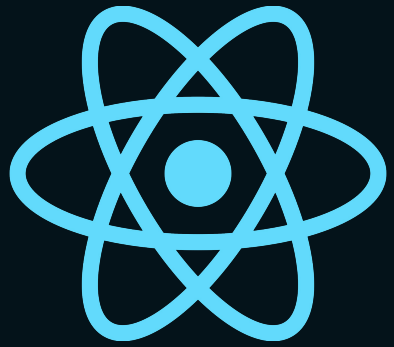
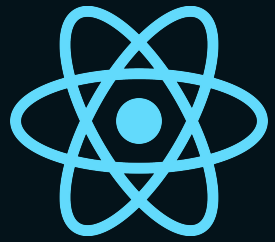


Introdução a React.js

introdução da seção

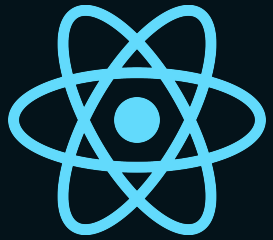
O que é React?

- React é uma **biblioteca JavaScript** para desenvolvimento de aplicações front-end;
- A categoria delas é **SPA** (Single Page Application);
- Podemos criar uma aplicação com React, ou inserir em um projeto já em andamento;
- A sua arquitetura é baseada em **componentes**;
- É mantido pelo **Facebook/Meta**;



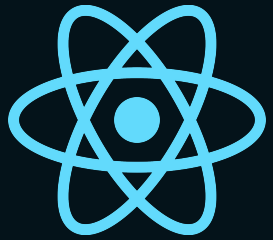
Pré requisitos para rodar React

- Para iniciar uma aplicação React da maneira convencional precisaremos de **Node.js**;
- Através do gerenciador de pacotes **npm**, é possível iniciar projetos;
- Vamos ver como instalar o Node!



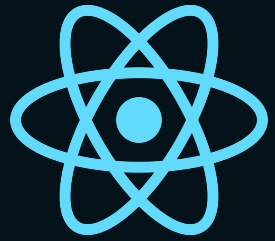
Hello World em React

- Para criar nossas aplicações utilizaremos o **Vite**;
- Antigamente era muito comum utilizar o **create-react-app**, porém ele tem uma pior performance;
- Apenas precisamos digitar no terminal: **npm create vite@latest** e seguir as opções;
- Vamos ver na prática!



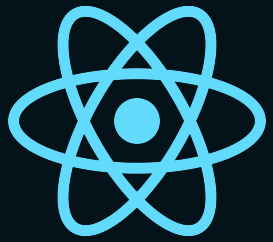
Estrutura base do React

- Há algumas pastas e arquivos muito importantes;
- **node_modules**: dependências do projeto;
- **public**: assets e arquivos estáticos;
- **src**: onde escrevemos o código da aplicação;
- **src/index.js**: arquivo de inicialização da aplicação;
- **src/App.js**: componente principal inicial (pode ser modificado);



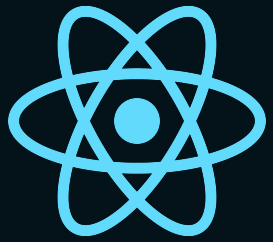
Extensão para React

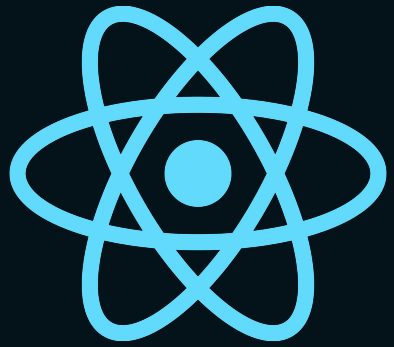
- Há diversas **extensões** interessantes para React no VS Code;
- A principal e mais utilizada é a: **ES7 React snippets**;
- Ela ajuda a criar rapidamente estruturas que utilizamos em todo projeto;
- Vamos baixar!



Preparando o Emmet para o React

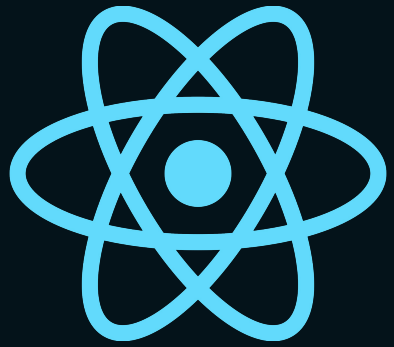
- O **Emmet** é uma extensão nativa do VS Code que ajuda a escrever código mais rápido;
- Mas ela não vem configurada para o React;
- Vamos acessar: **File > Settings > Extensions** e procurar o Emmet;
- Lá precisamos incluir: **javascript - javascriptreact**;
- Vamos configurar!





Introdução a React.js

Conclusão da seção

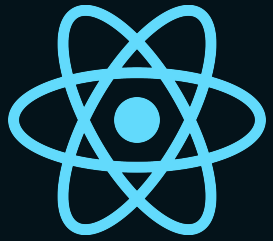


Fundamentos do React.js

Introdução da seção

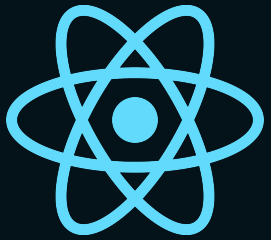
Criando componentes

- Os componentes ficam dentro de uma pasta chamada **components**, que criamos em src;
- Nomeados em CamelCase: **FirstComponent.jsx**;
- A utilização da extensão **.jsx** facilita a formatação para os editores;
- Dentro do componente precisamos **criar e exportar uma função**, que é a lógica dele;
- Vamos ver na prática!



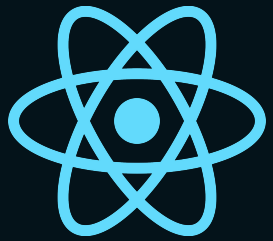
Importando componentes

- Para utilizar e reutilizar um componente é necessário o processo de **importação**;
- A sintaxe é: **import X from ‘./components/X.jsx’** onde X é o nome do componente;
- Para inserir o componente dentro de outro vamos utilizar a sintaxe de tag do HTML com o nome do componente: **<FirstComponent />**
- Vamos ver na prática!



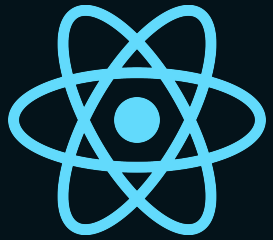
JSX

- **JSX é o HTML do React**, o código interno das funções de componentes, após o return;
- Vamos escrever as nossas tags e importar os outros componentes;
- Há algumas diferenças do HTML, ex: **class = className**;
- Podemos **escrever JavaScript dentro do JSX**;
- O JSX pode ter **apenas um elemento pai**;



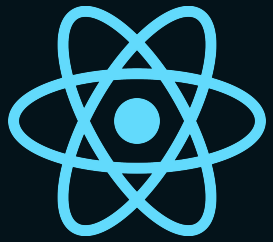
Comentários no Componente

- Há duas formas de inserir comentários em React;
- Podemos utilizar a sintaxe de JS fora e dentro das funções, com: **// Comentário**
- Ou no JSX com: **{ /* Algum comentário */}**
- As chaves são necessárias para executar qualquer instrução de JS;
- Vamos ver na prática!



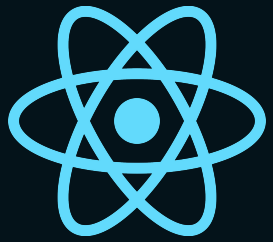
Template Expressions

- **Template Expression** é o recurso que permite a execução de JS no JSX;
- Podemos também inserir variáveis;
- A sintaxe é: **{ 2 + 2 }**
- Tudo que vai entre as chaves é entendido e executado como JavaScript;
- Vamos ver na prática!



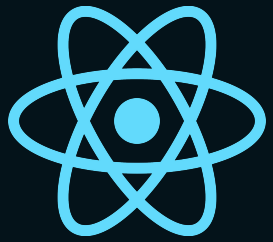
Hierarquia de componentes

- Os componentes podem ser **reutilizados** em outros componentes;
- Podemos montar também uma **hierarquia**, onde um componente é pai do outro;
- E ao importar o componente pai, todos os outros vem juntos;
- Vamos ver na prática!



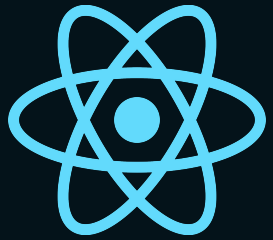
Evento de click

- Os eventos são essenciais para programar apps de front-end, vimos isso em **DOM**;
- **Em React temos os mesmos eventos**, só que de forma simplificada;
- Por exemplo: com **onClick**, conseguimos disparar um evento que ativa uma função ao clicar em um elemento;
- Vamos ver na prática!



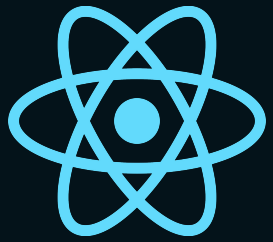
Funções nos eventos

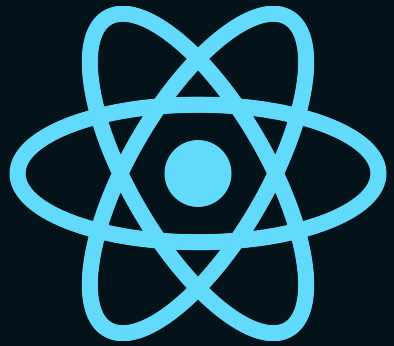
- Quando temos **lógicas complexas**, é mais indicado criar uma função para o evento;
- Isso vai separar as responsabilidades, e deixar nosso código mais de dar **manutenção**;
- Vamos ver na prática!



Funções de renderização

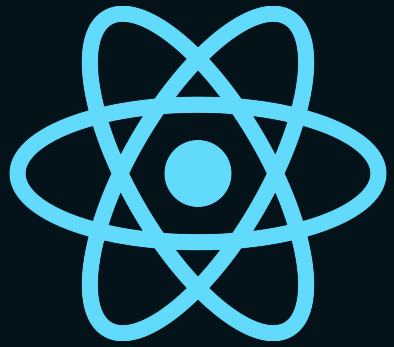
- Podemos criar funções nos componentes que **retornam JSX**;
- Isso pode ser utilizada para uma **renderização condicional**, por exemplo;
- Fazendo que o JSX varie dependendo do resultado da operação;
- Vamos ver na prática!





Fundamentos do React.js

Conclusão da seção

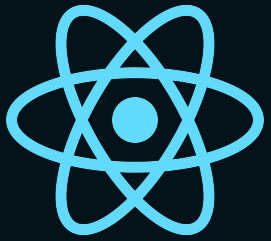


Avançando no React.js

Introdução da seção

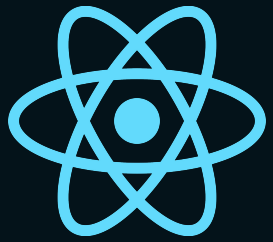
Imagens no React

- As imagens do projeto podem ficar na pasta **public**;
- Estando lá, elas **podem ser utilizadas diretamente no projeto**;
- A pasta public fica linkada com a src, exemplo: **“/imagem.png”**
- Vamos ver na prática!



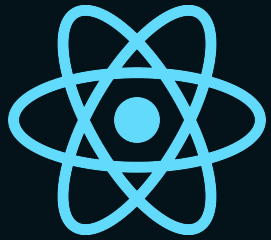
Imagens em assets

- Outro local comum de colocar as imagens em um projeto em React é na pasta **assets**;
- Em assets **precisamos importar a imagem**, como se fosse um componente;
- Estas duas abordagens são muito utilizadas;
- Vamos ver na prática!



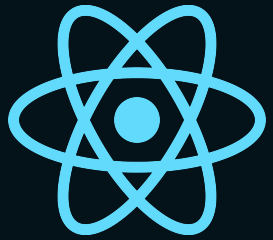
O que são hooks?

- Recursos do React que tem diversas funções, podemos criar os nossos também;
- Exemplo: **guardar e alterar o estado de algum dado**;
- **Os hooks precisam ser importados**, e sempre começam com a palavra **use**;
- Alguns bem utilizados são: **useState**, **useEffect**;
- Os hooks que nós criamos são chamados de custom hooks;
- Geralmente toda a aplicação usa pelo menos um hook;



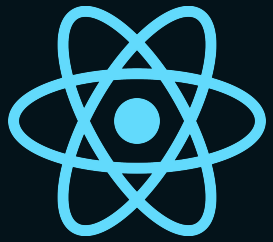
hook: useState

- O **useState** é um dos mais utilizados;
- Podemos **gerenciar o estado de um ou mais dados**, é como se fosse um getter/setter;
- Utilizamos este hook pois as **variáveis não funcionam como esperado**, elas não re-renderizam o componente;
- Para guardar um dado vamos utilizar **setNomeDoDado**;
- Vamos ver na prática!



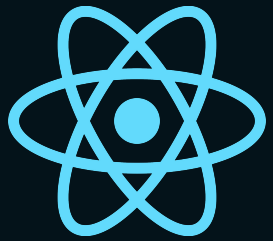
Renderização de lista

- Dados do **tipo array** são muito comuns em aplicações;
- Geralmente recebemos um **array de objetos**, e precisamos iterar nele e exibir os elementos;
- O método **map** fará a iteração;
- É possível inserir **JSX na execução**;
- Vamos ver na prática!



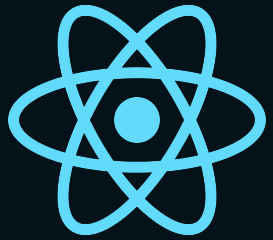
A propriedade key

- Iterar uma lista sem a **propriedade key**, gera um erro no console;
- O React precisa de uma **chave única** para cada elemento;
- Isso serve para ajudar a **renderização do componente**;
- O React utiliza isso para manipulação dos itens;
- Vamos ver na prática!



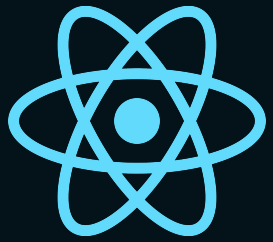
Previous state

- **Previous state** é um recurso do hook useState;
- Podemos pegar o **valor original dos dados**, e fazer alguma alteração;
- Muito utilizado em listas, pois pegamos o valor antigo e o modificamos;
- **O primeiro argumento do set** sempre é o previous state;
- Vamos ver na prática!



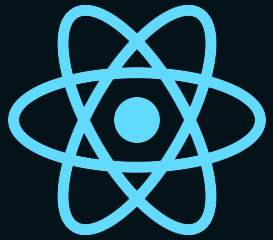
Renderização condicional

- **Renderização condicional** é quando parte do template é exibido por meio de uma condição;
- Que é simplesmente um **if no JSX**;
- Utilização: quando usuário está autenticado/não autenticado;
- Vamos ver na prática!



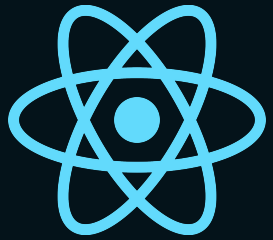
Adicionando um else

- A renderização condicional pode conter um **else** também;
- A estrutura é igual a do **if ternário**;
- Fica desta forma: condição ? execução1 : execução2
- Vamos ver na prática!



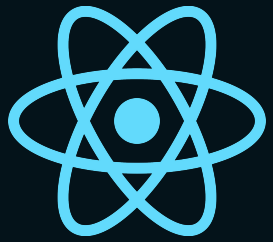
Props

- **Props** é um recurso fundamental do React;
- Permite a **passagem de dados** de um componente pai para um componente filho;
- Será útil para quando houver dados vindo de um banco de dados;
- As props vem em um **objeto no argumento da função** do componente;
- Vamos ver na prática!



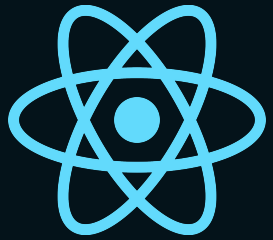
Desestruturando props

- Os componentes geralmente tem mais de uma props;
- Para facilitar o uso delas, podemos **desestruturar no parâmetro da função** do componente;
- Assim o objeto props vira o nome de cada propriedade, então não precisamos acessá-lo;
- Desta maneira: **MyComponent({name, age})**
- Utilizamos então name, em vez de props.age;
- Vamos ver na prática!



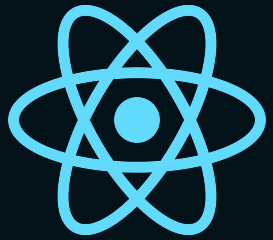
Reutilização de componentes

- Com o auxílio das props, **reutilizar componentes faz mais sentido**;
- Se temos 1000 dados de carros, **podemos representá-los com apenas um componente** repetido n vezes;
- Isso torna o código padronizado, e facilita a manutenção;
- Vamos ver na prática!



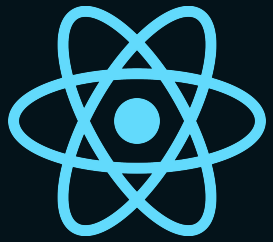
Reutilização com loop

- Os arrays podem ter muitos itens, e as vezes nem sabemos ao certo sua quantidade total;
- Então o correto é utilizar uma **estrutura de loop**, para poder percorrer os itens;
- Com isso conciliamos alguns conceitos aprendidos: **renderização de lista**, **reaproveitamento de componentes** e **props**;
- Vamos ver na prática!



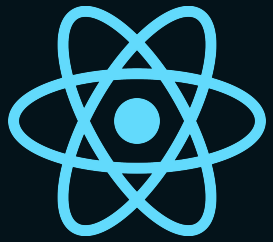
React Fragments

- Os **Fragments** são interessantes para quando há mais de um elemento pai no componente;
- Ou não queremos incluir HTML desnecessário no elemento pai, **não alterando sua estrutura**;
- A sintaxe é: **<> ... </>**
- Vamos ver na prática!



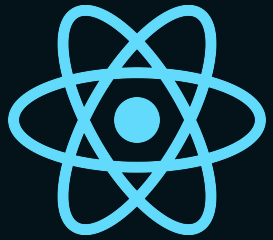
Children prop

- **Children prop** é utilizado quando um componente precisa ter JSX dentro dele;
- Porém o JSX vem do componente pai;
- Então o componente age como um **container**, abrigando esse JSX;
- E children entra como uma **prop do componente**;
- Vamos ver na prática!



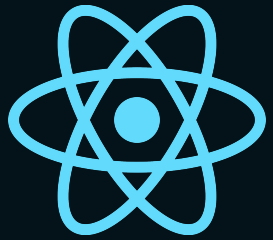
Funções em prop

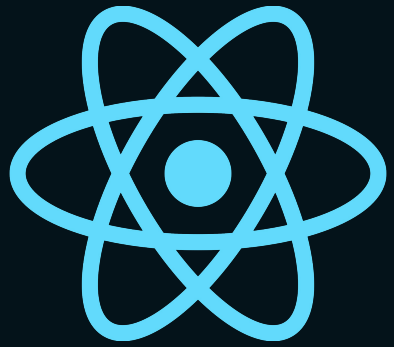
- Podemos passar **funções através de props**;
- Basta criar a função no componente pai, e enviar como prop;
- No componente filho, podemos utilizar para algum **evento**;
- Vamos ver na prática!



Elevação de state

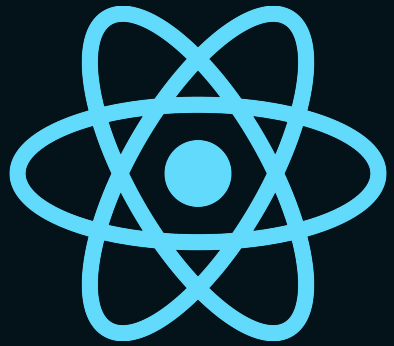
- Elevação de state ou **state lift**, é quando um valor é levado do componente filho para o pai;
- Geralmente temos um componente que usa o state e outro que o altera;
- Então **o componente pai vai gerenciar os valores** e passar para os filhos as alterações;
- Vamos ver na prática!





Avançando no React.js

Conclusão da seção

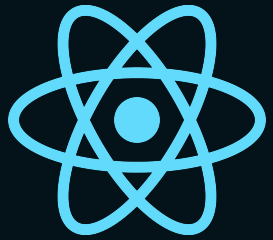


CSS no React.js

Introdução da seção

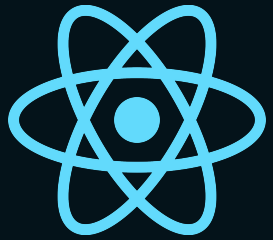
CSS Global

- O CSS global é utilizado para **aplicar estilos a todos elementos** do projeto;
- Utilizamos o arquivo **index.css** para isso, ele está na pasta src;
- Vamos ver na prática!



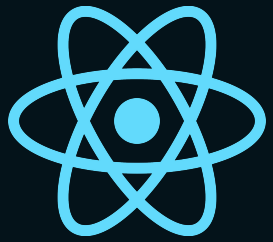
CSS de Componente

- O **CSS de componente** é utilizado apenas em um componente específico;
- Geralmente o arquivo é criado com o **mesmo nome do componente**;
- Lembre-se: ele **não é scoped**, ou seja, pode vazar para outros elementos do projeto;
- Vamos ver na prática!



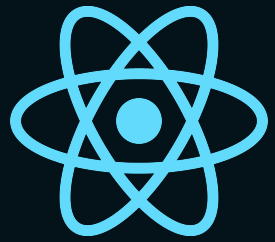
Inline style

- O **inline style** do React é igual ao do CSS;
- Por meio do **atributo style**, aplicamos regras de CSS diretamente a um elemento;
- As outras abordagens são mais interessantes que essa por questões de manutenção do código;
- Vamos ver na prática!



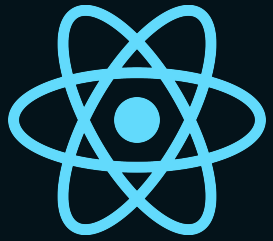
Inline style Dinâmico

- O **CSS dinâmico inline** consiste em uma técnica de aplicação de estilo por condição;
- Teremos o atributo inserido em um **if ternário**;
- Dependendo da condição e do resultado dela, um estilo diferente pode ser exibido;
- Vamos ver na prática!



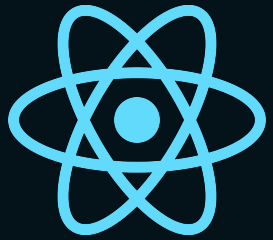
Classes dinâmicas

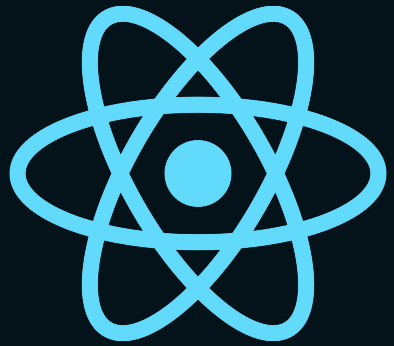
- Podemos também aplicar uma lógica para **adicionar classes a um elemento**;
- Utilizamos o **if ternário**;
- Essa abordagem é mais interessante que o CSS inline, pois o conteúdo da classe está no arquivo de CSS;
- Vamos ver na prática!



CSS Modules

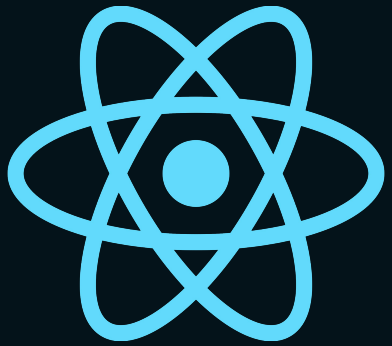
- CSS Modules permite deixar o CSS **scoped**;
- Ou seja, ele só funciona para o componente em questão;
- O nome do arquivo fica: **Component.module.css**;
- É necessário fazer importação também;
- Vamos ver na prática!





CSS no React.js

Conclusão da seção

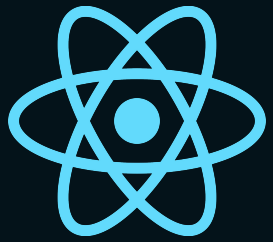


Formulários e React

Introdução da seção

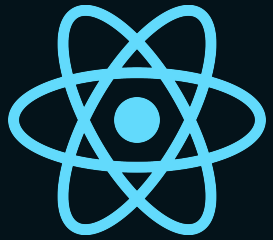
Formulários e React

- Para envio de dados é necessário um formulário, em React também utilizamos a **tag form**;
- As labels tem o atributo for alterado para **htmlFor**, que conta com o name do input;
- **Não utilizamos action**, o envio deve ser feito pelo JavaScript, de forma assíncrona;
- Vamos ver na prática!



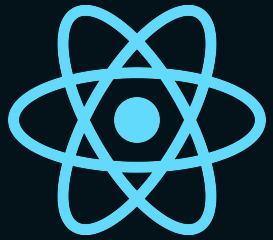
Label envolvendo input

- Em React um padrão muito utilizado é a label ser o elemento **pai do input;**
- O atributo for é opcional nesta abordagem;
- Simplifica o HTML e permanece a **semântica**;
- Vamos ver na prática!



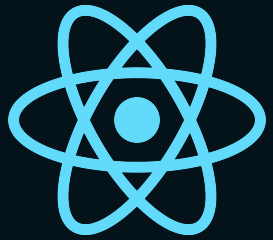
Manipulando valores

- Para manipular os valores de um formulários vamos utilizar o hook **useState**;
- Ou seja, armazenamos o valor com o **set**;
- O evento que vai nos inputs é o **onChange**, e nele teremos a função de alteração;
- Vamos ver na prática!



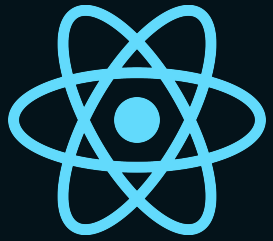
Simplificando a manipulação

- Quando temos diversos inputs no form, podemos simplificar a manipulação;
- Criamos uma **função inline dentro do onChange** e trocamos o valor do dado;
- Vamos ver na prática!



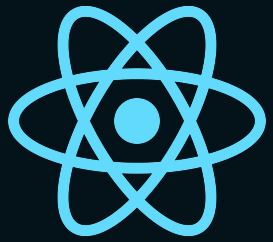
Envio de formulário

- Para enviar formulários utilizamos o evento **onSubmit**;
- Podemos executar uma função, assim como nos inputs;
- Temos que parar o envio do formulário com o **preventDefault**;
- E nesta função é que fazemos **validações** de dados;
- Vamos ver na prática!



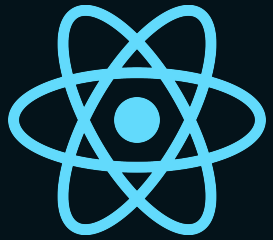
Controlled inputs

- Com o **Controlled input** podemos atribuir valores pré-existentes aos inputs dos forms;
- Precisamos igualar o atributo **value** ao state;
- E também fazer uma lógica que entrega uma **string vazia**, se não houver valor;
- Vamos ver na prática!



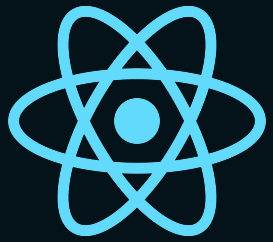
Limpando formulários

- Com o controlled inputs limpar o form fica simples;
- Basta atribuir **valores vazios aos states**;
- Geralmente isso é feito após o envio, para restar o formulário;
- Vamos ver na prática!



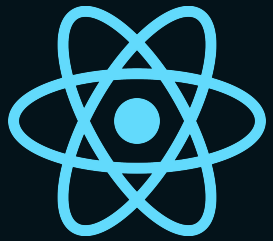
Input de Textarea

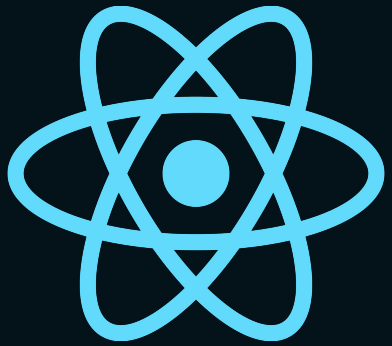
- O **textarea** pode ser aplicado como um **input normal** de texto;
- O atributo **value** pode ser utilizado para mudar o texto inicial;
- E com **onChange** mudamos o seu state;
- Vamos ver na prática!



Input de Select

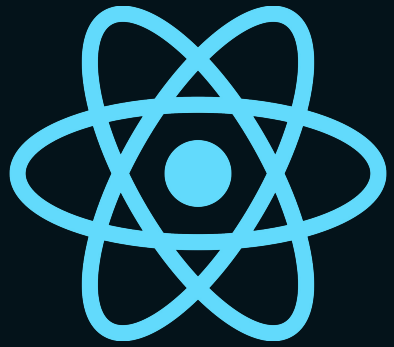
- O **select** é muito semelhante aos outros inputs;
- O evento **onChange** pode mudar o valor do seu state;
- E o **value** deve ser atribuído a uma das options;
- Vamos ver na prática!





Formulários e React

Conclusão da seção

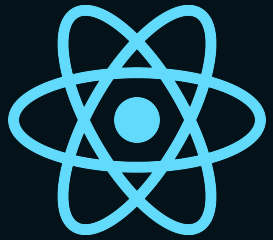


Requisições HTTP e React

Introdução da seção

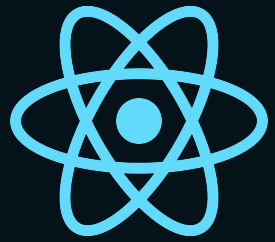
JSON server

- O **JSON server** é um pacote npm;
- Ele **simula uma API**, ou seja, podemos fazer requisições HTTP;
- Vamos integrar este pacote ao React;
- Este é o treino que faremos para APIs reais, que construíremos ao longo dos cursos;
- Isso facilita nossos estudos por **não precisar de um back-end**;
- Vamos criar um projeto e instalar o JSON server!



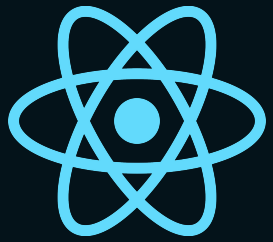
A importância do useEffect

- O **useEffect** faz com que seja possível controlar a execução de uma ação;
- Isso é interessante pois se não o utilizamos recursos podem ser re-executados a cada re-renderização;
- O componente é **re-renderizado a cada mudança**;
- O useEffect possui um **array de dependências** que coordena o que permite a execução do código;
- O useEffect é **muito comum** nas requisições HTTP;



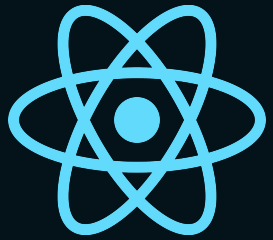
Resgatando dados com React

- Para resgatar dados de uma API temos um procedimento no React;
- Usar **useState** para salvar dados;
- Utilizar **useEffect** para chamar a API apenas quando necessário;
- Realizar a requisição da API com alguma ferramenta, **Axios ou Fetch API**;
- Vamos ver na prática!



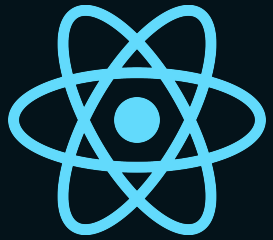
Adicionando dados

- Para adicionar dados via API vamos precisar dos inputs preenchendo os **useStates**;
- Reunimos os dados em uma função, que é disparada no evento de **onSubmit**;
- O verbo HTTP que utilizaremos é o **POST**;
- O processo é parecido com o resgate de dados;
- Vamos ver na prática!



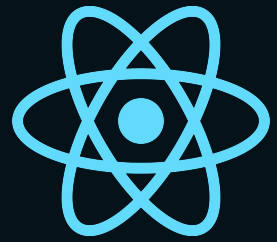
Adicionando dados

- Para adicionar dados via API vamos precisar dos inputs preenchendo os **useStates**;
- Reunimos os dados em uma função, que é disparada no evento de **onSubmit**;
- O verbo HTTP que utilizaremos é o **POST**;
- O processo é parecido com o resgate de dados;
- Vamos ver na prática!



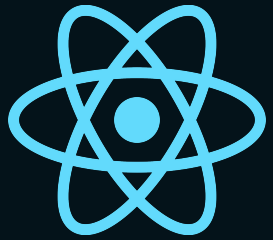
Carregamento dinâmico de dados

- Se a requisição obter êxito, **podemos adicionar no front um novo item a lista**;
- Já temos a informação dele e **não precisamos fazer outra requisição HTTP**;
- Isso deixa nossos projeto mais performático;
- Vamos ver na prática!



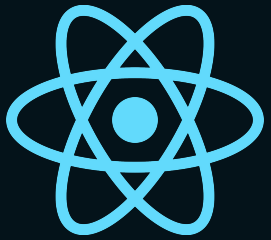
Custom hook para o fetch

- É normal separar as responsabilidades nos componentes;
- Ou seja, termos a função de requisição entre **outro arquivo**;
- Podemos criar o nosso próprio hook para isso;
- Isso é chamado de **custom hook**;
- A pasta geralmente utilizada é a **hooks**;
- Vamos ver na prática!



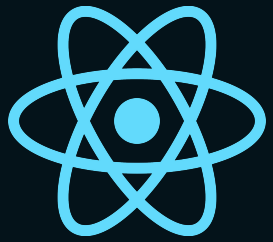
Refatorando o POST

- É possível reutilizar o hook para fazer o **POST**;
- **Vamos criar um useEffect** que mapeia uma outra mudança de estado;
- Após ela ocorrer, adicionamos o produto;
- **Nem sempre reutilizar um hook** para várias ações é a melhor estratégia;
- Vamos ver na prática!



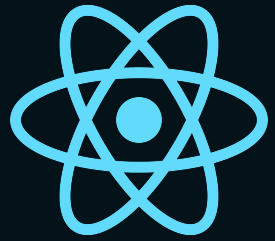
Estado de loading

- Quando fazemos requisições HTTP é normal que a resposta **demora um pouco a chegar;**
- Neste intervalo inserimos um **elemento de loading**;
- É possível inserir no nosso hook esta abordagem;
- Vamos ver na prática!



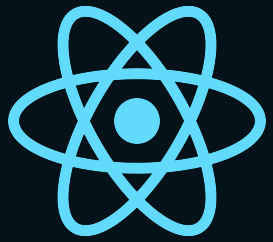
Estado de loading no POST

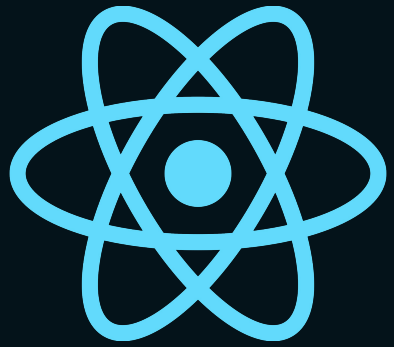
- Podemos bloquear ações **enquanto a requisição ocorre**;
- Isso é interessante para evitar **duplicação de eventos**;
- Podemos identificar um POST ocorrendo, e bloquear o input de envio;
- Vamos ver na prática!



Tratando erros

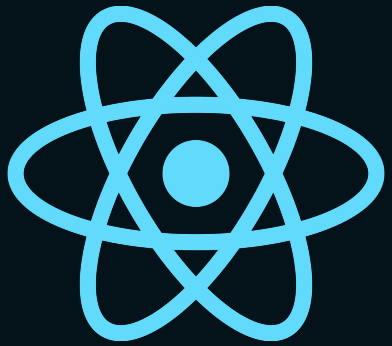
- Podemos tratar erros das requisições com **blocos try catch**;
- **É possível pegar os dados do erro**, e utilizar a mensagem para exibir algo na tela;
- Desta maneira é possível prever erros em todos os cenários do nosso app (resgate, envio, erro);
- Vamos ver na prática!





Requisições HTTP e React

Conclusão da seção

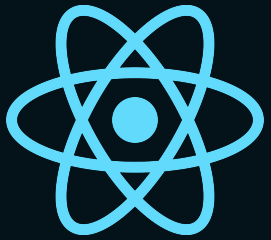


React Router

Introdução da seção

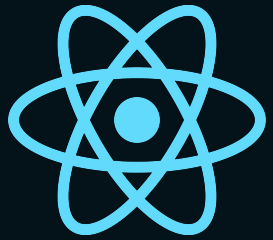
O que é React Router?

- **React Router** é o pacote mais utilizado para criar rotas em uma aplicação React;
- **Cada rota é uma página**, rota é a nomenclatura utilizada;
- Ou seja, permite nosso app SPA ter múltiplas páginas;
- Precisamos instalar e configurar no projeto;
- Há diversas funcionalidades no React Router: **redirecionamento**, **rotas aninhadas**, **404** e etc;
- Vamos criar o projeto e instalá-lo!



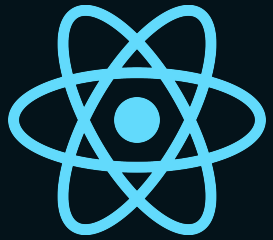
Configurando o React Router

- Para **configurar o React Router** vamos utilizar principalmente o arquivo **main.jsx**;
- Precisamos importar os componentes: **createBrowserRouter**, **RouterProvider**, **Route**;
- Eles serão utilizados na configuração e ao longo do projeto;
- Vamos ver na prática!



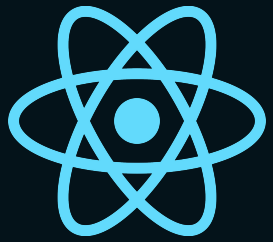
Página de erro / 404

- Podemos criar facilmente uma **página de erro**;
- Precisamos criar um componente, que será a página, geralmente o nome é ErrorPage;
- Depois vamos utilizar o hook **useRouteError** para obter as informações do erro;
- Por último configurar a propriedade **errorElement** em main.jsx como o componente criado;
- Vamos ver na prática!



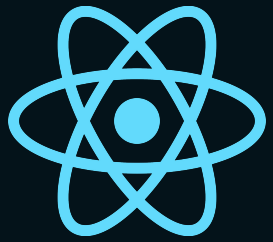
Criando e configurando páginas

- Primeiramente vamos criar o **componente da página**;
- Depois basta inserir um novo objeto em main.jsx.
- Ele deve conter:
  - **path**: caminho para acessar a página;
  - **element**: componente;
- Vamos ver na prática!



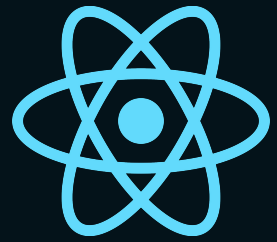
Criando componente base

- O **Outlet** é um componente que nos permite reaproveitar a estrutura das páginas;
- Podemos definir que um componente base seja esta **estrutura**;
- E todas páginas ficam dentro dele;
- As configurações de páginas devem ser feitas na propriedade **children** em main.jsx;
- Vamos ver na prática!



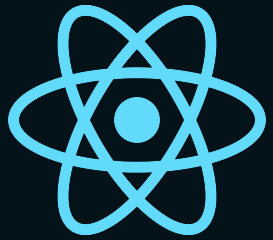
Criando links entre páginas

- Para criar links vamos utilizar o componente **Link**;
- Ele é configurado com a propriedade **to**, que leva a URL de destino;
- Isso permite uma mudança de páginas sem reload;
- Vamos ver na prática!



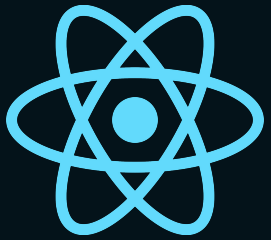
Carregando dados

- Nesta aula vamos utilizar nosso hook **useFetch** para exibir os produtos na Home;
- Isso nos dará possibilidade de explorar **outros recursos do React Router**;
- E também a rever os conceitos aprendidos em requisições de HTTP;
- Vamos ver na prática!



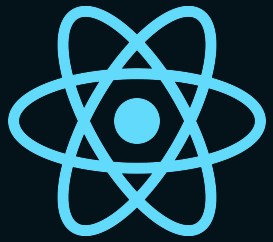
Carregando dados individuais

- O recurso de carregar rotas individuais é chamado de **rota dinâmica**;
- Ou seja, como temos vários produtos a URL de cada um vai variar, dependendo de alguma característica, que geralmente é o id;
- O formato de path é: **/produt/:id**
- Vamos ver na prática!



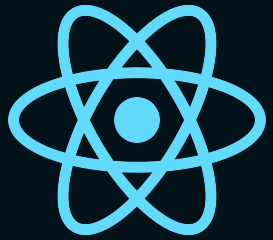
Rotas aninhadas

- As rotas aninhadas ou **nested routes**, são estruturas mais complexas;
- Onde combinamos rotas dinâmicas e criamos uma estrutura maior para acessar a página;
- Exemplo: **/products/:id/info**
- Vamos ver na prática!



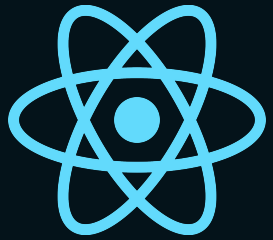
Link ativo

- Para identificar links ativos utilizamos o componente **NavLink** em vez de Link;
- Há uma propriedade isActive que pode **aplicar estilos diferenciados para este link**;
- Vamos ver na prática!



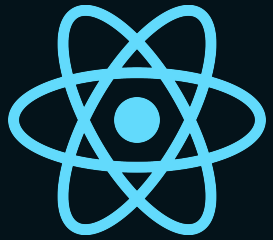
Search Params

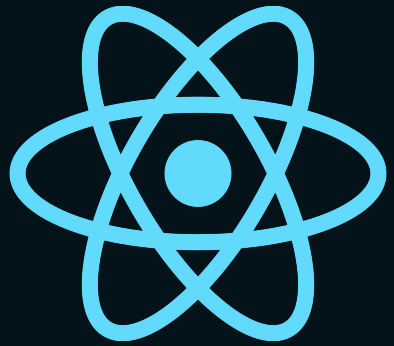
- O recurso de Search Params nos permite pegar **informações da URL**;
- Ele é muito interessante para fazer funcionalidades de busca em um site;
- O hook utilizado para resgatar estes dados é o **useSearchParams**;
- Vamos ver na prática!



Redirect

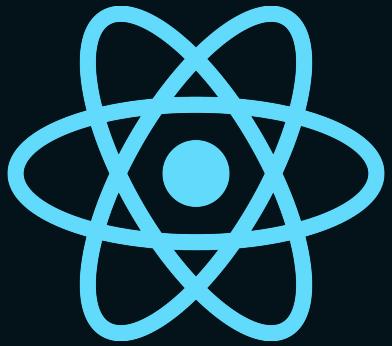
- Podemos criar um redirecionamento de páginas com o componente **Navigate**;
- Exemplo: uma URL que deixa de existir, mas queremos redirecionar o usuário para outra;
- A configuração é feita no próprio **main.jsx**;
- Vamos ver na prática!





React Router

Conclusão da seção

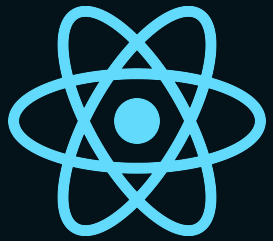


Context API

Introdução da seção

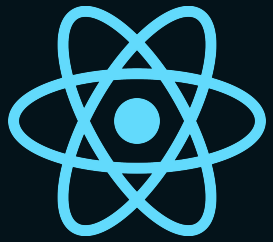
O que é Context API?

- Um recurso do React que facilita o **compartilhamento de dados entre os componentes**;
- Quando há a necessidade de **dados globais**, provavelmente utilizaremos o Context;
- Quando há muitas idas e vindas de props, também deve se considerar o Context;
- Geralmente ficam na **pasta context**;
- Vamos criar um projeto para esta seção!



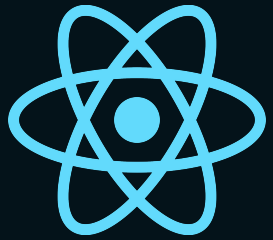
Criando o contexto

- O primeiro passo é **criar o Context**;
- O arquivo tem a **primeira letra maiúscula**, como nos componentes, e geralmente **termina com Context**;
- Exemplo: AlgumContext.js;
- A convenção é deixar na **pasta context** em src;
- Onde utilizamos o contexto, o arquivo precisa ser importado;
- Vamos ver na prática!



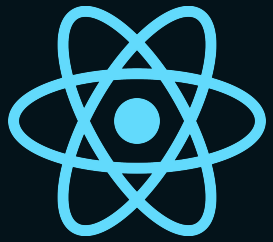
Criando o provider

- O provider vai **delimitar o escopo do contexto**;
- Ou seja, em que componentes teremos acesso aos dados;
- O provider deve **encapsular os componentes** que precisam do context;
- Geralmente é colocado em **main.jsx**;
- O provider tem a **prop children**, para inserirmos elementos dentro;
- Vamos ver na prática!



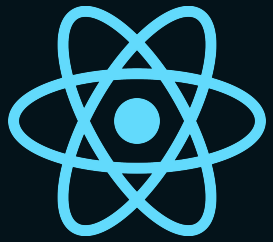
Criando uma estrutura

- Vamos agora criar uma **estrutura mínima** para ver o poder total de Context;
- Iremos instalar o react-router-dom;
- Criar duas páginas;
- E um componente de barra de navegação;
- Vamos lá!



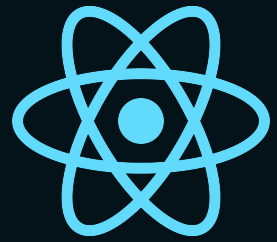
Alterando o contexto

- Agora vamos **exibir e alterar** o valor do contexto;
- Vamos utilizar o hook **useContext** para trazer o nosso contexto as componentes;
- E com este mesmo hook é possível trazer a função que altera o seu valor;
- Vamos ver na prática!



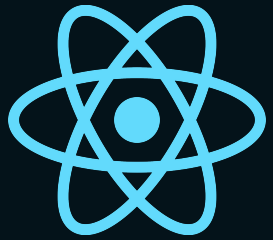
Refatorando contexto com hook

- Podemos **criar um hook** e trabalhar o contexto nele;
- Concentramos o **useContext em um só local**, que será no hook;
- E há um intervalo para uma possível validação na alteração do contexto;
- Vamos ver na prática!



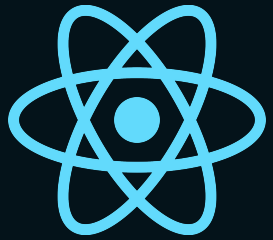
Contexto mais complexo

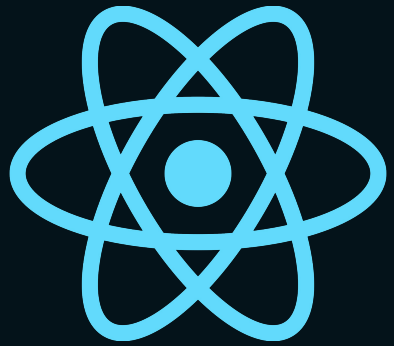
- **Contextos mais complexos** podem necessitar de variações no comportamento;
- Por isso o mais indicado é utilizar o hook **useReducer**;
- Ele funciona como um **useState**, mas com mais possibilidades;
- Vamos ver na prática!



Alterando Contexto complexo

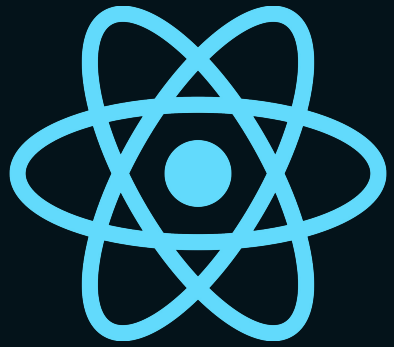
- Para alterar o contexto vamos utilizar uma função chamada **dispatch**;
- Ela também estará no **useReducer**;
- Precisamos enviar todas as informações necessárias para a alteração do valor do contexto;
- Ou seja, o **switch entrará em ação**;
- Vamos ver na prática!





Context API

Conclusão da seção

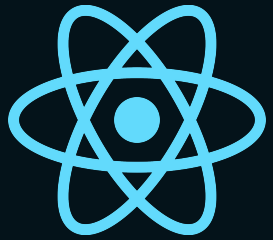


Os hooks do React

Introdução da seção

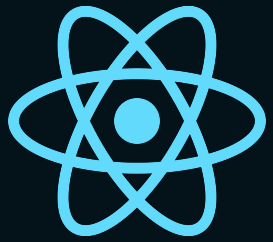
useState

- O **useState** é um dos principais hooks do React;
- Sua função é **gerenciar valores**;
- Podemos consultar e alterar;
- Isso nos permite **re-renderizar um componente**, o que não ocorre na manipulação de variáveis;
- Vamos ver na prática!



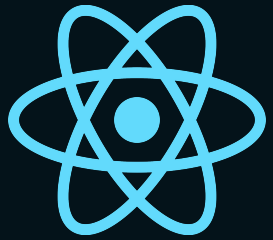
useState e inputs

- Podemos conciliar o **useState** aos valores dos inputs;
- O evento será o **onChange**;
- Com isso podemos salvar valores de um formulário, e posteriormente fazer o envio;
- Há também a estratégia de **Controlled Inputs**;
- Vamos ver na prática!



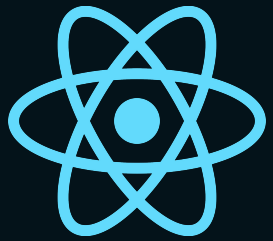
useReducer

- o **useReducer** tem função semelhante ao useState;
- Porém na manipulação de conteúdo podemos **executar uma função**;
- Então temos que o useReducer recebe um valor para gerenciar e uma função para alteração do valor;
- Vamos ver na prática!



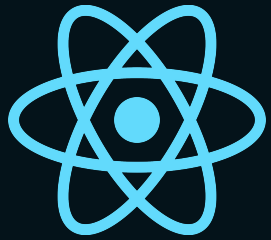
Avançando em useReducer

- Se o **useReducer** fosse utilizado como no exemplo passado, talvez o useState fosse suficiente;
- Por isso o reducer geralmente contém operações mais complexas, utilizando um **switch**;
- Vamos ver na prática!



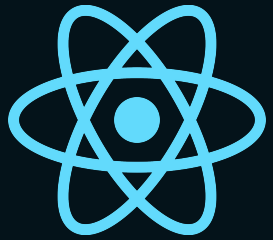
useEffect

- O **useEffect** é utilizado em várias situações, e está no ranking dos mais utilizados no React;
- Podemos fazer **alterações nos elementos** ou **requisições HTTP**;
- A grande vantagem é que **estas ações podem ser controladas**;
- O **array de dependências** possui os itens a serem monitorados;
- Vamos ver na prática!



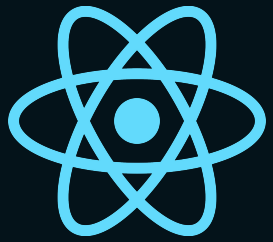
useEffect com array vazio

- As vezes precisamos executar uma ação **uma única vez**;
- Isso pode ser feito com o **array de dependências vazio**;
- Na primeira renderização do componente, o código é executado;
- Vamos ver na prática!



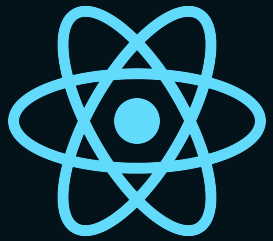
Mais sobre o array de dependências

- Podemos condicionar a execução do useEffect colocando **algo no array de dependências**;
- **Sempre que este dado for modificado**, o useEffect executa mais uma vez;
- Isso permite a **reutilização**, e de forma controlada;
- Vamos ver na prática!



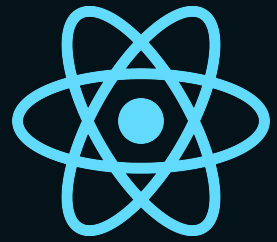
Limpeza do useEffect

- Algumas ações precisam ser limpas com uma técnica chamada de **cleanup**;
- Exemplo: uma ação executada de tempos em tempos, pode ser executada após uma mudança de página;
- Para resolver isso basta finalizar a ação no useEffect com um return;
- Vamos ver na prática!



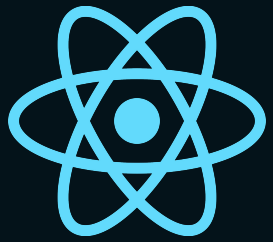
useContext

- O **useContext** é o hook utilizado para consumir um context, da **Context API**;
- Vamos precisar criar o contexto e o provedor (**Provider**);
- Envolver os componentes que vão receber os dados;
- E fazer o uso do hook onde necessário;
- Vamos ver na prática!



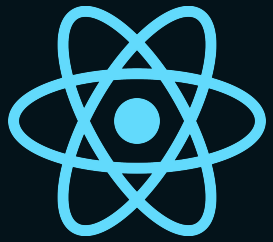
useRef

- O **useRef** pode ser utilizado como o useState, gerenciando valores;
- A diferença é que **ele é um objeto**, e seu valor está na propriedade **current**;
- Este hook **não re-renderiza o componente** ao ser utilizado, o que pode ser interessante em alguns casos;
- Vamos ver na prática!



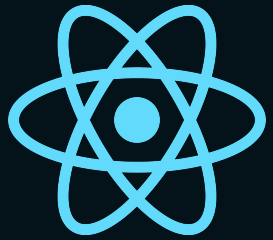
useRef e o DOM

- o useRef pode ser utilizado para **selecionar elementos no JSX**;
- Com isso podemos fazer **manipulação no DOM**, ou aplicar eventos como o **focus**;
- Que deixa o input como selecionado;
- Vamos ver na prática!



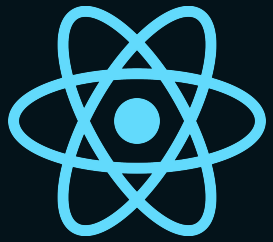
useCallback

- O **useCallback** pode ser utilizado em duas situações;
- Ele basicamente memoriza uma função, fazendo com que ela **NÃO seja reconstruída** a cada renderização;
- O primeiro caos é quando estamos prezando pela performance, então uma **função muito complexa** pode ser criada uma só vez;
- Já o segundo caso é **quando o React nos alerta** que uma função deveria estar no useCallback;
- Vamos ver na prática!



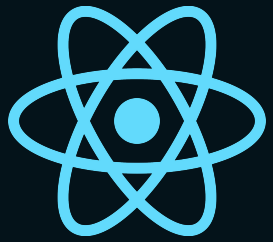
useMemo

- O **useMemo** pode ser utilizado para garantir a referência de um objeto;
- Fazendo com que algo possa ser atrelado a uma referência e não um valor;
- Com isso é possível **condicionar useEffects a uma variável** de maneira mais inteligente;
- Vamos ver na prática!



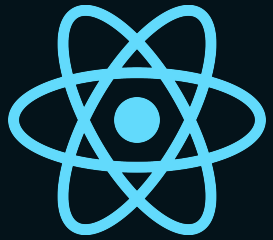
useLayoutEffect

- Muito parecido com o **useEffect**;
- A grande diferença é que este **hook roda antes da renderização** do componente;
- Ou seja, **o hook é síncrono**, bloqueando o carregamento da página para o sucesso da funcionalidade;
- A ideia é executar algo antes de qualquer elemento aparecer na página;
- Vamos ver na prática!



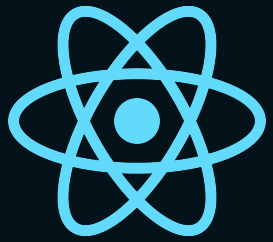
useImperativeHandle

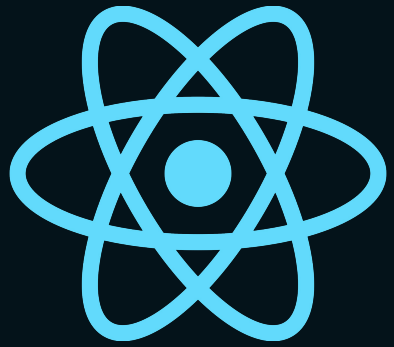
- Com o **useImperativeHandle** temos como acionar ações em outro componente, de forma imperativa;
- Como não podemos passar refs como props, precisamos utilizar a função **fowardRef**;
- Isso nos permite passar referências e torna nosso exemplo viável;
- Vamos ver na prática!



Custom hooks

- Os **Custom hooks** são os hooks que nós criamos;
- Estratégia utilizada para **abstrair funções complexas** do componente ou **reaproveitar** o código;
- Muito utilizado em projetos profissionais;
- Vamos ver na prática!





Os hooks do React

Conclusão da seção