**React**

**Introdução**

Os sistemas web nos anos 1990 eram páginas HTML estáticas, meramente informativas, cuja interação dos usuários era realizada através de links. Em 2000 é implantado o uso de programação dentro dos servidores web para a customização de páginas HTML através de dados, possibilitando assim sistemas interativos cuja base é web, *e-commerces* por exemplo.

A mudança cultural demorou a acontecer, trazendo à tona o estouro da bolha. Os sistemas web só retornaram à evidência em 2008 com o surgimento das redes sociais e com a necessidade de maiores interações, responsividade e que não necessitassem de irem e voltarem ao servidor (round-trip), o que causava o recarregamento da página. Nessa época se torna comum a utilização de Ajax, que permite consultas ao servidor em background e bibliotecas de manipulação do HTML.

Porém as redes sociais necessitavam de tecnologias web mais práticas dentro do seu contexto. Dessa necessidade nascem bibliotecas e frameworks voltados ao front-end. Com isso a lógica de servidor é trocada para uma lógica do lado cliente, tornando o uso do servidor apenas para acesso de dados.

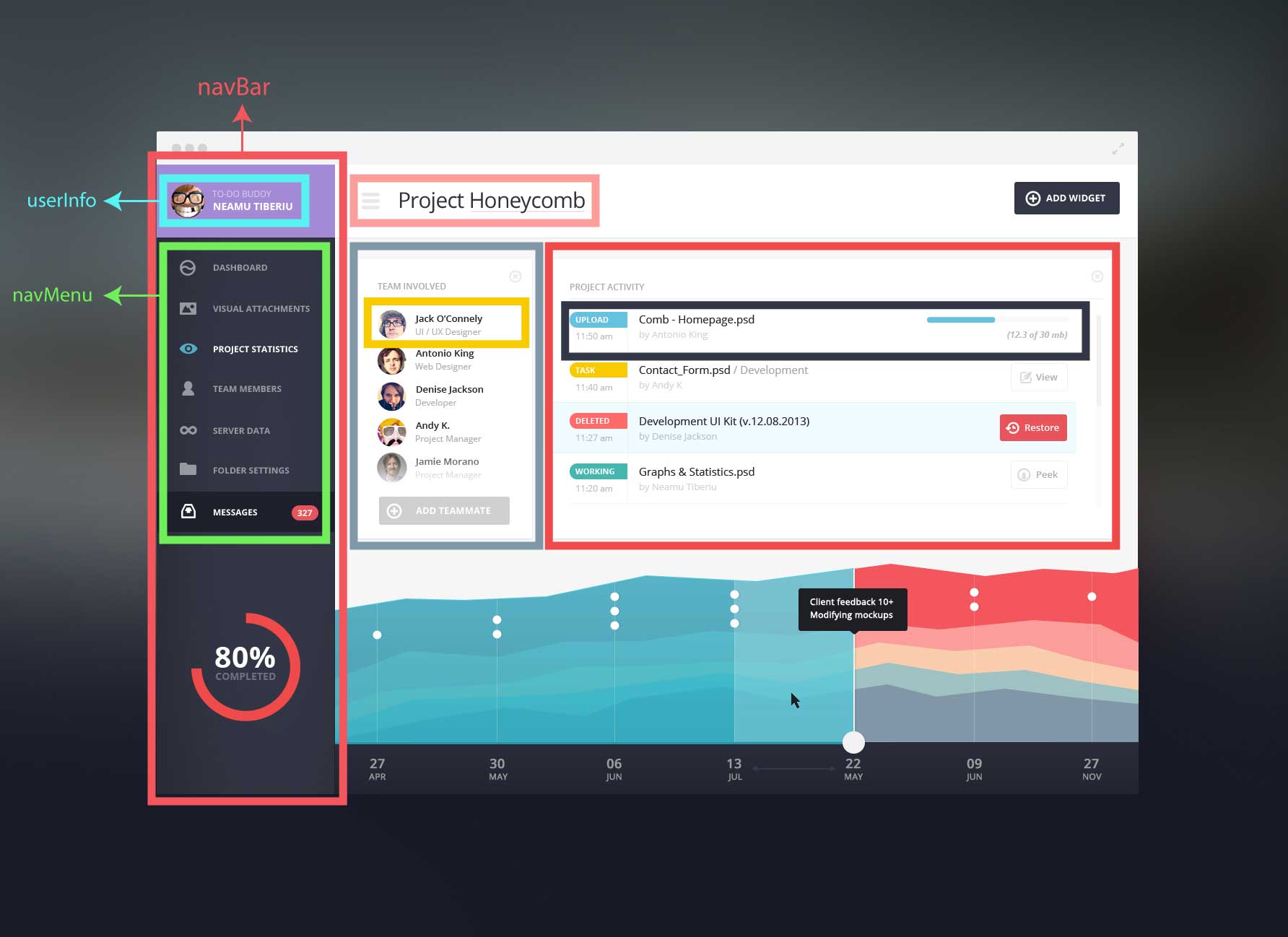
Nesse contexto surge o SPA – Single Page Applications, cuja função é utilizar uma única página HTML que sofre alterações conforme a interação dos usuários ou eventos do sistema interno. Dessa demanda surge o React, que hoje se tornou a principal biblioteca de front-end para a criação de SPA.

React se torna o pioneiro do conceito da programação reativa em front-end. O conceito é basicamente a reconstrução da página conforme houverem mudanças nos dados. Ele tem a capacidade de renderizar apenas partes da página que são diretamente dependentes dos dados que foram alterados e não o seu todo, desta forma possui ótimo desempenho.

Sua arquitetura é de componentes, portanto em sua maioria a programação em React se baseia em saber dividir bem uma página em componentes, fazendo isso de forma reutilizável admitindo composição com outros componentes.

**Introdução a Componentes**

Tudo que será renderizado pelo React deve ser um componente. Eles podem ser definidos como as “partes” de uma página e podem ser compostos por outros componentes também.

 Definimos como subcomponentes os que possuam ações, renderizem conteúdos a partir dos dados, que possuam listas ou qualquer tipo de conteúdo que não seja estático.

Todo o resto pode ser feito em HTML no interior do componente, ficando a nosso critério promover o conteúdo a componente ou não. Essa decisão é tomada levando em consideração alguns critérios:

1. São partes, que embora estáticas, podem ser reaproveitadas em outro contexto?
2. São complexas visualmente e carregadas em CSS?

Se a resposta for sim para ambas então elas são fortes candidatas para se tornarem componentes.

Componentes, em termos computacionais, podem ser criados por funções (Componentes Funcionais) ou por classes (Componentes Tipo Classe).

**NPM**

É um projeto Open Source criado em 2009 cujo objetivo é facilitar a troca de código JS, é o gerenciador de pacotes padrão do Node.js. Quando o assunto é NPM podemos estar falando de:

1. Do repositório aberto onde os pacotes estão armazenados;
2. De um cliente que permite o download de código do seu repositório;
3. De um site que permite a busca de informações de pacotes e a visualização da documentação do NPM

Ele utiliza um arquivo de configuração chamado package.json, que é o responsável pela configuração do projeto contendo nome, versão, atalhos de comando que o npm executa, etc. Uma de suas funções mais importantes é o armazenamento de uma lista de dependências que o projeto irá utilizar. Através desse arquivo o cliente consegue instalar todas as dependências com apenas um único comando.

Mas e quando o projeto possuí tantas dependências que torna a sua criação complexa. É ai que o Yarn se torna atrativo em comparação ao NPM.

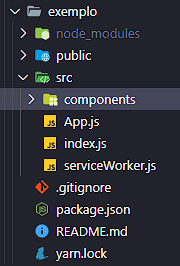
**YARN**

Como o NPM, ele é um gerenciador de dependências/pacotes, que permite o reaproveitamento de código, desta forma acelerando o processo de desenvolvimento em oposição ao armazenamento de códigos JS em outros projetos como antigamente.

“Afinal, se quando você vai fazer um bolo não precisa plantar o trigo e fazer a farinha, por que não aplicar essa mesma lógica ao desenvolvimento, não é mesmo?” – Cássio Bock.

Em 2016 o Facebook, Google, Exponent e Tilde lançaram o Yarn com o objetivo de tornar a instalação das dependências mais rápida e segura. Problemas em projetos do Facebook que dependiam do NPM apresentavam problemas como a demora no tempo de instalação, dependências que estavam em versões diferentes em diversas máquinas e na forma automáticas que o NPM executava códigos das dependências.

**Criação de Projeto React**



yarn create react-app meu-app

Utilizamos o comando acima no terminal Powershell ou no terminal inserido no Visual Studio. Após a instalação realizada pelo YARN utilizamos o seguinte código.

cd meu-app

yarn start

Após a instalação possuímos a estrutura ao lado. Todo o código é feito na pasta src. O arquivo index.js é o responsável pela injeção do React no HTML da página. Dentro dele temos a seguinte estrutura.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import \* as serviceWorker from './serviceWorker';

ReactDOM.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>,

  document.getElementById('root')

);

serviceWorker.unregister();

O método ReactDom.Render() é o responsável por renderizar um componente na página dentro de um elemento HTML. A linha document.getElementById('root') define que o elemento HTML onde o React será injetado é um que tenha um ID = 'root'.

Para que um componente seja renderizado ele deve estar dentro do método ReactDOM.render() ou dentro de outro componente que seja renderizado por este método. O código em HTML fica dentro da pasta public.

**Componentes Simples**

import React from 'react';

class App extends *React*.Component{

}

export default App;

Observe que o componente App do tipo classe utiliza herança da classe Component do React, dessa forma todo o comportamento de componente já está presente na sua classe desde o primeiro momento. O que quisermos mostrar na tela usando o componente criado será feito pelo método render(), que deve retornar uma HTML (é mais do que isso, porém será entendido mais a frente).

import React from 'react';

class App extends *React*.Component{

    render(){

       return (

<>

  <p>Meu primeiro parágrafo em React.</p>

</>

);

    }

}

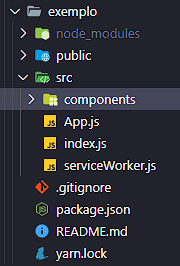
export default App;

O que foi inserido ao return é um dialeto chamado JSX que mistura elementos HTML e Javascript. Ele é utilizado para escrever HTML e Javascript fora de <strings>. O código JS é inserido entre {} para que seja renderizado. Por exemplo um código que retorna a data de hoje (no formato brasileiro).

<p>{ **new** *Date*().toLocaleDateString("pt-BR") }</p>

Esse código deve estar inserido dentro do método render() e no return desse método.

**Props**

Para uma melhor organização criamos uma pasta para armazenar todos os componentes da aplicação, App.js será movido para ela. Corrigimos o arquivo index.js, portanto a linha abaixo será alterada:

import App from './App';

Para:

import App from './components/App';

Tomamos como exemplo a criação de um componente que será reutilizável. Em programação sempre que queremos criar algo reutilizável tornamos o código mais genérico possível e parametrizamos suas informações. Utilizamos isso em funções, por exemplo em uma função que soma 1 e 2 é muito menos útil e reutilizável que uma que soma qualquer número.

Imagine um componente que produz uma caixinha com bordas na tela e tem um título e um texto. Se parametrizarmos o título e o texto poderemos usar essa caixinha em diversas partes de nossa aplicação, até mesmo em outras aplicações! No React, esses "parâmetros" de um componente são chamados de "props". Podemos receber props pelo construtor do componente.

import React from 'react';

class App2 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

  }

  render(){

    return ('OK');

  }

}

export default App2;

Vamos agora alterar o *render* para usar duas props, uma chamada title e uma chamada text esses nomes são arbitrários, pode ser o que você desejar.

render(){

  return(

  <div *className*="box">

      <div *className*="title">{*this*.props.title}</div>

      <div *className*="text">{*this*.props.text}</div>

  </div>);

}

Quando for utilizar o componente App2 você pode passar os valores dos props por nome da mesma forma que passaria atributos html. O código abaixo é inserido no indes.js.

<*App2* *title*='meu título' *text*='meu texto'/>

Caso o texto for muito grande e não puder ser passado por props, basta alterá-lo no render() para:

<div *className*="text">{*this*.props.children}</div>

E no arquivo index.js:

<*App2* *title*="Isso é um teste">

    Lorem ipsum dolor sit amet consectetur adipisicing elit. Nihil officia, quam sed officiis libero repellat voluptate dolores amet molestiae nostrum aperiam inventore veritatis aut quaerat, tenetur laudantium natus? Saepe, minus!

</*App2*>

Qualquer coisa que você colocar entre a abertura e o fechamento das tags do componente serão passados como props.children para o componente! Isso inclui HTML, outros componentes, e até javascript, contanto que esteja entre chaves.

**Estado**

O termo estado se origina das máquinas que foram precursoras do computador moderno. A máquina de estado mais simples é um interruptor, nele só possuem dois estados possíveis: ligado e desligado. Em determinado momento ele deve apresentar um, e somente um, desses estados possíveis.

Outro exemplo simples, porém, mais complexo, seria um semáforo, onde são apresentados 3 estados possíveis: verde, amarelo e vermelho. Mais uma vez em um dado momento ele precisará apresentar um, e somente um desses três estados.

Em programação moderna a quantidade de estados é muito mais difícil de ser calculada do que na máquina de estados. Imagine um objeto que tenha apenas um número inteiro dentro dele. Esse exemplo simples já apresenta 2³² estados possíveis (inteiros usam 32 bits de memória). Se houver uma string piora mais, os estados tenderiam ao infinito porque a string não tem limite de tamanho. Sendo assim, os estados possíveis são limitados pela capacidade de endereçamento da máquina ou da memória ram disponível, o que for menor. Sendo assim, não vamos nos preocupar na maioria dos casos em descobrir quantos estados possíveis existem, mas apenas em armazenar ou modificar o estado atual do componente.

Para criarmos um componente com estado no React é fácil, em seu construtor devemos inserir um modelo do estado no seu valor padrão. Utilizaremos então um método chamado de setState() para quando quisermos alterar o estado.

import React from 'react';

class App3 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { nome : undefined }

  }

  render(){

    return(

      <p>Olá {*this*.state.nome}</p>

    );

  }

}

export default App3;

O estado só deve ser atribuído diretamente uma vez no construtor. Todas as modificações subsequentes devem ser feitas pelo método setState(), pois ele indica ao React que o componente deve se atualizar na tela. Inserimos então um campo de texto em que o usuário consiga modificar o valor do nome no estado do componente:

import React from 'react';

class App3 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { nome : '' }

  }

  render(){

    return(

      <>

        nome: <input *type*='text' *value*={*this*.state.nome}/>

        <p>Olá {*this*.state.nome}</p>

      </>

    );

  }

}

export default App3;

Ao tentarmos escrever o campo não permite que seja alterado o valor, isso ocorre porque vinculamos ele ao valor do estado quando dissemos value={this.state.nome}.

Como fazemos então para alterar os valores?

Precisamos vincular o evento de change desse campo ao estado, para isso faremos uma função.

import React from 'react';

class App3 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { nome : '' }

  }

  changeNome = function(*evt*){

*this*.setState({ nome : *evt*.target.value});

  }

  render(){

    return(

      <>

        nome: <input *type*='text' *value*={*this*.state.nome} *onChange*={*this*.changeNome}/>

        <p>Olá {*this*.state.nome}</p>

      </>

    );

  }

}

export default App3;

Fizemos uma função que vai receber por parâmetro o objeto Event do JavaScript (o mesmo que receberíamos em JavaScript tradicional) e de dentro dele pegamos o target, que é o elemento HTML que gerou o evento.

De dentro do elemento, no caso de campos de texto, o valor do campo está em value.

Usamos a função setState e passamos para ela um objeto com a alteração que queremos fazer: no caso, queremos alterar o campo nome do estado para o valor do campo.

Esse código produz um erro, não podemos passar batido por ele. Quando fazemos métodos no React, em componentes tipo classe, sempre temos que fazer o bind do this.

Javascript tem um problema, funções tem sempre uma variável chamada this, mas ela muda de significado de função para função. Isso é uma das maiores dificuldades dessa linguagem.

Precisamos dizer para a função changeNome que o this dela deve ser o apontamento para o mesmo this no contexto de classe, um "apontamento para si mesmo". Se esquecer esse passo, nenhum método que utilize o this para ler/modificar o estado ou as props funcionará.

Uma linha resolve esse problema, basta adicionar ao construtor:

*this*.changeNome = *this*.changeNome.bind(*this*);

import React from 'react';

class App3 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { nome : '' }

*this*.changeNome = *this*.changeNome.bind(*this*);

  }

  changeNome = function(*evt*){

*this*.setState({ nome : *evt*.target.value});

  }

  render(){

    return(

      <>

        nome: <input *type*='text' *value*={*this*.state.nome} *onChange*={*this*.changeNome}/>

        <p>Olá {*this*.state.nome}</p>

      </>

    );

  }

}

export default App3;

Se isso parece uma gambiarra para você, uma alternativa é usar arrow functions para os métodos, como elas nunca alteram o contexto do this elas funcionam nesse contexto sem gerar o problema que tivemos.

import React from 'react';

class App3 extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { nome : '' }

  }

  changeNome = (*evt*) => {

*this*.setState({ nome : *evt*.target.value});

  }

  render(){

    return(

      <>

        nome: <input *type*='text' *value*={*this*.state.nome} *onChange*={*this*.changeNome}/>

        <p>Olá {*this*.state.nome}</p>

      </>

    );

  }

}

export default App3;

Observe que ao digitar qualquer letra o estado já vai se modificando e todos os elementos HTML que utilizam aquele estado são atualizados imediatamente.

Uma pequena observação sobre o setState(): não é necessário passar o objeto completo do estado para o setState, ele é inteligente o suficiente para fazer modificações parciais.

Ou seja se temos um estado como { nome : 'teste', idade : 20 } e fazemos setState({ idade : 21 }), apenas a idade é atualizada, o estado não é substituído pelo objeto que passamos perdendo o nome. Isso é bastante prático!

**Renderização Condicional**

Vimos que o estado de um componente guarda valores que podem ser usados para serem mostrados na tela ou alterados para que o componente reaja a eventos.

Agora usaremos um valor do estado para renderizar condicionalmente JSX diferentes.

import React from 'react';

class App4 extends *React*.Component {

  constructor(*props*) {

*super*(*props*);

*this*.state = {

      nome: undefined,

      txtNome: ''

    }

  }

  changeNome = (*evt*) => {

*this*.setState({ txtNome: *evt*.target.value });

  }

  persistTxtNome = () => {

*this*.setState({nome : *this*.state.txtNome});

  }

  render() {

    return (

      <>

      </>

    );

  }

}

export default App4;

Como no tópico anterior temos um componente com um estado que contém um nome, agora de valor padrão undefined. Também temos um txtNome que gravará alterações em um campo de texto.

O componente contém um método para alterar o nome e um método para pegar o valor de txtNome e colocar em nome.

Limpamos o método render porque é nele que vamos nos focar.

Algo importante a se saber é que em JSX estamos retornando um valor. Expressões de lógicas de programação que não produzam valores, não podem aparecer dentro do JSX, incluindo condicionais if e elsem, switch e laços de repetição.

No entanto, se desejarmos usar um condicional é possível: Ele deve aparecer antes do return no método render ou podemos usar condicional ternário. Vamos modificar o nosso método render para ter telas diferentes se tivermos ou não o nome para mostrar:

import React from 'react';

class App4 extends *React*.Component {

  constructor(*props*) {

*super*(*props*);

*this*.state = {

      nome: undefined,

      txtNome: ''

    }

  }

  changeTxtNome = (*evt*) => {

*this*.setState({ txtNome: *evt*.target.value });

  }

  persistTxtNome = () => {

*this*.setState({nome : *this*.state.txtNome});

  }

  render() {

    if(!*this*.state.nome){

      return (

        <>

          Nome: <input *type*='text' *onChange*={*this*.changeTxtNome}/>

          <button *onClick*={*this*.persistTxtNome}>Salvar</button>

        </>

      )

    }

    else{

      return <p>Olá {*this*.state.nome}</p>

    }

  }

}

export default App4;

Observe que pela limitação descrita tivemos que fazer dois returns. Tendo isso em consideração, é comum no React, até porque é boa prática evitar múltiplos return quando possível, fazer uma função que renderiza uma versão do componente e outra que renderiza a outra e decidir com ternário qual usar.

Fazendo isso temos o seguinte código:

import React from 'react';

class App4 extends *React*.Component {

  constructor(*props*) {

*super*(*props*);

*this*.state = {

      nome: undefined,

      txtNome: ''

    }

  }

  changeTxtNome = (*evt*) => {

*this*.setState({ txtNome: *evt*.target.value });

  }

  persistTxtNome = () => {

*this*.setState({ nome: *this*.state.txtNome });

  }

  render() {

    const renderForm = () => {

      return (

        <>

          Nome: <input *type*='text' *onChange*={*this*.changeTxtNome} />

          <button *onClick*={*this*.persistTxtNome}>Salvar</button>

        </>

      )

    };

    const renderText = () => (<p>Olá {*this*.state.nome}</p>);

    return !*this*.state.nome ? renderForm() : renderText(); //operador ternário

  }

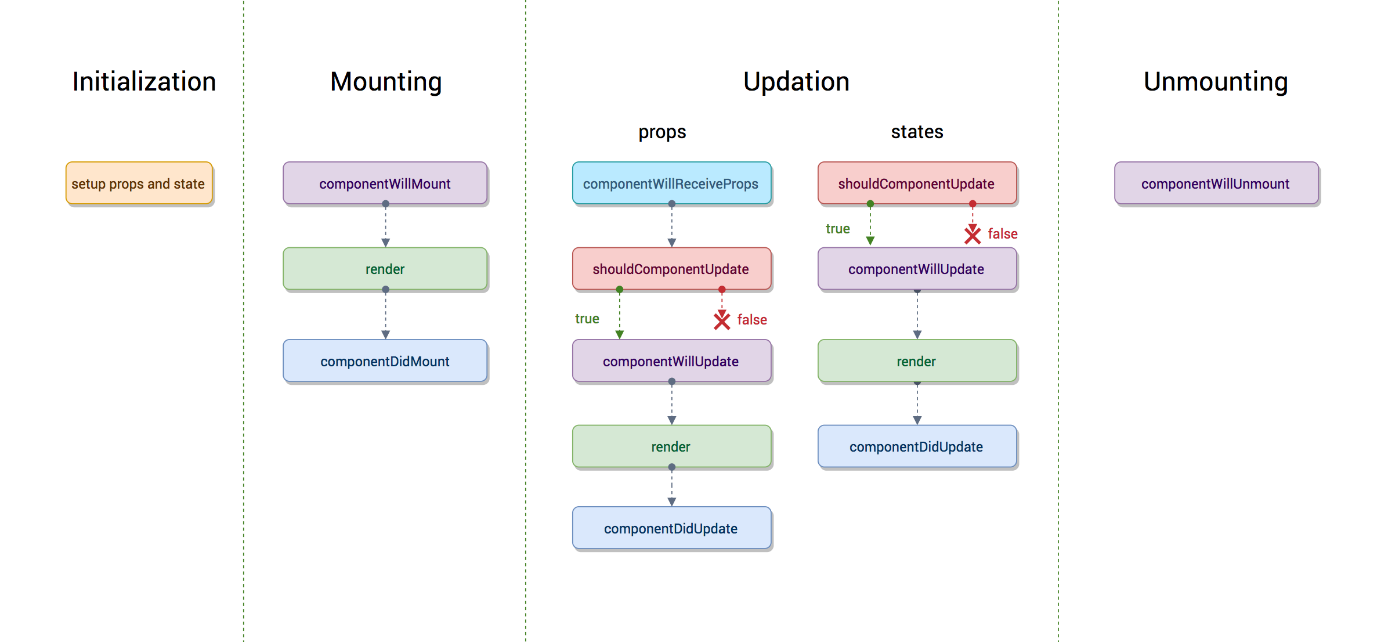
}

export default App4;

Vale ressaltar nesse exemplo que tivemos que fazer um campo no state para guardar os valores do campo de texto enquanto o botão não era clicado, para só então jogar esse valor no campo nome.

**Ciclo de Vida**

Assim como a gente os componentes também possuem um ciclo de vida. Imagine que você acessou uma página e lá há um componente. A primeira coisa que acontece com esse componente é ser construído e em seguida ser renderizado na tela. Depois disso ele entra em um processo de atualização dos seus valores, ou seja, conforme o usuário vai interagindo na página, os seus valores de props e estados são alterados e ele é renderizado na tela. Por fim a o processo de morte do componente, que é quando o usuário fecha a aba ou ele vai para outra página e o componente é destruído.

 Mas para que serve tudo isso? Imaginemos que você irá construir um componente que inicialmente você terá que popular ele com dados, ou seja, assim que ele é renderizado. O que você gostaria de fazer é ter acesso a esse componente nesse momento, para isso o React tem métodos específicos para cada etapa desse ciclo de vida. Você como programador pode criar um código que será executado em um momento específico do ciclo de vida do componente. Por exemplo salvar os dados de uso da sessão ao usuário sair dela.

Os estágios do ciclo de vida estão representados na imagem acima, caso seja necessário fazer algum processo em algum desses estágios, basta adicionar um método com mesmo nome. Por exemplo, em React o momento mais comum de interação com o ciclo de vida é o componente *componentWillMount* que é um estágio antes da renderização na tela. Porém, em novas versões do React, o uso desse método é desencorajado, sendo orientado a criação de lógicas pré-construção do componente no construtor da classe ou no *componentDidMount*.

É preciso prestar atenção porque alguns dos métodos do ciclo de vida irão se tornar depreciados nas próximas versões do React, ou seja, eles não estarão mais disponíveis, desta forma é importante evita-los.

Para uma maior compreensão vamos aos códigos criados em aula.

Utilizando o método *componentDidMount():*

import React from 'react';

class App5 extends *React*.Component {

  constructor(*props*) {

*super*(*props*);

*this*.state = {

      nome: undefined,

      txtNome: ''

    }

  }

  componentDidMount = () => {

  }

  changeTxtNome = (*evt*) => {

*this*.setState({ txtNome: *evt*.target.value });

  }

  persistTxtNome = () => {

*this*.setState({ nome: *this*.state.txtNome });

  }

  render() {

    const renderForm = () => {

      return (

        <>

          Nome: <input *type*='text' *onChange*={*this*.changeTxtNome} />

          <button *onClick*={*this*.persistTxtNome}>Salvar</button>

        </>

      )

    };

    const renderText = () => (<p>Olá {*this*.state.nome}</p>);

    return !*this*.state.nome ? renderForm() : renderText();

  }

}

export default App5;

Vamos usar o método para procurar no sessionStorage do navegador se ele tem um valor para o nome.

componentDidMount = () => {

  const nome = sessionStorage.getItem('nome');

  if(nome) *this*.setState({nome});

  }

Vamos aproveitar para colocar o nome no sessionStorage quando o usuário clicar no botão:

persistTxtNome = () => {

*this*.setState({ nome: *this*.state.txtNome });

  sessionStorage.setItem('nome', *this*.state.txtNome);

}

O componentDidMount busca o valor no sessionStorage do navegador (cookies) e altera o estado assim que o componente renderiza. Como informado anteriormente, é preciso estar atento a esses métodos do ciclo de vida, pois nas novas versões do React alguns estarão depreciados, tornando assim seu uso inadequado nessas novas versões.

**Renderização de listas**

Há duas abordagens no uso de lista que podemos tomar ao construir códigos em React:

1. Renderizar os items usando HTML simples;
2. Criar um componente que será utilizado como item da lista.

As duas apresentam vantagens e desvantagens. Na primeira é mais simples, porém em itens mais complexos pode se tornar confuso e dificilmente será reaproveitada. Na segunda a abordagem é mais complexa, podendo ter um ou até mesmo dois componentes extras responsáveis pela renderização da lista. Por sua vez, podem ser reaproveitados e tem uma melhor funcionalidade quando esses itens possuem um HTML complexo.

Vamos começar com uma lista simples:

import React from 'react';

class Lista extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = {items: ["item1", "item2", "item3", "item4"]}

  }

  render(){ }

}

export default Lista;

A forma mais simples e mais utilizada na renderização de listas em React é utilizando a função map(). Ela deve produzir um JSX para cada item da lista.

Completando o render():

render(){

  return (

    <ul>

      {*this*.state.items.map(item, *index* => <li *key*={*index*}>item</li>)}

    </ul>

  )

 }

Analisando o código acima, é importante ressaltar o uso do identificador KEY nas tags <li></li>. Ele é o responsável por permitir que o React identifique cada item como único, uma vez que cada item possuí um identificador, seja passado como parâmetro ou como índice do vetor que está inserido. Isso permite que caso ocorra uma alteração em um dos itens no state, apenas aquele item seja atualizado no componente.

Todas as vezes que criarmos itens através do map() é nessário que o Key seja passado, ele pode ser absolutamente qualquer coisa, desde que seja único, ou seja, nenhum item tenha um key igual ao outro. No caso acima utilizamos o índice (posição no array) do item para gerar seu key.

Vamos à um exemplo mais complexo de lista, cujo vetor possuí objetos em suas posições.

import React from 'react';

class Lista extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = {items: [

      {id: 1, nome: 'item1', completo: false},

      {id: 2, nome: 'item2', completo: false},

      {id: 3, nome: 'item3', completo: true},

      {id: 4, nome: 'item4', completo: false},

    ]}

  }

  render(){

    return (

    )

   }

}

export default Lista;

No exemplo acima temos algo mais complexo para renderizar. O vetor possui índice, nome e um completo, que é uma variável booleana para indicar se ele deve ser riscado ou não. Será necessário então a criação de um componente capaz de renderizar toda essa informação e riscar o item dependendo de seu valor booleano. Isso faz com que a complexidade do componente não esteja centrada no componente Lista, sendo ela dividida com o componente Item também.

class Item extends *React*.Component{

  constructor(*props*){

*super*(*props*);

  }

  render(){

    const textDecoration = *this*.props.completo ? 'line-through' : 'none'

    return (

      <li *data-id*={*this*.props.id} *style*={{ textDecoration }}>

        {*this*.props.children}

      </li>

    )

  }

}

Complementamos o render() do componente Lista:

render(){

  return (

    <ul>

      {*this*.state.items.map(item, *index* => (

        <*Item* *key*={item.id} *id*={item.id} *completo*={item.completo}>

          {item.nome}

        </*Item*>

      ))}

    </ul>

  )

 }

Como há um ID nos objetos do vetor ele foi passado para o Key para distinguir cada item. Vale apontar que outras linguagens de programação utilizam laços para fazer a renderização de listas. No React, vale a pena adotar a abordagem do uso do map, pois diferentemente dos laços ele produz um resultado JSX imediatamente.

**Componentes Aninhados**

Vamos explorar um pouco mais o uso do CHILDREN. Começamos modificando o exemplo anterior para permitir a passagem de itens também como filhos.

import React from 'react';

class Lista extends *React*.Component{

  constructor(*props*){

*super*(*props*);

*this*.state = { items: [

      {id: 1, nome: 'item1', completo: false},

      {id: 2, nome: 'item2', completo: false},

      {id: 3, nome: 'item3', completo: true},

      {id: 4, nome: 'item4', completo: false},

    ]}

  }

  render(){

    return (

      <ul>

        {*this*.props.children}

        {*this*.state.items.map(*item* => (

          <*Item* *key*={*item*.id} *id*={*item*.id} *completo*={*item*.completo}>

            {*item*.nome}

          </*Item*>

        ))}

      </ul>

    )

   }

}

class Item extends *React*.Component{

  constructor(*props*){

*super*(*props*);

  }

  render(){

    const textDecoration = *this*.props.completo ? 'line-through' : 'none'

    return (

      <li *data-id*={*this*.props.id} *style*={{ textDecoration }}>

        {*this*.props.children}

      </li>

    )

  }

}

export {Lista, Item};

Dessa forma, podemos passar os itens tanto como filhos para o componente como pegar os que estão no estado e renderizar. O uso seria na página index.js inserir o código abaixo, que apareceria acima dos que já estão presentes no estado.

<*Lista*>

    <*Item* *key*={999} *id*={999} *completo*={true}>teste</*Item*>

    <*Item* *key*={1000} *id*={1000} *completo*={false}>teste2</*Item*>

</*Lista*>

Dependendo de onde colocamos a linha this.props.children podemos colocar os items passados como filhos antes ou depois daqueles que estão no state. Basta colocar a linha antes ou depois do map(). Outra técnica bastante interessante é restringir que os filhos passados para o componente são de um tipo determinado, assim não são passados valores indevidos. Por exemplo, imagine que alguém inseriu elementos que não são do tipo lista.

<Lista>

Teste1

Teste2

<Item key={999} id={999} completo={true}>teste</Item>

<Item key={1000} id={1000} completo={false}>teste2</Item>

</Lista>

Com o código do jeito que está ele renderiza esses elementos como texto e não como lista. Isso pode ser evitado com o operador tenário e o map.

Trocamos então isto:

{*this*.props.children}

Para isto:

{*React*.Children.map(*child* => *child*.type == Item ? *child* : null)}

Esse map pode ser utilizado para fazer qualquer restrição ou transformação nos items passados como children do componente.

No exemplo, fizemos a verificação se o filho é uma instância da classe Item, se for retornamos ele, se não for retornamos null.

Além do map, o mesmo objeto React.Children tem outros métodos úteis como forEach, count, toArray, que são autoexplicativos e only que restringe que apenas um filho seja passado e renderizado.

**Formulários**

O nosso formulário terá um campo de texto para o nome, um *select* para escolher a linguagem de programação preferida dada as opções, um campo *radio* de marcação única se o usuário é programador ou estudante, um checkbox se o usuário dedica 8h semanais aos estudos e por fim uma área de texto para uma bio do usuário.

Então começamos adicionando a tag *form* e com o atributo *onSubmit*, que recebe uma função a ser executada quando o formulário for submetido.

import React from 'react';

class Formulario extends *React*.Component {

  constructor(*props*) {

*super*(*props*);

*this*.state = {nome: '', linguagem: 'JavaScript', tipo: 'programador', dedico: true, bio: ''};

*this*.handleSubmit = (*event*)=>{

*event*.preventDefault();

          console.log(*this*.state);

      }

*this*.changeName = (*event*)=>{

*this*.setState({nome: *event*.target.value})

      }

  }

  render(){

      return (

          <>

            <form *onSubmit*={*this*.handleSubmit}>

                <label>

                    Nome: <input *type*="text" *value*={*this*.state.nome} *onChange*={*this*.changeName} />

                </label>

                <br />

                <input *type*="submit" *value*="Salvar"/>

            </form>

          </>

      );

  }

}

export default Formulario;

Observe que mesmo estando no contexto React o HTML tentará se comportar como HTML comum ao submeter o formulário. Para evitar esse comportamento, que gera o *refresh* da página, precisamos utilizar o método preventDefault de Event:

event.preventDefault();

Para cada novo campo de *input* nós precisamos criar um método para processar as alterações do mesmo.

Vamos adicionar o *input* tipo *select*, no *render* colocamos:

<label>

Linguagem favorita:

<select *value*={*this*.state.linguagem} *onChange*={*this*.changeSelect}>

    <option *value*="JavaScript”>JavaScript</option>

    <option *value*="Python”>Python</option>

    <option *value*="C++”>C++</option>

</select>

</label>

<br />

No construtor adicionamos o método this.changeSelect:

*this*.changeSelect = (*event*)=>{

*this*.setState({linguagem: *event*.target.value})

}

Para o *input* tipo *radio* adicione no *render*:

<label>

    Sou:

    <input *type*="radio" *checked*={*this*.state.tipo == 'programador'} *onChange*={*this*.changeRadio} *value*="programador" /> Programador

    <input *type*="radio" *checked*={*this*.state.tipo == 'estudante'} *onChange*={*this*.changeRadio} *value*="estudante"/> Estudante

</label>

<br/>

Para termos a marcação de uma única opção, o campo *checked* verifica a condição da opção marcada no estado, se for vendadeiro é renderizado como marcado.

No construtor adicionamos o método this.changeRadio:

*this*.changeRadio = (*event*)=>{

*this*.setState({tipo: *event*.target.value})

}

Para o checkbox temos a mesma atenção ao campo *checked*:

<label>

    <input *type*="checkbox" *checked*={*this*.state.dedico} *onChange*={*this*.changeCheckbox} /> Dedico 8h semanais aos estudos.

    </label>

<br/>

Agora atenção para o método this.changeCheckbox, diferente dos outros casos não veficamos o *value* desse tipo de *input* para alterar o valor do estado do componente mas sim o *checked*:

*this*.changeCheckbox = (*event*)=>{

*this*.setState({dedico: *event*.target.checked})

}

Por fim para a área de texto usamos a tag *textarea*:

<label>

    Bio:

    <textarea *cols*="50" *value*={*this*.state.bio} *onChange*={*this*.changeBio} />

</label>

O método segue como os demais:

*this*.changeBio = (*event*)=>{

*this*.setState({bio: *event*.target.value})

}

No exemplo, fizemos diversas mudanças no estado baseadas em eventos do tipo onChange, também "sequestramos" a submissão do formulário usando onSubmit.

Ele impede que o comportamento padrão do formulário aconteça e nos permite processar o formulário de acordo com o padrão usado no React. Esquecer de invocar esse método faz com que o formulário seja mandado para o servidor e a página recarregue.

Com isso concluímos nosso estudo pelos componentes feitos a partir de classes. Vamos agora ver componentes funcionais.

**Introdução a Componentes Funcionais**

Os componentes funcionais eram considerados os "componentes burros" do React, eles eram componentes simplificados que só tinham props e renderizavam html na tela. Eles não eram capazes de ter estado e, portanto, de realizar alterações de estado. Não tinham os métodos de ciclo de vida, o que não permitia que coletassem dados externamente, o que chamamos na programação funcional de efeitos colaterais (side effects).

E por esses motivos eram usados apenas para coisas mais simples no React. No entanto, nas últimas atualizações da biblioteca, eles foram revistos e o React introduziu o conceito de Hooks. Os hooks permitem aos componentes funcionais terem estado, efeitos, modificar seu estado e uma série de outras capacidades.

Sendo assim, os componentes tipo classe se tornaram obsoletos e hoje a recomendação é que tudo seja feito por componentes funcionais. O time do React é funcional, portanto, é natural que eles tenham a tendência de não depender da abordagem orientada a objetos usada nos componentes tipo classe.

Componentes funcionais são de fato mais simples, eles são apenas uma função, veja um exemplo:

import React from 'react';

export default function Hello(){

    return (<h1>Hello!</h1>);

}

Simples assim construímos um componente funcional que renderiza um H1 com o texto "Hello!". Para fazer um paralelo com os componentes tipo classe que vimos, pense que o componente funcional é o método render do componente tipo classe. Seu retorno é o que será mostrado na tela.

Diferentemente do render() o componente funcional recebe props em seu parâmetro, então basta utilizá-las diretamente. Veja o exemplo:

import React from 'react';

export default function Hello(*props*){

    return (<h1>{*props*.text}</h1>);

}

O uso no index.js é idêntico ao tipo classe:

<*Hello* *text*='Hello!!!'></*Hello*>

Da mesma forma podemos passar children:

import React from 'react';

export default function Hello(*props*){

  return <>{*props*.children}</>;

}

O uso no index.js também é idêntico:

<*Hello*>

    <h1>Hello!</h1>

</*Hello*>

Uma prática muito comum em componentes funcionais é desestruturar o props imediatamente dentro do parâmetro do componente, o que nos desobriga de escrever props. sempre que o utilizarmos. Veja os exemplos anteriores usando desestruturação:

import React from 'react';

export default function Hello({*text*}){

    return (<h1>{*text*}</h1>);

}

import React from 'react';

export default function Hello({*children*}){

  return <>{*children*}</>;

}

Componentes extremamente simples podem ser criados como arrow functions:

import React from 'react';

const Titulo = ({*text*}) => <h1>{*text*}</h1>;

export default Titulo;

Mesmo que aparentemente não seja usada, não esqueça da importação do React nos componentes funcionais! Ela garante o uso do JSX no componente.

Você pode achar estranho que tanto uma função como uma classe podem ser componentes no React, mas lembre-se, a orientação a objetos do JavaScript é ilusória, por mais que estejamos fazendo uma classe, por baixo dos panos o JavaScript vai transformá-la em uma função (ele não tem o conceito de classe).

Sendo assim, classes e funções são a mesma coisa. Você também pode estar pensando: sacanagem mostrar componentes como classes para depois dizer que eles estão em desuso!

Fazemos isso porque em códigos legados existe grande chance de você cruzar com componentes como classes, pois eles foram por muito tempo a única forma de componente em que os conceitos de estado e ciclo de vida existiam.

Também porque, agora que você conhece todos os conceitos vistos nos componentes tipo classes, só precisamos adaptá-los aos componentes funcionais, você já sabe tudo o que é necessário. Sendo assim, só será necessário conhecer os hooks.

**Componentes Funcionais com Estado (useState hook)**

Como dito anteriormente, antigamente componentes funcionais não possuíam a capacidade de ter estados, fato alterado nas novas versões do React. O hook de estado é chamado de useState() e é necessário importa-lo de dentro do React antes de poder utilizá-lo em nossos componentes.

import React, {useState} from 'react';

UseState é uma função que produz um vetor, cuja primeira posição vem uma referência ao estado que acabou de criar. Na segunda posição retorna uma função que permite modificar o estado. Caso deseje um valor padrão inicial ele pode ser passado por parâmetro na chamada do useState.

Como é estranho usar um array dessa forma, sempre optamos pela desestruturação imediata do retorno do useState em duas constantes. Veja o exemplo:

import React, {useState} from 'react';

export default function App(){

  const [nome, setNome] = useState('teste');

  return (<h1>{nome}</h1>);

}

Para ler o estado do nome, utilize a constante nome.

Para alterar o estado no nome, utilize a função setNome().

Os nomes das constantes você escolhe (tanto da constante de leitura como da função de alteração), mas é muito conveniente manter essa convenção de usar na função o mesmo que foi usado na constante precedido da palavra set. Que é uma nomenclatura muito comum no Java, de onde o JavaScript herda sua sintaxe.

Uma diferença importantíssima entre a função de alteração do estado criada pelo useState e a função setState que usávamos nos componentes tipo classe, é que ela não aceita a alteração parcial de um objeto, ela sobrescreve o objeto completamente, que deve ser passado em sua totalidade.

Imagine o seguinte estado:

{ nome: 'teste', idade: 20 }

Em componentes como classe podíamos fazer:

setState({ idade: 21 });

Ele manteria o restante do objeto e alteraria apenas a idade.

Em componentes funcionais devemos fazer (assumindo que setState é nossa função de alteração):

setState({ nome : 'teste', idade: 21 });

Quer dizer que temos que saber quais são os valores de todas as outras propriedades do estado para fazer a alteração de apenas uma??

Na verdade não, novamente a desestruturação pode nos ajudar:

setState({ ...state, idade: 21 });

Usando desestruturação de objetos, todas as propriedades que ele tem, assim como seus valores, são passadas para o novo objeto. Assim podemos passar em seguida a propriedade que desejamos alterar e seu novo valor.

Essa sintaxe também serve para adicionar novas propriedades no objeto.

Vamos agora fazer um exemplo completo de uso de state com leitura e alteração do estado:

import React, {useState} from 'react';

export default function App(){

  const [nome, setNome] = useState('');

  return (

    <>

      <input *type*='text' *onChange*={(*evt*) => setNome(*evt*.target.value)} *value*={nome} />

      <h1>{nome}</h1>

    </>

  );

}

No exemplo acima criamos um estado chamado nome e uma função setNome para alterá-lo. O valor inicial do nome é dado pelo parâmetro passado ao setState, no caso string vazia.

Conforme o usuário digita no campo de texto, o nome é alterado para o valor do campo. Isso faz com que o React atualize imediatamente o H1 abaixo.

Lembre-se, da mesma forma que nos componentes tipo classe, nunca faça atribuições ao estado diretamente, sempre utilize a função. Senão o React não atualizará o componente para mostrar a mudança na tela.

Para garantir que esse erro não aconteça, sempre use const antes de desestruturar o useState nunca use let.

Nos componentes funcionais podemos fazer um estado para cada valor que desejamos utilizar, diferentemente das classes em que apenas um estado era possível e deveria sempre ser um objeto.

Por isso, é aconselhável fazer mais useState, por exemplo, um para cada valor em vez de fazer apenas um com um objeto dentro.

Claro que fica a critério e bom senso do programador quando usar objetos e quando guardar os valores separadamente.

Leve em conta a complexidade, quais deles geram atualizações na tela e quantos objetos você precisa guardar, se for mais de um, opte por colocá-los inteiros cada um em seu state em vez de fazer um state para cada propriedade.

Em seguida passamos para o hook que equivale ao ciclo de vida em um componente funcional.

**Componentes Funcionais com Efeitos (Ciclo de Vida)**

Dissemos que o ciclo de vida dos componentes tipo classe são utilizados para fazer "efeitos colaterais" (side effects), que é o nome que programadores funcionais dão para qualquer operação que seja feita fora de nossa função, por exemplo, consultas/alterações em bancos de dados ou chamadas em APIs.

Essa palavra é importante pois o hook que usamos para isso em componentes funcionais é chamado de useEffect. Esse é um hook bem mais complexo que useState.

O useEffect é uma função que recebe dois parâmetros, o primeiro é uma função de callback que rodará e produzirá o nosso efeito, em outras palavras, é o que queremos fazer. O segundo é um vetor, nesse vetor colocamos o nome de variáveis que serão monitoradas pelo effect, caso essas variáveis mudem de valor o effect rodará novamente.

As funções de callback passadas para o useEffect podem opcionalmente retornar uma outra função. Essa função seria usada para fazer uma limpeza do que foi feito pelo effect. Por exemplo, se fizemos uma consulta ao BD, podemos usar essa função para fechar a conexão.

Efeitos rodam imediatamente na montagem do componente, mas também rodam novamente todas as vezes que o componente fizer update. Por isso, geralmente trabalhamos com o vetor de variáveis e condicionais para garantir que ele não rode mais vezes do que o desejado.

Muito cuidado com efeitos que manipulam o estado, pois isso gera um update e consequentemente roda o efeito novamente. Se não restringirmos esse comportamento podemos fazer um laço infinito de atualizações que fará com que nosso componente não renderize.

Vamos começar usando o componente da lição passada e adicionar um effect nele que faz console.log do nome.

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

export default function App(){

  const [nome, setNome] = useState('');

  useEffect(() => {

    console.log('Effect: ', nome);

  });

  return (

    <>

      <input *type*='text' *onChange*={(*evt*) => setNome(*evt*.target.value)} *value*={nome} />

      <h1>{nome}</h1>

    </>

  );

}

Observe que ele faz um log imediatamente quando o componente monta com o nome vazio.

Depois ele roda todas as vezes que o nome é alterado, letra a letra.

Observe que não passamos um array para o effect, isso quer dizer que ele vai rodar toda vez que qualquer estado for alterado. Agora, vamos modificar o useEffect para adicionar o parâmetro que falta. Vamos passar um array vazio:

useEffect(() => {

  console.log('Effect: ', nome);

}, []);

Observe que agora nosso efeito rodou apenas uma vez na montagem do componente.

Ele não roda mais quando o nome é alterado porque não dissemos que ele deve monitorar o estado do nome.

O array vazio representa que esse efeito não deve monitorar nenhum estado no componente.

Se passarmos o nome nesse array, ele monitorará apenas o estado do nome, não rodando na alteração de outros estados.

useEffect(() => {

  console.log('Effect: ', nome);

}, [nome]);

Sendo assim podemos criar uma lógica que faça o effect ler o nome do sessionStorage (cookie) quando rodar pela primeira vez, e depois monitorar o nome e gravar no sessionStorage o novo valor quando ele for alterado.

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

export default function App(){

  const [nome, setNome] = useState(undefined);

  useEffect(() => {

    if(nome === undefined){

      setNome(sessionStorage.getItem('nome') || "");

    }

    else{

      sessionStorage.setItem('nome', nome);

    }

  }, [nome]);

  return (

    <>

      <input *type*='text' *onChange*={(*evt*) => setNome(*evt*.target.value)} *value*={nome} />

      <h1>{nome}</h1>

    </>

  );

}

Configuramos nosso efeito para rodar no início (como sempre) e a cada mudança do estado do nome (passado no array de dependências).

Quando começamos, o nome estará como undefined então pudemos usar um if para verificar se ele está com esse valor e se estiver tentamos ler o valor do sessionStorage usamos || "" como fallback caso não exista valor no sessionStorage também.

Se o efeito cair no else, isso quer dizer que o nome foi alterado, como não há nada que o usuário possa fazer para fazer o nome ficar undefined novamente essa técnica é razoavelmente segura.

No else, persistimos a alteração do nome no sessionStorage observe que ao escrever no campo, o valor do H1 muda como antes, mas que se você recarregar a página o último valor continua preenchido!

Com isso chegamos ao final. Agora você pode fazer com um componente funcional tudo o que os componentes tipo classe podiam fazer e de forma mais concisa e enxuta.

**Bibliografia**

<https://blog.umbler.com/br/npm-vs-yarn-e-agora-quem-podera-nos-defender/?gclid=CjwKCAjw87SHBhBiEiwAukSeUcFRFN7_XtjKklkaJGIEV415X9v9htSheqgO4n9jfujlTsIrM-9x8BoCT2gQAvD_BwE>