este é um componente reutilizável.</p>
  </div>
</main>

<style>
  div {
    border: 1px solid gray;
    padding: 10px;
  }
</style>
```

JavaScript - App (Utiliza MeuComponente)

JavaScript

```
<script>
  import MeuComponente from './Component.svelte';
</script>

<main>
  <h1>Página inicial</h1>
  <MeuComponente />
  <MeuComponente nome="João" />
```

```
</main>
```

Diretivas

As diretivas do Svelte são recursos de código que permitem adicionar comportamentos especiais aos seus elementos HTML, como controlar a renderização, lidar com eventos, criar ligações de dados, entre outros.
Tais diretivas são divididas em algumas categorias, cada qual com sua função específica:

Diretivas de Renderização Condicional

- *{#if condition} {/if}*: bloco de código renderizado quando a condição for verdadeira.
- *{#else if condition} {/else if}*: condição alternativa para o if.
- *{#else} {/else}*: bloco de código renderizado quando nenhuma condição anterior é satisfeita.

Diretivas de Repetição

- *{#each array as item}{/each}*: efetua a iteração sobre um array e renderiza um bloco de código para cada item.

Diretivas de Eventos

- *on:eventname:* executa uma função JavaScript quando um evento específico ocorre (ex: on:click, on:mouseover).

Diretivas de Ligação de Dados

- *bind:property:* cria uma ligação bidirecional entre uma propriedade do componente e um elemento HTML. Exemplo: *bind:value*, *bind:checked*.

JavaScript - Diretivas

(Renderização Condicional / Repetição / Eventos / Ligação de Dados)

JavaScript

```
<script>
  let name = "";
  let items = ["Carlos", "Daniela", "Eduardo"];
  let showMessage = false;

  function handleClick() {
    if(name != "") items.push(name);
    name = ""
```

```
    showMessage = !showMessage;
  }
</script>

<main>
  <input type="text" bind:value={name} placeholder="Digite seu nome">
  {#if name}
    <h1>Olá, {name}!</h1>
  {:else}
    <h1>Olá, Mundo!</h1>
  {/if}

  <button on:click={handleClick}>Clique aqui</button>
      <br><br>
  {#if showMessage}
            {#each items as item}
      <li>{item}</li>
    {/each}
  {/if}
</main>
```

Svelte - Exemplo Completo (Tarefas)

Estrutura de Pastas Adotada

- *src/lib/models/*: Interfaces TypeScript.
- *src/lib/state/*: Efetua o Gerenciamento de Estados.
- *src/lib/components/*: Componentes Svelte (Formulário e Tabela).

Definindo os Modelos de Dados (Tipos)

Na pasta ***"models"*** vamos criar o arquivo ***"task.model.ts"***, para definir o tipo tarefa, com os seguinte código:

TypeScript

```
// src/lib/models/task.model.ts

export interface Task {
  id: string;
```

```
  name: string;
  type: string;
  date: string;
}
```

Efetuando o Gerenciamento de Estados (Store)

No Svelte 5, usamos a rune $state para criar reatividade e $effect para obter persistência. Dentro da pasta ***"src/lib/states"*** vamos criar o arquivo ***"task.svelte.ts"***, contendo o seguinte código-fonte:

TypeScript

```
// src/lib/state/task.svelte.ts

import type { Task } from '../models/task.model';

// Criamos uma classe para encapsular o estado reativo
class TaskService {
  // A rune $state torna o array reativo
  items = $state<Task[]>([]);

  // Encapsulamos a lógica de persistência aqui
  init() {
    const saved = localStorage.getItem('tasks');
    if (saved) this.items = JSON.parse(saved);

    // $effect roda sempre que 'items' mudar
    $effect(() => {
      localStorage.setItem('tasks', JSON.stringify(this.items));
    });
  }

  add(task: Task) {
    this.items.push(task);
  }

  remove(id: string) {
    this.items = this.items.filter(i => i.id !== id);
  }
```

```
}

// Exportamos uma única instância (Singleton)
export const taskService = new TaskService();
```

Criando os Componentes

O Svelte usa ***bind:value*** para sincronizar os inputs com as variáveis. Na pasta ***"components"*** vamos criar os arquivos ***"TaskForm.svelte"*** e ***"TaskTable.svelte"***, que representam, respectivamente, os componentes de formulário de cadastro de atividades e tabela de atividades cadastradas.

Para o ***componente formulário***, temos o seguinte código:

TypeScript

```
// src/lib/components/TaskForm.svelte

<script lang="ts">
  import { taskService } from '../states/task.svelte';

  let name = $state('');
  let type = $state('');
  let date = $state('');

  function handleSubmit(event: Event) {
    event.preventDefault();
    if (name && type && date) {
      taskService.add({
        id: crypto.randomUUID(),
        name,
        type,
        date
      });
      // Resetar campos
      name = ''; type = ''; date = '';
    }
  }
</script>
```

```
<form onsubmit={handleSubmit} class="bg-white p-6 rounded-lg shadow-md mb-6 border border-gray-200">
  <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
          <input  bind:value={name}  placeholder="Nome  da  atividade" class="border p-2 rounded w-full" required />

        <select  bind:value={type}  class="border  p-2  rounded  w-full" required>
      <option value="" disabled selected>Selecione o tipo</option>
      <option value="Trabalho">Trabalho</option>
      <option value="Estudo">Estudo</option>
      <option value="Lazer">Lazer</option>
    </select>

       <input  type="date"  bind:value={date}  class="border  p-2  rounded w-full" required />
  </div>

   <button type="submit" class="mt-4 bg-orange-600 text-white px-6 py-2 rounded hover:bg-orange-700">
    Cadastrar Atividade
  </button>
</form>
```

Para o ***componente tabela***, temos o seguinte código:

TypeScript

```
// src/lib/components/TaskTable.svelte

<script lang="ts">
  import { taskService } from '../states/task.svelte';
</script>

<div  class="overflow-x-auto  bg-white  rounded-lg  shadow-md  border border-gray-200">
  <table class="w-full text-left border-collapse">
    <thead class="bg-gray-50">
```

```
      <tr>
        <th class="p-4 border-b">Nome</th>
        <th class="p-4 border-b">Tipo</th>
        <th class="p-4 border-b">Data</th>
        <th class="p-4 border-b">Ações</th>
      </tr>
    </thead>
    <tbody>
      {#each taskService.items as item (item.id)}
        <tr class="hover:bg-gray-50">
          <td class="p-4 border-b">{item.name}</td>
          <td class="p-4 border-b">
                  <span class="px-2 py-1 rounded text-xs bg-orange-100
text-orange-800">{item.type}</span>
          </td>
          <td class="p-4 border-b">{item.date}</td>
          <td class="p-4 border-b">
                   <button onclick={() => taskService.remove(item.id)}
class="text-red-500 hover:underline">
              Excluir
            </button>
          </td>
        </tr>
      {:else}
        <tr>
           <td colspan="4" class="p-8 text-center text-gray-500">Nenhuma
atividade cadastrada.</td>
        </tr>
      {/each}
    </tbody>
  </table>
</div>
```

Utilizando os Componentes na Página Principal
No arquivo ***"src/App.svelte"***, vamos utilizar o seguinte código-fonte:

TypeScript

```
// src/lib/models/task.model.ts
```

4. Framework Vue JS (https://br.vuejs.org/v2/guide/)

O Vue JS, frequentemente chamado apenas de ***Vue***, é um framework JavaScript progressivo de código aberto para a construção de interfaces de usuário. Criado por Evan You em 2014, o ***Vue*** foi projetado desde o início para ser adotável de forma incremental, com foco na reatividade e simplicidade. Isso significa que sua biblioteca principal foca exclusivamente na camada visual (view layer), facilitando a integração com outras bibliotecas ou projetos já existentes.

Diferente de frameworks monolíticos, o ***Vue*** permite que desenvolvedores comecem a utilizá-lo de maneira simples, adicionando apenas uma tag de script em uma página HTML, e gradualmente incorporem ferramentas mais avançadas de seu ecossistema para construir aplicações complexas.

Principais Conceitos

A arquitetura do Vue baseia-se em alguns pilares fundamentais que tornam o desenvolvimento intuitivo e eficiente. A seguir, detalhamos os conceitos mais importantes.

Reatividade

A reatividade é um dos conceitos mais poderosos do Vue. Quando os dados de um componente Vue são modificados, a interface do usuário é atualizada automaticamente para refletir essas mudanças. O desenvolvedor não precisa manipular o DOM diretamente; basta atualizar o estado dos dados, e o Vue cuida de atualizar a tela de forma otimizada.

Diretivas

Diretivas são atributos especiais com o prefixo ***v-*** que aplicam comportamentos específicos ao DOM renderizado. Elas reagem a mudanças nos dados e aplicam efeitos correspondentes.

DIRETIVA	DESCRIÇÃO	EXEMPLO DE USO
v-bind	Interliga atributos HTML dinamicamente a expressões	`<img v-bind:src="urlImagem">`

	Vue.	

v-if	Renderiza o elemento condicionalmente com base na veracidade da expressão.	`<p v-if="visivel">`
		`  Texto condicional`
		`</p>`

v-for	Renderiza uma lista de itens iterando sobre um array ou objeto.	`<li v-for="item in lista">`
		`  {{ item.nome }}`
		`</li>`

v-on	Anexa escutas de eventos aos elementos do DOM.	`<button v-on:click="saudar">`
		`  Clique`
		`</button>`

v-model	Cria uma interligação de mão dupla (two-way data binding) em elementos de formulário	`<input v-model="nome">`

Tratamento de Eventos e Métodos

Para permitir a interação do usuário, o Vue utiliza a propriedade methods dentro da instância. Os métodos são funções que podem ser chamadas em resposta a eventos do DOM, como cliques ou digitação, utilizando a diretiva v-on.

Eventos e Métodos

JavaScript

```
<script setup>
  import { ref } from 'vue'
  const contador = ref(0)

  function incrementar() {
    contador.value++;
  }
</script>

<template>
  <p>Contagem: {{ contador }}</p>
  <button v-on:click="incrementar">Adicionar</button>
</template>
```

Componentes

O sistema de componentes é uma abstração que permite a construção de aplicações de larga escala compostas por pequenos componentes auto-contidos e reutilizáveis. Um componente Vue é essencialmente uma instância Vue com opções pré-definidas. Eles encapsulam estrutura (HTML), estilo (CSS) e comportamento (JavaScript) em uma única unidade lógica.

Componente - BotaoContador

TypeScript

```
<script setup>
      import { ref } from 'vue'
      // Estado interno do componente
      const cliques = ref(0)
</script>

<template>
  <!-- O template define a estrutura visual -->
  <button @click="cliques++">
    Você me clicou {{ cliques }} vezes
  </button>
</template>

<style scoped>
      /* O "scoped" garante que este estilo afete apenas este componente */
      button {
        background-color: #42b883;
        color: white;
        border: none;
        padding: 10px 20px;
        border-radius: 4px;
        cursor: pointer;
        font-weight: bold;
      }
      button:hover {
        background-color: #35495e;
      }
</style>
```

Arquivo Principal - App.vue

TypeScript

```
<script setup>
// Importa o componente criado acima
import BotaoContador from './Comp.vue'
</script>
```

```
<template>
  <div>
    <h1>Minha Aplicação Vue</h1>
    <!-- Reutilizando o componente de forma independente -->
    <BotaoContador />
    <BotaoContador />
  </div>
</template>
```

Funcionalidades de Reatividade: watch e computed

Propriedades Computadas (Computed) são usadas para transformar dados existentes e retornar um novo valor, sendo re-computadas automaticamente apenas quando suas dependências mudam.

Já os Observadores (Watch) são usados para reagir a alterações específicas de dados e executar "efeitos colaterais", como chamadas assíncronas a APIs, manipulação do DOM ou salvar dados no localStorage.

TypeScript

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

// Estados reativos
const contador = ref(0)
const mensagem = ref('Clique no botão para iniciar')

// Computed: calcula automaticamente o dobro do contador
// Ele se atualiza sozinho sempre que 'contador' mudar
const dobro = computed(() => contador.value * 2)

// Watch: monitora o contador e registra o histórico de mudança
watch(contador, (novoValor, valorAntigo) => {
  mensagem.value = `Mudou de ${valorAntigo} para ${novoValor}`
})
</script>

<template>
```

```
  <div class="contador-container">
    <p>Valor atual: <strong>{{ contador }}</strong></p>
    <p>O dobro é: <strong>{{ dobro }}</strong></p>
    <p class="log">{{ mensagem }}</p>

    <button @click="contador++">Incrementar</button>
  </div>
</template>

<style scoped>
.contador-container {
  padding: 20px;
  font-family: sans-serif;
}
.log {
  color: #666;
  font-style: italic;
}
button {
  padding: 8px 16px;
  cursor: pointer;
}
</style>
```

Vue - Exemplo Completo (Tarefas)

Estrutura de Pastas Adotada

- *src/models/*: Interfaces TypeScript.
- *src/composables/*: Lógica de estados global (Equivale aos Services).
- *src/components/*: Componentes da Interface (Formulário e Tabela).

[1]A lógica de estados global no VueJS é um padrão de gerenciamento de dados que permite armazenar informações importantes da aplicação em um único local centralizado, tornando esses dados acessíveis e modificáveis por qualquer componente, independentemente de quão profunda seja a hierarquia da aplicação.

² Composables são funções JavaScript reutilizáveis que encapsulam lógica de estado e comportamento reativo, permitindo compartilhar funcionalidades entre componentes de forma organizada.

Definindo os Modelos de Dados (Tipos)
Na pasta ***"models"*** vamos criar o arquivo **"*task.model.ts*"**, para definir o tipo tarefa, com os seguinte código:

```typescript
// src/models/task.model.ts

export interface Task {
  id: string;
  name: string;
  type: string;
  date: string;
}
```

Definindo o Gerenciamento de Estados
No Vue, em vez de um Service com classe, usamos o conceito de *Composables*. Dentro da pasta em questão vamos criar o arquivo ***"useTasks.ts"*** com o seguinte código-fonte:

```typescript
// src/composables/useTasks.ts

import { ref, watch, readonly } from 'vue';
import type { Task } from '../models/task.model';

// Estado global fora da função para manter os dados entre componentes
const tasks = ref<Task[]>(JSON.parse(localStorage.getItem('tasks') || '[]'));

export function useTasks() {
  // Salva no LocalStorage sempre que a lista mudar
  watch(tasks, (newList) => {
    localStorage.setItem('activities', JSON.stringify(newList));
  }, { deep: true });
  // Força o watch a monitorar o que acontece dentro de um objeto ou array
```

```
  const addTask = (task: Task) => {
    tasks.value.push(task);
  };

  const removeTask = (id: string) => {
    tasks.value = tasks.value.filter(a => a.id !== id);
  };

  return {
    tasks: readonly(tasks), // Apenas leitura para os componentes
    addTask,
    removeTask
  };
}
```

Criando os Componentes

Na pasta "components" vamos criar os arquivos ***"TaskForm.vue"*** e ***"TaskTable.vue"***, que representam, respectivamente, os componentes de formulário de cadastro de atividades e tabela de atividades cadastradas.

Para o ***componente formulário***, temos o seguinte código:

TypeScript

```
// src/components/TaskForm.vue

<script setup lang="ts">
import { reactive } from 'vue';
import { useTasks } from '../composables/useTasks';

const { addTask } = useTasks();
// reactive: torna todas as 3 propriedades reativas (name,type date)
const form = reactive({
  name: '',
  type: '',
  date: ''
});

const submit = () => {
```

```
  if (form.name && form.type && form.date) {
    addTask({
      ...form,
      id: crypto.randomUUID()
    });
    // Resetar formulário
    form.name = '';
    form.type = '';
    form.date = '';
  }
};
</script>

<template>
     <form  @submit.prevent="submit"  class="bg-white  p-6  rounded-lg
shadow-md mb-6 border border-gray-200">
    <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
          <input  v-model="form.name"  placeholder="Nome  da  atividade"
class="border p-2 rounded w-full" required />

      <select v-model="form.type" class="border p-2 rounded w-full" required>
        <option value="" disabled>Selecione o tipo</option>
        <option value="Trabalho">Trabalho</option>
        <option value="Estudo">Estudo</option>
        <option value="Lazer">Lazer</option>
      </select>

       <input type="date" v-model="form.date" class="border p-2 rounded
w-full" required />
    </div>

      <button  type="submit"  class="mt-4  bg-emerald-600  text-white  px-6
py-2 rounded hover:bg-emerald-700">
      Cadastrar Atividade
    </button>
  </form>
</template>
```

Para o ***componente tabela***, temos o seguinte código:

TypeScript

```
// src/components/TaskTable.vue

<script setup lang="ts">
    import { useTasks } from '../composables/useTasks';
    const { tasks, removeTask } = useTasks();
</script>

<template>
    <div class="overflow-x-auto bg-white rounded-lg shadow-md border
border-gray-200">
    <table class="w-full text-left border-collapse">
      <thead class="bg-gray-50">
        <tr>
          <th class="p-4 border-b">Nome</th>
          <th class="p-4 border-b">Tipo</th>
          <th class="p-4 border-b">Data</th>
          <th class="p-4 border-b">Ações</th>
        </tr>
      </thead>
      <tbody>
        <tr v-for="item in tasks" :key="item.id" class="hover:bg-gray-50">
          <td class="p-4 border-b">{{ item.name }}</td>
          <td class="p-4 border-b">
            <span class="px-2 py-1 rounded text-xs bg-emerald-100 text-emerald-800">
                {{ item.type }}
            </span>
          </td>
          <td class="p-4 border-b">{{ item.date }}</td>
          <td class="p-4 border-b">
            <button @click="removeTask(item.id)" class="text-red-500 hover:underline">
                Excluir
            </button>
          </td>
        </tr>
        <tr v-if="tasks.length === 0">
          <td colspan="4" class="p-8 text-center text-gray-500">
             Nenhuma atividade cadastrada.
          </td>
```

```
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

Utilizando os Componentes na Página Principal
No arquivo ***"src/App.vue"***, vamos utilizar o seguinte código-fonte:

TypeScript

```
// src/App.vue
<script setup lang="ts">
  import TaskForm from './components/TaskForm.vue';
  import TaskTable from './components/TaskTable.vue';
</script>

<template>
  <main class="min-h-screen bg-slate-50 p-8">
    <div class="max-w-4xl mx-auto">
      <header class="mb-8">
            <h1 class="text-3xl font-bold text-slate-800">Vue Activity
Tracker</h1>
          <p class="text-slate-600">Organize suas tarefas com Vue 3 e
Vite</p>
      </header>

      <TaskForm />
      <TaskTable />
    </div>
  </main>
</template>
```

Conceitos Chave Utilizados no Vue:

- *Reatividade com ref e reactive*: o Vue rastreia automaticamente as mudanças. Diferente dos Signals do Angular que exigem uma função (), no Vue você acessa via .value no script e diretamente no template.

- *Composables (useTasks)*: uma forma elegante de compartilhar lógica entre componentes sem a necessidade de um sistema de injeção de dependência complexo.

- *SFC (<script setup>)*: reduz drasticamente o "boilerplate" (seções de código que precisam ser incluídas em muitos lugares com pouca ou nenhuma alteração). Todo código dentro de setup é exposto automaticamente ao template.

- *Diretivas (v-model, v-for, v-if)*: o Vue usa atributos especiais no HTML para lógica de template, o que muitos desenvolvedores acham mais intuitivo que a outras sintaxes, como o @for do Angular.

- *Watchers (watch)*: ideal para efeitos colaterais, como persistir dados no localStorage sempre que o estado reativo sofrer alteração.

---

5. Comparação: Angular, React, Svelte e VueJS

A tabela a seguir resume as principais diferenças entre as quatro tecnologias apresentadas anteriormente.

Aspecto	Vue JS	Angular	React	Svelte
Linguagem	JavaScript (com suporte a TypeScript)	TypeScript (obrigatório)	JavaScript (com suporte a TypeScript)	JavaScript (com suporte a TypeScript)
Tipo	Framework progressivo	Framework completo	iblioteca (requer frameworks para SPA)	Framework compilado
Curva de Aprendizado	Mais suave, ideal para iniciantes	Mais acentuada, requer conhecimento de TypeScript	Moderada, requer conhecimento de JSX	Muito suave, sintaxe próxima ao HTML, CSS e JS tradicional
Filosofia	Framework progressivo, flexível	Estrutura completa e opinada	Flexível, foco na view layer	Compilação em tempo de build, menos runtime
Data Binding	Reatividade automática com Proxy	Observables e Zone.js	Virtual DOM e reconciliação	Reatividade compilada em tempo de build
Sintaxe de Template	Diretivas com prefixo v-	Sintaxe Angular com parênteses	JSX (JavaScript XML)	Sintaxe HTML estendida com

		e colchetes		reatividade
Injeção de Dependência	Opcional (plugins)	Central e obrigatória	Não nativa (requer Context ou bibliotecas)	Não necessária (stores simples)
Virtual DOM	Sim	Não (change detection)	Sim (core do React)	Não (compilação)
Comunidade	Comunidade crescente	Comunidade grande e estabelecida	Comunidade muito grande e ativa	Comunidade em crescimento
Casos de Uso	Projetos pequenos a médios, prototipagem rápida	Aplicações empresariais de larga escala	Aplicações web modernas, SPAs, aplicações nativas (React Native)	Projetos com requisitos de performance extrema, aplicações leves
Performance	Boa	Boa em aplicações bem estruturadas	Excelente com otimizações	Excelente (menos código no navegador)

---

6. Conclusão

Os quatro frameworks e bibliotecas apresentam diferentes abordagens para o desenvolvimento de interfaces web modernas, cada qual com seus pontos fortes e casos de uso ideais.

Vue JS é ideal para desenvolvedores que buscam uma curva de aprendizado suave e uma abordagem progressiva, permitindo começar com simples templates HTML e evoluir gradualmente para aplicações complexas. Sua sintaxe intuitiva e documentação de qualidade o tornam excelente para prototipagem rápida e projetos de médio porte.

Angular é mais apropriado para projetos empresariais de grande escala que se beneficiam de uma estrutura robusta, tipagem estática obrigatória e um ecossistema completo. Sua abordagem opinada e sistema de injeção de dependências facilitam a manutenção de códigos complexos em equipes grandes.

React é uma biblioteca flexível e poderosa que se destaca pela sua comunidade massiva, ecossistema rico e capacidade de criar aplicações web modernas e nativas (React Native). Sua abordagem baseada em componentes e JSX oferece grande flexibilidade, embora requeira decisões arquiteturais adicionais.

Svelte representa uma abordagem inovadora com compilação em tempo de build, resultando em aplicações extremamente leves e performáticas. É a escolha ideal para projetos com requisitos de performance extrema, aplicações leves e desenvolvedores que preferem uma sintaxe mais próxima ao HTML, CSS e JavaScript tradicional.

A escolha entre essas tecnologias deve considerar o tamanho do projeto, a experiência da equipe, os requisitos de performance, as preferências arquiteturais e o tempo disponível para desenvolvimento. Para aplicações pequenas e prototipagem rápida, Vue JS ou Svelte são excelentes escolhas. Para sistemas complexos que exigem escalabilidade e manutenibilidade de longo prazo em equipes grandes, Angular oferece uma base sólida. React é a opção mais versátil e com maior comunidade, adequada para praticamente qualquer tipo de projeto web.

---

6. Utilizando Vite e Docker para Criação e Execução das Aplicações

Para criação das aplicações, utilizando Docker, foi escolhida a imagem “node:22-alpine”. A escolha se deve ao fato do Node.js 22 ser a versão **LTS** (Long Term Support) atual (em 2026), o que garante estabilidade e atualizações de segurança por um bom tempo. Quanto ao Linux Alpine, sua escolha traz os seguintes benefícios:

- Tamanho Reduzido: enquanto uma imagem baseada em no Debian (como a node:22) pode passar de 1GB, a versão alpine costuma ter cerca de 100MB a 150MB. Isso acelera o download (pull) e o deploy.

- Segurança: por ter menos pacotes instalados e uma superfície de ataque reduzida (pontos/portas de um sistema operacional que podem sofrer um ataque, ou seja, um usuário não autorizado pode tentar inserir ou extrair dados), ela é inerentemente mais segura.

- Performance: apresenta menos overhead de sistema operacional significa que mais recursos do seu host Docker ficam disponíveis para a aplicação.

	COMANDOS DOCKER (em 2026)
Angular	docker run --rm -it -v "$(pwd):/app" -w /app --user "$(id -u):$(id -g)" node:22-alpine npx -p @angular/cli ng new meu-projeto-angular

React	docker run --rm -it -v "$(pwd):/app" -w /app --user "$(id -u):$(id -g)" node:22-alpine npm create vite@latest meu-projeto-react -- --template react-ts
Svelte	docker run --rm -it -v "$(pwd):/app" -w /app --user "$(id -u):$(id -g)" node:22-alpine npm create vite@latest meu-projeto-svelte -- --template svelte-ts
Vue	docker run --rm -it -v "$(pwd):/app" -w /app --user "$(id -u):$(id -g)" node:22-alpine npm create vite@latest meu-projeto-vue -- --template vue-ts

Para todos os exemplos abaixo é considerado que o arquivo "*docker-compose.yml*" foi criado e configurado na pasta principal da aplicação, logo a após a execução do comando de criação de cada um dos projetos

---

TAILWIND NO ANGULAR

(repositório pronto: Gerenciamento de Tarefas | porta 12006)

Instalando

Durante a criação do projeto, via linha de comando, basta escolher o Tailwind como framework CSS padrão que o mesmo já será instalado e configurado automaticamente.

Agora para testar se o Tailwind está funcionando corretamente, vamos colocar o seguinte código no arquivo ./src/app/app.html:

```
TypeScript
<style></style>

<div class="min-h-screen bg-slate-900 flex items-center justify-center">
  <span class="text-5xl font-bold text-sky-400 drop-shadow-lg">
    Tailwind + Vite/Angular funcionando!
  </span>
</div>

<router-outlet />
```

TAILWIND NO REACT

(repositório pronto: Gerenciamento de Tarefas | porta 12007)

Instalando

(terminal dentro da pasta da aplicação)

```shell
docker compose exec app npm install tailwindcss @tailwindcss/vite

docker compose exec app npm install -D vite typescript @types/node
```

Configurando o Docker (host)

Por padrão, o Vite ouve o IP 127.0.0.1 (localhost dentro do container). No entanto, para que o Docker consiga repassar as chamadas da sua máquina real para dentro do container, o processo precisa ouvir o IP 0.0.0.0 (todos os endereços de rede do container).
Abra o arquivo “packgage.json” e efetue a seguinte configuração:

```typescript
/* package.json */

"scripts": {
    "dev": "vite --host 0.0.0.0",   /* Linha alterada, remova esse comentário */
    "build": "tsc -b && vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  },
```

Configurando o Tailwind

Abra o arquivo “vite.config.ts” e efetue a seguinte configuração:

```typescript
/* vite.config.ts */

import { defineConfig } from 'vite'
```

```
import react from '@vitejs/plugin-react'
import tailwindcss from '@tailwindcss/vite' // linha adiocinada

// https://vite.dev/config/
export default defineConfig({
  plugins: [react(), tailwindcss(),], // linha alterada
})
```

Crie (ou abra) um arquivo CSS global, geralmente em ./src/App.css, e adicione as três linhas mágicas do Tailwind:

TypeScript
```
@import "tailwindcss";
```

Agora para testar se o Tailwind está funcionando corretamente, vamos colocar o seguinte código no arquivo ./src/App.tsx:

TypeScript
```
import './App.css'

function App() {

  return (

    <div className="min-h-screen bg-slate-900 flex items-center justify-center">
      <span className="text-5xl font-bold text-sky-400 drop-shadow-lg">
        Tailwind + Vite/React funcionando!
      </span>
    </div>

  )
}

export default App
```

Como resultado você deve obter o seguinte resultado, no navegador:

INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação

Tailwind + Vite/React funcionando!

TAILWIND NO SVELTE

(repositório pronto: Gerenciamento de Tarefas | porta 12008)

Instalando

(terminal dentro da pasta da aplicação)

```shell
docker compose exec app npm install tailwindcss @tailwindcss/vite

docker compose exec app npm install -D vite typescript @types/node
```

Configurando o Docker (host)

Por padrão, o Vite ouve o IP 127.0.0.1 (localhost dentro do container). No entanto, para que o Docker consiga repassar as chamadas da sua máquina real para dentro do container, o processo precisa ouvir o IP 0.0.0.0 (todos os endereços de rede do container).
Abra o arquivo “packgage.json” e efetue a seguinte configuração:

```typescript
/* package.json */

"scripts": {
    "dev": "vite --host 0.0.0.0",   /* Linha alterada, remova esse comentário */
    "build": "vite build",
    "preview": "vite preview",
    "check": "svelte-check --tsconfig ./tsconfig.app.json && tsc -p tsconfig.node.json"
  },
```

Configurando o Tailwind

Abra o arquivo “vite.config.ts” e efetue a seguinte configuração:

```typescript
/* vite.config.ts */

import { defineConfig } from 'vite'
import { svelte } from '@sveltejs/vite-plugin-svelte'
import tailwindcss from '@tailwindcss/vite' // linha adiocinada

// https://vite.dev/config/
export default defineConfig({
  plugins: [svelte(), tailwindcss(),], // linha alterada
})
```

Crie (ou abra) um arquivo CSS global, geralmente em ./src/app.css, e adicione as três linhas mágicas do Tailwind:

```typescript
@import "tailwindcss";
```

Agora para testar se o Tailwind está funcionando corretamente, vamos colocar o seguinte código no arquivo ./src/App.svelte:

```typescript
<div class="min-h-screen bg-slate-900 flex items-center justify-center">
  <h1 class="text-5xl font-bold text-sky-400 drop-shadow-lg">
    Tailwind + Vite/Svelte funcionando!
  </h1>
</div>
```

Como resultado você deve obter o seguinte resultado, no navegador:

TAILWIND NO VUE

(repositório pronto: Gerenciamento de Tarefas | porta 12009)

Instalando

(terminal dentro da pasta da aplicação)

```shell
docker compose exec app npm install tailwindcss @tailwindcss/vite

docker compose exec app npm install -D vite typescript @types/node
```

Configurando o Docker (host)

Por padrão, o Vite ouve o IP 127.0.0.1 (localhost dentro do container). No entanto, para que o Docker consiga repassar as chamadas da sua máquina real para dentro do container, o processo precisa ouvir o IP 0.0.0.0 (todos os endereços de rede do container).
Abra o arquivo “packgage.json” e efetue a seguinte configuração:

```typescript
/* package.json */

"scripts": {
    "dev": "vite --host 0.0.0.0",   /* Linha alterada, remova esse comentário */
    "build": "vue-tsc -b && vite build",
    "preview": "vite preview"
  },
```

Configurando o Tailwind

Abra o arquivo “vite.config.ts” e efetue a seguinte configuração:

```typescript
/* vite.config.ts */

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import tailwindcss from '@tailwindcss/vite' // linha adiocinada

// https://vite.dev/config/
```

```
export default defineConfig({
  plugins: [vue(), tailwindcss(),], // linha alterada
})
```

Crie (ou abra) um arquivo CSS global, geralmente em ./src/style.css, e adicione as três linhas mágicas do Tailwind:

```TypeScript
@import "tailwindcss";
```

Agora para testar se o Tailwind está funcionando corretamente, vamos colocar o seguinte código no arquivo ./src/App.vue:

```TypeScript
<script setup lang="ts"></script>

<template>
  <div class="min-h-screen bg-slate-900 flex items-center justify-center">
    <h1 class="text-5xl font-bold text-sky-400 drop-shadow-lg">
      Tailwind + Vite/Vue funcionando!
    </h1>
  </div>
</template>
```