Objetivos:

I. React context;

II. Hook customizado.

I. React context

Enquanto, por props, as propriedades precisam ser passadas para cada componente aninhado, via context, as propriedades podem ser disponibilizadas para toda a árvore de componentes. Isso evita que componentes intermediários precisem passar explicitamente props entre cada nível.

O Context é recomendado para compartilhar dados que podem ser considerados "globais" para a árvore de componentes do React, por exemplo, o token de autenticação do usuário.

O Context é composto por três partes principais:

1. createContext: cria um objeto Context que contém duas propriedades principais: `Provider` e `Consumer`. No exemplo a seguir a variável `CountContext` recebeu um objeto do tipo `Context`, que pode conter os valores definidos na interface `ContextProps` ou ser `null`:

   ```
   const CountContext = createContext<ContextProps | null>(null);
   ```

2. Provider: permite definir os valores que estarão disponíveis para os componentes descendentes. No exemplo a seguir, a função `CountProvider` define os valores que o `Provider` irá propagar pela hierarquia de componentes. O conteúdo propagado está na propriedade `value`, que obrigatoriamente precisa conter as propriedades definidas na interface `ContextProps`:

   ```
   function CountProvider({children}:ChildrenProps) {
     const [count, setCount] = useState(0);

     function increment(){
       setCount((prev) => prev + 1);
     }

     function decrement(){
       setCount((prev) => prev - 1);
     }

     return (
       <CountContext.Provider value={{count, increment, decrement}}>
         {children}
       </CountContext.Provider>
     );
   }
   ```

   Escopo do Provider: a função `CountProvider` define o componente que usamos para delimitar o escopo do `Provider`. No exemplo a seguir, o corpo da tag `<CountProvider>` define o `{children}`. Desta

forma, os componentes A, B e C podem acessar as propriedades do contexto que foram definidas no CountProvider.

```
export default function App() {
  return (
    <CountProvider>
      <A />
      <div>
        <B />
        <C />
      </div>
    </CountProvider>
  )
}
```

3. Consumer (ou o Hook useContext): o Hook useContext permite que os componentes filhos acessem os valores fornecidos pelo Provider. No exemplo a seguir, através do CountContext, passado para useContext, temos acesso às propriedades propagadas pelo Provider.

```
function A(){
  const {count} = useContext(CountContext) as ContextProps;

  return <h3>Valor: {count}</h3>;
}
```

No código a seguir as propriedades count, increment e decrement são propagadas para todos os componentes usando Context.

```
import { createContext, useContext, useState } from 'react'

export default function App() {
  return (
    <CountProvider>
      <A />
      <div>
        <B />
        <C />
      </div>
    </CountProvider>
  )
}

interface ContextProps {
  count: number;
  increment: () => void;
  decrement: () => void;
}

interface ChildrenProps {
```

```
  children: React.ReactNode;
}

const CountContext = createContext<ContextProps | null>(null);

function CountProvider({children}:ChildrenProps) {
  const [count, setCount] = useState(0);

  function increment(){
    setCount((prev) => prev + 1);
  }

  function decrement(){
    setCount((prev) => prev - 1);
  }

  return (
    <CountContext.Provider value={{count, increment, decrement}}>
      {children}
    </CountContext.Provider>
  );
}

function A(){
  const {count} = useContext(CountContext) as ContextProps;

  return <h3>Valor: {count}</h3>;
}

function B(){
  const {increment} = useContext(CountContext) as ContextProps;

  return <button onClick={increment}>Aumentar</button>;
}

function C(){
  const {decrement} = useContext(CountContext) as ContextProps;

  return <button onClick={decrement}>Diminuir</button>;
}
```

Observação: o contexto não pode ser acessado fora do escopo do Provider. No exemplo a seguir, a chamada do componente `<A />`, que está fora do `<CountProvider>` resultará em um erro, pois não terá acesso às propriedades fornecidas pelo Contexto.

```
export default function App() {
  return (
    <>
      <CountProvider>
```

```
        <A />
        <div>
          <B />
          <C />
        </div>
      </CountProvider>
      <A /> {/* este componente A está fora do escopo do Provider */}
    </>
  );
}
```

II. Hook customizado

No React, Hooks são funções que permitem utilizar funcionalidades essenciais, como estado (`useState`) e contexto (`useContext`), em componentes funcionais. Um Hook customizado é uma função criada pelo desenvolvedor que utiliza um ou mais hooks nativos do React para encapsular lógica reutilizável.

Além de encapsular lógica comum, hooks customizados também podem ser usados com a Context API para garantir que a lógica de estado global seja acessada corretamente. Eles asseguram que o `useContext` seja sempre chamado dentro de um provedor válido, evitando erros comuns de acesso fora do escopo adequado.

No exemplo a seguir, definimos o Hook `useCountContext` para encapsular a lógica de acesso ao contexto `useContext(CountContext)`. Dessa forma, o acesso ao contexto nos componentes aninhados passou a ser realizado por meio do Hook `useCountContext()`, proporcionando uma abordagem mais segura e legível.

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

export default function App() {
  return (
    <CountProvider>
      <A />
      <div>
        <B />
        <C />
      </div>
    </CountProvider>
  );
}

interface ContextProps {
  count: number;
  increment: () => void;
  decrement: () => void;
}

interface ChildrenProps {
  children: React.ReactNode;
}
```

```
const CountContext = createContext<ContextProps | null>(null);

function CountProvider({ children }: ChildrenProps) {
  const [count, setCount] = useState(0);

  function increment(){
    setCount((prev) => prev + 1);
  }

  function decrement(){
    setCount((prev) => prev - 1);
  }

  return (
    <CountContext.Provider value={{ count, increment, decrement }}>
      {children}
    </CountContext.Provider>
  );
}

// Hook customizado para consumir o contexto
function useCountContext() {
  const context = useContext(CountContext);
  if (!context) {
    throw new Error("useCountContext deve ser usado dentro de um CountProvider");
  }
  return context;
}

function A() {
  const { count } = useCountContext();

  return <h3>Valor: {count}</h3>;
}

function B() {
  const { increment } = useCountContext();

  return <button onClick={increment}>Aumentar</button>;
}

function C() {
  const { decrement } = useCountContext();

  return <button onClick={decrement}>Diminuir</button>;
}
```

Benefícios do Hook customizado useCountContext:

- Encapsulamento seguro: evita erros de acesso ao contexto fora do provedor;
- Reutilização de lógica: simplifica o consumo do contexto em vários componentes;
- Legibilidade: reduz complexidade nos componentes consumidores;
- Validação explícita: aponta claramente quando o hook é utilizado de forma incorreta.

III. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercício 1 - https://youtu.be/CL6uen3Ehxo

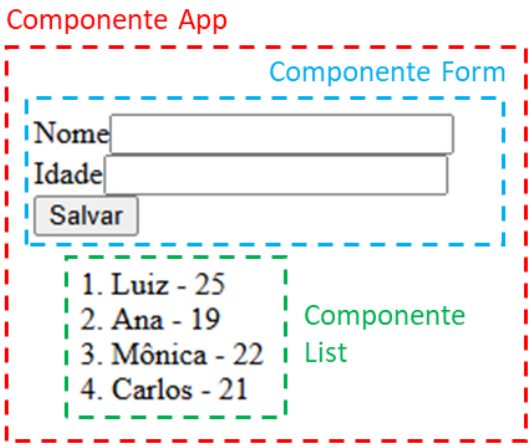
Exercício 2 - https://youtu.be/a1xoIpQNWHk

Exercício 3 - https://youtu.be/-5gENXugxQ0

Exercício 4 - https://youtu.be/OkTcEm0P5TI

Exercício 5 - https://youtu.be/Mq7zC4T2QtI

Exercício 1 – Complete o código a seguir para construir um aplicativo React para cadastrar usuários.



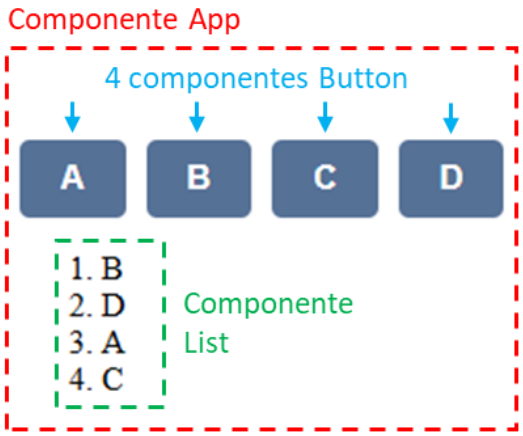
Requisitos:

1. O componente `Form` deve apenas capturar as entradas do usuário e chamar a função `add` do contexto para adicionar um novo usuário;
2. O componente `List` deve exibir o array de usuários armazenado no contexto;
3. Ao clicar com o botão direito sobre um item da lista, o respectivo item deve ser removido utilizando a função `remove` do contexto;
4. As propriedades do contexto devem ser acessadas nos componentes `Form` e `List` por meio de um Hook customizado.

```
interface ContextProps {
  users: User[];
  add: (user: User) => void;
  remove: (index: number) => void;
}

interface User {
  name: string;
  age: string;
}

interface ChildrenProps {
```

```
  children: React.ReactNode;
}

const UserContext = createContext<ContextProps | null>(null);

function UserProvider({ children }: ChildrenProps) {
  const [users, setUsers] = useState<User[]>([]);

  function add(user: User) {
    setUsers((prev) => [...prev, user]);
  }

  function remove(index: number) {
    setUsers(users.filter((_, i) => i !== index));
  }

  return (
    <UserContext.Provider value={{ users, add, remove }}>
      {children}
    </UserContext.Provider>
  );
}
```

Exercício 2 – Construir um aplicativo React para listar os cliques nos botões.



Requisitos:

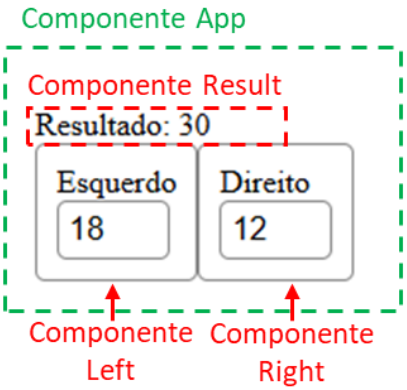
1. O contexto deve ter um array de strings para receber os rótulos dos botões clicados;
2. O contexto deve ter uma função de nome add para adicionar os rótulos no array;
3. O componente Button deve receber o rótulo por props e deve chamar a função add, do contexto, ao receber o clique;
4. O componente List deve exibir o array de cliques armazenado no contexto;
5. As propriedades do contexto devem ser acessadas nos componentes Button e List por meio de um Hook customizado.

Dica: utilize o código a seguir para estilizar o elemento <button> do componente Button.

```
const buttonStyle: CSSProperties = {
  backgroundColor: "#567196",
  color: "white",
  padding: "10px 20px",
  marginRight: "10px",
```

```
  fontSize: "16px",
  fontWeight: "bold",
  border: "none",
  borderRadius: "5px",
  cursor: "pointer",
};
```

Exercício 3 – Complete o código a seguir para construir um aplicativo React para somar dois números.



Requisitos:

1. A aplicação deve ser formada pelos componentes `Left`, `Right` e `Result`;
2. O usuário fornece números nos campos dos componentes `Left` e `Right`;
3. O componente `Result` exibe o resultado da soma dos valores fornecidos nos campos de entrada dos componentes `Left` e `Right`. O valor deverá ser exibido somente se ambos os campos tiverem valores numéricos;
4. As propriedades do contexto devem ser acessadas nos componentes `Left`, `Right` e `Result` por meio de um Hook customizado.

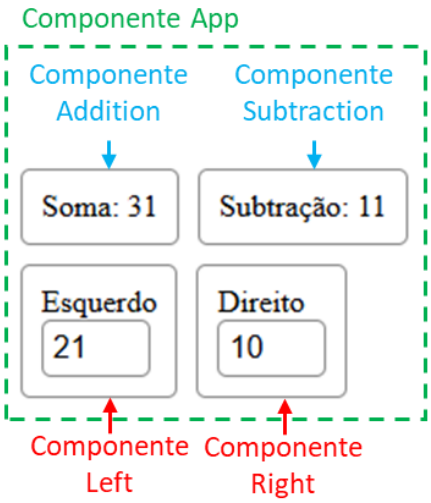
```
interface ContextProps {
  left: string;
  right: string;
  setLeft: (value: string) => void;
  setRight: (value: string) => void;
}

interface ChildrenProps {
  children: React.ReactNode;
}

const SumContext = createContext<ContextProps | null>(null);

function SumProvider({ children }: ChildrenProps) {
  const [left, setLeft] = useState("");
  const [right, setRight] = useState("");

  return (
    <SumContext.Provider value={{ left, right, setLeft, setRight }}>
      {children}
    </SumContext.Provider>
  );
```

```
}
```

Dica: utilize o código a seguir para estilizar o elemento `<input>` e a borda dos componentes `Left` e `Right`.

```
const wrapperStyle: CSSProperties = {
  display: "inline-flex",
  flexDirection: "column",
  border: "1px solid #888",
  borderRadius: "5px",
  padding: "10px",
};

const inputStyle: CSSProperties = {
  padding: "5px",
  fontSize: "16px",
  border: "1px solid #888",
  borderRadius: "5px",
  width: "45px",
};
```

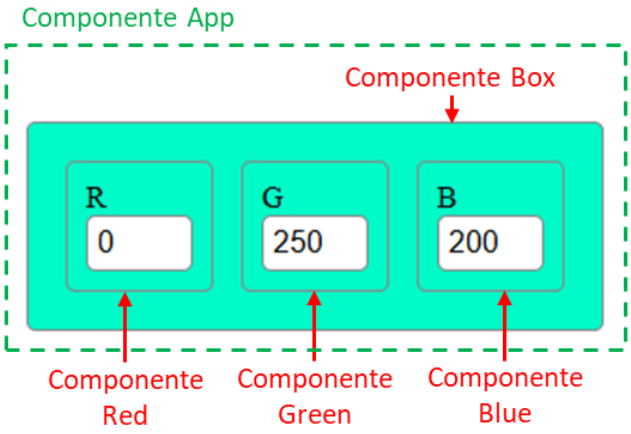
Exercício 4 – Altere o aplicativo do Exercício 3 para ter os componentes `Addition` e `Subtraction`, onde antes existia somente o componente `Result`.



Exercício 5 – Complete o código a seguir para construir um aplicativo React para compor cores RGB.

Requisitos:

1. A aplicação deve ser formada pelos componentes `Red`, `Green` e `Blue`. Cada componente tem um campo de entrada para o usuário fornecer um valor entre 0 e 255;
2. A cor de fundo do componente `Box` deve ser a



composição do valor de cada campo de entrada;

3. Use os estilos definidos nas variáveis `boxStyle` (componente `Box`), `wrapperStyle` e `inputStyle` (componentes `Red`, `Green` e `Blue`);
4. As propriedades do contexto devem ser acessadas nos componentes `Red`, `Green`, `Blue` e `Box` por meio de um Hook customizado.

Dica: use o código a seguir no componente Box para obter a cor dinamicamente:

```
const {getRGB} = useRGBContext();
const dynamicBoxStyle: CSSProperties = {
  ...boxStyle,
  backgroundColor: getRGB(), // define o fundo dinamicamente
};
```

```
interface ContextProps {
  r: string;
  g: string;
  b: string;
  setR: (value: string) => void;
  setG: (value: string) => void;
  setB: (value: string) => void;
  getRGB: () => string;
}

interface ChildrenProps {
  children: React.ReactNode;
}

const RGBContext = createContext<ContextProps | null>(null);

function RGBProvider({ children }: ChildrenProps) {
  const [r, setR] = useState("");
  const [g, setG] = useState("");
  const [b, setB] = useState("");

  function getRGB(){
    return `rgb(${r || 0}, ${g || 0}, ${b || 0})`;
  }

  return (
    <RGBContext.Provider value={{ r, g, b, setR, setG, setB, getRGB }}>
      {children}
    </RGBContext.Provider>
  );
}

const boxStyle: CSSProperties = {
  display: "inline-flex",
```

```
  flexDirection: "row",
  border: "1px solid #888",
  borderRadius: "5px",
  padding: "20px",
  gap: "15px"
};

const wrapperStyle: CSSProperties = {
  display: "inline-flex",
  flexDirection: "column",
  border: "1px solid #888",
  borderRadius: "5px",
  padding: "10px",
};

const inputStyle: CSSProperties = {
  padding: "5px",
  fontSize: "16px",
  border: "1px solid #888",
  borderRadius: "5px",
  width: "45px",
};
```