useEffect: permite conectar um componente a um sistema externo, como uma API, o DOM do navegador, uma animação ou um widget2. O useEffect recebe uma função que é executada após cada renderização do componente, e pode retornar outra função que é executada antes da próxima renderização. O useEffect também aceita um array de dependências como segundo argumento, que indica quando o efeito deve ser executado novamente. Por exemplo:

```
function ListaDeRepositorios() {
  useEffect(() => {
    async function carregaRepositorios() {
      const resposta = await fetch(
        "https://api.github.com/users/julio-cesar96/repos"
      );
      const repositorios = await resposta.json();
      return repositorios;
    }

    carregaRepositorios();
  }, []);

  return <> ... </>;
}
```

useMemo: permite guardar o resultado de um cálculo caro, evitando repeti-lo desnecessariamente em cada renderização2. O useMemo recebe uma função que retorna o valor a ser memorizado, e um array de dependências que indica quando o valor deve ser recalculado. O useMemo retorna o valor memorizado. Por exemplo:

```
function Fibonacci({ n }) {
  const fib = useMemo(() => {
    function f(n) {
      if (n < 2) return n;
      return f(n - 1) + f(n - 2);
    }
    return f(n);
  }, [n]);
```

```
  return <div>O termo {n} da sequência de Fibonacci é {fib}</div>;
}
```

useCallback: permite guardar a definição de uma função, evitando recriá-la em cada renderização2. O useCallback recebe uma função e um array de dependências que indica quando a função deve ser redefinida. O useCallback retorna a função guardada. Isso é útil para passar funções como props para componentes otimizados, como os que usam React.memo. Por exemplo:

```
function Contador() {
  const [contagem, setContagem] = useState(0);

  const incrementa = useCallback(() => {
    setContagem((c) => c + 1);
  }, []);

  return (
    <>
      <div>A contagem está em {contagem}</div>
      <Botao onClick={incrementa}>Incrementar</Botao>
    </>
  );
}

const Botao = React.memo(function Botao({ onClick, children }) {
  console.log("Renderizando botão");
  return <button onClick={onClick}>{children}</button>;
});
```

useState: permite declarar uma variável de estado que pode ser atualizada diretamente2. O useState recebe um valor inicial e retorna um array com dois elementos: o valor atual do estado e uma função para atualizá-lo. Essa função pode receber um novo valor ou uma função que recebe o valor anterior e retorna o novo valor. Por exemplo:

```
function Contador() {
  const [contagem, setContagem] = useState(0);
```

```
  function incrementa() {
    setContagem((c) => c + 1);
  }

  function zera() {
    setContagem(0);
  }

  return (
    <>
      <div>A contagem está em {contagem}</div>
      <button onClick={incrementa}>Incrementar</button>
      <button onClick={zera}>Zerar</button>
    </>
  );
}
```

Hooks criados pelo desenvolvedor: além dos hooks integrados, é possível criar seus próprios hooks personalizados, combinando os hooks existentes e outras lógicas3. Os hooks personalizados devem começar com a palavra “use” e seguir as regras dos hooks. Por exemplo:

```
function useDados(url) {
  const [dados, setDados] = useState(null);
  const [erro, setErro] = useState(null);
  const [carregando, setCarregando] = useState(false);

  useEffect(() => {
    setCarregando(true);
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        setDados(data);
        setErro(null);
      })
```

```
      .catch((err) => {
        setDados(null);
        setErro(err);
      })
      .finally(() => {
        setCarregando(false);
      });
  }, [url]);

  return { dados, erro, carregando };
}
function ListaDeUsuarios() {
  const { dados, erro, carregando } = useDados(
    "https://jsonplaceholder.typicode.com/users"
  );

  if (carregando) return <div>Carregando...</div>;
  if (erro) return <div>Erro: {erro.message}</div>;
  if (!dados) return null;

  return (
    <ul>
      {dados.map((usuario) => (
        <li key={usuario.id}>{usuario.name}</li>
      ))}
    </ul>
  );
}
```