**COMO O BROWSER ENTENDE O JSX?**

O babel e o webpack é quem convertem o JSX em JS para a compreensão do browser.

O babel pega o código e converte para JS.

O webback é um bundler, onde ele pega todos os seus arquivos (js, jsx, css, sass, jpg...), passa por dentro dele e converte em linguagem que o browser entende (js, css...).

**NPM RUN BUILD, FAZENDO SEU DEPLOY.**

**Npm run build** cria uma pasta que é seu projeto final para jogar em produção (em deploy). Você vai modificar toda sua pasta de desenvolvimento em SRC e ao final, quando seu projeto estiver pronto para o deploy, você vai criar a pasta build ao qual você não fará alterações e já é a sua aplicação montada de uma forma entendida pelo browser. Se você jogar o código todo do build em uma hospedagem, ele vai funcionar perfeitamente e otimizado.

**O REACT NÃO PRECISA ESTAR NA PÁGINA TODA!**

Se você abrir o index.html que está na pasta public (o arquivo do ‘root’), você consegue adicionar divs e continuar sua página usando apenas html. Ou seja, você não precisa usar react em sua página toda.

Você também pode criar uma outra div no index.html com a class “minha-div” e chama-la no index.js através de um document.querySelector.

**COMPONENTES FUNCIONAIS E DE CLASSE**

Componentes funcionais sempre retornam jsx. Podem ter estados com a importação do useState, que é o conhecido React Hooks.

Componentes de classe ficaram obsoletos com RH, sendo muito pouco usados, podendo ser com ou sem estado, como os funcionais, mas sem ter que importar o State. Dentro do componente de classe tem um render, que é uma função retornando JSX. Ou seja, o componente de classe é praticamente igual ao funcional, porém o funcional é mais moderno.

O que caracteriza o single page application é justamente os estados do componente. Quando você muda um estado você renderiza apenas o componente, sem renderizar a página por completo.

As arrows functions são extremamentes importantes em um componente de classe, visto que com elas você não precisa fazer o bind para acessar o estado de seu componente.

Sempre que ver o MAP lembrar de colocar o KEY no html do jsx.

**MÉTODOS DE CICLOS DE VIDA**

https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

**Inicialização**

Nesta fase, o componente React se prepara para sua inicialização, configurando os estados iniciais e props padrões se houverem.

**Montagem**

Depois de preparar com todas as necessidades básicas, estado e props, o nosso Componente React está pronto para ser montado no DOM do navegador. Esta fase fornece métodos que podem ser invocados antes e depois da montagem dos componentes. Os métodos que são chamados nesta fase são:

*componentWillMount*: É executado quando o componente estiver prestes a ser montado no DOM da página.

render, monta o componente no navegador. Este é um método puro, o que significa que ele dá a mesma saída sempre que a mesma entrada é fornecida.

*componentDidMount:* Este é o método que é executado depois que o componente foi montado no DOM. Este método é executado uma vez em um ciclo de vida de um componente e será após a primeira renderização. Com esse método podemos acessar o DOM, devemos inicializar bibliotecas JS como D3 ou Jquery, que precisa acessa-lo

**Atualização**

Esta fase começa quando o componente React já nasceu no navegador e cresce recebendo novas atualizações. O componente pode ser atualizado de duas maneiras, através do envio de novas props ou a atualização do seu estado.

Vamos ver a lista de métodos disponíveis quando o estado atual é atualizado chamando setState:

*shouldComponentUpdate:* Diz ao React que, quando o componente recebe novas props ou estado, o React re-renderiza ou pode ignorar a renderização do componente. Este método é uma questão, “O Componente foi atualizado?” Portanto, esse método deve retornar verdadeiro ou falso, e então o componente seria re-renderizado ou ignorado. Por padrão, esse método retorna verdadeiro.

*componentWillUpdate*: É executado somente quando shouldComponentUpdate devolver true. Este método é usado apenas para preparar a próxima renderização, semelhante ao componentWillMount ou ao construtor. Pode haver algum caso de uso quando precisar de algum cálculo ou preparação antes de renderizar algum item, este é um bom lugar para fazer isso.

*componentDidUpdate:* É executado quando o novo componente (já atualizado) foi atualizado no DOM. Este método é usado para reativar as bibliotecas de terceiros, usado para garantir que essas bibliotecas também atualizem e recarreguem.

**Desmontagem**

Nesta fase, o componente não é mais necessário e será desmontado do DOM. O método que se chama nesta fase é o seguinte:

*componentWillUnmount:* Este método é o último método no ciclo de vida. Isso é executado imediatamente antes de o componente ser removido do DOM.

Uso: neste método, fazemos todas as limitações relacionadas ao componente. Por exemplo, no logout, os detalhes do usuário e todos os tokens de autenticação podem ser apagados antes de desmontar o componente principal.

**DATA FETCHING**

Para consumir dados de um API em um class componente posso fazer assim:

//with fetch and promises.

  //componentDidMount () {

  //    fetch('https://jsonplaceholder.typicode.com/posts')

  //     .then (response => response.json())

  //     .then (posts => this.setState({ posts }))

  //  }

Ou fazer varias vezes uma única vez, dessa forma:

componentDidMount() {

*this*.loadPosts();

  }

  loadPosts = async () => {

    const postsResponse = fetch("https://jsonplaceholder.typicode.com/posts");

    const photosResponse = fetch("https://jsonplaceholder.typicode.com/photos");

    const [posts, photos] = await *Promise*.all([postsResponse, photosResponse]);

    const postsJson = await posts.json();

    const photosJson = await photos.json();

    const postsAndPhotos = postsJson.map((*post*, *index*) => {

      return { ...*post*, cover: photosJson[*index*].url };

    });

*this*.setState({ posts: postsAndPhotos });

  };

**ORGANIZAÇÃO DO APP**

Crio pasta componentes, onde colocarei meus componentes, no presente caso Posts e PostCard, button.

Crio pasta templates, onde irá meu home e as demais páginas de navegação.

Crio minha pasta útil que nada mais é para guardar funções, no presente caso é a função load-posts que faz o fetch da API.

No SRC deixo só o index.js e o setuptest.

**TESTE**

Para escrever o teste, você tem que componentizar o máximo possível sua aplicação, pois testar componente por componente é sempre muito mais fácil. Quando um componente fica muito grande, o teste fica mais difícil.

**QUAL A DIFERENÇA DE EU PASSAR UMA PROPRIEDADE OU UM ESTADO PARA MEU COMPONENTE?**

Quando tenho um estado, eu posso passar ele pra “baixo”. Por exemplo, eu tenho uma class Home e ela controla meu estado (estado global). Mas eu posso pegar o estado dessa home e enviar para outros componentes como props. Os componentes que são filhos (os retornos do render), podem receber essas propriedades e assim meu estado muda e os componentes também mudam, porque os valores das props que eu to passando para eles mudam.

Tanto o state quanto as props são privadas do componente que eu to, mas eu posso passar ele adiante.

OBS: Você não consegue mudar o estado de um componente pai através de um filho. Você pode criar novos estados para o filho quantas vezes você quiser, mas o pai ou você altera diretamente ele, ou não tem como voltar atrás.

<https://pt-br.reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down>

**É MUITO IMPORTANTE SABER ONDE VOCÊ COLOCAR O ESTADO!**

No caso do projeto do posts o ideal realmente era deixar todos os estados no home e sempre renderizar a página toda, pois nós estamos mexendo em toda a página e atualizando ela constantemente. Porém, em casos que nós só precisamos atualizar o estado de um componente, devemos colocar o estado exatamente neles, porque ai só renderizamos apenas o componente.

Em grandes aplicações, como no Facebook, se o estado dos componentes estivem todos em um “Home” e sempre tivéssemos que renderizar todos os componentes, a página ficaria muito pesada, pois a quantidade de componentes é enorme! Por isso, em grandes aplicações que contém vários componentes, o ideal é separar o estado por componente e só renderizar o necessário.

**EVENTOS SINTÉTICOS DO REACT**

Quando você manipula eventos no React, você está na maioria dos casos manipulando eventos sintéticos. Ou seja, eventos criados pelo próprio React mas que são muito similares a eventos criados pelo DOM. Exemplo: Evento onclick no html se escreve minúsculo e no react tem o mesmo evento mas se escreve em camel case (onClick).

<https://pt-br.reactjs.org/docs/events.html>

<https://pt-br.reactjs.org/docs/handling-events.html>