FACULDADE DE CIÊNCIAS E TECNOLOGIA – UNIVERSIDADE NOVA DE LISBOA

Programa de Introdução à Investigação Científica em Engenharia Informática

Tutorial React.js

Sofia Seixas, 41970

1. Apresentação

O React é uma biblioteca de JavaScript que serve para construir interfaces do utilizador, mais concretamente, aplicações single-page. Foi criado pelo facebook, tendo sido utilizado pela primeira vez no feed de notícias, em 2011.

As aplicações single-page são aplicações web que utilizam uma única página. Neste tipo de aplicação, todo o código necessário para o seu funcionamento (HTML, CSS, JavaScript) é carregado duma vez, sendo os recursos necessários carregados dinamicamente e adicionados à página conforme necessário.

Neste tutorial vamos construir uma pequena aplicação single-page utilizando a biblioteca do React. A aplicação vai consistir numa lista de compras, com funcionalidades para adicionar, editar e remover itens.

2. Instalação

Para efeitos deste tutorial vamos utilizar uma CDN, uma vez que é a maneira mais simples e rápida de instalar o React.

Vamos então criar um ficheiro index.html e incluir os ficheiros do React:

<script src="https://unpkg.com/react@15/dist/react.min.js"></script>

<script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js"></script>

Para tornar o desenvolvimento mais fácil e rápido, vamos utilizar o Babel, um compilador de JavaScript que nos permite utilizar ES6 e JSX no nosso código. O ES6 é um conjunto de funcionalidades de JavaScript que tornam o desenvolvimento mais fácil, e o JSX é uma extensão do Javascript que trabalha bem em conjunto com o React, e que vamos utilizar ao longo de todo este tutorial.

É importante realçar que toda a aplicação poderia ser feita com puro JavaScript, e que estas ferramentas apenas

servem para simplificar e acelerar o processo.

Para utilizar o Babel, vamos incluir o ficheiro:

<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.23.1/babel.min.js"></script>

E estamos prontos para começar a escrever código. O ficheiro index.html deverá ser semelhante a este:

```
<!DOCTYPE html>
<html>
<head>
    <title>Tutorial React</title>
    <script src="https://unpkg.com/react@15/dist/react.min.js"></script>
    <script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.23.1/babel.min.js"></script>
</head>
<body>

</body>
</html>
```

3. JSX

Quando inserimos uma tag *<script>*, o browser está à espera de encontrar JavaSript lá dentro. No entanto, nós vamos escrever código JSX e não JavaScript, pelo que se o inserirmos simplesmente dentro de uma tag *<script>*, o browser não vai perceber o que está lá dentro, e vai devolver um erro.

É aqui que entra o Babel, cuja função é "traduzir" o código JSX para JavaScript, para que o browser o saiba interpretar. Mas para que o Babel saiba que tem código para traduzir, temos que indicá-lo através de um atributo *type="text/babel"* nas tags *<script>* onde vamos escrever JSX, ou seja:

```
<script type="text/babel">

</script>
```

4. Componentes

Uma aplicação React é construída com vários componentes (components), uma das peças base do React. Um componente é, basicamente, uma parte de um website, que pode ser reutilizado ou combinado com outros componentes para fazer um maior. No contexto da nossa aplicação, o componente principal vai representar um item da lista de compras. O código do componente é o seguinte:

```
<div id="app"></div>

<script type="text/babel">

    var Item = React.createClass({
        render: function() {
            return (
                <div>
                    <hr/>
                    <p>Nome do Item</p>
                    <button>Editar</button>
                    <button>Remover</button>
                    <hr/>
                </div>
            );
        }
    });

    ReactDOM.render(<Item />, document.getElementById('app'));

</script>
```

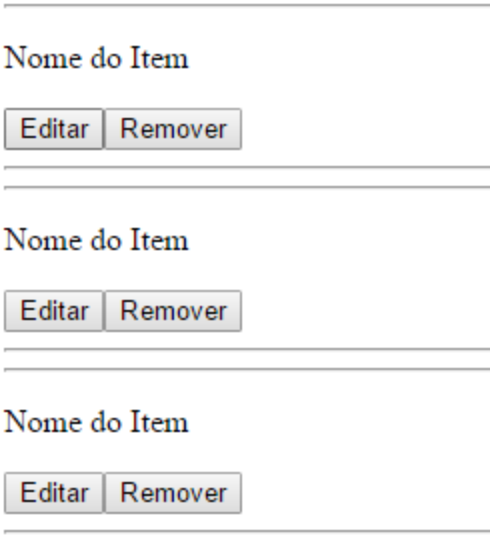
Em primeiro lugar criamos uma classe do React e guardamo-la numa variável (deve começar por letra maiúscula).

De seguida, implementamos a função render, cuja função é retornar um pedaço de HTML. Para o nosso item, queremos apresentar o nome do item, um botão para editar, e outro para remover. Visto que estamos a usar JSX, podemos escrever o HTML diretamente dentro da função, desde que todo o HTML esteja contido numa única tag (daí a tag * * que encapsula todos os elementos).

Com isto, temos um componente funcional e pronto para ser apresentado. Para isso usamos a função render do ReactDOM, e passamos o componente (como se fosse uma tag), bem como um container onde o componente irá ser renderizado. De notar que é possível renderizar vários componentes desta maneira, desde que os coloquemos dentro de uma única tag, por exemplo:

```
ReactDOM.render(<div><Item /><Item /><Item /></div>, document.getElementById('app'));
```

A nossa aplicação deverá ter, então, o seguinte aspecto:



5. Propriedades

Neste momento, temos uma lista de 3 itens, cada um com um botão de editar e remover, ainda sem qualquer lógica associada. O próximo passo será renderizar o nosso componente *Item* com um nome diferente. Para isso vamos utilizar uma propriedade (prop). Podemos passar o nome de duas maneiras:

```
ReactDOM.render(<Item text="Batatas"/>, document.getElementById('app'));
```

```
ReactDOM.render(<Item>Batatas</Item>, document.getElementById('app'));
```

No primeiro caso estamos a passar o nome através duma propriedade chamada text, à qual acedemos fazendo {this.props.text}.

No segundo passamos o nome dentro da tag do componente, ao qual acedemos através da prop especial {this.props.children}.

Componente utilizando uma prop “text”:

```
var Item = React.createClass({
    render: function() {
        return (
            <div>
                <hr/>
                <p>{this.props.text}</p>
                <button>Editar</button>
                <button>Remover</button>
                <hr/>
            </div>
        );
    }
});

ReactDOM.render(<Item text="Batatas" />, document.getElementById('app'));
```

6. Eventos

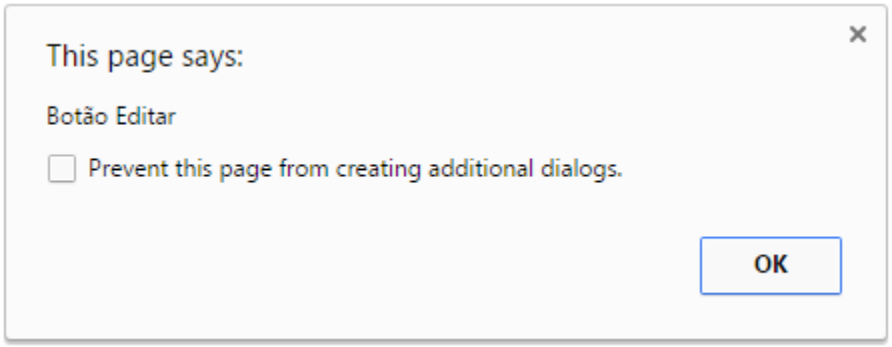
O nosso componente Item tem apenas definida a função render, mas é possível definir as nossas próprias funções dentro de um componente, e utilizá-las sempre que nos for conveniente. Vamos definir duas novas funções (*edit* e *remove*) que servirão para tratar da lógica dos nossos botões. Para começar, vamos preocupar-nos apenas em definir e chamar as funções:

```
var Item = React.createClass({
    edit: function() {
        alert("Botão Editar");
    },
    remove: function() {
        alert("Botão Remover");
    },
    render: function() {
        return (
            <div>
                <hr/>
                <p>{this.props.text}</p>
                <button onClick={this.edit}>Editar</button>
                <button onClick={this.remove}>Remover</button>
                <hr/>
            </div>
        );
    }
});
```

Definimos as nossas funções exatamente da mesma maneira que definimos a função *render*, neste momento colocamos apenas um alerta lá dentro, para verificar se está a ser chamada corretamente.

Chamar a função também é muito simples, basta utilizar um evento (neste caso utilizei um *onClick* nos botões) e indicar qual a função que pretendemos chamar, por exemplo *onClick={this.edit}*.

Se tudo estiver correto devemos agora receber um alerta ao clicar num botão:



7. Estados

Os estados (states) são semelhantes às props, mas são privados e controlados completamente pelos componentes. Enquanto que uma prop é "read-only", o estado de um componente pode ser alterado ao longo do seu ciclo de vida.

O nosso componente Item tem um botão de editar, pelo que precisamos que o nome do item passe a ser editável e que possamos carregar num botão para gravar um novo nome. Isto sugere que o nosso componente pode estar em dois modos: um de visualização e um de edição. Para implementar este conceito vamos utilizar o estado do componente.

A primeira coisa a fazer é implementar a função *getInitialState*, que retorna o estado inicial do componente:

```
var Item = React.createClass({
    getInitialState: function() {
        return {editing: false};
    },
    (...)
```

Assumimos como modo de visualização o estado *editing: false* e como modo de edição *editing: true*.

Podemos então alterar a nossa função *edit* para passar a alterar o estado:

```
edit: function() {
    this.setState({editing: true});
},
```

Agora que o nosso botão edit altera corretamente o estado do componente, podemos utilizar o estado para visualizar o modo correspondente. Para isto vamos criar duas novas funções, *renderNormal* e *renderForm*, e alterar a função *render* para chamar a respetiva função dependendo do estado:

```
save: function() {
    this.setState({editing: false});
},
renderNormal: function() {
    return (
        <div>
            <hr/>
            <p>{this.props.text}</p>
            <button onClick={this.edit}>Editar</button>
            <button onClick={this.remove}>Remover</button>
            <hr/>
        </div>
    );
},
renderForm: function() {
    return (
        <div>
            <hr/>
            <textarea defaultValue={this.props.text}></textarea>
            <br/>
            <button onClick={this.save}>Guardar</button>
            <hr/>
        </div>
    );
},
render: function() {
    if(this.state.editing){
        return this.renderForm();
    }
    else {
        return this.renderNormal();
    }
}
```

Se o estado tiver editing a *false* renderizamos o mesmo que tinhamos anteriormente. Se o estado tiver editing a *true*, renderizamos uma *textarea* com um botão para gravar. Foi também implementada uma função *save* que vai ser chamada ao clicar no botão gravar, cuja função é voltar a colocar o estado a *editing: false*.

O nosso componente em modo de edição deverá ter o seguinte aspeto:

A lógica de guardar o novo valor ainda não foi feita, mas voltaremos a esse ponto mais adiante.

8. Componentes filho

Neste momento a nossa aplicação tem um problema: temos um componente Item, que podemos renderizar as vezes que quisermos, mas cada um dos nossos componentes está “desligado” dos outros, ou seja, não tem conhecimento da sua existência. Precisamos duma maneira de gerir os nossos componentes, para possibilitar funcionalidades como reordenar os nossos items, adicionar um novo item ou remover um existente. Para isso vamos criar um componente ShoppingList, que vai conter todos os componentes Item:

```
var ShoppingList = React.createClass({
    getInitialState: function() {
        return {
            items: ["Batatas", "Cebolas", "Cenouras"]
        }
    },
    eachItem: function(item, i) {
        return (<Item text={item} index={i} />);
    },
    render: function() {
        return (
            <div>
                <h2>Lista de Compras</h2>
                <div>
                    {this.state.items.map(this.eachItem)}
                </div>
            </div>
        );
    }
});

ReactDOM.render(<ShoppingList />, document.getElementById('app'));
```

O estado da lista contém uma array de strings que representam os nomes de cada item, e contém inicialmente 3 strings (batatas, cebolas e cenouras). Foi criada uma função eachItem, que recebe um nome de um item e um índice, e retorna um novo componente Item com as props correspondentes (o índice servirá para identificar um componente Item dentro do componente ShoppingList). Por fim foi definida a função render, que renderiza um header “Lista de Compras”, bem como todos os items, com nomes iguais aos especificados no estado. A nossa aplicação deverá ter o seguinte aspecto:

Agora que temos um componente “pai” que contém todos os componentes Item, é facil perceber que as funcionalidades de adicionar, editar e remover um Item implicam a alteração da array *items*.

Vamos começar por implementar as funções para editar e remover:

```
var ShoppingList = React.createClass({
    getInitialState: function() { ··· },
    removeItem: function(i) {
        var tmp = this.state.items;
        tmp.splice(i, 1);
        this.setState({items: tmp});
    },
    updateItem: function(newText, i) {
        var tmp = this.state.items;
        tmp[i] = newText;
        this.setState({items: tmp});
    },
    eachItem: function(item, i) { ··· },
    render: function() { ··· }
});
```

A função removeItem recebe um índice i, remove a entrada correspondente da array de itens, e atualiza o estado. A função updateItem recebe o novo nome e um índice, altera o nome da entrada correspondente da array de items, e atualiza o estado. A ideia é que, ao carregar num botão de editar ou remover, sejam chamadas as funções da ShoppingList com os parâmetros apropriados.

Precisamos então de uma maneira de aceder a estas funções a partir de cada um dos componentes Item. Para tal, basta apenas que passemos as funções através de props, da seguinte maneira:

```
var ShoppingList = React.createClass({
    getInitialState: function() { ··· },
    removeItem: function(i) { ··· },
    updateItem: function(newText, i) { ··· },
    eachItem: function(item, i) {
        return (<Item text={item} index={i} update={this.updateItem} remove={this.removeItem} />);
    },
    render: function() { ··· }
});
```

Passámos a função updateItem na prop “update” e a função removeItem na prop “remove”. Podemos então atualizar as funções do componente Item, para que passem a chamar as funções da ShoppingList:

```
var Item = React.createClass({
    getInitialState: function() { ··· },
    edit: function() { ··· },
    remove: function() {
        this.props.remove(this.props.index);
    },
    save: function() {
        this.props.update("Novo Nome", this.props.index);
        this.setState({editing: false});
    },
    renderNormal: function() { ··· },
    renderForm: function() { ··· },
    render: function() { ··· }
});
```

E temos então os botões de editar e remover funcionais, restando-nos apenas inserir o texto que escrevemos na área de edição, em vez de inserirmos a string “Novo Nome”.

9. Referências

Neste momento já temos o botão editar a mostrar a área de edição, e o botão gravar a retornar ao modo de visualização, atualizando o nome do Item para “Novo Nome”. Mas ainda não tratámos da lógica de, efectivamente, guardar o novo nome do item. Mas como vamos aceder ao novo nome? Não podemos utilizar um id, pois podem existir múltiplos itens, por isso vamos utilizar uma referência (ref), um atributo especial do React que podemos associar a cada componente:

```
var Item = React.createClass({
    getInitialState: function() { ··· },
    edit: function() { ··· },
    remove: function() { ··· },
    save: function() {
        this.props.update(this.refs.newText.value, this.props.index);
        this.setState({editing: false});
    },
    renderNormal: function() { ··· },
    renderForm: function() {
        return (
            <div>
                <hr/>
                <textarea ref="newText" defaultValue={this.props.text}></textarea>
                <br/>
                <button onClick={this.save}>Guardar</button>
                <hr/>
            </div>
        );
    },
    render: function() { ··· }
});
```

Em primeiro lugar, foi adicionada a ref "newText" na textarea do modo de edição. De seguida, foi alterada a função save para passar a utilizar o valor dessa referência, ao qual se acede a partir de *this.refs.newText.value*.

10. Criar Componentes

Já implementámos as funcionalidades de editar e remover um Item, resta-nos então implementar um botão que adiciona um novo Item. Vamos então criar a função addItem, que recebe o nome do item, adiciona esse nome à array de nomes, e altera o estado:

```
var ShoppingList = React.createClass({
    getInitialState: function() { ··· },
    addItem: function(text) {
        var tmp = this.state.items;
        tmp.push(text);
        this.setState({items: tmp});
    },
    removeItem: function(i) { ··· },
    updateItem: function(newText, i) { ··· },
    eachItem: function(item, i) { ··· },
    render: function() { ··· }
});
```

E vamos alterar a função render, adicionando o botao Adicionar que chama a função addItem:

```
var ShoppingList = React.createClass({
    getInitialState: function() { ··· },
    addItem: function(text) { ··· },
    removeItem: function(i) { ··· },
    updateItem: function(newText, i) { ··· },
    eachItem: function(item, i) { ··· },
    render: function() {
        return (
            <div>
                <h2>Lista de Compras</h2>
                <button onClick={this.addItem.bind(null, 'Novo Item')}>Adicionar</button>
                <div>
                    {this.state.items.map(this.eachItem)}
                </div>
            </div>
        );
    }
});
```

E podemos então adicionar itens à nossa lista. O aspecto final da aplicação será, após adicionar um novo item:

11. Desafio

Ao construirmos esta aplicação ficámos a conhecer as principais funcionalidades do React. Para avaliarem os conhecimentos adquiridos, proponho que implementem uma funcionalidade para reordenar os Itens (por exemplo, um botão em cada item para mover para cima, e outro para mover para baixo).

Um exemplo duma solução do desafio, bem como o código da aplicação, pode ser consultada em:

https://github.com/sofiaseixas/PIICEIReact