**React**

O React é o framework SPA que usaremos. Ele será responsável pela construção das nossas interfaces ou front-end no futuro.

Ferramentas que usaremos em conjunto com o React:

* Node 🡪 permite que a gente rode o Javascript na nossa máquina. Ao instalar o node, por padrão o NPM (node package manager) também é instalado. Ele é o gerenciador de pacotes do Node, portanto podemos utilizar de códigos prontos de pacotes.

**Introdução Parte 2**

* gitignore 🡪 serve para ignorar alguns arquivos quando fizermos versionamento de código;
* package-lock.json 🡪 serve para identificar qual é a versão instalada de cada pacote;
* package.json 🡪 tem diversas funcionalidades. As mais importantes são: listar todas as dependências instaladas no nosso projeto, quais dependências de desenvolvimento estão instaladas no nosso projeto e armazenar o react-scripts. Os scripts servem para executar alguma função, por exemplo, o script start serve para rodar a nossa aplicação em react. Para executar o script, use no terminal: npm nome\_script.

Exemplo: npm start.

Já nos outros scripts além do start usa-se: npm run nome\_script.

Exemplo: npm run build

**Webpack & Babel**

São duas tecnologias que junto com o Javascript constroem e dão vida ao React.

O Webpack agrupa o código da sua aplicação (bundle), auxilia na separação do código em diversos arquivos e trabalha em conjunto com o npm na importação dos pacotes instalados.

Já o Babel transforma o código JS novo em código antigo (ele faz isso para sua aplicação ter mais disponibilidade em diversos browsers) e transforma o JSX em funções React.

**JSX**

JSX (Javascript XML) é uma extensão da sintaxe do JavaScript que permite a criação de elementos de interface do usuário (idêntico ao HTML) em aplicações baseadas em JavaScript, principalmente em frameworks como o React. O JSX combina o poder do JavaScript com a simplicidade e a familiaridade do HTML, permitindo que os desenvolvedores descrevam a estrutura e a aparência dos componentes de uma forma mais intuitiva.

Entretanto o JSX tem algumas diferenças em relação ao HTML:

1. Todos os atributos vão estar em Camel Case (camelCase);
2. className no JSX é equivalente ao class no HTML;

O JSX só pode ser escrito dentro do return e só podemos passar um único elemento diretamente. Exemplo:

Texto

Descrição gerada automaticamente

Veja que só passamos um elemento diretamente: <div className=“App”> e todos os outros elementos estão dentro dessa div. Para conseguir passar mais de um elemento diretamente, deve-se usar o <React.Fragment>. Para simplificar seu uso, o React.Fragment pode ser usado somente usando: <> </>. Exemplo:

Texto

Descrição gerada automaticamente

Uma das funcionalidades mais poderosas do JSX e que torna o React tão poderoso é que podemos escrever código Javascript dentro do nosso JSX. Conseguiremos mesclar a escrita do front-end idêntico ao HTML junto com código Javascript. Para usar o código Javascript dentro do JSX usamos as chaves {}. Exemplo:

Texto

Descrição gerada automaticamente

**Eventos**

É uma maneira de dispararmos uma determinada funcionalidade no React. Faremos isso principalmente por meio dos atributos do JSX.

Para disparar um evento (por exemplo o onClick) em uma tag qualquer do JSX, é só passar o evento como atributo. Exemplo:

<span onClick={() => evento()}>Disparar Evento</span>

Já para disparar em um componente (por exemplo no Button) você deve primeiro passar o onClick como propriedade do Button e depois chamá-lo. Exemplo:

Dentro do componente:

function Button({name, active, onClick}){

    return(

        <div className='containerButton'>

            <button onClick={onClick} className={active ? 'button' : 'disableButton'}>{name}</button>

            <span>Seus dados estarão a salvo conosco!</span>

        </div>

    );

}

Dentro do App.js:

<Button name='Dispare o evento em um componente' active onClick={() => evento()}/>

Agora vamos ver sobre os eventos que não vão ser disparados pelo JSX, por exemplo o scroll. Para acessar esses eventos usamos o window.addEventListener(). Exemplo:

window.addEventListener('scroll', () => console.log('Evento disparado!'))

Para o addEventListener nós passamos dois parâmetros, o primeiro é o evento que queremos capturar (no caso o scroll) e o segundo é a função callback que queremos executar quando capturarmos o evento (no caso mostrar no console que o evento foi disparado).

**Estado e Reatividade**

O state, em React, é onde guardamos os dados do nosso componente. Quando trabalhamos com programação nós definimos estado o tempo todo. O estado de uma aplicação nada mais é que as informações armazenadas no nosso programa em um determinado tempo. Por exemplo, logo ao abrir um site, ele aparece uma bolinha girando, como se estivesse carregando. Esse seria o estado de carregamento. Após carregar, o site abre. Esse seria o estado do site carregado.

Um site pode ter inúmeros estados diferentes que controlam diferentes funcionalidades. O React nos oferece os Hooks que vão facilitar a manipulação e a interação com esses estados.

O primeiro React Hook que vamos ver é o useState.

* **useState()**

Para usar o React Hook useState(), criamos uma variável e atribuímos a ela a função useState() (para esse caso que queremos manipular os estados). O parâmetro a ser passado dentro do useState é o estado inicial que você deseja. Ao dar um console.log na variável que recebeu a atribuição, percebemos que a função useState retorna um Array com dois valores: o valor inicial e uma função que irá mudar esse valor inicial. Portanto, podemos desestruturá-la, criando um React Hook da seguinte forma:

function App() {

  const [carregando, setCarregando] = useState(true)

  return (

    <div>

      {carregando ? <span>Carregando...</span> : <Button name='Botão 1' active />}

      <button onClick={() => setCarregando(false)}>Carregar site</button>

    </div>

  );

}

Nesse exemplo, desestruturamos o retorno da função para a variável carregando receber o valor inicial e a variável setCarregando ser a função responsável por alterar o estado inicial. Dessa forma, no botão criado, passamos o evento onClick com o callback que ao identificar o clique vai chamar o setCarregando() e o parâmetro passado dentro do setCarregando vai ser o novo estado da variável carregando.

Outro exemplo do uso do useState é para um contador. Exemplo:

function App() {

  const [carregando, setCarregando] = useState(true)

  const [contador, setContador] = useState(0)

  return (

    <div>

      {carregando ?

      <span>Carregando...</span>

      :

      <div>

        <button onClick={() => setContador(contador+1)}>Adicionar</button>

        <span>{contador}</span>

      </div>

      }

      <button onClick={() => setCarregando(!carregando)}>{carregando ? 'Carregar site' : 'Voltar para carregamento'}</button>

    </div>

  );

}

Nesse caso usamos o setContador com o valor inicial atribuído a variável contador igual a 0. Criamos um botão com um evento onClick e um callback que ao identificar o click vai somar mais 1 ao contador. O span é responsável por mostrar o valor do contador na tela.

Agora que entendemos melhor o funcionamento dos estados e do React Hook, especificamente o Hook useState, podemos entender também a **Reatividade** do React. Notamos que sempre que mudamos o estado, toda nossa página reage (recarrega) e muda para algo que nós definimos.

Exemplo usando subtração e condição para parar de subtrair quando contador chegar no 0:

function App() {

  const [carregando, setCarregando] = useState(true)

  let [contador, setContador] = useState(0)

  if(contador<0){

    contador = 0;

  }

  return (

    <div>

      {carregando ?

      <span>Carregando...</span>

      :

      <div>

        <button onClick={() => setContador(contador + 1)}>Adicionar</button>

        <span>{contador}</span>

        <button onClick={() => setContador(contador - 1)}>

          Subtrair

        </button>

      </div>

      }

      <button onClick={() => setCarregando(!carregando)}>{carregando ? 'Carregar site' : 'Voltar para carregamento'}</button>

    </div>

  );

}

Nesse exemplo, criei outro botão, responsável pela subtração. Para limitar o contador para mostrar apenas números maiores ou igual a 0, criei uma condicional que sempre que o contador é menor que 0, atribui-se o valor de 0 para ele.

* **useEffect()**

Também é um dos Hooks mais utilizados dentro do React. Antes de estudarmos o useEffect, precisamos entender sobre o **Ciclo de Vida de um Componente** no React. O primeiro momento do ciclo de vida de um componente é quando ele é construído em tela. O segundo momento é quando ele é atualizado (por exemplo, criamos um componente e toda vez que mudamos seu estado ele é atualizado, e esse é o segundo momento do ciclo de vida do componente). O terceiro momento é quando esse componente é removido da tela (deixa de existir da interface).

O useEffect() existe justamente para nos ajudar a lidar com essas mudanças do ciclo de vida de um componente. Por exemplo, imagine que queremos implementar um código que só queremos que execute quando o componente é construído em tela. Dessa forma, usaríamos o useEffect. O useEffect vai ser usado principalmente em 3 casos: Quando eu tenho um código que só quero que seja executado quando o componente for criado em tela. Quando eu tenho um código que só quero que seja executado quando o componente for atualizado. Quando eu tenho um código que só quero que seja executado quando um componente for removido da tela.

Para usar o useEffect() fazemos:

useEffect(callback, array)

Note que passamos dois parâmetros para o useEffect, o primeiro sendo a função callback que queremos que seja executada em determinado momento. E o segundo parâmetro é um array de dependências, que serve para passar os estados de dependência, que vão servir para informar para o useEffect que tal estado foi alterado. Se você deixar o array vazio, o callback só será executado na renderização, ou seja, no primeiro momento de ciclo de vida do componente. Exemplo:

  useEffect(() => {

    console.log('Carregou pela primeira vez')

  }, [])

Nesse caso, criamos um Hook useEffect para que quando o componente for renderizado pela primeira vez em tela, a mensagem ‘Carregou pela primeira vez’ apareça no console.

Exemplo do useEffect para acionar toda vez que o componente for atualizado, ou seja, no segundo momento do ciclo de vida:

 useEffect(() => {

    console.log('Componente foi atualizado')

  }, [contador])

Nesse caso, passamos o contador no array de dependências, dessa forma, toda vez que o contador for atualizado (mudar o seu valor) a mensagem ‘Componente foi atualizado’ será impressa no console.

Por fim um exemplo de um código para ser executado somente quando um componente for removido da tela, ou seja, o terceiro momento do ciclo de vida:

useEffect(() => {

    return () => {

      console.log('Componente removido da tela')

    }

  }, [carregando])

Note que para isso, passamos um return dentro do callback e esse return ira retornar um outro callback com o código a ser executado quando o componente for removido. Nesse caso, quando o componente carregando for removido da tela ele ira imprimir no console ‘Componente removido da tela’.

* **useRef()**

O useRef serve basicamente para a gente fazer referencia e ter acesso a um elemento criado em tela e poder ter acesso a seus métodos e propriedades. O useRef só funciona quando os elementos já estiverem renderizados em tela, por isso ou usamos ele dentro de um useEffect ou em um callback. Exemplos:

Com useEffect:

const video = useRef();

useEffect(() => {

  video.current.play();

})

return (

<video ref={video}/>

)

Com callback:

const video = useRef();

function callBack(){

    video.current.play()

  }

return (

<button onClick={() => callBack()}>Rodar vídeo</button>

  <video ref={video}/>

)

* **useMemo()**

É um método que vai receber uma função que não vai ser recarregada com a atualização do componente e o seu valor de retorno vai ficar salvo na variável que o criou. É utilizado em casos muito específicos para ganho de performance da aplicação. Por exemplo, se você tiver um calculo muito grande que você não quer que seja recarregada toda vez que um componente for atualizado, você usa o useMemo para salvar esse cálculo na variável e não ser mais recarregado. Exemplo de useMemo:

function App() {

  const [carregando, setCarregando] = useState(true)

  const [contador, setContador] = useState(0)

  const calculation = useMemo(() => expensiveCalculation(contador), [contador]);

  return (

    <div>

      {carregando ?

      <span>Carregando...</span>

      :

      <div>

        <button className='adicionar' onClick={() => setContador(contador + 1)}>Adicionar</button>

        <span>{contador}</span>

        <span>{calculation}</span>

        <button className='subtrair' onClick={() => setContador(contador - 1)}>

          Subtrair

        </button>

      </div>

      }

      <button onClick={() => setCarregando(!carregando)}>{carregando ? 'Carregar site' : 'Voltar para carregamento'}</button>

    </div>

    );

}

const expensiveCalculation = (num) => {

  console.log("Calculating...");

  for (let i = 0; i < 1000000000; i++) {

    num += 1;

  }

  return num;

};

Nessa função nós temos o componente de carregamento e o componente de adicionar e subtrair. Se nós não usássemos o useMemo, toda vez que recarregássemos qualquer componente (seja ele de carregamento ou de adicionar ou subtrair) o cálculo grande (expensiveCalculation) seria recarregada e faria com que a página demorasse para ser recarregada. Porém, nós queremos que o cálculo só seja executado quando os componentes de adicionar ou subtrair forem acionados. Por isso usamos o useMemo, com o primeiro parâmetro sendo a função que executa o cálculo e o segundo parâmetro sendo o array de dependência o contador, ou seja, quando o contador (adicionar ou subtrair) for recarregado, ai sim a função expensiveCalculation é chamada de novo, caso contrário ela não interfere na aplicação. Dessa forma, ao usar o carregamento a função não será recarregada e o carregamento será executado rapidamente.

* **useCallback()**

Tem quase a mesma função do useMemo, a diferença é que no useMemo nós salvamos um valor e esse valor não será atualizado, pois a função que o retorna não será recarregado a menos que passemos uma dependência para que ela seja recarregada. Já no useCallback nós salvamos a função inteira, para que ela seja executada a partir de um evento. Exemplo:

 const callBack = useCallback(() => {

    console.log('Qualquer coisa')

  }, [])

Nesse caso, criamos uma variável callBack que armazenará a função passada no useCallback. Note que também passamos um array de dependências, porém nesse caso deixamos ele vazio, o que significa que a função será recarregada somente na primeira renderização da página. Caso passássemos alguma dependência, a função seria recarregada conforme a dependência fosse renderizada.

**Rotas**

Para navegar as rotas e navegar entre as páginas dentro do React, precisaremos usar uma biblioteca externa chamada react-router-dom. Para instalar uma biblioteca externa usaremos no terminal:

npm install <nome\_biblioteca>

Dessa biblioteca, importaremos o BrowserRouter, o Router e o Route. O BrowserRouter tem que envolver tudo da aplicação que vai utilizar as rotas. Normalmente usaremos para envolver a raiz da nossa aplicação, já que toda a aplicação usará rotas.

O Router envolve todas as nossas rotas e o Route serve para a gente criar a nossa rota.

PAREI NO ROUTER