

**Objetivo:**

- I. State;
- II. Função de atualização.

**I. State**

No React, state (estado) é um objeto que armazena dados ou informações dinâmicas que podem ser alteradas ao longo do tempo durante o ciclo de vida do componente. Ele permite que um componente gerencie e rastreie suas próprias mudanças de estado e, quando esses dados mudam, o React re-renderiza automaticamente o componente para refletir essas alterações na interface do usuário.

A seguir tem-se um exemplo de código que define um estado, a cada chamada da função `setCount` o estado do componente é alterado e a interface de usuário é re-renderizada.

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

export default function App() {
  return <Contador />;
}

function Contador() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
      <button onClick={() => setCount(0)}>Resetar</button>
      <span>Contador: {count}</span>
    </div>
  );
}
```

O `useState` é um hook do React utilizado para adicionar estado ao componente. Ele recebe o valor inicial da propriedade, no exemplo anterior, o estado foi inicializado com o valor `0`. O `useState` retorna um array com dois elementos:

1. O estado atual: variável que contém o valor atual do estado;
2. Uma função de atualização: função que permite atualizar o valor desse estado.

O normal é desestruturarmos o retorno do `useState` usando um array com duas variáveis, por exemplo, `[count, setCount]`. Embora seja menos legível, podemos receber em uma variável (ex. variável `resposta`) e acessar os elementos usando índices de posição do array:

```
const resposta = useState(0);
const count = resposta[0];
const setCount = resposta[1];
```

Observações:

- O state é inicializado usando o hook `useState`;
- O valor inicial do estado pode ser qualquer tipo — string, number, boolean, array, objeto ou até null;
- Sempre que o state de um componente é alterado, o React re-renderiza automaticamente esse componente para refletir as mudanças;
- As atualizações no state podem ser assíncronas, ou seja, o React pode esperar mais de uma mudança no estado para atualizar o objeto state. Então é importante não depender de seu valor imediatamente após uma mudança;
- O estado não pode ser alterado sem a função de atualização. Por exemplo, a atribuição de valor `count = 100` é considerada inválida, o correto é usar a função de atualização `setCount(100)`.

O estado pode receber valores de qualquer tipo, inclusive tipos complexos como objetos e arrays. No exemplo a seguir o estado recebe um array de números que podem ter novos valores adicionados ou retirados.

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

export default function App() {
  return <Contador />;
}

function Contador() {
  const [nros, setNros] = useState<number[]>([]);

  function add() {
    const nro = aleatorio();
    setNros([...nros, nro]);
  }

  function remove() {
    // cria uma cópia do array
    const copy = [...nros];
    copy.pop(); // retira o último elemento
    setNros(copy);
  }

  return (
    <>
      <div>
        <button onClick={add}>Empilhar</button>
        <button onClick={remove}>Retirar</button>
      </div>
      <ul>
        {nros.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </>
  );
}
```

```

    </>
  );
}

function aleatorio() {
  //retorna um número aleatório entre 0 e 19
  return Math.floor(Math.random() * 20);
}

```

Observações:

- Como o Hook `useState` recebe um array vazio, o TS exige que o tipo do estado (`nros`) seja explicitamente definido como um array de números, utilizando a sintaxe de generic entre os colchetes angulares `<>`:

```
const [nros, setNros] = useState<number[]>([]);
```

Outra opção é usar a asserção de tipo (`as number[]`) para atribuir um tipo diretamente ao valor inicial (`[]`):

```
const [nros, setNros] = useState([] as number[]);
```

Prefira sempre a primeira abordagem (`useState<number[]>([])`) porque é mais legível, explícita e alinhada às boas práticas do TS.

- Como o conteúdo da variável `nros` não pode ser alterado, então, na função `add` tivemos de usar o operador `spread` para criar uma cópia do array e incluir um elemento:
  - `[...nros, nro]`: cria um array, copiando os números existentes (`...nros`) e adiciona o novo número (`nro`) ao final;
  - `setNros` atualiza o estado com o novo array.
- Da mesma forma, fizemos no método `remove`,
  - `[...nros]`: cria uma cópia do array original;
  - `copy.pop()`: remove o último elemento do array copiado;
  - `setNros(copy)`: atualiza o estado com o array modificado.

No React, a ligação entre o campo de entrada (`<input>`) e o estado (`state`) é frequentemente referida como "componente controlado" (controlled component). Isso significa que o valor exibido no campo de entrada é controlado pelo estado do React, garantindo sincronização entre a interface do usuário e os dados armazenados no estado. No exemplo a seguir os estados `name` e `mail` armazenam os valores dos campos de entrada.

```

import { useState } from "react";

export default function App() {
  return <Form />;
}

```

```
interface Person {
  name: string;
  mail: string;
}

function Form() {
  const [user, setUser] = useState<Person>();
  const [name, setName] = useState("");
  const [mail, setMail] = useState("");

  function handleName(e: React.ChangeEvent<HTMLInputElement>) {
    setName(e.target.value);
  }

  function handleMail(e: React.ChangeEvent<HTMLInputElement>) {
    setMail(e.target.value);
  }

  function save(){
    setUser({name, mail});
  }

  return (
    <>
      <div>
        <label htmlFor="name">Nome</label>
        <input id="name" value={name} onChange={handleName} />
      </div>
      <div>
        <label htmlFor="mail">E-mail</label>
        <input id="mail" value={mail} onChange={handleMail} />
      </div>
      <div>
        <button onClick={save}>Salvar</button>
      </div>
      <div>
        Resultado: {JSON.stringify(user)}
      </div>
    </>
  );
}
```

O valor do campo de entrada (**value**) é definido diretamente pelo estado. Isso significa que o campo exibe exatamente o que está no estado:

```
<input id="name" value={name} onChange={handleName} />
```

A cada alteração no campo de entrada, o evento **onChange** é disparado. A função **handleName** é chamada, recebendo o evento como parâmetro. A função **handleName** atualiza o estado com o novo valor digitado.

```
function handleName(e: React.ChangeEvent<HTMLInputElement>) {
  setName(e.target.value);
}
```

Explicação:

- o argumento `e` representa um evento de mudança (change event) que é disparado sempre que o valor de um campo de entrada é alterado. Como o evento é gerado especificamente por um elemento `<input>` do DOM, então usamos o tipo genérico `React.ChangeEvent<HTMLInputElement>`;
- `e.target` representa o elemento HTML que disparou o evento;
- `e.target.value` captura o valor digitado no campo. Sempre retorna uma string, pois o valor dos campos de entrada (`<input>`) é do tipo string por padrão;
- `setName(e.target.value)` atualiza o estado com o novo valor.

A interface `Person` é usada para definir a estrutura do objeto que será armazenado no estado `user` do componente. A interface atua como um contrato de tipos, garantindo que qualquer objeto colocado no estado `user` tenha exatamente as propriedades `name` e `mail` do tipo `string`.

```
const [user, setUser] = useState<Person>();
```

O tipo genérico `<Person>` garante que o estado `user` só aceitará valores que respeitem a estrutura definida pela interface `Person`.

## II. Função de atualização

O estado nem sempre estará sincronizado corretamente em renderizações consecutivas, porque o React pode "agrupar" chamadas ao `setState` ou até mesmo descartar algumas atualizações. A seguir, no exemplo da esquerda, a função `setCount` é chamada duas vezes na sequência (`setCount(count+1)`). Porém, ambas as chamadas usarão o mesmo valor antigo de `count`.

No exemplo da direita, passamos como argumento a função de atualização `(prev) => prev+1`. O React garante que a função receberá o estado mais recente, mesmo quando há múltiplas chamadas consecutivas para `setCount`. Isso torna a atualização do estado determinística e segura.

Para cada clique no botão será somado 1.	Para cada clique no botão será somado 2.
<pre>import { useState } from "react";  export default function App() {   return &lt;Contador /&gt;; }  function Contador() {   const [count, setCount] = useState(0);</pre>	<pre>import { useState } from "react";  export default function App() {   return &lt;Contador /&gt;; }  function Contador() {   const [count, setCount] = useState(0);</pre>

<pre>function increment(){   setCount(count+1);   setCount(count+1); }  return (   &lt;div&gt;     &lt;button onClick={increment}&gt;       Incrementar     &lt;/button&gt;     &lt;span&gt;Contador: {count}&lt;/span&gt;   &lt;/div&gt; ); }</pre>	<pre>function increment(){   setCount((prev) =&gt; prev+1);   setCount((prev) =&gt; prev+1); } return (   &lt;div&gt;     &lt;button onClick={increment}&gt;       Incrementar     &lt;/button&gt;     &lt;span&gt;Contador: {count}&lt;/span&gt;   &lt;/div&gt; ); }</pre>
--	---

Uma função de atualização no React é uma função que passamos para o `setState` ou para funções equivalentes de atualização de estado, como o `setCount` no `useState`. Ela recebe como argumento o valor anterior do estado para calcular o próximo valor. Essa abordagem é particularmente útil quando a nova atualização depende do valor do estado anterior, garantindo que o React use o valor mais recente do estado, mesmo que múltiplas atualizações sejam feitas de forma assíncrona ou em sequência.

### III. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercícios 1 e 2 - <https://youtu.be/cLO1g-fYb18>

Exercícios 3 a 6 - <https://youtu.be/bxHI2J-FO30>

**Exercício 1** – Fazer um aplicativo React TS com a interface mostrada ao lado. O aplicativo deverá ler do usuário o nome e idade e salvar em um array ao clicar no botão Salvar. Ao salvar os campos devem ser limpos e o conteúdo do array deverá ser exibido em uma lista.

Dicas:

- Defina o tipo usando a seguinte interface:

```
interface Person {
  name: string;
  age: string;
}
```

- Use a função de atualização juntamente com o spread operator para criar uma cópia do array e adicionar um elemento no final do array;
- Use o método `map` do array para percorrer os elementos do array.

Nome

Idade

- Carlos - 21
- Ana - 19
- Mônica - 22
- Luiz - 25

**Exercício 2** – Altere o aplicativo do Exercício 1 para exibir o resultado em uma tabela. Além disso, ao clicar no botão Salvar, o elemento deve ser colocado no início do array (no exemplo ao lado, Pedro foi o último elemento a ser adicionado no array).

Dicas:

- Para colocar o elemento no início do array, mude a ordem ao combinar o array copiado com o spread operator e o objeto criado;
- Use as variáveis `table` e `cell` a seguir para formatar os elementos `<table>`, `<th>` e `<td>`. Observação: foi necessário fazer uma tipagem explícita nas variáveis `table` e `cell` para evitar que o TS interpretasse `borderCollapse` como uma string genérica.

```
const table: CSSProperties = {
  tableLayout: "auto", // a largura da tabela se ajusta ao conteúdo
  borderCollapse: "collapse",
  marginTop: "20px",
};

const cell: CSSProperties = {
  border: "1px solid black",
  padding: "5px",
  whiteSpace: "nowrap", // garante que o conteúdo não quebre em várias linhas
};
```

Nome

Idade

Nome	Idade
Pedro	23
Ana	21
Catarina	25
Grabriel	22

**Exercício 3** – Altere o aplicativo do Exercício 2 para ao clicar sobre uma linha da tabela o elemento correspondente ser excluído do estado.

Dicas:

- Inclua o evento `onClick` em cada linha do corpo da tabela para capturar os eventos de clique com o botão esquerdo do mouse;
- O evento `onClick` deve chamar uma função que recebe como parâmetro o índice do elemento no array que está no estado do componente.

**Exercício 4** – Altere o aplicativo do Exercício 3 para reconhecer o clique com o botão direito do mouse.

Dicas:

- Utilize o evento `onContextMenu` para capturar cliques com o botão direito do mouse;
- O evento `onContextMenu` faz exibir o menu de contexto padrão do navegador, então precisamos chamar o método `e.preventDefault()` do evento, e ao chamar a função manuseadora teremos de passar o evento (variável `e`) e o índice de posição no array.

**Exercício 5** – Altere o aplicativo do Exercício 4 para que a idade seja do tipo `number`.

```
interface Person {
  name: string;
  age: number;
}
```

Dicas:

- Altere o estado `age` para permitir um conteúdo `""` ou um número. Utilize `useState<number | "">()`;
- Na função manuseadora do evento `onChange` converta o valor do campo para `Number` ou mantenha como `""` se estiver vazio:

```
const value = e.target.value;
setAge(value === "" ? "" : Number(value));
```

- Adicione as propriedades `type="number"` e `min="0"` no elemento `<input>` para garantir a entrada de números positivos.

**Exercício 6** – Altere o aplicativo do Exercício 5 para ter um campo para selecionar o gênero.

Dicas:

- Adicione a propriedade `gender` na interface `Person`;
- Adicione no estado a propriedade `gender`;
- Crie um campo de seleção e faça a ligação do `value` com o estado `gender`;
- Crie uma função manuseadora do evento `onChange` do campo de seleção para atualizar o estado.

Nome

Idade

Gênero Masculino ▼

Salvar

Nome	Idade	Gênero
Pedro	23	Masculino
Ana	21	Feminino
Catarina	25	Feminino
Gabriel	22	Masculino