**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.

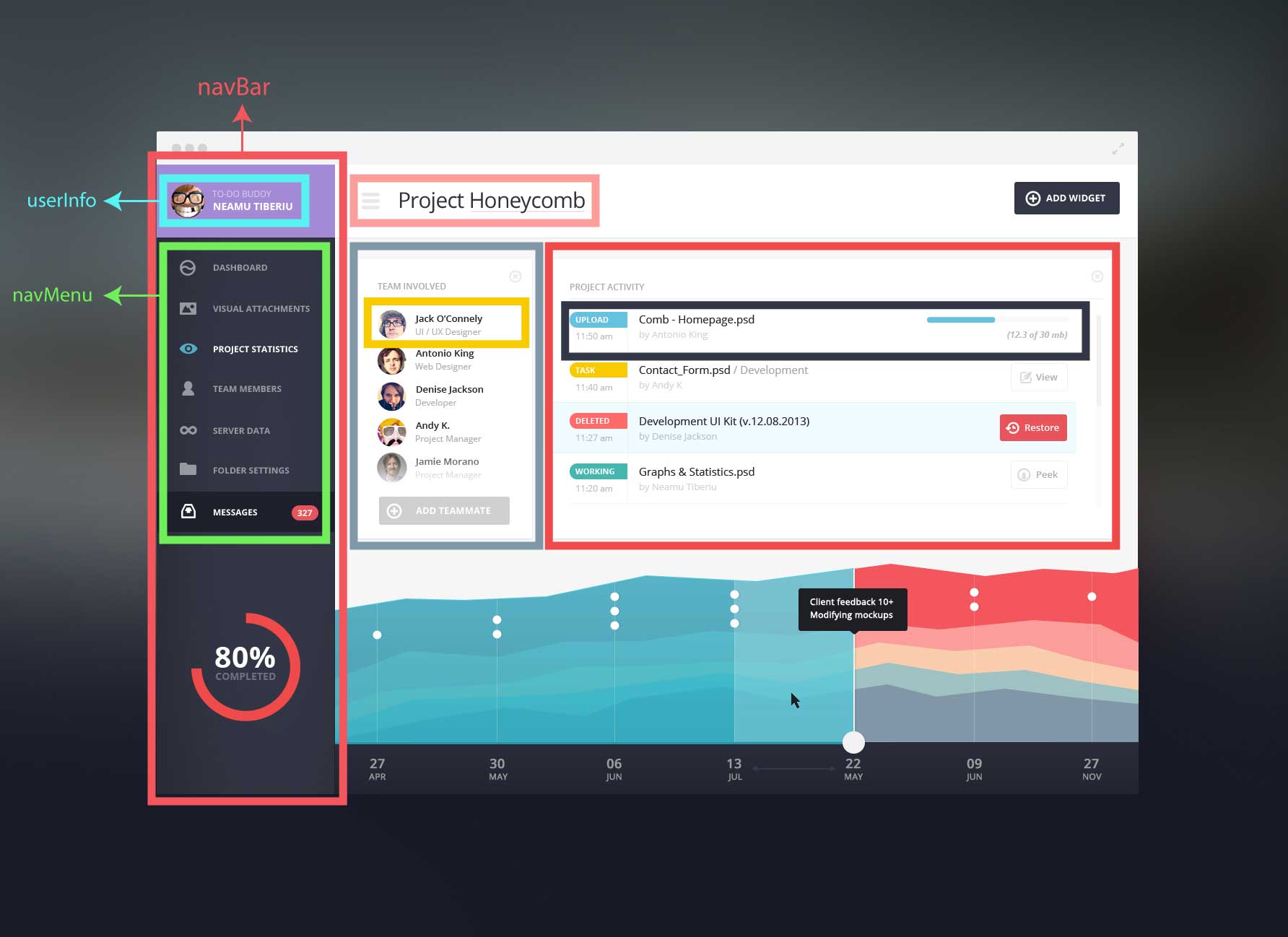
Nesse contexto surge o SPA – Single Page Applications, cuja função é utilizar uma única página HTML que sofre alterações conforme a interação dos usuários ou eventos do sistema interno. Dessa demanda surge o React, que hoje se tornou a principal biblioteca de front-end para a criação de SPA.

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.

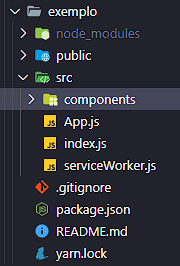
**YARN**

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>

</>

);

    }

}

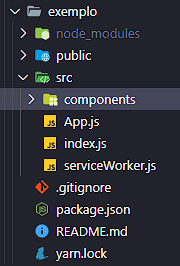
export default App;

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

  }

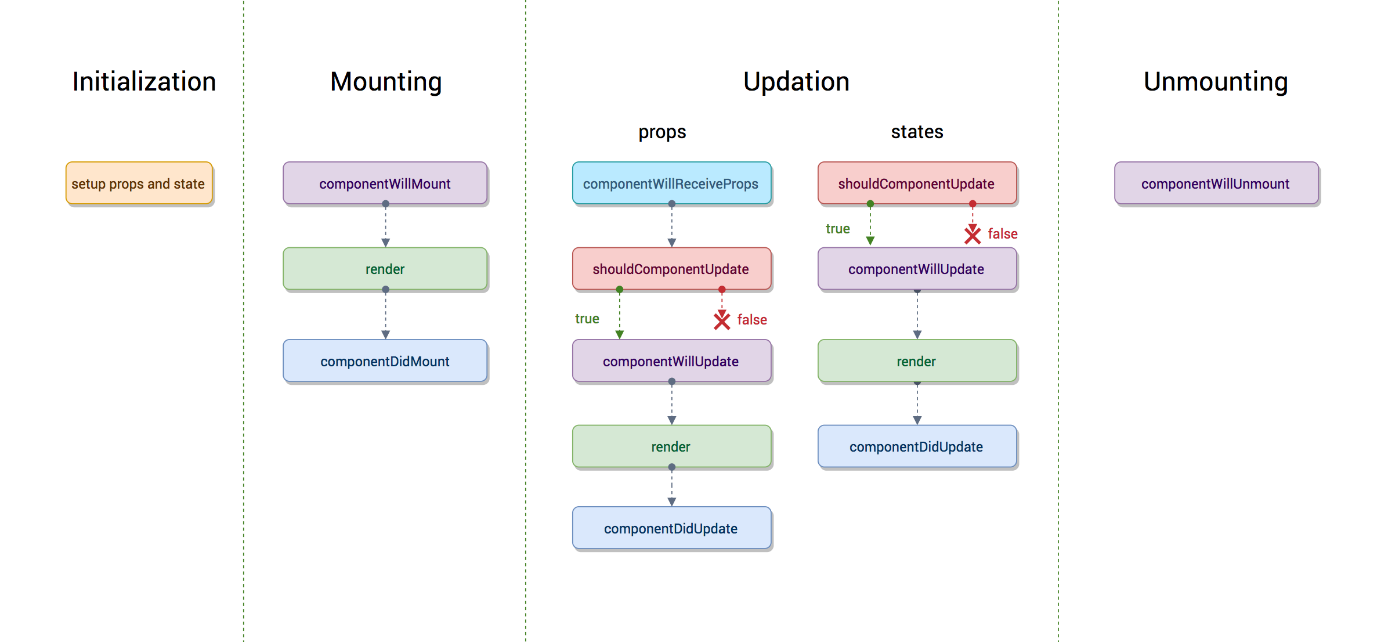
}

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.

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.

**Bibliografia**

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