GRADUAÇÃO EM ANÁLISE E DESENVOLVIMENTO
PARADIGMAS DE PROGRAMAÇÃO: REACT NATIVE

Aula 02: Conceitos Básicos – *React*
(https://reactjs.org/)

Obs.: *os códigos apresentados nesse documento foram codificados e testados na ferramenta https://codepen.io/ - os códigos exemplo do documento podem ser baixados aqui.*

INTRODUÇÃO

O conteúdo dessa aula tem por objetivo revisar (ou apresentar) conteúdos básicos sobre a biblioteca *React*. Esse conhecimento prévio será necessário a partir da aula 03, quando tais conceitos serão utilizados e aplicações *React Native* serão desenvolvidas. Como essa disciplina é ofertada aos alunos dos últimos anos do curso, considera-se que conceitos prévios de programação, como laço, *array* e objeto, já são conhecidos.

REACT

O *Rect* é uma biblioteca *JavaScript* desenvolvida para criar interfaces para usuários, sendo estruturada via componentes. A ideia central do *React* é possibilitar que os desenvolvedores criem aplicações através da utilização e criação de vários componentes. Essa dinâmica mostra-se interessante porque a codificação criada (diga-se também componentes) pode ser facilmente reutilizada em novos projetos.

CODIFICAÇÃO – CONCEITOS

A seguir são apresentadas codificações básicas que fazem uso da biblioteca React, tendo como objetivo conhecer os conceitos da biblioteca *React* de maneira prática. Todos os códigos utilizados foram implementados e testados no *https://codepen.io/*.

Configurações Iniciais - Importação de Bibliotecas

Para que seja possível utilizar as funcionalidades disponibilizadas pelo *React* é necessário importar para o projeto em desenvolvimento a sua biblioteca. Para isso, dentro do *codepen*, basta acessar a opção *settings* (canto superior direito do navegador), aba JavaScript e adicionar os pacotes *React* e *ReactDOM* (utilizado em aplicações Web). No momento em que essa aula foi montada, as versões eram:

https://cdnjs.cloudflare.com/ajax/libs/react/16.4.0/umd/react.development.js
https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.4.0/umd/react-dom.development.js

Gil Eduardo de Andrade

Hello World

HTML

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

CSS

JS

```
// createElement(elemento, proriedades, filhos);
// <h1>Olá</h1>
const texto = React.createElement("h1", null, "Hello Wolrd");

const div = document.getElementById("app");
// render(componente, container);
ReactDOM.render(texto, div);
```

Quando trabalhamos com *React* precisamos indicar o *container* onde a aplicação será exibida ou renderizada. Para isso, criamos o elemento HTML e damos a ele um identificador, nesse caso “app”. Como explicado anteriormente, o desenvolvimento de aplicações *React* baseia-se, principalmente, na criação de componentes, sendo assim utilizamos o primeiro pacote importado (*react.development.js*) para criar um novo elemento.

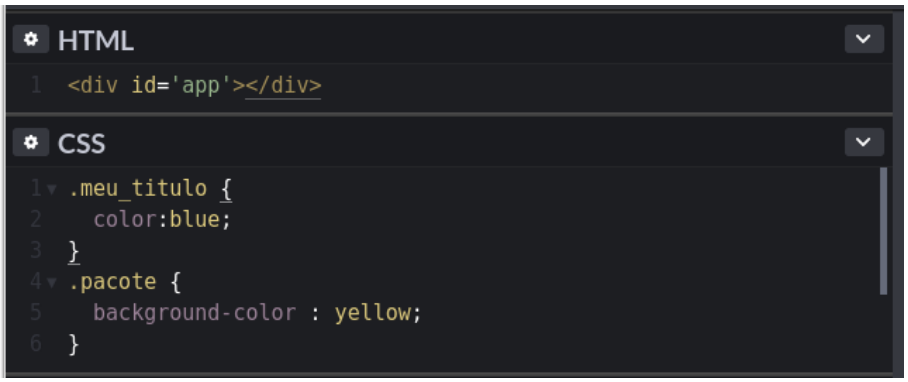
A função ***createElement()*** recebe três parâmetros: o *elemento* que será criado, no caso o exemplo anterior o HTML <h1>; as *propriedades* (estilos, etc) que serão aplicadas a esse elemento, no caso do exemplo *null* (nenhuma); e os *filhos* desse elemento, nesse caso o texto (‘Hello World’) que será exibido pelo <h1>.

Por fim, obtemos – ***getElementById()*** – a referência da responsável por exibir (renderizar) o elemento “texto” que acaba de ser criado, e invocamos a função ***render()*** passando como parâmetros o componente a ser exibido e o container que irá exibi-lo.

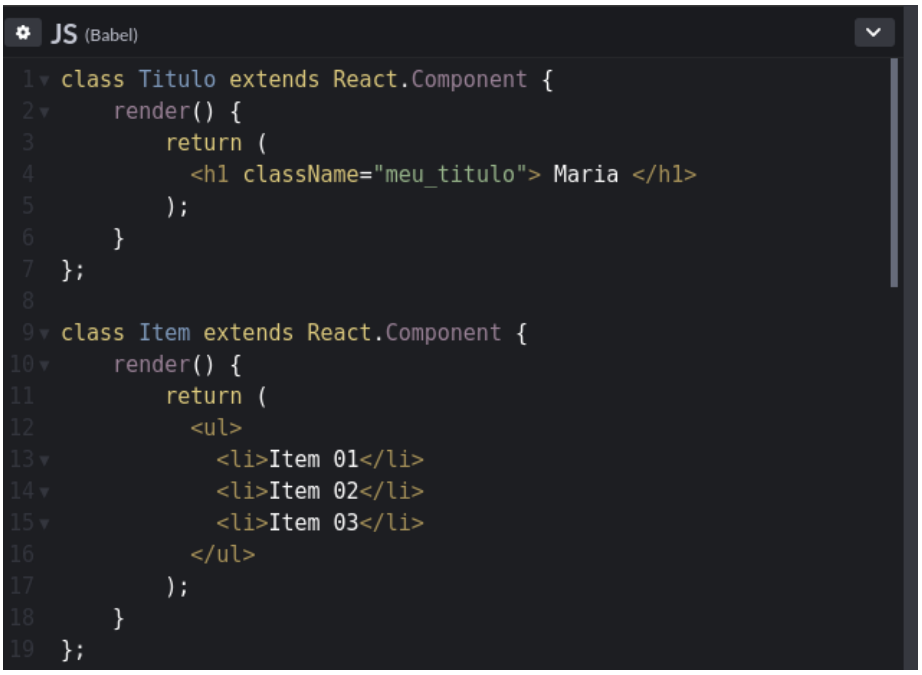
Lista de Itens

(Criando e Renderizando Componentes)

*Para o exemplo a seguir adicionar o **processador JavaScript Babel** – Menu Settings*



Assim como feito anteriormente, criamos e definimos um identificador para o container que irá renderizar a nossa aplicação *React* (codificação HTML). Além disso, também definimos dois novos estilos para aplicarmos aos nossos componentes (codificação CSS).

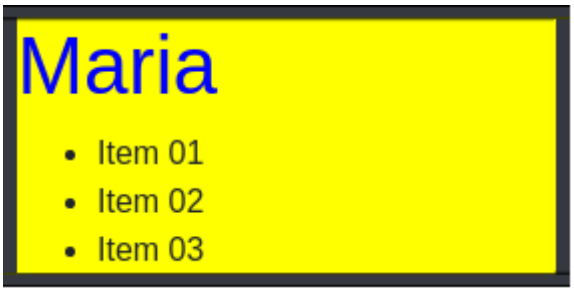


Para o exemplo em questão, foram criados dois componentes *React*: Titulo e Item. Observe que para criação dos componentes definidos uma nova classe e herdamos as funcionalidades da superclasse *Component*, disponibilizada pelo pacote *React*. Ao criamos um novo componente precisamos, obrigatoriamente, implementar o método render, responsável por retornar o conteúdo que desejamos exibir. Perceba que para o elemento HTML <h1> estamos atribuindo o estilo "meu_titulo" (cor da fonte como azul) definido no espaço de

codificação CSS. Para tal utilizamos a propriedade “className”.

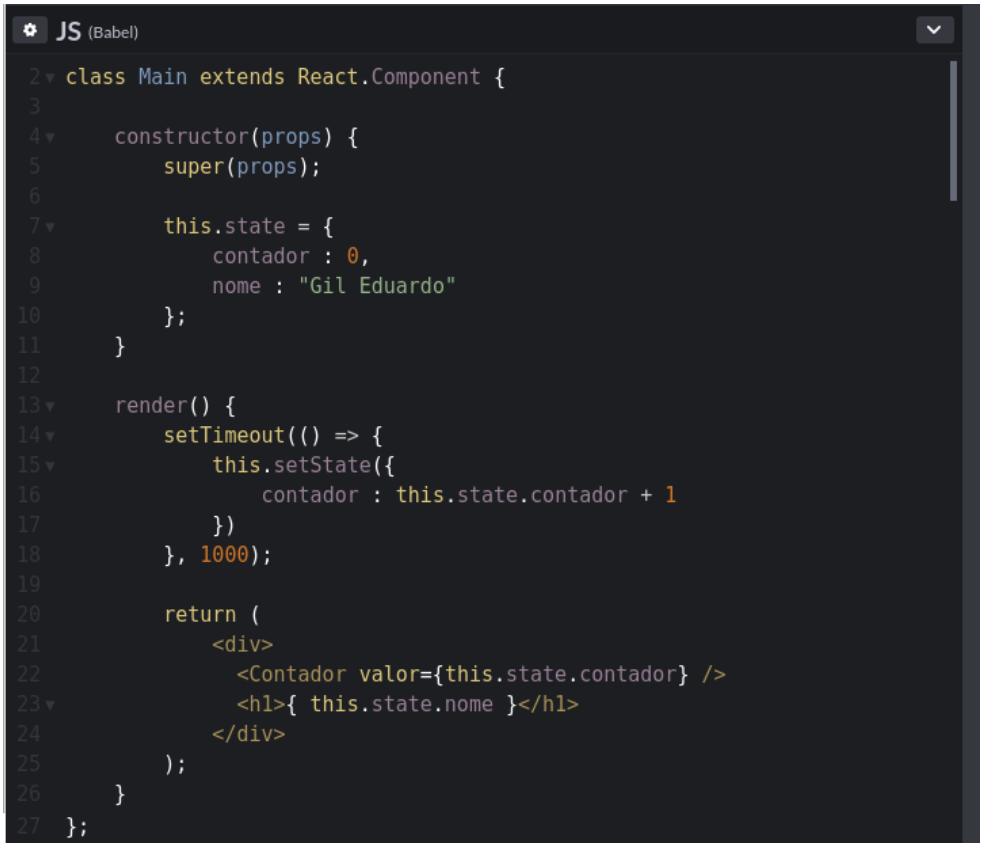
```
JS (Babel)
class Pacote extends React.Component {
    render() {
        return (
            <div className="pacote">
                <Titulo/>
                <Item/>
            </div>
        );
    }
};

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

Além dos dois componentes anteriores (Titulo e Item) foi criado um terceiro componente (“Pacote”) utilizado como container para eles. Essa abordagem foi adotada com intuito de apresentar, de modo prático, o princípio *React* de divisão da aplicação num conjunto de componentes. Essa metodologia facilita a manutenção e reaproveitamento de código. Observe que, se pensarmos num projeto real com vários arquivos fonte (o que faremos futuramente com *React Native*), poderíamos criar um diretório “components” onde cada componente criado ficaria num arquivo JavaScript. Por fim, ao redenderizarmos o componente “Pacote” temos toda a aplicação exibida.



Contador Automático

(Componentes com Estado – Stateful Components)
(Componentes Funcionais – Functional Components)



Assim como visto no exemplo anterior, estamos utilizando (herdando) a superclasse *Component* para criar um novo componente. Contudo, agora estamos trabalhando com dois conceitos novos disponibilizados pelo *React*, o ***props*** e o ***state***. O primeiro permite que dados (propriedades, configurações) sejam passados ao nosso componente quando eles são utilizados. Por exemplo, na linha 22, quando estamos criando uma instância do componente ***<Contador valor={this.state.contador}>*** estamos passando como propriedade ***valor*** o número atual para contagem que está sendo efetuada e que deve ser exibida pelo componente. Esse dado poderá ser obtido dentro da componente via ***props***, que pode ser visto como um *objeto* que contém todas as propriedades (ou dados) passados ao componentes quando o mesmo é criado.

Além do ***props***, também temos o conceito de ***state***, que representa o estado atual do componente, ou seja, ele é utilizado para permitir que o componente armazene dados necessários para sua correta execução, esses componentes são chamados de ***Componentes com Estado*** ou ***Stateful Components***. Quando trabalhamos com os dois conceitos juntos, é preciso implementar um construtor que recebe o ***props*** e passa-o para superclasse *Component*. Após isso, podemos então especificar os dados que serão mantidos via ***state***,

esses dados podem ser vistos como atributos da classe, armazenados via ***state***. Para o exemplo do componente ***<Main>*** temos como dados mantidos pelo ***state***: nome e contador.
A implementação acima ainda apresenta alguns detalhes importantes, a função ***setTimeOut()*** é utilizada para fazer com que a execução da função passada como parâmetro para ela (nesse caso uma *arrow function*) espere o tempo determinado em milisegundos, no caso do exemplo acima 1000 milisegundos ou 1 segundo. A função especificada nada mais faz que incrementar em um o valor do ***state contador***, contudo perceba que para alterarmos dados do ***state*** de um componente devemos utilizar a função ***setState***. Por fim, quando queremos enviar ou exibir o valor de uma variável ou ***state*** com o *React,* como nas linhas 22 e 23, devemos colocá-los entre chaves {}, indicando que trata-se de um valor e não de um texto puro.

```
42  // COMPONENT FUNCTION (aula 21)
43  // Utilizado quando os componentes não possuem estado, ou seja,
44  // quando não precisamos de uma classe com "atributos"
45  const Contador = (props) => (
46      <div>
47          <h2>{ props.valor} </h2>
48      </div>
49  )
50
51
52  ReactDOM.render(
53      <Main/>,
54      document.getElementById('app')
55  );
```

Além dos ***Stateful Components***, também podemos criar componentes que não possuem um estado interno, tendo funções mais simples, como apenas exibir alguma informação recebida, caso do componente ***<Contador>***. Nesses casos torna-se possível trabalharmos com Componentes Funcionais (também chamados de *Stateless Components*), que não precisam ser construídos via definição de uma classe e herança ***React.Component***. Componentes funcionais podem ser criados no formato de funções, já que não possuem um estado interno e recebem dados externos via ***props***. Essa abordagem é mostrada na codificação acima (linhas 45 – 49).

Contador Manual

```
JS (Babel)
1  const Contador = (props) => (
2      <div>
3          <h2>{ props.valor } </h2>
4      </div>
5  )
6
7  const Botao = (props) => (
8      <div>
9          <button onClick={ () => props.onClick() }> { props.label } </button>
10     </div>
11 )
```

Primeiramente foram criados dois componentes funcionais, um ***<Contador>*** com codificação e funcionalidade (exibição) idênticas ao exemplo anterior, e um ***<Botao>*** que utiliza o elemento <Button> HMTL para possibilitar que o usuário interaja com a aplicação. Observe que para deixar o componente genérico, o texto a ser exibido é recebido pelo ***props*** do componente, isso permite que ele seja utilizado várias vezes ao longo da aplicação e para diversas finalidades. Como trata-se de um <Button>, que possui o evento ***onClick***, podemos especificar qual função deve ser executada quando o usuário aciona o componente. Perceba que a função a ser executada pelo componente também é passada via ***props***, ou seja, o *React* permite que um componente primário ou pai (***<Main>***) contenha e controle o modo de funcionamento de um componente secundário ou filho (***<Botao>***).

O segundo componente criado (codificação abaixo) é um componente com estado, que possui como dado o valor atual da contagem que está sendo feita pelo usuário. Observe que a codificação relativa ao ***props*** e ao ***state*** é igual à apresentada anteriormente, sendo necessário codificar o método construtor e enviar o ***props*** recebido a superclasse *Component*. Entretanto, o componente ***<Main>*** também implementa outros dois métodos ***onClickInc()*** e ***onClickDec()***, responsáveis por incrementar e decrementar, respectivamente, o valor da contagem que está sendo efetuado pelo usuário através do *click* nos botões.

Essas funções, apesar de implementadas dentro do componente ***<Main>***, serão executadas quando o componente ***<Botão>*** for acionado, isso é possível porque elas são passadas a ele via ***props***. Em outras palavras, o componente ***<Main>*** é composto por dois componentes ***<Botao>***, e as funcionalidades desses botões são definidas pelo ***<Main>***. Esse paradigma de programação mostra-se muito interessante, pois permite a modularização e reutilização do código (componentes) criado.

```
class Main extends React.Component {

    constructor(props) {
        super(props);

        this.state = {
            contador : 0,
        };
    }

    onClickInc() {
        this.setState({
            contador : this.state.contador + 1
        })
    }

    onClickDec() {
        this.setState({
            contador : this.state.contador - 1
        })
    }

    render() {
        return (
            <div>
              <Botao label="Incrementar" onClick={() => this.onClickInc() } />
              <Contador valor={this.state.contador} />
              <Botao label="Decrementar" onClick={() => this.onClickDec() } />
            </div>
        );
    }
};
```