Introdução ao ReactJS

Aplicações para a Internet I

Steven Abrantes, Tiago Rebelo e Frederico Fonseca

state

Estado (state)

- O **estado** (*state*) é como o *props*, mas é privado e totalmente controlado pelo componente;
  - Só pode ser usado em componentes de classe;
- Valores internos do componente
- deve servir para guardar valores/estados da aplicação que mudam com o uso da mesma, para guardar uma alteração de estado que pode ter efeito na renderização do próprio componente, ser passado a um componente filho como o prop, ou que deve ser passado a algo externo à aplicação.

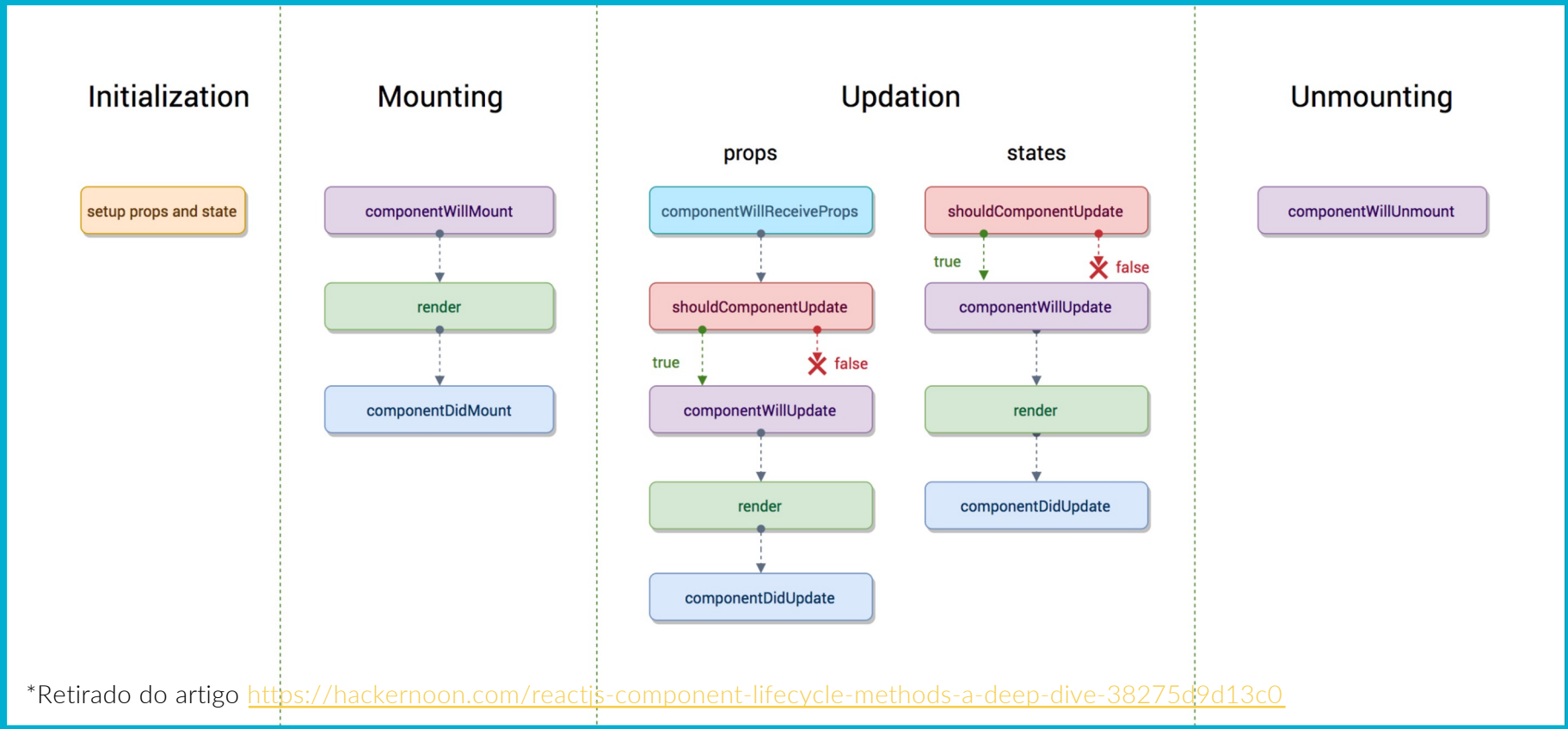
Estado (state)

- Os componentes filhos não são responsáveis por manter os *state*;
  - Por exemplo, quando enviamos um *state* de um componente pai para um componente filho, o responsável por gerir o estado é o pai através de funções/eventos criados para o efeito;

Lifecycle

dos componentes...

Component Lifecycle



*Retirado do artigo https://hackernoon.com/reactjs-component-lifecycle-methods-a-deep-dive-38275d9d13c0

Fases de um componente react

- **Initialization** (inicialização) - Inicia os *states* e *props* (se houverem);
- **Mounting** (criação) - cria o componente no DOM;
  - Métodos disponíveis:
    - **componentWillMount()** - é executado antes do componente ser criado no DOM;
    - **render()** - cria o componente no DOM;
    - **componentDidMount()** - é executado depois do componente ter sido criado no DOM;

Fases de um componente react

- **Update** (atualização) - atualiza o componente no DOM;
  - Este processo pode ser feito de 1 de 2 maneiras possíveis:
    - 1) através do envio de novas *props*;
    - 2) alteração do *state*;
  - Métodos disponíveis:
    - **shouldComponentUpdate()** - define através de true ou false, se o componente deve ser atualizado;
    - **componentWillUpdate()** - é executado somente quando o método anterior devolver true;

Fases de um componente react

- **Update** (atualização)
  - Métodos disponíveis (continuação):
    - **render()** - "atualiza" (cria novamente) o componente no DOM;
    - **componentDidUpdate()** - é executado quando o componente foi atualizado no DOM;
  - Método disponível apenas na atualização por ***props***:
    - **componentWillReceiveProps()** - é executado quando as props mudaram e não é a 1ª vez que são processadas;

Fases de um componente react

- **Unmounting** (destruição) - apaga o componente do DOM;
  - Métodos disponíveis:
    - **componentWillUnmount()** - é executado imediatamente antes do componente ser destruído. É o último método a ser executado no ciclo de vida do componente;

Component Lifecycle

- Vejamos um :

JS

```
'use strict';

class Contador extends React.Component {
    constructor(props) {
        super(props);
        this.state = { elapsed: 0 };
        this.tick = this.tick.bind(this);
    }

    componentDidMount() { this.timer = setInterval(this.tick, 1000); } //evento 1s
    componentWillUnmount() { clearInterval(this.timer); }
    tick() { this.setState({ elapsed: this.state.elapsed+1 }); }
    render() { return React.createElement("h1",null, this.state.elapsed+" segundo(s)");
}
};

ReactDOM.render(React.createElement(Contador), document.body);
```

O output seria:

0 segundo(s)
1 segundo(s)
2 segundo(s)
(...)

Events

Manipulação de Eventos (*events*)

- A manipulação de eventos em *react* é muito semelhante ao JS;
  - Apenas existe algumas diferenças de sintaxe:
    - Os eventos em *react* utilizam a notação *camelCase*;

```
React.createElement("button", { onClick: funcaoEvento }, "Exemplo");
```
JS

    - Não podemos retornar *false* para impedir o comportamento por defeito no *react*;
      - Devemos invocar o *preventDefault()*;

Manipulação de Eventos (*events*)

- Analisemos um exemplo:

```
<a href="#" onclick="console.log('Clicou no botão.');
return false">Clique aqui</a>
```

HTML

Esta seria a solução típica em HTML, mas que não é válida em *react*.

```
function ActionLink() {
    function handleClick(e) {
        e.preventDefault();
        console.log('Clicou no botão...');
    }

    return React.createElement("a", { href: "#", onClick:
        handleClick }, "Clique aqui");
}
```

JS

Em *react* é necessário invocar a função **preventDefault()**.

Manipulação de Eventos (*events*)

- No *react* não é necessário definir o *addEventListener* porque normalmente definimos os eventos na definição do componente;

Manipulação de Eventos (*events*)

```
use ‘strict’;

class StateButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { estado: false };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() { this.setState({ estado: !this.state.estado }); }

  render() {
    let status;
    if (this.state.estado) { status = ‘Estado atual: true’; } else { status = ‘Estado atual: false’; }

    return React.createElement(‘div’, null,
      React.createElement(‘h1’, null, status),
      React.createElement(‘button’, { onClick:this.handleClick }, ‘Muda de Estado’)
    );
  }
}
ReactDOM.render(React.createElement(StateButton), document.body);
```

JS

Manipulação de Eventos (*events*)

- Também podemos passar argumentos para um manipulador de eventos;
- Analisemos um exemplo:

```
this.handleClick = this.handleClick.bind(this, id); //bind

onClick={ (e) => this. handleClick(id, e) } //ES6
```

Renderização Condicional

Renderização Condicional

- Em *react* também é possível utilizar estruturas condicionais (if, switch), operadores lógicos (&& e ||), entre outros;
  - A chamada '**renderização condicional**';
- Um componente decide com base em uma ou mais condições, quais os elementos a retornar;
  - Por exemplo, um componente pode retornar uma lista de alunos ou uma mensagem a dizer "Desculpe, mas a lista está vazia.";
- Em suma, quando um componente tem renderização condicional, a instância do componente a renderizar pode ter aparências diferentes;

Renderização Condicional

- Analisemos um exemplo:

JS

```
function List(props) {
  if (!props.list) { return null; }

  if (!props.list.length) {
     return React.createElement("p", null, " Desculpe, a lista está vazia.");
  } else {
     return React.createElement("ul", null,
        props.list.map((item) => React.createElement("li", null, item))
     );
  }
}

var lista = [1,2,3]; //experimente apagar os valores da lista
ReactDOM.render(React.createElement(List, {list:lista}), document.body);
```

Listas

...

Listas

- Em JS a função *map()* é utilizada para converter um *array* numa lista;

JS

```
var numeros = [1, 2, 3, 4, 5];
var listaNumeros = numeros.map( (numero) => numero );
```

- Em react é possível tirar partido das listas para criar conjuntos de elementos;
- Por exemplo, de uma lista de itens:

JS

```
var numeros = [1, 2, 3];
ReactDOM.render(React.createElement("ul", null,
    numeros.map( (numero) => React.createElement("li", null, numero) )
), document.body);
```

O output seria:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

Listas

- É possível criar um componente que tire partido das listas:

JS

```
function ListaValores(props) {
    var valores = props.dados;
    var listaItens = valores.map(
                                function(valor) {
                                    return React.createElement("li", null, valor);
                                }
                               );
    return React.createElement("ul", null, listaItens);
}

var valores = [1, 2, 3];
ReactDOM.render(React.createElement(ListaValores, { dados: valores }), document.body);
```

O output seria:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

Formulários

...

Formulários

- Em react este valor está armazenado na propriedade *state* dos componentes, sendo atualizado pelo evento *setState()*;

```
class ExampleForm extends React.Component
{
    constructor(props) {
        super(props);
        this.state = { value:'' };
        this.handleChange = this.handleChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {
        this.setState( {value: event.target.value} );
    }

    handleSubmit(event) {
        alert('A idade inserida é : '+ this.state.value);
        event.preventDefault();
    }
```

Formulários

- Em react este valor está armazenado na propriedade *state* dos componentes, sendo atualizado pelo evento *setState()*;

```
    handleSubmit(event) {
        alert('A idade inserida é : '+ this.state.value);
        event.preventDefault();
    }

    render() {
        return React.createElement("form", { onSubmit: this.handleSubmit },
            React.createElement("div", { className: "form-group"},
            React.createElement("label", null, "Idade: "),
            React.createElement("input", { className: "form-control", type: "number", value: this.state.value, onChange:
this.handleChange})),
            React.createElement("input", { className: "btn btn-primary", type: "submit", value: "Enviar"})
         );
    }
}
```

Composição

Composição

- No desenvolvimento de uma aplicação é comum termos componentes que são “casos especiais” de outras componentes já existentes;
  - Por exemplo, o botão de limpar dados (*ResetButton*), pode ser visto como um caso especial de botão (*Button*), ou a janela de login (*LoginDialog*) como um caso especial de uma janela (*Dialog*), entre outros;
- Tendo por base os conceitos de **Programação Orientada a Objetos** (POO), em *react* dá-se preferência à **composição** em detrimento de **herança**;

Composição

- Em suma, um componente renderizar um outro mais genérico, incluindo os seus *props* (parâmetros de entrada);
- Vejamos um exemplo:

```
function Dialog(props) {
    return React.createElement("div", {className: props.class}, props.children);
}

function WelcomeDialog() {
    return React.createElement(Dialog, {class: "jumbotron"},
        React.createElement("h1", {className: "display-4"}, "Olá!"), //children1
        React.createElement("p", {className: "lead"}, "Bem vindo às aulas de ReactJS!") //children2
    );
}

ReactDOM.render(WelcomeDialog(), document.querySelector("#react_container"));
```