

ESTRUTURA E CONCEITOS DE UM FRAMEWORK FRONTEND WEB

FRAMEWORK

FRAMEWORKS OFERECE AO
DESENVOLVEDOR CRIAR UM SOFTWARE A
PARTIR DE UMA FUNDAÇÃO. SEU OBJETIVO É
OFERECER AO USUÁRIO A OPORTUNIDADE
DE NÃO PRECISAR COMEÇAR UM TRABALHO
DO ZERO.

BIBLIOTECAS

BIBLIOTECAS SÃO COLEÇÕES DE RECURSOS
USADOS POR PROGRAMA DE COMPUTADOR.
SÃO SUBPROGRAMAS UTILIZADOS NO
DESENVOLVIMENTO DO SOFTWARE.

E qual é a diferença?

- Um framework é como construir um apartamento modelo. Você tem a planta do imóvel e algumas escolhas limitadas em termos de arquitetura e design. No final, a decoração será diferente, mas as unidades são iguais entre os andares .
- Uma biblioteca é como ir a uma loja de móveis. Você já tem um apartamento, mas precisa de ajuda com os móveis. Não é sua intenção fazer uma mesa do zero, por exemplo. Na loja de móveis, você escolhe coisas diferentes que combinam com a sua casa. Você controla as ações.

O que é React.JS ?

React não é *framework*, **é biblioteca!!**

React é uma biblioteca JavaScript que visa simplificar o desenvolvimento de interfaces visuais. Desenvolvida pelo Facebook e entregue ao mundo em 2013, hoje está presente nas aplicações mais utilizadas da internet, como Instagram, Netflix, Spotify, e por aí vai.

JSX = JAVASCRIPT E XML



Parece uma mistura de HTML dentro do JavaScript, mas na realidade é tudo JavaScript.

Na verdade, será traduzido para JS por um *transpilador*.

Mas professor, o que é um
transpilador??

**Qual é a diferença entre um
compilador e um transpilador?**



O transpilador é um tipo de compilador que transporta (ou traduz) código fonte de uma linguagem de programação para outra.



Ao escrever em JSX, estamos codando algo que se parece com HTML, mas na verdade é uma sintaxe para escrever códigos JavaScript utilizando marcações.

SYNTATIC SUGAR



Todo código JSX é JavaScript com açúcar! O termo em inglês é esse mesmo: **Syntactic Sugar**. É um código que, se não existir, não impede o dev de programar com React. **JSX não é necessário para usar React, contudo, não existem motivos para não usar.** Ele aumenta a legibilidade do código, é amplamente documentado e usado pela comunidade. Consideramos intuitivo, pois se parece com HTML. Permite que devs que conhecem HTML entendam seu código com pouco esforço.



A comunidade React desenvolveu ferramentas que facilitam muito o trabalho. Vamos ver as mais comuns:

SANDBOX

É uma aplicação online para colaboração em grupo:

<https://codesandbox.io/>



BOILERPLATE

É um conjunto de ações prontas para minimizar o esforço de criar algo novo.
É uma forma de queimar etapas, reduzir esforço repetitivo.

Ao usar **boilerplates** ao invés de configurar do zero nossos apps React, significa que teremos: **Webpack, Babel, HMR updates etc.**, sem precisar nesse momento entender pra quê serve cada um deles.

DICA: pesquise o que representa cada uma dessas siglas!



CREATE REACT APP

O mais conhecido e usado pela comunidade. Demora muito na instalação, mas é completo. É mantido pelo time React.

```
npx create-react-app nome-em-minúsculas
```

VITE

Muito rápido e atende bem a pequenos projetos, principalmente projetos de aprendizado.

```
npm create vite@latest
```



Vamos abrir o main.jsx e fazer o “Olá mundo” em React.

```
ReactDOM.createRoot(document.getElementById("root")).render("Olá mundo");
```

Uma forma diferente de fazer a mesma coisa, dessa vez usando uma função da biblioteca *React*, ou *ReactDOM*.

Mas saca só: Utilizando *React*, essa será nossa única interação direta com o *DOM*! 😊



Single Page Application = Aplicativo de página única

Em SPAs, toda a interface da aplicação é executada pelo navegador. A geração das páginas fica toda no lado cliente, tanto telas, quanto transições e troca de conteúdo. Do servidor, apenas os dados são recuperados, utilizando APIs.




Em SPA, normalmente é empregada a navegação **sem refresh**. Todo o conteúdo da aplicação se concentra em **uma única página**. Ao clicar em links, botões, ou submeter formulários, o navegador se mantém na 'mesma página HTML'. A navegação seria algo como 'ocultar e exibir' elementos da página, ou criar elementos dinamicamente, sem recorrer a outra página ou mudar de endereço.



No SPA a criação ou modificação de elementos na página é feita utilizando *JavaScript* e funções de manipulação do **DOM**. A partir de agora, em React, vamos nos referir a esse processo como **RENDERIZAÇÃO**.

Várias bibliotecas de SPA utilizam a possibilidade de trabalhar com os **endereços na página local para permitir a navegação sem sair do lugar**. A essa forma de endereçamento chamamos de **rotas**. Rotas, portanto, são caminhos locais (**path**, em inglês), mas que se parecem com uma URL completa.



Para que exista o efeito da navegação, ao fazer a transição entre os conteúdos, é utilizada uma navegação local. Você deve conhecer a navegação local em HTML utilizando âncoras  ou desvios de página.



São **links criados em uma página**, que levam para uma parte específica dentro do seu próprio conteúdo. No endereço da barra do navegador é adicionado um **#**, e, após esse marcador, o nome do desvio desejado.

```
<a href="#playlist">Abrir a Playlist</a>
```



EXEMPLO:

```
http://localhost:3000/#/playlist
```

Nesse exemplo, apenas a página principal (index.html) do site será carregada. Porém, um desvio será feito para a área que estiver nomeada como **/playlist**.

Ainda no exemplo, **/playlist** é uma rota desse site. Não existe no servidor uma página interna **playlist.html**, porém o comportamento de navegação será idêntico.

JSX



Elementos são os menores blocos de construção de aplicativos React.

Podemos chamar as *tags* para o DOM de **objetos JSX**, ou **elementos**, ou nós.

As *tags* são sempre escritas em minúsculas.



A tag sempre precisa ser fechada

```
<p>Olá Serratec!</p>
```

Mesmo que com auto fechamento

```
<hr />
```



Os elementos podem ser atribuídos em variáveis.

```
const saudacao = <h1>Bora Serratec</h1>;
```

Cada instrução retorna somente um único elemento. Mas elementos podem conter outros elementos. Para retornar mais elementos, estes devem ser envolvidos em um elemento pai, seguindo a estrutura parecida com HTML normal.

O elemento pai será considerado o *único elemento*.

```
var bloco = <div><h2>Artista: Linkin Park</h2></div>;
```

ELEMENTO REACT



Elementos em múltiplas linhas devem ser colocados entre parênteses.

```
var bloco = (  
  <div>  
    <h2>Artista: Linkin Park</h2>  
    <hr />  
  </div>  
);
```

ATRIBUTOS



Um elemento pode conter atributos. Se o atributo é *string*, pode ser passado entre aspas.

```
const painel = <div id="painel">Lista de músicas</div>;
```

ATRIBUTOS



Para atributos de outros tipos de dados, devem ser utilizadas expressões React.

Os atributos em React devem ser escritos seguindo a notação camelCase.

```
<input type="text" readOnly />
```



class e **for** são palavras reservadas em JS. Portanto, esses atributos devem ser substituídos por **className** e **htmlFor**

EXPRESSÃO REACT



Uma **expressão React** é uma expressão JavaScript que permite também a utilização de elementos React. As **expressões React** são envolvidas entre chaves.

```
<textarea rows={3} />
```




Um inteiro deve ser passado para um atributo como *expressão*.

Podemos fazer associação com as expressões utilizadas em *template strings*.

EXPRESSÕES COM ELEMENTOS



Expressões podem ser incluídas em qualquer parte da instrução.

```
let nome = "Maria"
```

```
let oi = <p>Oi {nome + "!"}</p>;
```

E expressões podem renderizar outros elementos.

```
let oi = <p>Oi Maria!</p>;
```

```
let texto = <div>{oi}</div>;
```

ARRAYS DE ELEMENTOS



Uma vez que variáveis podem receber elementos React, então, arrays também podem. Uma expressão React pode renderizar um array inteiro de uma vez.

```
const frutas = [<em>🍓</em>, <strong>🍌</strong>, <span>🍉</span>];  
let feira = <p>{frutas}</p>;
```

ELEMENTOS REPETIDOS



Ao trabalhar com elementos que se repetem, é necessário a inclusão do atributo **key**.

Elementos repetidos precisam ter uma chave de identificação única.

```
<ul>  
  <li key={musica.id}>{musica.nome}</li>  
</ul>
```



Manipular eventos em elementos React lembra muito a manipulação de eventos no DOM.

Diferenças:

- Assim como atributos, eventos em React são nomeados usando camelCase.
- A função de manipulação deve ser referenciada em uma expressão.



Em React, estilos **não são *strings***, e sim, **objetos**. Você já reparou que **nada em React é string**. Devem ser passados ao elemento em uma expressão.

```
const estilo = {  
  background: "tomato",  
};  
let quadrado = <div style={estilo}>🍅</div>;
```

COMENTÁRIOS



Não é possível fazer um comentário usando tags JSX. Contudo, expressões React permitem comentários multilinha, e dessa forma podemos comentar trechos do código.

```
{/* <!-- tags podem ser comentadas? dentro de expressões, sim! -->  
  Papel não compila :)  
*/}
```


PROPS



As props são objetos que contêm um conjunto de valores que são passados para os componentes do React durante a sua criação, usando uma convenção de nomenclatura semelhante aos atributos tag do HTML.



Props tornam os componentes reutilizáveis, dando a eles a capacidade de receber dados de seu componente pai. Permitem a comunicação do componente pai com componentes filhos.

Pensando em HTML, *props* são como atributos da tag.



Essa comunicação é *de cima para baixo*, ou seja, pai para filho, top-down.

Imutabilidade: A informação da prop pertence ao pai, e portanto, não pode ser modificada dentro do componente filho.



Em componentes de função, *props* é o objeto recebido como argumento da função. Portanto, *grosso modo*, *props* são parâmetros.

```
function Album(props) {  
  return <h1>Ouvindo: {props.nome}</h1>;  
}
```



Podemos passar quantos atributos forem necessários ao componente. Todas entrarão no mesmo argumento *props*.

```
<Album artista="BTS" nome="Map of the Soul: 7" />
```



A **prop children** é um valor especial do objeto de propriedades, que é preenchida pelo React com o conteúdo "**entre a tag**" do componente pai. É útil para transportar componentes para outros componentes, ou para criar um componente que permite envolver outros componentes.

PROPS ESPECIAIS



```
const Musica = (props) => (  
  <div>  
    <h3>Música</h3>  
    {props.children}  
  </div>  
);
```

```
<Musica>  
  <Artista nome="BTS" />  
  <Album nome="Map of the Soul: 7" />  
</Musica>
```


HOOKS



Permite utilizar estado, ciclo de vida, entre outras funcionalidades, sem a necessidade de escrevermos componentes com classe.



Ao trocar componentes de classe por componentes de função, **existe uma redução** de escrita de código, além da facilidade para compreender melhor como o componente irá se comportar.



Um componente pode perfeitamente permanecer **sem estado**. Componentes de função eram chamados de *stateless*.

```
function Stateless(props) {  
  return <article>Componentes funcionais não possuem estados.</article>;  
}
```

HOOKS



Antes de **os React Hooks** serem introduzidos, os componentes funcionais eram muito limitados no que podiam fazer.

Hooks fornecem flexibilidade para reutilizar o código!
Ele permite que você use o estado e outros recursos do React sem escrever um componente de classe.

Hooks são *ganchos*, funções que se "conectam" no estado React e nos recursos de ciclo de vida dos componentes de função.

Hooks, portanto, são usados *para conectar recursos a componentes*.



De modo geral, são funções Javascript, com algumas regras adicionais:

- Não chame hooks dentro de loops, condições ou funções aninhadas.
- Apenas chame hooks no nível superior.
- Hooks são para componentes funcionais.



React Hooks: o que é e como funciona?

<https://www.zup.com.br/blog/react-hooks>

COMPREENDENDO ESTADO E IMUTABILIDADE



O estado (state) de um componente React tem uma função muito simples e específica. Ele **é uma propriedade do componente onde colocamos dados que, quando mudados, devem causar uma nova renderização.** Simples assim. Se deve causar mudança fica no estado, se não deve, não fica.

STATES



São maneiras de preservar valores entre as renderizações do componente. Em objetos, quando precisamos de estados, criamos um *get e set* ao invés de referenciar a variável diretamente.

Podemos usar a mesma ideia de *getters e setters* para entender os *states*.



Para criar um State é necessário Importar a biblioteca **React** e chamar **useState**.

useState é uma função que guarda no React uma variável de estado. Recebe como argumento o valor de inicialização da variável.



Ela retorna um *array* com dois elementos:

- Na primeira posição, temos o valor atual do *State*.
- Na segunda posição, temos um *setter*, uma função para atualizar o *state*.

useState



```
const state = React.useState("inicial");
```

```
const valor = state[0];
```

```
const setter = state[1];
```




A função **setter** aceita dois tipos de argumentos:

- **Literal:** novo valor do *state*.
- **Callback:** entrega um argumento com o estado anterior, que deve ser utilizado quando a atualização depender do valor prévio.

setState



Literal:

```
<button onClick={() => setter("novo valor")}>Alterar o valor</button>
```



Callback:

```
const manipulador = () =>  
  setter(function callback(anterior) {  
    return "novo valor";  
  });
```



A função **setState** agenda uma atualização para o *state* do componente.

Portanto, é uma operação assíncrona.

Toda vez que o *state muda*, o componente responde renderizando novamente.



O conceito de **imutabilidade** é originado principalmente da programação funcional e orientada a objetos.

Sempre que quisermos fazer alterações em alguns dados (por exemplo, em um objeto ou array), devemos coletar um novo objeto de volta com os dados atualizados, em vez de modificar diretamente o original.

IMUTABILIDADE



É a característica ou qualidade de algo que não se altera.
Em programação orientada a objetos, por exemplo, uma vez que um objeto imutável é criado/instanciado, o mesmo não poderá sofrer alterações no seu estado até o final da sua vida.

IMUTABILIDADE



React impõe a imutabilidade no estado, o que **significa que você não pode alterar diretamente valores com estado.**

Em componentes com hierarquia, isto é, componentes pai e filhos, a informação pertence ao pai, e portanto, não pode ser modificada dentro do componente filho.

COMPREENDENDO O CONCEITO DE COMPONENTIZAÇÃO.

Componentes **são *partes*** da aplicação:

Pequenos pedaços independentes, isolados e reutilizáveis.

O React não separa tecnologias em arquivos diferentes (HTML, CSS e JS em arquivos separados).

Ao invés disso, o React separa conceitos. O papel de um componente é retornar elementos React, que descrevem o que deve aparecer na tela ou em parte dela.

COMPOSIÇÃO DE COMPONENTES



Componentes podem retornar outros componentes.

Conter outros componentes sendo renderizados dentro do atual. Chamamos esse conceito de componentes filhos.



Um componente pode ser redesenhado na tela quantas vezes forem necessárias. A nova renderização irá ocorrer para *reagir* a mudanças e atualizar as informações da tela.



Esse redesenho (ou renderização) é controlado pelo React de forma inteligente: Apenas os componentes afetados são renderizados. E esse processo ocorre primeiramente em um DOM virtual, para só então serem aplicadas no documento.



DOM significa Document Object Model (Modelo de Objeto de Documento).

Ele é uma **API** que nos permite acessar e manipular documentos **HTML** e **XML** válidos.

O **React** implementa um sistema **DOM** independente ao navegador visando performance e compatibilidade entre navegadores.



Elementos são DOM. Componente é um agrupamento pequeno de elementos com significado semântico. Elementos compõe um componente.

Componentes podem retornar qualquer elemento JSX válido, assim como tipos primitivos: strings, numbers, booleans, e null; também arrays e **fragmentos**.

COMPONENTES REACT PODEM SER:



Funcionais:

- O componente é uma função JS tradicional, ou até uma *arrow function*.
- O resultado a ser exibido na tela será o return da função.

COMPONENTES REACT PODEM SER:



Classes:

- O componente de classe precisa ser uma herança de `React.Component`.
- O resultado a ser exibido na tela será o retorno do método `render()`.

CONVENÇÕES PARA COMPONENTES:



- Um componente deve ser iniciado por uma letra maiúscula. Variáveis em pascal-case, como HelloWorld, para ficar claro que um dado elemento JSX é um componente React e não apenas uma tag de HTML comum.
- Manter um arquivo para cada componente - um único componente por arquivo.

CONVENÇÕES PARA COMPONENTES:



- Criar o arquivo com a extensão **.jsx**
- Não tenha medo de dividir componentes em componentes menores.

ENTENDENDO A RENDERIZAÇÃO CONDICIONAL.



Também chamadas de *React conditionals*.
O resultado da renderização será o **retorno**
do componente.
E o retorno de uma função é uma
expressão.



Cuidado para não confundir que pode ser feito dentro de uma expressão com o que pode ser feito em declaração.



Uma função pode ter apenas um retorno. Porém, várias chamadas do **return** podem existir, atendendo a critérios de decisão.



É necessário apenas um if simples, já que após o retorno não são executadas as demais instruções.

Portanto, sem else ou else if:
o conceito da saída antecipada.

EXEMPLO:





Uma expressão React com operador ternário para controlar componentes internos. Será "montado" apenas o componente que atende à condição.

EXEMPLO:





Expressões semelhantes ao condicional com ternário.

Útil quando não existem valores **senão**.

EXEMPLO:





Criar um componente dedicado às decisões. Evita bagunçar o componente principal.

Pode conter as escolhas usando declarações como o **switch**.

EXEMPLO:



GERENCIANDO ROTAS.



Rotas são como páginas internas do site. Entretanto, não são acionados endereços do servidor, toda a renderização fica no próprio frontend.



Graças ao HTML5, o uso do # não é mais necessário para acessar âncoras. Com isso, o endereço na barra do navegador se parece com uma URL normal, mas é uma rota interpretada apenas no frontend.

INSTALANDO O ROUTER



npm install react-router-dom@5.2.0

O sufixo -dom é usado para fazer rotas em uma página Web. Existem rotas *mobile* também. Precisaremos fazer os imports do componentes do pacote usando esse mesmo nome.

DEFINIÇÕES



- Cada página será um **componente React**. A página será criada em um **arquivo jsx** separado (como qualquer componente).
- Normalmente incluímos no **main.jsx** o container para as páginas. Esse container será criado com o componente **<BrowserRouter />**.



- Deve existir um seletor de rotas. Lembra muito a declaração do seletor `switch case`. De fato, o componente será o `<Switch />`.
- Precisamos de um roteador, o "de/para" entre o caminho e a página.
Componente `<Route />`.



- Na hora de criar links, são utilizados componentes `<Link />` ao invés da tag `<a>` convencional.
Isso irá permitir acessar as rotas sem o recarregamento da página.

EXEMPLO:





Ainda de forma parecida com o a declaração switch, o seletor de rotas é sequencial. Isso significa que ele irá resolver a primeira rota que atender ao path digitado.



E assim como o case precisa do break para interromper comandos subsequentes, também precisamos interromper rotas com "nomes parecidos", utilizando o **exact**.



Fazendo uma página 404

Criar uma última rota, abaixo de todas, com path `"*"`.

EXEMPLO:



ROTAS COM PARÂMETROS



Um parâmetro é uma informação variável **passada pela URL**.
É utilizada a notação **dois pontos seguido de identificador**, bem comum em parâmetros.

EXEMPLO:





Expressões semelhantes ao condicional com ternário.

Útil quando não existem valores **senão**.

EXEMPLO:



OBTER O PARÂMETRO NO COMPONENTE



Dentro do componente que recebe o parâmetro, podemos utilizar o hook **useParams** para ter acesso ao valor passado na URL.

EXEMPLO:



ESTILIZANDO COMPONENTES



Possibilita criarmos um aplicativo com funcionalidades muito úteis, e claro para o usuário como utilizá-lo.



É uma biblioteca (extensão, plugin)
para React.

A ideia é criar componentes já com estilos. A estilização é baseada em conceitos CSS tradicional.



Com essa biblioteca, podemos utilizar **CSS dentro do *JavaScript*** para a criação de componentes primitivos (elementos HTML) estilizados.

STYLED COMPONENTS



Styled é uma biblioteca externa.

Para instalar:

npm install styled-components

E para importar:

import styled from "styled-components";

VANTAGENS



- Evita estilizar outros elementos da página "por acidente"
- Reaproveitamento de código e de estilos, pois estão sendo criados componentes.



- Facilidade de manutenção, pois centraliza o estilo no código.
- Permite aplicar estilização condicional

EXEMPLO:





Template strings podem conter expressões.

E o **styled components** usa essa técnica para a passagem de **props** para os estilos!



Por consequência, temos também a possibilidade de criar o CSS com variáveis!

Isso ajuda na redução de "**classes CSS**" e de código aninhado, por exemplo.

EXEMPLO:



EXEMPLO:



REQUISIÇÕES À API BACKEND



Nas requisições uma função javascript vai fazer uma requisição para a api retornar uma determinada informação.



A comunicação entre essas partes do sistema pode ser feita a partir do método HTTP, que é um protocolo de comunicação onde através dele é possível a comunicação entre os diferentes pontos de um sistema.



O protocolo HTTP possui métodos que são utilizados para diferentes finalidades.

Os principais que veremos são GET, POST, PUT e DELETE.



GET tem a função de pegar dados do servidor

EXEMPLO:





POST tem a função de transmitir dados para o servidor.

EXEMPLO:





PUT tem a função de substituir os dados do servidor.

EXEMPLO:



DELETE



DELETE tem a função de apagar dados do servidor.

EXEMPLO:



EXEMPLO:



EXEMPLO:

