Estados e Ciclo de vida

1. Estados de um Componente

Acho que se em algum momento você quis mudar uma variável durante o módulo anterior você deve ter se frustado um pouco na primeira tentativa - hehe - mas não se preocupe, essa é uma das coisas que desenvolver causa e não é nada que com algum tempo e esforço você não consiga fazer.

Mas agora vamos falar de coisa boa ~~vamos falar de TekPix~~ vamos falar sobre essa tal maneira de alterar as variáveis e fazer o React **reagir** literalmente e alterar o elemento que vemos na tela, com ***estados***.

Podemos levar em consideração os estados da água para isso em um componente:

```
class Water extends React.Component {
  constructor() {
    super()

    this.state = {
      form: 'liquid'
    }
  }

  render() {
    return (
      <img src={this.state.form} />
    )
  }
}
```

Como sabemos, a água possui 3 estados: liquido, fisico e gasoso. Todos são *disparados* por algo que muda o comportamento dela, se ela estiver no estado liquido e a aquecermos ficará no estado gasoso, se esfriarmos se tornará gelo e etc. Em aula vimos que para alterar o estado de algo temos que usar o `this.setState` e com muita atenção pois esse método **recebe um objeto** que altera todos os campos que enviarmos no `state`. Podemos **imaginar** que o `setState` faz isso:

```
// esse método é meramente ilustrativo
setState: function(newState) {
  this.state = {
    ...this.state,
    ...newState
  }
```

```
  this.reloadComponent()
}
```

2. Elevação de Estados

Em alguns momentos quando estamos desenvolvendo entramos em momentos que precisamos fazer com que o ***componente filho*** altere o estado do ***componente pai***, e se isso não for feito com precisão pode dar alguns erros na aplicação, o nome disso é **Elevar o Estado**, ou melhor enviar para o componente filho a **responsabilidade** de alterar o estado do componente pai. Não é possível ~~na verdade é mas é completamente uma prática errada~~ fazer o contrario.

A elevação de estados é muito comum em Modais (pop-ups, janelas flutuantes e etc):

```
class Page extends React.Component {
  constructor() {
    super()

    this.state = {
      isModalOpen: false
    }
  }

  render() {
    return (
      <div>
        <button onClick={() => this.setState({ isModalOpen: true })}>
          Open Modal
        </button>
        <Modal
          onClose={() => this.setState({ isModalOpen: false })}
          isOpen={isModalOpen}
        />
      </div>
    )
  }
}
```

Isso é necessário porque o componente `Modal` espera que o componente pai envie a instrução (no caso o estado `isModalOpen`) como `true` e apenas ai ele abre o pop-up. Mas depois disso resta o comportamento de fechar que não pode ser **executado** pelo componente pai pois pode gerar uma confusão para o usuário de "como eu devo fechar isso?”, então o critério de como, onde e quando o pop-up fechará fica com o componente `Modal`.

3. Renderização condicional

Essa aqui teve spoiler desde a primeira aula. Basicamente temos diversas formas de renderizar (ou não) algum componente ou elemento JSX e isso não foge da criação de condicionais, *if inlines* ou exibições de variáveis no meio do JSX. Vamos la:

Pré-renderização

Seria uma condição para forçar o componente exibir algo antes da renderização natural dele, bons exemplos dessa prática são erros de APIs ou exibição de variantes de algum componente dependendo de Props especificas.

```
const ProductPage = ({ product, error }) => {
  if (!product || error) {
    return <ErrorPage />
  }

  return (
    <>
      <ProductImage />
      <ProductName />
      <ProductPrice />
    </>
  )
}
```

Essa forma acaba sendo melhor pois assim criamos novos componentes com a capacidade de decidir o que será renderizado.

Durante a renderização

Aqui temos casos em que temos que decidir no meio do JSX se algo vai aparecer ou não. Acaba não sendo um boas prática pois força nosso JSX ter lógicas mas se usado com consciência é uma ótima forma de manter contextos lógicos da nossa aplicação.

Bons exemplos de uso são erros e mensagens de validação de um formulário:

```
render() {
  return (
    <>
      <label for="cpf">CPF</label>
      <input
        type="text"
        id="cpf"
```

```
          onChange={(event) => this.setState({ cpf: event.target.value })}
        />
        {this.state.required && <p>This field is required</p>}
        {this.state.error && <p>This is not a CPF</p>}
      </>
    )
  }
```

4. User Experience

Referências

- https://aelaschool.com/experienciadousuario/ux-design-o-que-e-e-como-atuar-na-area/
- https://aelaschool.com/experienciadousuario/psicologia-nos-projetos-de-ux-design/
- https://aelaschool.com/experienciadousuario/storytelling-como-usar-em-ux/
- https://aelaschool.com/experienciadousuario/design-thinking-e-como-aplicar/

5. Ciclo de Vida do Componente

Falei, falei e falei e finalmente mostrei. Os famosos métodos de ciclo de vida, e como um ciclo vimos que eles começam em certo ponto e sempre voltam para o `render`, esse também sendo um dos métodos.

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`
- `shouldComponentUpdate()`
- `render()`
- `componentDidUpdate()`

Cada um desses métodos foi explicado em aula então deixo aqui a documentação da implementação do `React.Component` essa documentação explica tim-tim por tim-tim como cada um desses métodos funciona, inclusive as propriedades do **State** e das **Props**, então depois que tiver dominado o uso deles de uma olhada nessa

documentação para entender um pouco mais como as coisas funcionam por debaixo dos panos.

https://pt-br.reactjs.org/docs/react-component.html