Introdução ao React

Os sistemas web mudaram bastante nas últimas duas décadas. Eles surgiram com HTML estático no final dos anos 1990, onde as páginas eram meramente informativas e a interação com usuário se dava unicamente por meio de links.

Nos anos 2000, começa o uso de programação dentro dos servidores web para criar páginas HTML customizadas a partir de dados, o que permite a criação de sistemas minimamente interativos baseados em web. Essa mudança levou ao grande crescimento da área de sistemas web impulsionado majoritariamente pelo e-commerces.

No entanto, a mudança cultural que levaria as pessoas a comprar pela web não foi tão rápida quanto o esperado e houve o estouro da bolha, como foi chamado.

Os sistemas web saíram de evidência nessa época sendo recuperados aproximadamente em 2008 quando surgiram as redes sociais digitais. Essas redes começaram a demandar sistemas mais interativos, responsivos e que não fizessem round-trip, a ida e volta ao servidor que faz com que a página recarregue.

Torna-se comum o uso de Ajax, que permite consultas ao servidor em background e bibliotecas de manipulação do HTML.

No entanto, as redes sociais demandavam tecnologias web mais práticas para seu contexto. Surgem então bibliotecas e frameworks focados em frontend. Começa haver uma migração da lógica de servidor para uma lógica de lado cliente, que usa o servidor apenas para acesso aos dados.

Aparece o contexto de SPA - Single Page Applications, que utiliza apenas uma página HTML modificada conforme a interação com o usuário ou eventos para um sistema inteiro. É nesse contexto em que surge o React, hoje ele é a principal biblioteca de frontend para a produção de SPA.

O React também inaugura o conceito de programação reativa em frontend. Que basicamente é a reconstrução da página sempre que houver mudança nos dados. Iso é feito de uma forma muito inteligente pois ele é capaz de renderizar apenas as partes da página que são dependentes dos dados que foram modificados e não o todo, tendo por resultado um ótimo desempenho.

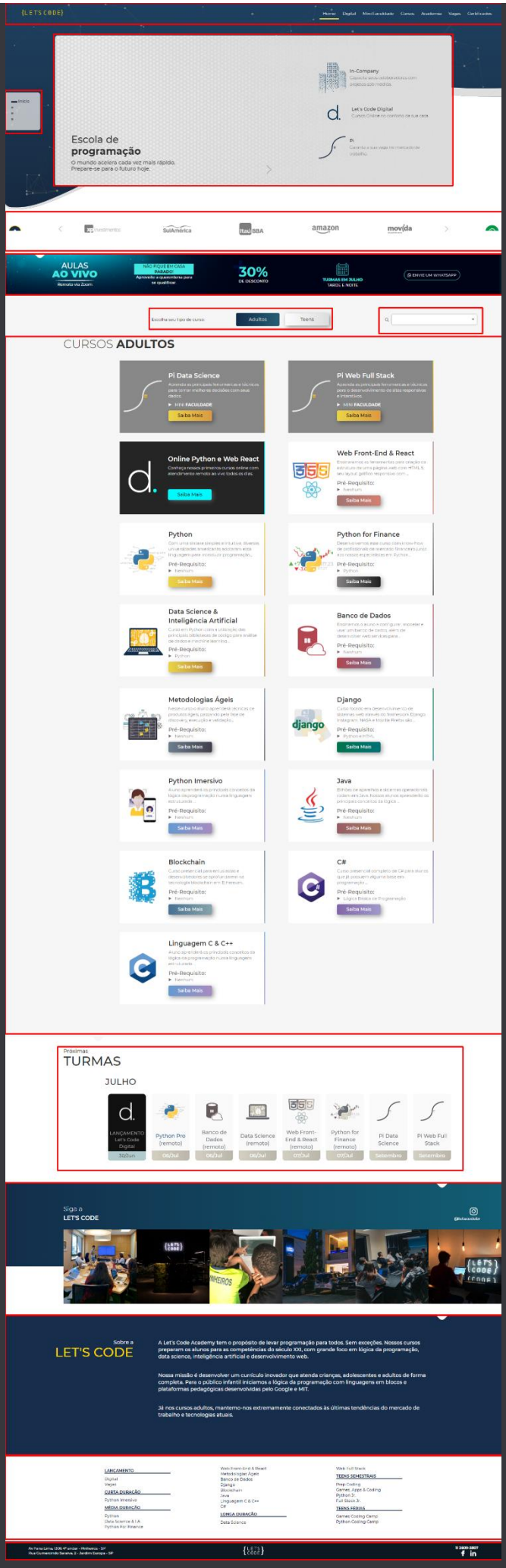
Para isso, o React usa uma arquitetura de componentes. Grande parte da programação em React está em saber dividir bem uma página em componentes e fazê-los de forma reutilizável e que admita composição com outros componentes.

Veremos o conceito de componente a seguir.

Introdução aos Componentes

Tudo o que será renderizado pelo React deve ser um componente. Componentes podem ser entendidos como as "partes" de uma página. Eles podem ser compostos por outros componentes.

Vamos começar vendo como dividir uma tela em componentes:


{LET'S CODE}
Escola de
programação
O mundo acelera cada vez mais rápido.
Prepare-se para o futuro hoje.
In-Company
Let's Code Digital
Pi
AULAS AO VIVO
Remoto via Zoom
30%
DE DESCONTO
TURMAS EM JULHO
TARDE E NOITE
ENVIE UM WHATSAPP
Escolha seu tipo de curso:
Adultos
Teens
CURSOS ADULTOS
Pi Data Science
Pi Web Full Stack
Online Python e Web React
Web Front-End & React
Python
Python for Finance
Data Science & Inteligência Artificial
Banco de Dados
Metodologias Ágeis
Django
Python Imersivo
Java
Blockchain
C#
Linguagem C & C++
Pré-Requisito:
Nenhum
Saiba Mais
Próximas
TURMAS
JULHO
LANÇAMENTO Let's Code Digital
Python Pro (remoto)
Banco de Dados (remoto)
Data Science (remoto)
Web Front-End & React (remoto)
Python for Finance (remoto)
Pi Data Science
Pi Web Full Stack
Siga a
LET'S CODE
Sobre a
LET'S CODE
A Let's Code Academy tem o propósito de levar programação para todos. Sem exceções. Nossos cursos preparam os alunos para as competências do século XXI, com grande foco em lógica de programação, data science, inteligência artificial e desenvolvimento web.
Nossa missão é desenvolver um currículo inovador que atenda crianças, adolescentes e adultos de forma completa. Para o público infantil iniciamos a lógica de programação com linguagens em blocos e plataformas pedagógicas desenvolvidas pelo Google e MIT.
Já nos cursos adultos, mantemo-nos extremamente conectados às últimas tendências do mercado de trabalho e tecnologias atuais.
LANÇAMENTO
CURTA DURAÇÃO
MÉDIA DURAÇÃO
LONGA DURAÇÃO
TEENS SEMESTRAIS
TEENS FÉRIAS

Elencamos para serem subcomponentes aqueles tenham ações, que renderizem conteúdos a partir de dados, que tenham listas ou qualquer tipo de conteúdo não estático.

Instalação

Criação de um projeto React

Para começar vamos criar um projeto.

Você precisa ter o YARN instalado na sua máquina.

Escolha uma pasta no seu computador para criar o projeto. Abra um terminal nessa pasta.

No windows 10, basta clicar com o botão direito do mouse em qualquer lugar da pasta e escolher a opção abrir janela do powershell aqui.

No Linux ou Mac, abra o terminal e navegue até a pasta usando *cd*.

Faremos então o seguinte comando:

```
yarn create react-app meu-app
```

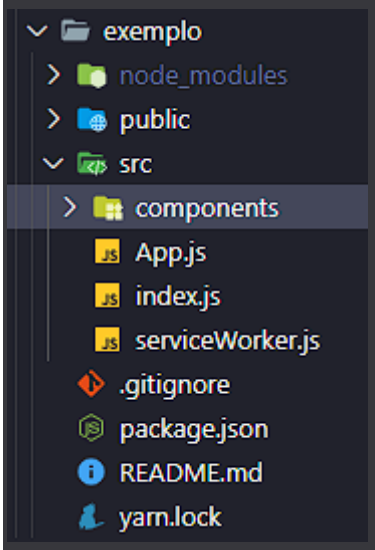
Quando o yarn terminar a instalação, o que leva um tempinho ele irá mostrar como rodar a aplicação.

Faça o que ele mostra:

```
cd meu-app
yarn start
```

Siga esses passos para criar um novo projeto React e abra-o com o visual studio code ou editor de sua preferência.

Teremos a seguinte estrutura:



Todo nosso código será feito na pasta `src`.

Vamos começar limpando o template básico criado por padrão pelo instalador, remova os seguintes arquivos:

```
/src/App.css
/src/App.test.js
/src/index.css
/src/logo.svg
/src/setupTests.js
```

Abra o arquivo App.js e deixe o conteúdo igual a esse (apenas removemos várias linhas):

```
import React from 'react';

function App() {
  return (
    <div className="App">
    </div>
  );
}
export default App;
```

Agora abra o arquivo index.js e deixe o conteúdo igual ao seguinte (apenas removemos várias linhas):

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

serviceWorker.unregister();
```

Aproveitando que estamos nesse arquivo, ele é o arquivo responsável pela injeção do react no HTML da página.

O método *ReactDOM.Render* renderiza um componente na página dentro de um elemento HTML.

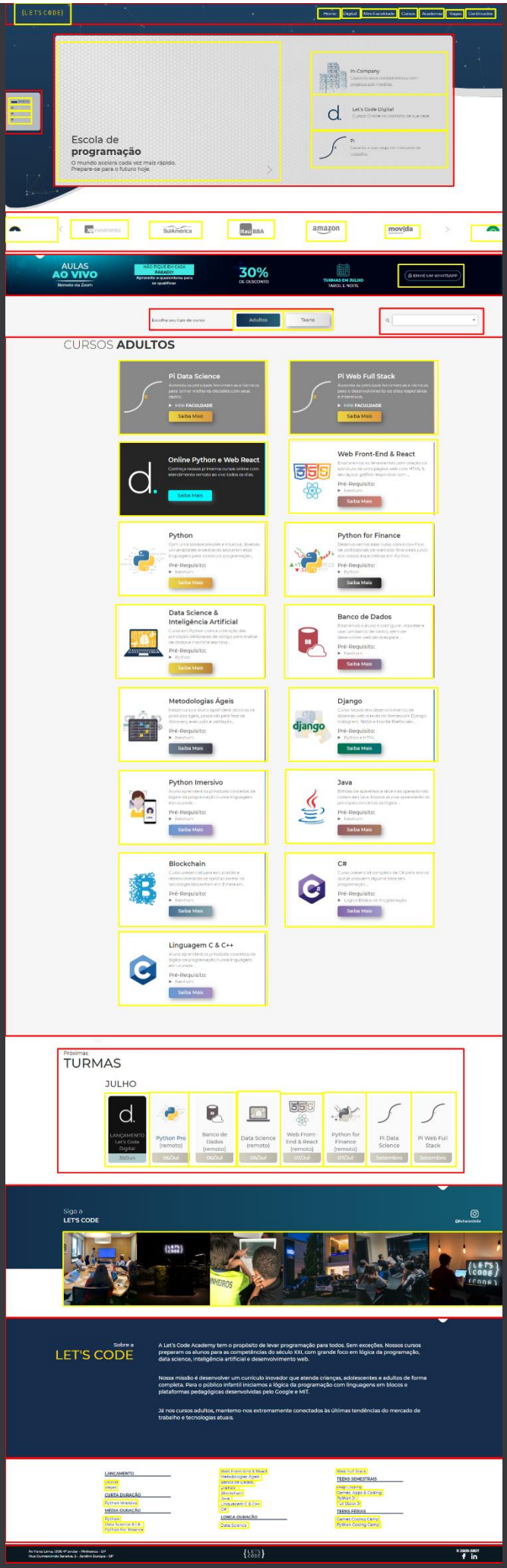
A linha `document.getElementById('root')` define que o elemento HTML onde o react será injetado é um que tenha um ID = 'root'.

Para que um componente seja renderizado ele deve estar dentro do método `ReactDOM.render()` ou dentro de outro componente que seja renderizado pelo método `ReactDOM.render()`.

Se desejar alterar algo no HTML da página por fora do React, você pode encontrá-lo na pasta `public`.

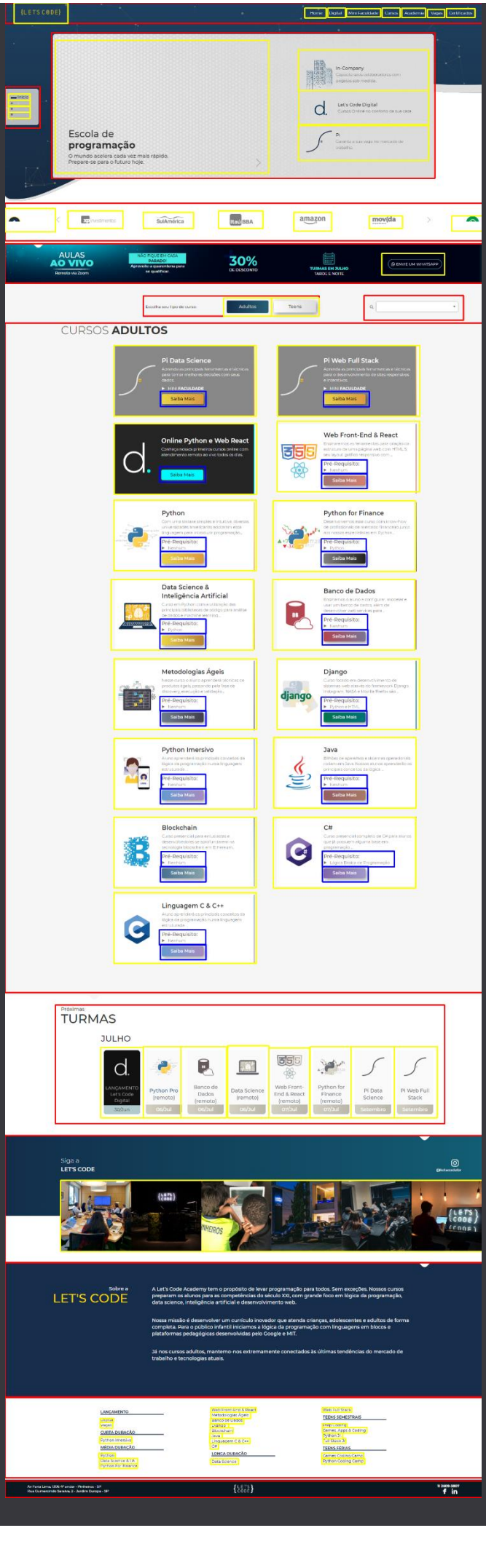
Agora você tem uma aplicação vazia que produz uma página em branco no navegador. Não deve haver erros em seu terminal/powershell.

Se estiver tudo certo, estamos prontos para continuar.


{LET'S CODE}
Escola de programação
O mundo acelera cada vez mais rápido. Prepare-se para o futuro hoje.
In-Company
Let's Code Digital
Pi
AULAS AO VIVO
30% DE DESCONTO
CURSOS ADULTOS
Pi Data Science
Pi Web Full Stack
Online Python e Web React
Web Front-End & React
Python
Python for Finance
Data Science & Inteligência Artificial
Banco de Dados
Metodologias Ágeis
Django
Python Imersivo
Java
Blockchain
C#
Linguagem C & C++
Próximas
TURMAS
JULHO
Python Pro (remoto)
Banco de Dados (remoto)
Data Science (remoto)
Web Front-End & React (remoto)
Python for Finance (remoto)
Pi Data Science
Pi Web Full Stack
Siga a
LET'S CODE
Sobre a
LET'S CODE
A Let's Code Academy tem o propósito de levar programação para todos. Sem exceções. Nossos cursos preparam os alunos para as competências do século XXI, com grande foco em lógica da programação, data science, inteligência artificial e desenvolvimento web.
Nossa missão é desenvolver um currículo inovador que atenda crianças, adolescentes e adultos de forma completa. Para o público infantil iniciamos a lógica da programação com linguagens em blocos e plataformas pedagógicas desenvolvidas pelo Google e MIT.
Já nos cursos adultos, mantemo-nos extremamente conectados às últimas tendências do mercado de trabalho e tecnologias atuais.

Todo o resto pode ser feito em HTML no interior do componente. Nesse caso, fica a nosso critério promover o conteúdo a um componente ou não, tomamos a decisão pensando em alguns critérios:

Por exemplo, partes que, ainda que estáticas, podem ser reaproveitadas em contexto diferente ou são complexas visualmente e carregadas em CSS, são candidatas a se tornarem componentes.


{LET'S CODE}
Escola de programação
O mundo acelera cada vez mais rápido. Prepare-se para o futuro hoje.
In-Company
Let's Code Digital
Pi
AULAS AO VIVO
Remoto via Zoom
30% DE DESCONTO
TURMAS EM JULHO
TARDE E NOITE
ENVIE UM WHATSAPP
Adultos
Teens
CURSOS ADULTOS
Pi Data Science
Pi Web Full Stack
Online Python e Web React
Web Front-End & React
Python
Python for Finance
Data Science & Inteligência Artificial
Banco de Dados
Metodologias Ágeis
Django
Python Imersivo
Java
Blockchain
C#
Linguagem C & C++
Saiba Mais
Próximas
TURMAS
JULHO
Python Pro (remoto)
Banco de Dados (remoto)
Data Science (remoto)
Web Front-End & React (remoto)
Python for Finance (remoto)
Pi Data Science
Pi Web Full Stack
Siga a
LET'S CODE
Sobre a
LET'S CODE
A Let's Code Academy tem o propósito de levar programação para todos. Sem exceções. Nossos cursos preparam os alunos para as competências do século XXI, com grande foco em lógica de programação, data science, inteligência artificial e desenvolvimento web.
Nossa missão é desenvolver um currículo inovador que atenda crianças, adolescentes e adultos de forma completa. Para o público infantil iniciamos a lógica de programação com linguagens em blocos e plataformas pedagógicas desenvolvidas pelo Google e MIT.
Já nos cursos adultos, mantemo-nos extremamente conectados às últimas tendências do mercado de trabalho e tecnologias atuais.
LANÇAMENTO
CURTA DURAÇÃO
MÉDIA DURAÇÃO
LONGA DURAÇÃO
TEENS SEMESTRAIS
TEENS FÉRIAS

Em termos computacionais os componentes podem ser criados de duas formas:

Por funções, chamados de componentes funcionais.

Por classes, que chamaremos de "componentes tipo classe".

Veremos em seguida os componentes do tipo classe.

Componentes Simples

Seguindo o guia para criação de um novo projeto você deve ter um projeto rodando e sem conteúdo.

Vamos começar a criar componentes nele. Começaremos pelos componentes criados como classes.

Podemos reaproveitar o componente App para esse exemplo. App é um componente funcional, vamos abordá-los mais para frente no curso. Por enquanto vamos transformá-lo em um componente tipo classe.

Para começar apague a função App:

```
import React from 'react';

export default App;
```

Vamos criar uma classe App no lugar:

```
import React from 'react';

class App extends React.Component{

}

export default App;
```

Observe que componentes tipo classe utilizam herança da classe Component do React, sendo assim, todo o comportamento de componente já está presente na sua classe desde o primeiro momento. Vamos apenas customizá-lo.

O que desejarmos mostrar na tela usando nosso componente é feito pelo método `render()`. Ele deve retornar um HTML (depois veremos que é mais do isso). Vamos adicioná-lo:

```
import React from 'react';

class App extends React.Component{

    render(){
        return <p>Meu primeiro parágrafo em React.</p>
    }
}

export default App;
```

Você já pode ver seu parágrafo aparecer no navegador. Se quiser fazer um HTML de várias linhas, basta colocar todo o conteúdo entre parênteses, mas lembre-se o React só é capaz de renderizar um elemento, portanto todo seu HTML deve ter um elemento raiz:

Não funciona

```
import React from 'react';

class App extends React.Component{

    render(){
        return (
          <p>Meu primeiro parágrafo em React.</p>
          <p>Meu segundo parágrafo em React.</p>
          <div>
            <pre>Cansei de parágrafos...</pre>
          </div>
        );
    }
}

export default App;
```

Funciona

```
import React from 'react';

class App extends React.Component{

    render(){
        return (
          <div>
            <p>Meu primeiro parágrafo em React.</p>
            <p>Meu segundo parágrafo em React.</p>
            <div>
                <pre>Cansei de parágrafos...</pre>
            </div>
          </div>
        );
    }
}

export default App;
```

Se você não quiser colocar um tag HTML como raiz (algumas vezes isso conflita com css da página e coisas do tipo), você pode colocar um `Fragment`, que é basicamente uma tag vazia:

```
import React from 'react';

class App extends React.Component{

    render(){
        return (
          <>
            <p>Meu primeiro parágrafo em React.</p>
            <p>Meu segundo parágrafo em React.</p>
            <div>
                <pre>Cansei de parágrafos...</pre>
            </div>
          </>
        );
    }
}

export default App;
```

Versões muito antigas do React podem não aceitar essa sintaxe de fragment, nesse caso use `<React.Fragment></React.Fragment>` no lugar.

Você criou seu primeiro componente no React. Se você já conhecia JavaScript em outro contexto deve ter percebido que o que acabamos de fazer no return do render não é javascript. Uma vez que o HTML não está dentro de uma string.

O que foi escrito dentro do *return*, que pode misturar componentes, HTML, e JavaScript é uma outra linguagem (ou melhor, dialeto) chamada JSX. Ele serve para podermos escrever tanto HTML como JavaScript fora de strings e se colocarmos uma tag que tenha o mesmo nome de um componente ele será renderizado onde a tag estiver. Isso lembra um pouco linguagens de lado servidor como PHP ou ASP.

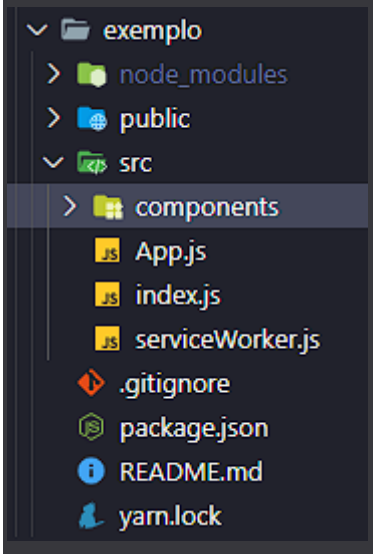
Nela podemos inserir código JavaScript entre chaves contanto que ele produza um retorno. Por exemplo, vamos mostrar a data de hoje (no formato brasileiro):

```
import React from 'react';

class App extends React.Component{

    render(){
        return (
          <>
            <p>Meu primeiro parágrafo em React.</p>
            <p>Meu segundo parágrafo em React.</p>
            <div>
              <pre>Cansei de parágrafos...</pre>
            </div>
            <p>{ new Date().toLocaleDateString("pt-BR") }</p>
          </>
        );
    }
}

export default App;
```

Fizemos um componente estático, que apresenta um HTML na tela. Veremos como deixar os componentes mais interessantes a seguir.

Props

Primeiramente vamos organizar um pouco melhor nosso projeto criando uma pasta dentro de `/src` chamada `components` assim todo componente será criado lá e não solto dentro do `/src`:



Vamos agora arrastar o App.js para lá. Isso vai gerar um erro porque o index.js está procurando pelo App.js na pasta antiga.

Vamos corrigir o arquivo `index.js`:

A linha que dizia:

```
import App from './App';
```

Deve ser alterada para:

```
import App from './components/App';
```

Agora imagine que você quer fazer um componente reutilizável. Por enquanto, vimos componentes com valores fixos, constantes. Não são muitas as possibilidades de reutilização de um componente constante.

Na programação, sempre que desejamos reutilizar algo tornamos o código mais genérico e parametrizamos suas informações. É o que fazemos com funções por exemplo. Uma função que soma 1 e 2 é muito menos útil e reutilizável do que uma que soma qualquer número...

Podemos aplicar a mesma lógica nos componentes do React. Imagine um componente que produz uma caixinha com bordas na tela e tem um título e um texto. Se parametrizarmos o título e o texto poderemos usar essa caixinha em diversas partes de nossa aplicação, até mesmo em outras aplicações!

No React, esses "parâmetros" de um componente são chamados de "props".Podemos receber props pelo construtor do componente.

Vamos ver como fazer isso:

Primeiro crie um arquivo App2.js na pasta components, nele vamos criar um construtor que recebe as props por parâmetro e chama o construtor da super classe.

```
import React from 'react';

class App2 extends React.Component{
  constructor(props){
    super(props);
  }
  render(){
    return ('OK');
  }
}

export default App2;
```

Vamos agora alterar o *render* para usar duas props, uma chamada `title` e uma chamada `text` esses nomes são arbitrários, pode ser o que você desejar.

```
import React from 'react';

class App2 extends React.Component{
  constructor(props){
    super(props);
  }
  render(){
    return(
    <div className="box">
```

```
        <div className="title">{this.props.title}</div>
        <div className="text">{this.props.text}</div>
    </div>);
  }
}

export default App2;
```

Quando for utilizar o componente App2 você pode passar os valores dos props por nome da mesma forma que passaria atributos html.

```
<App2 title='meu título' text='meu texto'/>
```

Se seu texto for grande ou contiver HTML ou qualquer outra coisa que torne-o inconveniente para passar por props (porque não poderia ser um atributo na tag), podemos usar outra abordagem.

```
<div className="text">{this.props.text}</div>
```

Para:

```
<div className="text">{this.props.children}</div>
```

Agora você pode usar seu componente assim:

```
<App2 title="Isso é um teste">
    Lorem ipsum dolor sit amet consectetur adipisicing elit. Nihil officia, quam sed officiis libero repellat voluptate dolores amet molestiae nostrum aperiam inventore veritatis aut quaerat, tenetur laudantium natus? Saepe, minus!
</App2>
```

Qualquer coisa que você colocar entre a abertura e o fechamento das tags do componente serão passados como `props.children` para o componente! Isso inclui HTML, outros componentes, e até javascript, contanto que esteja entre chaves.

Em seguida veremos como criar componentes com estado, ou seja, com valores internos, que quando modificados fazem com que o componente se renderize novamente na tela.

Estado

A palavra estado vem das máquinas de estado, que foram precursoras do computador moderno. A máquina de estado mais simples é um interruptor, nele temos apenas dois estados possíveis: ligado ou desligado. Em um dado momento o interruptor precisa apresentar um, e somente um, desses estados possíveis.

Um exemplo ainda simples, mas um pouco mais complexo, é um semáforo. Ele tem 3 estados possíveis: verde, amarelo e vermelho. Novamente, em um dado momento, um e apenas um desses estados possíveis é apresentado.

Quando falamos de programação moderna, a quantidade de estados é muito mais difícil de calcular do que nas máquinas de estado. Imagine um objeto que tenha apenas um número inteiro dentro dele. Esse exemplo simples já apresenta 2^{32} estados possíveis (inteiros usam 32 bits de memória). Se houver uma string piora mais, os estados tenderiam ao infinito porque a string não tem limite de tamanho. Sendo assim, os estados possíveis são limitados pela capacidade de endereçamento da máquina ou da memória ram disponível, o que for menor.

Sendo assim, não vamos nos preocupar na maioria dos casos em descobrir quantos estados possíveis existem, mas apenas em armazenar ou modificar o estado atual do componente.

Para criar um componente com estado no React é fácil, no construtor faremos um modelo do estado com valores padrão. Usaremos então um método chamado setState() quando quisermos alterar o estado.

Crie um App3.js com o seguinte código:

```
import React from 'react';

class App3 extends React.Component{
  constructor(props){
    super(props);
    this.state = { nome : undefined }
  }
  render(){
    return(
      <p>Olá {this.state.nome}</p>
    );
  }
}

export default App3;
```

O estado só deve ser atribuído diretamente uma vez no construtor. Todas as modificações subsequentes devem ser feitas pelo método `setState()`, pois ele indica ao React que o componente deve se atualizar na tela.

Vamos adicionar um campo de texto para que o usuário possa modificar o valor do nome no estado do componente:

```
import React from 'react';

class App3 extends React.Component{
  constructor(props){
    super(props);
    this.state = { nome : '' }
  }
  render(){
    return(
      <>
        nome: <input type='text' value={this.state.nome}/>
        <p>Olá {this.state.nome}</p>
      </>
    );
  }
}

export default App3;
```

Tente escrever no campo, o que acontece?

O campo não permite que você altere o valor porque vinculamos ele ao valor do estado quando dissemos `value={this.state.nome}`.

Como fazemos então para alterar os valores?

Precisamos vincular o evento de change desse campo ao estado, para isso faremos uma função.

```
import React from 'react';

class App3 extends React.Component{
  constructor(props){
    super(props);
    this.state = { nome : '' }
  }
  changeNome = function(evt){
    this.setState({ nome : evt.target.value});
  }
  render(){
    return(
      <>
        nome: <input type='text' value={this.state.nome} onChange={this.changeNome}/>
        <p>Olá {this.state.nome}</p>
```

```
      </>
    );
  }
}

export default App3;
```

Fizemos uma função que vai receber por parâmetro o objeto `Event` do JavaScript (o mesmo que receberíamos em JavaScript tradicional) e de dentro dele pegamos o target, que é o elemento HTML que gerou o evento.

De dentro do elemento, no caso de campos de texto, o valor do campo está em `value`.

Usamos a função setState e passamos para ela um objeto com a alteração que queremos fazer: no caso, queremos alterar o campo nome do estado para o valor do campo.

Esse código produz um erro, não podemos passar batido por ele. Quando fazemos métodos no React, em componentes tipo classe, sempre temos que fazer o `bind` do `this`.

Javascript tem um problema, funções tem sempre uma variável chamada `this`, mas ela muda de significado de função para função. Esse é uma das maiores dificuldades dessa linguagem.

Precisamos dizer para a função changeNome que o `this` dela deve ser o apontamento para o mesmo `this` no contexto de classe, um "apontamento para si mesmo". Se esquecer esse passo, nenhum método que utilize o `this` para ler/modificar o estado ou as props funcionará.

Uma linha resolve esse problema, basta adicionar ao construtor:

```
this.changeNome = this.changeNome.bind(this);
```

Se isso parece uma gambiarra para você, uma alternativa é usar arrow functions para os métodos, como elas nunca alteram o contexto do `this` elas funcionam nesse contexto sem gerar o problema que tivemos.

Solução 1

```
import React from 'react';

class App3 extends React.Component{
  constructor(props){
    super(props);
    this.state = { nome : '' }
    this.changeNome = this.changeNome.bind(this);
  }
  changeNome = function(evt){
    this.setState({ nome : evt.target.value});
  }
  render(){
    return(
      <>
        nome: <input type='text' value={this.state.nome} onChange={this.changeNome}/>
        <p>Olá {this.state.nome}</p>
      </>
    );
  }
}

export default App3;
```

Solução 2:

```
import React from 'react';

class App3 extends React.Component{
  constructor(props){
    super(props);
    this.state = { nome : '' }
```

```
  }
  changeNome = (evt) => {
    this.setState({ nome : evt.target.value});
  }
  render(){
    return(
      <>
        nome: <input type='text' value={this.state.nome} onChange={this.changeNome}/>
        <p>Olá {this.state.nome}</p>
      </>
    );
  }
}

export default App3;
```

Observe que ao digitar qualquer letra o estado já vai se modificando e todos os elementos HTML que utilizam aquele estado são atualizados imediatamente.

Uma pequena observação sobre o `setState()`: não é necessário passar o objeto completo do estado para o setState, ele é inteligente o suficiente para fazer modificações parciais.

Ou seja se temos um estado como `{ nome : 'teste', idade : 20 }` e fazemos `setState({ idade : 21 })`, apenas a idade é atualizada, o estado não é substituído pelo objeto que passamos perdendo o nome.

Isso é bastante prático!

Em seguida veremos como podemos usar o estado para renderizar condicionalmente coisas diferentes na tela.

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.

Renderização condicional

Vimos que o estado de um componente guarda valores que podem ser usados para serem mostrados na tela ou alterados para que o componente reaja a eventos.

Agora usaremos um valor do estado para renderizar condicionalmente JSX diferentes.

Crie um App4 e vamos ao código:

```
import React from 'react';

class App4 extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      nome: undefined,
      txtNome: ''
    }
  }
  changeNome = (evt) => {
    this.setState({ txtNome: evt.target.value });
  }
  persistTxtNome = () => {
    this.setState({nome : this.state.txtNome});
  }
  render() {
    return (
      <>
      </>
    );
  }
}
```

```
export default App4;
```

Como na aula anterior temos um componente com um estado que contém um nome, agora de valor padrão `undefined`. Também temos um txtNome que gravará alterações em um campo de texto.

O componente contém um método para alterar o nome e um método para pegar o valor de txtNome e colocar em nome.

Limpamos o método render porque é nele que vamos nos focar.

Uma lição muito importante em JSX é que estamos retornando um valor (nosso JSX inteiro). Expressões de lógica de programação que não produzam valores, não podem aparecer dentro do JSX. Isso inclue os condicionais if e else, switch e os laços todos.

No entanto, se desejarmos usar um condicional é possível: Ele deve aparecer antes do return no método render ou podemos usar condicional ternário.

Vamos modificar o nosso método render para ter telas diferentes se tivermos ou não o nome para mostrar:

```
import React from 'react';

class App4 extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      nome: undefined,
      txtNome: ''
    }
  }
  changeTxtNome = (evt) => {
    this.setState({ txtNome: evt.target.value });
  }
  persistTxtNome = () => {
    this.setState({nome : this.state.txtNome});
  }
  render() {
    if(!this.state.nome){
      return (
        <>
          Nome: <input type='text' onChange={this.changeTxtNome}/>
          <button onClick={this.persistTxtNome}>Salvar</button>
        </>
      )
    }
    else{
      return <p>Olá {this.state.nome}</p>
    }
  }
}

export default App4;
```

Observe que pela limitação descrita tivemos que fazer dois returns.

É comum no React, até porque é boa prática evitar múltiplos return quando possível, fazer uma função que renderiza uma versão do componente e outra que renderiza a outra e decidir com ternário qual usar.

Seria assim:

```
import React from 'react';

class App4 extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      nome: undefined,
      txtNome: ''
```

```
    }
  }
  changeTxtNome = (evt) => {
    this.setState({ txtNome: evt.target.value });
  }
  persistTxtNome = () => {
    this.setState({ nome: this.state.txtNome });
  }
  render() {
    const renderForm = () => {
      return (
        <>
          Nome: <input type='text' onChange={this.changeTxtNome} />
          <button onClick={this.persistTxtNome}>Salvar</button>
        </>
      )
    };

    const renderText = () => (<p>Olá {this.state.nome}</p>);

    return !this.state.nome ? renderForm() : renderText();
  }
}

export default App4;
```

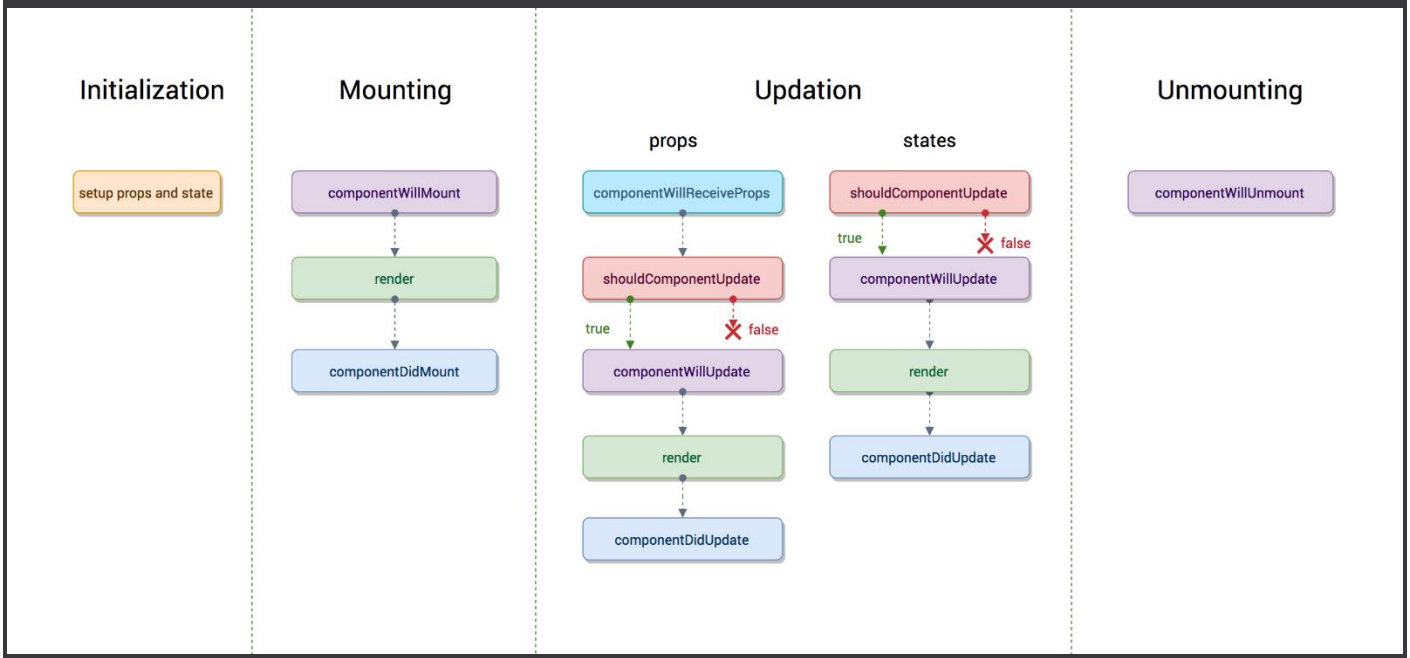
Vale ressaltar nesse exemplo que tivemos que fazer um campo no state para guardar os valores do campo de texto enquanto o botão não era clicado, para só então jogar esse valor no campo nome.

Em seguida veremos ciclo de vida de componentes tipo classe e em que momento podemos fazer chamadas externas para obter dados.

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.

Ciclo de Vida

Componentes do React tem um ciclo de vida que está representado abaixo:



Cada passo do ciclo de vida está representado por um diagrama nessa imagem. Se desejarmos fazer algum processamento durante uma das etapas do ciclo de vida do componente basta adicionar um método com o mesmo nome.

Por exemplo, o momento mais comum de interagir com o ciclo de vida de um componente é o `componentWillMount` que roda pouco antes do componente renderizar na tela.

No entanto, novas versões do React desencorajam o uso desse método e recomendam fazer lógicas pré-construção do componente no construtor da classe ou em `componentDidMount()`.

É muito comum fazer acesso a dados para popular o estado do componente usando esses métodos.

Vamos fazer um exemplo nesse sentido.

Crie um App5 com o mesmo conteúdo do App5 e adicione o método `componentDidMount()`:

```
import React from 'react';

class App5 extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      nome: undefined,
      txtNome: ''
    }
  }

  componentDidMount = () => {

  }

  changeTxtNome = (evt) => {
    this.setState({ txtNome: evt.target.value });
  }
  persistTxtNome = () => {
    this.setState({ nome: this.state.txtNome });
  }
  render() {
    const renderForm = () => {
      return (
        <>
          Nome: <input type='text' onChange={this.changeTxtNome} />
          <button onClick={this.persistTxtNome}>Salvar</button>
        </>
      )
    };

    const renderText = () => (<p>Olá {this.state.nome}</p>);

    return !this.state.nome ? renderForm() : renderText();
  }
}

export default App5;
```

Vamos usar o método para procurar no sessionStorage do navegador se ele tem um valor para o nome.

```
componentDidMount = () => {
const nome = sessionStorage.getItem('nome');
if(nome) this.setState({nome});
}
```

Vamos aproveitar para colocar o nome no sessionStorage quando o usuário clicar no botão:

```
persistTxtNome = () => {
  this.setState({ nome: this.state.txtNome });
  sessionStorage.setItem('nome', this.state.txtNome);
}
```

Se você preencher seu nome, clicar no botão e recarregar a tela, observe que ele se mantém. O `componentDidMount` buscou o valor no sessionStorage do navegador e alterou o estado assim que o componente renderizou.

Você pode usar qualquer um dos métodos da imagem mas temos que fazer algumas ressalvas:

- Já existe uma marcação do método componentWillMount para depreciação, ele não estará disponível nas futuras versões do React
- Da mesma forma que o método componentWillMount, o método componentWillRecieveProps já está marcado para depreciação, ele não estará disponível nas futuras versões do React.
- componentWillUpdate também é um método marcado para depreciação, ele não estará disponível nas futuras versões do React.

Os métodos marcado para depreciação serão substituídos mas continuarão disponíveis pelos nomes:

- UNSAFE_componentWillMount
- UNSAFE_componentWillReceiveProps
- UNSAFE_componentWillUpdate

Esses métodos permitem código blocantes, assim, causam problemas em contextos assíncronos.

O novo método que substitui o componentWillReceiveUpdate será:

- getDerivedStateFromProps

E o método que substituirá componentWillUpdate será o `getSnapshotBeforeUpdate(prevProps, prevState)` ele deverá retornar um objeto que estará disponível no método `componentDidUpdate(prevProps, prevState, snapshot)`.

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.

Renderização de Listas

Muitas vezes temos listas para renderizar usando React. Para fazer isso podemos decidir por duas abordagens quanto às listas:

A primeira é renderizar os items usando HTML simples.

A segunda é criar um componente que será usado como item da lista.

Ambas tem vantagens e desvantagens.

A primeira abordagem é mais simples, mas quando os items são complexos pode ser um pouco confusa e não é candidata ao reaproveitamento.

Já a segunda abordagem é mais complexa, no sentido de termos um ou até dois componentes extras gerenciando a renderização da lista. No entanto, eles pode ser reaproveitados e funcionam melhor do que a primeira abordagem quando os items tem um HTML complexo.

Vamos começar com uma lista simples:

```
import React from 'react';

class Lista extends React.Component{

  constructor(props){
    super(props);

    this.state = {items: ["item1", "item2", "item3", "item4"]}
  }

  render(){ }
```

```
}

export default Lista;
```

A forma mais simples e mais utilizada para renderizar uma lista com React é utilizando a função `map()`.

Ela deve produzir um JSX para cada item da lista.

Vamos completar o método `render()`:

```
  render(){
    return (
      <ul>
        {this.state.items.map(item, index => <li key={index}>item</li>)}
      </ul>
    )
   }
```

É muito importante observar que colocamos uma propriedade key nos `<li></li>` essa propriedade permite que o React distingua cada item, o que permite a atualização de apenas um item caso seu valor mude no state.

Toda vez que criamos items por map precisamos passar o key, que pode ser absolutamente qualquer coisa, contanto que nenhum item tenha um key igual ao do outro.

Nesse caso usamos o índice (posição no array) do item para gerar seu key.

Veja que o map produz um array de JSX. No entanto, o React renderiza esse array como JSX separados. Assim todos os items são renderizados na tela sem a necessidade de fazer joins ou outra estratégia para transformá-lo em string.

Agora vamos pensar em algo um pouco mais complexo que justifique o uso de um componente para os items da lista.

```
import React from 'react';

class Lista extends React.Component{

  constructor(props){
    super(props);

    this.state = {items: [
      {id: 1, nome: 'item1', completo: false},
      {id: 2, nome: 'item2', completo: false},
      {id: 3, nome: 'item3', completo: true},
      {id: 4, nome: 'item4', completo: false},
    ]}
  }

  render(){
    return (
    )
   }
}

export default Lista;
```

Nesse exemplo temos algo mais complexo para renderizar, a lista tem um nome, que aparecerá na tela, um ID que colocaremos em um campo `data-` e tem um completo que indica se ela deve ser tachada ou não.

Vamos então criar um componente para lidar com tudo isso, fazendo com que toda essa complexidade não esteja no componente Lista.

```
class Item extends React.Component{
  constructor(props){
    super(props);
  }
```

```
  render(){
    const textDecoration = this.props.completo ? 'line-through' : 'none'

    return (
      <li data-id={this.props.id} style={{ textDecoration }}>
        {this.props.children}
      </li>
    )
  }
}
```

Agora vamos completar o método `render()` da Lista:

```
render(){
    return (
      <ul>
        {this.state.items.map(item, index => (
          <Item key={item.id} id={item.id} completo={item.completo}>
            {item.nome}
          </Item>
        ))}
      </ul>
    )
   }
```

Observe que aqui também foi usado o key, mas como temos um id nos items esse id foi usado no key para distinguir cada item.

Optamos por passar o conteúdo da lista por children para permitir que seja usado conteúdo HTML ou outros componentes, também passamos por props as informações adicionais que precisam ser colocadas no elemento li.

O resultado é:

```
<ul>
    <li data-id="1" style="text-decoration: none;">item1</li>
    <li data-id="2" style="text-decoration: none;">item2</li>
    <li data-id="3" style="text-decoration: line-through;">item3</li>
    <li data-id="4" style="text-decoration: none;">item4</li>
</ul>
```

Assim exemplificamos as duas abordagens de renderização de listas.

Vale apontar que outras linguagens de programação utilizam laços para fazer a renderização de listas. No React, vale a pena adotar a abordagem do uso do map, pois diferentemente dos laços ele produz um resultado JSX imediatamente.

O código completo do exemplo ficou:

```
import React from 'react';

class Lista extends React.Component{

  constructor(props){
    super(props);

    this.state = [
      {id: 1, nome: 'item1', completo: false},
      {id: 2, nome: 'item2', completo: false},
      {id: 3, nome: 'item3', completo: true},
      {id: 4, nome: 'item4', completo: false},
    ]
  }

  render(){
    return (
      <ul>
        {this.state.map(item => (
          <Item id={item.id} completo={item.completo}>
```

```
            {item.nome}
          </Item>
        ))}
      </ul>
    )
   }
}

class Item extends React.Component{
  constructor(props){
    super(props);
  }

  render(){
    const textDecoration = this.props.completo ? 'line-through' : 'none'

    return (
      <li data-id={this.props.id} style={{ textDecoration }}>
        {this.props.children}
      </li>
    )
  }
}

export default Lista;
```

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.

Componentes Aninhados

Vamos explorar um pouco mais a fundo o uso do children.

Começamos modificando o exemplo anterior para permitir a passagem de items também como filhos:

```
import React from 'react';

class Lista extends React.Component{

  constructor(props){
    super(props);

    this.state = { items: [
      {id: 1, nome: 'item1', completo: false},
      {id: 2, nome: 'item2', completo: false},
      {id: 3, nome: 'item3', completo: true},
      {id: 4, nome: 'item4', completo: false},
    ]}
  }

  render(){
    return (
      <ul>
        {this.props.children}
        {this.state.items.map(item => (
          <Item key={item.id} id={item.id} completo={item.completo}>
            {item.nome}
          </Item>
        ))}
      </ul>
    )
   }
}
```

```
class Item extends React.Component{
  constructor(props){
    super(props);
  }

  render(){
    const textDecoration = this.props.completo ? 'line-through' : 'none'

    return (
      <li data-id={this.props.id} style={{ textDecoration }}>
        {this.props.children}
      </li>
    )
  }
}

export {Lista, Item};
```

Dessa forma, podemos passar os items tanto como filhos para o componente como pegar os que estão no estado e renderizar.

O uso seria, por exemplo:

```
<Lista>
    <Item key={999} id={999} completo={true}>teste</Item>
    <Item key={1000} id={1000} completo={false}>teste2</Item>
</Lista>
```

Dependendo de onde colocamos a linha `this.props.children` podemos colocar os items passados como filhos antes ou depois daqueles que estão no state. Basta colocar a linha antes ou depois do map();

Outra técnica bastante interessante é restringir que os filhos passados para o componente são de um tipo determinado, assim não são passados valores indevidos.

Imagine que alguém fez o seguinte:

```
<Lista>
    Teste1
    Teste2
    <Item key={999} id={999} completo={true}>teste</Item>
    <Item key={1000} id={1000} completo={false}>teste2</Item>
</Lista>
```

Como nosso código está renderiza o Teste1 e Teste2 fora de um `<li></li>` e produz um HTML assim:

```
<ul>
    Teste1 Teste2
    <li data-id="999" style="text-decoration: line-through;">teste</li>
    <li data-id="1000" style="text-decoration: none;">teste2</li>
    <li data-id="1" style="text-decoration: none;">item1</li>
    <li data-id="2" style="text-decoration: none;">item2</li>
    <li data-id="3" style="text-decoration: line-through;">item3</li>
    <li data-id="4" style="text-decoration: none;">item4</li>
</ul>
```

Podemos evitar isso usando um map especial para children do React e restringir quais podem ser renderizados e quais não podem.

Basta trocar a linha:

```
{this.props.children}
```

Por:

```
{React.Children.map(child => child.type == Item ? child : null)}
```

Esse map pode ser utilizado para fazer qualquer restrição ou transformação nos items passados como children do componente.

No exemplo, fizemos a verificação se o filho é uma instância da classe Item, se for retornamos ele, se não for retornamos null.

Além do map, o mesmo objeto `React.Children` tem outros métodos úteis como `forEach`, `count`, `toArray`, que são auto explicativos e `only` que restringe que apenas um filho seja passado e renderizado.

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.

Formulários

Vamos fazer um código com vários exemplos de campos para processar um formulário. O nosso formulário terá um campo de texto para o nome, um *select* para escolher a linguagem de programação preferida dada as opções, um campo *radio* de marcação única se o usuário é programador ou estudante, um checkbox se o usuário dedica 8h semanais aos estudos e por fim uma área de texto para uma bio do usuário.

Já sabemos fazer um input textual de exemplos anterior, então começamos adicionando a tag *form* e com o atributo *onSubmit*, que recebe uma função a ser executada quando o formulário for submetido.

```
import React from 'react';

class Formulario extends React.Component {
  constructor(props) {
      super(props);
      this.state = {nome: '', linguagem: 'JavaScript', tipo: 'programador', dedico:
true, bio: ''};

      this.handleSubmit = (event)=>{
          event.preventDefault();
          console.log(this.state);
      }

      this.changeName = (event)=>{
          this.setState({nome: event.target.value})
      }

  }

  render(){
      return (
          <>
            <form onSubmit={this.handleSubmit}>
                <label>
                    Nome: <input type="text" value={this.state.nome}
onChange={this.changeName} />
                </label>
                <br />
                <input type="submit" value="Salvar"/>
            </form>
          </>
      );
  }

}

export default Formulario;
```

Observe que mesmo estando no contexto React o HTML tentará se comportar como HTML comum ao submeter o formulário. Para evitar esse comportamento, que gera o *refresh* da página, precisamos utilizar o método `preventDefault` de `Event`:

```
event.preventDefault();
```

Para cada novo campo de *input* nós precisamos criar um método para processar as alterações do mesmo.

Vamos adicionar o *input* tipo *select*, no *render* colocamos:

```
<label>
Linguagem favorita
<select value={this.state.linguagem} onChange={this.changeSelect}>
    <option>JavaScript</option>
    <option>Python</option>
    <option>C++</option>
</select>
</label>
<br />
```

No construtor adicionamos o método `this.changeSelect`:

```
this.changeSelect = (event)=>{
    this.setState({linguagem: event.target.value})
}
```

Para o *input* tipo *radio* adicione no *render*:

```
<label>
    Sou:
    <input type="radio" checked={this.state.tipo == 'programador'}
onChange={this.changeRadio} value="programador" /> Programador
    <input type="radio" checked={this.state.tipo == 'estudante'}
onChange={this.changeRadio} value="estudante"/> Estudante
</label>
<br/>
```

Para termos a marcação de uma única opção, o campo *checked* verifica a condição da opção marcada no estado, se for vendadeiro é renderizado como marcado.

No construtor adicionamos o método `this.changeRadio`:

```
this.changeRadio = (event)=>{
    this.setState({tipo: event.target.value})
}
```

Para o checkbox temos a mesma atenção ao campo *checked*:

```
<label>
    <input type="checkbox" checked={this.state.dedico} onChange={this.changeCheckbox}
/> Dedico 8h semanais aos estudos.
    </label>
<br/>
```

Agora atenção para o método `this.changeCheckbox`, diferente dos outros casos não veficamos o *value* desse tipo de *input* para alterar o valor do estado do componente mas sim o *checked*:

```
this.changeCheckbox = (event)=>{
    this.setState({dedico: event.target.checked})
}
```

Por fim para a área de texto usamos a tag *textarea*:

```
<label>
    Bio:
    <textarea cols="50" value={this.state.bio} onChange={this.changeBio} />
</label>
```

O método segue como os demais:

```
this.changeBio = (event)=>{
    this.setState({bio: event.target.value})
}
```

O código completo:

```
import React from 'react';

class Formulario extends React.Component {
  constructor(props) {
      super(props);
      this.state = {nome: '', linguagem: 'JavaScript', tipo: 'programador', dedico:
true, bio: ''};

      this.handleSubmit = (event)=>{
          event.preventDefault();
          console.log(this.state);
      }

      this.changeName = (event)=>{
          this.setState({nome: event.target.value})
      }

      this.changeSelect = (event)=>{
        this.setState({linguagem: event.target.value})
      }

      this.changeRadio = (event)=>{
        this.setState({tipo: event.target.value})
      }

      this.changeCheckbox = (event)=>{
        this.setState({dedico: event.target.checked})
      }

      this.changeBio = (event)=>{
        this.setState({bio: event.target.value})
      }
  }

  render(){
      return (
          <>
            <form onSubmit={this.handleSubmit}>
                <label>
                    Nome: <input type="text" value={this.state.nome}
onChange={this.changeName} />
                </label>
                <br />
                <label>
                    Linguagem favorita
                    <select value={this.state.linguagem} onChange={this.changeSelect}>
                        <option>JavaScript</option>
                        <option>Python</option>
                        <option>C++</option>
                    </select>
                </label>
                <br />
                <label>
                    Sou:
                    <input type="radio" checked={this.state.tipo == 'programador'}
onChange={this.changeRadio} value="programador" /> Programador
                    <input type="radio" checked={this.state.tipo == 'estudante'}
onChange={this.changeRadio} value="estudante"/> Estudante
                </label>
                <br/>
                <label>
```

```
                    <input type="checkbox" checked={this.state.dedico}
onChange={this.changeCheckbox} /> Dedico 8h semanais aos estudos.
                </label>
                <br/>
                <label>
                    Bio:
                    <textarea cols="50" value={this.state.bio}
onChange={this.changeBio} />
                </label>
                <input type="submit" value="Salvar"/>
            </form>
          </>
      );
  }

}

export default Formulario;
```

No exemplo, fizemos diversas mudanças no estado baseadas em eventos do tipo `onChange`, também "sequestramos" a submissão do formulário usando `onSubmit`.

Ele impede que o comportamento padrão do formulário aconteça e nos permite processar o formulário de acordo com o padrão usado no React. Esquecer de invocar esse método faz com que o formulário seja mandado para o servidor e a página recarregue.

Com isso concluímos nosso estudo pelo componentes feitos a partir de classes. Vamos agora ver componentes funcionais.

Obs: Os cursos digitais dos processos seletivos não contêm os exercícios citados pelo instrutor.