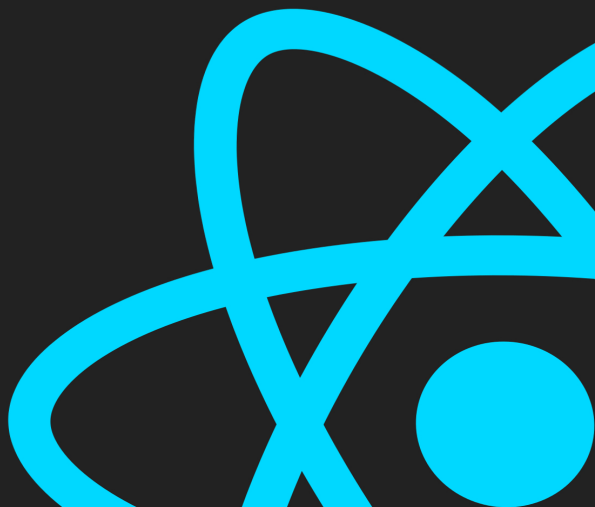
React

Guia do Iniciante



Daniel Schmitz

React - Guia do Iniciate

Domine a biblioteca javascript utilizada pelo Facebook e Instagram

Daniel Schmitz e Daniel Pedrinha Georgii

Esse livro está à venda em http://leanpub.com/react-guia-do-iniciate

Essa versão foi publicada em 2015-11-03

Outras Obras Desses Autores

Livros De Daniel Schmitz

AngularJS na prática (PT-BR)

Dominando Slim Framework (PT-BR)

Laravel e AngularJS (PT-BR)

Laravel and AngularJS

Livros De Daniel Pedrinha Georgii

Laravel and AngularJS

Laravel e AngularJS (PT-BR)

Conteúdo

Capítulo 1 - Introdução

Nesta obra estaremos abordando o React, uma biblioteca criada pela equipe do *Facebook* que tem como principal propósito implementar a camada de visualização de uma aplicação web através da criação de componentes, beneficiando-se de um uso muito bem elaborado da DOM, o que garante uma boa performance em relação às demais bibliotecas.

Esta é uma obra introdutória ao React, e pensamos em todos os detalhes possíveis para que o leitor possa a cada parágrafo ter uma progressão no entendimento do assunto. Nosso foco não é criar uma grande aplicação ou exibir detalhes muito técnicos sobre a biblioteca, mas sim exibir de forma fácil de entender os conceitos principais do React e em como podemos nos beneficiar dele.

Node e npm

Possivelmente você conhece Node, e npm também. Essas ferramentas já fazem parte do programador javascript, e nós acreditamos que não será necessário escrever sobre elas e seus detalhas.

Mas caso não conheça estas tecnologias, fique tranquilo. Elas não são essenciais para o seu entendimento em React, mas são muito importantes para qualquer tarefa no desenvolvimento web. Acesse este link[1] para conhecer mais sobre o npm.

Instalação do React

Como em toda biblioteca javascript, existem três formas distintas de instalação, cada um com seus prós e contras.

[1] http://www.sitepoint.com/beginners-guide-node-package-manager/

Via cdn

A primeira e mais fácil é utilizar arquivos CDN (Content Delivery Network) e criar um html básico importando as bibliotecas diretamente da Internet, conforme o exemplo a seguir:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.j\
s"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
  </head>
  <body>

  </body>
</html>
```

Neste arquivo adicionamos duas bibliotecas, sendo que a primeira é relativo ao `react`, e a segunda é o `babel-core`, que é um pré compilador para a linguagem JSX, amplamente usada nesta obra.

Via download

Também pode-se acessar o site https://facebook.github.io/react/[2] e clicar no botão "Download React", realizar o download da versão mais atual e referenciá-la no projeto, como no exemplo a seguir:

[2] https://facebook.github.io/react/

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <script src="js/react/build/react.js"></script>
    <script src="js/react/build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
  </head>
  <body>

  </body>
</html>
```

Via npm

Pode-se instalar as bibliotecas do React e Babel pelo `npm`, que é o gerenciador de pacotes do `node`. Recomenda-se inicialmente executar o seguinte comando na pasta onde o seu projeto será criado:

```
$ npm init
```

Através deste comando, o arquivo `package.json` será criado, incluindo algumas informações sobre o projeto. Após a criação do `package.json`, instalamos o react através do seguinte comando:

```
$ npm install react react-dom --save
```

A diretiva `--save` irá adicionar estes pacotes no arquivo `package.json` e todas as bibliotecas instaladas estarão no diretório `node_modules`. O arquivo `index.html` ficaria semelhante ao código a seguir:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  <script src="node_modules/react/dist/react.js"></script>
  <script src="node_modules/react-dom/dist/react-dom.js"></script>
    <!-- CDNs -->
     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/\
3.3.5/css/bootstrap.min.css">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
  </head>
  <body>

  </body>
</html>
```

Perceba que ainda precisamos do *babel* no formato cdn, e deixaremos desta forma para facilitar o aprendizado do React. Em sistemas reais, o babel não é adicionado no documento html, e sim utilizado para compilar o código JSX em Javascript através da linha de comando.

Também adicionamos o *bootstrap* no formato CDN para estilizar alguns componentes que iremos apresentar nesta obra.

Como versão final do documento html, incluímos o arquivo `main.js`, juntamente com uma `<div>` cujo id será `main`, da seguinte forma:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  <script src="node_modules/react/dist/react.js"></script>
  <script src="node_modules/react-dom/dist/react-dom.js"></script>
    <!-- CDNs -->
     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/\
3.3.5/css/bootstrap.min.css">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
    <!-- Main js File -->
   <script type="text/babel" src="main.js"></script>
  </head>
  <body>
      <div id="main" class="container"></div>
  </body>
</html>
```

É importante informar corretamente o tipo de arquivo javascript que será inserido pelo elemento `<script>`, neste caso temos `text/babel` conforme o detalhe a seguir:

```
 <script type="text/babel" src="main.js"></script>
```

Isto é, usamos `text/babel` ao invés de `text/javascript`, para que o babel possa compilar o JSX em tempo de execução e retornar o javascript ao navegador.

Nos próximos exemplos exibiremos somente o arquivo `main.js`, mas para testá-los é necessário abrir o arquivo index.html no navegador.

Hello World

Como primeiro exemplo criaremos o `Hello World`, já exibindo alguns detalhes do React. Temos então o seguinte código:

```
ReactDOM.render(
        <h1>Hello world!</h1>,
        document.getElementById('main')
      );
```

Ver no jsfiddle[3]

Neste exemplo, temos o uso do `ReactDOM` com o método `render` que irá renderizar (desenhar) um componente em um determinado elemento da página. O primeiro parâmetro deste método é justamente o componente, neste caso usamos o componente `h1` para criar o texto `Hello World`. O segundo parâmetro é o elemento onde o componente será desenhado, ou seja, o div cujo id é `main`.

Para conhecermos um pouco mais do React vamos realizar algumas mudanças no `Hello World` e criar um componente chamado `<HelloWorld>`. Este componente terá um atributo chamado `name` que será informado no render do `ReactDOM`, veja:

```
var HelloWorld = React.createClass({
  render: function(){
    return (
      <p>Hello World, <span className='label label-primary'>{this.props.name\
}</span></p>
    );
  }
});

ReactDOM.render(
  <HelloWorld name="Daniel"/>,
      document.getElementById('main')
  )
```

Neste código criamos uma classe cujo nome é indicado pelo nome da variável `HelloWorld`. Por convenção todas as classes do React começam com letra maiúscula e todas as classes que representam elementos do html começam com letra minúscula. O método `createClass` do React irá criar uma classe que pode possuir métodos

[3] https://jsfiddle.net/danielschmitz/7pvLdbcd/

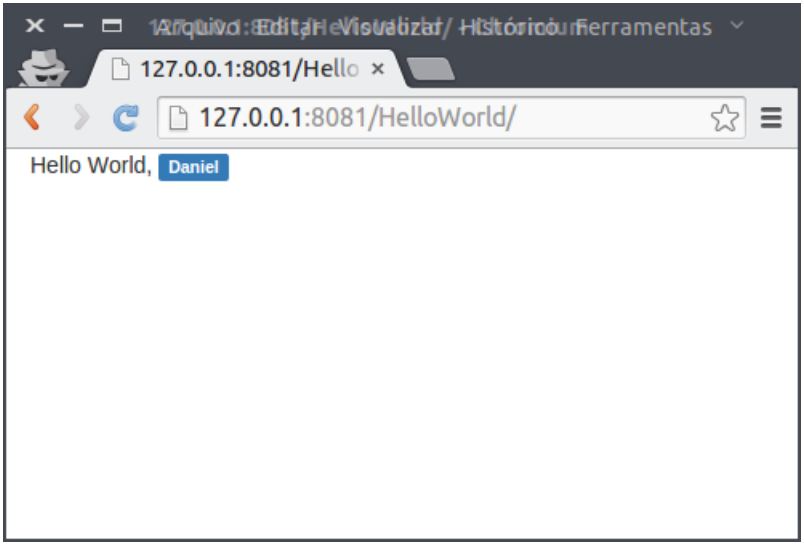
e propriedades e, neste exemplo, estamos utilizando o método `render` para definir como será a renderização da mesma, ou seja, como ela será desenhada.

Para propriedades da classe usamos `this.props` no qual podemos criar diversos atributos que poderão ser adicionados ao elemento `<HelloWorld>`. Estes atributos são referenciados pelo `render` através do uso de chaves `{ ... }`. O método `render` da `ReactDOM` irá renderizar o elemento `<HelloWorld>` da mesma forma que fez no exemplo anterior.

Perceba que quando criamos o elemento `span` utilizamos um pouco de Bootstrap para estilizar a propriedade `this.props.name`. No elemento `span` é utilizado a propriedade `className` fornecendo duas classes css, `label` e `label-primary`.

Neste exemplo compreendemos um pouco mais sobre o React. Criamos uma classe com o `createClass` e ligamos atributos com o `this.props`.

O resultado deste código é semelhante a figura a seguir.



Capítulo 2 - Compreendendo React

Neste capítulo veremos um exemplo mais complexo do que o Hello World do capítulo anterior. O objetivo deste capítulo é criar uma página capaz de desenhar um formulário que exibe os comentários de uma página web. Nesta página serão exibidos os comentários criados e um formulário para a criação de novos comentários.

Comece este exemplo criando o diretório `CommentsComponent` e o arquivo `index.html` com o seguinte código:

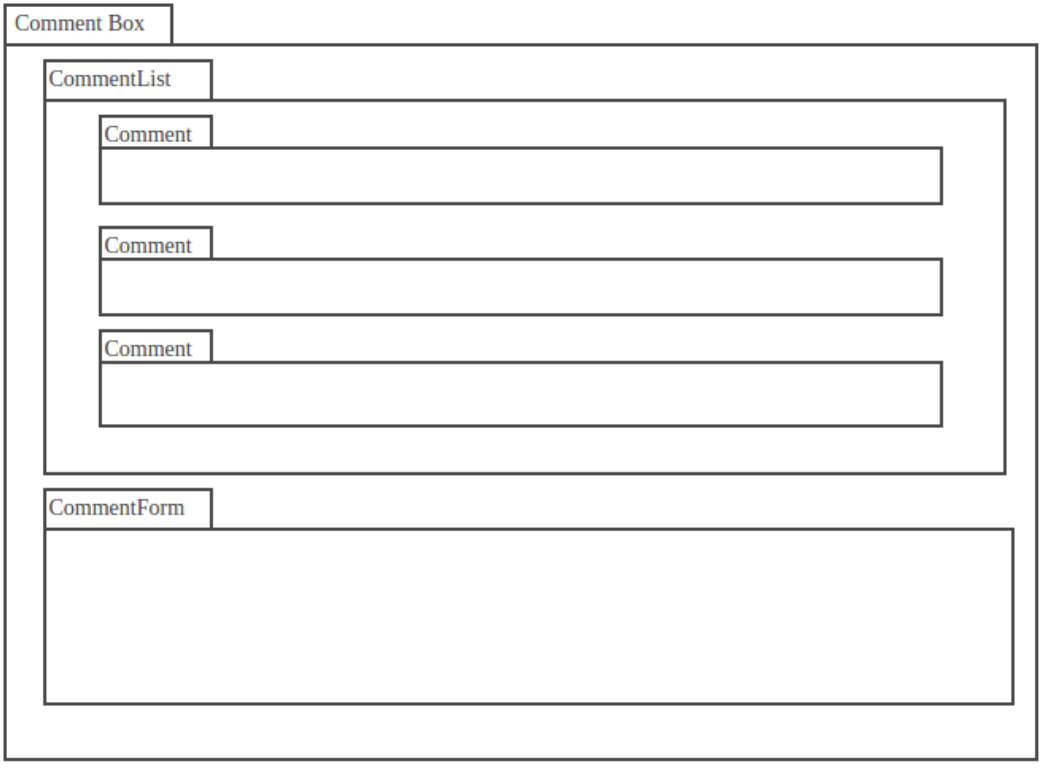
```
 <!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Comment Component</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.j\
s"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-d\
om.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3\
.3.5/css/bootstrap.min.css">
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="main.js"></script>
  </body>
</html>
```

O arquivo `index.html` contém a inclusão das bibliotecas `react`, `react-dom`, `babel` e `bootstrap`. No `<body>` criamos uma div com o id `content` e logo após a div

adicionamos o script `main.js` que conterá todos os nossos componentes. Com isso podemos focar apenas no arquivo `main.js` e na criação dos componentes. O principal propósito deste capítulo é mostrar que, com reat, nós não estamos mais criando código html para definir a camada de visualização da aplicação.

Ou seja, em React você não trabalha diretamente com HTML. Até mesmo os componentes `<a>`, `<p>`, `<h1>` do HTML comum são componentes do react. Isso significa que, ao escrever código em React, não será possível copiar e colar um conteúdo HTML em um componente.

O primeiro passo para criar um componente no React é desenhá-lo. Neste ponto podemos utilizar papel e lápis, ou uma ferramenta qualquer de *mockups*. O desenho pode ser simples, mas irá definir quais serão os componentes utilizados na tela a ser construída. Como estamos criando um formulário que irá exibir os comentários de uma página, podemos definir algo como:



Nesta imagem conseguimos diferenciar vários componentes. O primeiro deles é o `ComponentBox` que contém todos os outros. O `CommentList` é uma lista de componentes do tipo `Comment`, e no final temos o componente `CommentForm`. Assim como nos comentários de uma página de Blog qualquer, usamos o `CommentForm` para adicionar um item na lista de `Comments`.

Criando o primeiro componente

Vamos iniciar a criação dos componentes, começando pelo `CommentBox`. No arquivo `main.js`, insira:

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
       CommentBox
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

Neste código, usamos o `React.createClass` para criar uma classe do React. O método `render` já é conhecido, e a princípio retorna uma `div` simples com o texto `Hello World`. Após a criação do componente, usamos o `ReactDOM` para renderizar o `CommentBox` no elemento `content` da página HTML.

Porque usamos JSX?

Como já abordamos no capítulo anterior, todo código criado no método render se assemelha ao HTML, como a `<div>` criada no componente `commentBox`, mas esta div

nao é um elemento html. Em nenhum momento poderemos copiar código html e colar no método `render` do React, porque ele usa apenas os seus componentes.

Este é um dos principais motivos na qual usamos JSX. Com JSX podemos utilizar componentes do React como se estivéssemos utilizando XML comum. Se não existisse JSX, o componente `CommentBox` seria descrito da seguinte forma:

```
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    );
  }
});
```

O que seria muito mais complexo de compreender. Sem JSX, teríamos que usar métodos do React para descrever o componente, como o `createElement` do exemplo. Com JSX, usamos uma forma mais simples baseada no XML e usamos o Babel para traduzir todo o código em tempo de execução.

Para relembrar, em um servidor de produção, não existe código JSX, apenas o código javascript nativo.

Criando mais componentes

De acordo com a figura anterior sobre os componentes, ainda temos mais alguns para criar. Neste nível inicial, vamos criar os componentes vazios, apenas para que possamos utilizar a sua estrutura inicial.

O componente `CommentList` pode ser criado inicialmente da seguinte forma:

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        CommentList
      </div>
    );
  }
});
```

E o componente `CommentForm` segue o mesmo modelo:

```
var CommentForm = React.createClass({
  render: function() {
    return (
      <div className="commentForm">
        CommentForm
      </div>
    );
  }
});
```

Após a criação do `CommentList` e do `CommentForm`, podemos retornar ao `CommentBox` e alterá-lo para o seguinte código:

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h2>Comments</h2>
        <CommentList />
        <h2>Add a Comment</h2>
        <CommentForm />
      </div>
    );
  }
});
```

Perceba que adicionamos os dois componentes `CommentList` e `CommentForm` no `ComponentBox`. Antes de testarmos esta alteração no navegador, certifique-se que a inclusão dos componentes respeita a ordem de criação dos mesmos, isto é, se o `CommentBox` inclui o componente `CommentList`, este deve ser criado primeiro. Para facilitar, vamos a seguir exibir o código completo do arquivo `main.js`:

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        CommentList
      </div>
    );
  }
});
var CommentForm = React.createClass({
  render: function() {
    return (
      <div className="commentForm">
        CommentForm
      </div>
    );
  }
});
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h2>Comments</h2>
        <CommentList />
        <h2>Add a Comment</h2>
        <CommentForm />
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
```

```
  document.getElementById('content')
);
```

Neste momento, a aplicação se assemelha a imagem a seguir:

Criando o componente Comment

O componente `Comment` representa o comentário em si, que neste exemplo possui o nome da pessoa e o texto do comentário. Perceba que temos duas variáveis, que podem ser inseridas no componente através do `this.props`.

```
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h3>
          {this.props.author}
        </h3>
        {this.props.children}
      </div>
    );
  }
});
```

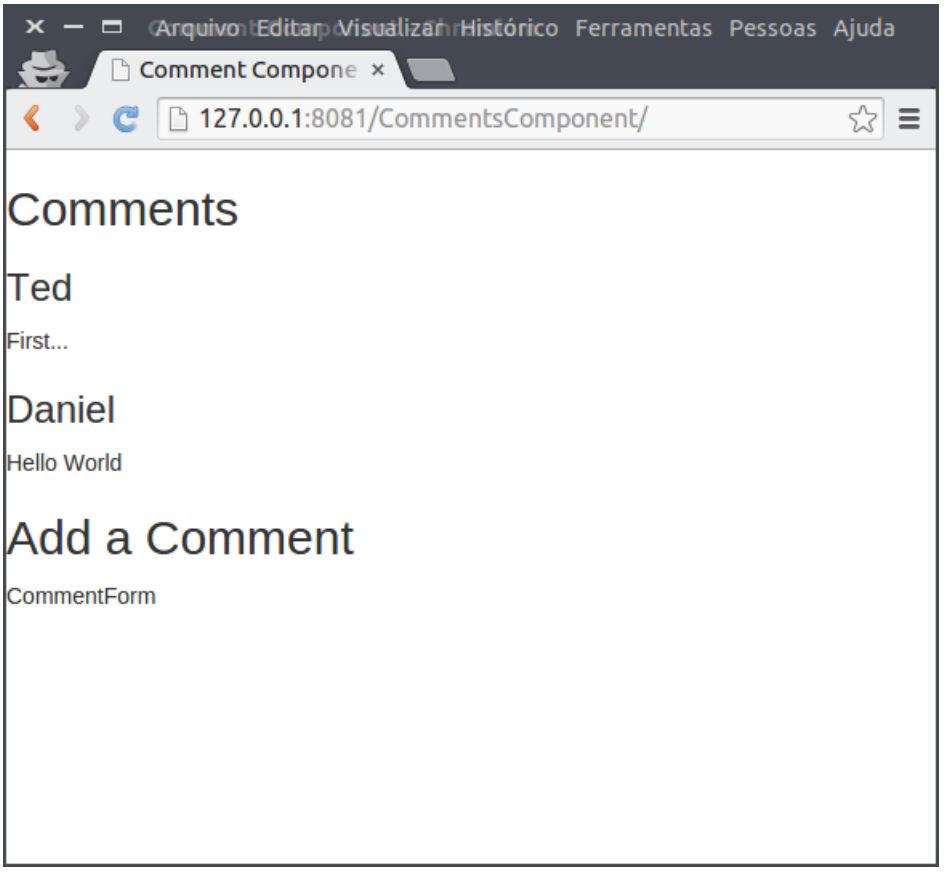
Através do `this.props.author` e `this.props.children` podemos inserir o nome do autor do comentário e o comentário em si. A variável `children` representa o texto que está compreendido no elemento xml `comment`, conforme o exemplo a seguir:

```
<Comment author="Daniel">Hello World !!!</Comment>
```

O componente `CommentList` pode então receber uma lista de componentes `Comment`, conforme o exemplo a seguir:

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
         <Comment author="Ted">First...</Comment>
         <Comment author="Daniel">Hello World</Comment>
      </div>
    );
  }
});
```

Perceba que, como `CommentList` adiciona o componente `Comment`, o componente `Comment` deve ser incluído antes do componente `CommentList`, no arquivo `main.js`. O resultado do código até este momento é semelhante a figura a seguir:



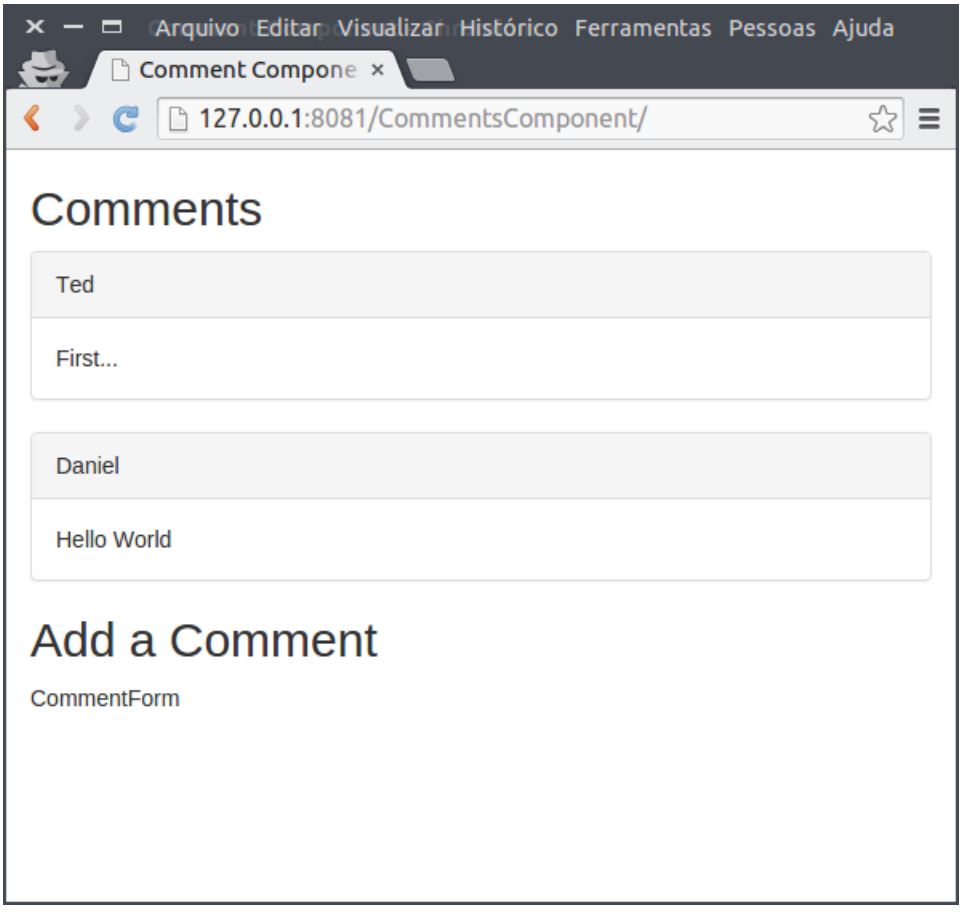
Estilizando com bootstrap

Como vimos na imagem anterior, ainda não temos um estilo aplicado diretamente aos componentes. Foram adicionados algumas propriedades do tipo `className='commentBox'` mas até o momento ainda não a implementamos.

Criar o estilo da aplicação não interfere em nada no aprendizado do React, então iremo refatorar a seguir o componente `Comment`, utilizando os estilos do bootstrap, veja:

```
var Comment = React.createClass({
  render: function() {
    return (
      <div className="panel panel-default comment">
        <div className="panel-heading">
            {this.props.author}
        </div>
        <div className="panel-body">
            {this.props.children}
        </div>
      </div>
    );
  }
});
```

O estilo acima produz o seguinte resultado:



Fique a vontade em estilizar a sua aplicação, observando sempre que os componentes html introduzidos no React não são os elementos do HTML, então observe que ao invés de `class='panel'` nós temos que usar `className='panel'`. Tome cuidado ao copiar/colar código html.

Quanto mais componentes, melhor!

Uma observação importante quando estamos programando em React é estar sempre buscando a "componentização". Por exemplo, o componente `Comment` possui código HTML de um `Panel` do *bootstrap*. Isso sugere que devemos criar um componente

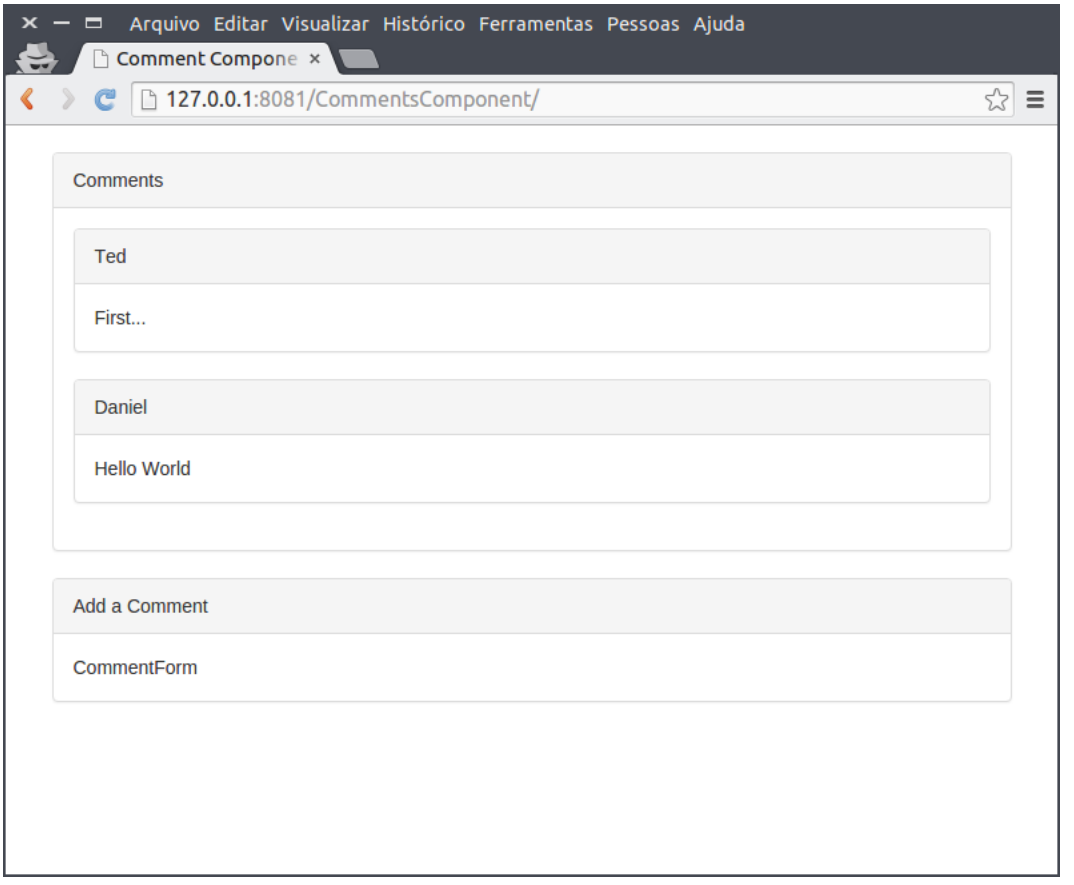
chamado `Panel` e fazer com que o `Comment` use ele. Como um `Panel` tem um título e um texto, ele pode ser reproduzido como:

```
var Panel = React.createClass({
  render: function() {
    return (
      <div className="panel panel-default comment">
        <div className="panel-heading">
            {this.props.title}
        </div>
        <div className="panel-body">
            {this.props.children}
        </div>
      </div>
    );
  }
});
var Comment = React.createClass({
  render: function() {
    return (
      <Panel title={this.props.author}>
        {this.props.children}
      </Panel>
    );
  }
});
```

Perceba que tiramos toda a implementação do `Panel` do componente `Comment` e criamos um novo componente. Agora, o componente `Panel` pode ser reutilizado em todo o sistema, como por exemplo:

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <Panel title="Comments">
            <CommentList />
        </Panel>
        <Panel title="Add a Comment">
            <CommentForm />
        </Panel>
      </div>
    );
  }
});
```

O código acima produz o seguinte resultado:



E os dados em JSON ?

No exemplo anterior criamos o componente `CommentList` repassando os dados do nome do autor e o texto do comentário diretamente pelo componente:

```
<Comment author="Ted">First...</Comment>
<Comment author="Daniel">Hello World</Comment>
```

Em um exemplo real, sabemos que os dados do componente serão obtidos no servidor, através de JSON. Ou seja, temos que alterar o componente `CommentList` para que ele crie dinamicamente os componentes `Comment`.

Inicialmente precisamos criar uma variável que contém estes dados em JSON. Não vamos a princípio usar Ajax para obter esses dados, para facilitar o entendimento do React.

No início do arquivo `main.js`, crie a variável `data`:

```
var data = [
                    {"author": "Ted", "text": "First"},
                    {"author": "Daniel", "text": "Hello World"}
    ];
```

Esta variável precisa ser fornecida no componente `CommentList`. Como o componente está inserido no componente `CommentBox`, precisamos primeiro adicionar esta variável a ele, isto é:

```
ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Com o atributo `data` preenchido, o `CommentList` pode usá-lo da seguinte forma:

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <Panel title="Comments">
            <CommentList data={this.props.data} />
        </Panel>
        <Panel title="Add a Comment">
            <CommentForm />
        </Panel>
      </div>
    );
  }
});
```

Agora o componente `CommentList` possui a variável `data` devidamente configurada, então podemos usá-la para adicionar os comentários dinamicamente.

Para fazer isso, basta criar um loop adicionando o componente `Comment`, da seguinte forma:

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
            {this.props.data.map(function(c){
                return <Comment author={c.author}>
                    {c.text}
                </Comment>;
            })}
      </div>
    );
  }
});
```

O novo componente `CommentList` possui no método `render` uma implementação para adicionar vários `Comment`, mas agora dinamicamente. Isso é obtido adicionando um loop dentro do componente. Este loop é provido pelo método `map`, que é nativo do javascript[4]. Este método faz com que uma função anônima seja chamada para cada item do array, e cada item deste array é repassado como um parâmetro, no nosso caso a variável `c`. O método `map` precisa de um retorno, só que como estamos utilizando JSX, podemos retornar um compnente qualquer do React, neste caso podemos fazer `return <Comment.....`, ou seja, para cada item do `data` temos um `return <Comment.....`.

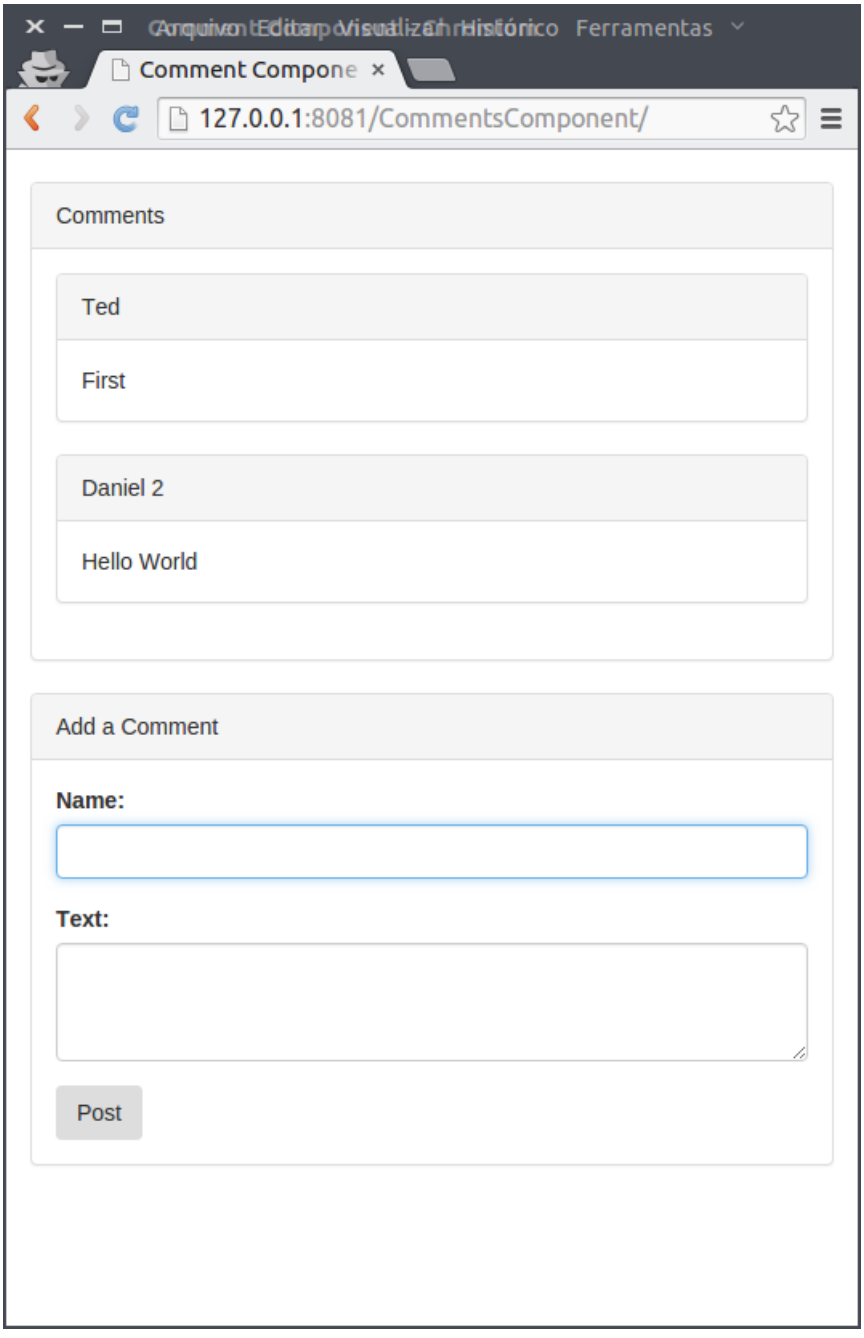
Adicionando comentários

Outro componete do `CommentBox` é o `CommentForm`, um formulário que contém os campos `Nome` e `Texto`, conforme o código a seguir:

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

```
var CommentForm = React.createClass({
  render: function() {
    return (
    <form className="commentForm" onSubmit={this.handleSubmit}>
          <div className="form-group">
            <label htmlFor="name">Author:</label>
            <input type="text" className="form-control" ref="author"/>
          </div>
          <div className="form-group">
           <label htmlFor="text">Text:</label>
            <textarea className="form-control"  rows="3" ref="text">
            </textarea>
          </div>
          <input type="submit" value="Post" className="btn" />
      </form>
    );
  }
});
```

Neste código criamos o formulário `<form>` e adicionamos dois campos, `Nome` e `Email`. Usamos várias classes do *bootstrap* para estilizar o formulário, que fica semelhante a imagem a seguir:


Arquivo Editar Visualizar Histórico Ferramentas
Comment Compone
127.0.0.1:8081/CommentsComponent/
Comments
Ted
First
Daniel 2
Hello World
Add a Comment
Name:
Text:
Post

Diversos elementos e propriedades foram utilizadas na criação do formulário, sendo que a maioria deles é conhecida. O elemento `<input>` cria uma caixa de texto no formulário, e possui propriedades como `id`, `name` (comuns ao jQuery/Bootstrap) além da propriedade `type` do HTML e uma **nova** propriedade do React chamada `ref`, que é uma referencia do `input` ao componente em si. É necessário utilizar `ref` para que o React consiga manipular a DOM corretamente.

O ultimo elemento do `CommentForm` é o botão `Post` que irá submeter o formulário. Para que possamos capturar a submissão do formulário, usamos o método `handleSubmit`, que é referenciado no elemento `<form>`, conforme o código a seguir:

```
<form className="commentForm" onSubmit={this.handleSubmit}>
```

Através deste método, podemos capturar o evento e manipulá-lo, da seguinte forma:

```
var CommentForm = React.createClass({
  handleSubmit: function(e){

      //cancela a propagação do evento
      e.preventDefault();

      var author = this.refs.author.value.trim();
      var text = this.refs.text.value.trim();

      if (!text || !author) {
        return;
      }

    //TODO: Precisa atualizar o DATA

    this.refs.author.value = '';
    this.refs.text.value = '';
    return;

  },
  render: function() {
    return (
```

```
      <form className="commentForm" onSubmit={this.handleSubmit}>
          <div className="form-group">
            <label htmlFor="name">Author:</label>
            <input type="text" className="form-control"
                                                        ref="author"/>
          </div>
          <div className="form-group">
           <label htmlFor="text">Text:</label>
            <textarea className="form-control"
                        rows="3" ref="text"></textarea>
          </div>
          <input type="submit" value="Post" className="btn" />
      </form>
    );
  }
});
```

O método `handleSubmit` referenciado no `<form>` é adicionado ao componente `CommentForm`. Este método é responsável em cancelar a propagação do submit do formulário, através do `e.preventDefault()`. Depois, usamos a propriedade `this.refs` para obter o nome do autor do comentário e o texto digitado. Se ambos estiverem preenchidos, temos então que atualizar a lista de comentários, tarefa a ser realizada logo a seguir. Após atualizar a lista, usamos novamente o `this.refs` para limpar os dados do formulário.

Eventos

Para que possamos atualizar a lista de comentários, precisamos entender como os eventos funcionam no React. Um evento é uma ação que ocorre e provoca uma determinada alteração em algum lugar. Os eventos geralmente são usados para garantir uma independência entre os componentes.

Felizmente, o javascript permite a passagem de funções anônimas entre as propriedades de um objeto e isso facilita muito a aplicação de eventos em sua linguagem.

Geralmente os eventos são tratados nos componentes imediatamente pai ao componente de origem. Então supondo que o componente `CommentForm` dispare um evento

chamado `onCommentSubmit`, deverá haver no `CommentBox` um método capaz de tratar este evento, no qual chamaremos de `handleCommentSubmit`.

Neste momento podemos criar um padrão, quando temos eventos disparados usamos o prefixo "on" e os eventos capturados possuem o prefixo "handle".

Ao alterarmos o `CommentBox` teremos um novo método chamado `handleCommentSubmit` e adicionaremos ao `CommentForm` o evento `onCommentSubmit`, conforme o código a seguir:

```
var CommentBox = React.createClass({
  handleCommentSubmit: function(comment){
    //TODO
  },
  render: function() {
    return (
      <div className="commentBox">
        <Panel title="Comments">
            <CommentList data={this.props.data} />
        </Panel>
        <Panel title="Add a Comment">
            <CommentForm onCommentSubmit={this.handleCommentSubmit}/>
        </Panel>
      </div>
    );
  }
});
```

O componente `CommentBox` irá receber o evento do formulário e tratar os dados, o que será feito posteriormente. Agora podemos retornar ao `CommentForm` e programar o disparo do evento no método `handleSubmit`, conforme o código a seguir:

```
var CommentForm = React.createClass({
  handleSubmit: function(e){
      //cancela a propagação do evento
      e.preventDefault();
      var author = this.refs.author.value.trim();
      var text = this.refs.text.value.trim();
      if (!text || !author) {
        return;
      }

    this.props.onCommentSubmit({author:author, text:text});

    this.refs.author.value = '';
    this.refs.text.value = '';
    return;
  },
  render: function() {
  .......continue.............
```

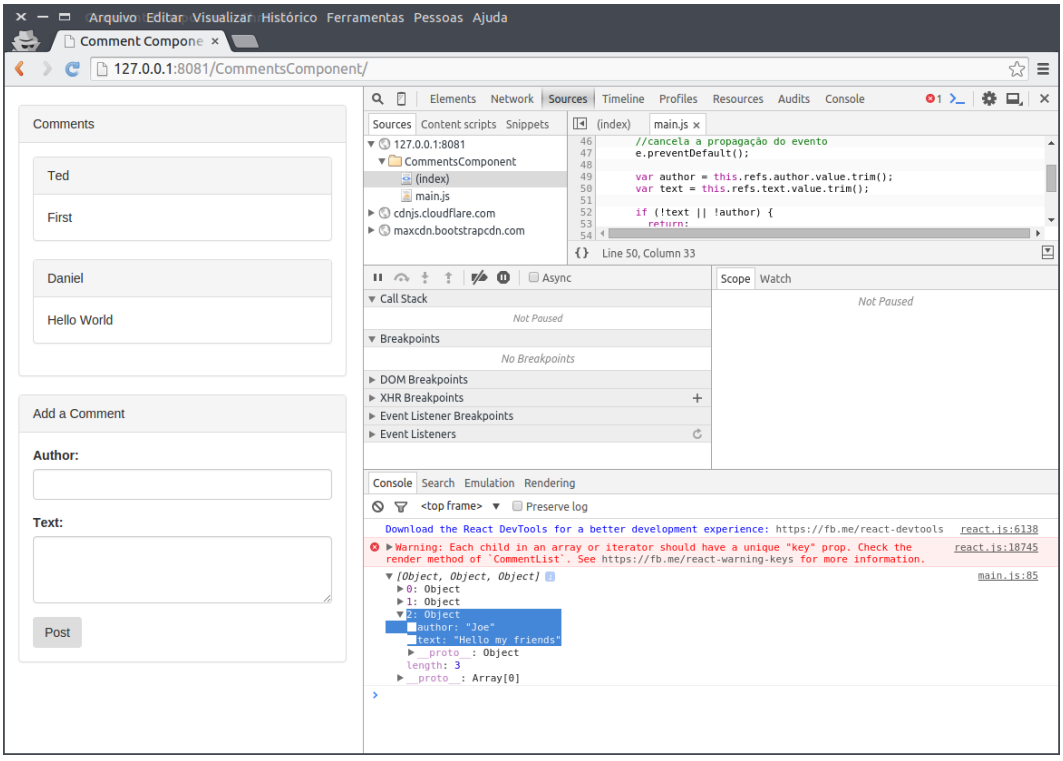
A única alteração no `CommentForm` é a inclusão da chamada do método `onCommentSubmit` repassando o objeto que representa o comentário. Desta forma, ao realizarmos esta chamada, o método `handleCommentSubmit` será chamado.

Adicionando dados a lista de comentários

Ao voltarmos no método `handleCommentSubmit`, temos a tarefa final de adicionar mais um objeto ao array `data`. Inicialmente, podemos supor o seguinte código:

```
var CommentBox = React.createClass({
  handleCommentSubmit: function(comment){
    this.props.data.push(comment);
    console.log(this.props.data);
  },.................
```

A princípio este código deveria funcionar. A resposta no `console.log`, visto na figura a seguir, indica que o item foi adicionado ao array `data`:



Se o array `data` foi atualizado, porque o comentário não apareceu na tela?

Compreendendo State

Aqui temos um novo conceito chamado de `state`. Quando precisamos alterar a interface do componente (ou reconstruí-la) através da alteração de dados na aplicação, não devemos utilizar `this.props`, pois esta propriedade é utilizada somente com valores estáticos.

Qualquer valor dinâmico deve ser implementado através do `state` e isso é feito através da manipulação de dois métodos: `setState` e `getInitialState`.

O método `getInitialState` deverá retornar os dados iniciais que serão a base para o render do componente. No nosso caso, este método será responsável em repassar o valor do array `data`.

O método `setState` deverá ser utilizando sempre que os dados forem alterados, e deseja-se que o componente seja redesenhado.

No nosso exemplo, precisamos alterar o componente `CommentBox` porque ele é o responsável em redesenhar a lista de comentários `CommentList`. No `CommentBox`, temos:

```
var CommentBox = React.createClass({
   getInitialState: function() {
    return {data: this.props.data};
  },
  handleCommentSubmit: function(comment){
    var dataNew = this.state.data;
    dataNew.push(comment)
    this.setState({data:dataNew});
  },
  render: function() {
    return (
      <div className="commentBox">
        <Panel title="Comments">
            <CommentList data={this.state.data} />
        </Panel>
        <Panel title="Add a Comment">
            <CommentForm onCommentSubmit={this.handleCommentSubmit}/>
        </Panel>
      </div>
    );
  }
});
```

O método `getInitialState` retorna o objeto inicial que deverá configurar o `this.state`. Neste caso, retornamos com o objeto `this.state.data` que foi fornecido no `<CommentBox data={data} />`. No método `handleCommentSubmit` criamos a variável `dataNew` apenas para facilitar o processo. Esta variável obtém o valor atual de `this.state.data`, adicionando logo em seguida o novo `comment` que foi enviado pelo evento do `CommentForm`. Após adicionar o comenário, usamos o método `setState` repassando novamente o objeto `date`. Desta forma, quando o `setDate` é chamado,

o React irá atualizar todos os componentes que usam o `this.state`, neste caso o `CommentList` será redesenhado.

É preciso entender que tanto o `CommentList` quanto o `Comment` não precisam manipular o seus respectivos *states*, já que o componente que deve ser redesenhado é o `CommentBox`, com a nova lista de comentários.

Compreender este processo é importante para implementarmos as mudanças de estado de um componente de acordo com os seus dados.

Capítulo 3 - React modularizado

No capítulo anterior criamos um exemplo chamado CommentComponent no qual criamos várias classes que representava os componentes de uma tela de comentários. Perceba que todo o processo foi desenvolvido em com dois arquivos, o `index.html` que continha a inclusão das bibliotecas do React e Bootstrap, e `main.js` com todos os componentes.

Para o aprendizado do React este formato é aceitável, mas para criar uma aplicação maior, ter apenas um arquivo Javascript com todos os componentes não é recomendando.

Este capítulo irá discutir todos os passos que devem ser realizados para criar uma aplicação modularizada, pronta para o servidor de produção. Neste contexto, nós precisamos utilizar Node e suas mais diversas ferramentas para que possamos criar uma plataforma de publicação, que pode ser executada sempre que precisarmos.

A estrategia neste ponto é criar uma estrutura para desenvolvimento, onde cada componente é um arquivo em separado e usar uma ferramenta de automação para que todos estes arquivos sejam "juntados", compactados em somente um único arquivo javascript. O mesmo deverá ser feito para os arquivos CSS, se houver.

Isso sugere que tenhamos uma estrutura na qual temos um diretório chamado `src` que é o fonte do projeto, ou seja, é onde escrevemos as classes do React de forma separada, e um outro diretório chamado `public` que possui os arquivos que serão expostos ao servidor web. A maioria dos arquivos da pasta `public` serão gerados pela ferramente de automação, que usará os arquivos da pasta `src` como fonte.

Criando a estrutura inicial

Primeiro crie o diretório `CommentComponent2` e nele crie as pastas `src` e `public`. Certifique-se de estar com Node e Npm instalados, e digite o seguinte comando: `npm init`. Este comando irá lhe requisitar várias informações sobre o projeto, como o

nome, licença, descrição, etc. Quando terminar, o arquivo `package.json` será criado. Ele contém todas as informações do seu projeto, principalmente as bibliotecas que serão utilizadas, como `React` e `React-dom`.

Para instalar as bibliotecas Javascript que iremos utilizar, execute o seguinte comando:

```
npm install react react-dom bootstrap jquery --save
```

Após instalar as bibliotecas, a pasta `node_modules` é criada e nela as bibliotecas e suas dependências são baixadas (o npm conecta no github de cada projeto e faz o download). O `--save` irá salvar estas bibliotecas no `package.json`, desta forma pode-se reinstalar elas a qualquer momento.

O próximo comando que iremos executar é mais extenso e é utilizado para que possamos utilizar a ferramenta de automação `gulp`, que irá reunir todas as bibliotecas da pasta `src` e `node_modules` e compactar para a a pasta `public`. Execute então os seguintes comandos:

```
npm install babelify browserify watchify --save-dev
npm install gulp gulp-concat gulp-sourcemaps --save-dev
npm install gulp-uglify gulp-uglifycss gulp-util --save-dev
npm install vinyl-buffer vinyl-source-stream --save-dev
```

O argumento –save-dev irá salvar a lista de bibliotecas instaladas no arquivo `package.json`, mas estas bibliotecas serão marcadas como 'dev', pois são usadas apenas quando estamos desenvolvendo o sistema e, teoricamente, não necessitam ser instaladas no servidor de produção.

Criando o arquivo de automação

Utilizaremos o *Gulp* como ferramente de automação. Como nao é o foco deste obra o seu entendimento, vamos apresentar a seguir o arquivo `gulpfile.js` que já contém toda a estrutura de automação pronta para uso.

Crie o arquivo `gulpfile.js` com o seguinte código:

```
var watchify = require('watchify');
var browserify = require('browserify');
var babelify = require('babelify');
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglifycss = require('gulp-uglifycss');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var gutil = require('gulp-util');

var b = watchify(browserify({
    entries: './src/index.jsx',
    debug: true,
    extensions: ['.jsx'],
    transform: ['babelify']
}));

gulp.task('default', process);
b.on('update', process);

function process(){
     b.bundle()
    .pipe(source('main.min.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
    .pipe(uglify())
    .on('error', gutil.log)
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('./public/js/'));
    gulp.src(["./node_modules/bootstrap/dist/css/bootstrap.min.css","./src/s\
tyle.css"])
    .pipe(concat('style.min.css'))
    .pipe(uglifycss())
    .pipe(gulp.dest('./public/css/'));
}
```

Neste arquivo utilizamos diversas tarefas que tem como principal funcionalidade criar os arquivos `public/js/main.min.js` e `public/css/style.min.css`. Por exemplo, a propriedade `transform: ['babelify']` irá realizar a conversão dos códigos em JSX para javascript comum, já o comando `browserify` é responsável em carregar as dependências de bibliotecas javascript.

Criando o arquivo public/index.html

O arquivo `public/index.html` será o arquivo html visível para a web, que deve incluir as bibliotecas javascript e css que foram geradas pelo *gulp*, veja:

```
<html>
<head>
    <title> Comment Component </title>
    <link rel="stylesheet" href="css/style.min.css">
</head>
<body>
    <div id="content" class="container"></div>
    <script type="text/javascript" src="js/main.min.js"></script>
</body>
</html>
```

Este arquivo html é semelhante ao arquivo `index.html` do capítulo anterior. A principal diferença é a não utilização das bibliotecas CDN e a conversão do JSX para JS, tarefas estas que são feitas pelo *gulp*.

Criando os componentes JSX

Agora vamos criar o arquivo `src/index.jsx` que é o arquivo principal da aplicação javascript. É através deste arquivo que iremos instanciar outros componentes do React.

Após criar o arquivo, adicione a seguinte código inicial:

```
import React from 'react';
import ReactDOM from 'react-dom'
import CommentBox from './comment/CommentBox';
```

Estes imports fazem parte da especificação *ECMAScript 6* do javascript, que ainda não são compatíveis com os navegadores atuais. Se você usar isso diretamente no navegador, não irá funcionar. Mas nós estamos trabalhando com Node e gulp justamente para prover este suporte. O gulp irá "entender" este comando e "importar" os arquivos javascript necessários, juntando tudo e criando o arquivo `public/js/-main.min.js`.

Veja que o comando `import React from 'react'` irá importar o react para este arquivo, e poderá ser utilizado da seguinte forma:

src/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom'
import CommentBox from './comment/CommentBox';

var data = [
                    {"author": "Ted", "text": "First"},
                    {"author": "Daniel", "text": "Hello World"}
];

ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Usamos o `ReactDOM.render` da mesma forma do capítulo anterior, e incluímos o componente `<CommentBox />` repassando a variável `data`, assim como foi feito no capítulo anterior. Neste momento, o que temos que fazer é separar todos aqueles componentes do arquivo `script.js` do capítulo anterior em arquivos distintos. Cada arquivo será um componente.

Componente CommentBox

No diretório `src\comment` iremos incluir todos os componentes da parte de comentários. O primeiro deles é o `CommentBox`, que foi referenciado no `index.jsx`:

src/comment/CommentBox.jsx

```
import React from 'react';
import Panel from '../common/Panel';
import CommentForm from './CommentForm';
import CommentList from './CommentList';

var CommentBox = React.createClass({
   getInitialState: function() {
    return {data: this.props.data};
  },
  handleCommentSubmit: function(comment){
    var dataNew = this.state.data;
    dataNew.push(comment)
    this.setState({data:dataNew});
  },
  render: function() {
    return (
     <div className="commentBox">
        <Panel title="Comments">
             <CommentList data={this.state.data} />
        </Panel>
        <Panel title="Add a Comment">
            <CommentForm onCommentSubmit={this.handleCommentSubmit}/>
        </Panel>
      </div>
    );
  }
});
export default CommentBox;
```

Perceba que as únicas novidades no componente estão no início e final do arquivo. No início temos os imports necessários que o componente utiliza, e no final temos

o comando `export` que irá definir o nome deste componente. Ambas mudanças são essenciais para que o *browserify* consiga importar todos os componentes da forma correta.

Componente Panel

Perceba que o componente `CommentBox` utiliza o componente `Panel` e que ele é definido pelo import `import Panel from '../common/Panel';`. Ou seja, o componente `Panel` está referenciado pelo arquivo `/src/common/Panel`, exibido a seguir:

src/common/Panel.jsx

```
import React from 'react';

var Panel = React.createClass({
  render: function() {
    return (
      <div className="panel panel-default comment">
        <div className="panel-heading">
            {this.props.title}
        </div>
        <div className="panel-body">
            {this.props.children}
        </div>
      </div>
    );
  }
});
export default Panel;
```

O componente Panel está no "namespace" `common` porque pode ser usado para outros projetos além do `Comment`. A definição de namespaces para a sua aplicação fica a sua escolha. Por exemplo, pode-se criar o namespace "bootstrap" e incluir o panel em `src/bootstrap/Panel`, já que este componente é o próprio componente Panel do bootstrap.

Componente CommentList

O `CommentList` é referenciado no `CommentBox`, e possui o seguinte cópigo:

```
import React from 'react';
import Comment from './Comment'

var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
            {this.props.data.map(function(c){
                return <Comment author={c.author}>
                    {c.text}
                </Comment>;
            })}
      </div>
    );
  }
});
export default CommentList;
```

Novamente as únicas diferenças em relação do `CommentList` do capítulo anterior está nos `imports` iniciais e no `export` do final do arquivo.

CommentForm

O `CommentForm` possui o seguinte código:

src/comment/CommentForm.jsx

```
import React from 'react';

var CommentForm = React.createClass({
  handleSubmit: function(e){
          //cancela a propagação do evento
          e.preventDefault();
          var author = this.refs.author.value.trim();
          var text = this.refs.text.value.trim();
          if (!text || !author) {
        return;
         }
        this.props.onCommentSubmit({author:author, text:text});
           this.refs.author.value = '';
        this.refs.text.value = '';
        return;
},
render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
      <div className="form-group">
      <label htmlFor="name">Author:</label>
      <input type="text" className="form-control"
      ref="author"/>
      </div>
      <div className="form-group">
      <label htmlFor="text">Text:</label>
      <textarea className="form-control"
      rows="3" ref="text"></textarea>
      </div>
      <input type="submit" value="Post" className="btn" />
      </form>
      );
}
});
export default CommentForm;
```

Comment

O último componente a ser extraído do arquivo `script.js` do capítulo anterior é o Comment, exibido a seguir:

src/comment/Comment.jsx

```
import React from 'react';
import Panel from '../common/Panel';

var Comment = React.createClass({
  render: function() {
    return (
      <Panel title={this.props.author}>
        {this.props.children}
      </Panel>
    );
  }
});
export default Comment;
```

Veja que o `Comment` usa o componente `Panel`, ou seja, os componentes podem ser reutilizados livremente, permitindo assim que possamos repassar quase todos os elementos HTML para componentes, e usá-los na nossa aplicação.

Capítulo 4 - Ajax

Neste capítulo vamos abordar um pouco sobre o Ajax, já que toda aplicação SPA (Single Page Application) necessita do uso constante de Ajax. O React em si não tem métodos para realizar chamadas ao servidor via Ajax, então precisamos incluir a biblioteca jQuery em nossa aplicação.

Exemplo

Neste primeiro exemplo, vamos carregar os *Gists* da sua conta do Github. Vamos utilizar CDN e escrever todo o código JSX no arquivo `main.js`, para facilitar.

Crie a pasta `gists`, e em seguida crie o arquivo `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title> My Gists </title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.j\
s"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-d\
om.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/br\
owser.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.\
min.js"></script>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3\
.3.5/css/bootstrap.min.css">
  </head>
  <body>
    <br/>
    <div id="content" class="container"></div>
```

```
    <script type="text/babel" src="main.js"></script>
  </body>
</html>
```

Perceba que, além de adicionar as bibliotecas do React e do babel, incluímos também o jQuery.

O arquivo `main.js` contém inicialmente a seguinte estrutura:

```
var UserGist = React.createClass({
 render: function() {
    return (
      <div>
        Your gist url: {this.props.source}
      </div>
    );
  }
});
ReactDOM.render(
  <UserGist source="https://api.github.com/users/<usuario>/gists" />,
  content
);
```

A primeira versão do `main.js` tem apenas a estrutura básica do componente `UserGist`, contendo a propriedade `source` que é o link para a api do GitHub que contém os seus Gists. Lembre de trocar o `<usuario>` pela sua conta.

Para relembrar, sabemos que `this.props` deve ser usado para variáveis estáticas no react, ou seja, variáveis que não influenciam na renderização do componente, ou seja, não mudam o visual do componente de acordo com alguma mudança de informação. Já `this.state` é usado para realizar mudanças no layout. Sempre que o método `this.setState()` é chamado, o método `render` é reexecutado de acordo com as novas informações do `this.state` e aind atemos o `getInitialState` que define o estado inicial do `state`.

Método componentDidMount

Como pode-se perceber, no React existem os lugares certos para manipular informação. No caso do ajax, existe um método especial chamado `componentDidMount` que é executado automaticamente pelo React quando o componente é renderizado. Sempre que precisamos carregar alguma informação ajax quando o componente é criado pela primeira vez, inserimos a chamada ajax no método `componentDidMount`.

No próximo código, criamos a variável `this.state.data` que irá conter todo o json obtido pela api do github, veja:

```
var UserGist = React.createClass({
  getInitialState:function(){
      return {
        data: []
      };
  },
  componentDidMount:function(){
    $.ajax(this.props.source).done(function(result){
      this.setState({data:result});
    }.bind(this));
  },
 render: function() {
    return (
      <div>
        Your gist url: {this.props.source}
        <br/>
        <div>Your data has {this.state.data.length} entries</div>
      </div>
    );
  }
});
ReactDOM.render(
  <UserGist source="https://api.github.com/users/danielschmitz/gists" />,
  content
);
```

Perceba que, como estamos manipulando o `state` no componente, criamos o método

`getInitialState`, que configura a variável `this.state.data` com o valor `[]`, que é um array vazio. Então usamo o método `componentDidMount` que é responsável em obter os dados via json, através do `$.ajax` do jQuery. Quando o Ajax retorna com os dados do servidor, o método `done` é executado e os dados estão no parâmetro `result`, que é atribuído ao `state`, através do método `this.setState`. É importante ter o conhecimento que o `this` do callback `done` representa o componente React em si, graças ao `.bind(this)`. Sem o `bind`, o `this` apontaria para o callback e `this.setState` não seria acessível.

O método render agora utiliza o `{this.state.data.length}` para mostrar a quantidade de itens do Array. Inicialmente é `0`, mas depois de alguns segundos (ou até menos) ele é atualizado. Essa atualização é definida pelo `setState`, que executa novamente o `render` com os dados atualizados.

Exibindo dados através de um loop

Para finalizar este exemplo, podemos utilizar a função `map` nativa do javascript para exibir a lista de Gists existentes. Pra isso, adicione o seguinte código no método render:

```
render: function() {
   return (
     <div>
       Your gist url: {this.props.source}
       <br/>
       <div>Your data has {this.state.data.length} entries</div>
       <ul>
           {this.state.data.map(function(item){
               return <li><a href={item.url}>{item.description}</a></li>;
           })}
       </ul>
     </div>
   );
 }
```

Perceba que criamos o elemento de lista `<ul>` e usamos `this.state.data.map` para criar uma função anônima para cada item do array. Esta função deve retornar

código React, neste caso retorna o componente `li` usando os dados json `item.url` e `item.description`.

O atributo key

No exemplo anterior, ao visualizá-lo no navegador, percebe-se um pequeno *warning* com a seguinte mensagem "Warning: Each child in an array or iterator should have a unique 'key' prop.". Este alerta diz que todo item de loop deve ter um atributo chamado `key`, para que o React possa otimizar a alteração da DOM nos elementos filhos do loop que criamos. Perceba que no método `map` repassamos o parâmetro `item`, que corresponde a cada item do array de gists existentes no json. Podemos utilizar `item.id` como um índice e fornecê-lo no parâmetro `key` do elemento `<li>`, veja:

```
{this.state.data.map(function(item){
  return <li key={item.id}><a href={item.url}>{item.description}</a></li>;
})}
```

Refatorando CommentsController

No exemplo do capítulo anterior, iniciamos a refatoração do componente `CommentsController` dividindo-o em vários componentes React. O componente `src/index.jsx` possui o seguinte código:

```
import React from 'react';
import ReactDOM from 'react-dom'
import CommentBox from './comment/CommentBox';

var data = [
                    {"author": "Ted", "text": "First"},
                    {"author": "Daniel", "text": "Hello World"}
];

ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Perceba que criamos a variável `data` inserindo alguns dados manualmente, o que não é a melhor forma de popular dados em um componente React, já que quem fornece estes dados é o servidor. Isto é, o servidor que é responsável em consultar o banco de dados, criar o JSON com os dados e responder ao cliente.

Neste caso, o componente `CommentBox` deve ter somente a responsabilidade de conhecer a URL que fornece estes dados, e acessá-la através de Ajax.

Observação: No código fonte desta obra, exibiremos esta refatoração na pasta `CommentsComponent3`. Lembre-se de usar o gulp para realizar o deploy da aplicação na pasta public.

Criando a propriedade url

O primeiro passo na refatoração é remover a variável `data` do componente, e inserir a url que contém os dados dos comentários, veja:

```
import React from 'react';
import ReactDOM from 'react-dom'
import CommentBox from './comment/CommentBox';

ReactDOM.render(
  <CommentBox url="comments.json" />,
  document.getElementById('content')
);
```

Perceba que a propriedade `url` aponta para `comments.json`, ou seja, o componente irá realizar uma requisição ajax a url `http://seu_servidor/pasta/comments.json`. Deve-se inserir o arquivo `comments.json` na pasta `public`, que é pasta onde testamos a aplicação. O arquivo `comments.json` possui o seguinte array em json:

```
[
    {"author": "Ted", "text": "First"},
    {"author": "Daniel","text": "Hello World"}
]
```

Alterando o método getInitialState

Como o `CommentBox` não possui mais o atributo `data` devemos alterá-lo no componente. O método `getInitialState` não obtém mais o `data` através do `this.props.data`. No início, `this.state.data` deve ser um array vazio:

```
import React from 'react';
import Panel from '../common/Panel';
import CommentForm from './CommentForm';
import CommentList from './CommentList';

var CommentBox = React.createClass({
   getInitialState: function() {
    return {data: []};
  },
  ......... (continua).........
```

Preparando o jQuery

O `CommentBox` deve realizar uma requisição ajax para obter os comentários. Isso significa que temos que utilizar uma biblioteca que realiza requisições ajax de uma forma fácil. Como usamos jQuery no exemplo anterior, vamos continuar utilizando-o, mas existem outras bibliotecas como *minifiedjs* ou *zeptojs*.

Como estamos em um desenvolvimento javascript "modularizado", o jquery deve ser introduzido no `CommentsController` da seguinte forma:

```
import jQuery from "jquery"
const $ = jQuery;
```

Utilizamos `const` para definir uma referência da biblioteca `jquery` ao símbolo `$`, para que possamos utilizar, por exemplo, `$.ajax` ao invés de `jQuery.ajax`. E, claro, devemos instalar o jQuery pelo `npm`, da seguinte forma:

```
npm install jquery --save
```

Realizando a requisição ajax

Após estes dois passos, podemos retornar ao `CommentBox` e adicionar o código que irá realizar a requisição Ajax no método `componentDidMount`, conforme o código a seguir:

```
mport jQuery from 'jquery';
import React from 'react';
import Panel from '../common/Panel';
import CommentForm from './CommentForm';
import CommentList from './CommentList';

const $ = jQuery;

var CommentBox = React.createClass({
   getInitialState: function() {
    return {data: []};
  },
  componentDidMount:function(){
    $.ajax(this.props.url).done(function(result){
        this.setState({data:result});
      }.bind(this));
  },
  ... continua ...
```

Perceba que usamos `$.ajax` utilizando `this.props.url` como url para acesso ao servidor, que foi informado no `<CommentBox url='comments.json'/>`. Após a

requisição ser realizada no servidor e este retornar com a resposta, o método `done` será chamado e usaremos `this.setState` para repassar novamente a variável `data`. Desta forma, o método `render` será executado novamente.

Conclusão

O objetivo principal desta obra é fornecer um material introdutório ao estudo do React como ferramenta de desenvolvimento para a criação de componentes em javascript. Por mas simples que isso possa parecer, o React é, em sua essência, um modo de criar componentes para aplicações web SPA (Single Page Application).

O próximo passo para o estudo do framework é a criação de novos componentes para uma aplicação completa, e esta aplicação será realizada na próxima obra, *React na prática*, a ser lançada muito em breve.

Como esta obra foi criada utilizando os recursos oferecidos pelo site leanpub.com, mais conteúdo pode ser adicionado, mas isso depende exclusivamente de vocês, leitores. Entrem na página de feedback do livro e requisitem conteúdo adicional.