

Objetivos:

- I. React;
- II. Criar um projeto React TypeScript;
- III. Estrutura de um projeto React TypeScript;
- IV. React component;
- V. Comunicação entre componentes usando props;
- VI. Objeto state;
- VII. Objeto context;
- VIII. Hooks;
- IX. Ciclo de vida do componente;
- X. HTTP Request;
- XI. Gerar a versão de produção da aplicação.

I. React

O React é uma biblioteca JS de código aberto desenvolvida pelo Facebook para a criação de interfaces de usuário (UI) interativas e reativas em aplicativos da web. Ele permite que os desenvolvedores dividam a interface do usuário em componentes reutilizáveis e gerenciem o estado desses componentes de forma eficiente.

Características e conceitos:

- Componentes: o React utiliza uma abordagem baseada em componentes para a construção de interfaces de usuário. Os componentes são blocos de construção reutilizáveis que podem ser compostos para criar interfaces complexas. Isso torna o código mais organizado e fácil de manter;
- Virtual DOM (Documento de Objeto Modelo): é uma representação virtual da árvore de elementos da interface do usuário. Quando ocorrem alterações no estado de um componente, o React compara o Virtual DOM com o DOM real e atualiza apenas as partes que mudaram, em vez de recriar toda a interface do usuário. Isso melhora o desempenho e a eficiência das atualizações;
- Reatividade: o React torna a UI reativa às mudanças no estado dos componentes. Quando o estado de um componente muda, o React atualiza automaticamente a interface do usuário para refletir essas mudanças, sem a necessidade de manipulação manual do DOM;
- JSX: o React utiliza uma sintaxe especial chamada JSX (JavaScript XML) para descrever a estrutura da interface do usuário. O JSX permite escrever código JS parecido com HTML, tornando mais fácil a criação de componentes;
- Gerenciamento de estado: o React fornece uma maneira eficaz de gerenciar o estado da aplicação. Podemos definir o estado de um componente e atualizá-lo conforme necessário. Quando o estado muda, o React cuida da atualização da UI de maneira eficiente;
- Comunidade ativa: o React possui uma comunidade de desenvolvedores ativa e uma ampla gama de bibliotecas e ferramentas de terceiros que podem ser usadas para estender suas funcionalidades;
- Facilidade de teste: devido à sua estrutura modular, os componentes React são facilmente testáveis.

II. Criar um projeto React TypeScript

Siga os passos para criar uma aplicação React TS:

- Utilizaremos a ferramenta CRA - Create React App (<https://create-react-app.dev/docs/getting-started>) para criar os projetos React. Primeiramente abra o terminal de comando (Prompt CMD) e verifique se você tem instalado o pacote create-react-app:

```
C:\Windows\System32\cmd.exe
D:\>npx create-react-app -V
5.0.1

D:\>npx create-react-app --version
5.0.1
```

Se você não tem instalado, use o comando `npm i create-react-app -g` para instalar a aplicação globalmente no computador. Recomenda-se que você tenha instalada a última versão, então verifique no site <https://www.npmjs.com/package/create-react-app>.

Para atualizar a versão você terá de desinstalar e reinstalar a versão global:

```
npm uninstall create-react-app -g
```

- Acesse pelo prompt do CMD o local que você deseja criar o projeto React e digite o comando a seguir para criar o projeto React:

```
npx create-react-app front --template typescript
```

O projeto será criado na pasta `front`.

O parâmetro `--template` é usado para criar o projeto baseado no template de aplicação React TypeScript.

A CLI (Command Line Interface – Interface de Linha de Comando) usa npm ou yarn para instalar as dependências, dependendo da ferramenta que você use para rodar o comando `create-react-app`. Aqui o projeto usará npm pelo fato de termos usado npx para rodar `create-react-app`.

- NPM é uma ferramenta de interface de linha de comando, cujo propósito é facilitar a instalação e o gerenciamento de dependências hospedadas no registro do npm.
- No CMD acesse a pasta `front` e abra ela no VS Code;
- Ao lado tem-se a estrutura de pastas e arquivos da aplicação criada pelo CRA;
- Para simplificar o projeto:
 - Delete os arquivos sinalizados pela seta vermelha;
 - Substitua os códigos dos arquivos `index.html` (Figura 1), `index.tsx` (Figura 2) e `App.tsx` (Figura 3);
 - Para subir o projeto digite `npm run start` ou `npm start` no



terminal do VS Code. A aplicação estará na porta padrão 3000.

Observações:

- O arquivo package.json é necessário pelo fato de a aplicação ser baseada em Node;
- O arquivo tsconfig.json especifica as configurações de compilação necessária para um projeto TS;
- Não precisamos instalar ou configurar ferramentas como Babel, Webpack e ESLint. Eles são pré-configurados e ocultos no projeto pelo CRA. Babel é um transpilador que transforma JavaScript ES6+ ([ECMAScript](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Referência/ECMAScript)) para JavaScript ES5 (compatível com o JS suportado pelos navegadores antigos). Em outras palavras, ele traduz códigos mais modernos em versões compatíveis com a ES5;
- Webpack é um empacotador (bundler) de módulos (arquivos) estáticos para aplicativos JS. Uma aplicação React é formada por módulos que possuem dependências entre si, o Webpack cria internamente um gráfico de dependências a partir de um ou mais pontos de entrada e, em seguida, combina todos os módulos que o projeto precisa em um ou mais módulos (<https://webpack.js.org/concepts>). O processo de agrupar os módulos reduz as solicitações HTTP melhorando o desempenho do aplicativo ao ser consumido pelo navegador;
- ESLint é uma ferramenta que analisa estaticamente o código JS e TS e aponta os erros durante a digitação. A análise estática de um programa é aquela realizada sem executá-lo;
- O arquivo src/react-app-env.d.ts é apenas para garantir que os tipos essenciais create-react-app sejam selecionados pelo compilador TS. É melhor não remover ou alterar, pois pode ser atualizado por possíveis alterações no aplicativo CRA.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Front</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Figura 1 – Código do arquivo public/index.html.

```
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render( <App /> );
```

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

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

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

III. Estrutura de um projeto React TypeScript

A aplicação React é um SPA (Single-Page Application), desta forma, toda a renderização é feita em apenas uma página (documento web). O corpo desse documento é atualizado por meio de APIs JS à medida que o usuário interage com a página. O Facebook, Instagram, WhatsApp Web, Netflix, Twitter e Uber são exemplos de aplicações que construídas usando React. Uma aplicação React é formada por componentes que são carregados à medida que eles se tornam necessários. O arquivo `public/index.html` possui a única página da aplicação, por este motivo ela é uma SPA. Todo o conteúdo do site será renderizado dentro da seguinte marcação que possui id `root` (Figura 1):

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

O arquivo `index.tsx` (Figura 2) faz a ligação do código TS com o código HTML. O método `render` coloca - dentro da marcação que possui o `id="root"` - o código XML retornado pela função `App()` (Figura 3). Desta forma, o corpo da página terá o seguinte conteúdo HTML:

```
<body>  
  <noscript>You need to enable JavaScript to run this app.</noscript>  
  <div id="root">  
    <div> boa noite </div>  
  </div>  
</body>
```

Observação: um arquivo `tsx` é aquele que possui códigos TypeScript e XML.

IV. React component

O React é estruturado a partir de componentes. Cada componente pode ser criado usando a notação de classe ou função construtora, assim como é mostrado a seguir:

```
function App() {  
  return <div> boa noite </div>;  
}
```

O React utiliza JSX (JavaScript XML) para traduzir código HTML em JS em tempo de execução. O JSX aceita código XML/HTML dentro de instruções JavaScript/TypeScript. No exemplo anterior a instrução `return` deveria retornar um valor compatível com a linguagem TS, mas retornará a marcação `<div>`.

Para colocarmos código TS dentro do código XML temos de usar um par de chaves. A seguir será exibido como resultado o conteúdo da variável `nome`:

```
function App() {
  const nome:string = "Ana";
  return <div>boa noite {nome}</div> ;
}
```

Para criarmos objetos a partir de uma função deveríamos usar o construtor, mas no React utilizamos uma marcação XML. No exemplo a seguir o componente **App** faz uso dos componentes **Pessoa** e **Produto**. Em tempo de execução, as marcações **<Pessoa />** e **<Produto />** serão substituídas pelo retorno das funções **Pessoa()** e **Produto()**, respectivamente:

O que codificamos em React:

```
function App() {
  return (
    <div>
      <div>Descrição:</div>
      <Pessoa />
      <Produto />
    </div>
  );
}

export default App;

function Pessoa() {
  return (
    <div>
      <div>Nome: Ana</div>
      <div>Idade: 20</div>
    </div>
  );
}

function Produto() {
  return (
    <div>
      <div>Nome: Notebook</div>
      <div>Valor: R$3000</div>
    </div>
  );
}
```

O que de fato será retornado pela função APP:

```
function App() {
  return (
    <div>
      <div>Descrição:</div>
      <div>
        <div>Nome: Ana</div>
        <div>Idade: 20</div>
      </div>
      <div>
        <div>Nome: Notebook</div>
        <div>Valor: R$3000</div>
      </div>
    </div>
  );
}
```

O que será exibido no navegador:

```
Descrição:
Nome: Ana
Idade: 20
Nome: Notebook
Valor: R$3000
```

No exemplo anterior o componente **App** chama os componentes **Pessoa** e **Produto**, mas os componentes podem ser chamados de forma aninhada. No exemplo a seguir o componente **Produto** é chamado de dentro da função **Pessoa**:

O que codificamos em React:

```
function App() {
  return (
    <div>
      <div>Descrição:</div>
```

O que de fato será retornado pela função APP:

```
function App() {
  return (
    <div>
      <div>Descrição:</div>
```

```

    <Pessoa />
  </div>
);
}

export default App;

function Pessoa() {
  return (
    <div>
      <div>Nome: Ana</div>
      <div>Idade: 20</div>
      <Produto />
    </div>

  );
}

function Produto() {
  return (
    <div>
      <div>Nome: Notebook</div>
      <div>Valor: R$3000</div>
    </div>

  );
}

```

```

<div>
  <div>Nome: Ana</div>
  <div>Idade: 20</div>
  <div>
    <div>Nome: Notebook</div>
    <div>Valor: R$3000</div>
  </div>
</div>
);
}

```

Observações:

- Precisamos ter algo à direita do `return`:

Está certo por existir parênteses à direita do `return`:

```

function App() {
  return (
    <div>bom dia </div> );
}

```

Está certo por existir uma marcação à direita do `return`:

```

function App() {
  return <div>
    bom dia Ana </div>;
}

```

Está errado por não existir algo à direita do `return`:

```

function App() {
  return
    <div>bom dia </div>;
}

```

- Elementos JSX precisam ter um elemento pai:

Está errado pelas marcações `div` não terem um elemento pai:

```

function App() {
  return (

```

```

    <div>bom dia</div>

```

Está certo pelas DIVs internas estarem envolvidas pela div externa:

```
function
  App() {
    return (
      <div>
```

```
    <div>boa tarde</div>
  );
}
```

```
    <div>bom dia</div>
    <div>boa tarde</div>
  </div>
);
}
```

- Os fragmentos são marcações sem tagname `<></>`, mas eles podem ser declarados usando a marcação `<React.Fragment>`. Os fragmentos são uma alternativa a marcação `<div>` para envolver o retorno do componente. Podemos usar `<>` e `</>` da mesma forma que usamos qualquer outro elemento, exceto pelo fato dele não suportar propriedades e chaves (<https://pt-br.reactjs.org/docs/fragments.html#short-syntax>).

O componente Fragment precisa ser importado:

```
import react from "react";

function App() {
  return (
    <React.Fragment>
      <div>bom dia</div>
      <div>boa tarde</div>
    </React.Fragment>
  );
}
```

Sintaxe curta do fragmento:

```
function App() {
  return (
    <>
      <div>bom dia</div>
      <div>boa tarde</div>
    </>
  );
}
```

V. Comunicação entre componentes usando props

A comunicação entre objetos aninhados pode ser feita através do objeto `props`. No exemplo a seguir os valores `Ana`, `20` e `3250` foram passados para o componente `Pessoa` como propriedades de um objeto JSON com a estrutura `{nome:string,idade:number,valor:number}` definida usando o tipo de dado `PessoaProps`.

```
function App() {
  return (
    <div>
      <div>Descrição:</div>
      <Pessoa nome={"Ana"} idade={20} valor={3250} />
    </div>
  );
}
```

```
export default App;
```

```
interface PessoaProps {
  nome: string;
  idade: number;
  valor: number;
}
```

```
interface ProdutoProps {
  valor: number;
}
```



```
function Pessoa(props:PessoaProps) {
  return (
    <div>
      <div>Nome: {props.nome}</div>
      <div>Idade: {props.idade}</div>
      <Produto valor={props.valor} />
    </div>
  );
}

function Produto(props:ProdutoProps) {
  return (
    <div>
      <div>Nome: Notebook</div>
      <div>Valor: {props.valor}</div>
    </div>
  );
}
```

Observações:

- As propriedades podem ser repassadas na árvore de componentes. Nesse exemplo a propriedade `valor` é repassada do componente Pessoa para o componente Produto;
- Os valores são passados na marcação XML como propriedades da marcação XML, isto é, neste exemplo `nome`, `idade` e `valor` são propriedades XML:

```
<Pessoa nome={"Ana"} idade={20} valor={3250} />
```

O JSX encarrega de estruturar as propriedades XML no seguinte objeto JSON:

```
{nome:"Ana",idade:20,valor: 3250}
```

para serem passados como parâmetro para a função componente.

- Podemos utilizar o spread operator `{...props}` para repassar as propriedades na chamada do componente Produto. No exemplo a seguir será passado como parâmetro para a função Produto uma cópia do objeto `{nome:"Ana",idade:20,valor: 3250}`, porém a função Produto receberá apenas a propriedade `valor`, por ela ser a única definida no cabeçalho da função através do tipo `ProdutoProps`:

```
function Pessoa(props:PessoaProps) {
  return (
    <div>
      <div>Nome: {props.nome}</div>
      <div>Idade: {props.idade}</div>
      <Produto {...props} />
    </div>
  );
}

function Produto(props:ProdutoProps) {
```

```

    return (
      <div>
        <div>Nome: Notebook</div>
        <div>Valor: {props.valor}</div>
      </div>
    );
  }
}

```

VI. Objeto state

O objeto `state` é usado para manter propriedades que podem ser acessadas em todo o componente. Para adicionar propriedades no `state` usamos a função `useState`. A função `useState` recebe como parâmetro de entrada o valor inicial da propriedade, no exemplo a seguir a propriedade `nome` é inicializada com `Ana`.

```

import { useState } from "react";

function App() {
  const [nome, setNome] = useState("Ana");
  return (
    <div>
      <div>
        <label htmlFor="nome">Nome</label>
        <input
          id="nome"
          value={nome}
          onChange={(e) => setNome(e.target.value)}
        />
      </div>
      <Pessoa nome={nome} />
    </div>
  );
}

export default App;

interface PessoaProps {
  nome: string;
}

function Pessoa(props: PessoaProps) {
  return (
    <div>
      <div>Nome: {props.nome}</div>
    </div>
  );
}

```

Observações:

- A função `useState` precisa ser importada do pacote `react`;

- Uma propriedade do estado do componente não deve ser modificada diretamente, por exemplo, `nome = "Maria"`. Para modificar uma propriedade do estado usamos a função devolvida na definição do estado, por exemplo, `setNome("Maria")`;
- A cada modificação no estado o componente é renderizado novamente;
- O objeto `state` estará disponível apenas dentro do próprio componente. Para passar um estado para outro componente é necessário usar o objeto `props`;
- As chamadas para modificar o `state` são assíncronas, isto é, a mudança no estado não ocorrerá imediatamente após a chamada da função associada.

VII. Objeto context

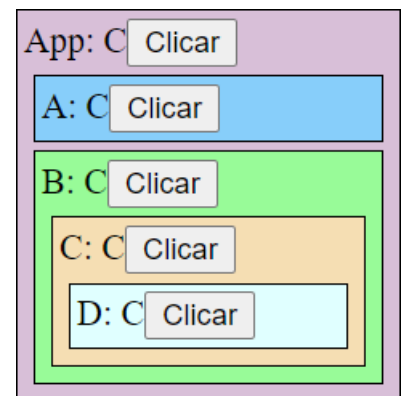
Enquanto por `props` as propriedades precisam ser passadas para cada componente aninhado, via `context` pode-se passar as propriedades para toda a árvore evitando que componentes intermediários tenham de passar explicitamente `props` entre cada nível (<https://react.dev/learn/passing-data-deeply-with-context>).

Contexto é usado principalmente quando algum dado precisa ser acessado por muitos componentes em diferentes níveis. Com ele é possível compartilhar dados que podem ser considerados “globais” para a árvore de componentes do React, por exemplo, o token de autenticação do usuário.

Considere como exemplo a aplicação ao lado:

- Nela os componentes A e B estão dentro do componente App;
- O componente C está dentro do componente B;
- O componente D está dentro do componente C.

Ao clicar em um botão todas as mensagens são atualizadas. Para fazer a comunicação fluir por toda a árvore de componentes precisaríamos definir um estado no componente App e passar ele via `props` para os componentes filhos e esses por sua vez teriam de passar sucessivamente as propriedades para seus filhos.



As funções `createContext` e `useContext` são usadas, respectivamente, para criar e usar um contexto. O objeto contexto estará disponível para qualquer filho da marcação `<Clicado.Provider>`.

`useContext` é um Hook, assim como `useState`, então basta ser chamado para poder usar.

```
import { createContext, useContext, useState } from "react";

function App() {
  const [botao, setBotao] = useState('App');
  return (
    <Clicado.Provider value={{ botao, setBotao }}>
      <div style={{...estilos.geral,...estilos.app}}>
        <span>App: {botao}</span>
        <button onClick={() => setBotao('App')}>Clicar</button>
        <A />
        <B />
      </div>
    </Clicado.Provider>
  );
}
```

```
    );  
  }  
  
  export default App;  
  
  interface Props {  
    botao: string;  
    setBotao: Function;  
  }  
  
  const Clicado = createContext<Props>({} as Props);  
  
  function A() {  
    const { botao, setBotao } = useContext(Clicado);  
    return (  
      <div style={{...estilos.geral,...estilos.aa}}>  
        <span>A: {botao}</span>  
        <button onClick={() => setBotao('A')}>Clicar</button>  
      </div>  
    );  
  }  
  
  function B() {  
    const { botao, setBotao } = useContext(Clicado);  
    return (  
      <div style={{...estilos.geral,...estilos.bb}}>  
        <span>B: {botao}</span>  
        <button onClick={() => setBotao('B')}>Clicar</button>  
        <C />  
      </div>  
    );  
  }  
  
  function C() {  
    const { botao, setBotao } = useContext(Clicado);  
    return (  
      <div style={{...estilos.geral,...estilos.cc}}>  
        <span>C: {botao}</span>  
        <button onClick={() => setBotao('C')}>Clicar</button>  
        <D />  
      </div>  
    );  
  }  
  
  function D() {  
    const { botao, setBotao } = useContext(Clicado);  
    return (  
      <div style={{...estilos.geral,...estilos.dd}}>  
        <span>D: {botao}</span>
```

```

        <button onClick={() => setBotao('D')}>Clicar</button>
    </div>
  );
}

const estilos = {
  geral: {
    padding: 5,
    margin: 4,
    borderWidth: 1,
    borderStyle: 'solid',
    borderColor: 'black',
  },
  app: {backgroundColor: 'thistle'},
  aa: {backgroundColor: 'LightSkyBlue'},
  bb: {backgroundColor: 'PaleGreen'},
  cc: {backgroundColor: 'wheat'},
  dd: {backgroundColor: 'LightCyan'}
};

```

VIII. Hooks

No React, `useState`, assim como qualquer outra função começando com “use”, é chamada de Hook.

Hooks só podem ser chamados no nível superior de seus componentes ou dentro dos Hooks que você criar.

Hooks não podem ser chamados dentro de condições, loops ou outras funções aninhadas.

A seguir tem-se exemplo anterior codificado usando Hook. O primeiro passo foi colocar as propriedades `botao` e `setBotao` numa função – aqui chamada de `ClicadoProvider`. Esta função precisa retornar um `Provider` com a propriedade `value` contendo aquilo que será disponibilizado pelo Hook.

```

function ClicadoProvider({children}:any){
  const [botao, setBotao] = useState('App');
  return (
    <Clicado.Provider value={{botao,setBotao}}>
      {children}
    </Clicado.Provider>
  );
}

```

A função `ClicadoProvider` recebe como parâmetro na variável `children` a árvore de componentes passada na função `App`.

```

function App() {
  return (
    <ClicadoProvider>
      <Principal />
    </ClicadoProvider>
  );
}

```

Desta forma, qualquer componente filho de `Principal` poderá invocar o Hook `useClicado`. O Hook simplesmente retorna o Contexto que dá acesso as propriedades disponibilizadas em `value`.

```
function useClicado(){
  const context = useContext(Clicado);
  return context;
}
```

O acesso ao Hook useClicado se dá da seguinte forma. Veja que o consumidor do Hook não precisa saber detalhes, bastando saber as propriedades disponíveis:

```
const { botao, setBotao } = useClicado();
```

A seguir tem-se o código completo:

```
import { createContext, useContext, useState } from "react";

function App() {
  return (
    <ClicadoProvider>
      <Principal />
    </ClicadoProvider>
  );
}

function Principal(){
  const { botao, setBotao } = useClicado();
  return (
    <div style={{...estilos.geral,...estilos.app}}>
      <span>App: {botao}</span>
      <button onClick={() => setBotao('App')}>Clicar</button>
      <A />
      <B />
    </div>
  );
}

export default App;

interface Props {
  botao: string;
  setBotao: Function;
}

const Clicado = createContext<Props>({} as Props);

function ClicadoProvider({children}:any){
  const [botao, setBotao] = useState('App');
  return (
    <Clicado.Provider value={{botao,setBotao}}>
      {children}
    </Clicado.Provider>
  );
}
```

```
// useClicado é um hook que definimos
// ele retorna as propriedades do value do Clicado.Provider
function useClicado(){
  const context = useContext(Clicado);
  return context;
}

function A() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{...estilos.geral,...estilos.aa}}>
      <span>A: {botao}</span>
      <button onClick={() => setBotao('A')}>Clicar</button>
    </div>
  );
}

function B() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{...estilos.geral,...estilos.bb}}>
      <span>B: {botao}</span>
      <button onClick={() => setBotao('B')}>Clicar</button>
      <C />
    </div>
  );
}

function C() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{...estilos.geral,...estilos.cc}}>
      <span>C: {botao}</span>
      <button onClick={() => setBotao('C')}>Clicar</button>
      <D />
    </div>
  );
}

function D() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{...estilos.geral,...estilos.dd}}>
      <span>D: {botao}</span>
      <button onClick={() => setBotao('D')}>Clicar</button>
    </div>
  );
}
```

```
const estilos = {
  geral: {
    padding: 5,
    margin: 4,
    borderWidth: 1,
    borderStyle: 'solid',
    borderColor: 'black',
  },
  app: {backgroundColor: 'thistle'},
  aa: {backgroundColor: 'LightSkyBlue'},
  bb: {backgroundColor: 'PaleGreen'},
  cc: {backgroundColor: 'wheat'},
  dd: {backgroundColor: 'LightCyan'}
};
```

O código anterior está centralizado no arquivo App.tsx. Porém, é aconselhado organizar o código em pacotes e módulos de acordo com as suas características, assim como é mostrado ao lado:

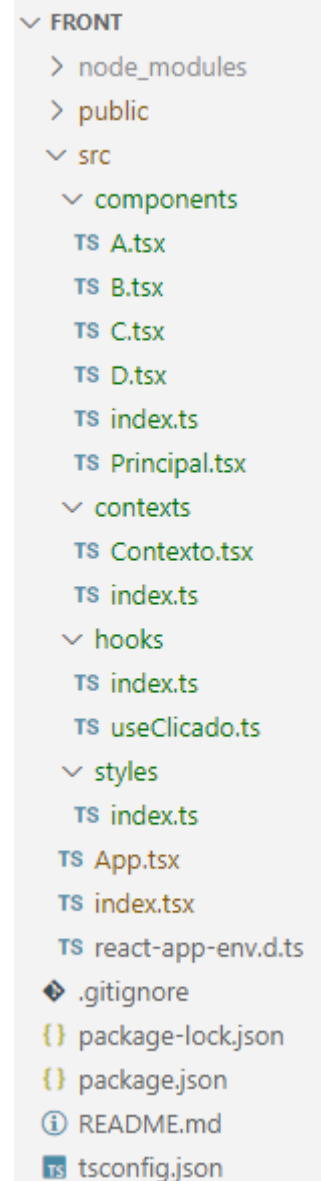
- Pasta components: possui os componentes visuais da aplicação. Utilize os códigos da Figura 4 a Figura 9 nos arquivos desta pasta. Para uma pasta se tornar um pacote temos de criar um arquivo index que exporta os recursos da pasta. Veja como exemplo que o módulo components/index.ts apenas importa e exporta os recursos dos outros módulos do pacote components:

```
import Principal from "./Principal";
import A from "./A";
import B from "./B";
import C from "./C";
import D from "./D";

export { Principal, A, B, C, D };
```

- Pasta contexts: possui os códigos responsáveis pela gestão do contexto. Utilize os códigos da Figura 10 e Figura 11 nos arquivos desta pasta. Por dois motivos é recomendado criar um index mesmo quando existe apenas um módulo na pasta:
 - Primeiro: podemos adicionar novos módulos na pasta e alterar apenas o módulo index para exportar;
 - Segundo: a instrução de importação não expõe o nome do arquivo. Como exemplo, o caminho /contexts não expõe o local onde se encontra o ClicadoProvider:

```
import { ClicadoProvider } from "./contexts";
```



- Pasta hooks: possui os códigos responsáveis pela criação dos Hooks. Utilize os códigos da Figura 12 e Figura 13 nos arquivos desta pasta;
- Pasta styles: possui os códigos responsáveis pela criação dos estilos (Figura 14). Os estilos foram colocados diretamente no arquivo index, porém, o mais aconselhado seria colocar em outro arquivo e deixar o index apenas para a importação e exportação;
- Como o projeto foi fragmentado, o módulo src/index (Figura 15) possui apenas o escopo do aplicativo.

```
import { useClicado } from "../hooks";
import estilos from "../styles";

export default function A() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{ ...estilos.geral, ...estilos.aa }}>
      <span>A: {botao}</span>
      <button onClick={() => setBotao("A")}>Clicar</button>
    </div>
  );
}
```

Figura 4 – Código do arquivo src/components/A.tsx.

```
import { useClicado } from "../hooks";
import estilos from "../styles";
import C from "./C";

export default function B() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{ ...estilos.geral, ...estilos.bb }}>
      <span>B: {botao}</span>
      <button onClick={() => setBotao("B")}>Clicar</button>
      <C />
    </div>
  );
}
```

Figura 5 – Código do arquivo src/components/B.tsx.

```
import { useClicado } from "../hooks";
import estilos from "../styles";
import D from "./D";

export default function C() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{ ...estilos.geral, ...estilos.cc }}>
```

```

    <span>C: {botao}</span>
    <button onClick={() => setBotao("C")}>Clicar</button>
    <D />
  </div>
);
}

```

Figura 6 – Código do arquivo src/components/C.tsx.

```

import { useClicado } from "../hooks";
import estilos from "../styles";

export default function D() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{ ...estilos.geral, ...estilos.dd }}>
      <span>D: {botao}</span>
      <button onClick={() => setBotao("D")}>Clicar</button>
    </div>
  );
}

```

Figura 7 – Código do arquivo src/components/D.tsx.

```

import { useClicado } from "../hooks";
import estilos from "../styles";
import A from "./A";
import B from "./B";

export default function Principal() {
  const { botao, setBotao } = useClicado();
  return (
    <div style={{ ...estilos.geral, ...estilos.app }}>
      <span>App: {botao}</span>
      <button onClick={() => setBotao("App")}>Clicar</button>
      <A />
      <B />
    </div>
  );
}

```

Figura 8 – Código do arquivo src/components/Principal.tsx.

```

import Principal from "../Principal";
import A from "./A";
import B from "./B";
import C from "./C";
import D from "./D";

```

```
export { Principal, A, B, C, D };
```

Figura 9 – Código do arquivo src/components/index.ts.

```
import { createContext, useState } from "react";

interface Props {
  botao: string;
  setBotao: Function;
}

export const Clicado = createContext<Props>({} as Props);

export function ClicadoProvider({ children }: any) {
  const [botao, setBotao] = useState("App");
  return (
    <Clicado.Provider value={{ botao, setBotao }}>{children}</Clicado.Provider>
  );
}
```

Figura 10 – Código do arquivo src/contexts/Contexto.tsx.

```
import { Clicado, ClicadoProvider } from "../Contexto";

export { Clicado, ClicadoProvider };
```

Figura 11 – Código do arquivo src/contexts/index.ts.

```
import { useContext } from "react";
import { Clicado } from "../contexts";

// useClicado é um hook que definimos
// ele retorna as propriedades do value do Clicado.Provider
export function useClicado() {
  const context = useContext(Clicado);
  return context;
}
```

Figura 12 – Código do arquivo src/hooks/useClicado.ts.

```
import { useClicado } from "../useClicado";

export { useClicado };
```

Figura 13 – Código do arquivo src/hooks/index.ts.

```
const estilos = {
  geral: {
    padding: 5,
```

```
    margin: 4,
    borderWidth: 1,
    borderStyle: "solid",
    borderColor: "black",
  },
  app: { backgroundColor: "thistle" },
  aa: { backgroundColor: "LightSkyBlue" },
  bb: { backgroundColor: "PaleGreen" },
  cc: { backgroundColor: "wheat" },
  dd: { backgroundColor: "LightCyan" },
};

export default estilos;
```

Figura 14 – Código do arquivo src/styles/index.ts.

```
import { Principal } from "../components";
import { ClicadoProvider } from "../contexts";

function App() {
  return (
    <ClicadoProvider>
      <Principal />
    </ClicadoProvider>
  );
}

export default App;
```

Figura 15 – Código do arquivo src/index.tsx.

IX. Ciclo de vida do componente

Cada componente React possui um ciclo de vida que podemos acessar através da função de efeito colateral `useEffect` (<https://react.dev/learn/lifecycle-of-reactive-effects>). O ciclo de vida está vinculado as fases de montagem (mounting – criar o componente), atualização (updating – alterações nas propriedades dos objetos state ou props) e desmontagem (unmounting – excluir o componente):

1. Fase mounting: quando o componente é adicionado na tela;

No exemplo a seguir a função `useEffect` é chamada ao montar o componente A:

```
useEffect( ()=>console.log("Mounting A"), [] );
```

`useEffect` recebe dois parâmetros:

- 1º parâmetro: função call-back a ser executada cada vez que o state sofrer qualquer alteração;
- 2º parâmetro: array com os estados a serem monitorados, qualquer alteração em um dos estados fará a função call-back ser invocada. Porém, como passamos um array vazio, então a função call-back será chamada somente ao criar o componente.

2. Fase updating: o componente é atualizado quando ocorre a alguma modificação nas propriedades do props ou state.

No exemplo a seguir a função `useEffect` é chamada ao atualizar qualquer propriedade do state:

```
useEffect( ()=>console.log('Updating state') );
```

No exemplo a seguir a função `useEffect` é chamada somente ao atualizar a propriedade `idade` do state:

```
useEffect( ()=>console.log('Updating idade'), [idade] );
```

3. Fase unmounting: quando o componente é removido da tela. Para a função `useEffect` ser chamada ao destruir o componente ela precisa retornar uma função e o segundo parâmetro precisa ser um array vazio:

```
useEffect( ()=>{
  return ()=>console.log("Unmounting A");
}, [] );
```

Use o código a seguir para testar o `useEffect`. O resultado será exibido no console do navegador:

```
import { useEffect, useState } from "react";

export default function App() {
  const [nome, setNome] = useState("");
  const [idade, setIdade] = useState("");
  useEffect(() => console.log("Updating idade"), [idade]);
  useEffect(() => console.log("Updating state"));
  return (
    <>
      <div>
        <label htmlFor="nome">Nome</label>
        <input
          id="nome"
          value={nome}
          onChange={(e) => setNome(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="idade">Idade</label>
        <input
          id="idade"
          value={idade}
          onChange={(e) => setIdade(e.target.value)}
        />
      </div>
      {nome !== "" && <A nome={nome} idade={idade} />}
    </>
  );
}

function A(props: { nome: string; idade: string }) {
  useEffect(() => console.log("Mounting A"), []);
  useEffect(() => {
    return () => console.log("Unmounting A");
  }, []);
}
```

```
return (  
  <div>  
    <div>Nome: {props.nome}</div>  
  </div>  
)  
};  
}
```

X. HTTP Request

Uma requisição HTTP em um servidor é formada pelos parâmetros enviados ao servidor e devolvidos pelo servidor. Para fazer requisições HTTP o JS provê a Fetch API que possui o método global fetch (um método global não precisa ser importado).

As requisições na rede são assíncronas, por este motivo o método fetch retorna uma promise (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch).

O exemplo a seguir faz uso do método fetch para fazer uma requisição no webservice da ViaCEP (<http://viacep.com.br>). O webservice retorna em diferentes formatos, mas aqui utilizamos apenas o retorno em JSON, por exemplo, a URL <https://viacep.com.br/ws/12243750/json> retornará

```
{  
  "cep": "12243-750",  
  "logradouro": "Rua Engenheiro Prudente Meireles de Moraes",  
  "complemento": "",  
  "bairro": "Vila Adyana",  
  "localidade": "São José dos Campos",  
  "uf": "SP",  
  "ibge": "3549904",  
  "gia": "6452",  
  "ddd": "12",  
  "siafi": "7099"  
}
```

A URL <https://viacep.com.br/ws/12247099/json/> retornará:

```
{  
  "erro": true  
}
```

Código de exemplo:

```
import { useState } from "react";  
  
export default function App() {  
  const [cep, setCep] = useState("12243750");  
  const [resposta, setResposta] = useState("");  
  
  const url = `https://viacep.com.br/ws/${cep}/json/`;  
  
  const obter = () => {  
    fetch(url)  
      //converte a resposta para JSON  
      .then((response) => response.json())  
      .then((json) => {
```

```

        setResposta(JSON.stringify(json));
    });
};
return (
    <>
        <label>Nome</label>
        <input value={cep} onChange={(e) => setCep(e.target.value)} />
        <button onClick={obter}>Buscar</button>
        <div>{resposta}</div>
    </>
);
}

```

O pacote `axios` (<https://www.npmjs.com/package/axios>) também é usado para processar requisições HTTP. Ele possui vantagens sobre a API `fetch` e uma delas é converter automaticamente a resposta para JSON. A seguir tem-se o código anterior usando `axios`, lembre-se que é necessário adicionar o pacote `axios` como dependência no seu projeto (arquivo `package.json`).

```

import axios from "axios";
import { useState } from "react";

export default function App() {
    const [cep, setCep] = useState("12243750");
    const [resposta, setResposta] = useState("");
    const url = `https://viacep.com.br/ws/${cep}/json/`;

    const obter = () => {
        axios
            .get(url)
            //o conteúdo da resposta da requisição será colocada no objeto data,
            //por este motivo fez-se a desestruturação
            .then(({ data }) => {
                setResposta(JSON.stringify(data));
            });
    };

    return (
        <>
            <label>Nome</label>
            <input value={cep} onChange={(e) => setCep(e.target.value)} />
            <button onClick={obter}>Buscar</button>
            <div>{resposta}</div>
        </>
    );
}

```

Podemos usar a função `useEffect` para chamar a função `obter()` ao montar o componente na tela:

```

import axios from "axios";
import { useEffect, useState } from "react";

```

```
export default function App() {
  const [cep, setCep] = useState("12243750");
  const [resposta, setResposta] = useState("");

  const url = `https://viacep.com.br/ws/${cep}/json/`;

  useEffect(() => {
    if( cep.length === 8 ){
      obter();
    }
  }, []);

  const obter = () => {
    axios
      .get(url)
      .then(({ data }) => {
        setResposta(JSON.stringify(data));
      });
  };

  return (
    <>
      <label>Nome</label>
      <input value={cep} onChange={(e) => setCep(e.target.value)} />
      <button onClick={obter}>Buscar</button>
      <div>{resposta}</div>
    </>
  );
}
```

XI. Gerar a versão de produção da aplicação

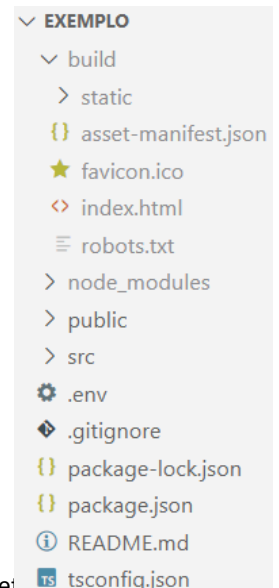
O primeiro passo é executar o comando `npm run build`. Esse comando dispara o comando que está na propriedade `scripts>build` do `package.json`:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

Será criada a pasta `build`, na raiz do projeto, com a compilação de produção do aplicativo. A versão de produção é aquela que o usuário/cliente pode usar no ambiente de trabalho.

O arquivo `build/index.html` não consegue ser executado diretamente no navegador por ser um projeto, então teremos de subir o serviço usando `serve` (<https://www.npmjs.com/package/serve>).

O pacote `serve` é usado para servir sites estáticos, aplicativos de página única ou apenas um arquivo estático.



Primeiramente verifique se você tem a instalação global do pacote `serve`.

Caso não tenha, utilize o comando a seguir para fazer a instalação global:

```
npm i serve -g
```

```
C:\Windows\system32\cmd.exe
C:\>serve -v
14.2.0
```

Para subir a aplicação acesse o local onde se encontra a pasta `build` e digite o comando `serve -s build`. O serviço estará na porta 3000. O parâmetro `-l` indica a porta que queremos usar.

```
C:\Windows\system32\cmd.exe
D:\exemplo>serve -s build

Serving!
- Local:    http://localhost:3000
- Network:  http://192.168.0.101:3000

Copied local address to clipboard!
```

```
C:\Windows\system32\cmd.exe
D:\exemplo>serve -s build -l 3100

Serving!
- Local:    http://localhost:3100
- Network:  http://192.168.0.101:3100

Copied local address to clipboard!
```

XII. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercício 1 - <https://youtu.be/A8tizlh5noM>

Exercício 2 - <https://youtu.be/gAL6M-MqVCc>

Exercício 3 - <https://youtu.be/Xo4Fpxdrha0>

Exercício 4 - https://youtu.be/W88_TvfJaa0

Exercício 5 - https://youtu.be/UnXt6Ju_N_k

Exercício 6 - <https://youtu.be/iqTpZgXdPqE>

Exercício 7 - https://youtu.be/tOcCW_RBT8E

Exercício 8 - <https://youtu.be/mlmiYPRapWE>

Exercício 9 - <https://youtu.be/8Fh361t7sbU>

Exercício 10 - <https://youtu.be/kb07htnPGjM>

Exercício 11 - <https://youtu.be/gstlm5s2QSw>

Exercício 12 - <https://youtu.be/AeZkiMMXu8o>

Exercício 1 – Fazer um aplicativo React TS com a interface mostrada ao lado.

O aplicativo é formado pelo componente App. Ao mudar o campo de entrada o nome será alterado na tela.

Dica: crie uma propriedade no estado do componente e faça a vinculação do campo de entrada com essa propriedade do estado.

localhost:3000

Exercício 1

Nome

Nome: Ana Maria

Exercício 2 – Alterar o aplicativo do Exercício 1 para exibir em uma lista os nomes digitados no campo de entrada.

Dicas:

- Crie uma propriedade no estado do componente para receber o array de nomes;
- Crie uma função para ser chamada ao clicar no botão Salvar. No corpo dessa função adicione o nome no array. Use o operador de spread (espalhamento) para clonar o array e, na sequência, setar na propriedade do estado;
- Use o método map, do objeto Array, para percorrer o array de nomes e construir a lista na tela.

Exercício 2

Nome

1. Pedro
2. Luiza
3. Maria

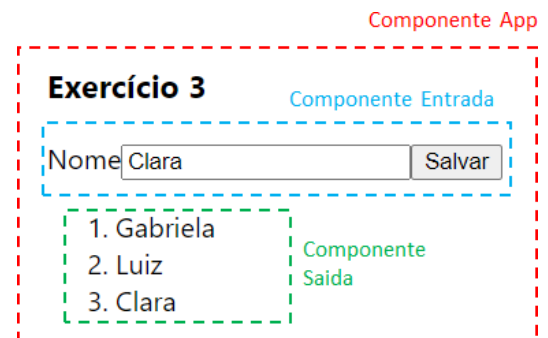
Exercício 3 – Alterar o aplicativo do Exercício 2 para organizar o código nos componentes Entrada e Saida, assim como é mostrado ao lado.

Dicas:

- Mantenha o array com os nomes no componente App;
- Mantenha a função para adicionar os elementos no array no componente App. Vamos supor que essa função chame `add` e receba como parâmetro uma string;
- Crie uma função de nome `Entrada` que receba como parâmetro a função `add`. Ao clicar no botão Salvar será chamada a função `add` passando a propriedade nome;
- Mova a propriedade de estado que possui o nome do campo de entrada para o componente Entrada.
- Crie uma função de nome `Saida` que receba como parâmetro o array de nomes.
- Use as definições de tipo a seguir para os parâmetros de entrada das funções Entrada e Saida:

```
interface SaidaProps {
  lista: string[];
}

interface EntradaProps {
  add: (nome:string) => void
}
```



Exercício 4 – Alterar o aplicativo do Exercício 3 para incluir o campo idade.

Dicas:

- Adicione a propriedade idade no estado do componente Entrada;
- Inclua o parâmetro idade na função add (essa função está no componente App);
- Altere a interface EntradaProps para ter na assinatura a idade


```
interface EntradaProps {
  add: (nome:string,idade:string) => void
}
```
- Crie a definição de tipo Pessoa:


```
interface Pessoa {
  nome:string;
  idade:string;
}
```
- Altere o tipo de dado da propriedade que recebe o array de nomes para `Pessoa[]`. Antes essa propriedade recebia um array de `string[]`:


```
const [lista, setLista] = useState([] as Pessoa[]);
```
- Altere a função chamada ao clicar no botão Salvar. Antes os elementos do array eram strings, agora precisam ser objetos do tipo Pessoa, ou seja, objetos com as propriedades `{nome,idade}`:


```
function add(nome:string, idade:string) {
  const temp = [...lista, {nome,idade}];
  setLista(temp);
}
```
- Altere o tipo SaidaProps para ser um array de `Pessoa[]`:


```
interface SaidaProps {
  lista: Pessoa[];
}
```
- Altere a função call-back recebida pelo método map para exibir a idade na página.

Exercício 4

Nome

Idade

1. Manuela - 21
2. Rodrigo - 25
3. Débora - 20

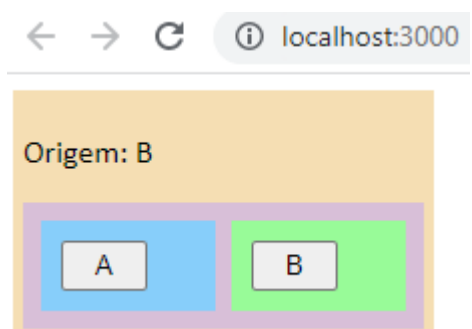
Exercício 5 – Fazer um aplicativo React TS com a interface mostrada ao lado.

O aplicativo é formado pelos componentes App, A e B. O texto “Origem: B” está no componente App e os botões A e B estão nos componentes A e B, respectivamente. Ao clicar em cada botão o texto “Origem: B” será alterado para exibir a origem do botão clicado.

Dicas:

- Coloque os estilos a seguir no arquivo App.css e faça a importação desse arquivo no App.tsx:

```
.body {
  font-family: calibri;
}
```



```

    background-color: wheat;
    padding: 5px;
    display: flex;
    flex-direction: column;
    max-width: 200px;
  }

  .linha {
    background-color: thistle;
    padding: 5px;
    display: flex;
    flex-direction: row;
  }

  button {
    padding: 3px 15px;
  }

  .aa {
    background-color: LightSkyBlue;
    padding: 10px;
    margin: 4px;
    flex: 1;
  }

  .bb {
    background-color: PaleGreen;
    padding: 10px;
    margin: 4px;
    flex: 1;
  }

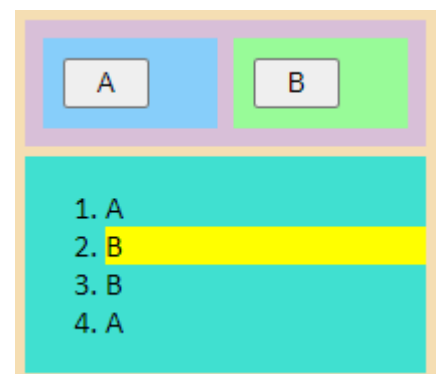
```

- As funções A e B recebem como parâmetro a função que altera a propriedade de estado origem.

Exercício 6 – Alterar o Exercício 5 para os botões clicados serem armazenados num array do componente App e ser exibido em um componente de nome C. Desta forma, o aplicativo terá os componentes App, A, B e C. O item da lista deverá ser removido ao clicar com o botão direito sobre ele.

Dicas:

- Crie uma propriedade de estado no componente App para receber array de origens;
- Crie uma função no componente App para adicionar as origens no array e envie essa função como propriedade para os componentes A e B;
- Crie uma função no componente App para remover os elementos do array e envie essa função como propriedade para o componente C. A 1ª instrução dessa função terá de ser a chamada do método



preventDefault() para evitar que o popup menu seja exibido;

- Use o evento onContextMenu para pegar o botão direito do mouse sobre o item da lista. Esse evento deverá chamar a função que remove o elemento do array;
- Adicione os estilos a seguir no arquivo App.css:

```
.cc {  
  background-color: Turquoise;  
  margin-top: 5px;  
  flex: 1;  
}  
  
li {  
  cursor: pointer;  
}  
  
li:hover {  
  background-color: yellow;  
}
```

Exercício 7 – Alterar o Exercício 6 para passar as propriedades e operações pelo objeto context, ou seja, não é permitido usar o objeto props.

Dicas:

- Crie um contexto usando a função createContext. Esse contexto deverá ser criado e colocado numa variável fora de qualquer componente;
- Envolve todas as marcações do componente App pelo Contexto.Provider. Desta forma, qualquer componente aninhado poderá acessar as propriedades. Coloque na propriedade value do Contexto.Provider o array com as origens e as funções para adicionar e remover elementos do array;
- Em cada componente aninhado poderá ser usada a função useContext para obter as propriedades propagadas pela propriedade value do Contexto.Provider.

Exercício 8 – Alterar o Exercício 4 para passar as propriedades e operações pelo objeto context, ou seja, não é permitido usar o objeto props.

Dicas:

- Crie um contexto usando a função createContext. Esse contexto deverá ser criado e colocado numa variável fora de qualquer componente;
- Envolve todas as marcações do componente App pelo Contexto.Provider. Desta forma, qualquer componente aninhado poderá acessar as propriedades. Coloque na propriedade value do Contexto.Provider o array de {nome,idade} e a função para adicionar e elementos no array;
- Em cada componente aninhado poderá ser usada a função useContext para obter as propriedades propagadas pela propriedade value do Contexto.Provider.

Exercício 9 – Codificar o Exercício 8 usando Hooks.

Dicas:

- Crie uma função para retornar o Contexto.Provider. Migre o array e a função para adicionar elementos no array do componente App para a função criada. Na prática estamos movendo o estado para a função criada;
- Envolve a árvore de componentes do App pela função criada;
- Crie uma função para retornar o contexto. Essa função precisa ter o nome use**, pois todo Hook precisa começar com o prefixo use;
- Nos componentes Entrada e Saida substitua useContext pela função use** criada.

Exercício 10 – Codificar o Exercício 7 usando Hooks.

Exercício 11 – O IBGE possui uma API para disponibilização de dados de localidades do país. Nela é possível listar as regiões, estados e mesorregiões do país (<https://servicodados.ibge.gov.br/api/docs/localidades>). Fazer um aplicativo React TS para fazer uma requisição na API do IBGE para obter as regiões do país e carregar o resultado numa lista ordenada, assim como é mostrado ao lado.

1. Centro-Oeste
2. Nordeste
3. Norte
4. Sudeste
5. Sul

Dicas:

- Instale o pacote axios no projeto (npm i axios);
- Como a lista de regiões deverá ser carregada ao criar o componente APP, então coloque a requisição no corpo da função de efeito colateral useEffect. Para a função useEffect ser chamada ao construir o componente tem-se de passar dois parâmetros para o useEffect, sendo o 1º uma função call-back e o 2º um array vazio (assim como é mostrado a seguir). No corpo da função call-back coloca-se a requisição na URL.

```
useEffect( ()=>console.log("Mounting A"), [] );
```

- Faça uma requisição na URL <https://servicodados.ibge.gov.br/api/v1/localidades/regioes?orderBy=nomelistaar>;
- Coloque o resultado da requisição numa variável de estado.

Exercício 12 – Alterar o Exercício 11 para o usuário clicar sobre o nome de uma região e serem listados os estados dessa região.

Dicas:

- Crie uma função que recebe o **id** da região e faz uma requisição na URL [https://servicodados.ibge.gov.br/api/v1/localidades/regioes/\\$id/estados?orderBy=nome](https://servicodados.ibge.gov.br/api/v1/localidades/regioes/$id/estados?orderBy=nome) para obter os estados da região. Observe que o **id** está envolvido pela expressão **\${}**, desta forma, essa URL terá de ser envolvida pelos template literais ``;
- Adicione o evento onclick nos elementos . Esse evento deverá chamar a função criada para obter os estados da região;
- Guarde a lista de estados obtida numa variável de estado.

Regiões

1. Norte
2. Nordeste
3. Sudeste
4. Sul
5. Centro-Oeste

UF

1. Paraná
2. Rio Grande do Sul
3. Santa Catarina