

Apostila Treinamento ReactJS

Sumário

Introdução	3
Estrutura de um projeto React.....	4
Configurando seu primeiro app React	7
Estado (State)	21
Componentes	22
sem estado (stateless).....	22
com estado (stateFull).....	24
Props	27
Props padrão	29
Estado (state) e props	30
Validação de props	32
Component API	35
Ciclo de Vida do Componente.....	40
Formulários	44
Eventos.....	46
Child Events	48
Refs.....	50
Keys	52
Router (Roteador)	55
Conceito Flux.....	70
Usando Flux	71
React Redux.....	79
Por que usar o React Redux?	79
Arquitetura Redux	80
Princípios do Redux	81
O que é uma action?	83
Fluxo de Dados	84
Fundamentos de Actions.....	87
funções puras	89
Reducers.....	91
Middleware	95
Usando React-Redux	98
React Hooks.....	106

Regras dos Hooks	107
Hooks State	108
Hooks Effect	110
Custom Hooks	112
Built-in Hooks	114
React Map	114
Componentes de high order	117
Animações	119
React Bootstrap.....	126
Bootstrap como Dependency.....	128
React Bootstrap Package.....	129
Usando o reactstrap	133
React Table	136
Referencias:.....	141

Introdução

ReactJS é uma biblioteca JavaScript declarativa, eficiente e flexível para a construção de componentes de IU reutilizáveis. É uma biblioteca de front-end baseada em componentes de código aberto responsável apenas pela camada de visualização do aplicativo. Ele foi criado por **Jordan Walke**, que era engenheiro de software no **Facebook**. Ele foi inicialmente desenvolvido e mantido pelo Facebook e depois foi usado em seus produtos como **WhatsApp** e **Instagram**. O Facebook desenvolveu o ReactJS em **2011** em sua seção de feed de notícias, mas ele foi lançado ao público no mês de **maio de 2013**.

Hoje, a maioria dos sites é construída usando a arquitetura MVC (model view controller). Na arquitetura MVC, React é o 'V' que representa a visualização, enquanto a arquitetura é fornecida pelo Redux ou Flux.

Um aplicativo ReactJS é feito de vários componentes, cada componente responsável por gerar uma pequena parte reutilizável de código HTML. Os componentes são o coração de todos os aplicativos React. Esses componentes podem ser aninhados com outros componentes para permitir que aplicativos complexos sejam construídos a partir de blocos de construção simples. ReactJS usa mecanismo baseado em DOM virtual para preencher dados em HTML DOM. O DOM virtual funciona rápido, pois só muda elementos DOM individuais em vez de recarregar o DOM completo todas as vezes.

Para criar o aplicativo React, escrevemos componentes React que correspondem a vários elementos. Organizamos esses componentes dentro de componentes de nível superior que definem a estrutura do aplicativo. Por exemplo, assumimos um formulário que consiste em muitos elementos, como campos de entrada, rótulos ou botões. Podemos escrever cada elemento do formulário como componentes React e, em seguida, combiná-lo em um componente de nível superior, ou seja, o próprio componente do formulário. Os componentes do formulário especificariam a estrutura do formulário junto com os elementos dentro dele.

Por que aprender ReactJS?

Hoje, muitos frameworks JavaScript estão disponíveis no mercado (como angular, node), mas ainda assim, React entrou no mercado e ganhou popularidade entre eles. Os frameworks anteriores seguem a estrutura tradicional de fluxo de dados, que usa o DOM (Document Object Model). DOM é um objeto criado pelo navegador cada vez que uma página da web é carregada. Ele adiciona ou remove dinamicamente os dados no backend e quando qualquer modificação for feita, cada vez que um novo DOM é criado para a mesma página. Essa criação repetida de DOM torna desnecessário o desperdício de memória e reduz o desempenho do aplicativo.

Portanto, uma nova estrutura de tecnologia ReactJS inventada que remove essa desvantagem. O ReactJS permite dividir todo o seu aplicativo em vários componentes. O ReactJS ainda usa o mesmo fluxo de dados tradicional, mas não está operando diretamente no Document Object Model (DOM) do navegador imediatamente; em vez disso, ele opera em um DOM virtual. Isso significa que, em vez de manipular o documento em um navegador após as alterações em nossos dados, ele resolve as alterações em um DOM construído e executado inteiramente na memória. Depois que o DOM virtual foi atualizado, o React determina quais alterações foram feitas no DOM do navegador real. O React Virtual DOM existe inteiramente na memória e é uma representação do DOM do navegador da web. Por isso, quando gravamos um componente React, não gravamos diretamente no DOM; em vez de,

Estrutura de um projeto React

Segundo seu slogan oficial, React é uma biblioteca para construção de interfaces de usuário. React não é um framework – nem mesmo é exclusivo para web. É utilizado com outras bibliotecas para renderização em certos ambientes. Por exemplo, React Native pode ser usado para construção de aplicativos móveis; React 360 pode ser usado para construir aplicações de realidade virtual; e muitas outras possibilidades.

Para construir para web, desenvolvedores usam React em conjunto com ReactDOM. React e ReactDOM são frequentemente discutidos nos mesmos espaços e utilizados para resolver os mesmos problemas como outros arcabouços(frameworks). Quando referimos React como "arcabouço"(framework) estamos trabalhando com o termo/entendimento coloquial.

A meta primária do React é minimizar os erros que ocorrem quando os desenvolvedores estão construindo UIs(User Interface). Isto é devido ao uso de componentes - autocontidos, partes lógicas de códigos que descrevem uma parte da interface do usuário. Estes componentes são adicionados para criar uma UI completa e o React concentra muito do trabalho de renderizar, proporcionando que se concentre no projeto de UI.

Casos de uso

Diferente de outros frameworks comentados neste módulo, React não implementa regras restritas no código como convenções ou organizações de arquivos. Isto permite que times criem convenções próprias que melhor se adequem e para adotar o React do jeito que desejar. React pode manusear um botão único, poucas partes da interface ou a interface inteira de um app.

Enquanto React pode ser utilizado por pequenos pedaços de interface e não "cai" em uma aplicação com uma biblioteca como jQuery ou até mesmo como um framework como Vue - é mais acessível quando você constrói todo o app com React.

Além disso, muitos dos benefícios da experiências de desenvolvimento de uma aplicação React, tais como escrever interfaces com JSX, requerem um processo de compilação. Adicionar um compilador como o Babel em um website faz o código funcionar lentamente, então os desenvolvedores geralmente configuram algumas ferramentas para fazer compilações em etapas. React, sem dúvidas, tem um grande ecossistema de ferramentas, mas isso pode ser aprendido.

Como React usa Javascript?

React utiliza características de Javascript moderno para muitos de seus padrões. O maior desvio do React para o JavaScript dá-se pela utilização sintaxe JSX. O JSX estende a sintaxe padrão do Javascript habilitando-o a utilizar código similar a HTML que pode viver lado a lado ao JSX. Por exemplo:

```
const heading = <h1> Texto dentro de Tags</h1>;
```

A constante *heading* acima é conhecida como uma **expressão JSX**. React pode utilizá-la para renderizar a *tag* `<h1>` em nosso aplicativo.

Suponha que quiséssemos conter nosso cabeçalho em uma tag `<header>`, por razões semânticas? A aproximação em JSX permite-nos aninhar nossos elementos dentro uns dos outros, do mesmo jeito que fazemos com o HTML:

```
const header = (  
  <header>  
    <h1>Mozilla Developer Network</h1>  
  </header>  
) ;
```

Obs.:: Os parenteses no recorte de código anterior não são exclusivos ao JSX e não têm nenhum efeito na sua aplicação. Eles estão lá para sinalizar para você (e seu computador) que as múltiplas linhas de código dentro do mesmo são parte da mesma expressão. Você poderia muito bem escrever a expressão do cabeçalho do seguinte jeito:

```
const header = <header>  
  <h1>Mozilla Developer Network</h1>  
</header>
```

Entretanto, isso é meio estranho, porquê a tag `<header>` que inicia a expressão não está alinhada na mesma posição que sua tag de fechamento correspondente.

Claro, seu navegador não é capaz de ler o JSX sem alguma ajuda. Quando compilado (utilizando uma ferramenta como Babel ou Parcel), nossa expressão de cabeçalho ficaria assim:

```
const header = React.createElement("header", null,
```

```
React.createElement("h1", null, "Mozilla Developer  
Network")  
);
```

É possível pular o processo de compilação e utilizar `React.createElement()` para escrever sua UI você mesmo. Ao fazer isso, entretanto, você perde o benefício declarativo do JSX, e seu código torna-se mais difícil de ler. Compilação é um passo adicional no processo de desenvolvimento, porém muitos desenvolvedores na comunidade do React acham que a legibilidade do JSX vale a pena. Ainda mais, ferramentas populares fazem a parte de compilar JSX-para-Javascript parte do próprio processo de configuração. Você não vai ter que configurar a compilação você mesmo, a não ser que você queira.

Por conta do JSX ser uma mistura de HTML e Javascript, muitos desenvolvedores acham o JSX intuitivo. Outros dizem que a natureza mista torna o mesmo mais confuso. Entretanto, assim que você estiver confortável com o JSX, este irá permitir que você construa interfaces de usuários mais rapidamente e intuitivamente, e permitirá que outros melhor entendam seu código com apenas algumas olhadas.

Configurando seu primeiro app React

Existem muitos jeitos de utilizar o React, mas nós iremos utilizar a ferramenta de interface da linha de comando (ILC), *create-react-app*, como mencionado anteriormente, que acelera o processo de desenvolvimento da aplicação em React instalando alguns pacotes e criando alguns arquivos para você, lidando com os processos de automação mencionados acima.

É possível adicionar React á um website sem *create-react-app* copiando alguns elementos `<script>` em um arquivo HTML, mas a interface de linha de comando *create-react-app* é um ponto de partida comum para aplicações em React. Utilizar-lo vai permitir que você passe mais tempo construindo seu aplicativo e menos tempo incomodando-se com configurações.

Requisitos

Para começar a utilizar o *create-react-app*, você precisa ter o Node.js instalado. É recomendado que você utilize a versão com suporte de longa data (SLT). *Node* inclui o *npm* (o gerenciador de pacotes node), e o *npx* (o executador de pacotes do node).

Mantenha em mente também que React e ReactDOM produzem aplicativos que funcionam apenas em navegadores consideravelmente modernos, IE9+ (Internet Explorer 9) com o auxílio de alguns *polyfills*. É recomendado que você utilize um navegador moderno com o Firefox, Safari, entre outros

Inicializando seu app

O create-react-app leva apenas um argumento: o nome que você quer dar ao seu aplicativo. create-react-app utiliza este nome para criar uma nova pasta, e então cria os arquivos necessários para o funcionamento do seu aplicativo dentro desta pasta.

instalar create-react-app

Para que o comando, mencionada acima, funcione adequadamente, precisamos fazer a instalação de seu módulo da seguinte forma:

insira o comando abaixo via prompt de comando (esse comando instalará globalmente):

```
npm install -g create-react-app
```

Certifique-se de utilizar o comando `cd` até o local em seu computador que você deseja que seu aplicativo viva dentro de seu disco rígido, feito isso, utilize o seguinte comando em seu terminal:

```
npx create-react-app my-app
```

Isto criará a pasta `my-app`, e também faz mais algumas coisas dentro desta:

- Instala alguns pacotes *npm* essenciais para a funcionalidade do app.

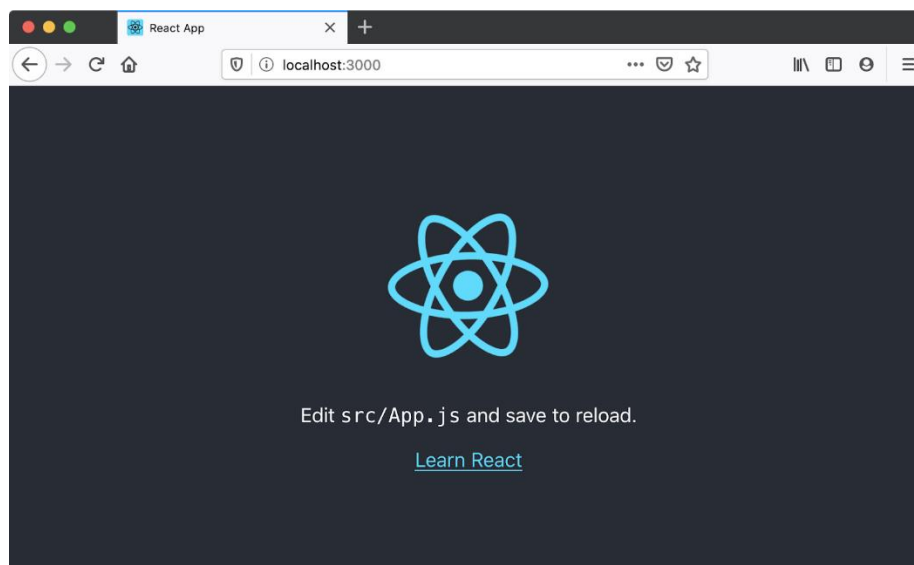
- Escreve scripts para iniciar e servir a aplicação.
- Cria a estrutura de arquivos e pastas que define a arquitetura básica do aplicativo.
- Inicializa o diretório como um *repositório git*, se você tem o *git* instalado em seu computador.

Nota: Se você tem o gerenciador de pacotes Yarn instalado, create-react-app vai utilizá-lo por padrão em vez de utilizar o npm. Se você tem ambos gerenciadores de pacotes instalados e explicitamente quer utilizar o NPM, você pode adicionar a opção `--use-npm` quando você executar o create-react-app:

```
npx create-react-app my-app --use-npm
```

create-react-app vai mostrar várias mensagens em seu terminal enquanto ele trabalha; isto é normal! Isso pode levar alguns minutos.

Quando o processo finalizar, navegue até a pasta de seu projeto **my-app** e execute o comando **npm start**. Os scripts instalados pelo *create-react-app* começarão a serem servidos em um servidor local, no endereço *localhost:3000*, e abrirão o aplicativo em uma nova aba em seu navegador. Seu navegador vai mostrar algo como isto:



Estrutura da aplicação

create-react-app dá para você tudo que você precisa para desenvolver uma aplicação React. A estrutura inicial do arquivo vai ficar assim:

```
my-app
├── README.md
├── node_modules
├── package.json
├── package-lock.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── reportWebVitals.js
```

A pasta **src** é onde nós iremos ficar a maior parte do nosso tempo, é onde o código fonte da nossa aplicação vive.

A pasta **public** contém arquivos que serão lidos pelo navegador enquanto você desenvolve o aplicativo; o mais importante de todos estes arquivos é o **index.html**. O React irá injetar seu código neste arquivo para que seu navegador possa executá-lo. Existem outras marcações que ajudam o **create-react-app** a funcionar, então cuidado para não editar estas, a não ser que você saiba o que você está fazendo. Você é encorajado a mudar o texto dentro do elemento **<title>** neste arquivo, esta mudança irá refletir no título de sua aplicação.

A pasta **public** também será publicada quando você construir e lançar uma versão de produção de seu aplicativo.

O arquivo **package.json** contém informação sobre nosso projeto que o **Node.js/npm** use para mantê-lo organizado. Esse arquivo não é exclusivo para aplicações em React; o **create-react-app** simplesmente encarregasse de popular este.

Explorando seu primeiro componente React — <App/>

No React, um **componente** é um módulo reutilizável que renderiza parte de nosso aplicativo. Estas partes podem ser grandes ou pequenas, mas elas geralmente são claramente definidas: elas servem um único propósito, um propósito óbvio.

Vamos o arquivo **src/App.js** dado que nosso navegador está nos instigando a editá-lo. Esse arquivo contém nosso primeiro componente, **App**, e algumas outras linhas de código.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

O arquivo **App.js** consiste de três partes principais: algumas declarações de **import** no topo, o componente **App** no meio, e uma declaração de **export** na parte de baixo. A maioria dos componentes React segue este padrão.

Declarações de import

As declaração de **import** no topo de nosso arquivo **App.js** nos permitem utilizar código que foi definido em outro lugar fora de nosso arquivo. Vamos Observar:

```
import React from 'react';  
import logo from './logo.svg';  
import './App.css';
```

A primeira declaração importa a própria biblioteca React. Por conta do React transformar o JSX que nós escrevemos em declarações de **React.createElement()**, todos componentes React devem importar o módulo **React**. Se você pular este passo, sua aplicação irá resultar em um erro.

A segunda declaração importa um logo de **./logo.svg**. Note que o **./** no começo do caminho e a extensão **.svg** no final — nos avisa que o arquivo é local e que não é um arquivo de Javascript. De fato, o arquivo **logo.svg** localiza-se na pasta base do projeto.

Nós não escrevemos um caminho ou extensão quando importando o módulo **React** — este não é um arquivo local; em vez disso, é listado como uma dependência em nosso arquivo **package.json**.

A terceira declaração importa o CSS relacionado ao nosso componente App. Note que não existe um nome de variável e também não há a diretriz **from**. Essa declaração de **import** em particular não é nativa à sintaxe de módulos do Javascript — esta vem do **Webpack**, a ferramenta que o aplicativo **create-react-**

app utiliza para agrupar todos os nossos arquivos de Javascript e servi-los ao navegador.

O componente App

Depois dos *imports*, temos uma função chamada **App**. Enquanto a maior parte da nossa comunidade Javascript prefere nomes utilizando o padrão *camel-case* como **helloWorld**, os componentes React utilizam o padrão de formatação para variáveis em *pascal-case*, como **HelloWorld**, para ficar claro que um dado elemento JSX é um componente React e não apenas uma *tag* de HTML comum. Se você mudasse o nome da função **App** para **app** o seu navegador iria mostrar um erro.

Vamos observar a função App:

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

A função **App** retrona uma expressão JSX. Essa expressão define o que, no fim, o seu navegador irá renderizar para o DOM.

Alguns elementos na expressão têm atributos, que são escritos assim como no HTML, seguindo o seguinte padrão de **atributo="valor"**. Na linha 3, a tag `<div>` de abertura tem o atributo **className**. Isso é o mesmo que o atributo **class** no HTML, porém por conta do JSX ser Javascript, nós não podemos utilizar a palavra **class** – porque é reservada, isso quer dizer que o Javascript já utiliza-a para um propósito específico e causaria problemas no nosso código inseri-lá aqui. Alguns outros atributos de HTML são escritos diferentes em JSX em comparação com o HTML, pela mesma razão.

Vamos alterar a tag `<p>` na linha 6; vamos inserir o texto: **"Hello, world!"**, e então salve o arquivo. Você irá notar que esta mudança é imediatamente refletida e renderizada no servidor de desenvolvimento executando em **`http://localhost:3000`** em seu navegador. Agora delete a tag `<a>` e salve; o link **"Learn React"** vai desaparecer.

Seu componente **App** deve estar assim agora:

```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo"  
alt="logo" />  
        <p>  
          Hello, World!  
        </p>  
      </header>  
    </div>  
  );  
}
```

Declarações export

Bem no final do seu arquivo **App.js**, a declaração **export default App** faz com que seu componente **App** esteja disponível para outros módulos.

Arquivo index

Vamos abrir **src/index.js**, porque é onde nosso componente **App** está sendo utilizado. Esse arquivo é o ponto de entrada para nosso aplicativo, e inicialmente parece-se com o código indicado abaixo:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />,
document.getElementById('root'));

// If you want your app to work offline and load
// faster, you can change
// unregister() to register() below. Note this comes
// with some pitfalls.
// Learn more about service workers:
https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Assim como em **App.js**, o arquivo começa importando todos os módulos de JS (Javascript) e outros recursos que precisa executar. **src/index.css** contém estilos globais (CSS) que são aplicados em todo nosso aplicativo. Nós podemos também ver nosso componente **App** importado aqui; este é disponibilizado para ser importado graças à declaração de **export** no final do nosso arquivo **App.js**. A linha 7 invoca a função **ReactDOM.render()** com dois argumentos:

- O componente que queremos renderizar, **<App />** neste caso.
- O elemento do DOM que queremos que nosso componente seja renderizado dentro, neste caso é o elemento com o ID de *root*. Se você olhar dentro de **public/index.html**, você verá que existe um elemento **<div>** logo ali dentro do elemento **<body>**.

Tudo isso diz para o React que nós queremos renderizar nossa aplicação React como o componente **App** como a raiz do app, ou o primeiro componente.

Obs.: No JSX, componentes React e elementos HTML precisam ser barras de fechamento. Escrever apenas <App> ou apenas irá causar um erro.

Pode-se, também, criar um arquivo **Service workers** que serão pedaços interessantes de código que ajudam a performance da aplicação e permitem a utilização de algumas características de sua aplicação de web a funcionarem *offline*.

Web Vitals

Por padrão, o comando create-react-app inclui um retransmissor de desempenho que permite medir e analisar o desempenho de seu aplicativo usando diferentes métricas.

Para medir qualquer uma das métricas suportadas, você só precisa passar uma função para a função **reportWebVitals** em **index.js**.

Web Vitals são um conjunto de métricas úteis que visam capturar a experiência do usuário em uma página da web. No Create React App, uma biblioteca de terceiros é usada para medir essas métricas (web-vitals).

Seu arquivo **index.js** final deve da seguinte forma

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(<App />,
document.getElementById('root'));
```

Variáveis no JSX

De volta ao **App.js** vamos focar na linha 9:

```
<img src={logo} className="App-logo" alt="logo" />
```

Aqui, na tag **** o atributo **src** está entre chaves ({ }). É assim que o JSX reconhece variáveis. React irá ver **{logo}**, saberá que você está referindo-se ao **import** do logo na linha 2 do nosso aplicativo, e então buscar o arquivo logo e renderizá-lo.

Vamos tentar fazer uma variável própria. Antes da declaração de **return** de **App** adicione **const subject = 'React';**. Seu componente **App** deve estar assim agora:

```
function App() {  
  const subject = "React";  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo"  
alt="logo" />  
        <p>  
          Hello, World!  
        </p>  
      </header>  
    </div>  
  );  
}
```

Mude a linha 8 para usar a nossa variável **subject** em vez da palavra "world", desta forma:

```
function App() {  
  const subject = "React";  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo"  
alt="logo" />  
        <p>  
          Hello, {subject}!  
        </p>  
      </header>  
    </div>  
  );  
}
```

```

        </header>
      </div>
    );
  }

```

Quando você salvar, seu navegador deverá mostrar "Hello, React!", em vez de mostrar "Hello, world!"

Variáveis são conveniente, mas esta que nós definimos não faz jus aos ótimos recursos do React. É aí que entram as *props*.

props de Componentes

Uma **prop** é qualquer dado passado para um componente React. *Props* são escritos dentro de invocações de componentes e utilizam a mesma sintaxe que atributos de HTML - **prop="valor"**. Vamos abrir o *index.js* e dar à nossa invocação do **<App/>** nossa primeira *prop*.

Adicione a *prop* **subject** para a invocação do componente **<App/>**, com o valor **Dino**. Quando você terminar, seu código deve estar assim:

```

ReactDOM.render(<App subject="Dino" />,
document.getElementById('root'));

```

De volta ao **App.js**, vamos revisitar a própria função App, que é lida da seguinte forma (com a declaração de return encurtada, a fim de ser breve.)

```

function App() {
  const subject = "React";
  return (
    // return statement
  );
}

```

Mude a definição da nossa função **App** para que aceite **props** como um parâmetro. Assim como qualquer outro parâmetro, você pode colocar **props** em

um **`console.log()`** para ler o que este contém no console de seu navegador. Vá em frente e faça justamente isto depois da sua constante **`subject`** porém antes da sua declaração de **`return`**, da seguinte forma:

```
function App(props) {  
  const subject = "React";  
  console.log(props);  
  return (  
    // return statement  
  );  
}
```

Salve seu arquivo e dê uma olhada no console do navegador. Você deve ver algo assim nos *logs*:

```
Object { subject: "Dino" }
```

A propriedade **`subject`** deste objeto corresponde à ***prop subject*** que nós adicionamos à nossa chamada do componente **`<App />`**, e a *string* **`Dino`** corresponde ao seu valor. ***props*** de componentes no React são sempre coletadas em objetos neste mesmo estilo.

Agora que **`subject`** é uma de nossas *props*, vamos utilizá-la em **`App.js`**. Mude a constante **`subject`** para que, em vez de ler a string que diz **`React`**, você está lendo o valor de **`props.subject`**. Você também pode deletar o **`console.log()`**, se você quiser.

```
function App(props) {  
  const subject = props.subject;  
  return (  
    // return statement  
  );  
}
```

Quando você salvar o arquivo, o aplicativo agora deve dizer "Hello, Clarice!". Se você retornar ao `index.js`, editar o valor de `subject` e salvar, seu texto irá mudar.

Comentários

Ao escrever comentários, precisamos colocar keys `{}` quando queremos escrever um comentário dentro da seção filho de uma tag. É uma boa prática sempre usar `{}` ao escrever comentários, uma vez que queremos ser consistentes ao escrever o aplicativo.

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
        { //End of the line Comment...}
        { /*Multi line comment...*/}
      </div>
    );
  }
}

export default App;
```

Convenção de nomes

As tags HTML sempre usam nomes de tags em **letras minúsculas**, enquanto os componentes do React começam com **letras maiúsculas**.

Obs.: - Você deve usar **className** e **htmlFor** como nomes de atributos XML em vez de **class** e **for**.

Isso é explicado na página oficial do React como -

Como JSX é JavaScript, identificadores como **class** e **for** são desencorajados como nomes de atributos XML. Em vez disso, os componentes React DOM esperam nomes de propriedade DOM, como **className** e **htmlFor**, respectivamente.

Estado (State)

Estado (State) é o lugar de onde vêm os dados. Devemos sempre tentar tornar nosso estado o mais simples possível e minimizar o número de componentes com estado. Se tivermos, por exemplo, dez componentes que precisam de dados do estado, devemos criar um componente de contêiner que manterá o estado de todos eles.

Usando estado

O código de exemplo a seguir mostra como criar um componente com monitoração de estado usando a sintaxe EcmaScript2016.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      header: "Header from state...",
      content: "Content from state..."
    }
  }
  render() {
    return (
      <div>
        <h1>{this.state.header}</h1>
        <h2>{this.state.content}</h2>
      </div>
    );
  }
}

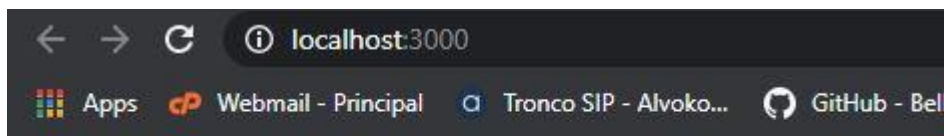
export default App;
```

index.js

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

ReactDOM.render(<App />,
document.getElementById('root'));
```

Isso produzirá o seguinte resultado.



Header from state...

Content from state...

Componentes

Neste passo, aprenderemos como combinar componentes para tornar o aplicativo mais fácil de manter. Esta abordagem permite atualizar e alterar seus componentes sem afetar o resto da página.

sem estado (stateless)

Nosso primeiro componente no exemplo a seguir é o **aplicativo**. Este componente é proprietário do **Cabeçalho** e **Conteúdo**. Estamos criando **Header** e **Content** separadamente e apenas adicionando-o dentro da árvore JSX em nosso componente de **aplicativo**. Apenas o componente do **aplicativo** precisa ser exportado.

App.js

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <Header/>
        <Content/>
      </div>
    );
  }
}

class Header extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
      </div>
    );
  }
}

class Content extends React.Component {
  render() {
    return (
      <div>
        <h2>Content</h2>
        <p>The content text!!!</p>
      </div>
    );
  }
}

export default App;
```

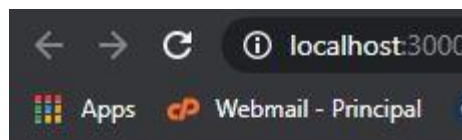

Para ser capaz de renderizar isso na página, precisamos importá-lo no arquivo **index.js** e chamar **ReactDOM.render ()** . Já fizemos isso enquanto configuramos o ambiente.

index.js

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

ReactDOM.render(<App />,
document.getElementById('root'));
```

O código acima irá gerar o seguinte resultado.



Header

Content

The content text!!!

com estado (stateFull)

Neste exemplo, definiremos o estado do componente proprietário (**App**). O componente **Header** é adicionado como no último exemplo, pois não precisa de nenhum estado. Em vez de tag de conteúdo, estamos criando elementos **table** e **tbody** , onde inseriremos dinamicamente **TableRow** para cada objeto do array de **dados** .

Pode-se ver que estamos usando a sintaxe de seta EcmaScript 2015 (=>), que parece muito mais limpa do que a antiga sintaxe JavaScript. Isso nos ajudará

a criar nossos elementos com menos linhas de código. É especialmente útil quando precisamos criar uma lista com muitos itens.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor() {
    super();
    this.state = {
      data: [
        {
          "id": 1,
          "name": "Foo",
          "age": "20"
        },
        {
          "id": 2,
          "name": "Bar",
          "age": "30"
        },
        {
          "id": 3,
          "name": "Baz",
          "age": "40"
        }
      ]
    }
  }
  render() {
    return (
      <div>
        <Header/>
      </div>
    );
  }
}
```

```

        <table>
          <tbody>
            {this.state.data.map((person, i) =>
<TableRow key = {i}
              data = {person} />)}
          </tbody>
        </table>
      </div>
    );
  }
}

class Header extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
      </div>
    );
  }
}

class TableRow extends React.Component {
  render() {
    return (
      <tr>
        <td>{this.props.data.id}</td>
        <td>{this.props.data.name}</td>
        <td>{this.props.data.age}</td>
      </tr>
    );
  }
}

export default App;

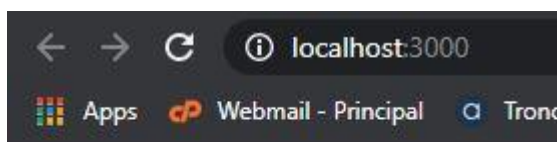
```

index.js

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

ReactDOM.render(<App/>,
document.getElementById('root'));
```

\obs.: Observe que estamos usando **key = {i}** dentro da função **map()**. Isso ajudará o React a atualizar apenas os elementos necessários, em vez de renderizar novamente a lista inteira quando algo mudar. É um grande aumento de desempenho para um grande número de elementos criados dinamicamente.



Header

1 Foo 20
2 Bar 30
3 Baz 40

Props

A principal diferença entre o estado e os props é que os **props** são imutáveis. É por isso que o componente do contêiner deve definir o estado que pode ser atualizado e alterado, enquanto os componentes filhos devem apenas passar dados do estado usando props.

Usando props

Quando precisamos de dados imutáveis em nosso componente, podemos apenas adicionar props à função **ReactDOM.render ()** em **index.js** e usá-la dentro de nosso componente.

App.js

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}

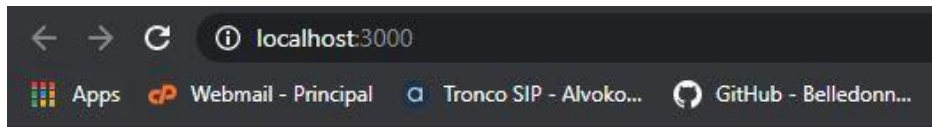
export default App;
```

index.js

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

ReactDOM.render(<App headerProp = "Header from
props..." contentProp = "Content
from props..." />, document.getElementById('root'));
```

Isso produzirá o seguinte resultado.



Header from props...

Content from props...

Props padrão

Você também pode definir valores de propriedade padrão diretamente no construtor do componente, em vez de adicioná-lo ao elemento `ReactDOM.render()`.

App.js

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}

App.defaultProps = {
  headerProp: " Header from props...",
  contentProp: "Content from props..."
}

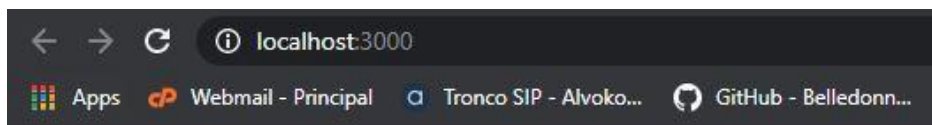
export default App;
```

index.js

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

ReactDOM.render(<App/>,
document.getElementById('root'));
```

A saída é a mesma de antes.



Header from props...

Content from props...

Estado (state) e props

O exemplo a seguir mostra como combinar **estado** e props em seu aplicativo. Estamos definindo o estado em nosso componente pai e passando-o para a árvore de componentes usando **props**. Dentro da função **render**, estamos definindo **headerProp** e **contentProp** usados em componentes filhos.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      header: "Header from props...",
      content: "Content from props..."
    }
  }
}
```

```

        }
    }
    render() {
        return (
            <div>
                <Header headerProp = {this.state.header}/>
                <Content contentProp =
{this.state.content}/>
            </div>
        );
    }
}

class Header extends React.Component {
    render() {
        return (
            <div>
                <h1>{this.props.headerProp}</h1>
            </div>
        );
    }
}

class Content extends React.Component {
    render() {
        return (
            <div>
                <h2>{this.props.contentProp}</h2>
            </div>
        );
    }
}

export default App;

```

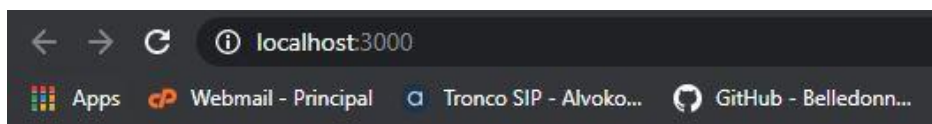
index.js

```
import React from 'react';
```



```
import ReactDOM from 'react-dom';  
import App from './App';  
  
ReactDOM.render(<App/>,  
document.getElementById('root'));
```

O resultado será novamente o mesmo que nos dois exemplos anteriores, a única coisa que é diferente é a fonte de nossos dados, que agora vem originalmente do **estado**. Quando quisermos atualizá-lo, precisamos apenas atualizar o estado e todos os componentes filhos serão atualizados. Mais sobre isso no passo Eventos.



Header from props...

Content from props...

Validação de props

A validação de propriedades é uma forma útil de forçar o uso correto dos componentes. Isso ajudará durante o desenvolvimento a evitar bugs e problemas futuros, uma vez que o aplicativo se tornar maior. Também torna o código mais legível, pois podemos ver como cada componente deve ser usado.

Validando Suportes

Neste exemplo, estamos criando o componente **App** com todos os **props** de que precisamos. **App.propTypes** é usado para validação de props. Se alguns dos props não estiverem usando o tipo correto que atribuímos, receberemos um aviso do console. Depois de especificar os padrões de validação, definiremos **App.defaultProps**.

App.js

```
import React from 'react';
import PropTypes from 'prop-types';

class App extends React.Component {
  render() {
    return (
      <div>
        <h3>Array: {this.props.propArray}</h3>
        <h3>Bool: {this.props.propBool ? "True..."
: "False..."}</h3>
        <h3>Func: {this.props.propFunc(3)}</h3>
        <h3>Number: {this.props.propNumber}</h3>
        <h3>String: {this.props.propString}</h3>
        <h3>Object:
{this.props.propObject.objectName1}</h3>
        <h3>Object:
{this.props.propObject.objectName2}</h3>
        <h3>Object:
{this.props.propObject.objectName3}</h3>
      </div>
    );
  }
}

App.propTypes = {
  propArray: PropTypes.array.isRequired,
  propBool: PropTypes.bool.isRequired,
  propFunc: PropTypes.func,
  propNumber: PropTypes.number,
  propString: PropTypes.string,
  propObject: PropTypes.object
}
```

```
App.defaultProps = {
  propArray: [1,2,3,4,5],
  propBool: true,
  propFunc: function(e){return e},
  propNumber: 1,
  propString: "String value...",

  propObject: {
    objectName1:"objectValue1",
    objectName2: "objectValue2",
    objectName3: "objectValue3"
  }
}

export default App;
```

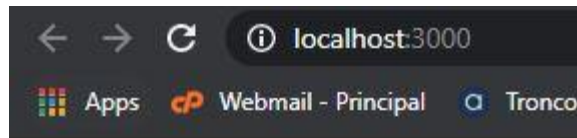
index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));
```



Array: 12345

Bool: True...

Func: 3

Number: 1

String: String value...

Object: objectValue1

Object: objectValue2

Object: objectValue3

Component API

Neste passo, vamos entender a API do componente React. Discutiremos três métodos: **setState ()**, **forceUpdate** e **ReactDOM.findDOMNode ()**. Em novas classes ES6, temos que vincular isso manualmente. Usaremos **this.method.bind (this)** nos exemplos.

Definir estado

O método **setState ()** é usado para atualizar o estado do componente. Este método não substituirá o estado, mas apenas adicionará alterações ao estado original.

```
import React from 'react';

class App extends React.Component {
  constructor() {
    super();

    this.state = {
      data: []
    }

    this.setStateHandler =
this.setStateHandler.bind(this);
  };
  setStateHandler() {
    var item = "setState..."
    var myArray = this.state.data.slice();
    myArray.push(item);
    this.setState({data: myArray})
  };
  render() {
    return (
      <div>
        <button onClick =
{this.setStateHandler}>SET STATE</button>
        <h4>State Array: {this.state.data}</h4>
      </div>
    );
  }
}
```

```
export default App;
```

index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));
```

Começamos com uma matriz vazia. Cada vez que clicarmos no botão, o estado será atualizado. Se clicarmos cinco vezes, obteremos a seguinte saída.



State Array: setState...setState...setState...

Forçar atualização

Às vezes, podemos querer atualizar o componente manualmente. Isso pode ser feito usando o método **forceUpdate ()**.

```
import React from 'react';

class App extends React.Component {
  constructor() {
    super();
  }
}
```

```

        this.forceUpdateHandler =
this.forceUpdateHandler.bind(this);

        };
        forceUpdateHandler() {
            this.forceUpdate();
        };
        render() {
            return (
                <div>
                    <button onClick =
{this.forceUpdateHandler}>FORCE UPDATE</button>
                    <h4>Random number: {Math.random()}</h4>
                </div>
            );
        }
    }
export default App;

```

index.js

```

import React from 'react';

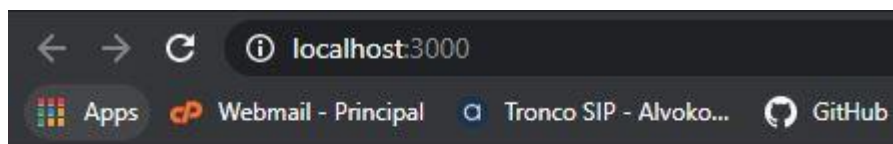
import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));

```

Estamos definindo um número aleatório que será atualizado toda vez que o botão for clicado.



Random number: 0.838340856106156

Encontrando o Dom Node

Para manipulação de DOM, podemos usar o método **ReactDOM.findDOMNode ()**. Primeiro, precisamos importar React .

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  constructor() {
    super();
    this.findDomNodeHandler =
this.findDomNodeHandler.bind(this);
  };
  findDomNodeHandler() {
    var myDiv = document.getElementById('myDiv');
    ReactDOM.findDOMNode(myDiv).style.color =
'green';
  }
  render() {
    return (
      <div>
        <button onClick =
{this.findDomNodeHandler}>FIND DOME NODE</button>
        <div id = "myDiv">NODE</div>
      </div>
    );
  }
}
```



```

    );
  }
}

export default App;

```

index.js

```

import React from 'react';

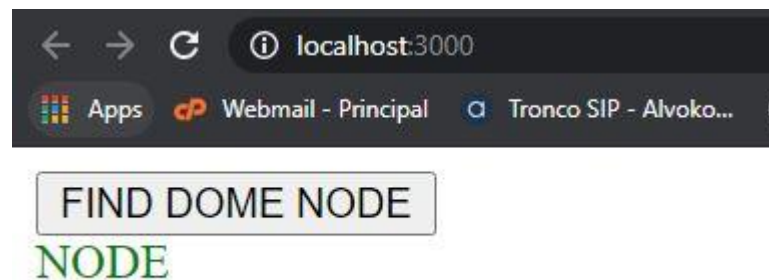
import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));

```

A cor do elemento **myDiv** muda para verde assim que o botão é clicado.



Obs.: - Desde a atualização 0.14, a maioria dos métodos API de componentes mais antigos foram descontinuados ou removidos para acomodar o ES6.

Ciclo de Vida do Componente

Neste passo, abordaremos os métodos de ciclo de vida do componente.

Métodos de Ciclo de Vida

- **componentWillMount** é executado antes da renderização, tanto no servidor quanto no cliente.

- **componentDidMount** é executado após a primeira renderização apenas no lado do cliente. É aqui que as solicitações AJAX e as atualizações de DOM ou de estado devem ocorrer. Este método também é usado para integração com outras estruturas JavaScript e quaisquer funções com execução atrasada, como **setTimeout** ou **setInterval** . Estamos usando para atualizar o estado para que possamos acionar os outros métodos de ciclo de vida.
- **componentWillReceiveProps** é chamado assim que os props são atualizados antes que outra renderização seja chamada. Nós o **acionamos a** partir de **setNewNumber** quando atualizamos o estado.
- **shouldComponentUpdate** deve retornar um valor **verdadeiro** ou **falso** . Isso determinará se o componente será atualizado ou não. Isso é definido como **verdadeiro** por padrão. Se você tiver certeza de que o componente não precisa ser renderizado depois que o **estado** ou os **props** forem atualizados, você pode retornar um valor **falso** .
- **componentWillUpdate** é chamado antes da renderização.
- **componentDidUpdate** é chamado logo após a renderização.
- **componentWillUnmount** é chamado depois que o componente é desmontado do dom. Estamos desmontando nosso componente no **index.js** .

No exemplo a seguir, definiremos o **estado** inicial na função do construtor. O **setNewnumber** é usado para atualizar o **estado** . Todos os métodos de ciclo de vida estão dentro do componente.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: 0
    }

    this.setNewNumber = this.setNewNumber.bind(this)
  };

  setNewNumber() {
    this.setState({data: this.state.data + 1})
  }
}
```

```

    }

    render() {
        return (
            <div>
                <button onClick =
{this.setNewNumber}>INCREMENT</button>
                <Content myNumber =
{this.state.data}></Content>
            </div>
        );
    }
}

class Content extends React.Component {
    componentWillMount() {
        console.log('Component WILL MOUNT!')
    }
    componentDidMount() {
        console.log('Component DID MOUNT!')
    }
    componentWillReceiveProps(newProps) {
        console.log('Component WILL RECIEVE PROPS!')
    }
    shouldComponentUpdate(newProps, newState) {
        return true;
    }
    componentWillUpdate(nextProps, nextState) {
        console.log('Component WILL UPDATE!');
    }
    componentDidUpdate(prevProps, prevState) {
        console.log('Component DID UPDATE!')
    }
    componentWillUnmount() {
        console.log('Component WILL UNMOUNT!')
    }
}

```

```

    render() {
      return (
        <div>
          <h3>{this.props.myNumber}</h3>
        </div>
      );
    }
  }
  export default App;

```

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

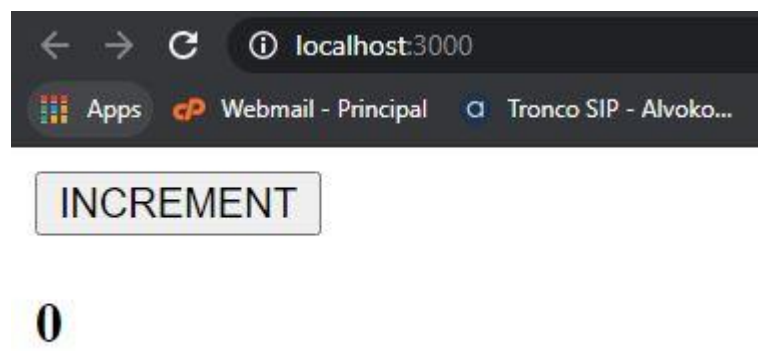
ReactDOM.render(<App/>,
document.getElementById('root'));

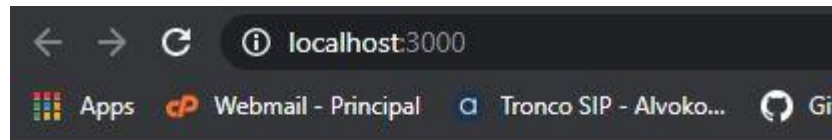
setTimeout(() => {

ReactDOM.unmountComponentAtNode(document.getElementById('root'));;}, 10000);

```

Após a renderização inicial, obteremos a seguinte tela.





2

```

Component DID MOUNT!
Component WILL RECIEVE PROPS!
Component WILL UPDATE!
Component DID UPDATE!
Component WILL RECIEVE PROPS!
Component WILL UPDATE!
Component DID UPDATE!
Component WILL RECIEVE PROPS!
Component WILL UPDATE!
Component DID UPDATE!
Component WILL RECIEVE PROPS!
Component WILL UPDATE!
Component DID UPDATE!
Component WILL RECIEVE PROPS!

```

Formulários

Neste passo, aprenderemos como usar formulários no React.

Exemplo

No exemplo a seguir, definiremos um formulário de entrada com **valor = {this.state.data}**. Isso permite atualizar o estado sempre que o valor de entrada muda. Estamos usando o evento **onChange** que observará as alterações de entrada e atualizará o estado de acordo.

App.js

```

import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState(e) {
    this.setState({data: e.target.value});
  }
  render() {
    return (
      <div>
        <input type = "text" value =
{this.state.data}
          onChange = {this.updateState} />
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}

export default App;

```

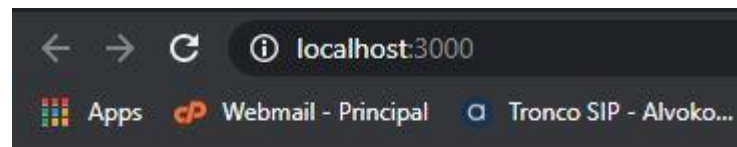
index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));

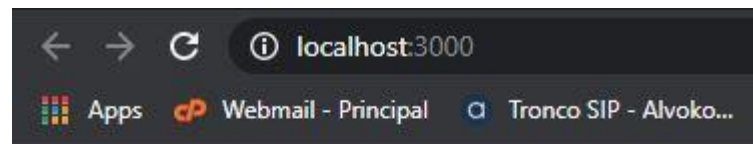
```



Initial data...

Initial data...

Quando o valor do texto de entrada muda, o estado é atualizado.



Ola Mundo

Ola Mundo

Eventos

Neste passo, aprenderemos como usar eventos.

Exemplo

Este é um exemplo simples em que usaremos apenas um componente. Estamos apenas adicionando o evento **onClick** que irá acionar a função **updateState** assim que o botão for clicado.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated...'})
  }
  render() {
    return (
      <div>
        <button onClick =
{this.updateState}>CLICK</button>
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}

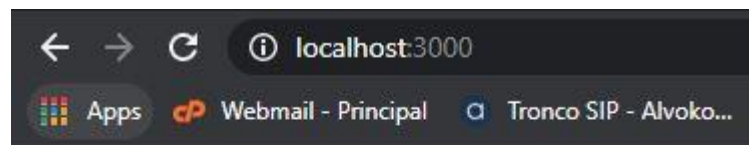
export default App;
```


index.js

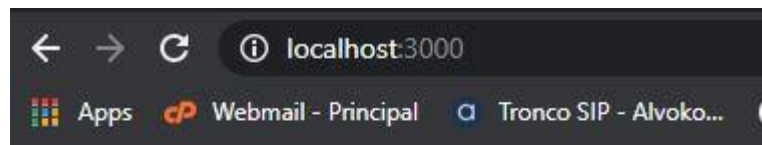
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));
```

Isso produzirá o seguinte resultado.



Initial data...



Data updated...

Child Events

Quando precisamos atualizar o **estado** do componente pai de seu filho, podemos criar um manipulador de eventos (**updateState**) no componente pai e passá-lo como um prop (**updateStateProp**) para o componente filho, onde podemos apenas chamá-lo.

App.js

```

import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated from the
child component...'})
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp =
{this.updateState}></Content>
      </div>
    );
  }
}

class Content extends React.Component {
  render() {
    return (
      <div>
        <button onClick =
{this.props.updateStateProp}>CLICK</button>
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}

```

```

    );
  }
}
export default App;

```

index.js

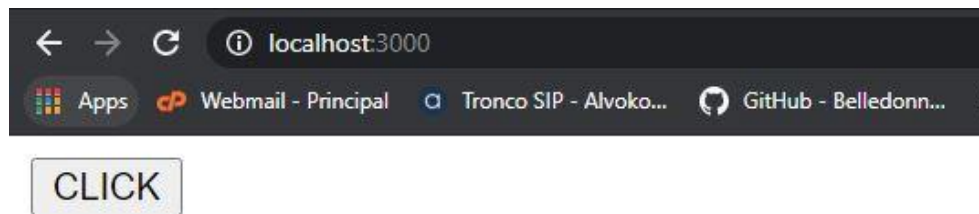
```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));

```

Isso produzirá o seguinte resultado.



Data updated from the child component...

Refs

O **ref** é usado para retornar uma referência ao elemento. **Refs** devem ser evitados na maioria dos casos, no entanto, eles podem ser úteis quando precisamos de medições DOM ou para adicionar métodos aos componentes.

Usando Refs

O exemplo a seguir mostra como usar refs para limpar o campo de entrada. A função **ClearInput** procura o elemento com o valor **ref = "myInput"**, redefine o estado e adiciona o foco a ele após o botão ser clicado.

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: ''
    }

    this.updateState = this.updateState.bind(this);
    this.clearInput = this.clearInput.bind(this);
  };

  updateState(e) {
    this.setState({data: e.target.value});
  }

  clearInput() {
    this.setState({data: ''});
    ReactDOM.findDOMNode(this.refs.myInput).focus();
  }

  render() {
    return (
      <div>
        <input value = {this.state.data} onChange
= {this.updateState}
        ref = "myInput"></input>
        <button onClick =
{this.clearInput}>CLEAR</button>
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}
```

```

        </div>
      );
    }
  }
}

export default App;

```

index.js

```

import React from 'react';

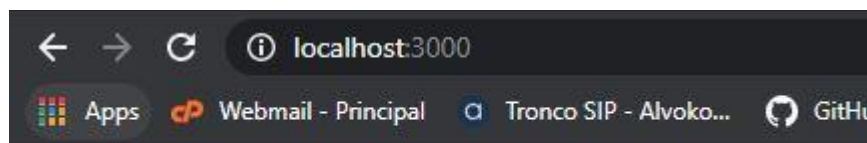
import ReactDOM from 'react-dom';

import App from './App';

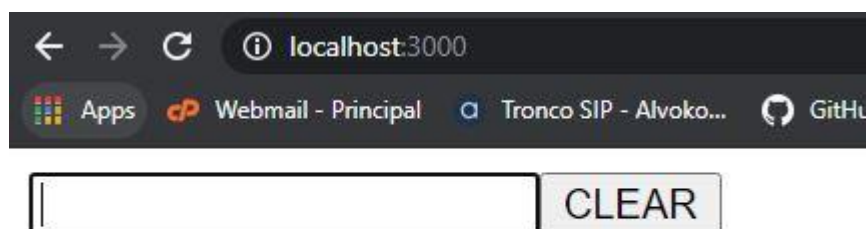
ReactDOM.render(<App/>,
document.getElementById('root'));

```

Assim que o botão for clicado, a **entrada** será apagada e focada.



Texto



Keys

As **keys** React são úteis ao trabalhar com componentes criados dinamicamente ou quando suas listas são alteradas pelos usuários. Definir o valor- **key** manterá seus componentes identificados de forma exclusiva após a alteração.

Usando Keys

Vamos criar elementos de **conteúdo** dinamicamente com índice exclusivo (i). A função de **mapa** criará três elementos de nosso array de **dados**. Como o valor da **key** precisa ser único para cada elemento, atribuiremos i como uma key para cada elemento criado.

App.js

```
import React from 'react';

class App extends React.Component {
  constructor() {
    super();

    this.state = {
      data:[
        {
          component: 'First...',
          id: 1
        },
        {
          component: 'Second...',
          id: 2
        },
        {
          component: 'Third...',
          id: 3
        }
      ]
    }
  }
}
```

```

        ]
    }
}

render() {
    return (
        <div>
            <div>
                {this.state.data.map((dynamicComponent,
i) => <Content
                                key = {i} componentData =
{dynamicComponent}/>)}
            </div>
        </div>
    );
}

class Content extends React.Component {
    render() {
        return (
            <div>

<div>{this.props.componentData.component}</div>
                <div>{this.props.componentData.id}</div>
            </div>
        );
    }
}

export default App;

```

index.js

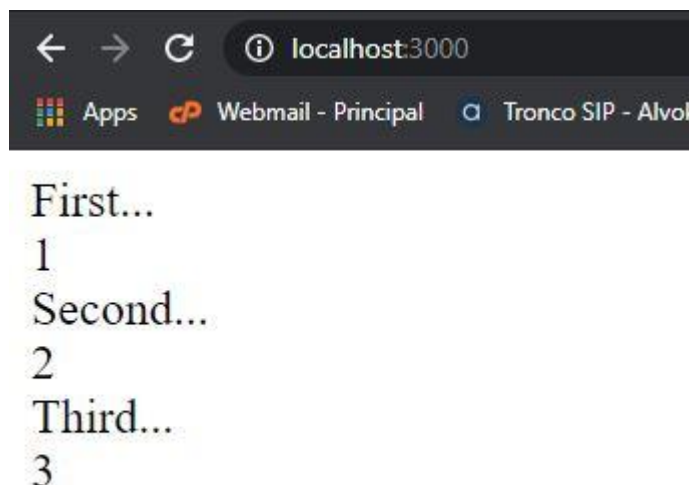
```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

```

```
ReactDOM.render(<App/>,
document.getElementById('root'));
```

Obteremos o seguinte resultado para os valores-key de cada elemento.



Se adicionarmos ou removermos alguns elementos no futuro ou alterarmos a ordem dos elementos criados dinamicamente, o React usará os valores- **key** para rastrear cada elemento.

Router (Roteador)

O roteamento é um processo no qual um usuário é direcionado para páginas diferentes com base em sua ação ou solicitação. O Roteador ReactJS é usado principalmente para desenvolver aplicativos da Web de página única. O React Router é usado para definir várias rotas no aplicativo. Quando um usuário digita uma URL específica no navegador e se esse caminho da URL corresponde a qualquer 'rota' dentro do arquivo do roteador, o usuário será redirecionado para essa rota específica.

React Router é um sistema de biblioteca padrão construído sobre o React e usado para criar roteamento no aplicativo React usando o pacote React Router. Ele fornece o URL síncrono no navegador com dados que serão exibidos na página da web. Ele mantém a estrutura padrão e o comportamento do

aplicativo e é usado principalmente para desenvolver aplicativos da Web de uma única página.

React Router

O React Router desempenha um papel importante para exibir várias visualizações em um único aplicativo de página. Sem o React Router, não é possível exibir várias visualizações nos aplicativos React. A maioria dos sites de mídia social, como Facebook, Instagram usa React Router para renderizar várias visualizações.

Instalação React Router

O React contém três pacotes diferentes para roteamento. Esses são:

- **react-router:** fornece os principais componentes e funções de roteamento para os aplicativos React Router.
- **react-router-native:** É usado para aplicativos móveis.
- **react-router-dom:** É usado para design de aplicações web.

Não é possível instalar o react-router diretamente na sua aplicação. Para usar o roteamento react, primeiro você precisa instalar os módulos react-router-dom em seu aplicativo. O comando abaixo é usado para instalar o react router dom.

```
$ npm install react-router-dom --save
```

Components no React Router

Existem dois tipos de componentes do roteador:

- **<BrowserRouter>:** É usado para lidar com o URL dinâmico.
- **<HashRouter>:** É usado para lidar com a solicitação estática.

Exemplo

Etapa 1: Em nosso projeto, criaremos mais dois componentes junto com **App.js** , que já está presente.

Criaremos os arquivos e implementaremos os códigos de acordo com a ordem abaixo:

Criação do arquivo About.js: na pasta src do nosso projeto clicaremos com o botão direito do mouse e selecionaremos a opção **New File; salve o arquivo com o nome About.js. Na sequencia, implemente o código abaixo:**

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

Criação do arquivo Contact.js: na pasta src do nosso projeto clicaremos com o botão direito do mouse e selecionaremos a opção **New File; salve o arquivo com o nome Contact.js. Na sequencia, implemente o código abaixo:**

Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact</h1>
  }
}
export default Contact
```

O arquivo App.js será composto pelo código abaixo:

App.js

```
import React from 'react'
class App extends React.Component {
  render() {
```

```

    return (
      <div>
        <h1>Home</h1>
      </div>
    )
  }
}
export default App

```

Etapa 2: Para roteamento, abra o arquivo `index.js` e importe todos os três arquivos componentes nele. Aqui, você precisa importar a linha: **import {Route, Link, BrowserRouter as Router}** de `'react-router-dom'` que nos ajuda a implementar o roteamento. Agora, nosso arquivo `index.js` se parece com o abaixo.

O que é rota?

É usado para definir e renderizar o componente com base no caminho especificado. Ele aceitará componentes e renderizará para definir o que deve ser exibido. Observe o arquivo ***index.js***:

Index.js

```

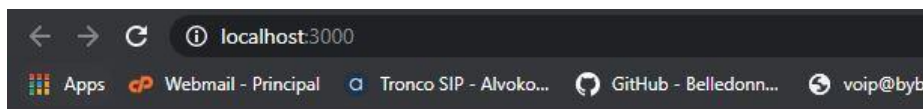
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)

```

```
ReactDOM.render(routing, document.getElementById('root'));
```

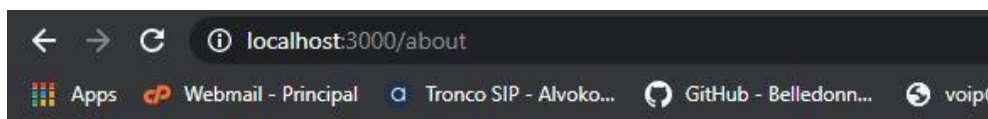
Etapas 3: Abra o **prompt de comando** , vá para o local do projeto e digite **npm start** . Você verá a seguinte tela.



React Router Example

Home

Agora, se você inserir **manualmente** no navegador: **localhost: 3000 / about** , verá que o componente **Sobre** é renderizado na tela



React Router Example

Home

About

Passo 4: Na tela acima, você pode ver que o componente **Home** ainda está renderizado. É porque o caminho inicial é **' / '** e sobre o caminho é **' / About '** , então você pode observar que a **barra** é comum em ambos os caminhos que renderizam os dois componentes.

Index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'))
);

```

Adicionando navegação usando o componente Link

Às vezes, queremos **vários** links em uma única página. Quando clicamos em qualquer **link** específico, ele deve carregar a página associada a esse caminho sem **recarregar** a página da web. Para fazer isso, precisamos importar o componente **<Link>** no arquivo **index.js**.

O que é o componente <Link>?

Este componente é usado para criar links que permitem **navegar** em diferentes **URLs** e renderizar seu conteúdo sem recarregar a página da web.

Exemplo

Index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';

```

```

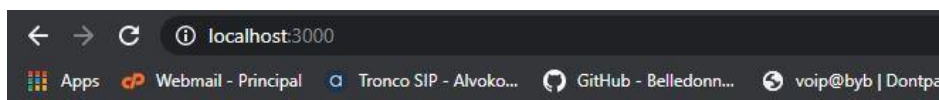
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>

ReactDOM.render(routing, document.getElementById('root'))
);

```

Resultado

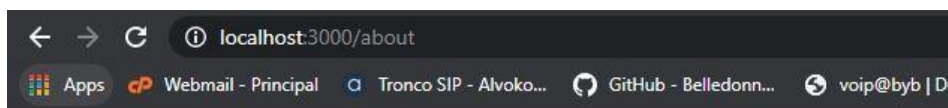


React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Bem-vindo a Tela Home

Depois de adicionar o Link, você pode ver que as rotas são renderizadas na tela. Agora, se você clicar em **About**, verá que a URL está mudando e o componente Sobre é renderizado.



React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Bem-vindo a tela About

Agora, precisamos adicionar alguns **estilos** ao Link. Para que, ao clicarmos em um determinado link, seja facilmente **identificado** qual Link está **ativo**. Para fazer isso, o react router fornece um novo truque **NavLink** em vez de **Link**. Agora, no arquivo index.js, substitua Link do Navlink e adicione as propriedades **activeStyle**. As propriedades activeStyle significam que quando clicamos no Link, ele deve ter um estilo específico para que possamos diferenciar qual está ativo no momento.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink }
  from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
```

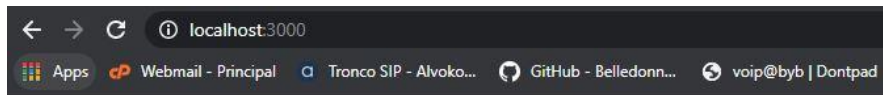
```

<h1>React Router Example</h1>
<ul>
  <li>
    <NavLink to="/" exact activeStyle={
      {color:'red'}
    }>Home</NavLink>
  </li>
  <li>
    <NavLink to="/about" exact activeStyle={
      {color:'green'}
    }>About</NavLink>
  </li>
  <li>
    <NavLink to="/contact" exact activeStyle={
      {color:'magenta'}
    }>Contact</NavLink>
  </li>
</ul>
<Route exact path="/" component={App} />
<Route path="/about" component={About} />
<Route path="/contact" component={Contact} />
</div>
</Router>
)
ReactDOM.render(routing, document.getElementById('root')
);

```

Resultado

Quando executamos o programa acima, obteremos a seguinte tela na qual podemos ver que o link **Home** é da cor vermelha e é o único link ativo no momento

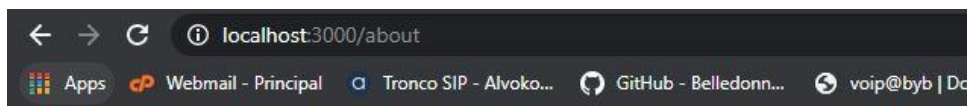


React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Bem-vindo a Tela Home

Agora, quando clicamos no link **About**, sua cor mostrada em **verde** é o link **ativo** no momento.



React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Bem-vindo a tela About

<Link> vs <NavLink>

O componente Link permite navegar pelas diferentes rotas nos sites, enquanto o componente NavLink é usado para adicionar estilos às rotas ativas.

React Router Switch

O componente `< Switch >` é usado para renderizar componentes apenas quando o caminho for **correspondido** . Caso contrário, ele retorna ao componente **não encontrado (notfound)**.

Para entender isso, primeiro, precisamos criar um componente **notfound** .

notfound.js

```
import React from 'react'
const Notfound = () => <h1>Not found</h1>
export default Notfound
```

Agora importe componente no arquivo index.js. Isso pode ser visto no código abaixo.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import Notfound from './notfound'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}}
            >Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
```

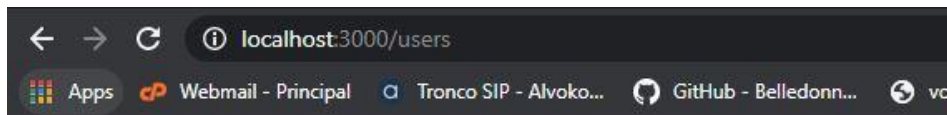
```

        {color:'green'}
      }>About</NavLink>
    </li>
    <li>
      <NavLink to="/contact" exact activeStyle={
        {color:'magenta'}
      }>Contact</NavLink>
    </li>
  </ul>
  <Switch>
    <Route exact path="/" component={App} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
    <Route component={NotFound} />
  </Switch>
</div>
</Router>
)
ReactDOM.render(routing, document.getElementById('root')
);

```

Resultado

Se inserirmos manualmente o caminho errado , o erro não encontrado será exibido.



React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Not found

Roteador React <Redirecionar>

Um componente `<Redirect>` é usado para redirecionar para outra rota em nosso aplicativo para manter os URLs antigos. Ele pode ser colocado em qualquer lugar na hierarquia da rota.

Encaminhamento aninhado no React

O roteamento aninhado permite que você renderize **sub-rotas** em seu aplicativo. Isso pode ser entendido no exemplo abaixo.

Exemplo

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink,
Switch } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import NotFound from './notfound'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
            {color:'green'}
          }>About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" exact activeStyle={
            {color:'magenta'}
          }>Contact</NavLink>
        </li>
      </ul>
    </div>
  </Router>
)
```

```

    </ul>
    <Switch>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
      <Route component={NotFound} />
    </Switch>
  </div>
</Router>
)
ReactDOM.render(routing, document.getElementById('root'))
);

```

No arquivo **contact.js**, precisamos importar o componente **React Router** para implementar os **sub - rotas** .

contact.js

```

import React from 'react'
import { Route, Link } from 'react-router-dom'

const Contacts = ({ match }) => <p>{match.params.id}</p>

class Contact extends React.Component {
  render() {
    //const { url } = this.props.match
    return (
      <div>
        <h1>Welcome to Contact Page</h1>
        <strong>Select contact Id</strong>
        <ul>
          <li>
            <Link to="/contact/1">Contacts 1 </Link>
          </li>
          <li>
            <Link to="/contact/2">Contacts 2 </Link>
          </li>
          <li>
            <Link to="/contact/3">Contacts 3 </Link>
          </li>
          <li>
            <Link to="/contact/4">Contacts 4 </Link>
          </li>
        </ul>
      </div>
    )
  }
}

```

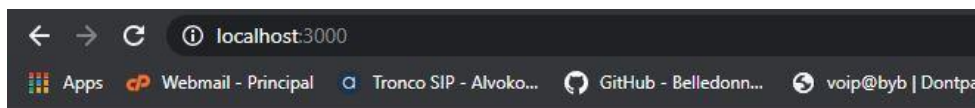
```

        <Route path="/contact/:id" component={Contacts}
      />
    </div>
  )
}
}
export default Contact

```

Resultado

Quando executarmos o programa acima, obteremos a seguinte saída.

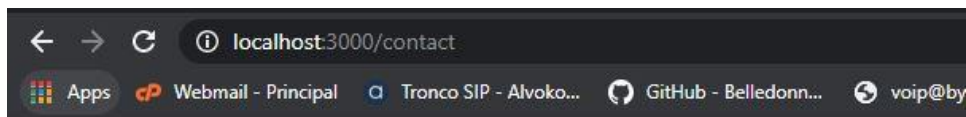


React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Bem-vindo a Tela Home

Após clicar no link **Contact**, obteremos a lista de contatos. Agora, selecionando qualquer contato, obteremos a saída correspondente. Isso pode ser mostrado no exemplo abaixo.



React Router Example

- [Home](#)
- [About](#)
- [Contact](#)

Welcome to Contact Page

Select contact Id

- [Contacts 1](#)
- [Contacts 2](#)
- [Contacts 3](#)
- [Contacts 4](#)

Benefícios do roteador React

Os principais benefícios do React Router são:

- não é necessário definir o histórico do navegador manualmente.
- Link é usado para navegar nos links internos do aplicativo. É semelhante à tag âncora<a>
- Ele usa o recurso Switch para renderização.
- O roteador precisa apenas de um único elemento filho.

Conceito Flux

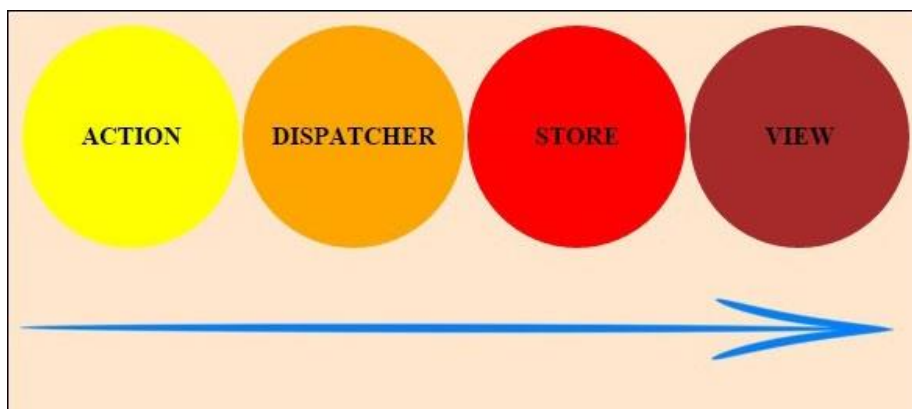
Flux é um conceito de programação, onde os dados são **unidirecionais**. Esses dados entram no aplicativo e fluem por ele em uma direção até que sejam renderizados na tela.

Elementos de fluxo

A seguir está uma explicação simples do conceito de **fluxo** . No próximo passo, aprenderemos como implementar isso no aplicativo.

- **Actions** - as ações são enviadas ao despachante para acionar o fluxo de dados.
- **Dispatcher** - Este é um hub central do aplicativo. Todos os dados são despachados e enviados para as stores.
- **Store** - Store é o local onde o estado e a lógica do aplicativo são mantidos. Cada store está mantendo um determinado estado e será atualizado quando necessário.
- **Visualização** - A **visualização** receberá dados da store e renderizará novamente o aplicativo.

O fluxo de dados é ilustrado na imagem a seguir.



Flux Pros

- O fluxo de dados direcional único é fácil de entender.
- O aplicativo é mais fácil de manter.
- As partes do aplicativo são desacopladas.

Usando Flux

Neste passo, aprenderemos como implementar o padrão de fluxo em aplicações React. Usaremos o framework **Redux** . O objetivo deste passo é apresentar o exemplo mais simples de cada peça necessária para conectar **Redux** e **React** .

Etapa 1 - Instalar Redux

Vamos instalar o Redux através da janela do **prompt de comando** .

```
C:\Users\username\Desktop\my-app>npm install react-redux
```

Etapa 2 - Criar arquivos e pastas

Nesta etapa, criaremos pastas e arquivos para nossas **ações (actions), redutores (reducers) e componentes (components)** . Depois que terminarmos com isso, é assim que a estrutura de pastas ficará.

```
C:\Users\username\Desktop\my-app>mkdir actions
```

```
C:\Users\username\Desktop\my-app>mkdir components
```

```
C:\Users\username\Desktop\ my-app>mkdir reducers
```

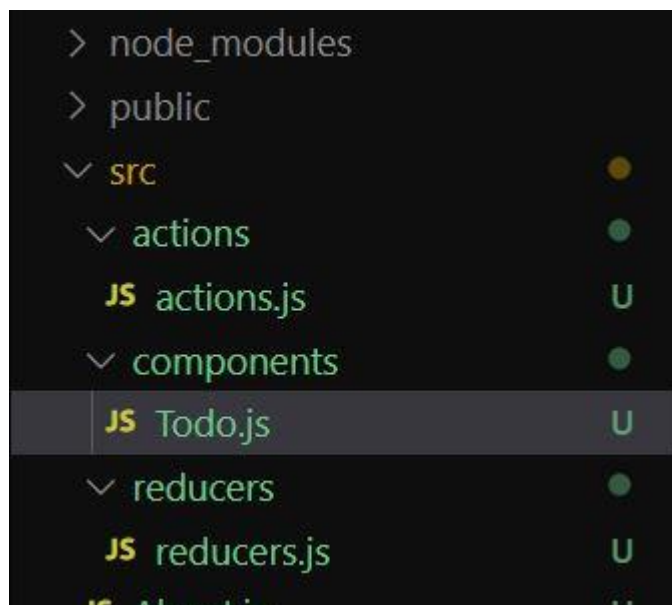
```
C:\Users\username\Desktop\ my-app > actions/actions.js
```

```
C:\Users\username\Desktop\ my-app> reducers/reducers.js
```

```
C:\Users\username\Desktop\ my-app> components/AddTodo.js
```

```
C:\Users\username\Desktop\reactApp>type nul > components/ToDo.js
```

```
C:\Users\username\Desktop\reactApp>type nul >  
components/ToDoList.js
```



Etapa 3 - Ações

Actions, como visto anteriormente, são objetos JavaScript que usam a propriedade **type** para informar sobre os dados que devem ser enviados para a store. Estamos definindo a ação **ADD_TODO** que será usada para adicionar um novo item à nossa lista. A função **addTodo** é um criador de ação que retorna nossa ação e define um **id** para cada item criado.

actions / actions.js

```
export const ADD_TODO = 'ADD_TODO'

let nextTodoId = 0;

export function addTodo(text) {
  return {
    type: ADD_TODO,
    id: nextTodoId++,
    text
  };
}
```

Etapa 4 – Redutores (Reducers)

Embora as ações apenas acionem mudanças no aplicativo, os **redutores** especificam essas mudanças. Estamos usando a instrução **switch** para pesquisar uma ação **ADD_TODO**. O redutor é uma função que usa dois parâmetros (**estado** e **ação**) para calcular e retornar um estado atualizado.

A primeira função será usada para criar um novo item, enquanto a segunda irá empurrar esse item para a lista. No final, estamos usando a função auxiliar **combineReducers**, onde podemos adicionar quaisquer novos redutores que possamos usar no futuro.

reducers / reducers.js

```
import { combineReducers } from 'redux'
import { ADD_TODO } from '../actions/actions'

function todo(state, action) {
  switch (action.type) {
    case ADD_TODO:
      return {
        id: action.id,
        text: action.text,
      }
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        todo(undefined, action)
      ]
    default:
```

```

        return state
    }
}
const todoApp = combineReducers({
  todos
})
export default todoApp

```

Etapa 5 - Armazenar

A store é um local que mantém o estado do aplicativo. É muito fácil criar uma store depois de ter redutores. Estamos passando a propriedade da store para o elemento **provedor (provider)**, que envolve nosso componente de rota.

index.js

```

import React from 'react'

import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'

import App from './App'
import todoApp from './reducers/reducers'

let store = createStore(todoApp)
let rootElement = document.getElementById('root')

render(
  <Provider store = {store}>
    <App />
  </Provider>,

  rootElement
)

```

Etapa 6 - Componente Raiz

O componente **App** é o componente raiz do app. Apenas o componente raiz deve estar ciente de um redux. A parte importante a notar é a função de **conexão** que é usada para conectar nosso **aplicativo** componente raiz à **store** .

Esta função leva a função de **seleção** como um argumento. A função Select pega o estado da store e retorna os props (**visibleTodos**) que podemos usar em nossos componentes.

App.js

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions/actions'

import AddTodo from '../components/AddTodo.js'
import TodoList from '../components/TodoList.js'

class App extends Component {
  render() {
    const { dispatch, visibleTodos } = this.props

    return (
      <div>
        <AddTodo onAddClick = {text
=>dispatch(addTodo(text))} />
        <TodoList todos = {visibleTodos}/>
      </div>
    )
  }
}

function select(state) {
  return {
    visibleTodos: state.todos
  }
}
```

```

}

export default connect(select)(App);

```

Etapa 7 - Outros componentes

Esses componentes não devem estar cientes do redux.

Crie o arquivo **AddTodo.js** dentro da pasta **components**

components / AddTodo.js

```

import React, { Component, PropTypes } from 'react'

export default class AddTodo extends Component {
  render() {
    return (
      <div>
        <input type = 'text' ref = 'input' />

        <button onClick = {(e) =>
this.handleClick(e)}>
          Add
        </button>
      </div>
    )
  }
  handleClick(e) {
    const node = this.refs.input
    const text = node.value.trim()
    this.props.onAddClick(text)
    node.value = ''
  }
}

```

components / Todo.js

```

import React, { Component, PropTypes } from 'react'

```

```
export default class Todo extends Component {
  render() {
    return (
      <li>
        {this.props.text}
      </li>
    )
  }
}
```

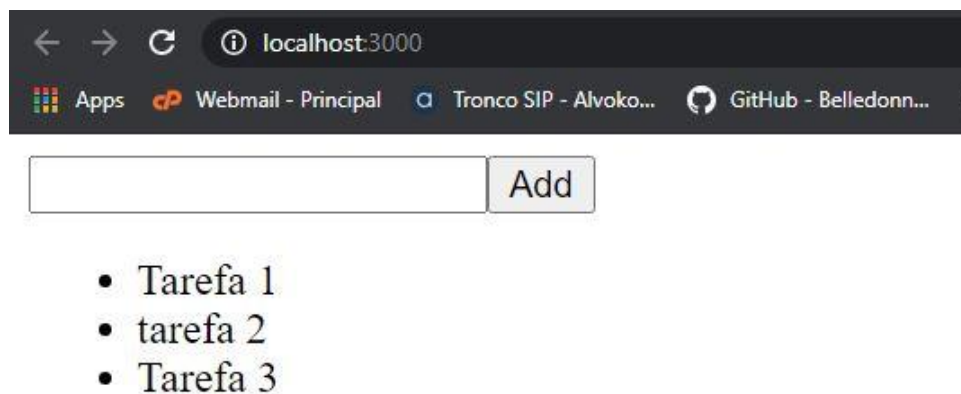
Criamos o arquivo `TodoList.js` dentro da pasta `components`

components / TodoList.js

```
import React, { Component, PropTypes } from 'react'
import Todo from './Todo.js'

export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.todos.map(todo =>
          <Todo
            key = {todo.id}
            {...todo}
          />
        )}
      </ul>
    )
  }
}
```

Quando iniciarmos o aplicativo, seremos capazes de adicionar itens à nossa lista.



React Redux

Redux é uma biblioteca JavaScript de código aberto usada para gerenciar o estado do aplicativo. React usa Redux para construir a interface do usuário. Foi apresentado pela primeira vez por Dan Abramov e Andrew Clark em 2015 .

React Redux é a ligação oficial do React para Redux. Ele permite que os componentes do React leiam dados de um Redux Store e enviem Actions para o Store para atualizar os dados. Redux ajuda os aplicativos a escalar, fornecendo uma maneira sensata de gerenciar o estado por meio de um modelo de fluxo de dados unidirecional. React Redux é conceitualmente simples. Ele assina a store Redux, verifica se os dados que seu componente deseja foram alterados e renderiza novamente seu componente.

Redux foi inspirado no Flux. Redux estudou a arquitetura do Flux e omitiu a complexidade desnecessária.

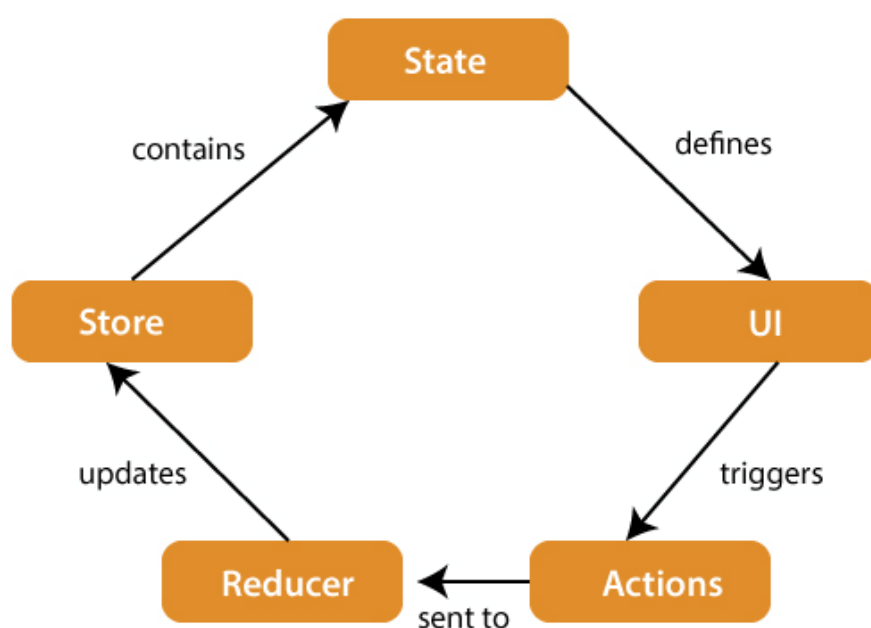
- Redux não possui o conceito de Dispatcher.
- Redux tem uma única store, enquanto o Flux tem muitas stores.
- Os objetos Action serão recebidos e tratados diretamente pela Store.

Por que usar o React Redux?

Os principais motivos para usar o React Redux são:

- React Redux é a interface de usuário oficial para o aplicativo react. Ele é mantido atualizado com todas as alterações da API para garantir que seus componentes React se comportem conforme o esperado.
- Encoraja uma boa arquitetura 'React'.
- Ele implementa muitas otimizações de desempenho internamente, o que permite que os componentes sejam renderizados novamente apenas quando realmente for necessário.

Arquitetura Redux



Os componentes da arquitetura Redux são explicados a seguir.

STORE: Uma store é um lugar onde todo o estado de seu aplicativo é listado. Ele gerencia o status do aplicativo e tem uma função de dispatcher (ação). É como um cérebro responsável por todas as partes móveis do Redux.

ACTION: A ação é enviada ou despachada da visualização, que são cargas úteis que podem ser lidas por Redutores. É um puro objeto criado para armazenar as informações do evento do usuário. Inclui informações como tipo de ação, hora da ocorrência, local da ocorrência, suas coordenadas e que estado pretende alterar.

REDUCER: o redutor lê as cargas úteis das ações e, em seguida, atualiza o armazenamento por meio do estado de acordo. É uma função pura retornar um novo estado do estado inicial.

Princípios do Redux

A previsibilidade do Redux é determinada pelos três princípios mais importantes, conforme abaixo:

- **Fonte Única**

O estado de todo o seu aplicativo é armazenado em uma árvore de objetos dentro de um único armazenamento. Como todo o estado do aplicativo é armazenado em uma única árvore, isso torna a depuração fácil e o desenvolvimento mais rápido.

- **O estado é somente leitura**

A única maneira de mudar o estado é emitir uma ação, um objeto que descreve o que aconteceu. Isso significa que ninguém pode alterar diretamente o estado do seu aplicativo.

- **As mudanças são feitas com funções puras**

Para especificar como a árvore de estado é transformada por ações, você escreve redutores puros. Um redutor é um local central onde ocorre a modificação de estado. Redutor é uma função que assume estado e ação como argumentos e retorna um estado recém-atualizado.

Instalação

Antes de instalar o Redux, **temos que instalar o Nodejs e o NPM** . Abaixo estão as instruções que o ajudarão a instalá-lo. Você pode pular essas etapas se já tiver Nodejs e NPM instalados em seu dispositivo.

- Visite <https://nodejs.org/> e instale o arquivo do pacote.
- Execute o instalador, siga as instruções e aceite o contrato de licença.
- Reinicie seu dispositivo para executá-lo.

- Você pode verificar a instalação bem-sucedida abrindo o prompt de comando e digite `node -v`. Isso mostrará a versão mais recente do Node em seu sistema.
- Para verificar se o npm foi instalado com êxito, você pode digitar `npm -v` que retorna a versão mais recente do npm.

Para instalar o redux, você pode seguir os passos abaixo:

Execute o seguinte comando em seu prompt de comando para instalar o Redux.

```
npm install --save redux
```

Para usar o Redux com o aplicativo react, você precisa instalar uma dependência adicional da seguinte forma:

```
npm install --save react-redux
```

Para instalar ferramentas de desenvolvedor para Redux, você precisa instalar o seguinte como dependência:

Execute o comando abaixo em seu prompt de comando para instalar Redux dev-tools.

```
npm install --save-dev redux-devtools
```

Se você não deseja instalar Redux dev tools e integrá-lo em seu projeto, você pode instalar **Redux DevTools Extension** para Chrome e Firefox.

Vamos assumir que o estado do nosso aplicativo é descrito por um objeto simples chamado **initialState** que é o seguinte:

```
const initialState = {  
  isLoading: false,  
  items: [],  
  hasError: false  
};
```

Cada pedaço de código em seu aplicativo não pode alterar esse estado. Para alterar o estado, você precisa “despachar” uma action.

O que é uma action?

Uma ACTION é um objeto simples que descreve a intenção de causar mudança com uma propriedade de tipo. Deve ter uma propriedade de tipo que informa o tipo de ação que está sendo executada. O comando para a ação é o seguinte:

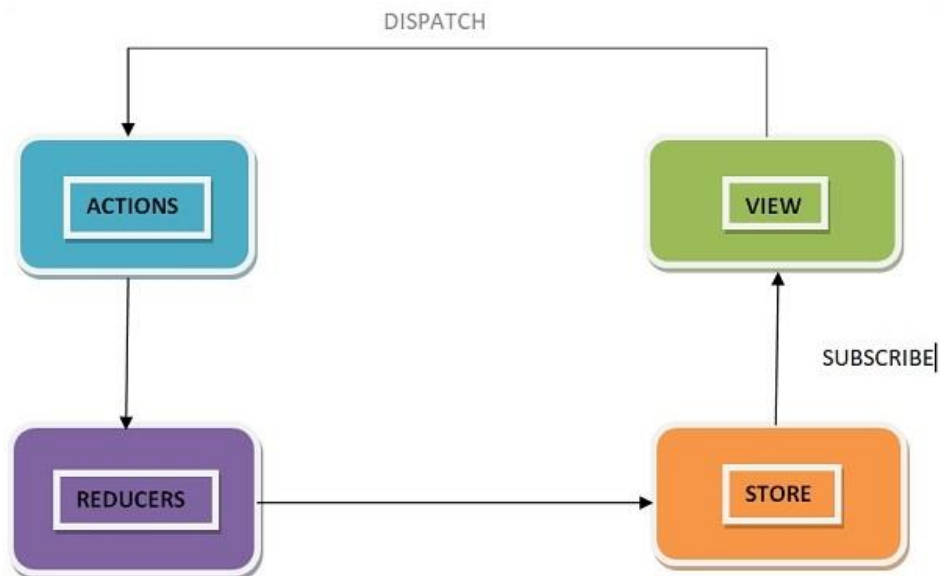
```
return {  
  type: 'ITEMS_REQUEST', //action type  
  isLoading: true //payload information  
}
```

Ações e estados são mantidos juntos por uma função chamada Redutor. Uma ação é despachada com a intenção de causar mudança. Essa mudança é realizada pelo redutor. Redutor é a única forma de alterar estados no Redux, tornando-o mais previsível, centralizado e depurável. Uma função redutora que lida com a ação 'ITEMS_REQUEST' é a seguinte :

```
const reducer = (state = initialState, action) => {  
  //es6 arrow function  
  switch (action.type) {  
    case 'ITEMS_REQUEST':  
      return Object.assign({}, state, {  
        isLoading: action.isLoading  
      })  
    default:  
      return state;  
  }  
}
```

Redux tem um único armazenamento que mantém o estado do aplicativo. Se você deseja dividir seu código com base na lógica de tratamento de dados, você deve começar a dividir seus redutores em vez de armazená-los no Redux.

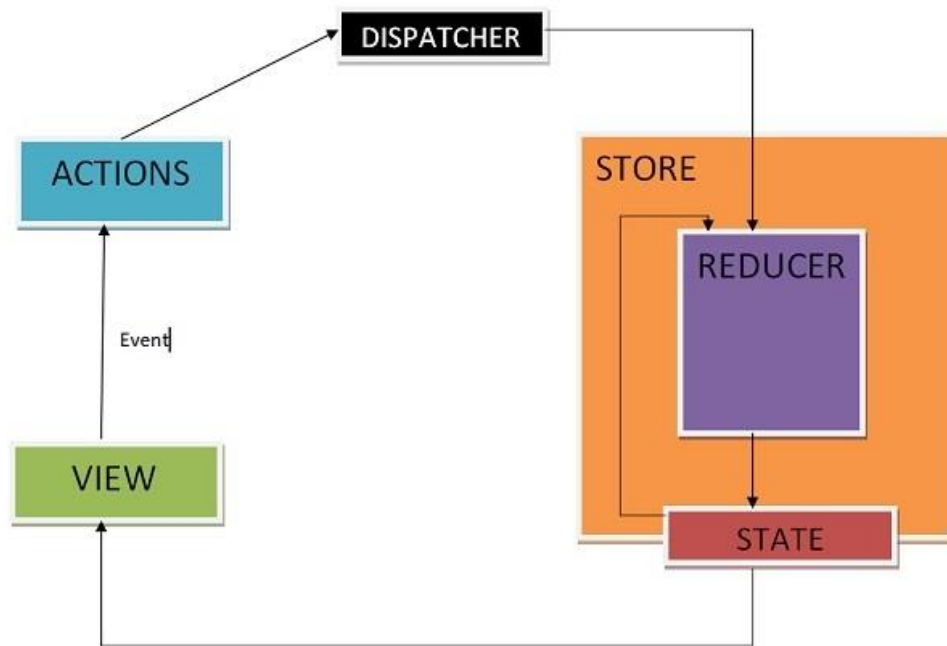
Os componentes Redux são os seguintes:



Fluxo de Dados

Redux segue o fluxo de dados unidirecional. Isso significa que os dados do seu aplicativo seguirão em um fluxo de dados de ligação unilateral. Conforme o aplicativo cresce e se torna complexo, é difícil reproduzir problemas e adicionar novos recursos se você não tiver controle sobre o estado do aplicativo.

Redux reduz a complexidade do código, reforçando a restrição sobre como e quando a atualização de estado pode acontecer. Dessa forma, é fácil gerenciar os estados atualizados. Já conhecemos as restrições conforme os três princípios do Redux. O diagrama a seguir ajudará você a entender melhor o fluxo de dados Redux:



- Uma ação é despachada quando um usuário interage com o aplicativo.
- A função do redutor raiz é chamada com o estado atual e a ação despachada. O redutor raiz pode dividir a tarefa entre funções redutoras menores, o que acaba retornando um novo estado.
- A store notifica a visualização executando suas funções de retorno de chamada.
- A visualização pode recuperar o estado atualizado e renderizar novamente.

Store

Uma store é uma árvore de objetos imutáveis no Redux. Uma store é um contêiner de estado que contém o estado do aplicativo. Redux pode ter apenas uma store em seu aplicativo. Sempre que uma store é criada no Redux, você precisa especificar o redutor.

Vamos ver como podemos criar uma store usando o método **createStore** do Redux. É necessário importar o pacote createStore da biblioteca Redux que suporta o processo de criação de store conforme mostrado abaixo:

```
import { createStore } from 'redux';

import reducer from './reducers/reducer'

const store = createStore(reducer);
```

Uma função `createStore` pode ter três argumentos. A seguir está a sintaxe:

```
createStore(reducer, [preloadedState], [enhancer])
```

Um redutor é uma função que retorna o próximo estado do aplicativo. Um `preloadedState` é um argumento opcional e é o estado inicial do seu aplicativo. Um realçador também é um argumento opcional. Isso o ajudará a aprimorar a store com recursos de terceiros.

Uma store tem três métodos importantes, conforme mostrado abaixo:

getState

Ele ajuda você a recuperar o estado atual de sua store Redux.

A sintaxe para `getState` é a seguinte:

```
store.getState()
```

Dispatch

Ele permite que você despache uma ação para alterar um estado em seu aplicativo.

A sintaxe para envio é a seguinte:

```
store.dispatch({type: 'ITEMS_REQUEST'})
```

subscribe

Ele ajuda você a registrar um retorno de chamada que a store Redux chamará quando uma ação for despachada. Assim que o estado Redux for atualizado, a visualização será renderizada novamente automaticamente.

A sintaxe para envio é a seguinte:

```
store.subscribe(()=>{ console.log(store.getState());})
```

Observe que a função de inscrição retorna uma função para cancelar a inscrição do ouvinte. Para cancelar a inscrição do ouvinte, podemos usar o código abaixo -

```
const unsubscribe =
store.subscribe(()=>{console.log(store.getState());});
unsubscribe();
```

Fundamentos de Actions

Actions são a única fonte de informação para a loja conforme documentação oficial do Redux. Ele carrega uma carga útil de informações de seu aplicativo para o armazenamento.

Conforme discutido anteriormente, as ações são objetos JavaScript simples que devem ter um atributo type para indicar o tipo de ação executada. Isso nos conta o que aconteceu. Os tipos devem ser definidos como constantes de string em sua aplicação, conforme mostrado abaixo:

```
const ITEMS_REQUEST = 'ITEMS_REQUEST';
```

Além deste atributo de tipo, a estrutura de um objeto de ação é totalmente de responsabilidade do desenvolvedor. Recomenda-se manter seu objeto de ação o mais leve possível e passar apenas as informações necessárias.

Para causar qualquer mudança na loja, você precisa despachar uma ação primeiro usando a função `store.dispatch()`. O objeto de ação é o seguinte -

```
{ type: GET_ORDER_STATUS , payload: {orderId,userId }
}
```

```
{ type: GET_WISHLIST_ITEMS, payload: userId }
```

Criadores de actions

Os criadores de action são as funções que encapsulam o processo de criação de um objeto de ação. Essas funções simplesmente retornam um objeto

Js simples que é uma ação. Promove a escrita de código limpo e ajuda a alcançar a capacidade de reutilização.

Deixe-nos aprender sobre o criador de ação que permite enviar uma ação, 'ITEMS_REQUEST', que solicita os dados da lista de itens de produto do servidor. Enquanto isso, o estado **isLoading** torna-se verdadeiro no redutor no tipo de ação 'ITEMS_REQUEST' para indicar que os itens estão sendo carregados e os dados ainda não foram recebidos do servidor.

Inicialmente, o estado **isLoading** era falso no objeto **initialState**, assumindo que nada estava carregando. Quando os dados são recebidos no navegador, o estado **isLoading** será retornado como falso no tipo de ação 'ITEMS_REQUEST_SUCCESS' no redutor correspondente. Este estado pode ser usado como um suporte em componentes de reação para exibir o carregador / mensagem em sua página enquanto a solicitação de dados está ativada. O criador da ação é o seguinte:

```
const ITEMS_REQUEST = 'ITEMS_REQUEST' ;
const ITEMS_REQUEST_SUCCESS = 'ITEMS_REQUEST_SUCCESS'
;

export function itemsRequest(bool, startIndex, endIndex)
{
    let payload = {
        isLoading: bool,
        startIndex,
        endIndex
    }
    return {
        type: ITEMS_REQUEST,
        payload
    }
}

export function itemsRequestSuccess(bool) {
    return {
        type: ITEMS_REQUEST_SUCCESS,
```

```

        isLoading: bool,
      }
    }
  }

```

Para invocar uma função de dispatcher, você precisa passar a ação como um argumento para a função de dispatcher.

```

dispatch(itemsRequest(true, 1, 20));
dispatch(itemsRequestSuccess(false));

```

Você pode despachar uma ação diretamente usando `store.dispatch()`. No entanto, é mais provável que você o acesse com o método auxiliar react-Redux chamado **connect()**. Você também pode usar o método **bindActionCreators()** para vincular muitos criadores de ação à função de dispatcher.

funções puras

Uma função é um processo que recebe entradas chamadas de argumentos e produz alguma saída conhecida como valor de retorno. Uma função é chamada pura se obedecer às seguintes regras:

- Uma função retorna o mesmo resultado para os mesmos argumentos.
- Sua avaliação não tem efeitos colaterais, ou seja, não altera os dados de entrada.
- Sem mutação de variáveis locais e globais.
- Não depende do estado externo como uma variável global.

Tomemos o exemplo de uma função que retorna duas vezes o valor passado como entrada para a função. Em geral, é escrito como $f(x) \Rightarrow x * 2$. Se uma função for chamada com um valor de argumento 2, a saída seria 4, $f(2) \Rightarrow 4$.

Vamos escrever a definição da função em JavaScript conforme mostrado abaixo:

```

const double = x => x*2; // es6 arrow function

console.log(double(2)); // 4

```

Aqui, o dobro é uma função pura.

De acordo com os três princípios no Redux, as alterações devem ser feitas por uma função pura, ou seja, redutor no Redux. Agora, surge uma questão de por que um redutor deve ser uma função pura.

Suponha que você deseja despachar uma ação cujo tipo é **'ADD_TO_CART_SUCCESS'** para adicionar um item ao seu aplicativo de carrinho de compras clicando no botão adicionar ao carrinho.

Vamos supor que o redutor está adicionando um item ao seu carrinho conforme mostrado abaixo:

```
const initialState = {
  isAddedToCart: false;
}

const addToCartReducer = (state = initialState,
action) => { //es6 arrow function
  switch (action.type) {
    case 'ADD_TO_CART_SUCCESS' :
      state.isAddedToCart = !state.isAddedToCart;
      //original object altered
      return state;
    default:
      return state;
  }
}

export default addToCartReducer ;
```

Vamos supor que **isAddedToCart** é uma propriedade no objeto de estado que permite decidir quando desabilitar o botão 'adicionar ao carrinho' para o item, retornando um valor booleano **'verdadeiro ou falso'** . Isso evita que o usuário adicione o mesmo produto várias vezes. Agora, em vez de retornar um novo objeto, estamos alterando a prop **isAddedToCart** no estado como acima. Agora, se tentarmos adicionar um item ao carrinho, nada acontece. O botão Adicionar ao carrinho não será desativado.

A razão para este comportamento é a seguinte -

Redux compara objetos novos e antigos pela localização de memória de ambos os objetos. Ele espera um novo objeto do redutor se alguma mudança acontecer. E também espera obter o objeto antigo de volta se nenhuma mudança ocorrer. Nesse caso, é o mesmo. Por este motivo, Redux assume que nada aconteceu.

Portanto, é necessário que um redutor seja uma função pura no Redux. A seguir está uma maneira de escrever sem mutação :

```
const initialState = {
  isAddedToCart: false;
}

const addToCartReducer = (state = initialState,
action) => { //es6 arrow function
  switch (action.type) {
    case 'ADD_TO_CART_SUCCESS' :
      return {
        ...state,
        isAddedToCart: !state.isAddedToCart
      }
    default:
      return state;
  }
}

export default addToCartReducer;
```

Reducers

Reducers são uma função pura no Redux. Funções puras são previsíveis. Redutores são a única maneira de alterar estados no Redux. É o único lugar onde você pode escrever lógica e cálculos. A função redutora aceitará o estado anterior do aplicativo e da ação despachada, calculará o próximo estado e retornará o novo objeto.

As seguintes coisas nunca devem ser realizadas dentro do redutor:

- Argumentos de mutação de funções
- Chamadas API e lógica de roteamento
- Chamando uma função não pura, por exemplo, `Math.random ()`

A seguir está a sintaxe de um redutor:

```
(state, action) => newState
```

Vamos continuar com o exemplo de exibição da lista de itens de produtos em uma página da web, discutida no módulo criadores de ações. Vejamos a seguir como escrever seu redutor.

```
const initialState = {
  isLoading: false,
  items: []
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ITEMS_REQUEST':
      return Object.assign({}, state, {
        isLoading: action.payload.isLoading
      })
    case 'ITEMS_REQUEST_SUCCESS':
      return Object.assign({}, state, {
        items: state.items.concat(action.items),
        isLoading: action.isLoading
      })
    default:
      return state;
  }
}

export default reducer;
```

Em primeiro lugar, se você não definir o estado como 'initialState', Redux chama redutor com o estado indefinido. Neste exemplo de código, a função `concat ()` do JavaScript é usada em 'ITEMS_REQUEST_SUCCESS', o que não altera a matriz existente; em vez disso, retorna uma nova matriz.

Desta forma, você pode evitar a mutação do estado. Nunca escreva diretamente para o estado. Em 'ITEMS_REQUEST', temos que definir o valor do estado da ação recebida.

Já foi discutido que podemos escrever nossa lógica no redutor e podemos dividi-la com base nos dados lógicos. Vamos ver como podemos dividir os redutores e combiná-los como redutor de raiz ao lidar com uma grande aplicação.

Suponha que queremos criar uma página da web onde um usuário possa acessar o status do pedido do produto e ver as informações da lista de desejos. Podemos separar a lógica em diferentes arquivos de redutores e fazê-los trabalhar de forma independente. Vamos supor que a ação GET_ORDER_STATUS seja despachada para obter o status do pedido correspondente a algum ID de pedido e ID de usuário.

```
/reducer/orderStatusReducer.js
import { GET_ORDER_STATUS } from
'../constants/appConstant';
export default function (state = {} , action) {
  switch(action.type) {
    case GET_ORDER_STATUS:
      return { ...state, orderStatusData:
action.payload.orderStatus };
    default:
      return state;
  }
}
```

Da mesma forma, suponha que a ação GET_WISHLIST_ITEMS seja despachada para obter as informações da lista de desejos do usuário em relação a um usuário.

```
/reducer/getWishlistDataReducer.js
```

```

import { GET_WISHLIST_ITEMS } from
'../constants/appConstant';

export default function (state = {}, action) {
  switch(action.type) {
    case GET_WISHLIST_ITEMS:
      return { ...state, wishlistData:
action.payload.wishlistData };
    default:
      return state;
  }
}

```

Agora, podemos combinar os dois redutores usando o utilitário Redux `combineReducers`. Os `combineReducers` geram uma função que retorna um objeto cujos valores são funções redutoras diferentes. Você pode importar todos os redutores no arquivo do redutor de índice e combiná-los como um objeto com seus respectivos nomes.

```

/reducer/index.js
import { combineReducers } from 'redux';
import OrderStatusReducer from './orderStatusReducer';
import GetWishlistDataReducer from
'./getWishlistDataReducer';

const rootReducer = combineReducers ({
  orderStatusReducer: OrderStatusReducer,
  getWishlistDataReducer: GetWishlistDataReducer
});
export default rootReducer;

```

Agora, você pode passar este `rootReducer` para o método `createStore` da seguinte maneira:

```

const store = createStore(rootReducer);

```

Middleware

O próprio Redux é síncrono, então, como as operações **assíncronas**, como **solicitação de rede**, funcionam com o Redux? Aqui, os middlewares são úteis. Conforme discutido anteriormente, os redutores são o local onde toda a lógica de execução é escrita. O redutor não tem nada a ver com quem o executa, quanto tempo está levando ou registrando o estado do aplicativo antes e depois da ação ser despachada.

Nesse caso, a função de middleware Redux fornece um meio para interagir com a ação despachada antes que alcancem o redutor. Funções de middleware personalizadas podem ser criadas escrevendo funções de alta ordem (uma função que retorna outra função), que envolve alguma lógica. Vários middlewares podem ser combinados para adicionar novas funcionalidades e cada middleware não requer nenhum conhecimento do que veio antes e depois. Você pode imaginar middlewares em algum lugar entre ação despachada e redutor.

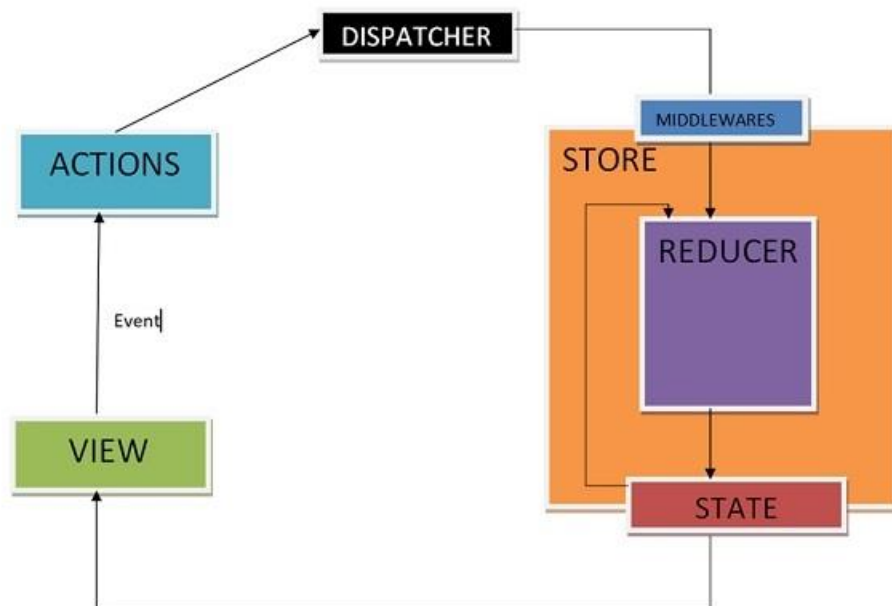
Normalmente, middlewares são usados para lidar com ações assíncronas em seu aplicativo. Redux fornece uma API chamada `applyMiddleware` que nos permite usar middleware customizado, bem como middlewares Redux como `redux-thunk` e `redux-promessa`. Aplica middlewares para armazenar. A sintaxe de uso da API `applyMiddleware` é :

```
applyMiddleware(...middleware)
```

E isso pode ser aplicado para armazenar da seguinte forma:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/index';

const store = createStore(rootReducer,
  applyMiddleware(thunk));
```

Middlewares permitem que você escreva um despachante de ação que retorna uma função em vez de um objeto de ação. Um exemplo para o mesmo é mostrado abaixo:

```
function getUser() {
  return function() {
    return axios.get('/get_user_details');
  };
}
```

O envio condicional pode ser escrito dentro do middleware. Cada middleware recebe o dispatcher da loja para que possam despachar uma nova ação, e `getState` funciona como argumentos para que possam acessar o estado atual e retornar uma função. Qualquer valor de retorno de uma função interna estará disponível como o valor da própria função de dispatcher.

A seguir está a sintaxe de um middleware:

```
({ getState, dispatch }) => next => action
```

A função `getState` é útil para decidir se novos dados devem ser buscados ou se o resultado do cache deve ser retornado, dependendo do estado atual.

Vejamos um exemplo de função de log de middleware personalizado. Ele simplesmente registra a ação e o novo estado.

```
import { createStore, applyMiddleware } from 'redux'

import userLogin from './reducers'

function logger({ getState }) {

  return next => action => {

    console.log('action', action);

    const returnVal = next(action);

    console.log('state when action is dispatched',
getState());

    return returnVal;

  }

}
```

Agora aplique o middleware logger para a loja, escrevendo a seguinte linha de código

```
const store = createStore(userLogin , initialState=[ ]
, applyMiddleware(logger));
```

Despache uma ação para verificar a ação despachada e o novo estado usando o código abaixo:

```
store.dispatch({

  type: 'ITEMS_REQUEST',

  isLoading: true

})
```

Outro exemplo de middleware onde você pode controlar quando mostrar ou ocultar o carregador é fornecido abaixo. Este middleware mostra o carregador

quando você está solicitando qualquer recurso e o oculta quando a solicitação do recurso é concluída.

```
import isPromise from 'is-promise';

function loaderHandler({ dispatch }) {
  return next => action => {
    if (isPromise(action)) {
      dispatch({ type: 'SHOW_LOADER' });
      action
        .then(() => dispatch({ type: 'HIDE_LOADER'
}))
        .catch(() => dispatch({ type:
'HIDE_LOADER' }));
    }
    return next(action);
  };
}

const store = createStore(
  userLogin , initialState = [ ] ,
  applyMiddleware(loaderHandler)
);
```

Usando React-Redux

Abaixo vamos implementar um exemplo de aplicação React-Redux. Você O código de amostra para aumentar ou diminuir o contador é fornecido abaixo:

Este é o arquivo raiz – index.js localizado na pasta src - responsável pela criação do store e renderização do nosso componente react app.

src/index.js

```
import React from 'react'

import { render } from 'react-dom'

import { Provider } from 'react-redux'

import { createStore } from 'redux';

import reducer from '../src/reducer/index'

import App from '../src/App'

import './index.css';

const store = createStore(

  reducer,

  window.__REDUX_DEVTOOLS_EXTENSION__ &&

  window.__REDUX_DEVTOOLS_EXTENSION__()

)

render(

  <Provider store = {store}>

    <App />

  </Provider>, document.getElementById('root')

)
```

Este é o nosso componente raiz do react. É responsável por renderizar o componente do contêiner do contador como componente-filho.

src/app.js

```
import React, { Component } from 'react';

import './App.css';

import Counter from '../src/container/counterContainer';

class App extends Component {

  render() {

    return (

      <div className = "App">

        <header className = "App-header">

          <Counter/>

        </header>

      </div>

    );

  }

}

export default App;
```

Na sequencia, está o componente do contêiner que é responsável por fornecer o Redux State para o react component. Para implementar esse código vamos criar uma nova pasta em **src/**. Clique com o botão direito do mouse na pasta **src/** e escolha a opção **New Folder**, dê o nome de **container**. Depois de criar essa pasta, selecione-a e, com o botão direito do mouse, crie um novo

arquivo dentro dela; o nome do arquivo será **counterContainer.js**: Na sequência, dentro do novo arquivo criado, implemente o código abaixo:

src/container/counterContainer.js

```
import { connect } from 'react-redux'

import Counter from '../component/counter'

import { increment, decrement, reset } from '../actions';

const mapStateToProps = (state) => {

  return {

    counter: state

  };

};

const mapDispatchToProps = (dispatch) => {

  return {

    increment: () => dispatch(increment()),

    decrement: () => dispatch(decrement()),

    reset: () => dispatch(reset())

  };

};

export default connect(mapStateToProps,
mapDispatchToProps)(Counter);
```

Agora, implementaremos o react component responsável pela visualização. Para implementar esse código vamos criar uma nova pasta em

src/. Clique com o botão direito do mouse na pasta **src/** e escolha a opção **New Folder**, dê o nome de **component**. Depois de criar essa pasta, selecione-a e, com o botão direito do mouse, crie um novo arquivo dentro dela; o nome do arquivo será **counter.js**: Na sequência, dentro do novo arquivo criado, implemente o código abaixo:

src/component/counterContainer.js

```
import React, { Component } from 'react';

class Counter extends Component {

  render() {

    const {counter,increment,decrement,reset} = this.props;

    return (

      <div className = "App">

        <div>{counter}</div>

        <div>

          <button  onClick  =  {increment}>INCREMENT  BY
1</button>

          </div>

          <div>

            <button  onClick  =  {decrement}>DECREMENT  BY
1</button>

            </div>

            <button onClick = {reset}>RESET</button>

          </div>

        );

      )
    }
```

```
}

export default Counter;
```

A seguir estão os action creators responsáveis por criar uma action. Para implementar esse código vamos criar uma nova pasta em **src/**. Clique com o botão direito do mouse na pasta **src/** e escolha a opção **New Folder**, dê o nome de **actions**. Depois de criar essa pasta, selecione-a e, com o botão direito do mouse, crie um novo arquivo dentro dela; o nome do arquivo será **index.js**: Na sequência, dentro do novo arquivo criado, implemente o código abaixo:

src/actions/index.js

```
export function increment() {

  return {

    type: 'INCREMENT'

  }

}

export function decrement() {

  return {

    type: 'DECREMENT'

  }

}

export function reset() {

  return { type: 'RESET' }

}
```

Abaixo, observados o código do arquivo **reducer** que é responsável por atualizar o **redux state**. Para implementar esse código vamos criar uma nova pasta em **src/**. Clique com o botão direito do mouse na pasta **src/** e escolha a

opção **New Folder**, dê o nome de **reducer**. Depois de criar essa pasta, selecione-a e, com o botão direito do mouse, crie um novo arquivo dentro dela; o nome do arquivo será **index.js**: Na sequência, dentro do novo arquivo criado, implemente o código abaixo:

src/reducer/index.js

```
const reducer = (state = 0, action) => {  
  
  switch (action.type) {  
  
    case 'INCREMENT': return state + 1  
  
    case 'DECREMENT': return state - 1  
  
    case 'RESET' : return 0 default: return state  
  
  }  
  
}  
  
export default reducer;
```

Execute sua aplicação e, ao observar o resultado, encontraremos algo semelhante ao exibido abaixo:



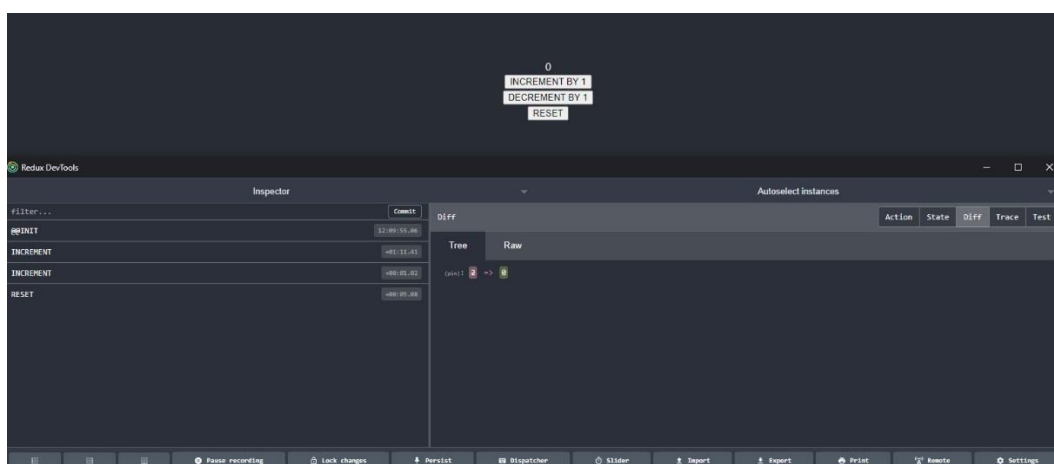
Ao clicar no botão Increment, duas vezes, a tela de saída será como mostrado abaixo:



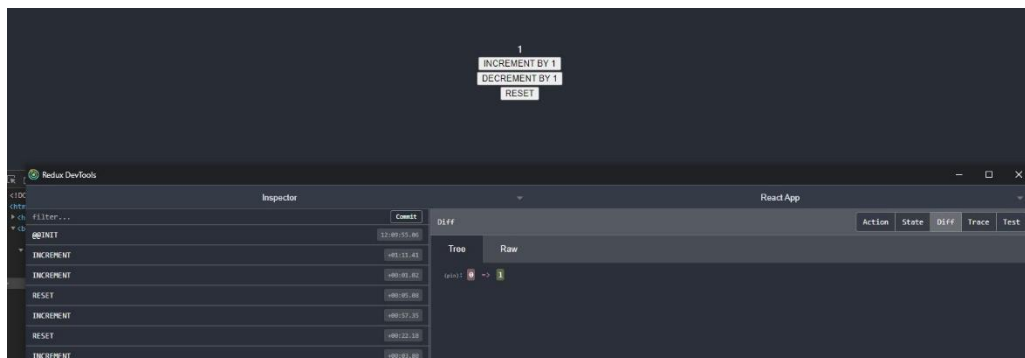
Quando o decrementamos 4 vezes, o resultado exibido na tela é o seguinte:



Ao clicar no botão Reset, o aplicativo volta ao estado inicial, que é o valor do contador 0. Observe abaixo:



Vamos entender o que acontece com as ferramentas de desenvolvimento Redux quando a primeira ação de incremento ocorre



O estado do aplicativo será movido para o momento em que apenas a ação de incremento é despachada e o restante das ações é ignorado.

Incentivamos o desenvolvimento de um pequeno aplicativo Todo como uma atribuição sua e a entender melhor a ferramenta Redux.

React Hooks

Hooks são o novo recurso introduzido na versão React 16.8. Ele permite que você use o estado e outros recursos do React sem escrever uma classe. Hooks são as funções que "engancham" no estado React e nos recursos de ciclo de vida dos componentes de função. Não funciona dentro das classes.

Os hooks são compatíveis com versões anteriores, o que significa que não contêm alterações importantes. Além disso, não substitui seu conhecimento dos conceitos do React.

Quando usar Hooks

Se você escrever um componente de função e quiser adicionar algum estado a ele, faça isso anteriormente convertendo-o em uma classe. Mas, agora você pode fazer isso usando um hook dentro do componente de função existente.

Regras dos Hooks

Os hooks são semelhantes às funções JavaScript, mas você precisa seguir essas duas regras ao usá-los. A regra de hooks garante que toda a lógica com estado em um componente seja visível em seu projeto. Essas regras são:

- **faça call Hooks no top level**

Não chame Hooks dentro de loops, condições ou funções aninhadas. Os hooks devem sempre ser usados no nível superior das funções do React. Esta regra garante que os Hooks sejam chamados na mesma ordem sempre que um componente for renderizado.

- **Faça call Hooks de React functions**

Você não pode chamar Hooks a partir de funções JavaScript regulares. Em vez disso, você pode chamar Hooks dos componentes da função React. Hooks também podem ser chamados de Hooks personalizados.

Pré-requisitos para React Hooks

- Node versão 6 ou superior
- NPM versão 5.2 ou superior
- Ferramenta create-react-app para executar o aplicativo React

Instalação React Hooks

Para usar o React Hooks, precisamos executar os seguintes comandos:

```
$ npm install react
$ npm install react-dom
```

O comando acima instalará as versões alfa do React e React-DOM mais recentes que suportam React Hooks. Certifique-se de que o arquivo **package.json** lista as dependências React e React-DOM conforme fornecido abaixo.

```
"react": "^17.0.1",
```

```
"react-dom": "^17.0.1",
```

Hooks State

O estado do hook é a nova forma de declarar um estado no aplicativo React. O Hook usa o componente funcional `useState()` para definir e recuperar o estado. Vamos entender o estado de Hook com o exemplo a seguir.

App.js

```
import React, { useState } from 'react';

function CountApp() {
  // Declare a new state variable, which we'll call
  "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>

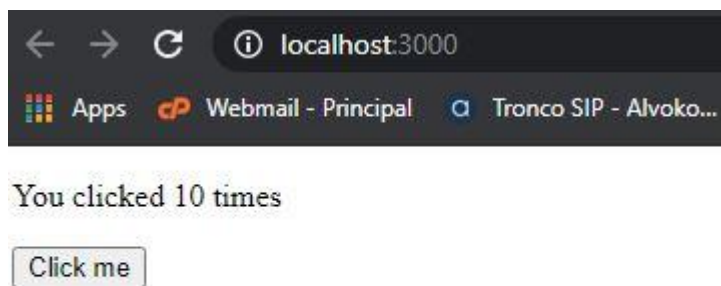
        Click me
      </button>
    </div>
  );
}
export default CountApp;
```

index.js

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

ReactDOM.render(
  <CountApp />,
  document.getElementById('root')
);
```

Saida



No exemplo acima, `useState` é o Hook que precisa chamar dentro de um componente de função para adicionar algum estado local a ele. O `useState` retorna um par onde o primeiro elemento é o valor do estado atual / valor inicial, e o segundo é uma função que nos permite atualizá-lo. Depois disso, chamaremos essa função de um manipulador de eventos ou de outro lugar. O `useState` é semelhante a **`this.setState`** na classe. O código equivalente sem Hooks se parece com o abaixo.

App.js

```
import React, { useState } from 'react';

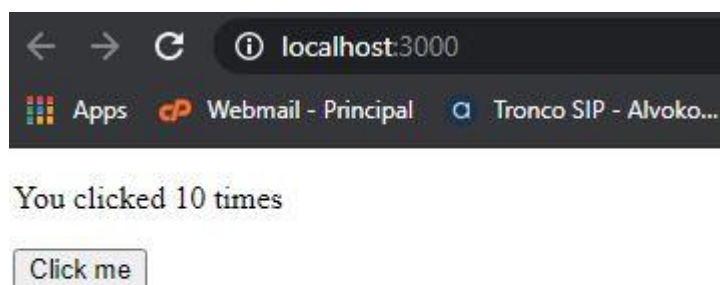
class CountApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p><b>You clicked {this.state.count} times<
/b></p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

```
export default CountApp;
```

index.js

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

```
ReactDOM.render(<CountApp />,
  document.getElementById('root')
);
```



Hooks Effect

O Hook de Efeito nos permite realizar efeitos colaterais (uma ação) nos componentes da função. Ele não usa métodos de ciclo de vida de componentes que estão disponíveis em componentes de classe. Em outras palavras, os Hooks de efeitos são equivalentes aos métodos de ciclo de vida `componentDidMount()`, `componentDidUpdate()` e `componentWillUnmount()`.

Os efeitos colaterais têm recursos comuns que a maioria dos aplicativos da web precisa executar, como:

- Atualizando o DOM,
- Buscar e consumir dados de uma API de servidor,
- Configurando uma assinatura, etc.

Vamos entender o Efeito Hook com o exemplo a seguir.

```
import React, { useState, useEffect } from 'react';
```

```

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

  // Similar to componentDidMount and componentDidU
pdate:
  useEffect(() => {
    // Update the document title using the browser
API
    document.title = `You clicked ${count} times`;

  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>

        Click me
      </button>
    </div>
  );
}
export default CounterExample;

```

index.js

```

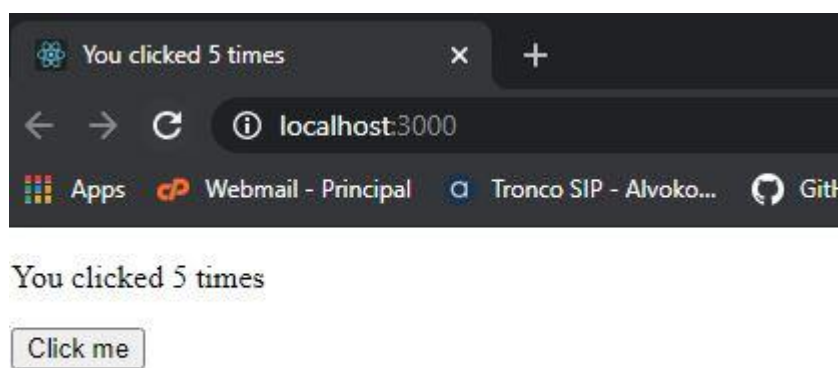
import React from 'react';
import ReactDOM from 'react-dom';
import CounterExample from './App';

ReactDOM.render(<CounterExample />,
  document.getElementById('root')
);

```

O código acima é baseado no exemplo anterior com um novo recurso em que definimos o título do documento para uma mensagem personalizada, incluindo o número de cliques.

Resultado:



No componente React, existem dois tipos de efeitos colaterais:

- Efeitos sem cleanup
- Efeitos com limpeza

Efeitos sem Cleanup

Ele é usado em `useEffect`, que não impede que o navegador atualize a tela. Isso torna o aplicativo mais responsivo. O exemplo mais comum de efeitos que não requerem uma limpeza são mutações DOM manuais, solicitações de rede, registro, etc.

Effects com Cleanup

Alguns efeitos requerem limpeza após a atualização do DOM. Por exemplo, se quisermos configurar uma assinatura para alguma fonte de dados externa, é importante limpar a memória para não introduzir um vazamento de memória. O React realiza a limpeza da memória quando o componente é desmontado. No entanto, como sabemos disso, os efeitos são executados para todos os métodos de renderização e não apenas uma vez. Portanto, o React também limpa os efeitos da renderização anterior antes de executar os efeitos da próxima vez.

Custom Hooks

Um Hook personalizado é uma função JavaScript. O nome do Hook personalizado começa com "use", que pode chamar outros Hooks. Um Hook personalizado é como uma função regular, e a palavra "use" no início diz que

essa função segue as regras dos Hooks. Construir Hooks customizados permite que você extraia a lógica do componente em funções reutilizáveis.

Vamos entender como os Hooks customizados funcionam no exemplo a seguir.

```
import React, { useState, useEffect } from 'react';

const useDocumentTitle = title => {
  useEffect(() => {
    document.title = title;
  }, [title])
}

function CustomCounter() {
  const [count, setCount] = useState(0);
  const incrementCount = () => setCount(count + 1);

  useDocumentTitle(`You clicked ${count} times`);
  // useEffect(() => {
  //   document.title = `You clicked ${count} times
  // });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={incrementCount}>Click me</button>
    </div>
  )
}

export default CustomCounter;
```

No fragmento acima, `useDocumentTitle` é um Hook personalizado que recebe um argumento como uma string de texto que é um título. Dentro desse Hook, chamamos `useEffect` Hook e definimos o título, desde que ele seja alterado. O segundo argumento executará essa verificação e atualizará o título apenas quando seu estado local for diferente do que estamos transmitindo.

Obs: Um Hook personalizado é uma convenção que segue naturalmente o design dos Hooks, em vez de um recurso React.

Built-in Hooks

Aqui, descrevemos as APIs para os Hooks integrados no React. Os Hooks embutidos podem ser divididos em duas partes, que são fornecidas abaixo.

Hooks Básicos

- useState
- useEffect
- useContext

Hooks Adicionais

- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useEffect
- useDebugValue

React Map

Map é um tipo de coleta de dados em que os dados são armazenados na forma de pares. Ele contém uma chave exclusiva. O valor armazenado no mapa deve ser mapeado para a chave. Não podemos armazenar um par duplicado no map(). É por causa da exclusividade de cada chave armazenada. Ele é usado principalmente para pesquisar e pesquisar dados rapidamente.

Em React, map() é o método usado para percorrer e exibir uma lista de objetos semelhantes de um componente. Um mapa não é uma característica do React. Em vez disso, é a função JavaScript padrão que pode ser chamada em qualquer array. O método map () cria uma nova matriz chamando uma função fornecida em cada elemento da matriz de chamada.

Exemplo

No exemplo dado, a função `map()` pega uma matriz de números e duplica seus valores. Atribuímos a nova matriz retornada por `map()` à variável `doubleValue` e a registramos.

```
var numbers = [1, 2, 3, 4, 5];
const doubleValue = numbers.map((number) => {
  return (number * 2);
});
console.log(doubleValue);
```

Percorrendo o elemento da lista.

Exemplo

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';

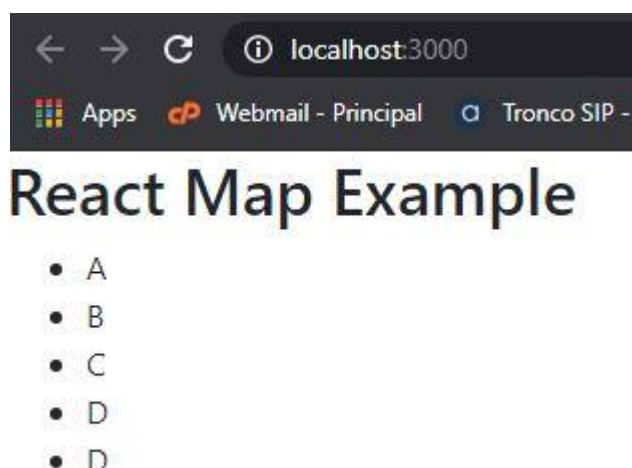
function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((myList) =>
    <li key={myList.toString()}>
      {myList}
    </li>
  );
  return (
    <div>
      <h2>React Map Example</h2>
      <ul>{listItems}</ul>
    </div>
  );
}

const myLists = ['A', 'B', 'C', 'D', 'E'];
ReactDOM.render(
  <NameList myLists={myLists} />,
  document.getElementById('root')
);

// export default App;
```

Obs.: No código acima, estamos renderizando tudo dentro do arquivo `App.js`; portanto, a linha `export default App`; pode ser comentada

Resultado



Percorrer o elemento da lista com chaves.

Exemplo

```
import React from 'react';
import ReactDOM from 'react-dom';

function ListItem(props) {
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
      value={number} />
  );
  return (
    <div>
      <h2>React Map Example</h2>
      <ul> {listItems} </ul>
    </div>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,

```

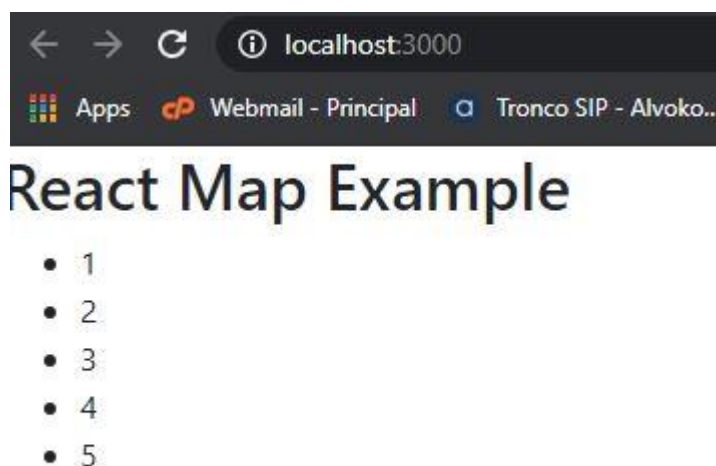
```

    document.getElementById('root')
  );
  // export default App;

```

Obs.: No código acima, estamos renderizando tudo dentro do arquivo App.js; portanto, a linha export default App; pode ser comentada

Saida



Componentes de high order

Componentes de ordem superior são funções JavaScript usadas para adicionar funcionalidades adicionais ao componente existente. Essas funções são **puras**, o que significa que estão recebendo dados e retornando valores de acordo com esses dados. Se os dados mudarem, as funções de ordem superior serão executadas novamente com entrada de dados diferente. Se quisermos atualizar nosso componente de retorno, não precisamos alterar o HOC. Tudo o que precisamos fazer é alterar os dados que nossa função está usando.

O Componente de Ordem Superior (HOC) envolve o componente "normal" e fornece entrada de dados adicional. Na verdade, é uma função que pega um componente e retorna outro componente que envolve o original.

Vamos dar uma olhada em um exemplo simples para entender facilmente como esse conceito funciona. O **MyHOC** é uma função de ordem superior usada apenas para passar dados para **MyComponent**. Esta função pega **MyComponent**, aprimora-o com **newData** e retorna o componente aprimorado que será renderizado na tela.

```
import React from 'react';

var newData = {
  data: 'Data from HOC...',
}

var MyHOC = ComposedComponent => class extends
React.Component {
  componentDidMount() {
    this.setState({
      data: newData.data
    });
  }
  render() {
    return <ComposedComponent {...this.props}
{...this.state} />;
  }
};

class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.data}</h1>
      </div>
    )
  }
}
```

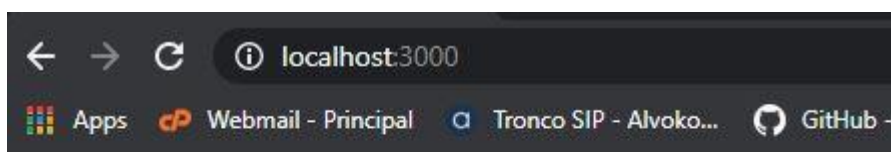
```
export default MyHOC(MyComponent);
```

index.js

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

ReactDOM.render(<MyHOC/>,
document.getElementById('root'));
```

Se executarmos o aplicativo, veremos que os dados são passados para **MyComponent** .



Data from HOC...

Obs.: - Componentes de ordem superior podem ser usados para diferentes funcionalidades. Essas funções puras são a essência da programação funcional. Depois de se acostumar com isso, você notará como seu aplicativo está se tornando mais fácil de manter ou atualizar.

Animações

Neste passo, aprenderemos como animar elementos usando React.

A animação é uma técnica na qual as imagens são manipuladas para parecerem imagens em movimento. É uma das técnicas mais utilizadas para fazer uma aplicação web interativa. No React, podemos adicionar animação usando um grupo explícito de componentes conhecido como **React Transition Group**.

React Transition Group é um componente adicional para gerenciar estados de componentes e útil para definir transições de **entrada** e **saída**. Não é capaz de animar estilos por si só. Em vez disso, ele expõe estados de transição, gerencia classes e elementos de grupo e manipula o DOM de maneiras úteis. Isso torna a implementação de transições visuais muito mais fácil.

O grupo React Transition tem principalmente **duas APIs** para criar transições. Esses são:

- **ReactTransitionGroup**: é usado como uma API de baixo nível para animação.
- **ReactCSSTransitionGroup**: é usado como uma API de alto nível para implementar transições e animações CSS básicas.

Instalação

Precisamos instalar o **react-transition-group** para criar a animação no aplicativo da Web React. Você pode usar o comando abaixo.

```
npm install react-transition-group --save
npm install @types/react-transition-group
```

Componentes do Grupo de Transição React

A API React Transition Group fornece **três** componentes principais. Esses são:

- Transição
- CSSTransition
- Grupo de Transição

Transição

Tem uma API de componente simples para descrever uma transição de um estado de componente para outro ao longo do tempo. É usado principalmente para animar a **montagem** e **desmontagem** de um componente. Ele também pode ser usado para estados de transição in-loco.

Podemos acessar o componente de transição em quatro estados:

- entering
- entered
- exiting
- exited

CSSTransition

O componente CSSTransition usa classes de folha de estilo CSS para escrever a transição e criar animações. É inspirado na biblioteca **ng-animate**. Ele também pode herdar todos os adereços do componente de transição. Podemos dividir a "CSSTransition" em **três** estados. Esses são:

- Appear
- Enter
- Exit

O componente CSSTransition deve ser aplicado em um par de nomes de classes aos componentes filhos. A primeira classe está na forma de **estágio** de **nome** e a segunda classe está no **estágio de nome ativo**. Por exemplo, você fornece o nome `fade` e, quando se aplica ao estágio de 'entrada', as duas classes serão **fade-enter** e **fade-enter-active**. Também pode ser necessário um adereço como `Timeout`, que define o tempo máximo para animar

TransitionGroup

Este componente é usado para gerenciar um conjunto de componentes de transição (Transição e CSSTransition) em uma lista. É uma máquina de estado que controla a **montagem** e **desmontagem** de componentes ao longo do

tempo. O componente `Transition` não define nenhuma animação diretamente. Aqui, a forma como o item de 'lista' é animado é baseado no componente de transição individual. Isso significa que o componente `"TransitionGroup"` pode ter animações diferentes dentro de um componente.

Vejamos o exemplo abaixo, que ajuda claramente a entender o `React Animation`.

Exemplo

App.js

No arquivo `App.js`, importe o componente `react-transition-group` e crie o componente `CSSTransition` que usa como um wrapper do componente que você deseja animar. Usaremos **`transiçãoEnterTimeout`** e **`transiçãoLeaveTimeout`** para a **`transição`** CSS. As animações de entrada e saída usadas quando queremos inserir ou excluir elementos da lista.

App.js

```
import React, { Component } from 'react';
import { CSSTransitionGroup } from 'react-transition-group';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['Blockchain', 'ReactJS', 'TypeScript', 'JavaTpoint']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter Item Name')
    ]);
    this.setState({items: newItems});
  }
}
```

```

    }

    handleRemove(i) {
      let newItems = this.state.items.slice();
      newItems.splice(i, 1);
      this.setState({items: newItems});
    }

    render() {
      const items = this.state.items.map((item, i) => (
        <div key={item} onClick={() => this.handleRemove(i
      ))>
        {item}
      </div>
    ));

      return (
        <div>
          <h1>Animation Example</h1>
          <button onClick={ this.handleAdd}>Insert Item
        </button>

          <CSSTransitionGroup
            transitionName="example"
            transitionEnterTimeout={800}
            transitionLeaveTimeout={600}>
            {items}
          </CSSTransitionGroup>
        </div>
      );
    }
  }
}
export default App;

```

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root')
);

```

style.css

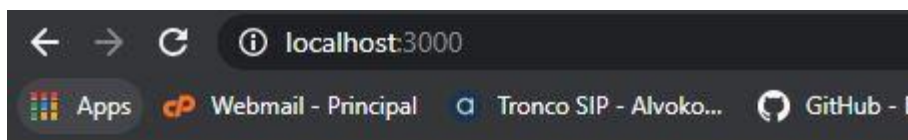
Adicione o arquivo `style.css` em seu aplicativo e adicione os seguintes estilos CSS. Agora, para usar este arquivo CSS, você precisa adicionar o **link** deste arquivo em seu arquivo HTML.

```
.example-enter {  
  opacity: 0.01;  
}  
  
.example-enter.example-enter-active {  
  opacity: 1;  
  transition: opacity 500ms ease-in;  
}  
  
.example-leave {  
  opacity: 1;  
}  
  
.example-leave.example-leave-active {  
  opacity: 0.01;  
  transition: opacity 300ms ease-in;  
}
```

No exemplo acima, as durações da animação são especificadas no **CSS** e no método de **renderização**. Diz ao componente React quando remover as classes de animação da lista e se está saindo quando remover o elemento do DOM.

Resultado

Quando executamos o programa acima, ele fornece a saída abaixo.



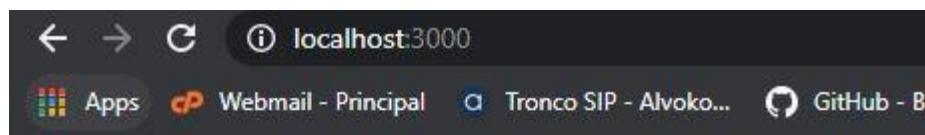
Animation Example



Clique no botão '**Insert Item**', a seguinte tela aparecerá:



Depois de inserir o item e pressionar **Ok**, o novo item pode ser adicionado à lista com o estilo fade. Aqui, também podemos excluir qualquer item da lista clicando no link específico.



Animation Example

Insert Item

Blockchain

ReactJS

TypeScript

Novo Topico

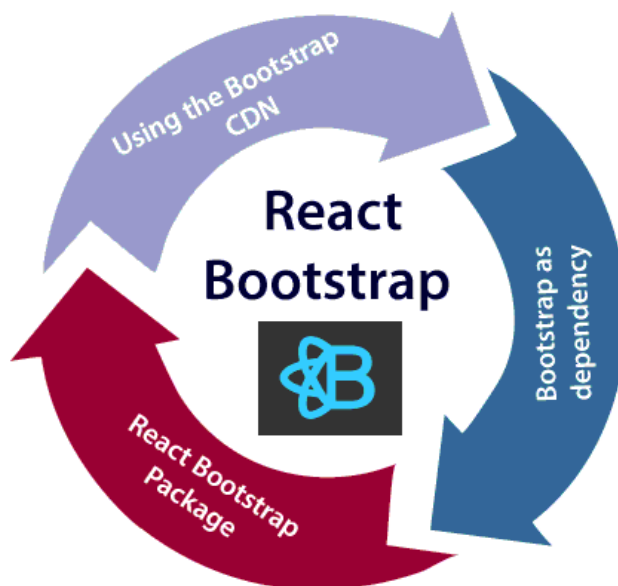
React Bootstrap

Aplicativos de página única ganhando popularidade nos últimos anos, muitos frameworks front-end foram introduzidos, como Angular, React, Vue.js, Ember, etc. Como resultado, jQuery não é um requisito necessário para construir aplicativos web. Hoje, React tem a estrutura JavaScript mais usada para construir aplicativos da web, e o Bootstrap se tornou a estrutura CSS mais popular. Portanto, é necessário aprender várias maneiras de usar o Bootstrap nos aplicativos React, que é o objetivo principal desta seção.

Adicionando Bootstrap no React

Podemos adicionar Bootstrap ao aplicativo React de várias maneiras. As **três** formas mais comuns são fornecidas abaixo:

- Usando o CDN Bootstrap
- Bootstrap como dependência
- Pacote React Bootstrap



Usando o CDN Bootstrap

É a maneira mais fácil de adicionar Bootstrap ao aplicativo React. Não há necessidade de instalar ou baixar o Bootstrap. Podemos simplesmente colocar um `<link>` na seção `<head>` do arquivo `index.html` do aplicativo React, conforme mostrado no fragmento a seguir.

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
```

Se houver necessidade de usar componentes Bootstrap que dependem de JavaScript / jQuery no aplicativo React, precisamos incluir **jQuery**, **Popper.js** e **Bootstrap.js** no documento. Adicione as seguintes importações nas marcas `<script>` próximas ao final da marca `</body>` de fechamento do arquivo `index.html`.

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-
```



```
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86d
IHNDz0W1" crossorigin="anonymous"></script>
```

```
<script src="https://stackpath.bootstrapcdn.com/boot
strap/4.3.1/js/bootstrap.min.js" integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xI
M+B07jRM" crossorigin="anonymous"></script>
```

No snippet acima, usamos a versão slim do jQuery, embora também possamos usar a versão completa. Agora, o Bootstrap foi adicionado com sucesso ao aplicativo React, e podemos usar todos os utilitários CSS e componentes de IU disponíveis no Bootstrap no aplicativo React

Bootstrap como Dependency

Se estivermos usando uma ferramenta de construção ou um bundler de módulo, como Webpack, importar o Bootstrap como dependência é a opção preferida para adicionar o Bootstrap ao aplicativo React. Podemos instalar o Bootstrap como uma dependência do aplicativo React. Para instalar o Bootstrap, execute os seguintes comandos na janela do terminal.

```
$ npm install bootstrap --save
```

Assim que o Bootstrap estiver instalado, podemos importá-lo no arquivo de entrada do aplicativo React. Se o projeto React foi criado com a ferramenta **create-react-app**, abra o arquivo **src / index.js** e adicione o seguinte código:

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Agora, podemos usar as classes e utilitários CSS no aplicativo React. Além disso, se quisermos usar os componentes JavaScript, precisamos instalar os pacotes **jquery** e **popper.js** do **npm**. Para instalar os pacotes a seguir, execute o seguinte comando na janela do terminal.

```
$ npm install jquery popper.js
```

Em seguida, vá para o arquivo **src / index.js** e adicione as seguintes importações.

```
import $ from 'jquery';
import Popper from 'popper.js';
import 'bootstrap/dist/js/bootstrap.bundle.min';
```

Agora, podemos usar componentes JavaScript Bootstrap no aplicativo React.

React Bootstrap Package

O pacote React Bootstrap é a forma mais popular de adicionar bootstrap no aplicativo React. Existem muitos pacotes Bootstrap construídos pela comunidade, que visam reconstruir componentes Bootstrap como componentes React. Os **dois** pacotes de Bootstrap mais populares são:

1. **react-bootstrap:** É uma reimplementação completa dos componentes do Bootstrap como componentes do React. Não precisa de dependências como bootstrap.js ou jQuery. Se a configuração do React e o React-Bootstrap estiverem instalados, temos tudo o que precisamos.
2. **reactstrap:** É uma biblioteca que contém componentes React Bootstrap 4 que favorecem a composição e o controle. Não depende de jQuery ou Bootstrap JavaScript. No entanto, o react-popover é necessário para o posicionamento avançado de conteúdo, como dicas de ferramentas, popovers e menus suspensos de inversão automática.

Instalação React Bootstrap

Vamos criar um novo aplicativo React usando o comando **create-react-app** da seguinte maneira.

```
$ npx create-react-app react-bootstrap-app
```

Depois de criar o aplicativo React, a melhor maneira de instalar o Bootstrap é por meio do pacote npm. Para instalar o Bootstrap, navegue até a pasta do aplicativo React e execute o seguinte comando.

```
$ npm install react-bootstrap bootstrap --save
```

Importando Bootstrap

Agora, abra o arquivo **src / index.js** e adicione o seguinte código para importar o arquivo Bootstrap.

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Também podemos importar componentes individuais, **como importar {SplitButton, Dropdown} do 'react-bootstrap'**; em vez de toda a biblioteca. Ele fornece os componentes específicos que precisamos usar e pode reduzir significativamente a quantidade de código.

No aplicativo React, crie um novo arquivo chamado **ThemeSwitcher.js** no diretório **src** e coloque o código a seguir.

```
import React, { Component } from 'react';
import { SplitButton, Dropdown } from 'react-
bootstrap';

class ThemeSwitcher extends Component {

  state = { theme: null }

  chooseTheme = (theme, evt) => {
    evt.preventDefault();
    if (theme.toLowerCase() === 'reset') { theme =
null }
    this.setState({ theme });
  }

  render() {
    const { theme } = this.state;
    const themeClass = theme ? theme.toLowerCase()
: 'default';

    const parentContainerStyles = {
      position: 'absolute',
      height: '100%',
      width: '100%',
      display: 'table'
    };

    const subContainerStyles = {
      position: 'relative',
```

```

        height: '100%',
        width: '100%',
        display: 'table-cell',
    };

    return (
        <div style={parentContainerStyles}>
            <div style={subContainerStyles}>

                <span className={`h1 center-block text-
center text-
${theme ? themeClass : 'muted'}} style={{ marginBottom:
25 }}>{theme || 'Default'}</span>

                <div className="center-block text-
center">

                    <SplitButton bsSize="large" bsStyle={th
emeClass} title={` ${theme || 'Default Block'} Theme`}>
                        <Dropdown.Item eventKey="Primary Bloc
k" onSelect={this.chooseTheme}>Primary Theme</Dropdown.I
tem>

                        <Dropdown.Item eventKey="Danger Block
" onSelect={this.chooseTheme}>Danger Theme</Dropdown.Ite
m>

                        <Dropdown.Item eventKey="Success Bloc
k" onSelect={this.chooseTheme}>Success Theme</Dropdown.I
tem>

                        <Dropdown.Item divider />
                        <Dropdown.Item eventKey="Reset Block"
onSelect={this.chooseTheme}>Default Theme</Dropdown.Ite
m>

                    </SplitButton>
                </div>
            </div>
        </div>
    );
}
}
export default ThemeSwitcher;

```

Agora, atualize o arquivo **src / index.js** com o seguinte trecho.

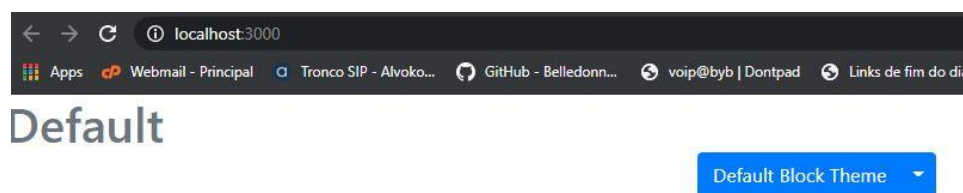
Index.js

```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

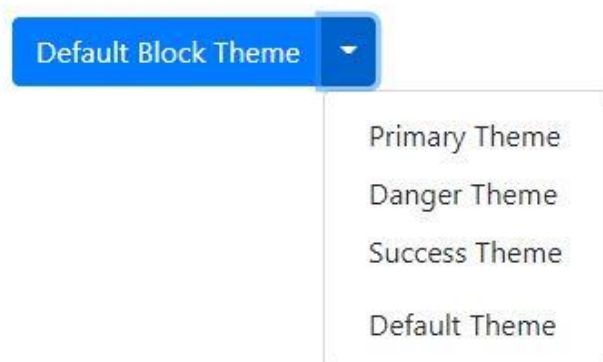
ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

Saida

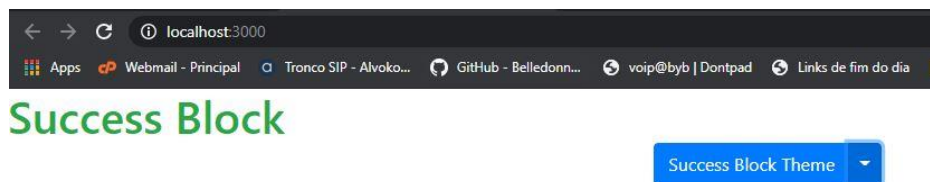
Quando executamos o aplicativo React, devemos obter a saída conforme abaixo.



Clique no menu suspenso. Iremos obter a seguinte tela.



Agora, se escolhermos o **Tema de Sucesso**, teremos a tela abaixo.



Usando o reactstrap

Vamos criar um novo aplicativo React usando o comando create-react-app da seguinte maneira.

```
$ npx create-react-app reactstrap-app
```

Em seguida, instale o **reactstrap** por meio do pacote npm. Para instalar o reactstrap, navegue até a pasta do aplicativo React e execute o seguinte comando.

```
$ npm install bootstrap reactstrap --save
```

Importando Bootstrap

Agora, abra o arquivo **src / index.js** e adicione o seguinte código para importar o arquivo Bootstrap.

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Também podemos importar componentes individuais, **como importar {Button, Dropdown} de 'reactstrap'**; em vez de toda a biblioteca. Ele fornece os componentes específicos que precisamos usar e pode reduzir significativamente a quantidade de código.

No aplicativo React, crie um novo arquivo chamado **ThemeSwitcher.js** no diretório **src** e coloque o código a seguir.

```
import React, { Component } from 'react';
import { Button, ButtonDropdown, DropdownToggle, DropdownMenu, DropdownItem } from 'reactstrap';

class ThemeSwitcher extends Component {
  state = { theme: null, dropdownOpen: false }
```

```

toggleDropdown = () => {
  this.setState({ dropdownOpen: !this.state.dropdownOpen });
}

resetTheme = evt => {
  evt.preventDefault();
  this.setState({ theme: null });
}

chooseTheme = (theme, evt) => {
  evt.preventDefault();
  this.setState({ theme });
}

render() {
  const { theme, dropdownOpen } = this.state;
  const themeClass = theme ? theme.toLowerCase()
: 'secondary';

  return (
    <div className="d-flex flex-wrap justify-content-center align-items-center">

      <span className={`h1 mb-4 w-100 text-center text-${themeClass}`}>{theme || 'Default'}</span>

      <ButtonDropdown isOpen={dropdownOpen} toggle={this.toggleDropdown}>
        <Button id="caret" color={themeClass}>{theme || 'Custom'} Theme</Button>
        <DropdownToggle caret size="lg" color={themeClass} />
        <DropdownMenu>
          <DropdownItem onClick={e => this.chooseTheme('Primary', e)}>Primary Theme</DropdownItem>
          <DropdownItem onClick={e => this.chooseTheme('Danger', e)}>Danger Theme</DropdownItem>
          <DropdownItem onClick={e => this.chooseTheme('Success', e)}>Success Theme</DropdownItem>
          <DropdownItem divider />
          <DropdownItem onClick={this.resetTheme}>Default Theme</DropdownItem>
        </DropdownMenu>
      </ButtonDropdown>
    </div>
  );
}

```

```

        </DropdownMenu>
      </ButtonDropdown>

    </div>
  );
}
}
export default ThemeSwitcher;

```

Agora, atualize o arquivo **src / index.js** com o seguinte trecho.

Index.js

```

import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

ReactDOM.render(<ThemeSwitcher />, document.getElementBy
entById('root'));

```

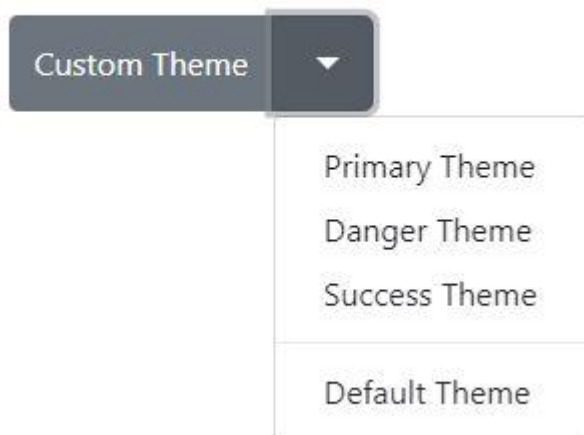
Resultado

Quando executamos o aplicativo React, devemos obter a saída conforme abaixo.



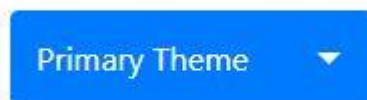
Clique no menu suspenso. Iremos obter a seguinte tela.

Default



Agora, se escolhermos o tema de **Primary**, obteremos a tela abaixo.

Primary



React Table

Uma tabela é um arranjo que organiza as informações em linhas e colunas. Ele é usado para armazenar e exibir dados em um formato estruturado.

A tabela react é uma Datagrid leve, rápida, totalmente personalizável (JSX, templates, state, styles, callbacks) e extensível construída para React. É totalmente controlável por meio de props e retornos de chamada opcionais.

Características

- É leve em 11kb (e só precisa de mais 2kb para estilos).
- É totalmente personalizável (JSX, modelos, estado, estilos, retornos de chamada).

- É totalmente controlável por meio de props e retornos de chamada opcionais.
- Tem paginação do lado do cliente e do lado do servidor.
- Possui filtros.
- Dinâmica e agregação
- Design minimalista e facilmente temático

Instalação

Vamos criar um **aplicativo React** usando o seguinte comando.

```
$ npx create-react-app myreactapp
```

Em seguida, precisamos instalar a **react table** . Podemos instalar a react table por meio do comando npm, que é fornecido abaixo.

Obs.: para este projeto vamos usar a versão 6 de react Table

```
$ npm install react-table-6
```

Assim que tivermos instalado a tabela reativa, precisamos **importar** a tabela reativa para o componente reativo. Para fazer isso, abra o arquivo **src / App.js** e adicione o seguinte snippet.

```
import ReactTable from "react-table-6";
```

Vamos supor que temos dados que precisam ser renderizados usando a React Table.

```
const data = [{  
  name: 'Ayaan',  
  age: 26  
}, {  
  name: 'Ahana',  
  age: 22  
}, {  
  name: 'Peter',  
  age: 40  
}, {  
  name: 'Virat',  
  age: 30  
}, {  
  name: 'Rohit',
```

```

    age: 32
  }, {
    name: 'Dhoni',
    age: 37
  }]

```

Junto com os dados, também precisamos especificar as **informações da coluna** com os **atributos da coluna**.

```

const columns = [{
  Header: 'Name',
  accessor: 'name'
}, {
  Header: 'Age',
  accessor: 'age'
}]

```

Dentro do método render, precisamos vincular esses dados com a tabela reativa e, em seguida, retornar a react table.

```

return (
  <div>
    <ReactTable
      data={data}
      columns={columns}
      defaultPageSize = {2}
      pageSizeOptions = {[2, 4, 6]}
    />
  </div>
)

```

Agora, nosso arquivo **src / App.js** se parece com o abaixo.

```

import React, { Component } from 'react';
import ReactTable from "react-table-6";
import "react-table-6/react-table.css";

class App extends Component {
  render() {
    const data = [{
      name: 'Ayaan',
      age: 26
    }, {

```

```

        name: 'Ahana',
        age: 22
      }, {
        name: 'Peter',
        age: 40
      }, {
        name: 'Virat',
        age: 30
      }, {
        name: 'Rohit',
        age: 32
      }, {
        name: 'Dhoni',
        age: 37
      }
    ]
    const columns = [{
      Header: 'Name',
      accessor: 'name'
    }, {
      Header: 'Age',
      accessor: 'age'
    }]
    return (
      <div>
        <ReactTable
          data={data}
          columns={columns}
          defaultPageSize = {2}
          pageSizeOptions = {[2, 4, 6]}
        />
      </div>
    )
  }
}
export default App;

```

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

```

```
ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

Resultado

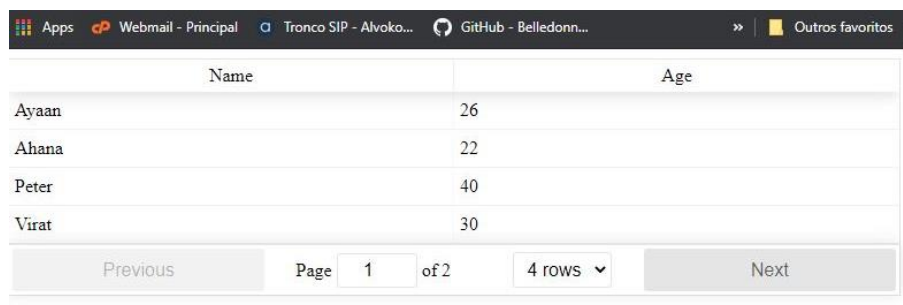
Quando executarmos o aplicativo React, obteremos a saída conforme abaixo.



Name	Age
Ayaan	26
Ahana	22

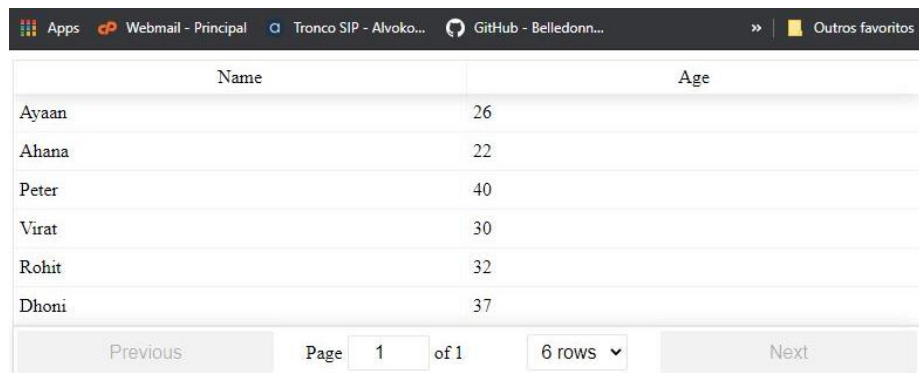
Previous Page 1 of 3 2 rows Next

Agora, altere o menu suspenso de linhas, obteremos a saída conforme abaixo.



Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30

Previous Page 1 of 2 4 rows Next



Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30
Rohit	32
Dhoni	37

Previous Page 1 of 1 6 rows Next

Referencias:

https://developer.mozilla.org/pt-BR/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Comecando_com_React

<https://www.tutorialspoint.com/reactjs/>

<https://create-react-app.dev/docs/measuring-performance/>

<https://www.javatpoint.com/reactjs-tutorial>