Shift

Aula 06

Javascript e React

+ Projeto Star Wars!

O QUE O MATERIAL COBRE

1. REQUISIÇÕES ASSÍNCRONAS: FETCH
2. TEMPORIZADORES
3. PROJETO STAR WARS
4. SUGESTÕES DE EXERCÍCIOS

O QUE O MATERIAL COBRE

5 CONCEITOS DE DESIGN TOKENS E COMPONENTES

6 INTRODUÇÃO AO REACT

7 COMPONENTES E PROPRIEDADES

8 SUGESTÕES DE EXERCÍCIOS

1

Requisições assíncronas

Fetch

O que são requisições assíncronas?

Hoje em dia o termo **assíncrono (ou async)** é muito falado, principalmente quando estamos no mundo da programação web.

Nós vimos que com a tag **<form>** e sua propriedade **action** podemos fazer uma requisição ao servidor, porém essa requisição é síncrona, ou seja em sequencia, ela sai da página e volta.

Porém existe uma maneira que proporciona uma melhor experiência e não precisa sair da página que são as **requisições assíncronas** dentro do Javascript.

Javascript: fetch() básico

```
// Utilizando .then()/.catch()
let data = null;
function fetchData() {
  fetch('https://api.github.com/users/guscsales')
    .then((response) => response.json())
    .then((responseData) => {
      data = responseData;
    })
    .catch((error) => {
      console.error('Ocorreu um erro', error);
    });
}
```

```
// Utilizando async/await
async function fetchData() {
  try {
    const response = await
fetch('https://api.github.com/users/guscsales');
    const data = response.json();

    return data;
  } catch (error) {
    console.error('Ocorreu um erro', error);
  }
}
```

Uma das maneiras de fazer requisição assíncrona no Javascript é utilizando o método “fetch”.

Para receber a resposta com os dados vindos do servidor é necessário utilizar **uma de duas abordagens**:

- **.then()/.catch()** – funções que retornam a resposta após a requisição finalizada, sendo sucesso no then e erro no catch;
- **async/await** – faz a mesma coisa que o primeiro, porém é uma “sugar sintax” para o processo.

Recomendo o uso do async/await sempre que possível.

Javascript: fetch() com mais propriedades

```
async function createUser() {
  try {
    const response = await
fetch('https://api.github.com/users', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: {
        name: 'Gustavo Sales',
        email: 'gustavo@gsales.io',
        password: 'secret'
      }
    });
    const data = response.json();

    return data;
  } catch (error) {
    console.error('Ocorreu um erro', error);
  }
}
```

Com o fetch é possível passar mais informações para o servidor caso necessário, como o método, headers e parâmetros (payload), em um segundo parâmetro que é um objeto ao chamar a função:

- **method** – tipo da chamada (GET, POST, PUT, PATCH ou DELETE);
- **headers** – objeto com headers como por exemplo content-type;
- **body** – dados a serem enviados ao servidor (backend).

Lembrando que essas propriedades são opcionais.

2

Temporizadores

setTimeout e setInterval

Javascript: temporizadores

```
setTimeout(() => {
  alert('Olá mundo');
}, 5000);
```

```
let timesToRun = 5;
let runningCounter = 0;

const interval = setInterval(() => {
  runningCounter++;
  console.log(new Date().toLocaleTimeString())

  if (runningCounter >= timesToRun) {
    console.log('Intervalo parado');
    clearInterval(interval);
  }
}, 1000);
```

É possível criar temporizadores no javascript para executar tarefas depois de alguns segundos, minutos ou horas ou até mesmo fazer com que essas tarefas rodem continuamente com base no tempo, para isso usamos algumas funções:

- **setTimeout** – executa apenas uma vez após o tempo proposto;
- **setInterval** – executa continuamente após o tempo proposto (a primeira chamada conta o tempo também) até ser finalizado de alguma maneira;
- **clearInterval** – remove o temporizador.

O tempo deve ser passado sempre em milisegundos, ou seja, 3 segundos = 3000 milisegundos.

3

Projeto
Star Wars

Projeto: Star Wars

Vamos criar um projeto utilizando uma API pública chamada **SWAPI,** para consumir dados sobre o universo Star Wars.

Para essa a stack do projeto vamos utilizar HTML, TailwindCSS, VanillaJS com fetch e outros eventos.

A API de busca será:

https://swapi.dev/api/<resource>/?search=<text>

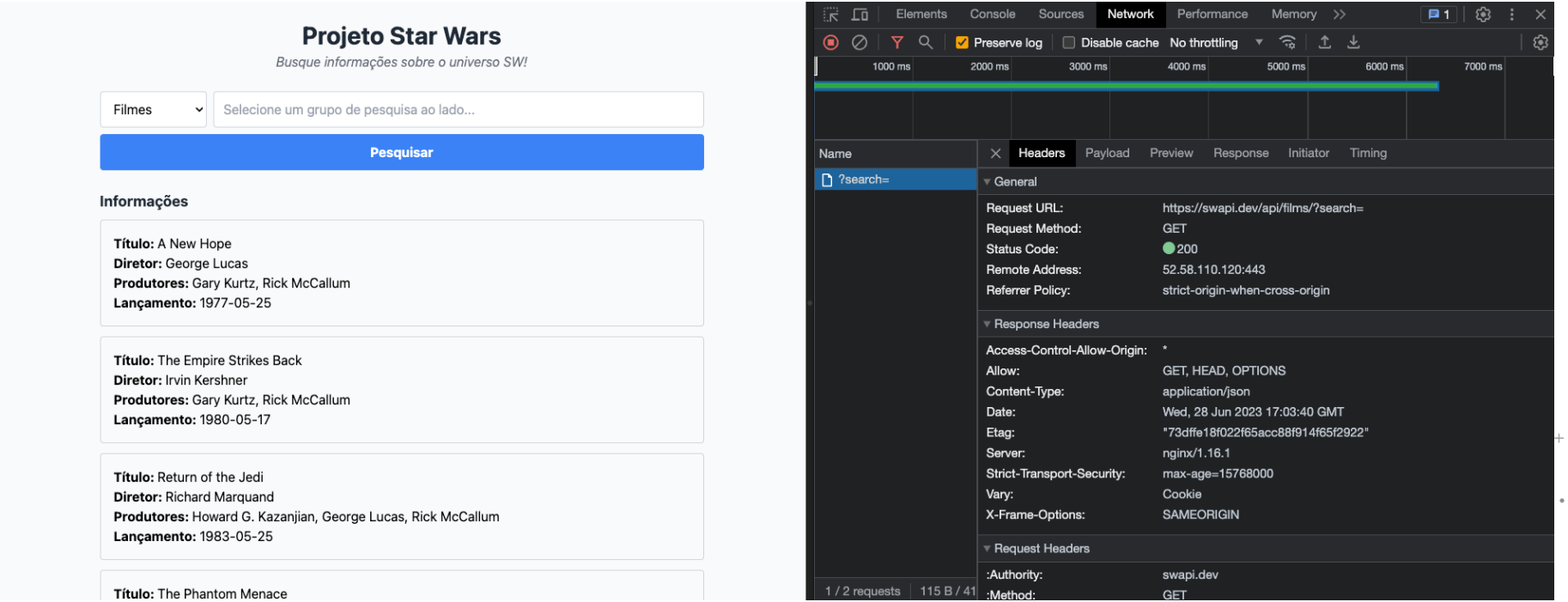
Sendo “resource” o que queremos buscar como personagens, planetas e “text” o termo da busca.

Mais detalhes em: https://swapi.dev/

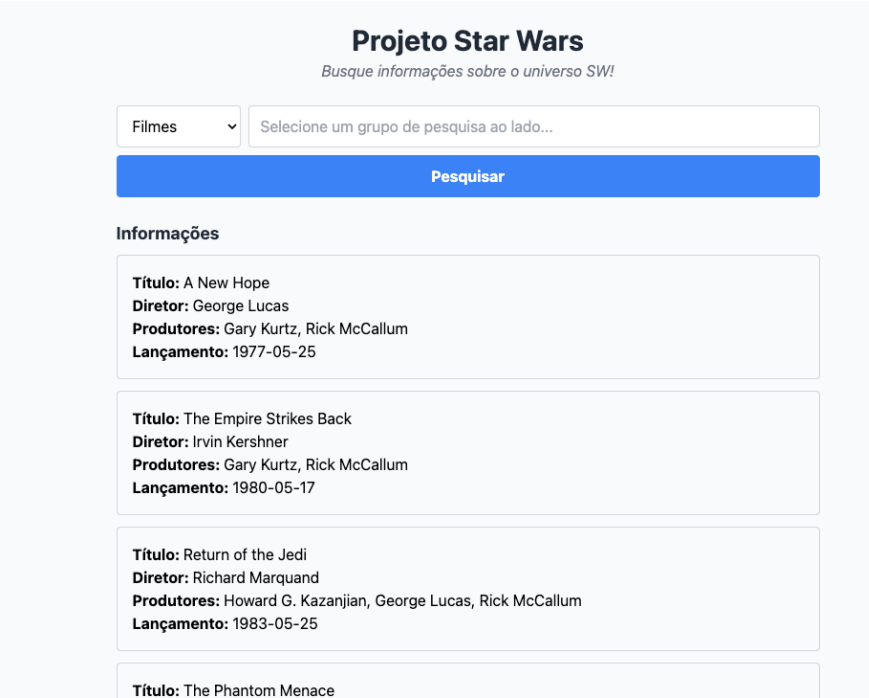
Projeto: Star Wars – Tela Inicial

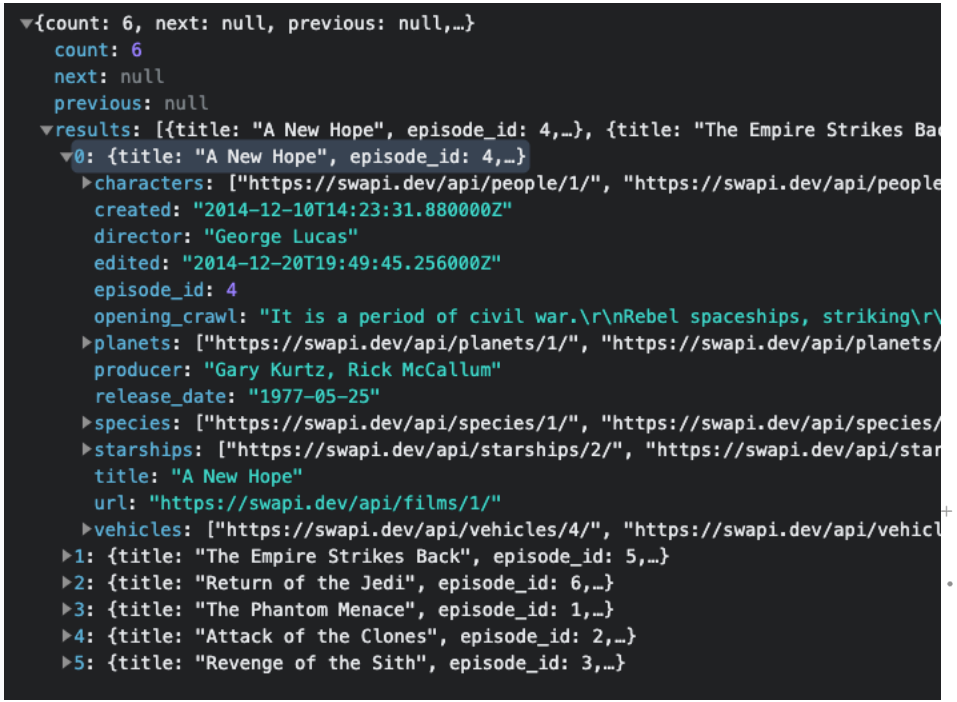
Projeto: Star Wars – Buscar Pessoas

Projeto: Star Wars – Buscar Filmes URL da API



Projeto: Star Wars – Buscar Filmes Retorno da API





Projeto: Star Wars – Estado Vazio (Sem Resultados)

Projeto: Star Wars – Estado de Carregamento

4

Sugestões de **Exercícios**

Exercícios para treinar Requisições e Temporizadores

Para requisições

- Crie um projeto que liste usuários e ao clicar no usuário mostre mais detalhes sobre o mesmo – use sua criatividade no layout!
  - Utilize a API https://randomuser.me/api/ para uma lista fake de usuários
  - Documentação com mais detalhes: https://randomuser.me/documentation

Para temporizadores

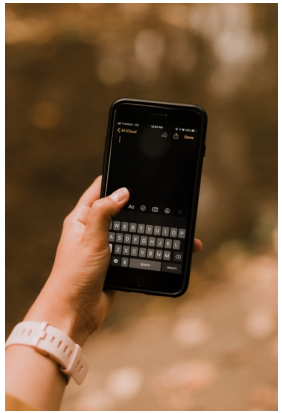
- Crie um relógio digital que envie uma notificação de pausa a cada 25 minutos para você seguindo o conceito de pomodoro.
  - Dica: utilize a API de notificações no navegador: https://developer.mozilla.org/en-US/docs/Web/API/Notification

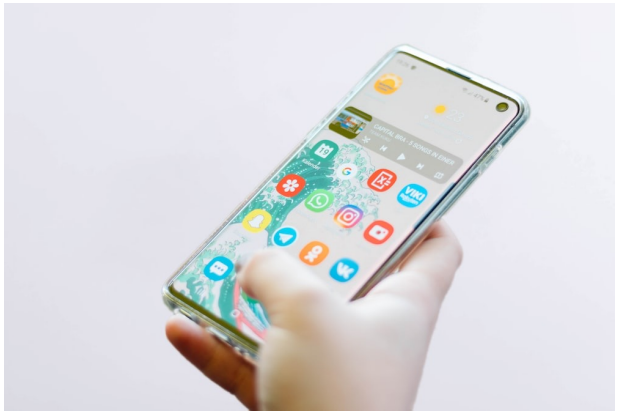
Bons estudos! 😀

5

Conceitos de
Design Tokens e Componentes

Design Tokens e Componentes





Quando falamos de programação em geral, mas principalmente em frontend hoje em dia precisamos sempre pensar em **otimização, performance e entregas rápidas com qualidade**.

Tudo o que você faz no código, **pode interferir diretamente ou indiretamente** na vida de uma pessoa que utiliza aplicativos no computador ou celular, seja para o trabalho, lazer ou qualquer outra razão dela.

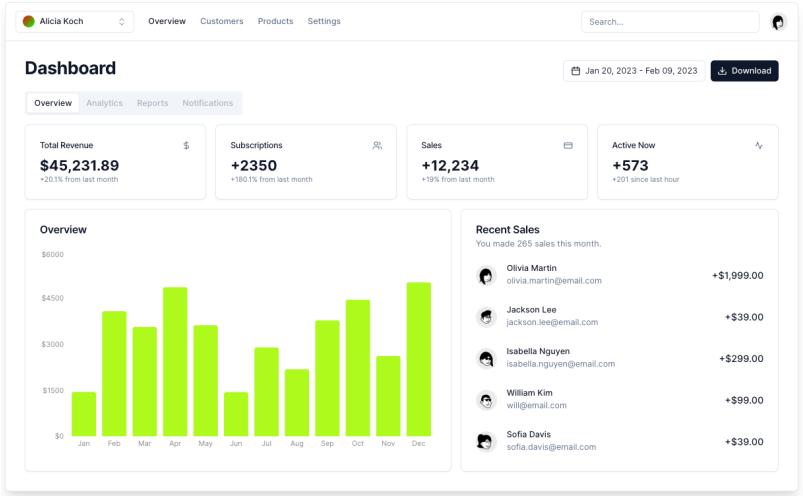
Design Tokens e Componentes

Já vimos que o TailwindCSS nos oferece muitos padrões prontos que apenas com classes podemos utilizar em nosso código, como margens, cores e mais.

Esses padrões de definição são chamados de **design tokens** e eles facilitam muito a criação de **componentes** que são blocos de código criados que podem ser reutilizados em outras partes maiores da aplicação como a criação de páginas inteiras.

Imagine que criar um site em geral é como montar lego, você junta as peças e tem a obra feita no final.

Design Tokens e Componentes



Os frameworks modernos de frontend como React tem em sua base a criação e reutilização de components.

Normalmente ao trabalhar em uma empresa a equipe de design já tem os **design tokens** e uma **biblioteca de componentes (ou design system)** já definidos no Figma, se é um projeto novo esse processo todo deve fazer parte do roadmap.

Após concluído essa parte fundamental no Figma, um dos trabalhos do frontend é transcrever e manter sempre atualizados tanto os tokens quanto os components da biblioteca, e aí você pode usar a melhor forma que desejar.

A parte de customização do TailwindCSS ajuda muito nisso!

Se liga...

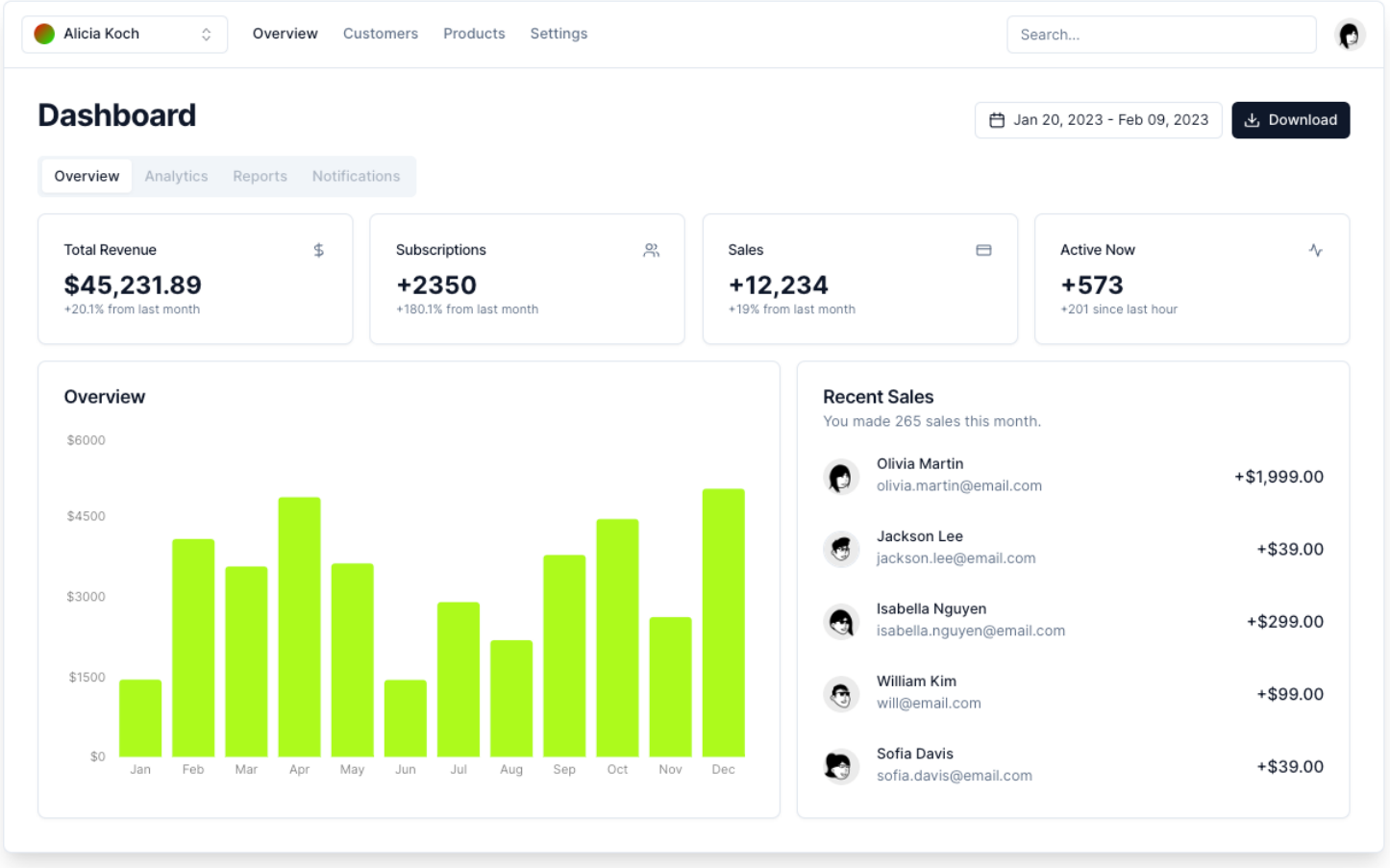
Estamos indo para o próximo nivel do frontend...

...na Dica!

A partir de agora, olhe para os layouts e enxergue componentes, mesmo que pequenos!

Procure componentizar! 😀

Quantos componentes você vê nessa imagem?



6

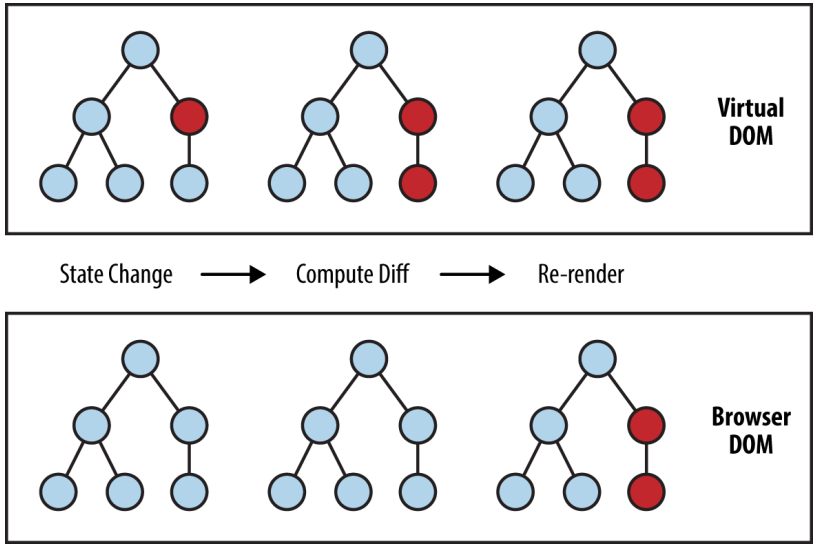
Introdução ao **React**

React: a biblioteca para interfaces web e nativas

Criado e mantido pelo time de open source do Facebook o **ReactJS** é uma biblioteca Javascript open source usada para criar interfaces interativas, reativas e componentizadas. Com o React é possível criar aplicações para web e também para celulares com o **React Native**.

Até a versão 16.7 o React utilizava como padrão a programação reativa, mas a partir da versão 16.8 o conceito de **programação funcional** foi completamente adaptado e é o recomendado.

React: DOM x Virtual DOM e Render Tree



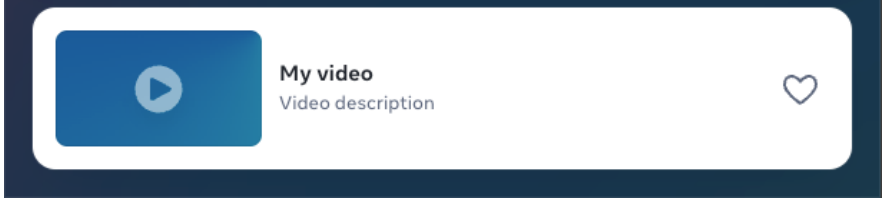
O React utiliza o conceito chamado **“Virtual DOM”** onde ele faz uma comparação dom o DOM original do navegador para renderizar e/ou aplicar alterações apenas onde for realmente necessário, assim aumentando muito a performance e eficiência.

Ele possui um conceito de **árvore de renderização (render tree)** que nos recomenda a transitar os dados através de propriedades sempre de **cima para baixo, do pai para o filho**.

React: funcionamento básico

Video.js

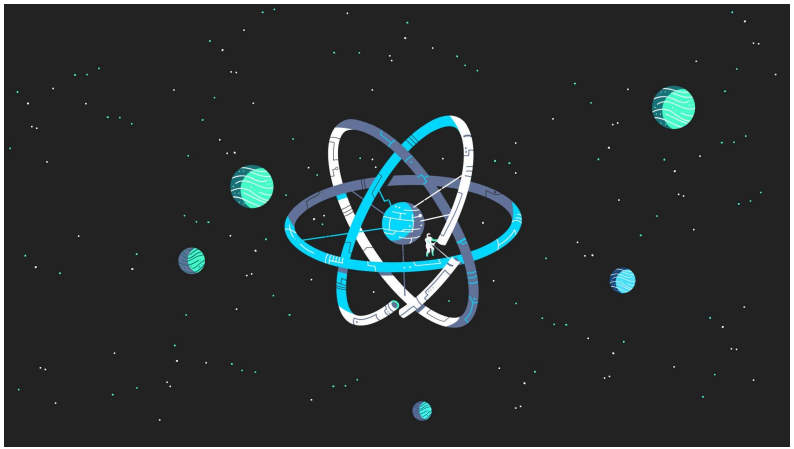
```
function Video({ video }) {
  return (
    <div>
      <Thumbnail video={video} />
      <a href={video.url}>
        <h3>{video.title}</h3>
        <p>{video.description}</p>
      </a>
      <LikeButton video={video} />
    </div>
  );
}
```



Os principais conceitos de funcionamento básico do React incluem:

- Propriedades customizadas;
- Estados;
- Componentes;
- JSX (escrever HTML no JS);
- Fluxo de dados unidirectional (render tree) através de propriedades;
- Ciclo de vida dos componentes.

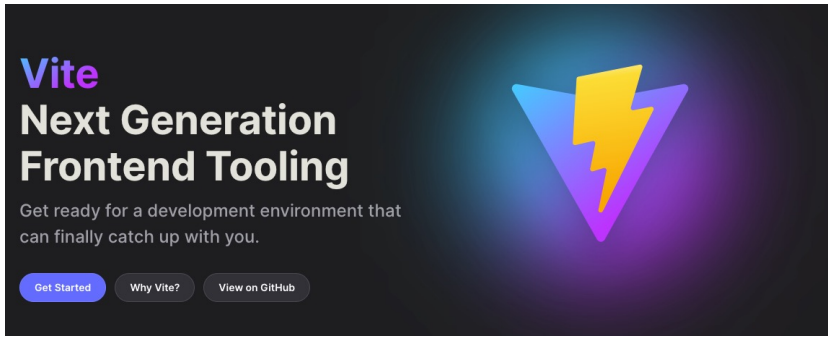
React: biblioteca e não framework



O React é focado em **renderização de interfaces** através de componentes e apenas isso, por isso não é considerado um framework.

Para fazer roteamento, validação de formulários, e outras tarefas, como até mesmo iniciar um projeto em React do zero de maneira simples são necessárias outras bibliotecas em conjunto.

React: criando um projeto com Vite



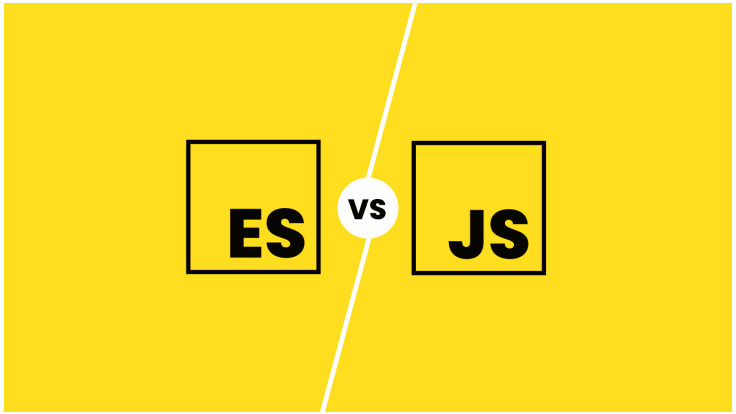

```
npm create vite@latest
```

Existem diversas maneiras de se inicializar um projeto com React, uma das bibliotecas mais recomendadas é o **Vite**, onde com apenas alguns comandos já teremos um ambiente completamente preparado para programar utilizando o React.

Site do Vite: https://vitejs.dev/

No terminal digite o commando ao lado para criar um projeto novo.

React: agora é ECMAScript e não mais Javascript puro



Até agora tudo o que fizemos com o Javascript foi com os conceitos dele puro. Onde não é necessária uma compilação para o navegador compreender.

Programando em React vai mudar um pouco e vamos para o **ECMAScript** que são os padrões de linguagens em script, sendo o Javascript a principal linguagem.

No fim mudam algumas sintaxes, mas com certeza para melhor.

Tudo vira Javascript após compilado.

React: TailwindCSS no Vite

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

```
/** @type {import('tailwindcss').Config} */
export default {
  content: ['index.html', 'src/**/*.{js,jsx,css}'],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

A instalação do TailwindCSS muda um pouco. Para utilizar com React é necessário mais outros dois pacotes, o **postcss** e o **autoprefixer**.

- **postcss** – permite sintaxes extras no CSS ao criar o código, que no fim se tornam CSS puro;
- **autoprefixer** – coloca automaticamente os prefixos dos motores, por exemplo --webkit- para chromium, --moz- para mozilla etc.

Rode o comando de inicialização do tailwind com a flag **-p** no final.

7

Componentes e **Props**

React: componentes customizados

```
export default function MyComponent() {
  return (
    <div className="italic font-light text-lg text-gray-500">
      Hello world from React!
    </div>
  );
}
```

```
import MyComponent from './MyComponent';

function App() {
  return <MyComponent />;
}

export default App;
```

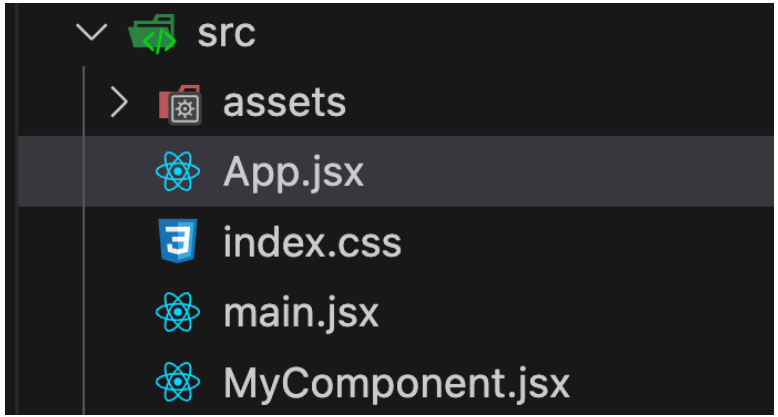
O React é formado por componentes, e dentro desses componentes podemos criar o HTML que será retornado e exibido na tela.

Cada componente é uma função Javascript que **obrigatóriamente precisa ter um retorno**.

Esse componente será utilizado como se fosse uma tag HTML personalizada com o nome da função que você criou.

Obrigatóriamente os componentes React precisam começar com letra maiuscula, seguindo o padrão **PascalCase**.

React: .js ou .jsx?



Essa é uma pergunta que podemos falar “depende”. Cada projeto está estruturado de uma maneira.

A diferença principal do JS pro JSX é que o JSX garante que a renderização HTML irá funcionar sem problemas, então a principio sempre que um componente React for criado, escolha a extensão **.jsx** para o seu arquivo.

React: propriedades

```
npm i prop-types
```

Outra grande caracteristica do React e dos componentes é que podemos criar **propriedades customizadas**, que são chamadas de **props**.

São como se fossem propriedades do HTML, mas com o nome que quisermos dar e com o valor que quisermos enviar. Através delas podemos enviar dados de um componente para o outro de forma simples.

Com a ajuda da biblioteca prop-types podemos também colocar validações para as propriedades.

Não é obrigatório, mas é uma boa prática.

React: propriedades

```
import PropTypes from 'prop-types';

export default function MyComponent({ value }) {
  return <div className="italic font-light text-lg text-gray-500">{value}</div>;
}

MyComponent.propTypes = {
  value: PropTypes.string.isRequired,
};
```

```
import MyComponent from './MyComponent';

function App() {
  return (
    <div className="grid gap-2 p-6">
      <MyComponent value="Hello world!" />
      <MyComponent value="Olá mundo!" />
    </div>
  );
}

export default App;
```

8

Sugestões de **Exercícios**

Exercícios para treinar Componentes e Props no React

Com o site que você criou na primeira parte

- Crie componentes com os elementos que estão lá e transforme faça com que a home seja exibida através de um projeto React criado com Vite

Com o projeto SWAPI

- Crie componentes personalizados como o input, select, botão e refaça o projeto criado com VanillaJS, mas com React.

Bons estudos! 😃

OBRIGADO

@guscsales - gsales.io

FIAP


FIAP