Componentes e Props

1. Configurando seu ambiente React

React é uma biblioteca que utiliza o JSX mas ele precisa ser ***transpilado*** para JavaScript vanilla antes de chegar ao navegador. Normalmente, usamos o ***Babel*** para fazer isso. Podemos executar o Babel com uma ferramenta de build como o ***Webpack***, que ajuda a agrupar todos nossos arquivos de JavaScript, CSS, imagens e etc para projetos na web.

Para simplificar essas configurações iniciais, podemos usar a CLI do `create-react-app` para cuidar de toda essa configuração para nós! Essa ferramenta é incrivelmente útil para iniciar a criação de um aplicativo em React, uma vez que configura tudo de que precisamos sem nenhuma configuração manual.

Além de configurar seu ambiente de desenvolvimento para utilizar as funcionalidades mais recentes do JavaScript, ele fornece uma experiência de desenvolvimento agradável e otimiza o seu app para produção.

```
npm install create-react-app --global
create-react-app my-app
cd my-app
npm start
```

Comandos:

- `create-react-app my-app`, onde `my-app` será a pasta que a CLI irá criar. Em aula usamos `.` pois eu ja estava **dentro da pasta** que ficaria o nosso projeto.
- `cd my-app`, para entrarmos no diretório do novo app.
- `npm start`, inicia o servidor de desenvolvimento.
- `npm install`, instala as dependencias do `package.json`, caso necessário.

Docs:

- Create a New React App
- Create React App Github

3. Componentes de Função

Como vimos esses componentes são os mais simples de se criar. Podemos usar tanto a notação de `function` quanto de arrow function `() => {}` .

```
function App() {
  return <h1>Hello World!</h1>
}

const App = () => {
  return <h1>Hello World!</h1>
}
```

Mas para criar componentes devemos ter outra coisa em mente: **Organização**. Teremos um módulo para falar especificamente disso, mas por enquanto vamos nos a tentar a algumas coisas:

- Sempre nomear os componentes com **PascalCase**
- Exportar componentes com `export default`

Com isso vamos criar um padrão que perpetuará por todo nosso código. Lembrando que as maneiras **diferentes de exportação** mudam como o componente, função ou variável é **importada.**

```
// export default Component:
import Component from './Component'
// ou:
import ComponentXPTO from './Component'

// export Component:
import { Component } from './Component'
```

Props

Uma característica que Componentes possuem em similar com funções é que podemos enviar "argumentos” para ele também, a diferença esta em como esses "parâmetros" são enxergados no React. Damos o nome de `props` todo e qualquer dado que enviamos, o que parecem ser atributos, para o Componente.

```
<Component foo="bar" bar="foo" />
```

Mas muita atenção em duas coisas

1. Estamos comparando com funções apenas para você ter algo para se basear, na verdade as Props serão enviadas como um objeto para o Componente
2. Para exibir variáveis no JSX é necessário deixar-la entre chaves `{}`

```
const Component = (props) => {
  return (
    <div>
      <p>{props.foo}</p>
      <p>{props.bar}</p>
    </div>
  )
}

/* Irá renderizar o seguinte HTML:
<div>
  <p>bar</p>
  <p>foo</p>
</div>
*/
```

4. Componentes de Classe

É aqui que as coisas ficam mais complicadas, diferente de funções pudemos ver que Classes são bem mais complexas e isso acontece simplesmente porque elas precisam executar uma gama muito maior de funcionalidades que os componentes de função.

```
class Component extends React.Component {
  render() {
    return <h1>Hello World!</h1>
  }
}
```

O método `render` é o primeiro que vocês vão se familiarizar para mandar o React exibir algo na tela. A segunda coisa que temos que ver são as Props, que aqui não temos um parâmetro para usar no `render` mas na verdade utilizar `this.props` para acessar o objeto.

```
class Component extends React.Component {
  render() {
    return <h1>Hello {this.props.name}!</h1>
  }
}
```

Passamos rapidamente pelo `this` quando falamos de funções em objetos, aqui pudemos ver mais sobre como podemos acessar as propriedades de uma classe com essa variável.

Além disso, temos as grandes diferenças entre esses Componentes:

- Componentes de Classe possuem **métodos do ciclo de vida do componente**
- Componentes de Classe possuem **estados e como manipula-los**

5. Composição de Componentes

Agora sim estamos botando em prática a **organização** do nosso código. Resumidamente a composição de componentes se da em dois casos:

1. Poder chamar o mesmo componente várias vezes com valores diferentes e todos eles renderizarem o mesmo resultado visual.
2. Ter a possibilidade de extrair elementos HTML em componentes com o mesmo comportamento a fim de reutilizar código.

Como vimos em aula, não podemos ter medo ao ver um JSX se repetindo em diversos lugares ou até mesmo no mesmo componente e a partir disso criar um componente para ele e reutilizar esse código.

Componentes de Contexto

Para explicar em poucas palavras, são componentes bem específicos para um caso e que dificilmente vamos reutiliza-los. Mas então se não vamos reutilizar, para que criar um novo componente? **Por pura organização do nosso código!**

É muito importante mantermos os componentes organizados, eu entendo que agora isso não é um problema, mas tenho certeza que durante a sua carreira você vai encontrar diversos componentes GRANDES de verdade que se fossem construídos com essa organização com certeza estariam muito menores e mais simples de entender.

Componentes Utilitários

Esses componentes são os queridinhos, geralmente vão ter muitas Props para poder manipular em vários lugares os respectivos visuais que esperamos.

Componentes são como funções e por isso torna toda essa lógica de abstrair elementos JSX em outros componentes de forma fácil e rápida. Esses componentes

que são mais visuais podem ser um pouco mais complexos para componentizar pois em alguns lugares eles vão ter alguma cor especifica, em outros lugares um tamanho maior. E para driblar essa dificuldade podemos usar a Prop especial `children`.

```
const Text = (props) => {
  return <p className="text">{props.children}</p>
}

// Dessa forma podemos criar textos mais dinamicos:
<Text>
  <span className="red-and-big-text">Texto grande e vermelho</span>
</Text>
```