

Objetivos:

- I. React;
- II. Vite;
- III. Projeto React TypeScript usando Vite;
- IV. Componente React;
- V. React.Fragment.

I. React

O React é uma biblioteca JS (JavaScript), criada pelo Facebook, que simplifica a construção de interfaces de usuário interativas para aplicações web. Ele divide a interface em componentes reutilizáveis, facilitando a manutenção e a organização do código.

Características e conceitos principais:

- Componentes: blocos de construção independentes que podem ser combinados para criar interfaces complexas. Essa abordagem torna o código mais modular, organizado e fácil de manter;
- Virtual DOM (Documento de Objeto Modelo): uma representação em memória da interface do usuário, que permite ao React aplicar as mudanças de forma eficiente, otimizando o desempenho. Quando o estado de um componente muda, o React compara o Virtual DOM com o DOM real e atualiza apenas as partes modificadas, em vez de recarregar toda a interface. Isso melhora o desempenho e a eficiência das atualizações;
- JSX (JavaScript XML): sintaxe similar ao HTML que torna a criação de componentes mais intuitiva e expressiva;
- Gerenciamento de estado: o React oferece formas eficazes de gerenciar o estado dos componentes. Quando o estado muda, o React se encarrega de renderizar a UI novamente, garantindo uma atualização eficiente e previsível;
- Comunidade ativa: grande comunidade de desenvolvedores contribui para o crescimento e evolução do React.

II. Vite

Embora seja frequentemente usado em conjunto com frameworks como React, Vue e outros, o Vite se concentra em otimizar o processo de desenvolvimento e construção de aplicações web. Ele oferece um servidor de desenvolvimento rápido e um ambiente de construção eficiente, utilizando tecnologias modernas como ES modules nativos do navegador (<https://vite.dev>).

Para entender melhor a diferença entre framework e ferramenta de build:

- Frameworks: fornecem uma estrutura e componentes pré-construídos para agilizar o desenvolvimento de aplicações, definindo padrões e convenções. Exemplos: React, Angular e Vue;

- Ferramentas de build: são responsáveis por transformar o código fonte em um formato otimizado para ser executado no navegador, gerenciando módulos, compilando código, minificando e otimizando assets. Outras ferramentas de build para projetos JS são o Webpack, Parcel, Rollup e ESBUILD.

O principal objetivo do Vite é resolver as limitações das ferramentas de build tradicionais, como o Webpack, usado pelo Create React App (CRA), proporcionando um ambiente de desenvolvimento mais ágil e otimizado.

III. Projeto React TypeScript usando Vite

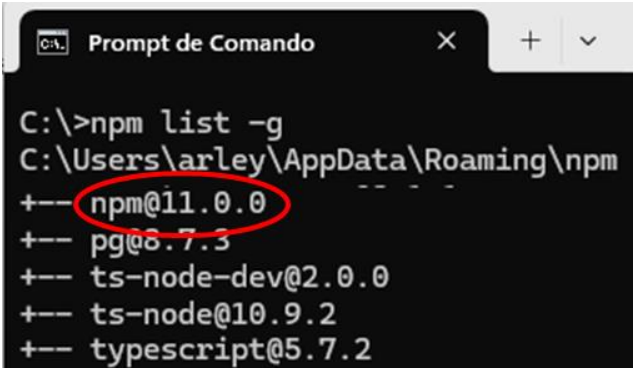
Antes de começar é necessário ter instalado o Node.js no seu computador. Abra o prompt e digite o comando

```
npm list -g
```

para exibir a lista de pacotes instalados globalmente no sistema (computador). É sempre bom trabalhar com as versões mais recentes do npm (Node Package Manager). Se desejar, use o comando a seguir para atualizar a versão instalada do NPM:

```
npm i npm -g
```

O parâmetro `-g` é usado para instalar globalmente, ou seja, estará disponível em todo o sistema.



```
C:\>npm list -g
C:\Users\arley\AppData\Roaming\npm
+-- npm@11.0.0
+-- pg@8.7.3
+-- ts-node-dev@2.0.0
+-- ts-node@10.9.2
+-- typescript@5.7.2
```

Para criar um novo projeto React utilizando TS e a ferramenta de build Vite, execute o seguinte comando no terminal (prompt):

```
npm create vite@latest front -- --template react-ts
```

Entendendo o comando:

- `npm create vite@latest`: inicia o processo de criação de um novo projeto utilizando a versão mais recente do Vite. O comando baixa e executa o script de criação diretamente do repositório oficial do Vite;
- `front`: define o nome da pasta que será criada o projeto. Podemos substituir "front" por outro nome;
- `--`: a partir da versão 7 do npm, é necessário adicionar um duplo hífen (`--`) antes de especificar os templates;
- `--template react-ts`: especifica que será usado o template de projeto React com TS.

Ao executar o comando pela primeira vez, pode pedir para instalar o pacote `create-vite`, assim como é mostrado a seguir:

```
D:\aulas>npm create vite@latest front -- --template react-ts
Need to install the following packages:
create-vite@6.0.1
Ok to proceed? (y) y

> npx
> create-vite front --template react-ts

Scaffolding project in D:\aulas>npm create vite@latest front -- --template react-ts

Done. Now run:

  cd front
  npm install
  npm run dev
```

Após criar o projeto será necessário entrar na pasta `front` (usando o comando `cd front`) e instalar as dependências usando o comando `npm i` ou `npm install`.

Abra o projeto no VS Code para visualizar o código (use o comando `code .`) e no terminal digite o comando `npm run dev` para subir/iniciar o servidor.

Para ver o resultado abra a URL <http://localhost:5173/> no navegador.

```
PS D:\aulas\front> npm run dev
> front@0.0.0 dev
> vite

VITE v6.0.3 ready in 879 ms
  → Local:   http://localhost:5173/
  → Network: use --host to expose
  → press h + enter to show help
```

Ao lado tem-se a estrutura do projeto criada usando o build Vite:

- `node_modules/`: contém todas as dependências instaladas pelo npm. Qualquer projeto Node precisa dessa pasta;
- `public/`: opcional, pode não ser gerada. Pasta para arquivos públicos que não passam pelo sistema de build. Tudo aqui é servido diretamente;
- `index.html`: arquivo HTML principal da aplicação, responsável por renderizar o componente raiz do React no elemento `root`:

```
<div id="root"></div>
```

- `src/`: pasta principal do código-fonte da aplicação, ou seja, é aqui que iremos colocar o nosso código. A seguir tem-se alguns arquivos dessa pasta:

- `main.tsx`: ponto de entrada do React. Ele coloca no elemento `<div id="root"></div>` aquilo que o componente `App` retorna:

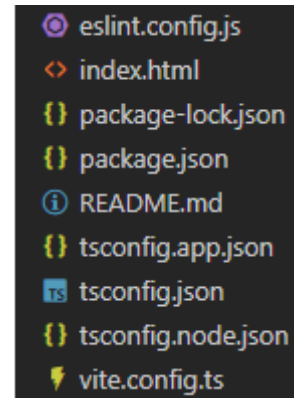
```
createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

```
▼ FRONT
  > node_modules
  ▼ public
    vite.svg
  ▼ src
    ▼ assets
      react.svg
    # App.css
    App.tsx
    # index.css
    main.tsx
    TS vite-env.d.ts
```

- App.tsx: componente principal da aplicação;
- vite-env.d.ts: serve para fornecer informações de tipo específicas para o ambiente de desenvolvimento Vite. A linha

```
/// <reference types="vite/client" />
```

informa ao TS que deve incluir declarações de tipo fornecidas pelo pacote `vite/client`. Isso permite que o TS reconheça variáveis, módulos e funcionalidades específicas do Vite;
- assets: pasta para armazenar arquivos estáticos, como imagens, PDF etc.



- package.json: configuração do projeto, incluindo scripts e dependências do npm. Neste arquivo está a propriedade `dev` com o comando para subir o projeto;
- tsconfig.json: configuração principal do TS. Define regras de compilação;
- vite.config.ts: arquivo de configuração do Vite, onde podemos personalizar o comportamento do servidor de desenvolvimento, opções de build, plugins e outras configurações específicas do projeto.

A Figura 1 possui o código do arquivo `index.html`. O React precisa de um "ponto de montagem" (mounting point) no DOM para renderizar seus componentes. O elemento `<div id="root"></div>` é onde o React renderizará a árvore de componentes da aplicação. No código `main.tsx` (Figura 2), o React faz referência a essa `div`, `document.getElementById('root')` para inserir o conteúdo dinamicamente.

Fluxo de inicialização do projeto:

1. O navegador carrega o arquivo `index.html`;
2. O script `<script type="module" src="/src/main.tsx"></script>` é executado;
3. No `main.tsx`, o React monta o componente raiz (`<App />`) dentro da `<div id="root"></div>`;
4. A árvore de componentes do React é exibida dentro da `div#root`, criando a interface do usuário.

```
<!doctype html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React + TS</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

Figura 1 – Código do arquivo `index.html`.

Detalhando o código do arquivo main.tsx:

StrictMode:

- É um componente do React usado para identificar potenciais problemas na aplicação durante o desenvolvimento;
- Ele não afeta a aplicação em produção;
- Garante boas práticas e ajuda a identificar possíveis problemas relacionados ao ciclo de vida dos componentes.

index.css:

- Aponta para um arquivo de estilos globais que afetam a aparência dos componentes React.

App:

- Importa o componente raiz da aplicação;
- Representa o ponto inicial da árvore de componentes React.

createRoot:

- Método introduzido no React 18 para gerenciar a renderização no DOM de forma mais eficiente;
- Cria um ponto de montagem React no elemento DOM com o `id root`;
- O `document.getElementById('root')` **!** assegura ao TS que o elemento root não será null.

render:

- Renderiza o componente raiz (`<App />`) dentro do elemento `div#root` definido no index.html;
- O método render substitui qualquer conteúdo anterior na `div#root` pela interface gerada pelo React.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Figura 2 – Código do arquivo src/main.tsx.

IV. Componente React

Um componente React é essencialmente uma função JS que retorna um elemento JSX (JavaScript XML) representando parte da interface do usuário. A Figura 3 possui um exemplo de código para o componente App:

Função:

- O componente é declarado como uma função chamada App.

Retorno JSX:

- A função retorna JSX, que é a estrutura visual do componente. Todo componente obrigatoriamente precisa exportar um JSX.

Exportação padrão:

- O componente é exportado usando `export default` para que possa ser importado e utilizado em outros arquivos.

Na estrutura do projeto o elemento `<div id="root"></div>` receberá o JSX retornado pelo componente App, desta forma, será exibido no navegador `<div id="root"><div>Boa noite</div></div>`.

```
export default function App() {
  return <div>Boa noite</div>;
}
```

Figura 3 – Código do arquivo src/App.tsx.

O JSX é uma extensão de sintaxe do JS, utilizada pelo React, que permite escrever código que se assemelha ao HTML diretamente dentro do JS. Como exemplo, a instrução `return` a seguir é um código JS que possui a sua direita uma marcação HTML, o JSX permite colocarmos estruturas XML dentro do código JS:

```
return <div>Boa noite</div>;
```

O JSX não é compreendido diretamente pelos navegadores. É necessário que ferramentas como Babel, TS ou esbuild (utilizado pelo Vite) convertam JSX para JS puro antes da execução.

No JSX, as chaves `{}` permitem a inserção de expressões JS dentro do código JSX. Essas expressões são avaliadas e renderizadas como parte do componente React.

Podemos fazer operações matemáticas dentro das chaves. Aqui o navegador exibirá: Resultado 12	Podemos exibir valores de variáveis. Aqui o navegador exibirá: Olá Ana e Pedro!
<pre>export default function App() { return <div>Resultado {2+2*5}</div>; }</pre>	<pre>export default function App() { const um = "Ana", dois = "Pedro"; return <div>Olá {um} e {dois}!</div>; }</pre>

Podemos fazer a chamada de funções. Aqui o navegador exibirá: Resultado 5	Podemos acessar propriedades de objetos JSON. Aqui o navegador exibirá: Nome: Maria Sobrenome: Souza
<pre>export default function App() {</pre>	<pre>export default function App() {</pre>

<pre> return <div>Resultado {somar(2,3)}</div>; } function somar(a:number,b:number){ return a + b; } </pre>	<pre> return (<div> <div>Nome: {usuario.nome}</div> <div>Sobrenome: {usuario.sobrenome}</div> </div>); const usuario = { nome: "Maria", sobrenome: "Souza", }; </pre>
--	--

<p>Podemos usar o método map do array para percorrer os elementos do array. Porém, cada elemento renderizado precisa ter a propriedade key com valor único. Aqui o navegador exibirá:</p> <ol style="list-style-type: none"> domingo segunda-feira terça-feira 	<p>Podemos renderizar condicionalmente. O resultado da condição do operador ternário determinará quais dos resultados serão retornados pelo componente App. Aqui o navegador exibirá:</p> <p>7 é maior que 4</p>
<pre> export default function App() { return ({ dias.map((dia,indice) => (<li key={indice}>{dia})) }); } const dias = ['domingo','segunda- feira','terça-feira']; </pre>	<pre> export default function App() { const nro = aleatorio(); return (<div> {nro < 5? <div>{nro} é menor que 5</div> : <div>{nro} é maior que 4</div> } </div>); } function aleatorio(){ //retorna um número aleatório entre 0 e 9 return Math.floor(Math.random()*10); } </pre>

Podemos renderizar usando o operador **&&** (AND lógico). Se a condição for verdadeira, a expressão após o **&&** será renderizada. Aqui o navegador exibirá:

Resultado:3 é menor que 5

```

export default function App() {
  const nro = aleatorio();
  return (
    <div>

```

```

    Resultado:
    {nro < 5 && <span>{nro} é menor que 5</span>}
  </div>
);
}

function aleatorio() {
  //retorna um número aleatório entre 0 e 9
  return Math.floor(Math.random() * 10);
}

```

Podemos usar expressões para renderizar usando estilos inline dinâmicos. No exemplo a seguir o resultado receberá estilos de acordo com o resultado da expressão condicional:

```

export default function App() {
  const nro = aleatorio();
  return (
    <div>
      {nro < 5?
        <div style={menor}>{nro} é menor que 5</div> :
        <div style={maior}>{nro} é maior que 4</div>
      }
    </div>
  );
}

function aleatorio(){
  //retorna um número aleatório entre 0 e 9
  return Math.floor(Math.random()*10);
}

const menor = {
  color: 'blue',
  fontStyle: 'italic'
};

const maior = {
  color: 'red',
  fontWeight: 'bold'
};

```

Observação:

O componente obrigatoriamente deve retornar apenas um elemento pai. A expressão JSX a seguir está errada pelo fato de retornar dois elementos `div`.

```

export default function App() {
  return (
    <div>Sábado</div>

```



```

    <div>Domingo</div>
  );
}

```

O correto é envolver o retorno por um componente pai, assim como é mostrado em amarelo no código a seguir:

```

export default function App() {
  return (
    <div>
      <div>Sábado</div>
      <div>Domingo</div>
    </div>
  );
}

```

V. React.Fragment

É um recurso do React que permite agrupar vários elementos filhos sem adicionar um nó extra ao DOM. Ele é útil quando precisamos retornar múltiplos elementos de um componente React, mas não desejamos envolver esses elementos em uma `<div>` ou outro elemento HTML adicional.

Vantagens do React.Fragment:

- Evita nós desnecessários no DOM:
 - Adicionar elementos extras ao DOM pode interferir no layout e no estilo, especialmente ao usar CSS Flexbox ou Grid;
 - O React.Fragment evita essa poluição estrutural.
- Melhora a performance:
 - Menos nós no DOM significam melhor desempenho na renderização.
- Semântica mais limpa:
 - Torna o código mais legível e alinhado com a estrutura esperada.

Podemos usar o React.Fragment de duas formas:

Forma explícita com <code>React.Fragment</code> . Observe que é necessário importar o <code>React</code> .	Forma abreviada com <code><></code> e <code></></code> (shorthand). A forma abreviada (<code><></code>) é mais comum por ser mais concisa, mas não suporta atributos.
<pre> import React from "react"; export default function App() { return (<React.Fragment> <div>Sábado</div> <div>Domingo</div> </React.Fragment>); } </pre>	<pre> export default function App() { return (<> <div>Sábado</div> <div>Domingo</div> </>); } </pre>

}	
---	--

VI. Exercícios

Exercício 1 – Um componente pode ser chamado por outro componente usando a notação de elemento XML. Complete o código do componente App para que ele faça a chamada do componente Message.

```
export default function App() {
  return _____;
}

function Message(){
  return <div>Bom dia!</div>;
}
```

Exercício 2 – O componente Message ao lado retorna a saudação “Bom dia!”, “Boa tarde!” ou “Boa noite!” de acordo com a hora. Alterar o código do componente Message para usar o operador ternário ao invés da estrutura if-else.

```
function Message() {
  // Obtém a hora atual (0-23)
  const currentHour = new Date().getHours();
  let greeting;

  if (currentHour < 12) {
    greeting = "Bom dia!";
  } else if (currentHour < 18) {
    greeting = "Boa tarde!";
  } else {
    greeting = "Boa noite!";
  }

  return <div>{greeting}</div>;
}
```

Exercício 3 – Alterar o componente Message para fazer uso dos estilos que estão nas variáveis dia, tarde e noite.

```
const dia = {
  backgroundColor: 'yellow'
};

const tarde = {
  backgroundColor: 'orange'
};

const noite = {
  backgroundColor: '#888'
};
```

Exercício 4 – Complete o código do componente App para que os elementos do array sejam renderizado em uma lista, assim como é mostrado a seguir.

```
export default function App() {
  const nomes = ["Ana", "Bruno", "Carla", "Daniel"];

  return _____;
}
```

Dica: use o método `map` para percorrer o array `nomes`.

1. Ana
2. Bruno
3. Carla
4. Daniel

Exercício 5 – Complete o código do componente `App` para que os elementos do array sejam renderizado em uma lista, assim como é mostrado a seguir. Os elementos que possuem o gênero feminino devem usar o estilo que está variável `f` e os demais devem usar o estilo que está na variável `m`.

1. Ana
2. Bruno
3. Carla
4. Daniel

```
export default function App() {  
  const nomes = [  
    {  
      name: "Ana",  
      gender: "F",  
    },  
    {  
      name: "Bruno",  
      gender: "M",  
    },  
    {  
      name: "Carla",  
      gender: "F",  
    },  
    {  
      name: "Daniel",  
      gender: "M",  
    },  
  ],  
};  
  
  return _____;  
}  
  
const f = {  
  backgroundColor: "orange",  
};  
  
const m = {  
  backgroundColor: "palegreen",  
};
```

Exercício 6 – Complete o código do componente `App` para que os elementos do array sejam renderizados em uma lista, assim como é mostrado a seguir. Os elementos que possuem idade menor que 18 devem ter o texto (*menor*).

Dica: use o operador `&&` (AND lógico).

```
export default function App() {  
  const nomes = [  
    {  
      name: "Ana",  
      gender: "F",  
      age: 21  
    },  
    {  

```

1. Ana
2. Bruno (menor)
3. Carla (menor)
4. Daniel

```
        name: "Bruno",
        gender: "M",
        age: 17
    },
    {
        name: "Carla",
        gender: "F",
        age: 15
    },
    {
        name: "Daniel",
        gender: "M",
        age: 22
    },
];

return _____;
}

const f = {
    backgroundColor: "orange",
};

const m = {
    backgroundColor: "palegreen",
};
```