



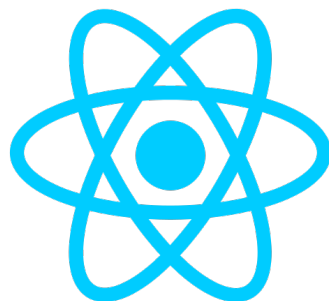
**INSTITUTO FEDERAL**  
Sul-rio-grandense

Câmpus  
Pelotas



# Hook useRef e Introdução a Context API

Referência a valores ou objetos entre renderizações, criação e gerenciamento de estados globais com a API de Contexto do ReactJS



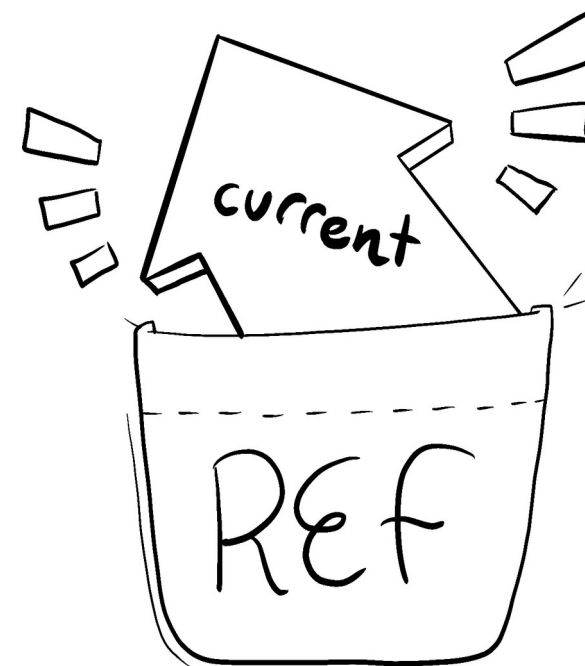
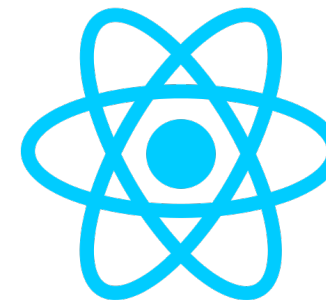
**Prof. Gill Velleda Gonzales** | [gillgonzales@ifsul.edu.br](mailto:gillgonzales@ifsul.edu.br) | [@g1ll](https://twitter.com/g1ll)  
CSTSI - Novembro - 2024

# Desenvolvimento Front-End II

## Tópicos



- **Hook useRef:**
  - Conceito e exemplos
  - Uso em campos *input*
- **Contextos**
  - Criação com *createContext*
  - Componente **Provedor do Contexto**
  - Uso com *useContext*
  - Exemplos de introdução
    - Exercícios
- **Refatoração do todoList App**
  - Substituição de *props drilling* por contexto
- **Atividades**

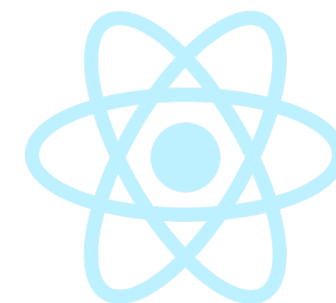
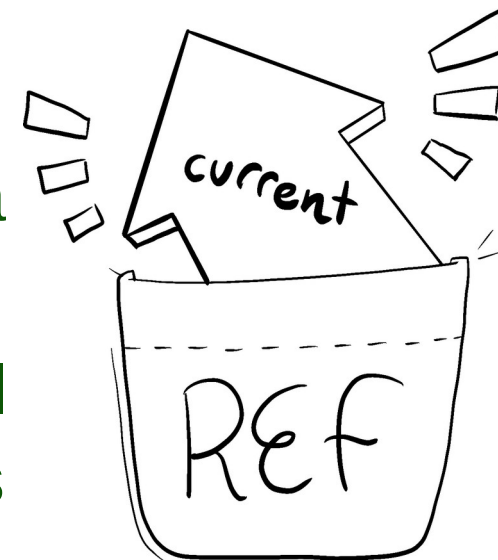


# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



- O hook **`useRef`** oferece um mecanismo para guardar informações de variáveis entre renderizações.
- Diferente do **`useState`**, o componente não será renderizado a cada modificação de uma variável criada com o **`useRef`**.
- A ideia é exatamente guardar a **referência** (**`useRef`**) à variável da renderização anterior, além de evitar renderizações desnecessárias.
- O objeto criado com o hook **`useRef`** na verdade possui uma estrutura específica com o atributo **`current`**, onde realmente é armazenado o valor que foi inicializado ou alterado e que será lembrado entre as renderizações.
- **[Veja a Documentação!](#)**



# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



- Neste exemplo, o contador de renderizações é criado com o **`useRef()`**.
- A cada renderização, ou seja, ao pressionar o botão do contador (**`count`**) atualizaremos o valor da propriedade **`current`** da variável de referência (**`countRef`**).
- **Exemplo 01**: (veja no console o valor de **`countRef.current`**)

```
6  function App() {  
7      const [count, setCount] = useState(0);  
8      const countRef = useRef(0);  
9      console.log({ render: (countRef.current += 1) });  
10     return (  
11         <>
```



# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



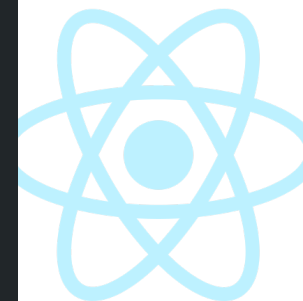
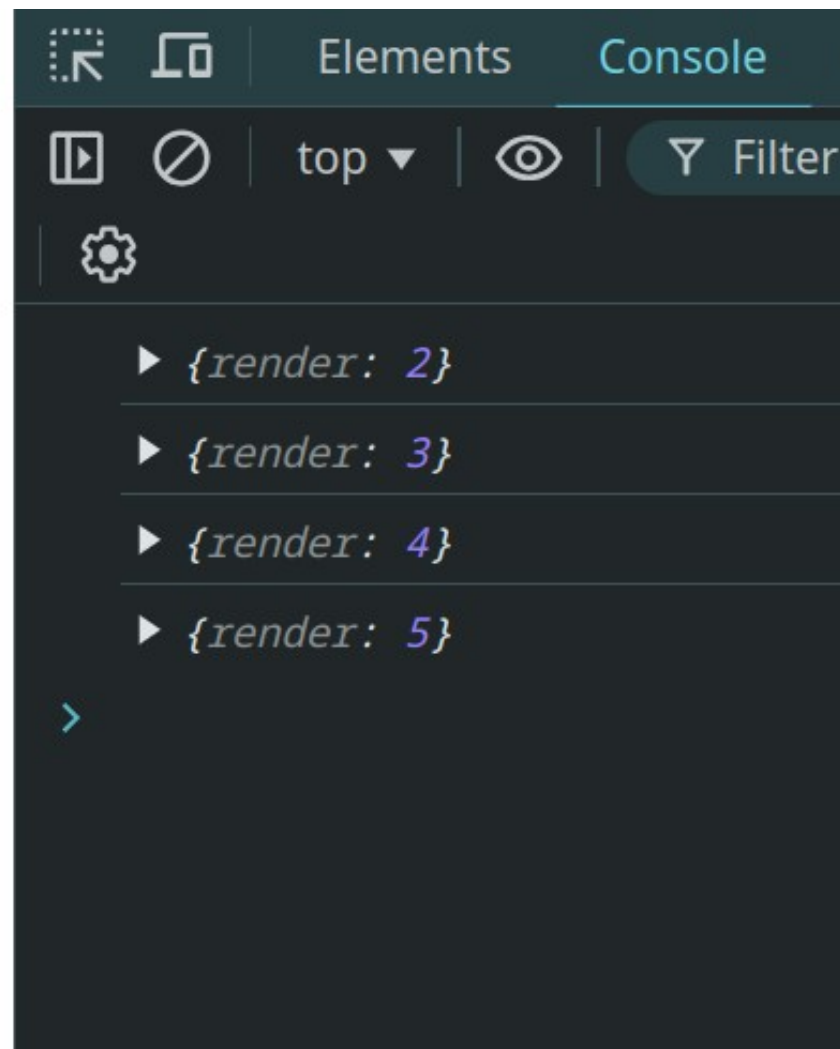
- **Exemplo 01**: (veja no console o valor de `countRef.current`)

## Exemplo useRef 1

count is 4 (useState)

Clique o botão e veja no console o valor do `countRef` contando a quantidade de renderizações.

[Documentação do useRef](#)



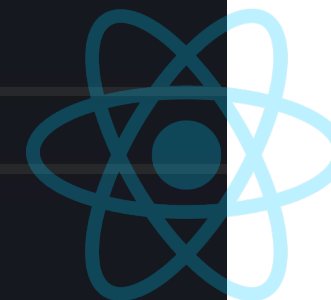
# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



- Neste exemplo, incrementamos o valor de `countRef` através da função `countRefInc`, cada vez que clicamos em um botão.
- Mas, ao contrário do primeiro botão, que usa um `useState`, a tela não é atualizada com o valor do incremento da `countRef`.
- Fica claro que como o `useRef` **NÃO GERA NOVAS RENDERIZAÇÕES**.
- **Exemplo 02**: (veja no console o valor de `countRef.current`)

```
6  ✓ function App() {  
7      const [count, setCount] = useState(0);  
8      const countRef = useRef(0);  
9  ✓  const countRefInc = () => {  
10         countRef.current = countRef.current + 1;  
11         console.log({ click: countRef.current });  
12     };
```



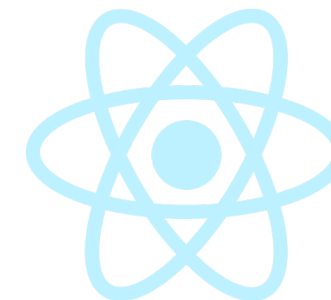
# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



- Já no terceiro exemplo, contamos quantas vezes o componente é renderizado utilizando como contador o próprio `useRef`.
- Apenas no primeiro render, onde `count` (`useState`), ainda é zero, não incrementamos o `renderizou`, após, a cada renderização, o código do **IF** é executado e o valor do `useRef renderizou` é incrementado.
- **Exemplo 03**: (veja também no console o valor da variável `renderizou`)

```
6  function App() {  
7      const [count, setCount] = useState(0);  
8      const renderizou = useRef(1);  
9      if (count) renderizou.current = renderizou.current + 1;  
10     console.log(renderizou.current);  
11 }
```





# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



- **Exemplo 03**: (veja também no console o valor de **`renderizou`**)

```
12  ✓  return (  
13  ✓    <>  
14      <h1>Exemplo useRef 3</h1>  
15  ✓    <div className="card">  
16  ✓      <button onClick={() => setCount((count) => count + 1)}>  
17        count is {count} (useState) (renderiz  
18        {renderizou.current.valueOf() > 1 ? 'ado ' : 'ou '  
19        {renderizou.current.valueOf()} vez  
20        {renderizou.current.valueOf() > 1 ? 'es' : ''})  
21      </button>
```

- Neste exemplo, conseguimos perceber ainda mais a vantagem de **`useRef`** de manter valores entre renderizações, sem a necessidade de um estado (**`useState`**).





# Hook `useRef()`

Evitando Re-renders, ou renderizações desnecessárias ao “lembrar de variáveis”



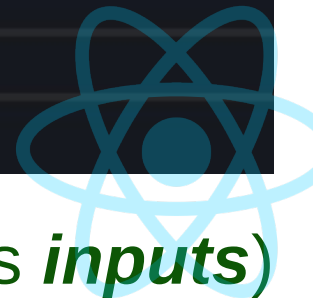
- Vamos analisar, no quarto exemplo, o uso de `useState` para a manipulação do conteúdo de elementos do tipo *input*.

```
6  ∨ function App() {  
7    const [listTodos, setListTodos] = useState([]);  
8    const [title, setTitle] = useState('');
```

A cada modificação (*onChange*) no conteúdo do *input* de “*título*”, o estado correspondente é atualizado, gerando uma nova renderização.

```
60  ∨  
61    type="text"  
62    placeholder="Título da tarefa"  
63    value={title}  
64    onChange={(e) => setTitle(e.target.value)}  
65    />
```

- **Exemplo 04:** (veja no console o conteúdo dos *states* para os campos *inputs*)



# Hook useRef()

Evitando renderizações excessivas

- O problema desta abordagem é a geração excessiva de renderizações.
- A cada modificação (**onChange**) no conteúdo do **input** de “**título**”, o estado correspondente é atualizado. Gerando novas renderizações, como é possível observar no console.
- **Exemplo 04.**

## React ToDoApp

Criar uma nova Tarefa

Título:

Texto:

Crie e organize suas tarefas!!!

html body



Console

What's new

Autofill

Net



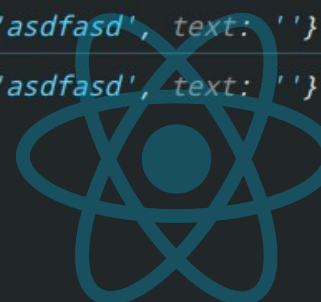
top



Filter

```
renderiza ▶ {title: 'a', text: ''}
renderiza ▶ {title: 'a', text: ''}
renderiza ▶ {title: 'as', text: ''}
renderiza ▶ {title: 'as', text: ''}
renderiza ▶ {title: 'asd', text: ''}
renderiza ▶ {title: 'asd', text: ''}
renderiza ▶ {title: 'asdf', text: ''}
renderiza ▶ {title: 'asdf', text: ''}
renderiza ▶ {title: 'asdfa', text: ''}
renderiza ▶ {title: 'asdfa', text: ''}
renderiza ▶ {title: 'asdfas', text: ''}
renderiza ▶ {title: 'asdfas', text: ''}
renderiza ▶ {title: 'asdfasd', text: ''}
renderiza ▶ {title: 'asdfasd', text: ''}
```

>



# Hook `useRef()`

Evitando renderizações excessivas



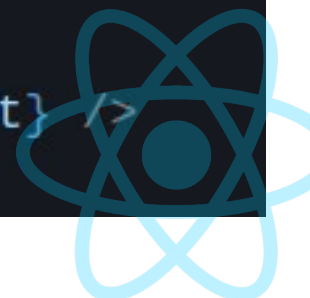
- No próximo exemplo, alteramos os `useState` por `useRef` para tratar os campos *input* e evitar renderizações em excesso.

```
6  ✓ function App() {  
7      const [listTodos, setListTodos] = useState([]);  
8      const inputTitle = useRef();  
9      const inputText = useRef();
```

- Neste caso, precisamos usar o atributo `ref` no elemento *input*, e **não usar** mais a propriedade `value`.

```
68      <label>Título:</label>  
69      <input type="text" placeholder="Título da tarefa" ref={inputTitle} />  
70      <br />  
71      <label>Texto:</label>  
72      <input type="text" placeholder="Texto da tarefa" ref={inputText} />  
73      <hr />
```

- Exemplo 05:** (observe o console deste exemplo)



# Hook `useRef()`

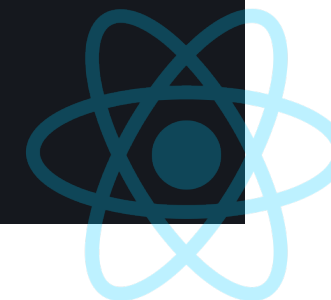
Evitando renderizações excessivas



- E quando for necessário acessar o valor do elemento ***input***, usamos novamente a propriedade ***current***, seguido do atributo próprio deste elemento do DOM, ***value***. (aqui usamos também ***optional chaining*** **?.**)

```
17  const createTodo = () => {  
18      let title = inputTitle?.current?.value;  
19      let text = inputText?.current?.value;  
20      if (!title || !text) return;  
21      newTodo({ title, text });  
22      inputTitle.current.value = '';  
23      inputText.current.value = '';  
24  };  
25
```

- **Exemplo 05**: (observe o console deste exemplo)



# A API de Contexto

## Introdução a API de Contexto



- Notamos na aplicação do **TodoList** do exemplo anterior (**Exemplo 05**) a necessidade de repassar vários atributos e até funções de componentes mais superiores (**App.jsx**) aos componentes mais internos (**Todo.jsx**) da arquitetura da aplicação (**component tree**)
- Isso é chamado de **Props Drilling**, em aplicações maiores ficaria ainda mais complexo e de difícil manutenção se seguissemos essa prática.
- Quando era necessário compartilhar além dos dados mas também a lógica, o React resolvia isso com o padrão de Componentes de Alta Ordem, High Order Components (**HOC**), mas era apenas um padrão.
- Atualmente, nas versões mais recentes do ReactJs, usamos a **API de Contexto** e o hook **useContext**, facilitando o trabalho com estados e funções **GLOBAIS** em nossa aplicação.



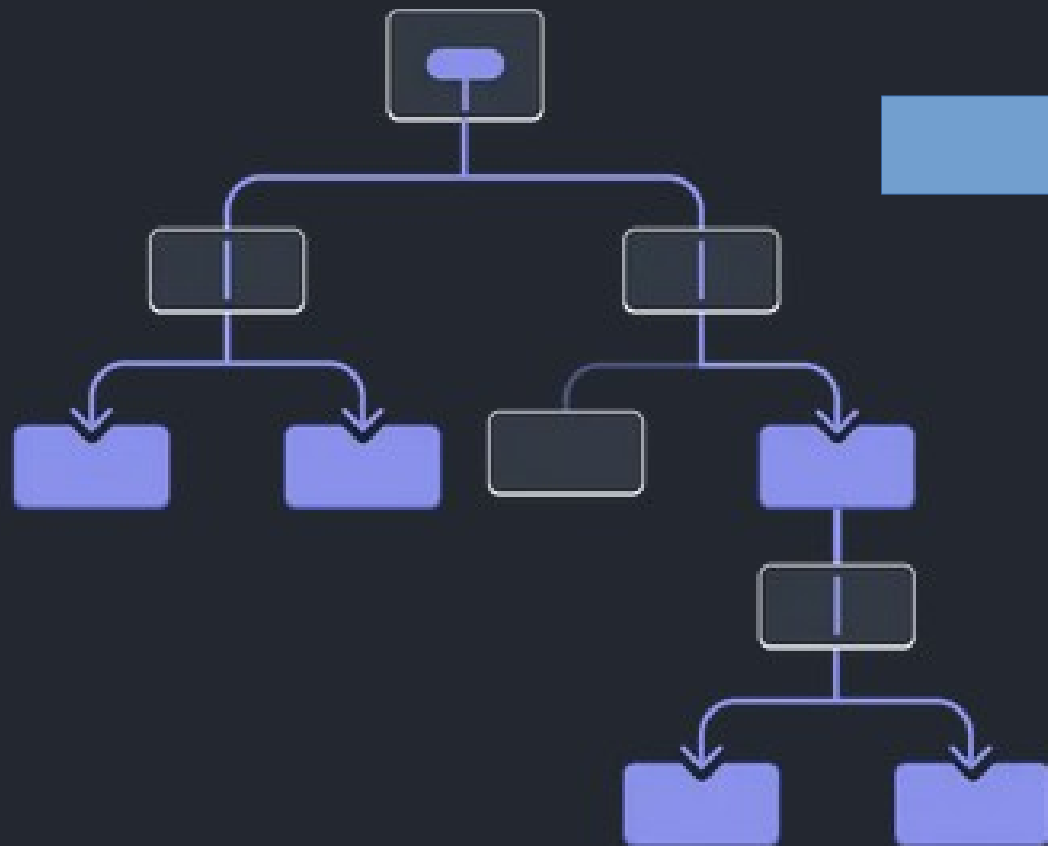


# A API de Contexto

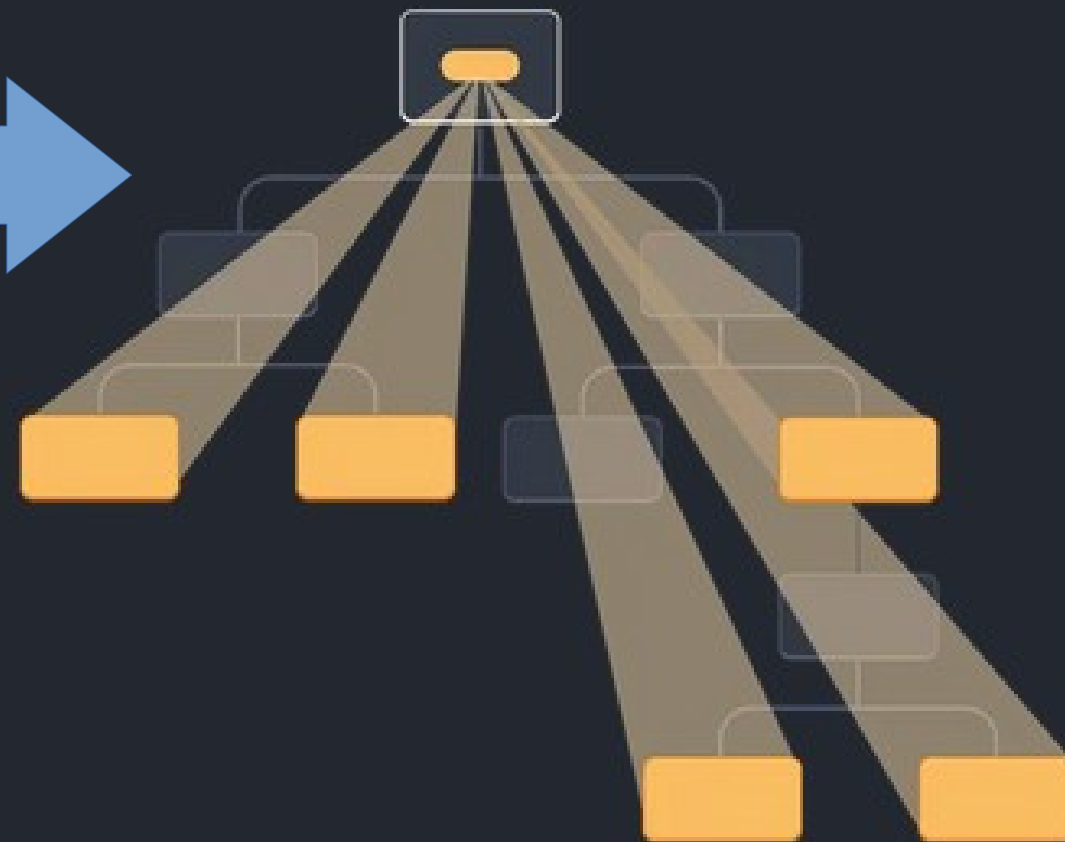
Diferenças entre os estados globais com contexto e passagem de propriedades



## Props Drilling



## Context API



Fonte: <https://react.dev/learn/passing-data-deeply-with-context>



# A API de Contexto

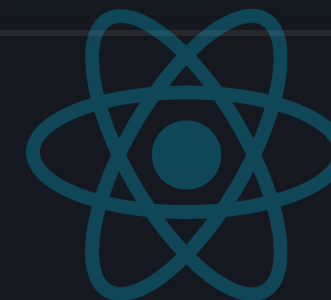
Criação de um contexto para compartilhar o tema da aplicação



- No exemplo a seguir, iremos criar um contexto para compartilhar o tema da aplicação entre todos os componentes.
- Para isso usamos a função ***createContext*** para criar o contexto.
- Inicializamos como um ***string*** o vazia, apenas para definir o tipo de dado.
- Para funções é possível passar a notação de ***arrow function*** (***()=>{}***)

## • Exemplo 06

```
App.jsx X
4
5  import { createContext } from 'react';
6
7  export const ThemeContext = createContext('');
8
9  function App() {
10     const [count, setCount] = useState(0);
11     const [theme] = useState('light');
```



# A API de Contexto

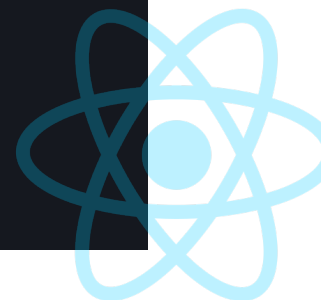
Criação de um contexto para compartilhar o tema da aplicação



- Em nosso componente principal criamos o state do **theme**, inicializando com o valor para temas claros, e repassamos como valor inicial do contexto.
- A propriedade **value** do componente “**Provedor**”, inicializará o contexto que irá prover o estado global do tema para os demais componentes filhos (**Home.jsx**)

```
11  const [theme, setTheme] = useState('light');
12
13  return (
14    <>
15      <ThemeContext.Provider value={theme}>
16        <Home setCount={setCount} count={count} />
17      </ThemeContext.Provider>
18    </>
19  );
```

## • Exemplo 06



# A API de Contexto

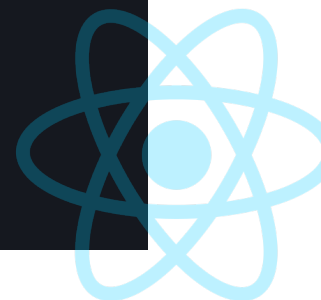
Criação de um contexto para compartilhar o tema da aplicação



- O componente Home é um filho do **Provider** de **ThemeContext** (**ThemeContext.Provider**).
- Tanto o Home, como os seus descendentes na árvore, terão acesso ao estado do tema da aplicação (**theme**).
- **Não sendo necessário**, portanto, repassar como uma propriedade (**props**)

```
11     const [theme, setTheme] = useState('light');
12
13     return (
14       <>
15         <ThemeContext.Provider value={theme}>
16           <Home setCount={setCount} count={count} />
17         </ThemeContext.Provider>
18       </>
19     );
```

## • Exemplo 06



# O hook useContext

Utilizando o contexto nos componentes filhos



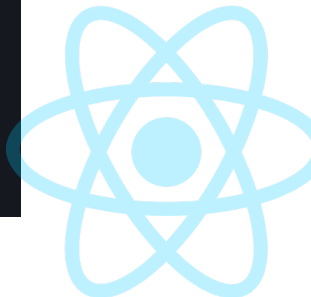
- No **App.jsx** nós exportamos o contexto, portanto ele poderá ser importado em outro módulo JS.

```
App.jsx  X  Home.jsx  Card.jsx
7  export const ThemeContext = createContext('');
8
9  function App() {
```

- No **Home.jsx**, importamos o contexto **ThemeContext**, e obtemos acesso ao seu estado com o hook **useContext**.

```
7  export function Home({ count, setCount }) {
8    const theme = useContext(ThemeContext);
9    console.log(theme);
10 }
```

- **Exemplo 06**



# O hook useContext

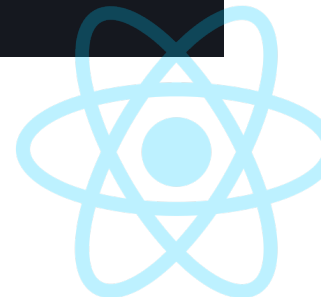
Utilizando o contexto nos componentes filhos



- Também usaremos o **useContext** e importaremos o **ThemeContext** no componente **Card.jsx**, filho do **Home.jsx**.

```
App.jsx  Home.jsx  Card.jsx  X
1  import { useContext } from 'react';
2  import { ThemeContext } from '../App';
3
4  export function Card({ count, setCount }) {
5    const theme = useContext(ThemeContext);
6    return (
7      <div className={`card ${theme}`}>
8        <button onClick={() => setCount((count) => count + 1)}>
```

- Exemplo 06**



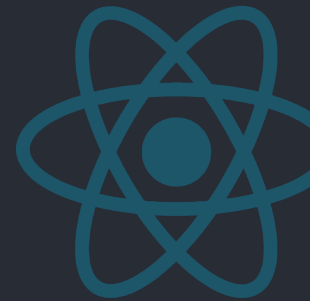
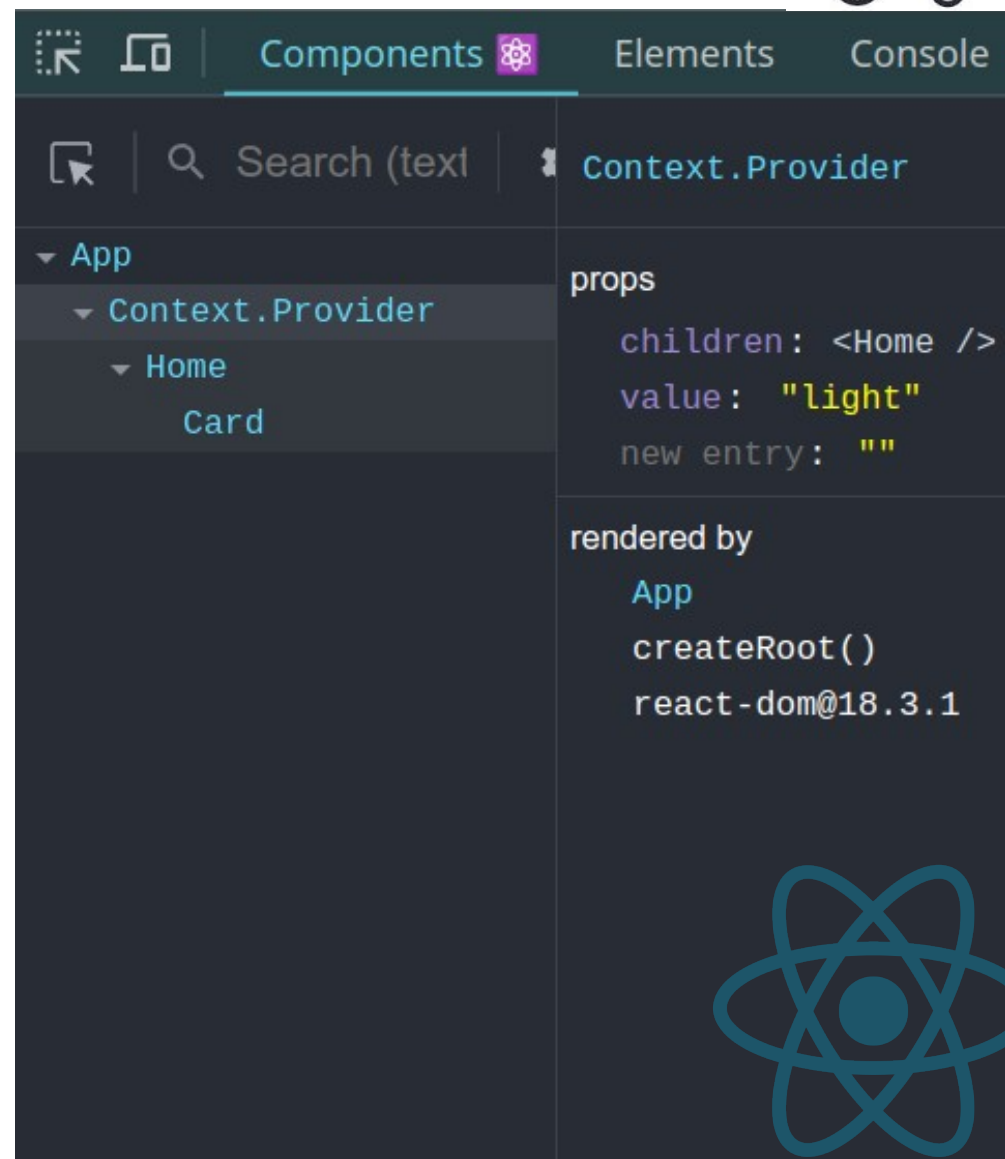


# O hook useContext

## Utilizando o contexto nos componentes filhos



- Utilizando a **React Developer Tools** no seu navegador, é possível observar o contexto sendo utilizado e ainda na posição hierárquica em que foi criado.
- Ao lado observamos os filhos **Home** e **Card**, logo abaixo do contexto criado pelo **App**.
- Ao selecionar o **Context.Provider**, ao lado, observamos suas propriedades (**props**) e estado (**value**).
- Experimente alterar o **value** para **'dark'** e observe o resultado.
- **Exemplo 06**





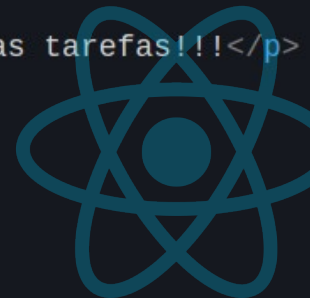
# Refatoração para contextos

Exemplo prático de uma aplicação com Props e sua versão com Contextos



- Veremos agora a versão com **props-drilling** da aplicação **ToDoApp** do React.
- Para ter acesso a lista de tarefas (**listTodos**) e as funções que manipulam esta lista, precisamos repassar várias propriedades aos componentes filhos.
- Como podemos observar os componentes **TodoFields** e **ListaTodo**, recebem como propriedades (**props**), não só a lista de tarefas (**listTodos**), mas também as funções (**editTodo**, **deleteTodo** e **createTodo**) além das referências (**useRef**) dos campos inputs.
- **Exemplo 07 (props)**

```
App.jsx x
61
62   return (
63     <>
64       <h1>React ToDoApp</h1>
65       <TodoFields
66         inputTitleRef={inputTitle}
67         inputTextRef={inputText}
68         createTodo={createTodo}
69         editedTodo={editedTodo}
70       />
71       <div className="card">
72         {listTodos.length > 0 ? (
73           <ListaTodo
74             listTodos={listTodos}
75             editTodo={editTodo}
76             deleteTodo={deleteTodo}
77           />
78         ) : (
79           <p>Crie e organize suas tarefas!!!</p>
80         )}
81       </div>
82     </>
83   );
84 }
```



# Refatoração para contextos

Exemplo prático de uma aplicação com Props e sua versão com Contextos



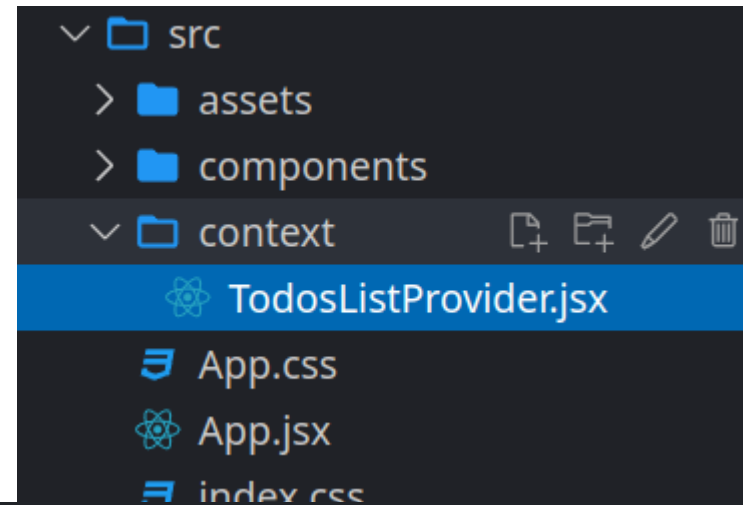
- Criamos então um novo componente apenas para gerenciar e **prover** um **contexto**, o qual chamamos de provedor ou **Provider**.
- Nele repassamos todas aquelas propriedades (atributos e funções) que desejamos compartilhar de forma **GLOBAL**.
- Portanto, removemos toda a lógica que estava no **App.jsx** e passamos para o novo provedor.

## ► **<TodosListProvider>**

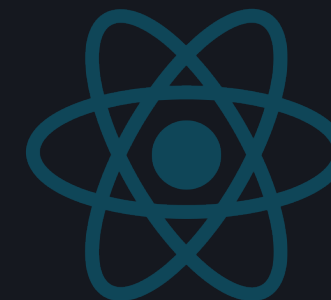
- E criamos o contexto **StateTodoList**

- **ListTodos**
- **EditedTodo**
- **NewTodo, deleteTodo e**
- **setEditedTodo**

- **Exemplo 08 (context)**



```
TodosListProvider.jsx X
1  import { useState, createContext } from 'react';
2
3  export const StateTodosList = createContext({
4    listTodos: [],
5    editedTodo: {},
6    newTodo: () => {},
7    deleteTodo: () => {},
8    setEditedTodo: () => {},
9  });
```



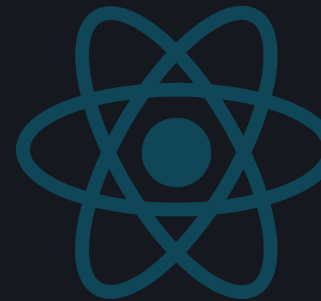
# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- A lógica das funções e o gerenciamento dos estados, será realizado em um novo componente que chamamos também de:
  - ▶ ***<TodosListProvider>***
- Retornamos então o componente Provider do StateTodoList com as propriedades que precisamos compartilhar.
- Desta forma, os componentes filhos (***children***), terão acesso a estas propriedades de forma **GLOBAL**, desde de que se utilizem do ***useContext***.
- **Exemplo 08 (context)**

TodosListProvider.jsx X

```
10
11 export const TodosListProvider = ({ children }) => {
12   const [listTodos, setListTodos] = useState([]);
13   const [editedTodo, setEditedTodo] = useState({});
14
15   > const newTodo = ({ title, text }) => { ...
29   };
30
31   > const deleteTodo = (_todo) => { ...
35   };
36
37   return (
38     <StateTodosList.Provider
39       value={{
40         listTodos,
41         editedTodo,
42         newTodo,
43         deleteTodo,
44         setEditedTodo,
45       }}
46     >
47       {children}
48     </StateTodosList.Provider>
49   );
50 };
51
```



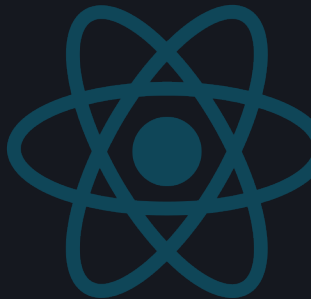
# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- Vamos repassar, neste caso, toda a aplicação como filha do provedor **<TodosListProvider>**
- No arquivo main.jsx, portanto, importamos o componente **<TodosListProvider>** e passamos o **<App/>** como um filho.
- A partir de agora, todos os componentes da aplicação poderão fazer uso das propriedades compartilhadas pelo contexto **StateTodoListe**.

- **Exemplo 08**

```
main.jsx X
5  import { TodosListProvider } from './context/TodosListProvider';
6
7  createRoot(document.getElementById('root')).render(
8    <StrictMode>
9      <TodosListProvider>
10        <App />
11      </TodosListProvider>
12    </StrictMode>
13  );
```





# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- Utilizando o **React Dev Tools**, podemos visualizar melhor o que está acontecendo.
- Temos o nosso componente provedor ***TodosListProvider*** que retorna um ***Contexto***, o ***StateTodosList*** e, que por sua vez, possui como filhos (***children***) o componente ***<App/>***

```
TodosListProvider.jsx X
1  import { useState, createContext } from 'react';
2
3  export const StateTodosList = createContext({
4    listTodos: [],
5    editedTodo: {},
6    newTodo: () => {},
7    deleteTodo: () => {},
8    setEditedTodo: () => {},
9  });
```

### Exemplo 08

Components | Profiler | Elements | Console

Search (text or)

TodosListProvider

▼ TodosListProvider

▼ Context.Provider

▼ App

TodoFields

props

children: <App />

new entry: ""

hooks

1 State: []

2 State: {}

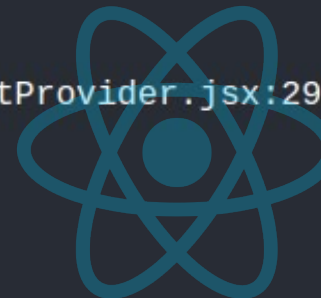
rendered by

createRoot()

react-dom@18.3.1

source

TodosListProvider.jsx:29

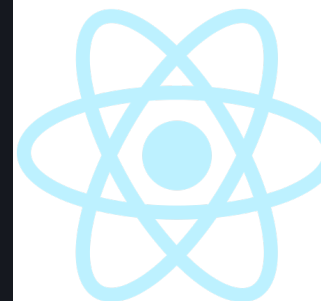


# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- Agora, nos componentes filhos, acessamos o contexto com o hook ***useContext()***.
- Importamos o **estado *StateTodosList*** do componente ***TodosListProvider*** e repassamos como argumento ao hook ***useContext()***.
- **Desestruturamos** o objeto de retorno para acessar as propriedades.

```
App.jsx X
1  /* eslint-disable no-unused-vars */
2  import { useContext, useEffect, useRef } from 'react';
3
4  import './App.css';
5  import TodoFields from './components/TodoFields/TodoFields';
6  import ListTodo from './components/ListTodo/ListTodo';
7  import { StateTodosList } from './context/TodosListProvider';
8
9  function App() {
10     const { listTodos, newTodo, setEditedTodo } = useContext(StateTodosList);
11     const inputTitle = useRef();
12     const inputText = useRef();
```





# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- Não é obrigatório acessar todas as propriedades do contexto retornado pelo *hook* **`useContext()`**, no componente **`TodoFields`**, por exemplo, interessa apenas o estado da propriedade **`editedTodo`**, para saber qual tarefa esta sendo editada.

```
TodoFields.jsx X
3
4 > export default function TodoFields({ ...
8 }) {
9   const { editedTodo } = useContext(StateTodosList);
10  return (
11    <fieldset>
12      <legend>
13        {' '}
14        {editedTodo?.title ? 'Edite a' : 'Criar uma nova'} Tarefa
15      </legend>
```

### • Exemplo 08

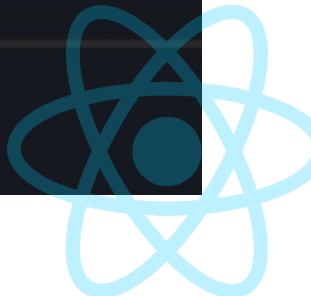
# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- Já o componente **ListTodo.jsx**, necessita apenas do estado da propriedade *listTodos*.
- Observe também que o uso de contexto não exclui totalmente o uso de **props**, neste exemplo o **ListTodo** ainda recebe a **props editTodo**.

```
•  TodoFields.jsx  ListTodo.jsx ✕  
1 import { useContext } from 'react';  
2 import Todo from '../Todo/TODO';  
3 import { StateTodosList } from '../../context/TodosListProvider';  
4  
5 export default function ListTodo({ editTodo }) {  
6   const { listTodos } = useContext(StateTodosList);  
7   return (  
8     <>  
9   )
```

### • Exemplo 08



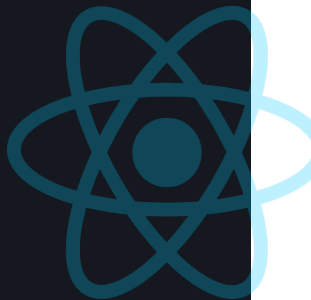
# Refatorar com contextos

## Exemplo prático de uma aplicação com Contextos

- O componente **Todo.jsx**, por sua vez, recebe as suas *props* e também usa o contexto para acessar a função *deleteTodo*.
- Para cada tarefa renderizada na tela é necessário criar o botão para remoção (*deleteTodo*) e edição (*editTodo*).

```
TodoFields.jsx  ListTodo.jsx  Todo.jsx X
1  import { useContext } from 'react';
2  import { StateTodosList } from '../context/TodosListProvider';
3
4  export default function Todo({ todo, editTodo }) {
5      const { deleteTodo } = useContext(StateTodosList);
6      return (
7          <li>
8              <h2>{todo.title}</h2>
9              <h5>{todo.text}</h5>
```

Exemplo 08



# Refatoração com contextos

Exemplo prático de uma aplicação com Props e sua versão com Contextos



- *Debugando a versão final com contextos do exemplo **ToDoApp***

## React ToDoApp

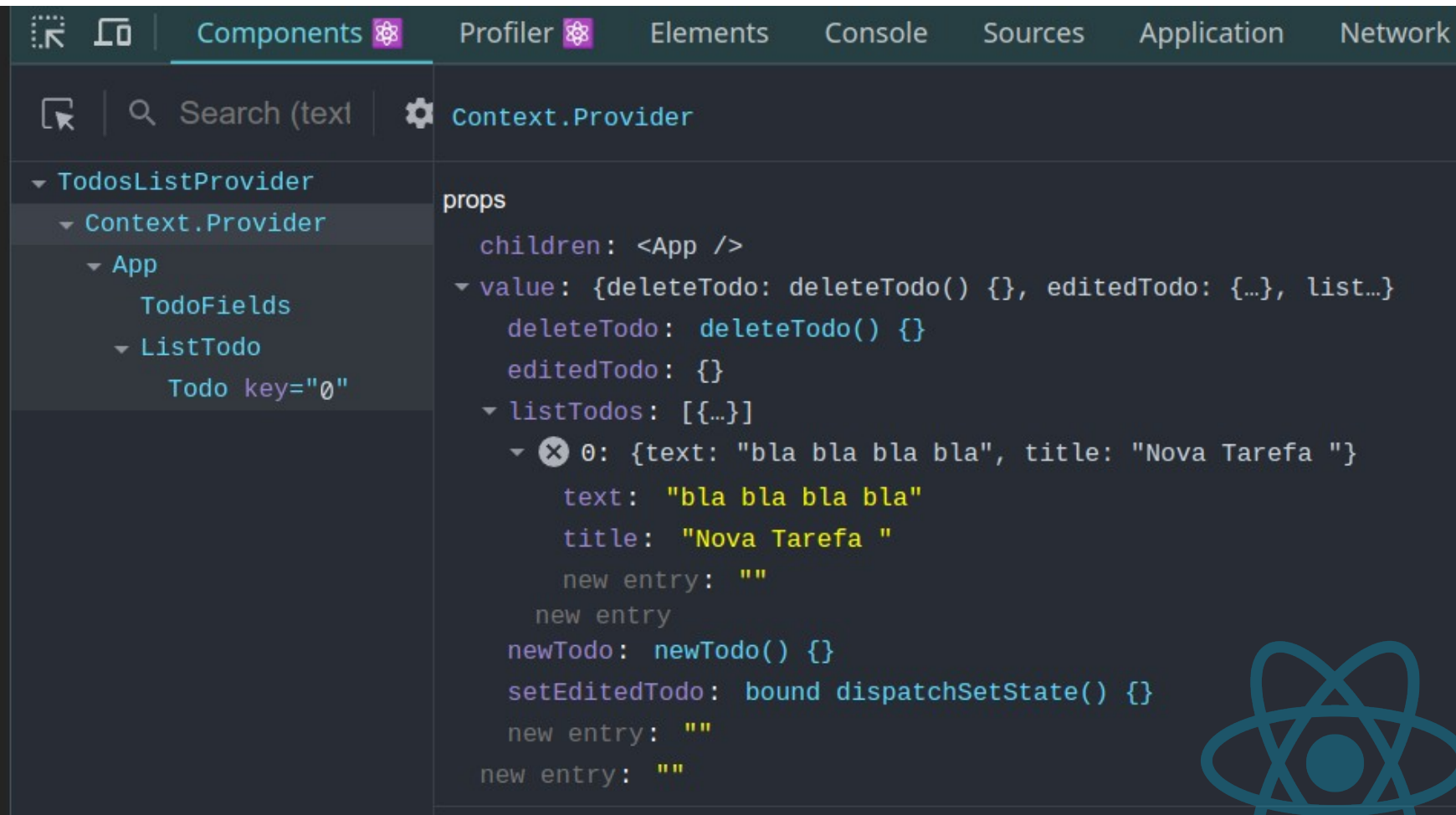
Criar uma nova Tarefa

Título:

Texto:

Suas Tarefas:

1. **Nova Tarefa**  
bla bla bla bla  
[Editar](#) | [X](#)

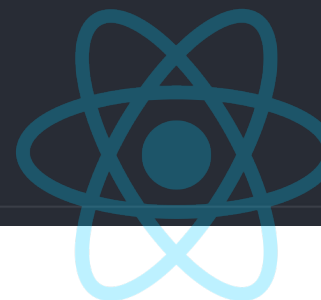


The screenshot shows the React DevTools interface. The Components panel on the left shows a tree structure: TodosListProvider > Context.Provider > App > TodoFields > ListTodo > Todo key="0". The Props panel on the right shows the props for the selected Context.Provider component. The props include children, value (an object with deleteTodo, editedTodo, and listTodos), and listTodos (an array of todo objects). The first todo object in the list has text "bla bla bla bla" and title "Nova Tarefa".

```
Context.Provider

props
  children: <App />
  value: {deleteTodo: deleteTodo() {}, editedTodo: {...}, list...}
    deleteTodo: deleteTodo() {}
    editedTodo: {}
    listTodos: [{...}]
      0: {text: "bla bla bla bla", title: "Nova Tarefa "}
        text: "bla bla bla bla"
        title: "Nova Tarefa "
        new entry: ""
        new entry
        newTodo: newTodo() {}
        setEditedTodo: bound dispatchSetState() {}
        new entry: ""
        new entry: ""
```

- **Exemplo 08 (context)**





# A API de Contexto

Diferenças entre os estados globais com contexto e passagem de propriedades

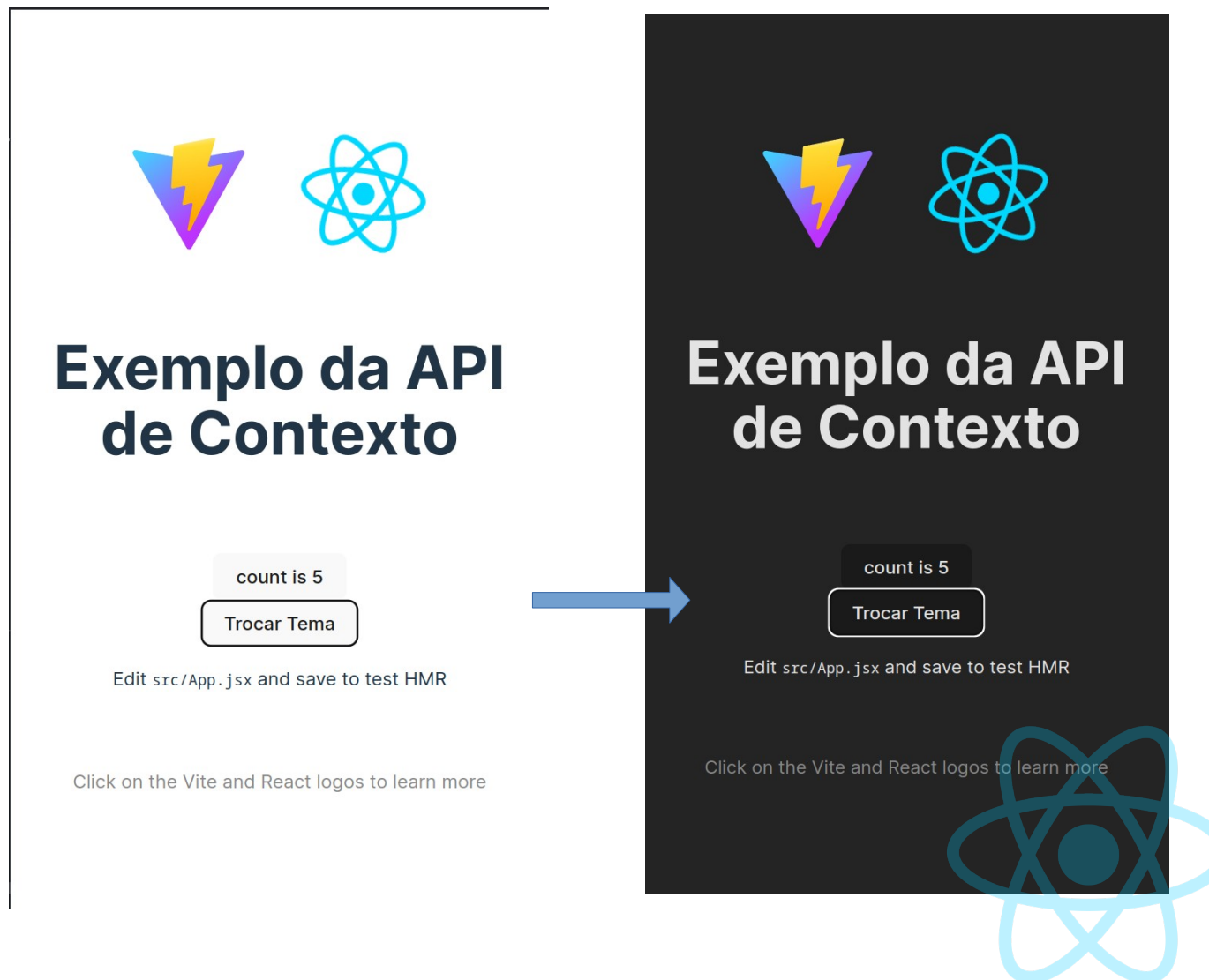


- Exercício:

Altere no exemplo 6, primeiro de contexto, o **theme**, para criar um contexto que controle o estado do tema, e não apenas o seu valor.

- Coloque no contexto uma função chamada **toggleTheme**, para alternar entre os temas claro e escuro.

- **Exemplo 06**



# A API de Contexto

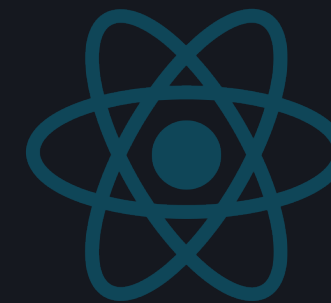
Diferenças entre os estados globais com contexto e passagem de propriedades



- Exercício:
- Refazer o exemplo do **TodoList** para não passar **nenhuma informação por prop**.
- Ou seja, **TodoFields** e **ListTodo** não devem receber **props**, como mostra o código da figura ao lado.
- Apenas o componente **Todo** pode receber a sua tarefa (**todo**) por propriedade.

```
App.jsx  Todo.jsx  X
import { useContext } from 'react';
import { StateTodosList } from '../context
export default function Todo({ todo }) {
```

```
App.jsx  X
return (
  <>
    <h1>React ToDoApp</h1>
    <TodoFields />
    <div className="card">
      {listTodos.length > 0 ? (
        <ListTodo />
      ) : (
        <p>Crie e organize suas
      )}
    </div>
  </>
);
}
export default App;
```





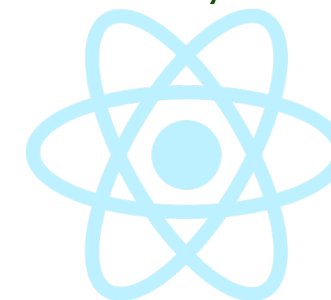
# A API de Contexto

Diferenças entre os estados globais com contexto e passagem de propriedades



- **Atividades:**

- 1) Refazer o seu aplicativo do ***ToDoList*** utilizando ***useRef*** e a **API de Contexto**, mantenha o salvamento da lista de tarefas no **localStorage**.
- 2) Altere o seu projeto de **front-end** para usar contexto nos dados **mockados** e evitar o uso de **props-drilling**.
- 3) Consulte a documentação do React Router para utilizar o componente **<Navigate>** ou o **hook useNavigate**, e crie páginas de acesso restrito ao seu projeto de *front-end*.
- 4) Permita o acesso as páginas restritas apenas aos usuários autenticados, simule através do **localStorage**.





## • REFERÊNCIAS

**SILVA, Maurício Samy. React Aprenda Praticando.** São Paulo: Novatec, 2021.

**META INC. Quick Start – React: 2022** Meta Platforms, Inc. Disponível em: <https://react.dev/learn>. Acesso em: 17/10/2024.

**SILVA, Maurício Samy. CSS3 – Desenvolva aplicações web profissionais com o uso dos poderosos recursos de estilização das CSS3.** São Paulo: Novatec, 2011.

**SILVA, Maurício Samy. HTML5 – A linguagem de marcação que revolucionou a web.** São Paulo: Novatec, 2011.

**SILVA, Maurício Samy. JavaScript: guia do programador.** São Paulo: Novatec, 2010.

**Tutorial | React Router - reactrouter.com.** Disponível em: <https://reactrouter.com/en/main/start/tutorial>. Acesso em: 28/10/2024

**DJIRDEH, Hassan. Server-Side Routing vs. Client-Side Routing.** Disponível em: <https://www.telerik.com/blogs/server-side-routing-vs-client-side-routing>, Acesso em: 28/10/2024