

VERSÃO 5.0



GUIA PARA APRENDER **FRONTEND**

Navegando no universo da web: Um guia
para iniciantes em criação de sites.



por Iuri Silva

Sinopse

Um desenvolvedor frontend precisava dominar só HTML, CSS e JavaScript para ter uma boa colocação no mercado. Claramente, outras coisas eram incluídas nesse "só", como por exemplo: responsividade, semântica do HTML, acessibilidade, performance entre outras. Mas tudo girava em torno das três tecnologias principais.

A web mudou. Entraram outras habilidades como versionamento de arquivos, automatização de tarefas, pré-processadores, bibliotecas, frameworks, gerenciador de pacotes, e entre outras tecnologias do mundo JavaScript. Mas a base de tudo ainda gira em torno das três tecnologias principais.

Isso quer dizer que não importa a época ou a tecnologia, a base sempre vai ser a mesma. Então antes de pular para o tão famoso framework, temos que aprender a base dele para realmente entender o que está por trás dos panos. Sei que encontrar material de qualidade e com uma linguagem simples para quem está começando não é uma tarefa muito fácil. Com essa premissa em mente, criei esse ebook onde irei ensinar alguns fundamentos do frontend, mas não irei me limitar ao básico. Iremos nos aprofundar nas tecnologias mais utilizadas no mercado de desenvolvimento frontend.

Sumário

HTML

15 tópicos neste módulo

O que é	13
Iniciando	13
Sintaxe	15
Indentação adequada	16
Títulos	17
Textos	17
Imagen	18
Div	24
Link	25
Comentários	25
Tabela	26
Listas	29
Formulário	30

Tags semânticas	34
Meta tags	39
CSS	16 tópicos neste módulo
O que é	42
Iniciando	42
Sintaxe	44
Comentários	44
Variáveis	45
Aplicando fontes	46
Reset	47
Unidades de medidas	49
Cores	51
Seletores	53
Preenchimentos	55
Display	58

Position	59
Flexbox	64
Responsividade	72
Grid	76

JavaScript

14 tópicos neste módulo

O que é	87
Iniciando	87
Variáveis	88
Tipos de variáveis	91
Operadores	94
Declarações condicionais	98
Function	100
Object	101
Array	102
Métodos de array	104

Loops	107
DOM	112
API com Axios	121
JavaScript Assíncrono: Async/Await	124

Sass

9 tópicos neste módulo

O que é	130
Iniciando	131
Sintaxe	135
Comentários	136
Variáveis	136
Import/Use	137
Extend	138
Mixin	139
Nested	142

Bootstrap

8 tópicos neste módulo

O que é	143
O que é framework CSS	144
Iniciando	145
Componentes	147
Cores	150
Spacing	150
Sizing	152
Responsividade	153

React

10 tópicos neste módulo

O que é	157
Create React App	158
Estrutura da pasta	163
JSX	167
Componentes	173

Props	177
CSS Modules	179
Eventos	182
Hooks	185
API	194

React Router

6 tópicos neste módulo

O que é	200
Iniciando	201
Criando as rotas	202
Navegação de links	207
Rota 404	208
Rotas dinâmicas	210

TypeScript

7 tópicos neste módulo

O que é	214
---------	-----

Iniciando	214
Tipos	216
Interface	225
Propriedades opcionais	228
Estendendo Interfaces	229
TypeScript no React	230

styled-components

7 tópicos neste módulo

O que é	238
Iniciando	240
Aplicando estilos	241
Estilos dinâmicos	243
Estendendo estilos	245
Arquivos separados	246
Estilos globais	248

Tailwind CSS

7 tópicos neste módulo

O que é	251
Iniciando	252
Responsividade	255
Estados dos elementos	256
Dark mode	257
Diretivas	258
Extra	261

React Hook Form

8 tópicos neste módulo

O que é	265
Iniciando	266
Registrando campo	268
Aplicando validação	273
Criando componentes	276
Watch	277

Focus	278
Form Provider	279

Radix UI

6 tópicos neste módulo

O que é	284
Iniciando	284
Aplicando estilos	287
Cores	290
Ícones	291
Componentes	293

Storybook

6 tópicos neste módulo

O que é	300
Iniciando	300
Estrutura do Storybook	304
Criando componente	306

Documentando o componente	309
Entendendo o estrutura gerada	317

Cypress

3 tópicos neste módulo

O que é	322
Iniciando	323
Aplicando testes no projeto	329

Next.js

7 tópicos neste módulo

O que é	339
Iniciando	339
Estrutura	340
Rotas	347
API Reference	352
Rotas API	356
Data Fetching	357

Bônus

6 tópicos neste módulo

Vite	316
React Query	364
Node	370
i18n	384
Swagger	391
Next 13	397



MÓDULO I

HTML

Pronto café! Agora é hora de mergulhar nesse módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes com os conhecimentos adquiridos através de um questionário no Módulo.

+ 15 TÓPICOS NESTE MÓDULO

O que é

HTML (Hypertext Markup Language ou Linguagem de Marcação de HiperTexto) é uma linguagem de marcação (não de programação) da web - cada vez que você carrega uma página da web, você está carregando um código HTML. Pense no HTML como o esqueleto de uma página, ele é responsável pelos textos, links, listas e imagens.

Iniciando

HTML é escrito em arquivos com o formato .html. Para criar uma página em HTML é fácil, entre em seu editor de código (se não tiver uma instalado eu recomendo o Visual Studio Code) e salve o arquivo em branco como index.html (você pode nomeá-lo como quiser, mas não esqueça do .html).

A estrutura inicial do seu HTML será essa:

```
1 <!DOCTYPE html>
2 <html>
3   <head></head>
4   <body></body>
5 </html>
```

<!doctype html> - Essa tag (não tem fechamento) informa ao navegador que o arquivo faz parte de um documento HTML5.

<html> - Representa a raiz de um documento HTML.

<head> - O head contém informações sobre sua página, mas não o conteúdo que vai aparecer na tela da sua página. Ela conterá coisas como: links para folhas de estilo (CSS), título da página, links de fontes e meta tags e entre outros.

<body> - No body contém todo o conteúdo que vai aparecer na tela da sua página. Todo o código que você queira apresentar na página deverá estar dentro dele.

Sintaxe

Os elementos HTML são escritos usando tags. Todas as tags tem uma chave de abertura e fechamento. A tag de fechamento tem uma barra após o primeiro colchete.



```
1 <tag>iuricode</tag>
```

Por exemplo, se você deseja criar um parágrafo, usaremos as chaves de abertura `<p>` e fechamento `</p>`.



```
1 <p>Programador sem café é um poeta sem poesia</p>
```

Os elementos também podem entrar dentro de outros elementos:



```
1 <pai>
2   <filho>Esta tag está dentro de outra tag</filho>
3 </pai>
```

Indentação adequada

As quebras de linha entre suas tags são super importantes para escrever um bom código HTML. Abaixo temos um bom exemplo disso.

```
1 ...  
2 <body>  
3   <h1>Aqui temos um título</h1>  
4   <p>E aqui um parágrafo</p>  
5 </body>  
6 ...
```

Abaixo temos um indentação não recomendada:

```
1 <!DOCTYPE html> <html> <head> </head> <body> <h1>  
2 Não faça isso</h1> </body> </html>
```

Títulos

As tags de títulos são representadas em até seis níveis. `<h1>` é o nível mais alto e `<h6>` é o mais baixo. Quanto maior for o nível da tag, maior vai ser o tamanho da fonte apresentada na tela e sua importância no SEO.

```
1 <h1>Título do h1</h1>
2 <h2>Título do h2</h2>
3 <h3>Título do h3</h3>
4 <h4>Título do h4</h4>
5 <h5>Título do h5</h5>
6 <h6>Título do h6</h6>
```

Textos

As tags de texto, definem diferentes formatações para diversos tipos de texto. Desde estilos de fonte, parágrafos, quebra de linha ou até mesmo spans. Iremos conhecê-las:

`<p>` - Sendo a principal tag de texto, é usada para criar um parágrafo.

 - Mesmo tendo a sua funcionalidade parecida com o uso dos parágrafos, os spans geralmente são utilizados para agrupar uma pequena informação.

 - Deixa o texto em negrito.

<i> - Deixa o texto em itálico.

<hr> - Cria uma linha horizontal.

 - Adiciona importância (SEO) no texto.

Tags de medias

As tags de medias ajuda na criação de experiências multimídia em uma página da web. Isso é, adicionar em nossas páginas imagens, vídeos, áudios e entre outros conteúdos de medias.

Imagen

Para adicionar uma imagem na página é bem simples, vejamos:

```
1 
```

O src vem de source, ele é atributo da tag , nele vai conter o caminho da imagem que será inserida por você.

Além dele, temos o atributo alt. Recomendo que você sempre coloque o atributo alt para que pessoas com deficiências saibam (através de um leitor) do que se trata a imagem na página.

Pontos importantes:

- A tag não tem a chave de fechamento. Geralmente fechamos utilizando o "/".
- Se você tiver a imagem de uma pasta, deve colocar o caminho dela dentro do atributo src.

Audio

A tag <audio> cria um reproduutor de áudio em uma página da web. Ele oferece suporte a controles de mídia, como reprodução, pausa, volume e mudo .

```
1 <audio controls>
2   <source src="arquivo.mp3" type="audio/mpeg" />
3 </audio>
```

A tag `<audio>` faz referência a um ou mais arquivos de áudio com um atributo `src` ou o elemento `<source>`.

O navegador escolherá o primeiro arquivo com um formato de arquivo compatível. Os formatos de áudio suportados são MP3 , WAV e OGG.

Para ativar os controles de áudio: reproduzir, pausar, volume, mudo e download a tag `audio` precisa do atributo "controls".

A tag `<source>` define uma fonte de arquivo de áudio, e junto a ela temos dois atributos:

- `src` - URL ou nome do arquivo do áudio a ser reproduzido.
- `type` - Formato de arquivo do áudio.

Video

A tag <video> cria um player de vídeo em uma página da web.

Este player oferece suporte à reprodução de vídeo diretamente dentro da página com controles de mídia, como iniciar, parar, volume e outros.

```
1 <video width="320" controls>
2   <source src="video.mp4" type="video/mp4" />
3 </video>
```

Este elemento suporta três formatos diferentes: MP4, WebM e OGG.

O elemento <video> também pode reproduzir arquivos de áudio, mas a tag <audio> fornece uma melhor experiência do usuário.

Podemos também definir o height e width do vídeo.

- height - Define a altura do player de vídeo
- width - Define a largura do player de vídeo

Iframe

A tag <iframe> renderiza outra página ou recurso HTML em uma página. É comumente usado para incorporar uma página HTML ou um documento PDF.

A renderização de páginas é um pouco atrasada porque a página é carregada de um servidor remoto.



```
1 <iframe src="https://www.iuricode.com/efront"
2   style="width: 100%; height: 350px;">
3 </iframe>
```

As páginas com frames não podem ser manipuladas com JavaScript quando a página vem de um domínio diferente, o mesmo com oposto.

Um bom exemplo da utilização do elemento <iframe> é incorporar um vídeo do YouTube em nossa página. Essa tag será responsável por realizar essa ação.

Para adicionar um vídeo do YouTube em sua página basta clicar em compartilhar (no vídeo) e escolher a opção “incluir”, logo em seguida colocar todo o código do <iframe> no documento HTML.



Canvas

A tag <canvas> cria um container onde os gráficos podem ser desenhados com JavaScript.

Canvas é uma boa escolha para desenvolvimento de jogos com uso intensivo de gráficos, então se você deseja criar um jogo em JavaScript a utilização da tag <canvas> será essencial.

```

 1 <canvas id="jogo" width="400" height="250">
 2   Opa, ocorreu um erro!
 3 </canvas>

```

O elemento <canvas> pode renderizar gráficos 2D e 3D (como linhas, círculos, retângulos, polígonos, imagens de bitmap, texto e muito mais) usando JavaScript.

Se o navegador não suportar tela, o texto dentro do elemento <canvas> será exibido, em nosso caso, será a mensagem "Opa, correu um erro!"

Div

A tag <div> é utilizada para dividir elementos para fins de estilos (usando class ou id). Ele deve ser utilizado somente quando não tiver outro elemento de semântica.

Mesmo tendo a sua funcionalidade parecida com o que são geralmente utilizados para guardar uma pequena informação. A <div> é usada para uma divisão de um conteúdo pois, como o uso dela ajuda a quebrar os elementos em linhas, deixando melhor a visualização.

```
1 <div>
2   <p>Entrou dentro de uma div</p>
3 </div>
```

Link

A tag `<a>` (ou elemento âncora) é um hiperlink, que é usado para vincular uma página a outra. O conteúdo dentro de cada `<a>` precisará indicar o destino do link utilizando o atributo `href`. Se a tag `<a>` não tiver o atributo `href`, ela será apenas um espaço reservado para um hiperlink.

Por padrão, os links aparecem da seguinte forma em todos os navegadores:

- Um link não visitado está sublinhado e azul;
- Um link visitado é sublinhado e roxo;
- Um link ativo está sublinhado e vermelho;



```
1 <a href="www.iuricode.com/efront">eFront</a>
```

Comentários

Para adicionar um comentário em um código HTML, é preciso utilizar as tags próprias para esse propósito, nosso é `<!-- -->`.

Portanto, todo o código que estiver dentro dessa tag não será executado.



```
1 <!-- <p>sou um comentário</p> -->
```

Tabela

A criação de uma tabela em uma página HTML é feita com a tag `<table>`.



```
1 <table></table>
```

Entretanto, somente essa tag não é o suficiente para formatar a tabela da maneira correta, pois ela precisa de outras tags para exibir a formatação adequada. Confira quais são eles.

TR

Em todas as tabelas existem linhas. Para isso, utilizamos a tag `<tr>`, ela pode ser inserida em diferentes áreas da tabela, como no cabeçalho, no corpo e no rodapé.



```
1 <tr></tr>
```

TH

A tag <th> é utilizada para inserir o cabeçalho na tabela, ela pode ser inserida em diferentes áreas. Na região do corpo da tabela, ela serve para dar destaque e identificar cada coluna específica.



```
1 <th></th>
```

TD

Além das linhas, toda tabela precisa de uma coluna. Isso significa que é preciso adicionar a tag <td> em todas as linhas para criar as colunas desejadas e inserir o conteúdo a ser exibido.



```
1 <td></td>
```

Outras tags da tabela:

- Definindo a legenda de uma tabela: <caption>
- Criando um rodapé: <tfoot>
- Especifica um grupo de uma ou mais colunas em uma tabela para formatação: <colgroup>
- Agrupa o conteúdo do cabeçalho em uma tabela: <thead>
- Agrupa o conteúdo do corpo em uma tabela: <tbody>
- Agrupa o conteúdo do rodapé em uma tabela: <tfoot>

Exemplo simples de uma tabela:

```
1 <table>
2   <tr>
3     <th>Nome</th>
4   </tr>
5   <tr>
6     <td>Iuri</td>
7   </tr>
8 </table>
```

Listas

Vamos agora falar um pouco sobre as listas (ordenadas e desordenadas) e como elas funcionam. As listas são muito importantes quando queremos listar itens na página.

O elemento `` é usado para representar um item que faz parte de uma lista. Este item deve estar contido em um elemento pai: uma lista ordenada (``) ou uma lista desordenada (``).

Listas ordenadas

As listas ordenadas (ou numeradas) são usadas para indicar alguma sequência ou numeração.

```
● ● ●  
1 <ol>  
2 <li>HTML</li>  
3 <li>CSS</li>  
4 <li>JavaScript</li>  
5 <li>React</li>  
6 </ol>
```

Listas desordenadas

As listas desordenadas são usadas para listar itens, sem se preocupar com sua sequência. Chamamos apenas de lista de marcadores. Ela segue o mesmo padrão da ordenada apenas mudando de `` para ``. Em seu visual ele mudará de números para pontos.

```
1 <ul>
2   <li>HTML</li>
3   <li>CSS</li>
4   <li>JavaScript</li>
5   <li>React</li>
6 </ul>
```

Formulário

Formulários são um dos principais pontos de interação entre o usuário e sua página. Um formulário HTML é feito de um ou mais widgets. Esses widgets podem ser campos de texto, caixas de seleção, botões, checkboxes, radio buttons e entre outros elementos HTML.

Para construir o nosso formulário de contato, vamos utilizar os seguintes elementos: `<form>`, `<label>`, `<input>`, `<textarea>` e `<button>`.

Form

O elemento `<form>` define um formulário e os seus atributos definem a maneira como esse formulário se comporta.



```
1 <form action="/" method="post"></form>
```

Label

O elemento `<label>` é a maneira formal de definir um rótulo. Esse elemento é importante para acessibilidade - quando implementados corretamente, os leitores de tela falarão o rótulo de um elemento de formulário juntamente com as instruções relacionadas.



```
1 <label>Nome:</label>
```

Input

O elemento `<input>` é usado para criar controles interativos que receber dados do usuário. A semântica de um `<input>` varia consideravelmente dependendo do valor de seu atributo `type`.

```
1 <input type="email"></input>
```

No input temos o atributo type. Esse atributo define o tipo do nosso input. Exemplo: o type="email" ele define que o campo aceita só endereço de email.

O tipo padrão de um input é text, se este atributo não for especificado. Mas temos outros valores possíveis:

- Um botão sem comportamento padrão: type="button".
- Uma caixa de marcação: type="checkbox".
- Um controle para especificar cores: type="color".
- Um controle para inserir uma data: type="date".
- Um campo para editar um endereço de e-mail: type="email".
- Um controle que permite ao usuário selecionar um arquivo: type="file".
- Um campo de senha cujo valor é ocultado: type="password".
- Um campo de texto com uma só linha para digitar termos de busca: type="search".

Fieldset

O elemento <fieldset> é usado para agrupar elementos em um formulário.



```
1 <fieldset></fieldset>
```

Legend

O elemento <legend> representa um rótulo para o conteúdo do <fieldset>.



```
1 <legend>Contato</legend>
```

Exemplo de um formulário completo:



```
1 <form action="/" method="post">
2   <label>Nome:</label>
3   <input type="text"/>
4   ...
```

```
1 ...
2 <label>Email:</label>
3 <input type="email">
4
5 <input type="submit" value="Enviar" />
6 </form>
```

Tags semânticas

Na vida sempre temos uma forma correta de fazer as coisas, no HTML não é diferente. As tags semânticas além de deixar o código melhor para o SEO, ele ajuda outros desenvolvedores a entender seu código.

Informações importantes:

- As tags semânticas não têm nenhum efeito na apresentação na página.
- As tags semânticas tem significado e deixam seu conteúdo claro.

Elementos não semânticos: <div> e .

Elementos semânticos: <form>, <table>, <nav>, <aside>, <article>, <footer>, <section> e entre outros.

Veja que <div> é amplo, mas a tag <footer> da um significado (que é o rodapé).

Portanto ao invés de fazer isso:



```
1 <div>Sou o rodapé</div>
```

Seria melhor:



```
1 <footer>Sou o rodapé</footer>
```

Section

O elemento <section> é utilizada para marcar as seções de conteúdo de uma página. Com Esse elemento agrupamos de forma lógica nosso conteúdo, separando a informação em áreas diferentes. O principal objetivo é retirar essa responsabilidade das divs.



```
1 <section></section>
```

Article

O elemento `<article>` representa uma composição independente em um documento. Isso é, não precisa do resto do site para contextualizar. Este poderia ser um simples card, um artigo de revista ou jornal, um post de um blog, ou qualquer outra forma de conteúdo independente.

Para um melhor SEO a tag `<article>` precisa de um título (`<h1>` - `<h6>`) acompanhado.



Aside

O elemento `<aside>` representa uma seção de uma página que consiste de conteúdo que poderia ser considerado separado do conteúdo principal. Essas seções são, muitas vezes, representadas como barras laterais.

Um bom exemplo de uso são os sites de notícias, a maioria tem uma seção lateral para ver outras notícias que não estão relacionadas com a notícia principal da página.



```
1 <aside></aside>
```

Header

O elemento `<header>` (cabeçalho) representa um container para conteúdo introdutório (muitas vezes usado na primeira seção da página).



```
1 <header></header>
```

Footer

O elemento `<footer>` representa um rodapé de um conteúdo. Assim como o `<header>` podemos utilizar ele em vários lugares pela página.



```
1 <footer></footer>
```

Nav

O elemento `<nav>` (navegação) representa uma seção de uma página que aponta para outras páginas ou para outras áreas da página, ou seja, uma seção com grupos de links de navegação.



```
1 <nav></nav>
```

Main

O elemento `<main>` define o conteúdo principal da página. Ele representa o conteúdo mais importante da página, que está diretamente relacionado ao tópico central do documento. É importante utilizar apenas uma vez esse elemento por arquivos HTML criado.



```
1 <main></main>
```

Meta tags

SEO (Search Engine Optimization) é o conjunto de técnicas responsáveis por fazer com que determinada página de um site apareça nas primeiras páginas do Google. Quanto melhor for o SEO de um site, maiores as chances de que a página apareça logo de cara para quem pesquisa por determinado assunto.

A tag <meta> define metadados (informações) sobre um documento HTML. As tags <meta> sempre vão dentro do elemento <head> e normalmente são usadas para especificar o conjunto de caracteres, descrição da página, palavras-chave, autor do documento e configurações da janela de visualização.

Informações importantes:

- Os metadados não serão exibidos na página, mas podem ser analisados por máquina.
- Os metadados são usados por navegadores, mecanismos de busca e outros serviços da web.

Title

Title define o título do documento. Ele é mostrado na aba da página do navegador.



```
1 <title>Nome da página</title>
```

Description

Description contém a descrição do conteúdo da página, vários navegadores usam esta meta como descrição padrão da página quando é marcada.



```
1 <meta name="description"
2   content="Descrição sobre a página"
3 />
```

Charset

Este atributo define a codificação de caracteres usados na página. O conjunto de caracteres UTF-8 cobre quase todos os caracteres e símbolos do mundo. Sempre tenha ele em suas páginas.

```
1 <meta charset="UTF-8">
```

Keywords

Keywords é uma meta de palavras-chaves associadas ao conteúdo da página, contendo strings separadas por vírgulas.

```
1 <meta name="keywords" content="css, html, js">
```

Author

Author define o nome do autor do documento.

```
1 <meta name="author" content="Iuri">
```

Adquira meus outros materiais com 10% de desconto!

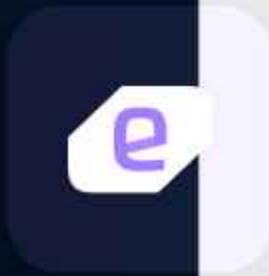
UTILIZE O CUPOM: **FRONTIO**


Este material é destinado a pessoas que querem aprender a programar.
Aprendendo com exemplos simples, fazer depuração e entender
o funcionamento de código é muito mais fácil quando se entende o que
está acontecendo por trás das linhas de código.

[QUERO COMPRAR TAMBÉM](#)


Adquira este material e obterá um **guião de boas práticas para a
criação de interfaces**. Além de ensinar como aplicar
tratamentos específicos ao redor da sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)


Se você está tentando resolver problemas para entrar na área de
programação, este material é para você! Ele explica os conceitos em suas
mídias. Porém isso, envolve muitos termos técnicos. OHHH,
perdão, é só porque para entender a complexa classificação de

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 2

CSS

Pronto café! Agora é hora de mergulhar nesse módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes com os conhecimentos adquiridos através de um questionário no Moodle.

• 16 TÓPICOS NESTE MÓDULO

O que é

CSS (Cascading Style Sheets ou Folhas de Estilo em Cascata) é a linguagens de marcação (não de programação, assim como o HTML) responsável por adicionar estilos em nossas páginas feitas com HTML, como cores, tamanhos, posicionamentos e entre outros. Sem ele, as páginas são apenas um grupo de textos, links e imagens.

Iniciando

Etapa 1: Criar um arquivo CSS. A primeira coisa que precisamos fazer é criar um arquivo CSS do qual nossa página HTML possa obter seu estilo. Então em seu editor de código basta você criar um novo arquivo chamado ele de style.css. Lembre-se que o nome pode ser qualquer um, mas precisar ter o .css no formato do arquivo.

Etapa 2: Linkar seu CSS no HTML. Depois de criado precisamos conectar nosso arquivo style.css no documento HTML. Dentro da tag `<head>` do documento HTML, vamos adicionar uma tag `<link>` para conectar a nossa nova folha de estilo.

```
1 <link rel="stylesheet" href="style.css">
```

Certifique-se de que sua página HTML e sua folha de estilo CSS estão no mesmo nível de pasta, caso ele esteja dentro de uma pasta diferente do arquivo HTML basta colocar o nome da pasta antes do nome do arquivo separado com uma barra (/).

Etapa 3: Chame e dê estilo aos elementos. Experimente selecionar um elemento e estilizá-lo.

```
1 h1 {  
2   color: #00f;  
3   font-size: 26px;  
4 }
```

Acabamos de chamar todas tags <h1> do nosso documento HTML e adicionar o tamanho da fonte para 26px e com a cor azul (sempre atualize a página em seu navegador assim que aplicar algo novo em seu arquivo CSS ou HTML).

Sintaxe

A sintaxe do CSS é bem simples.

Primeiro precisamos indicar um seletor (pela tag, id ou class) e dentro das chaves inserimos os comandos referente à formatação, que são as propriedade e valor.

```
1 seletor {  
2   propriedade: valor;  
3 }
```

Comentários

Para adicionar um comentário em nosso documento CSS, é preciso utilizar o comando `/* */`. Portanto, todo o código que estiver dentro dessa tag não será executado.

```
1 div {  
2   /* color: blue; */  
3 }
```

Variáveis

Uma aplicação grande geralmente tem uma quantidade muito grande de arquivos CSS, e com uma quantidade de repetição de valores muito frequente. Por exemplo, a mesma cor pode ser usada em vários lugares diferentes em nosso documento, caso seja requerido uma substituição na cor, teríamos que procurar todos os lugares que essa cor foi adicionada. Já com variáveis CSS a história muda. Ela permite que um valor seja guardado em uma variável, para ser referenciado em muitos outros lugares. Isso é ótimo para nós desenvolvedores pois, caso seja solicitado a substituição de trocar o valor dessa variável, apenas um lugar será substituído para que todo nosso documento sofra alteração.

Elas são configuradas usando esta notação:

```
1 :root {  
2   --cor: black;  
3 }
```

O :root se equipara à raiz de uma árvore, que representa o documento.

E são acessadas usando a função var():

```
1 h1 {  
2   var(--cor);  
3 }
```

Aplicando fontes

Para colocar uma fonte em sua página primeiro é preciso chamar ele no seu HTML, para isso vamos entrar no site do Google Fonts. Logo em seguida, procure a fonte que você deseja e clique no botão "Select this style" para assim adicionar os estilos de fontes que você deseja.

Perceba que quando você selecionou abriu uma aba na lateral na direita. Nessa aba vai ter duas opções para aplicar a fonte em sua página, uma é o `<link>` e o outro `@import`. O `<link>` é para importar pelo seu documento HTML e o `@import` é para o seu documento CSS. Você pode escolher qualquer um, isso vai com seu gosto, mas importar pelo HTML nos trás uma melhor performance.

Perceba que logo depois de selecionar a opção `<link>` é apresentado um código HTML e uma propriedade CSS, o `font-family`. É exatamente ele que usaremos em nossa página CSS para escolher quais elementos irão receber a nossa fonte. Isso porque podemos ter muitas fontes no CSS.

Em nosso documento HTML dentro da tag <head> coloque o código HTML apresentado no site do Google Fonts e no documento CSS coloque propriedade font-family e o nome da fonte.

```
1 * {  
2   font-family: 'Nome da fonte aqui';  
3 }
```

Reset

Agora vamos desfazer os padrões dos navegadores.

"Mas espera, o que isso quer dizer?"

Quer dizer que os navegadores tem um padrão, e alguns desses padrões não são legais para nós desenvolvedores, então vamos tirar alguns deles.

Quando criamos uma página HTML, por padrão, nosso site tem um espaçamento em volta da página. Para tirar esse padrão usamos duas coisas: o padding (espaçamento dentro do conteúdo) e o margin (espaçamento fora do conteúdo). Iremos zerar eles.

```
1 body {  
2   padding: 0;  
3   margin: 0;  
4 }
```

E por último, vamos aplicar a propriedade box-sizing: border-box. Esse é super importante na criação dos elementos pois, quando criamos um container sempre usamos uma largura e altura (no nosso exemplo vai ser 300px em cada um) mas caso você queira colocar um espaçamento interno nele (padding) de 30px, o elemento vai deixar de ser 300px e será 330px.

Mas a gente não quer isso, não é? Queremos que o elemento continue 300px e com um espaçamento interno. É aí que entra o nosso querido box-sizing: border-box. Ele irá manter o tamanho do container e com o valor do espaçamento aplicado nele.

```
1 * {  
2   padding: 0;  
3   margin: 0;  
4   box-sizing: border-box;  
5 }
```

Unidades de medidas

O CSS oferece um número de unidades diferentes para a expressão de comprimento. Iremos citar só duas delas: PX (medida absoluta) e REM (medida relativa).

Medidas absolutas: Essas são as mais comuns que vemos no dia a dia. São medidas que não estão referenciadas a qualquer outra unidade, ou seja, não dependem de um valor de referência. São unidades de medidas definidas pela física, como o pixel, centímetro, metro, etc...

Medidas relativas: O uso delas é mais apropriado para que possamos fazer ajustes em diferentes dispositivos garantindo um layout consistente e fluido em diversas mídias.

PX

Definir unidades de texto em pixels traz uma desvantagem. Quando o usuário tenta mudar o tamanho do texto pelo navegador ou na configuração do celular, o texto na nossa aplicação não aumenta. Pelo simples motivo de que o texto está definido em pixels (medidas absoluta). Isso é um problema sério de acessibilidade. É por isso que muitos desenvolvedores preferem definir o tamanho dos textos utilizando outras medidas em vez de trabalhar com pixels.

```
1 h1 {  
2   font-size: 18px;  
3 }
```

REM

Diferente do pixel, o rem é uma unidade escalável e relativa, ele varia de acordo com a dimensão root do seu navegador (por padrão essa unidade é 16px), na maior parte das vezes $1\text{rem} = 16\text{px}$. Podendo variar se o tamanho da fonte raiz for alterado.

```
1 h1 {  
2   font-size: 1rem; // Igual a 16px  
3 }
```

Como disse acima, o valor base é 16px, e isso pode acabar gerando dificuldades para que encontremos alguns tamanhos padrões que costumam ser utilizados. Por exemplo, como faríamos para atingir um tamanho de 10px utilizando rem? Para isso precisamos calcular. Mas ficar calculando esses números não é algo muito "amigável", porém, podemos adicionar um pequeno truque para nos ajudar.

Consiste em mudar o valor do font-size do html para 62,5%, que seria igual 10px. Dessa forma 1rem = 10px, fazer a conversão torna-se bem mais simples já que basta dividir o valor em pixel por 10 e se obterá o valor em rem. Exemplo: 1.2rem = 12px, 2.6rem = 26px, 5.6rem = 56px.

```
● ● ●  
1 html {  
2   font-size: 62,5%;  
3 }  
4  
5 h1 {  
6   font-size: 1.2rem; /*equivalente a 12px*/  
7 }
```

Cores

RGB e HEX são formatos de composição de cores. Após compreender esse tema você pode escolher um dos formatos para utilizá-la mais frequentemente nos projetos. Mas esses não são os únicos formatos disponível no CSS, porém são os mais utilizados.

RGB

O processo é simples: como na vida real onde você mistura cores para obter uma outra cor como resultado, você faz a mesma coisa como o RGB.

Sua sintaxe utiliza a soma de 3 valores: Red, Green e Blue. Onde o valor máximo de todas as cores é 255 e o mínimo é 0.

```
● ● ●  
1 h1 {  
2   color: rgb(255, 200, 10);  
3 }
```

Graças a função RGBA (red-green-blue-alpha) podemos adicionar opacidade no formato RGB.

```
● ● ●  
1 h1 {  
2   color: rgba(255, 200, 10, 0.4);  
3 }
```

O alpha define a opacidade como um número entre 0,0 (totalmente transparente) e 1 (totalmente opaco).

HEX

Hex, ou Hexadecimal, tem sintaxe muito mais curta que o RGB. O código Hexadecimal consiste em seis letras ou números precedidos por um "#".

Os seus dois primeiros valores representam a intensidade do vermelho, terceiro e quarto a intensidade de verde e os dois últimos a intensidade de azul.

```
1 h1 {  
2   color: #00ff00;  
3 }
```

Seletores

"Okay, mas se eu tiver mais de um elemento <p> e quiser mudar a cor do texto só de um?".

É aí que entram os identificadores ID e Class:

- As Classes são uma forma de identificar um grupo de elementos. Através delas, pode-se atribuir formatação a VÁRIOS elementos de uma vez.
- O ID é uma forma de identificar UM elemento, e devem ser única para cada elemento. Através dele, pode-se atribuir formatação a um elemento em especial.

Para fazermos referência a uma Classe usando um . (ponto) com o nome da Classe e para fazermos referência a um id usando um # (hashtag) no lugar do . (ponto). Podemos ter os dois no mesmo elemento também.

```
1 .souClass {  
2   color: #f00;  
3 }
```

Nosso HTML ficará assim ao aplicar a Classe "souClass" ou ID.

```
1 <p class="souClass">Sou um elemento com Class</p>  
2  
3 <p id="souID">Sou um elemento com ID</p>
```

Outros seletores

Dependendo da estrutura da nossa aplicação, selecionar elementos acabam sendo complexos de certa forma, mas existem algumas formas de selecionar tags, ID e Classes. Vou apresentar somente três delas.

Seleciona o primeiro parágrafo que vem exatamente após a div:

```
1 div + p {  
2   color: #f00;  
3 }
```

Seleciona todos os elementos <p> com a Class "iuricode":

```
1 p.iuricode {  
2   color: #f00;  
3 }
```

Seleciona todos os parágrafos que são filhos diretos da div:

```
1 div > p {  
2   color: #f00;  
3 }
```

Preenchimentos

As propriedades de preenchimento CSS são usadas para gerar espaço em torno ou dentro do conteúdo.

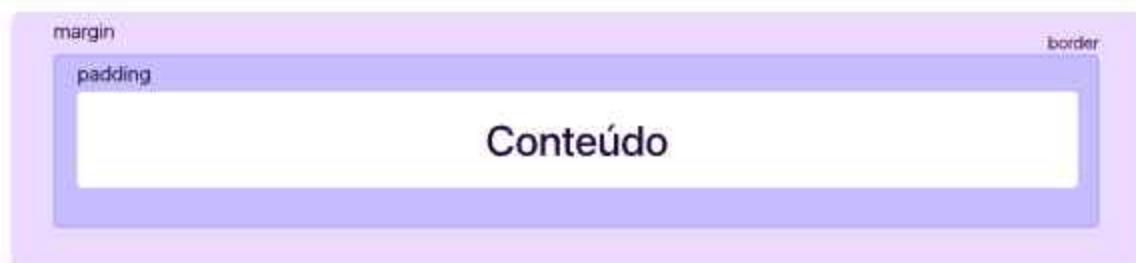
Padding

A propriedade padding é usada para gerar espaço em torno do conteúdo. O padding "limpa" uma área ao redor do conteúdo (dentro da borda) de um elemento.

Margin

A propriedade margin define o tamanho do espaço de fora da borda. O margin é usado para gerar espaço em torno do elemento.

Resumindo: Padding é o espaço entre o conteúdo e a borda, enquanto margin é o espaço fora da borda



Elas podem ser aplicadas nos quatro lados de um elemento: superior, direita, inferior e esquerda (nessa ordem).

No exemplo a seguir vamos usar o margin mas também pode ser aplicada a propriedade padding.

Irei aplicar margin de 20px iguais nos quatros lados do elemento:

```
1 .class {  
2   margin: 20px;  
3 }
```

Agora irei aplicar uma margin superior e inferior de 5px e margin esquerda e direita de 10px:

```
1 .class {  
2   margin: 5px 10px;  
3 }
```

Você também pode definir os espaços individualmente, mas lembre-se sempre da ordem que é: top, right, bottom e left.

```
1 .class {  
2   margin: 0px 5px 10px 15px;  
3 }
```

Display

Basicamente todos os elementos têm um valor padrão da propriedade display. A maioria dos elementos tem seu display configurado em block ou inline. Irei descobrir alguns elementos e seus valores do display.

Block

O elemento block, não aceita elementos na mesma linha que ele, ou seja, quebra a linha após o elemento, e sua área ocupa toda a linha onde ele é inserido.

Alguns elementos que têm como padrão block: `<div>`, `<h1>` até `<h6>`, `<p>`, `<form>`, `<header>`, `<footer>`, `<section>`, `<table>`.

Inline

O elemento inline não inicia em uma nova linha nem quebra de linha após o elemento, pois ele ocupa somente o espaço de seu conteúdo.

Alguns elementos que têm como padrão inline: ``, `<a>`, ``.

Position

A propriedade position especifica como um elemento será posicionado na tela, podemos até posicionar em um ponto específico controlando com os parâmetros top, right, bottom e left.

São quatro tipos de posicionamento disponíveis: static, relative, fixed e absolute. A única configuração que não permite escolher um posicionamento para o elemento é static.

- top - Desloca o elemento na vertical (Y), o valor é a distância do elemento com o topo.
- right - Desloca o elemento na horizontal (X), o valor é a distância do elemento com a borda direita.
- bottom - Desloca o elemento na vertical (Y), o valor é a distância do elemento com a borda inferior.
- left - Desloca o elemento na horizontal (X), o valor é a distância do elemento com a borda esquerda.

Static

Este é o valor padrão de todos os elementos HTML, neste posicionamento os elementos não podem ser controlados por top, right, bottom e left, e não tem seu posicionamento afetado pelo posicionamento de outros elementos.

Relative

Um elemento com relative tem seu posicionamento relacionado com o elemento anterior.

Absolute

Um elemento com absolute tem seu posicionamento relacionado com o elemento pai e não com o elemento anterior, desta maneira elementos anteriores não irão afetar seu posicionamento.

Fixed

O elemento com fixed tem o mesmo comportamento do absolute, só que como o nome já diz, ele fica fixo na tela, isso é, mesmo se acontecer a rolagem ele ficará fixado na página.

Exemplo ilustrativo:

body



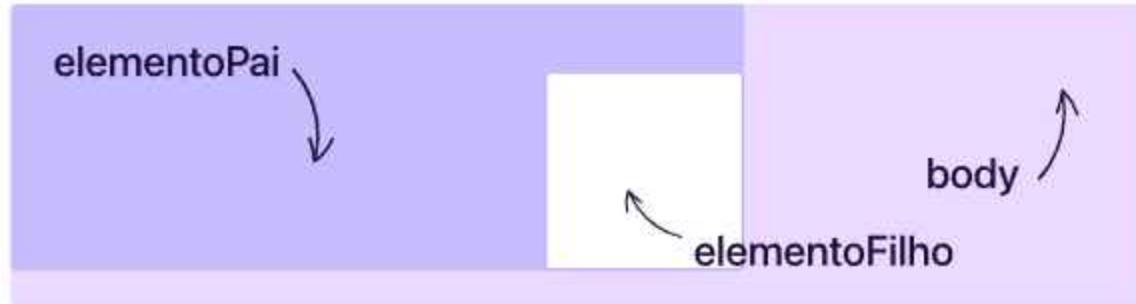
elementoFilho



```
1 .elementoFilho {  
2   position: absolute;  
3   bottom: 0;  
4   right: 0;  
5 }
```

Nesse exemplo, o nosso retângulo roxo é o body da nossa página. Por padrão, sempre que colocamos position absolute em um elemento ele irá se referenciar ao seu ancestral mais próximo (que no nosso caso é body). Além do retângulo roxo, temos nosso quadrado verde, ele é o nosso elemento com position absolute, o elemento fica posicionado de forma absoluta em relação ao fluxo do documento, mas de forma relativa ao seu ancestral.

Que tal um exemplo melhor?



```
1 .elementoPai {  
2   position: relative;  
3 }  
4  
5 .elementoFilho {  
6   position: absolute;  
7   bottom: 0;  
8   right: 0;  
9 }
```

Perceba que nosso elementoFilho continua com as mesmas propriedades, porém adicionamos o elementoPai e atribuímos a ele um position relative.

O que isso muda? Muda tudo!

Lembra que eu falei que o position absolute em um elemento ele irá se referenciar ao seu ancestral mais próximo? Foi o que aconteceu! Como o elementoPai é o ancestral mais próximo do elementoFilho, obrigatoriamente o elementoFilho irá se referir a ele.

"E como ficou nosso HTML?"

Perceba que nossa classe elementoFilho está dentro da classe elementoPai, por isso que ele se referenciou a ele.

```
1 <body>
2   <div class="elementoPai">
3     <div class="elementoFilho"></div>
4   </div>
5 </body>
```

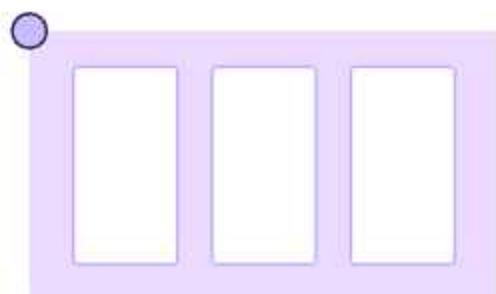
"Okay luri, quer dizer que se o elementoFilho ficar fora do elementoPai ele irá se referenciar ao body?"

Sim, isso mesmo!

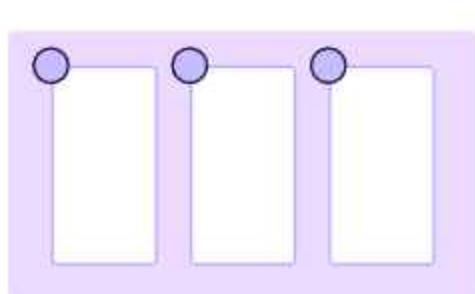
Flexbox

Flexbox são regras que configuram o alinhamento, posicionamento e distribuição de elementos através de regras em uma caixa pai, responsável por organizar o conteúdo dentro do espaço que ela possui. Facilitando, assim, a implementação de sites responsivos.

Container pai



Elementos filhos



Dá aos filhos

- Direção
- Justificação
- Alinhamento

Podem sozinhos

- Se ordenar
- Se alinhar individualmente

justify-content

A propriedade justify-content define como o navegador distribui o espaço entre e ao redor dos itens ao longo do eixo principal de um container flexível. A propriedade só terá efeito se a mesma tiver com a propriedade display flex.

```
1 .elementoPai {  
2   display: flex;  
3   justify-content: flex-start;  
4 }
```



display flex e
justify-content ↑

flex-start (padrão)

Os itens são alinhados no começo do eixo principal. Como pode ver no exemplo anterior colocamos "justify-content: flex-start;" no elemento pai.



flex-end

Os itens são alinhados no final do eixo principal.



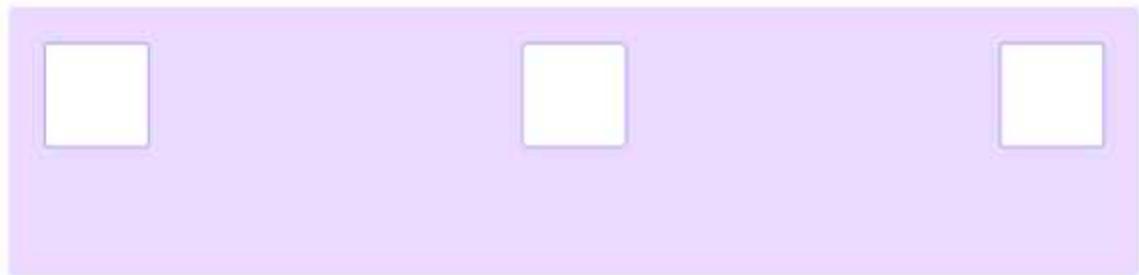
flex-center

Os itens são alinhados no meio do eixo principal.



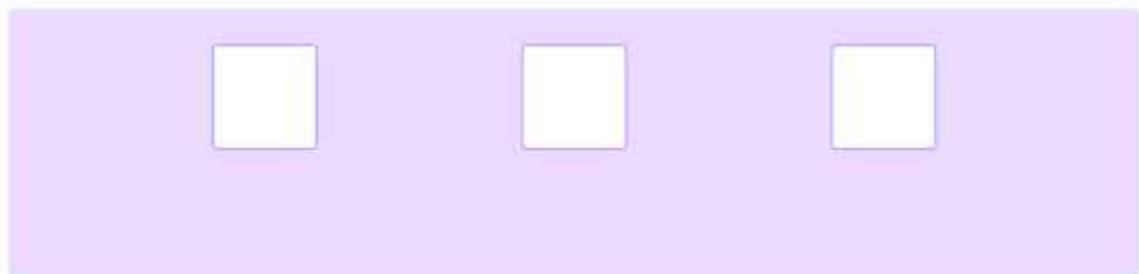
space-between

O primeiro item fica no começo do eixo principal e o último no fim, os restantes ficam alinhados entre si no meio.



space-around

Pega todo o espaço que sobrar na linha e distribui igualmente entre os elementos, deixando o espaço igual em toda a linha do eixo principal.



space-evenly

O espaço é distribuído igualmente entre cada elemento, ou seja, um ao lado do outro, com exceção do primeiro e último do eixo principal.



align-items

Nós usamos o justify-content para alinhar os itens no container, que neste caso é o eixo indo horizontalmente. Para centralizar verticalmente nós usamos a propriedade align-items, para alinhar nossos itens no eixo transversal. Tenha em mente que se o eixo principal mudar o eixo transversal muda com ele.

```
1 .elementoPai {  
2   display: flex;  
3   align-items: flex-start;  
4 }
```



display flex e
align-items



flex-start

Os itens são postos no começo do eixo transversal.



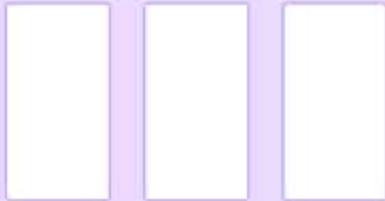
flex-end

Os itens são alocados no final do eixo transversal.



stretch

Os itens se estenderão por todo o eixo.



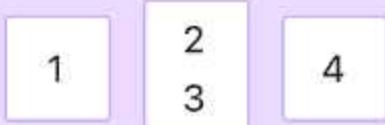
center

Os itens são alinhados no meio do eixo.



baseline

Alinha os itens de acordo com a linha base da tipografia.

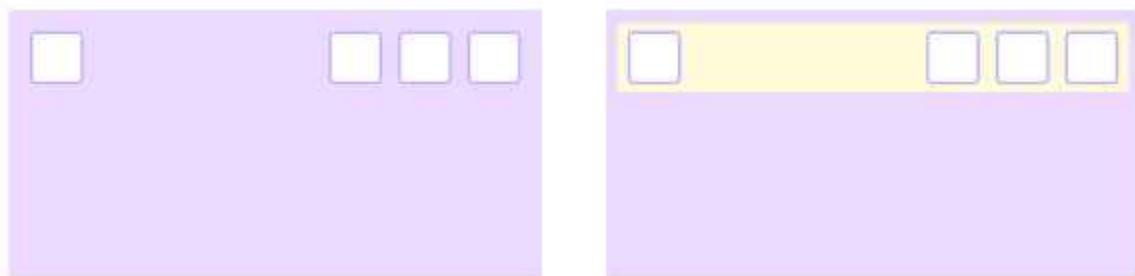


Outras propriedades

Existem outras propriedades do flexbox como flex-basis, flex-direction, flex-grow, flex-shrink, flex-wrap, order, gap, align-self e entre outros. Caso você queira ver outras propriedades, clique [aqui](#) e leia mais.

Posicionamento com flexbox

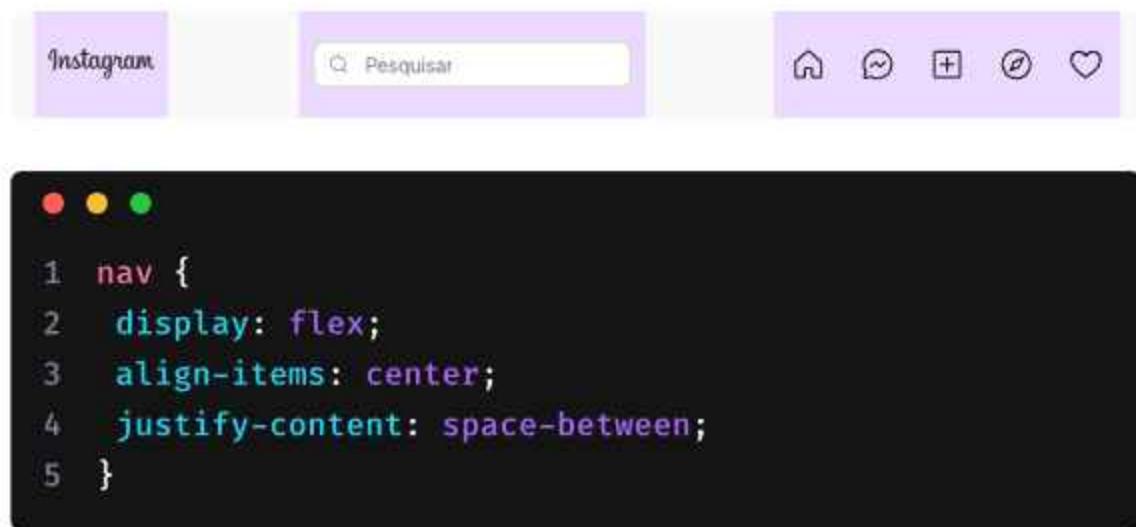
Quando falamos em posicionamento, primeiro temos que pensar qual vai ser a estrutura HTML e assim conseguimos trabalhar com flexbox.



A estrutura acima segue o mesmo padrão de um menu, certo? Perceba a forma que separei a estrutura dele.

A parte em amarelo é o nosso menu, dentro do menu temos duas coisas, um quadrado na esquerda e um grupo de quadrados na direita. Para posicionar eles precisamos agrupar esses quadrados na direita pois, caso isso não ocorra, seria difícil posicionar esses elementos.

Veremos esse exemplo em uma aplicação real.



Para replicar esse menu do Instagram precisaríamos de quatro tags HTML para posicionar os elementos.

É claro que isso leva a outros fatores como tamanhos, alturas, espaçamentos e entre outras coisas. Mas se você está utilizando flexbox para criar uma estrutura e posicionar elementos, essa sempre vai ser a base.

Responsividade

Sites responsivos são aqueles que adaptam o tamanho das suas páginas ao tamanho das telas que estão sendo exibidos, como as telas de celulares, tablets e televisões (como a Netflix e o YouTube).

"Qual a forma de aplicar?"

Temos algumas formas de aplicar responsividade em nossos sites, as mais utilizadas são com o uso de frameworks e media queries. Se você está fazendo um site e tem um prazo curto para entregar ele, recomendo o uso de frameworks pois já está tudo pronto para um layout responsivo.

"Quero deixar responsivo com media queries, como eu faço?"

A responsividade é um estilo, então o media queries será aplicado em sua folha de estilo CSS. Mas antes temos que entender que temos alguns tipos de media, eles são chamados de media types.

Os types definem para que tipo de media um certo código CSS será aplicado.

Media type

Os types mais utilizados são:

- all: usado para todos os dispositivos de tipo de mídia.
- print: usado para impressoras.
- screen: usado para telas de computador, tablets, smartphones e etc.
- speech: usado para leitores de tela que "leem" a página em voz alta.

"Mas como eu faço para aplicar?"

O exemplo a seguir altera a cor de fundo da página para roxo se a janela de visualização tiver 480 pixels de largura ou menos.

```
1 @media screen and (max-width: 480px) {  
2   body {  
3     background-color: #00ff00;  
4   }  
5 }
```

Sempre que vamos fazer algo no media queries, temos antes que dizer que é para um media que estamos desenvolvendo para isso colocamos @media. Logo em seguida o tipo dessa media (no nosso caso foi o screen) e um operador lógico (no nosso caso foi o and). Entre os parênteses passamos o tamanho da largura que aquele media irá ser aplicado.

Perceba que usamos o max, nesse caso estamos dizendo que será aplicado quando tiver 480 pixels de largura ou menos.

Vamos usar o site da Apple como um exemplo real de responsividade.



Veja como o site da Apple aplica responsividade em suas páginas. A grande sacada foi aumentar o tamanho dos card para quando chegar em um determinado tamanho para dispositivos mobile.

```
1 @media screen and (max-width: 480px) {  
2   .card {  
3     width: 100%;  
4   }  
5 }
```

Uma coisa super importante é que você não precisa escrever todas as propriedades de uma class ou id dentro do seu @media, somente o que você quer que mude. Que no caso o tamanho de cada card terá o tamanho da resolução do dispositivo.

Observação: caso seja feito com flexbox você precisa colocar a propriedade `flex-wrap: wrap;`. Ela define se os itens ficam na mesma linha ou se podem ser quebrados em várias linhas. Que no caso dos card da Apple os card quebraram a linha e foram para baixo.

Padrões de resoluções

- Desktops: (`min-width: 1281px`)
- Laptops, desktops: (`min-width: 1025px`) and (`max-width: 1280px`)
- Tablets, ipads: (`min-width: 768px`) and (`max-width: 1024px`)
- Tablets de baixa resolução, celulares: (`min-width: 481px`) and (`max-width: 767px`)
- A maioria dos smartphones móveis: (`min-width: 320px`) and (`max-width: 480px`)

Fonte: <https://gist.github.com/janily/8453473>

CSS Grid

O CSS Grid é um sistema de estruturação de layout. A diferença dele pro Flexbox é que ele nos permite configurar layouts em duas dimensões (linhas e colunas) sendo que no Flexbox é apenas um dimensão.

Grid Container

Para criar um grid container utilizamos a propriedade `display: grid` (ou `display: inline-grid` para tornar o elemento um grid container porém com comportamento inline). Assim declarando, todos os elementos filhos diretos daquele container se transformam em grid items.

```
● ● ●  
1 .container {  
2   display: grid;  
3 }
```

```
● ● ●  
1 <div class="container">  
2   <div class="item">1</div>  
3   <div class="item">2</div>  
4   <div class="item">3</div>  
5   <div class="item">4</div>  
6 </div>
```

Isso irá criar uma grid do tipo block, significando que o `.container` irá ocupar a linha inteira.

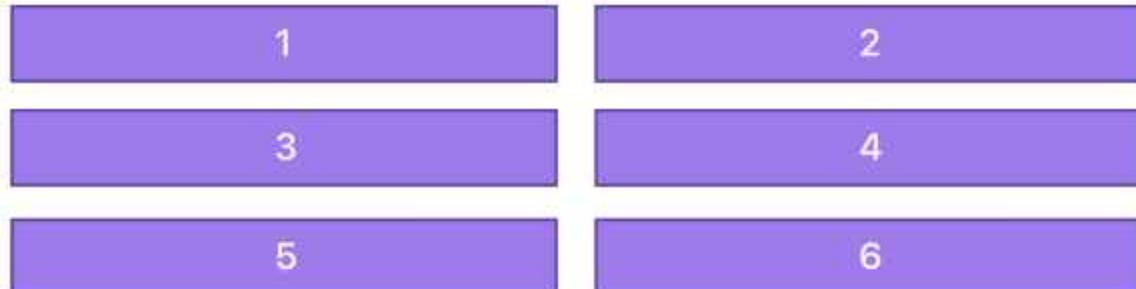
Uma coisa interessante que podemos utilizar algumas propriedade do Flexbox no CSS Grid, como justify-items, align-items, justify-content, align-content. Nesse caso, iremos utilizar e adicionar o gap de 16px entre os elementos filhos (item). Mas podemos utilizar as propriedade grid-column-gap, grid-row-gap do CSS Grid para criar espaçamentos em lugares mais específicos.



grid-template-columns

Para criar colunas, usaremos a propriedade grid-template-columns, que recebe os valores de tamanho de cada coluna separados por espaço. Iremos, por exemplo, criar duas colunas com 200px cada.





Mas também podemos criar um layout em que a primeira coluna ocupa um espaço fixo e a segunda ocupa todo o resto do espaço disponível. Isto é possível usando a unidade `fr`, introduzida exatamente para ser utilizada como valor de tamanho para linhas e colunas em `container`.

```
● ● ●
1 .container {
2   display: grid;
3   grid-template-columns: 200px 1fr;
4 }
```

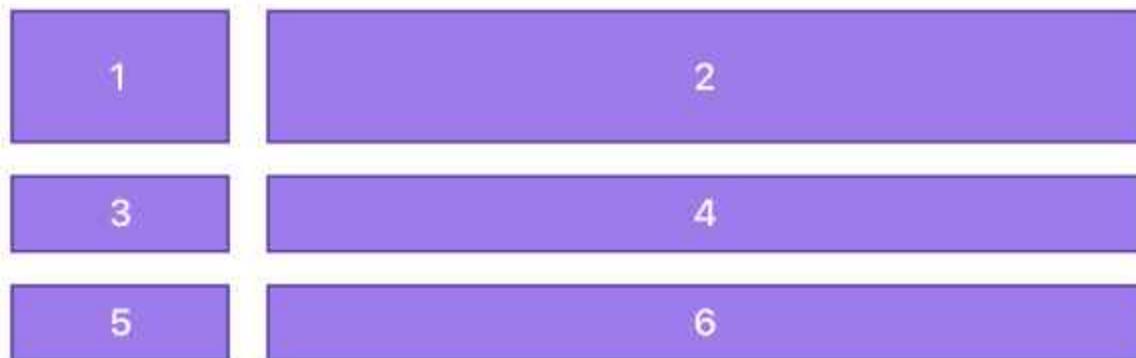


A unidade `fr` representa uma fração do espaço livre disponível em uma linha ou coluna. Podemos misturar diversos valores `fr` para formar layouts automáticos que respeitam certas proporções.

grid-template-rows

Exatamente da mesma maneira nós declaramos as linhas utilizando a propriedade `grid-template-rows`.

```
1 .container {  
2   display: grid;  
3   grid-template-columns: 200px 1fr;  
4   grid-template-rows: 200px 100px;  
5 }
```



Dentro do nosso container agora teremos 2 linhas declaradas com os valores de 200px e 100px. Assim como nossas colunas, o tamanho das linhas preenchem o espaço disponível proporcionalmente.

grid-template-areas

Você deve ter notado que nossos itens vão se organizando na grid de acordo com a ordem em que estão no HTML.

Porém, podemos dar nomes para cada área da nossa grade e depois indicar onde cada elemento deve ir.

Vamos alterar o nosso HTML. Iremos colocar as classes header, sidenav, main e footer nos itens da grid para podermos indicar a posição de cada um deles depois.

```
1 <div class="container">
2   <div class="item header">Header</div>
3   <div class="item sidenav">Sidenav</div>
4   <div class="item main">Main</div>
5   <div class="item footer">Footer</div>
6 </div>
```

Para cada item, vamos declarar a propriedade grid-area, que serve para indicar o nome da área em que cada elemento deverá ocupar na grid.

```
1 .header {  
2   grid-area: header;  
3 }  
4 .sidenav {  
5   grid-area: sidenav;  
6 }  
7 .content {  
8   grid-area: main;  
9 }  
10 .footer {  
11   grid-area: footer;  
12 }
```

Para dar nomes às áreas de uma mesma linha, escrevemos dentro de "" (aspas). Ao abrir novas aspas estaremos indicando que estamos indo para a linha seguinte.

```
1 .container {  
2   grid-template-areas: "header header"  
3                         "sidenav main"  
4                         "footer footer";  
5 }
```

No código acima, repetimos header e footer duas vezes porque declaramos duas colunas e queremos que eles ocupem as duas.

```
1 .container {  
2   display: grid;  
3   grid-template-columns: 30% 60%;  
4   grid-template-rows: 30px 80px 20px;  
5   grid-template-areas: "header header"  
6                         "sidenav main"  
7                         "footer footer";  
8 }
```

Header

Aside

Main

Footer

grid-auto-rows

Mas temos um problema. Nem sempre todo o conteúdo de um container grid será previsível. E se tivéssemos uma lista dinâmica? Nossos itens irão se comportar dessa forma:

Um

Dois

Três

Quatro

Cinco

Seis

Sete

Os elementos “Cinco”, “Seis” e “Sete” continuam organizados em colunas como os anteriores, mas eles já não seguem os mesmos tamanhos das linhas. Isso acontece porque quando temos mais elementos do que células no nosso container grid, os elementos que sobram criam novas células implícitas que ocupam as colunas disponíveis e criam novas linhas que não tinham sido declaradas.

Para controlar o grid explícito, podemos a propriedade `grid-auto-rows` para definir como o nosso grid implícito será disposto.

```
1 .container {  
2   display: grid;  
3   gap: 10px;  
4  
5   /* grid explícito */  
6   grid-template-columns: 2fr 1fr;  
7   grid-template-rows: 100px 100px 100px;  
8  
9   /* grid implícito */  
10  grid-auto-rows: 100px;  
11 }
```

Um

Dois

Três

Quatro

Cinco

Seis

Sete

Com o uso de `grid-auto-rows: 100px` definimos agora que todas as novas linhas terão 100px de tamanho. Como a propriedade `grid-auto-rows` aceita muito mais do que apenas um valor, podemos apresentar nossa lista dinâmica.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem aprender a programar em JavaScript. Aprendendo a construir funções, fazer loops, implementar variáveis e métodos de código e muitas outras estruturas de dados que são fundamentais para o desenvolvimento de aplicações web.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de ensinar como aplicar esse tratamento, você também realizará sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com medo de enfrentar problemas para entrar na área de programação, este é o seu guia. Ele ensina os conceitos básicos em suas regras. Neste guia, encontra-se uma introdução ao Node.js, GitHub, portfólio e muito mais para ajudá-lo a chegar aos resultados que sempre sonhou.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 5

JavaScript

Pronto para aprender mais um módulo? Lembrando que, ao final de todos módulos, você pode fazer testes com os conhecimentos adquiridos em um questionário no Modulo.

→ 14 TÓPICOS NESTE MÓDULO

O que é

JavaScript, ou JS é uma linguagem de programação, sendo a principal linguagem do frontend. Ela é responsável pelo dinamismo de uma página, ou seja, promove a interação com o usuário. Um exemplo clássico de JavaScript são os menus mobile, quando clicamos nele é feito uma interação com o usuário (no caso abrindo os itens do menu).

Iniciando

JavaScript é escrito em arquivos .js. Para criar um documento JavaScript é fácil, basta salvar o arquivo como script.js (lembro que você pode nomeá-lo como quiser, mas não esqueça do .js) dessa forma teremos nosso documento semipronto para o uso.

Agora iremos criar a conexão do JavaScript com nosso arquivo HTML. Antes do fechamento da tag <body> de seu documento HTML, vamos adicionar uma tag <script> para conectá-los. Isso é, na tag <script> atribuímos o atributo src="" (que vem do inglês source), esse atributo é responsável em fazer a conexão com outros arquivos.

Variáveis

Variáveis são lugares na memória onde colocamos valores para podermos trabalhar com eles posteriormente. As variáveis são um dos fatores para mantermos o código dinâmico, fácil de ser lido e compreendido Isso é, uma vez armazenado um valor em uma memória podemos utilizar seu valor ao longo do nosso código.

No JavaScript não há necessidade de declarar o tipo da variável, mas isso não significa que ela não tem tipo. Isso torna o JavaScript uma linguagem de tipagem dinâmica, o tipo é inferido pelo valor do dado e a checagem (type checking) é feita em tempo de execução (runtime). Então se você tiver uma variável chamado "nome" com o valor de "iuri" e ao longo do nosso código mudar seu valor para 10, para o JavaScript isso está certo. Isso tem seus lados bons e ruins.

Por exemplo, essa é a declaração de uma variável:

```
1 let nome = "Iuri"; // Está correto
2 nome = 10; // Está correto também
3 // Resultado final de nome é 10
```

Observação: o // são comentários. Comentários são linhas de códigos não executáveis.

A barra dupla // serve para comentar uma linha de código.

Começa com /* e termina com */ podemos comentar várias linhas de código em JavaScript.

"Ok Iuri, mas o que é esse let em nosso código?"

Bem, para declarar uma variável temos três opções: var, let e const.

- Uso do var: Uma variável declarada com var possui o que chamamos de escopo de função. Isso significa que se criarmos uma variável deste tipo dentro de uma função, você pode modificar em qualquer parte desta função, mesmo se criar outra função dentro dela.
- Uso do let: Diferente de var, declarar como let leva em conta o bloco de código onde foi declarada. Isso significa que se a declararmos dentro de um uma função, ela será "enxergada" apenas dentro deste escopo.
- Uso do const: Uma variável const é usada para definir uma constante. Diferente de var e let, as variáveis de const não podem ser atualizadas nem declaradas novamente.

Var

```
● ● ●
1 function exibirNome() {
2   var nome = "Iuri";
3 }
4
5 console.log(nome); // Erro: nome não está definido
```

Let

```
● ● ●
1 let nome = "Iuri";
2
3 if (true) {
4   let nome = "Kenay";
5   console.log(nome); // Retornará "Kenay"
6 }
7
8 console.log(nome); // Retornará "Iuri"
```

Por que isso não retorna um erro? Porque as duas instâncias são tratadas como variáveis diferentes, já que são de escopos diferentes.

Const

```
1 const nome = "Iuri";
2 nome = "Kenay";
3 // Erro: atribuição à uma variável constante.
```

Assim como as declarações de let, os consts sofrerem o hoisting para o topo do escopo, mas não são inicializadas.

Tipos de variáveis

String

Uma string é uma sequência de caracteres usados para representar texto. São declaradas usando aspas simples ou aspas duplas.

```
1 var nome = "Iuri";
2 // Ou
3 var nome = 'Iuri';
```

Number

Number é um tipo de dado numérico. O JavaScript não diferencia números inteiros, em ponto flutuante, double...



```
1 var numero = 10;  
2 var pi = 3.14;
```

Boolean

Um booleano é um tipo de dado lógico que pode ter apenas um de dois valores possíveis: true (verdadeiro) ou false (falso).



```
1 var javascript = true;  
2 var java = false;
```

Object

Objetos são tipos de dados especiais que podem armazenar vários valores ao mesmo tempo.

```
1 var pessoa = {  
2   nome: "Iuri",  
3   idade: 22,  
4   frontend: true,  
5 };
```

Function

Uma função é um conjunto de instruções que executa uma tarefa ou calcula um valor.

```
1 function nome() {  
2   console.log("Iuri");  
3 }
```

Undefined

Uma variável que não foi atribuída a um valor específico, assume o valor `undefined` (indefinido).



```
1 var texto
```

Null

O valor null é um valor nulo ou "vazio" (exemplo: que aponta para um objeto inexistente).



```
1 var texto = null;
```

Operadores

Operadores em JavaScript são símbolos especiais que envolvem um ou mais operandos com a finalidade de produzir um determinado resultado. Não irei citar todos os operadores, somente os mais utilizados. Caso queria ver todos os operadores clique [aqui](#) e leia mais.

Operadores de atribuição

"Um operador de atribuição atribui um valor ao operando à sua esquerda baseado no valor do operando à direita. O operador de atribuição básica é o igual (=), que atribui o valor do operando à direita ao operando à esquerda. Isto é, $x = y$ atribui o valor de y a x ." - MDN Web Docs

Operador	Operador encurtador
Atribuição	$x = y$
Atribuição de adição	$x += y$
Atribuição de subtração	$x -= y$
Atribuição de multiplicação	$x *= y$
Atribuição de divisão	$x /= y$
Atribuição de resto	$x %= y$

Operadores aritméticos

"Operadores aritméticos tomam valores numéricos como seus operandos e retornam um único valor numérico. Os operadores aritméticos padrão são os de soma (+), subtração (-), multiplicação (*) e divisão (/). Estes operadores trabalham da mesma forma como na maioria das linguagens de programação quando utilizados com números" - MDN Web Docs

Operador	Descrição
Módulo (%)	Retorna o inteiro restante da divisão dos dois operadores.
Incremento (++)	Adiciona um ao seu operando.
Decremento (--)	Subtrai um de seu operando.
Negação (-)	Retorna a negação de seu operando.
Adição (+)	Tenta converter o operando em um número.

Operadores de comparação

"Um operador de comparação compara seus operandos e retorna um valor lógico baseado em se a comparação é verdadeira. Na maioria dos casos, se dois operandos não são do mesmo tipo, o JavaScript tenta convertê-los para um tipo apropriado. Isto geralmente resulta na realização de uma comparação numérica." - MDN Web Docs

Operador	Descrição
Igual (==)	Retorna true caso os operandos sejam iguais.
Não igual (!=)	Retorna true caso os operandos não sejam iguais.
Estritamente igual (====)	Retorna true caso os operandos sejam iguais e do mesmo tipo.
Estritamente não igual (!==)	Retorna true caso os operandos não sejam iguais e do mesmo tipo.
Maior que (>)	Retorna true caso o operando da esquerda seja maior que o da direita.
Maior que ou igual (>=)	Retorna true caso o operando da esquerda seja maior ou igual que o da direita.
Menor que (<)	Retorna true caso o operando da esquerda seja menor que o da direita.
Menor que ou igual (<=)	Retorna true caso o operando da esquerda seja menor ou igual que o da direita.

Declarações condicionais

As estruturas condicionais são utilizadas para verificar uma condição e definir se algo deve ou não acontecer.

Podemos representar condições em JavaScript utilizando o comando de estrutura condicional if (se) da seguinte maneira:

```
1 if (condicao) {  
2     // O código será executado caso a condição  
3     // seja verdadeira  
4 }
```

Agora você deve estar se perguntando "Mas Iuri, e se essa condição for falsa?". Nesse nosso exemplo, se a condição for falsa o fluxo irá continuar sem executar nosso código dentro do if.

Declarações condicionais

Agora, caso você queira executar algo quando a condição for falsa, utilizamos o else (se não). O else sempre será executado caso o if seja falso.

```
● ● ●  
1 if (condicao) {  
2 // Será executado caso a condição seja verdadeiro  
3 } else {  
4 // Será executado caso a condição seja falso  
5 }
```

No contexto de estruturas condicionais... vamos imaginar que temos um sistema de verificação de nome, e você queria verificar apenas se dois nomes existem em nosso sistema. Você não concorda comigo que fazer dois if e else para verificar os dois nomes ficaria algo repetitivo? Ainda mais que nosso else iria retornar a mesma coisa nas duas condições.

O else if (se não se) nos ajuda a fazer mais do que uma condição, ou seja se a condição anterior não for verdadeira verifique uma nova condição.

```
● ● ●  
1 if (condicao) {  
2 // Será executado caso a condição seja verdadeiro  
3 } else if (condicaoDois) {  
4 // Será executado caso a primeira condição seja falsa  
5 } else {  
6 // Será executado caso todas as condições sejam falsas  
7 }
```

Function

Uma função, ou function é um conjunto de instruções que executa uma tarefa.

Imagine que você tenha duas tarefas, uma é executar a soma de dois números e a outra é mostrar o nome do usuário no sistema. Percebeu que essas tarefas são duas funções diferentes? Imagine essas duas tarefas espalhadas em nosso código, iria atrapalhar muito, mas com uma função teremos um agrupamento de código que faz uma determinada tarefa em nossa aplicação.

A definição da função (também chamada de declaração de função) consiste no uso da palavra-chave `function`, seguida por:

- Nome da função.
- Lista de argumentos para a função, entre parênteses e separados por vírgulas.
- Declarações JavaScript que definem a função, entre chaves {}.

Por exemplo, o código a seguir define uma função simples chamada somar:

```
function somar(numero) {  
    return numero * numero;  
}  
  
console.log(somar(10));
```

Veja que a função de somar recebe um argumento chamado numero. Isso significa que a nossa função está esperando o valor de numero (que no exemplo é 10) para multiplicar por si mesmo (numero * numero). A declaração return especifica o valor retornado pela função.

Object

Imagine que você tenha uma variável chamada pessoa e queira ter uma coleção de propriedades, como nome, idade, cidade... Isso é possível? Claro que é, isso é chamado de objeto. Objeto é uma coleção de propriedades, sendo cada propriedade definida como uma sintaxe de chave: valor. A chave pode ser uma string e o valor pode ser qualquer informação.

```
1 const pessoa = {  
2   nome: "Iuri",  
3   idade: 22,  
4 };
```

Para acessar uma propriedade de um objeto, use uma das duas notações (syntaxes):

- Notação de ponto (`objeto.propriedade`).
- Notação de array (`objeto["propriedade"]`).

```
1 pessoa.nome; // Resultado: Iuri  
2 pessoa["nome"]; // Resultado: Iuri
```

Array

Arrays, ou matrizes são lista de objetos. Elas são objetos que contêm múltiplos valores armazenados em uma lista.

Se nós não tivéssemos arrays, teríamos que armazenar cada item em uma variável separada. Isso é, uma variável chamada, frutas1, outras chamadas fruta2, fruta3, fruta4.... imagine se fosse 100 frutas. Isto seria muito mais longo de escrever e acessar.

Arrays são construídas de colchetes, os quais contém uma lista de itens separada por vírgulas.

Vamos supor que queremos armazenar a nossa lista de frutas, nós temos o seguinte código.



```
1 var frutas = ["banana", "maçã", "uva", "kiwi"];
```

Você pode acessar itens individuais em um array usando a notação de colchetes, da mesma forma que você acessa as letras em uma string.



```
1 frutas[2];
```

Observação: o índice do array sempre começa na posição 0.

Métodos de array

Arrays nos fornece vários métodos para facilitar nosso trabalho em certas situações. Com cada método que vamos abordar não estão disponíveis em objetos ou qualquer outra coisa além de Arrays.

Veremos os seguintes métodos de Array (é bom lembrar que existem outros métodos de array):

- concat()
- join()
- push()
- pop()
- shift()
- slice()
- splice()
- reverse()

Concat

O método "concat()", que serve para concatenar dois arrays, no exemplo vamos concatenar o array front com o array back.

```
1 var front = ["html", "css", "javascript"];
2 var back = ["java", "php", "node"];
3
4 front = front.concat(back);
5 console.log(front);
6 // Resultado será um array com os elementos dois arrays
```

Join

O método "join()" puxa elementos de um array e lista no formato de string.

```
1 var front = ["html", "css", "javascript"];
2 front = front.join("-");
3
4 console.log(front);
5 // Resultado será as propriedades do array
6 // separar com um traço
```

Push

O método "push()" serve para adicionarmos elementos no final do array.

```
1 var front = ["html", "css", "javascript"];
2 front.push("react");
3
4 console.log(front);
5 // Resultado será um novo elemento no final do array
```

Pop

O método “pop()” remove o último elemento de um array.

```
1 var front = ["html", "css", "javascript"];
2 front.pop();
3
4 console.log(front);
5 // Resultado será a remoção do último elemento do array
```

Shift

O método “shift()” remove o primeiro elemento do array.

```
1 var front = ["html", "css", "javascript"];
2 front.shift();
3
4 console.log(front);
5 // Resultado será a remoção do primeiro elemento do array
```

Reverse

O método "reverse()" inverte a ordem dos elementos do array.

```
1 var front = ["html", "css", "javascript"];
2 front.reverse();
3
4 console.log(front);
5 // Resultado será reversão dos elementos do array
```

Loops

Laços de repetição, ou loops como o próprio nome já diz, ele faz com que blocos de códigos sejam repetidos diversas vezes até que certa condição seja cumprida.

Um loop geralmente possui um ou mais dos seguintes itens:

- O contador, que é inicializado com um certo valor - este é o ponto inicial do loop.
- A condição de saída, que é o critério no qual o loop para - geralmente o contador atinge um certo valor.
- A expressão, que geralmente incrementa o contador em uma pequena quantidade a cada loop, sucessivamente, até atingir a condição de saída.

For

O primeiro que você usará na maior parte do tempo, é o loop for, ele tem a seguinte sintaxe:

```
● ● ●
1 for (inicializador; condição; expressão) {
2   // Código que será executado
3 }
```

No exemplo do For temos:

- A palavra-chave for, seguido por parênteses.
- Dentro do parênteses temos três itens, separados por ponto e vírgula:

- O inicializador — geralmente é uma variável configurada para um número, que é incrementado para contar o número de vezes que o loop foi executado. É também por vezes referido como uma variável de contador.
- A condição — como mencionado anteriormente, aqui é definido quando o loop deve parar de executar. Geralmente, essa é uma expressão que apresenta um operador de comparação, um teste para verificar se a condição de saída foi atendida.
- A expressão — isso sempre é avaliado (ou executado) cada vez que o loop passou por uma iteração completa. Geralmente serve para incrementar (ou, em alguns casos, decrementar) a variável do contador, aproximando-a do valor da condição de saída.

For na prática

```
● ● ●  
1 for (let i = 1; i <= 5; i++) {  
2   console.log(i); // Saída: 1, 2, 3, 4, 5  
3 }
```

Explicando: dentro do nosso laço for temos o "let i=1;" ele é o nosso inicializador, ele basicamente está falando que nossa variável de inicialização é o "i" que tem o valor de 1.

Logo em seguida temos o "i<=5", ele é o condição, como o próprio nome já diz, ele é a condição que irá executar em nosso loop até que seja cumprida. Como o valor inicial de i é 1 ele será executado até seu valor ser igual a 5.

E por último temos o "i++", ele é o iterador, ou seja, é o responsável pelo incremento da nossa variável i até chegar em 5.

While

"While funciona de maneira muito semelhante ao loop for, exceto que a variável inicializadora é definida antes do loop, e a expressão final é incluída dentro do loop após o código a ser executado - em vez de esses dois itens serem incluídos dentro dos parênteses. A condição de saída está incluída dentro dos parênteses, que são precedidos pela palavra-chave while e não por for." - MDN Web Docs

```
● ● ●  
1 inicializador;  
2 while (condição) {  
3 // Código  
4 expressão;  
5 }
```

Do while

```
● ● ●  
1 inicializador;  
2 do {  
3 // Código  
4 expressão;  
5 } while (condição);
```

Nesse caso, a condição é avaliada depois que o bloco de código é executado, ou seja, o código dentro do "do" será executado pelo menos uma vez antes de ter certeza que a condição é verdadeira.

DOM

"O Document Object Model (DOM) é uma interface de programação para os documentos HTML e XML. Representa a página de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo. O DOM representa o documento com nós e objetos, dessa forma, as linguagens de programação podem se conectar à página." - MDN Web Docs

O que são eventos no JavaScript?

Os eventos são ações que são realizadas em um determinado elemento da página, seja ele um texto ou uma imagem, por exemplo. Muitas das interações do usuário que está visitando sua página com o conteúdo do seu site podem ser consideradas eventos.

Principais eventos:

- **onload:** Disparado quando documento é carregado.
- **onunload:** Disparado quando documento é descarregado de janela ou de frame.
- **onsubmit:** Disparado quando formulário é submetido.
- **onreset:** Disparado quando formulário é "limpado" via botão de reset.
- **onselect:** Disparado quando texto é selecionado numa área de entrada de texto.
- **onchange:** Disparado quando elemento perde o foco e foi modificado.
- **onclick:** Disparado quando botão de formulário é selecionado via click do mouse.
- **onfocus:** Disparado quando o elemento recebe foco: clicando o mouse dentro do elemento ou entrando no mesmo via Tab.
- **ondblclick:** Disparado quando ocorre um click duplo do mouse.
- **onmousedown:** Disparado quando mouse é pressionado enquanto está sobre um elemento.
- **onmouseup:** Disparado quando mouse é despressionado.
- **onmouseover:** Disparado quando cursor do mouse é movido sobre elemento
- **onmouseout:** Disparado quando mouse é movido fora do elemento onde estava
- **onkeydown:** Disparado quando tecla é pressionada.

Existem diversas maneiras de se aplicar esses eventos aos elementos HTML, são elas:

- Inline.
- Em um arquivo externo, usando um manipulador de eventos.

Qual é melhor? Bem, usar a maneira de arquivo externo deixa nosso documento HTML limpo de código JavaScript. Já com inline, aplicamos diretamente na tag HTML o evento JavaScript.

```
1 <button onclick="alert('Oi')">
2   Sou um evento inline
3 </button>
```

O exemplo anterior está funcionando corretamente, aos clicarmos no botão irá disparar um evento chamado "onclick" que irá abrir uma simples caixas de diálogo (alert) porém, estamos adicionando JavaScript junto com código HTML. Irei reproduzir o mesmo evento em um arquivo JavaScript.

```
document.querySelector("button").addEventListener(  
  "click", () => {  
    alert("oi");  
  });
```

Sei que olhando esse código parece que utilizar eventos inline é mais fácil e rápido, mas estamos utilizando só um exemplo, imagina uma aplicação cheia de animações e interações do usuário, você mesmo ficaria perdido e terá um código JavaScript maior.

Tenha calma que irei explicar cada parte desse código futuramente.

Selecionando elementos HTML

Seletores são métodos que selecionam tags HTML (e seu conteúdo), seja pelo id, class ou pela própria tag. Isso é muito útil quando queremos alterar a DOM, seja adicionando novos elementos ou alterando o que já existe nela. Veremos os seguintes métodos:

- getElementById()
- getElementsByClassName()
- querySelector()
- querySelectorAll()

Observação: todas as propriedades, métodos e eventos disponíveis para manipular e criar páginas são organizados em objetos, por exemplo, o objeto document representa o próprio documento.

getElementById

Este método retorna um elemento correspondente ao id passado como parâmetro. Veja o exemplo abaixo.



```
1 let eFront = document.getElementById("ebook");
```

Este método retorna um elemento correspondente ao id passado como parâmetro. Veja o exemplo abaixo.

getElementsByClassName

Como o próprio nome já diz, essa função vai retornar os elementos que possuem uma mesma classe passada.



```
1 let eFront = document.getElementsByClassName("ebook");
```

Enquanto a função anterior retorna um único elemento, esta retorna uma `HTMLCollection` (uma coleção) de elementos.

querySelector

Diferente dos outros métodos, este método utiliza “.” para indicar a seleção de uma classes, “#” para indicar seleção de ids ou a própria tag.

```
● ● ●  
1 let teste1 = document.querySelector("#souId");  
2 let teste2 = document.querySelector(".souClass");  
3 let teste3 = document.querySelector("div");
```

Um ponto interessante do `querySelector()` só retorna o primeiro elemento encontrado que tem a mesma seleção.

querySelectorAll

Este método é similar ao método `querySelector` que também utiliza os seletores do CSS, porém retorna uma `NodeList` com todos os elementos que correspondem ao seletor criado.

```
1 let teste = document.querySelectorAll("div");
2 // Irá retornar todas as divs da página
```

Interações do usuário

Existem duas formas comuns (existem outras formas) de adicionar estilos CSS em nossa página com JavaScript:

- Estilos Inline.
- Classes CSS a elementos.

A propriedade `style` em JavaScript retorna o estilo de um elemento na forma de um objeto `CSSStyleDeclaration` que contém uma lista de todas as propriedades de estilos. Dessa forma conseguimos adicionar estilos CSS em nossa aplicação através do JavaScript.

```
1 var button = document.querySelector("button");
2
3 button.addEventListener("click", () => {
4   button.style.backgroundColor = "red";
5});
```

Perceba que diferente das propriedades do CSS, no JavaScript não temos o “-” (hifen) e substituímos ele pela convenção de nomenclatura lowerCamelCase.

Dessa forma adicionamos background-color do CSS em nosso botão de forma de estilos inline. Mas veja que isso leva a um problema que tivemos com o JavaScript dentro do HTML.

Adicionando classes CSS nos elementos

Utilizando a propriedade classList você pode obter, definir ou remover classes CSS facilmente de um elemento. O exemplo a seguir mostrará como adicionar uma classe chamada “outraCor” em um elemento <div> com id="iuricode".

```
1 var elementoDiv = document.getElementById("iuricode");
2
3 elementoDiv.addEventListener("click", () => {
4   elementoDiv.classList.toggle("outraCor");
5});
```

Uma observação: eu criei uma classe no CSS que tem a propriedade background-color, assim quando ela for chamada através do click, irá mudar a cor do elemento selecionado (que no caso é a div com o id="iuricode").

addEventListener

O método addEventListener (ou conhecido como evento de escuta) permite configurar funções a serem chamadas quando um evento acontece, que em nosso exemplo, quando um usuário clica em um botão. No exemplo a seguir mostrar que o elemento nomeElemento está sendo escutado caso o usuário faça um click.

Sintaxe do addEventListener:



```
1 alvo.addEventListener(evento, função);
```

- alvo: é o elemento HTML ao qual você deseja adicionar os eventos.
- evento: é onde você especifica qual é o tipo de evento que irá disparar a ação/função. Uma coisa interessante no evento é que diferente do JavaScript no HTML, ele não tem o "on" dos eventos como onclick, onchange, onblur...

- função: especifica a função a ser executada quando o evento é detectado. Em nossos exemplos foram utilizados a sintaxe arrow function que deixa a expressão da nossa função mais curta e anônima. Mas você pode simplesmente aplicar uma outra função código no lugar.



```
1 var nomeElemento = document.getElementById("div");
2
3 nomeElemento.addEventListener("click", () => {
4   ...
5});
```

O arrow function não terá uma explicação mais afundo pois ele é um recursos introduzido no ES6 e estamos apenas abordando os fundamentos. Mas caso queria saber sobre ele clique [aqui](#).

API com Axios

API, ou Application Programming Interface, é um conjunto de definições e protocolos para integrar softwares de aplicações. Um exemplo clássico de API é a do Google Maps. Por meio de seu código, conseguimos ter acesso a localização, muitas outras aplicações utilizam os dados do Google Maps adaptando-o da melhor forma.

O que é Axios?

Axios é uma biblioteca JavaScript baseada em promessas (promises) para interceptar requisições (requests) HTTP. Esses interceptadores são úteis quando queremos pegar ou alterar requisições HTTP. Cada requisição HTTP pode usar um dos métodos de requisição existente, temos sete tipos de requisição, mas nesse ebook iremos abordar somente o método GET para consultar os dados em uma API.

A primeira coisa a se fazer quando estamos trabalhando com uma biblioteca ou framework, é procurar o CDN da biblioteca para importá-la em nossa aplicação.

```
1 <script
2   src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
3 </script>
4 <script src="main.js"></script>
```

Dessa forma temos acesso aos recursos do Axios.

Uma observação importante: a importação do CDN tem que vir antes do nosso arquivo script que iremos utilizar para consumir a API (isso se aplica a qualquer biblioteca).

Adicionando a API

Em nosso exemplo iremos utilizar a API de usuário do GitHub.

```
1 axios.get("https://api.github.com/users/iuricode");
```

Primeiro, você importa o Axios para que possa ser usado na aplicação. Em seguida, executa uma requisição utilizando o GET através do método get() para obter uma promessa que retorna um objeto de resposta que pode dar sucesso ou erro, dependendo da resposta da API.

```
1 axios
2 .get("https://api.github.com/users/iuricode")
3 .then(function (response) {
4   console.log(response.data);
5   console.log(response.data.login);
6 })
7 .catch(function (error) {
8   console.log(error);
9 });
```

"Ok Iuri, mas o que é esse then?"

Lembra que eu falei que o Axios trabalha com promessas? Exatamente isso que está acontecendo! Caso a promessa for cumprida, o argumento `then()` será chamado; se a promessa for rejeitada, o argumento `catch()` será chamado.

Dentro do argumento `then()` estamos imprimindo duas coisas no console do navegador: `response.data` e `response.data.login`. O `response.data` está imprimindo todos os dados da nossa API no console, já o `response.data.login` está imprimindo o nosso username do GitHub.

Mas Axios não é a única solução. Temos nativamente no JavaScript o `fetch()`. Ele é perfeitamente capaz de reproduzir as principais funcionalidades do Axios! Mas o Axios é uma biblioteca que traz bastante benefícios quando estamos trabalhando com APIs.

JavaScript Assíncrono: Async/Await

Agora que você já sabe trabalhar com Promises no Javascript, iremos aprender sobre `async/await`.

O `async/await` é uma nova forma de tratar Promises dentro do nosso código, evitando a criação de cascatas de `.then` como vimos no exemplo passado.

Primeiramente gostaria de explicar como o JavaScript funciona por debaixo dos panos, tornando mais fácil o entendimento do assíncrono.

O comportamento do JavaScript de "executar uma coisa por vez". Com o JavaScript assíncrono, o comportamento vai separar seu código em duas partes: coisas que rodam AGORA, coisas que vão rodar DEPOIS de algo acontecer. Calma que já vai ficar mais fácil de entender.

Para exemplificar, podemos pensar em comunicação:

Uma ligação/chamada é um exemplo de comunicação síncrona: quando falamos ao telefone, as informações chegam e saem em sequência, uma após a outra.

Por outro lado, uma conversa online via a mensagem, como o WhatsApp, é um exemplo de comunicação assíncrona: quando enviamos uma mensagem e as informações não chegam logo em sequência, a gente espera até a outra pessoa responder.

Sintaxe

Utilizamos o prefixo `async` antes de definir uma função para indicar que estamos tratando de um código assíncrono, e com o prefixo adicionado, podemos utilizar o `await` antes das Promises indicando um ponto a ser aguardado pelo código. Já irei explicar melhor sobre `Async` e `Await`.

```
1 // Sem o Async/Await
2 function fetchUser(user) {
3   api.get().then(response => {
4     console.log(response); });
5 }
6
7 // Com o Async/Await
8 async function fetchUser(user) {
9   const response = await api.get();
10  console.log(response);
11 }
```

Veja como o código ficou mais limpo, não precisamos mais declarar os `.then` ou `.catch` e ter medo de alguma Promise não executar antes de utilizarmos seu resultado pois o `await` faz todo papel de aguardar com que a requisição retorne seu resultado.

- Async. Essa palavra pode ser usada ao criar uma função. Quando adicionamos esse identificador na criação da função, nós definimos que ela será uma função assíncrona, e o melhor, retornará uma promessa. Quando usarmos a expressão return estaremos, na realidade, resolvendo uma promessa.
- Await. Essa palavra será usada com o objetivo de esperar a resolução de uma função assíncrona. Se houver uma série de funções assíncronas, a expressão await definirá que o código só terá sequência quando a função anterior for resolvida. Um detalhe muito importante: a expressão await só será aceita em uma função que já for assíncrona, ou seja, que possuir o identificador async.

Vamos substituir o then?

O `async/await` surgiu como uma opção mais "clean" ao `.then()`, mas é possível usar os dois métodos em um mesmo código. O `async/await` simplifica a escrita e a interpretação do código, mas não é possível comparar os dois, pois possuem funcionamentos lógicos diferentes. Ao usarmos o `then`, as Promises são executadas em paralelo, enquanto o `async/await` trata as Promises de forma sequencial, como se estivesse realmente executando um código síncrono, que imprime os resultados um após o outro.

Confuso ainda? Vamos esclarecer agora!

Um exemplo: uma inserção de dados no banco pode demorar um pouco, então para receber uma resposta positiva antes de tomar outra ação, condicionamos a inserção em uma função assíncrona.

Assim é possível esperar todo o processo do banco de dados finalizar para darmos andamento ao nosso sistema.

```
1 function primeiraFuncao() {  
2   console.log("Segundo");  
3 }  
4  
5 async function segundaFuncao() {  
6   await primeiraFuncao()  
7  
8   console.log("Primeiro");  
9 }  
10  
11 segundaFuncao()
```

Imagine que dentro da função “primeiraFunção” tenha as informações do bando de dados, é que o console “Primeiro” dependa das informações da função “primeiraFuncao” para realizar a ação. Para que a função “segundaFuncao” seja executada com as informações necessária, precisamos transforma-la em uma função assíncrona.

Sei que esse conceito é um pouco complexo de entender, mas com muitas práticas você irá entender melhor o funcionamento de funções assíncronas no JavaScript.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem aprender a programar em JavaScript. Aprendendo a construir funções, fazer loops, implementar variáveis e métodos de código e muitas outras estruturas de dados que são fundamentais para o desenvolvimento de aplicações web.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de ensinar como aplicar esse tratamento, você também realizará sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com medo de enfrentar problemas para entrar na área de programação, este é o seu guia. Ele ensina os conceitos básicos em suas regras. Neste guia, encontra-se uma introdução ao Node.js, GitHub, portfólio e muito mais para ajudá-lo a chegar aos resultados que sempre sonhou.

[QUERO COMPRAR TAMBÉM](#)



Módulo 4

Sass

Prezado(a) professor(a),
você está no módulo 4 do curso de
Front-end. No final de cada módulo, você pode fazer
conhecimentos por meio de um questionário no Moodle.

* 9 TÓPICOS NESTE MÓDULO

Introdução

Assim como qualquer linguagem, sempre temos que evoluir e com CSS não é diferente. Neste pequeno módulo iremos abordar o famoso Sass. Com ele, podemos separar o nosso CSS em módulos, criar funções, implementar lógica de programação, e muito mais. Mas fique tranquilo, Sass e CSS são muito semelhantes, então se você tem a base de CSS a curva de aprendizado com Sass será curta. Além disso, depois que você aprender a utilizar Sass em suas aplicações nunca mais vai voltar pro CSS puro.

O que é

A sigla Sass significa "Syntactically Awesome Style Sheets" – ou seja, Folhas de Estilo com Sintaxe Espetacular. A sua ideia é adicionar recursos especiais ao escrever estilos como mixins, importar grupos de estilos e outras opções que iremos comentar. Assim tornando o processo de desenvolvimento de estilos mais simples e eficiente do que com o puro CSS.

Iniciando

É bem simples iniciar com Sass, basta entrar em seu editor de código e criar um arquivo com o formato .scss.

Dessa forma estamos criando um documento Sass que irá criar estilos para o nosso documento HTML. Mas tenha calma, talvez você tenha tentando chamar o documento Sass no HTML, mas não deu certo.

Isso aconteceu porque estamos estilizando com Sass, mas não quer dizer que o nosso documento Sass irá ser referenciado no HTML, mas sim o arquivo CSS.

Confuso né?

O que vai acontecer vai ser o seguinte: vamos desenvolver os estilos em Sass, mas no final iremos ter um CSS normal referenciado no HTML.

"Espera luri, quer dizer que todo nosso estilo com Sass vai virar um CSS no final?"

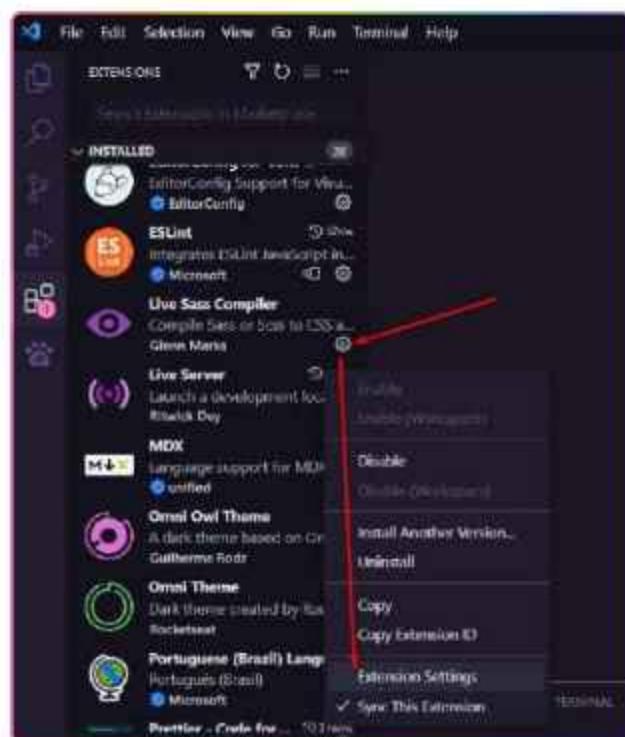
Exato! Mas calma, isso vai trazer muitas vantagens que iremos descobrir ao decorrer do módulo.

"Ok Iuri, mas como tenho acesso a esse CSS que foi feito pelo Sass?".

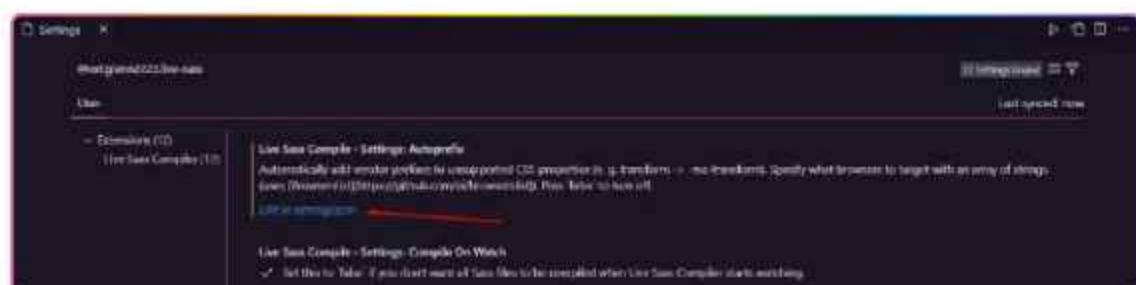
Primeiro precisamos de uma ferramenta ou extensão para transpilar o documento Sass para CSS. Recomendo a extensão "Live Sass Compiler".

Primeiro vamos criar o diretório onde será "cuspido" o nosso CSS e assim referenciar ele no nosso HTML.

Desse modo, depois de instalar a extensão "Live Sass Compiler" clique na engrenagem e vá em "Configurações de Extensão".



Depois clique em "Editar em settings.json".

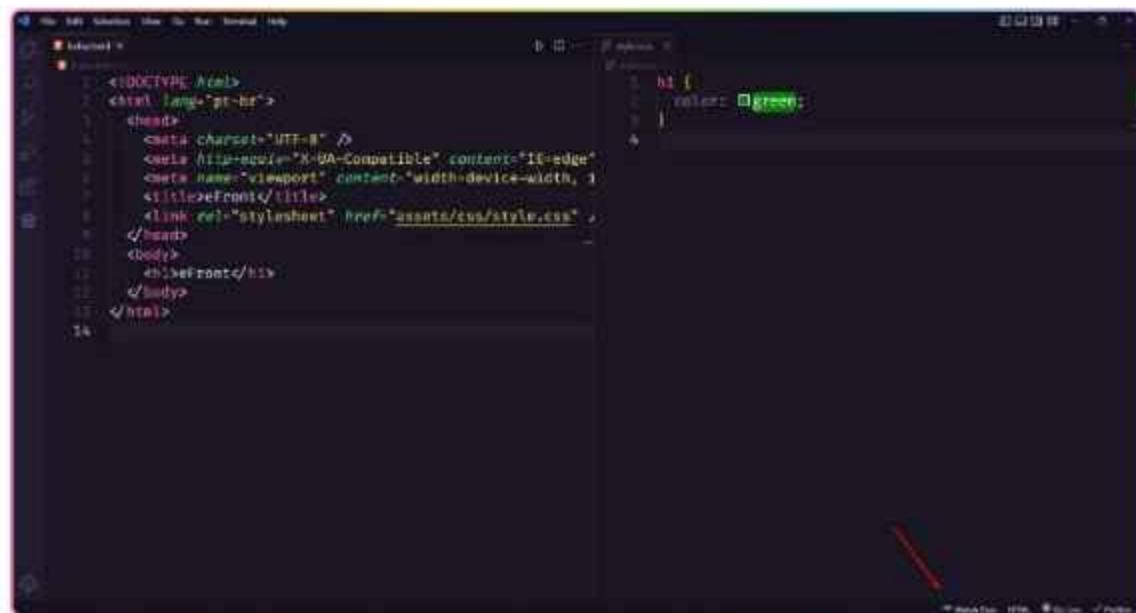


Na aba que se abre procure no "liveSassCompile" o "savePath" nele iremos colocar o caminho onde será transpilado o arquivo CSS. Veja que também temos outras duas opções, uma é o "extensionName" ele falam qual será o formato que será transpilado, que em nosso caso é o .css e temos também a opção "format" que diz que se o CSS "cuspido" será extenso ou minificado.

```
"editor.defaultFormatter": "esbenp.prettier-vscode",
"liveSassCompile.settings.formats": [
  {
    "format": "compressed",
    "extensionName": ".css",
    "savePath": "assets/css/"
  }
],
"vsicons.dontShowNewVersionMessage": true.
```

Vamos fazer um teste para ver se nosso CSS está sendo transpilado corretamente. Para isso, crie um estilo qualquer e clique em "Watch Sass" localizado no barra inferior do editor de código (lembrando que essa opção só irá parecer se tiver utilizando a extensão "Live Sass Compiler").

Ao fazer esse processo o próprio Sass deverá criar automaticamente o diretório, sendo assim precisamos só referenciar no nosso HTML o CSS transpilado pelo Sass.



The screenshot shows a code editor window with an HTML file open. The HTML code includes a link to a CSS file:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Front</title>
    <link rel="stylesheet" href="assets/css/style.css" />
  </head>
  <body>
    <h1>Front</h1>
  </body>
</html>
```

```
1 <link rel="stylesheet" href="assets/css/style.css">
```

Sintaxe

O Sass tem duas sintaxe diferentes, depende do formato do arquivo.

 .SCSS

```
1 button {
2   padding: 10px 20px;
3   border-radius: 5px;
4 }
```

 .sass

```
1 button
2   padding: 10px 20px
3   border-radius: 5px
4
```

SCSS: É bastante semelhante ao CSS. Você pode até dizer que SCSS é um superconjunto de CSS, o que significa que todo CSS válido também é SCSS válido. Devido à sua semelhança com CSS, é a sintaxe SASS mais fácil e popularmente usada.

SASS: Essa sintaxe tem todos os mesmos recursos do SCSS, a única diferença é que o SASS usa indentação em vez das chaves e ponto-e-vírgulas do SCSS.

Comentários

Os comentários no Sass funcionam de maneira semelhante aos comentários em outras linguagens, como JavaScript. Comentários de linha única começam com // e vão até o final da linha. Esses comentários não são mostrados no CSS compilado, somente comentários com /**

```
1 // Este comentário não irá aparecer no CSS.  
2 /* Mas este comentário irá, exceto no modo compactado. */
```

Variáveis

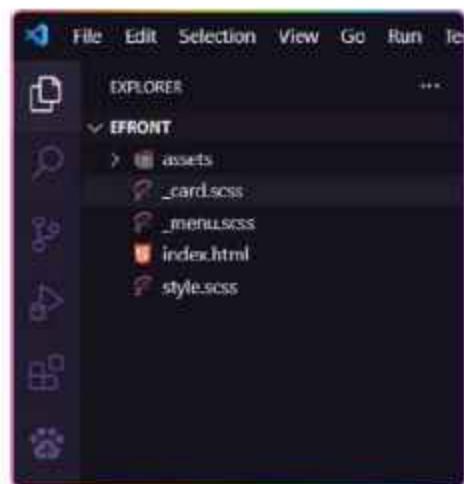
Podemos declarar uma variável utilizando o operador \$, então passamos a propriedade. No Sass geralmente utilizamos variáveis para guardar o valor de uma cor, fonte, tamanho da fonte, largura ou altura, etc.

```
1 $cor: #0f0;  
2  
3 h1 {  
4   color: $cor;  
5 }
```

Import/Use

Muitas vezes queremos trabalhar em arquivos independentes (como componentes no React) pois, isso nos trás a vantagem na manutenção já que não utilizamos só um documento em toda nossa aplicação. Exemplo, se você tem um menu e um card no seu site, você pode separar os estilos deles em parciais (parciais são arquivos que são incluídos, mas não transpilam para arquivos CSS), isso é, podemos ter três arquivos .scss e somente um será responsável por transpilado para um documento CSS.

Para incluir um parcial, você precisa seguir uma convenção de nomenclatura, que é colocar um `_` (underline) no início do nome de cada arquivo. Ao fazer a importação, você não precisa colocar o underline (é opcional). Muitas vezes você vai ver esses arquivos parciais sendo importado com `@import` ou `@use`.



O arquivo `_card.scss` e `_menu.scss` são nossos arquivos parciais, já o `style.scss` é o arquivo responsável por receber todas essas parciais e compilar para CSS.

```
1 @import "menu.scss";
2 // Importando o _menu.scss no documento style.scss
```

Extend

O @extend permite compartilhar um conjunto de propriedades de um seletor para outro. Isso é muito útil se você tiver elementos de estilo quase idênticos que diferem apenas em alguns pequenos detalhes.

```
1 .button-simples {
2   font-size: 16px;
3   color: #0f0;
4 }
5
6 .button-vermelho {
7   @extend .button-simples;
8   background-color: red;
9 }
```

No exemplo acima tanto a Classe .button-simples e .button-vermelho tem font-size de 16px e cor de texto em #0f0.

Esse é o resultado final do nosso CSS transpilado:

```
1 .button-simples, .button-vermelho {  
2   font-size: 16px;  
3   color: #0f0;  
4 }  
5  
6 .button-vermelho {  
7   background-color: red;  
8 }
```

Mixin

Algumas coisas em CSS são um pouco trabalhosa de escrever. Um Mixin permite criar grupos de declarações CSS que você deseja reutilizar em todo o site. Você pode até passar valores para tornar seu Mixin mais flexível.

```
1 @mixin theme {  
2   font-size: 14px;  
3   color: #fff;  
4 }
```

```
1 .erro {  
2   @include theme;  
3   background-color: red;  
4 }
```

Agora você deve estar se perguntando: "Mas luri, isso é praticamente o extend". De fato! Se usarmos Mixins dessa forma não estamos aproveitando seu potencial.

Passando variáveis para um Mixin

Agora iremos utilizar o potencial do Mixin. Os Mixins aceitam argumentos. Dessa forma, você pode passar variáveis para um Mixin.

```
1 @mixin theme {  
2   background-color: $cor;  
3 }  
4  
5 .erro {  
6   @include theme($cor: red);  
7 }
```

No exemplo estamos dizendo que temos um mixin que espera receber um valor para o nosso background-color através do \$cor.

Valor padrão para um Mixin

Também é possível definir valores padrão para variáveis do Mixin:

```
● ● ●
1 @mixin theme($cor: green) {
2   background-color: $cor;
3 }
4
5 h1 {
6   @include theme();
7 }
8
9 .erro {
10  @include theme($cor: red);
11 }
```

No exemplo estamos dizendo que caso não seja definido o \$cor no @include ele receba o valor padrão de "green".

Nested

O Sass permitirá que você aninhe seus seletores CSS de uma maneira que siga a mesma hierarquia visual do seu HTML. Enquanto em CSS, as regras são definidas uma a uma (não aninhadas):

.SCSS

```
1 section {  
2   padding: 80px;  
3   h2 {  
4     font-size: 32px;  
5   }  
6 }
```

.CSS

```
1 section {  
2   padding: 80px;  
3 }  
4 section h2 {  
5   font-size: 32px;  
6 }
```

Adquira meus outros materiais com 10% de desconto!

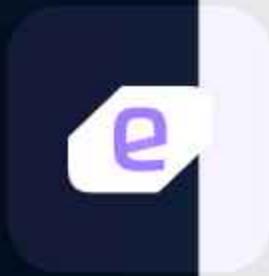
UTILIZE O CUPOM: **FRONTIO**


Este material é destinado a pessoas que querem aprender a programar.
Aprendendo com exemplos simples, fazer depuração e entender
o funcionamento de código é muito mais fácil quando se entende o que
está acontecendo por trás das linhas de código.

[QUERO COMPRAR TAMBÉM](#)


Adquira este material e obterá um **guião de boas práticas para a
criação de interfaces**. Além de ensinar como aplicar
tratamentos específicos ao redor da sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)


Se você está tentando resolver problemas para entrar na área de
programação, este é o seu guia. Ele explica os conceitos básicos em suas
mídias. Aprenda sobre estruturas de dados, algoritmos, lógica, matemática, etc.
Portanto, é perfeito para quem quer aprimorar os resultados.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 05

Bootstrap

Resumo café expresso sobre o módulo de Bootstrap. Lembrando que, ao final de cada módulo, você pode fazer testes e questionários por meio de um questionário no Moodle.

* 8 TÓPICOS NESTE MÓDULO

Introdução

À medida que você evolui como desenvolvedor frontend, é mais provável que você use tecnologias que o ajudem a fazer mais escrevendo menos código. O framework Bootstrap é uma maneira de se conseguir isso.

O que é

Bootstrap é um framework frontend que fornece estruturas de CSS para a criação de sites e aplicações responsivas de forma rápida e simples. Ele foi desenvolvido para o Twitter por um grupo de desenvolvedores e se tornou uma das estruturas de frontend e projetos de código aberto mais populares do mundo.

A principal característica do Bootstrap é a responsividade do site, ou seja, seu objetivo é permitir que os elementos da página sejam readaptados para o acesso em diferentes dispositivos, como notebooks, tablets, smartphones e, até mesmo, para monitores maiores que os tradicionais.

O que é framework CSS

Frameworks CSS são conjuntos de componentes que provêem uma estrutura básica de elementos reutilizáveis. São utilizados principalmente para construção de páginas na web. Após integrá-los ao seu próprio código, fica muito mais fácil organizar layouts, inserir componentes personalizados e criar sites totalmente responsivos.

Os frameworks CSS geralmente apresentam definições de formatação os elementos mais comuns de uma página: formulários, cabeçalhos, estilos de textos e imagens. Alguns apresentaram opções para a estruturação do conteúdo baseado em grids.

Esse tipo de ferramenta é muito utilizado para agilizar o trabalho, para você tem ideia de todos os benefícios, vamos listar uma série de vantagens possíveis, confira:

- Construção acelerada: Com um framework CSS é possível agilizar todo o processo de criação deixando tudo mais rápido.
- Compatibilidade entre navegadores: Uma coisa bastante comum é quando um navegador interpreta de modo diferente um mesmo código fonte. Mas isso pode ser resolvido ao utilizar um framework.

- Manutenção: Ao utilizar esse tipo de ferramenta no desenvolvimento fica muito mais fácil a criação e manutenção.
- Trabalho em grupo: Em projetos em grupo esse método pode ser muito vantajoso, visto que a agilidade no desenvolvimento é um fator importante.

Iniciando

Para iniciar com Bootstrap é bem simples. Só precisamos incluir o CSS e JavaScript que contém o código de todos os componentes prontos do Bootstrap via CDN sem a necessidade de nenhuma etapa de instalação.

Para isso basta colocar a tag `<link>` dentro da tag `<head>` e a tag `<script>` antes do fechamento `</body>`.



```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport"
6       content="width=device-width, initial-scale=1.0" />
7   ...
```

```
8 ...
9   <title>Bootstrap</title>
10  <link
11    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"
12    rel="stylesheet"
13    crossorigin="anonymous"
14  />
15  </head>
16  <body>
17
18
19  <script
20    src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.min.js"
21    crossorigin="anonymous"
22  ></script>
23
24
25 </body>
26 </html>
```

Você pode acessar esse código clicando [aqui](#).

Pronto! Somente com esses dois CDNs já está tudo pronto para utilizar os componentes do Bootstrap.

Componentes

Na documentação, você encontra uma [série de componentes](#) como referência para suas próprias criações, como alertas, botões, menu, formulários, entre muitos outros. Então só irei abordar alguns, e não irei apresentar o código de cada componente pois são enormes.

Alerts

O Bootstrap possibilita a configuração de forma simples e rápida de diferentes tipos de alertas, com cores específicas, de acordo com a situação. Para mostrar um alerta ao usuário para indicar atenção, por exemplo, basta utilizar a classe `.alert-danger` e será exibido uma caixa de texto com o fundo vermelho. Veja no exemplo abaixo:



```
1 <div class="alert alert-danger" role="alert">
2 Olá, sou um alert do Bootstrap!
3 </div>
```

Olá, sou um alert do Bootstrap!

Carousel

Um componente muito utilizado no Bootstrap é o Carousel. Trata-se de um slideshow, ou seja, uma ferramenta que permite a exibição de imagens de forma responsiva. Ele também possibilita a inclusão de efeitos especiais para a transição da imagem e controles de exibição, como os indicadores de próximo e anterior.

Navbar

Outra ferramenta que não podemos deixar de citar é o navbar. Com ele, é possível criar menus de navegação responsivos e com um layout agradável ao usuário. Também é útil em formulários, além de permitir a inserção de pedaços de texto.

Card

Um cartão, ou card é um container de conteúdo. Inclui opções para cabeçalhos e rodapés, uma ampla variedade de conteúdo, cores de fundo contextuais e opções de exibição poderosas. Geralmente muito utilizado em portais de notícias.

Pagination

A paginação indica que existe uma série de conteúdo relacionado em várias páginas. O Bootstrap constrói a paginação com elementos HTML de lista para que os leitores de tela possam anunciar o número de links disponíveis. Além que também que o código da paginação é empacotado na tag <nav> para identificá-lo como uma seção de navegação para leitores de tela.

Buttons

O Bootstrap inclui vários estilos de botão predefinidos, cada um servindo a seu próprio propósito semântico, com alguns extras incluídos para maior controle.



```
1  <button type="button" class="btn btn-danger">
2    Eu sou um botão
3  </button>
```

Eu sou um botão

Cores

As cores estão disponíveis por meio de variáveis Sass e CSS com o objetivo de facilitar a personalização dos componentes. As cores no Bootstrap são definidas dessa forma:

```
1 <div class="p-3 rounded-3 bg-info">
2   Sou um fundo azul
3 </div>
```



Para cor de texto aplicamos da mesma forma, só mudar o "bg" para "text".

Spacing

O Bootstrap contém um sistema de espaçamento que ajuda a manter uma escala consistente em todo design no site. Tudo que você tem que saber é a sintaxe para aplicar cada utilidade.

A seguir estão as utilidades para adicionar padding aos seus elementos:

- p: determina o preenchimento de todo o elemento
- py: determina padding padding-top e padding-bottom
- px: determina padding-left e padding-right
- pt: determina padding-top
- pr: determina padding-right
- pb: determina padding-bottom
- pl: determina padding-left

Para aplicá-los aos seus elementos, você teria que usar os números apropriados fornecidos pelo Bootstrap:

```
● ● ●  
1 <div class="p-3 mt-5 rounded-3 bg-info">  
2 Sou um fundo azul com margin-top e padding  
3 </div>
```

As utilidades pré-definidas para padding e margin são muito similares. Você só tem que substituir o "p" por um "m".

Sizing

O Bootstrap também tem um sistema que define a largura e altura de um elemento. As utilidades pré-definidas para padding e margin são muito similares. Você só tem que substituir o "w" por um "h".

Abaixo definimos a largura de 25% no componente alert.

```
1 <div class="p-3 mt-5 w-25 rounded-3 bg-info">  
2 Sou um fundo azul com largura de 25%  
3 </div>
```

Width 25%

Width 50%

Width 75%

Width 100%

Width auto

Responsividade

O container garante que o seu layout vai ficar alinhado corretamente na página. Ele pode definir as margens laterais da página, ou deixar sem margens e, também, posiciona o conteúdo no centro da página.

O código abaixo mostra um exemplo de container implementado em uma página com Bootstrap:

```
1 <div class="container">
2 ...
3 </div>
```

	Extra small <576px	Small ≥ 576px	Medium ≥ 768px	Large ≥ 992px	X-Large ≥ 1200px	XX-Large ≥ 1400px
.container	100%	540px	720px	960px	1140px	1320px
.container-sm	100%	540px	720px	960px	1140px	1320px
.container-md	100%	100%	720px	960px	1140px	1320px
.container-lg	100%	100%	100%	960px	1140px	1320px
.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

Columns & Rows

As rows (linhas) definem as divisões horizontais do seu layout. Essas rows devem ficar dentro do container, e podem ser aplicadas a qualquer tag que defina estrutura, como div, header e footer. Para definir um container de linha basta colocar a classe "row":

```
1 <div class="container">
2   <div class="row">
3     ...
4   </div>
5 </div>
```

Seu layout pode ter quantas rows forem necessárias. E você pode colocar rows dentro de rows, também. As rows sempre irão ficar uma abaixo da outra.

As columns (colunas) definem as divisões verticais das rows do seu layout. Columns devem estar sempre dentro das rows, e elas definem espaços na row para que você coloque os conteúdos.

Para criar uma column, você pode criar uma div com os prefixos pré-definidos pelo Bootstrap, como no exemplo:

```

1 <div class="container">
2   <div class="row">
3     <div class="col-md-6"></div>
4     <div class="col-md-6"></div>
5   </div>
6 </div>

```

Neste exemplo, temos uma linha (.row) com duas colunas (.col-md-6).

O número 6, no final de cada classe de coluna, define o espaço que ela ocupa na linha. Assim, neste exemplo, teríamos a linha (row) dividida exatamente no meio por duas colunas, já que usamos o número 6 dentro de uma possibilidade de 12 colunas.

Os prefixos de colunas servem para indicar em quais tipos de tela a coluna vai se manter posicionada. Os prefixos têm o seguinte padrão:

Para celulares	Para tablets	Para desktops	Para telas grandes
.col-xs-*	.col-sm-*	.col-md-*	.col-lg-*

Você deve substituir os asteriscos pelo tamanho da coluna.

Por último não menos importante, Bootstrap Icons.

Bootstrap Icons é uma biblioteca com mais de 1.800 ícones, nele temos ícones line e preenchidos para usar em seus projetos. Para a sua utilização basta adicionar o CND e adicionar a tag "i" com o nome do ícones que deseja. Você pode ver a lista completa de ícones [aqui](#).

```
1 <link rel="stylesheet"
2   href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.3/font/bootstrap-icons.css"
3 >
```

```
1 <i class="bi-alarm"></i>
```

Observação: sempre teremos o "bi-" antes do nome do ícone.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fiquei desejoso de implementar o conhecimento de design que já havia aprendido e também para quando desejasse formar uma startup.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, trazendo resultados melhores para suas competências de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **medo** de problemas para entrar na área de programação, esse é o seu guia para começar a programar em suas regras. Pensei sobre isso, ensinei pessoas comuns e fiz o **curso online**, perfeito e gratuito, para quem quer achar que pode realmente ser.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 6

React

Reúno os cafés provando novas técnicas de estudo. Lembrarei de que, ao final de cada módulo, você pode fazer testes com os conhecimentos adquiridos em um questionário no Moodle.

→ 10 TÓPICOS NESTE MÓDULO

Introdução

Essa sem dúvida é a tecnologia mais requisitada do mercado frontend. Quando queremos falar em flexibilidade no desenvolvimento com aplicações rápidas e de alta escalabilidade, o React vem logo em mente. O React irá te mostrar um novo mundo do JavaScript e o poder que ele tem nas criações de sites. Você está pronto para entrar para o mundo do React? Então vamos nessa!

O que é

React é uma biblioteca JavaScript para a criação de interfaces de usuário que utilizam o padrão SPA (Single Page Application).

"Está bem luri, mas o que é uma SPA?"

SPA é um padrão em que seu carregamento dos recursos (como CSS, JavaScript e HTML das páginas) ocorre apenas uma única vez: na primeira vez em que o usuário acessa a aplicação. Isso faz com que o carregamento das novas páginas seja bem mais rápido.

React foi criado pelo Meta (antiga empresa Facebook) em meados de 2011. Com o intuito de resolver um problema que os engenheiros do Facebook identificaram, que é a renderização da aplicação. Com React, na medida em que os dados mudam apenas os pedaços (que contém os dados) de uma determinada parte é renderizada. Diferente quando trabalhamos em uma aplicação feita com HTML, CSS e JavaScript onde a renderização é feita na página toda.

"E como isso é possível?"

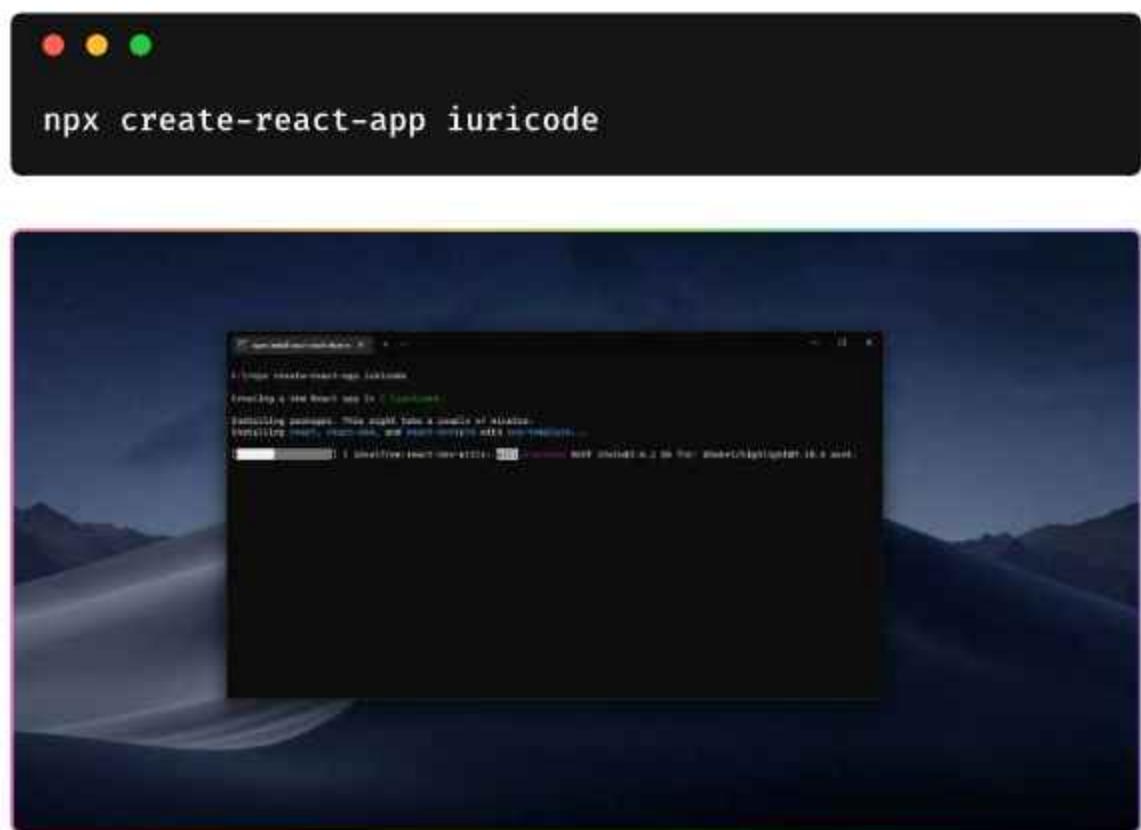
Com o React, as interfaces do usuário são compostas por pedaços de código isolados, chamados de "componentes". Isso torna o código mais fácil de dar manutenção e reutiliza o mesmo código para outras funcionalidades.

Create React App

O CRA (create-react-app) é um conjunto de ferramentas e funcionalidades pré-configuradas para que você possa iniciar um projeto React.

Além de configurar seu ambiente de desenvolvimento para utilizar as funcionalidades mais recentes do JavaScript, ele fornece uma experiência de desenvolvimento agradável, e otimiza a sua aplicação para produção. Será necessário ter o [Node](#) na sua máquina para criar um projeto em React.

Após instalar o Node já podemos criar um projeto, para isso damos o seguinte comando em nosso terminal:



Explicando o comando:

- npx: npx é um executor responsável por executar as bibliotecas que podem ser baixadas do site npm.
- create-react-app: responsável por criar as funcionalidades.
- iuricode: nome do projeto.

Após o npx instalar os pacotes, iremos dar o seguinte comando para entrar na aplicação criada.



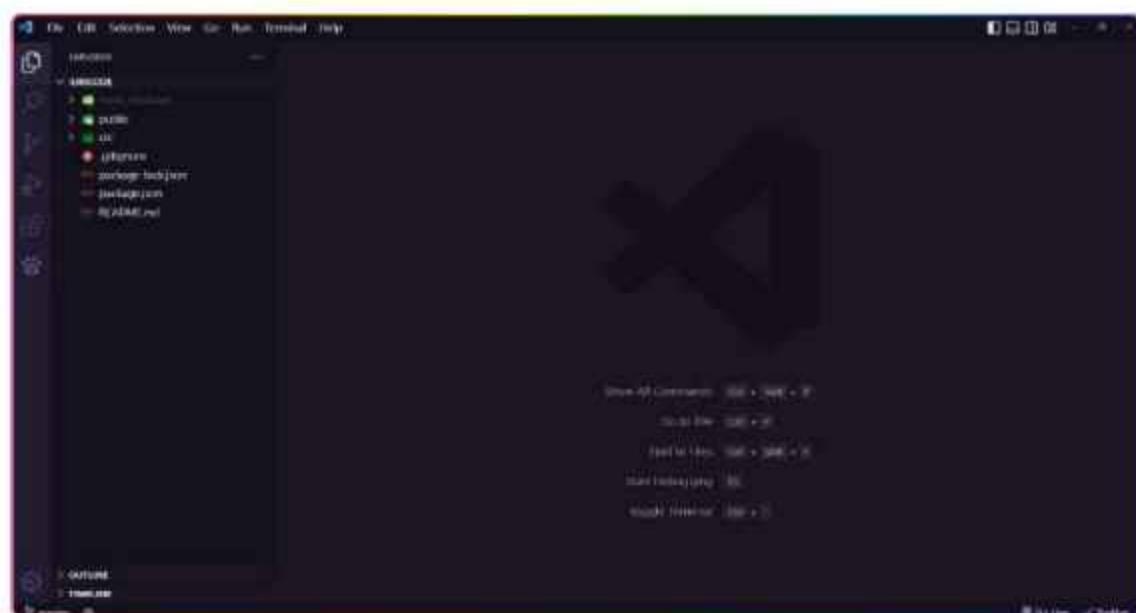
```
cd iuricode
```

- cd: permite a mudança do diretório atual, nesse caso acessamos a pasta iuricode

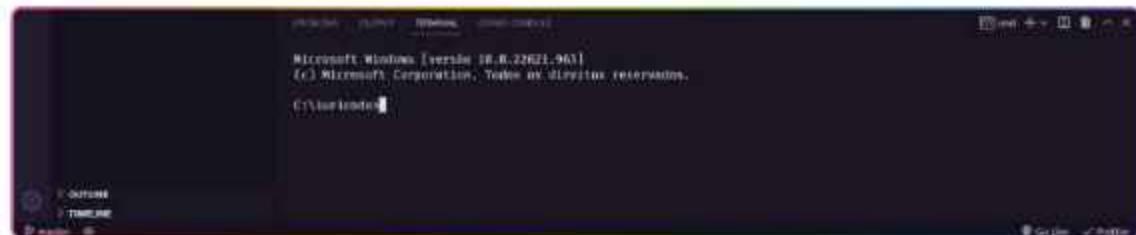
Depois de estar no diretório do projeto criado podemos dar o seguinte comando para abrir o projeto dentro do nosso Visual Studio Code:



```
code .
```



Após abrir o Visual Studio Code pressione as teclas Ctrl + ' (aspas simples), dessa forma irá abrir um terminal integrado no Visual Studio Code.



Em nosso terminal integrado iremos dar o seguinte comando para rodar nossa aplicação:

```

    You can now view duriicode in the browser.

    Local:          http://localhost:3000
    On Your Network: http://10.146.1.111:3000

    Note that the development build is not optimized.
    To create a production build, run npm run build.

    webpack compiled successfully
  
```

The terminal window shows the command 'npm start' being run. The output indicates that the application is running at port 3000 and provides the URL 'http://localhost:3000'. It also mentions that the development build is not optimized and suggests running 'npm run build' for a production build.

- npm start: é usado para executar a aplicação.

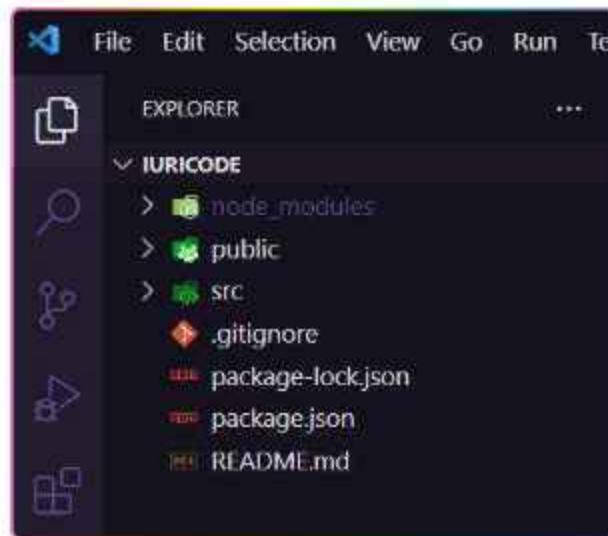
Após realizar o comando irá abrir (automaticamente) uma aplicação com o padrão do CRA na porta 3000 (três mil). Caso não seja iniciado automaticamente a aplicação no navegador você pode acessá-la em:

<http://localhost:3000/>

Uma coisa super importante é que às vezes será preciso parar o comando npm start e rodar novamente. Em muitos dos casos será necessário fazer essa tarefa quando mexemos nas configurações do projetos, mudamos os formatos dos arquivos, instalamos um novo pacote e entre outros casos.

Estrutura da pasta

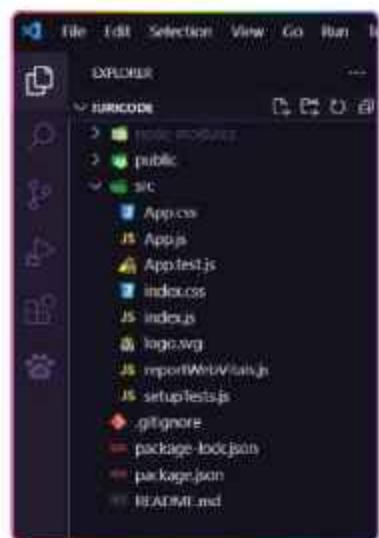
Está bem, mas o que exatamente o CRA criou para nós, luri? Como dito anteriormente, o CRA criou um conjunto de ferramentas e funcionalidades pré-configuradas para a gente iniciar nossa aplicação React. Na imagem a seguir iremos ver quais foram os conjuntos criados:



- **node_modules**: É o diretório criado pelo npm para rastrear cada pacote que você instala. Dependemos dele para que nossos pacotes sejam executados.
- **public**: que contém os recursos públicos da aplicação, como imagens. Além disso, temos um arquivo HTML que contém o id "root". É neste id que nossa aplicação React será renderizada e vai ser exibida.

- `src`: Contém os arquivos do componente React que é o responsável por renderizar a aplicação toda.
- `.gitignore`: Aqui ficaram os arquivos ignorados na hora de subir no Git.
- `package-lock.json`: Ele descreve as características das dependências usadas na aplicação, como versões, sub-pendências, links de verificação de integridade, dentre outras coisas.
- `package.json`: É um repositório central de configurações para ferramentas. Também é onde o npm armazena os nomes e versões de todos os pacotes instalados.
- `README.md`: É um arquivo que contém informações necessárias para entender o objetivo do projeto.

Dentro da pasta `src` temos alguns arquivos que não iremos utilizar na aplicação, mas antes deletá-los irei explicar sobre os mesmos pois é sempre bom saber para que servem.



- App.js: Esse arquivo contém todo o conteúdo que aparece em tela quando damos o comando npm start.
- index.js: Esse arquivo é feito a conexão do código React com o arquivo index.html que contém o id "root".
- App.test.js: É um arquivo de teste, ele é executado quando você roda o comando npm test.
- App.css & index.css: São arquivos de estilos CSS.
- logo.svg: Esse arquivo SVG é a logo do React.

- reportWebVitals.js: Esse arquivo permite medir o desempenho e a capacidade de resposta da sua aplicação.
- setupTests.js: Tudo o que ele possui são alguns métodos expect personalizados.

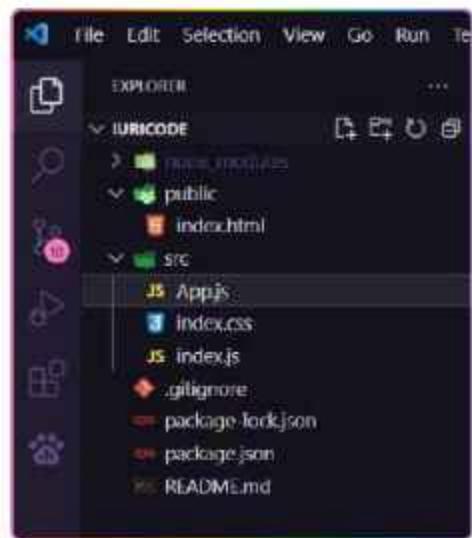
Geralmente deletamos os seguintes arquivos dentro da pasta src:

- App.css
- logo.svg
- reportWebVitals.js
- setupTests.js
- App.test.js

Além desses arquivos que não serão utilizado por nós, iremos limpar também os arquivos dentro da pasta public e as importações das imagens no index.html:

- favicon.ico
- logo192.png
- logo512.png
- manifest.json
- robots.txt

O resultado final da nossa aplicação será essa:



JSX

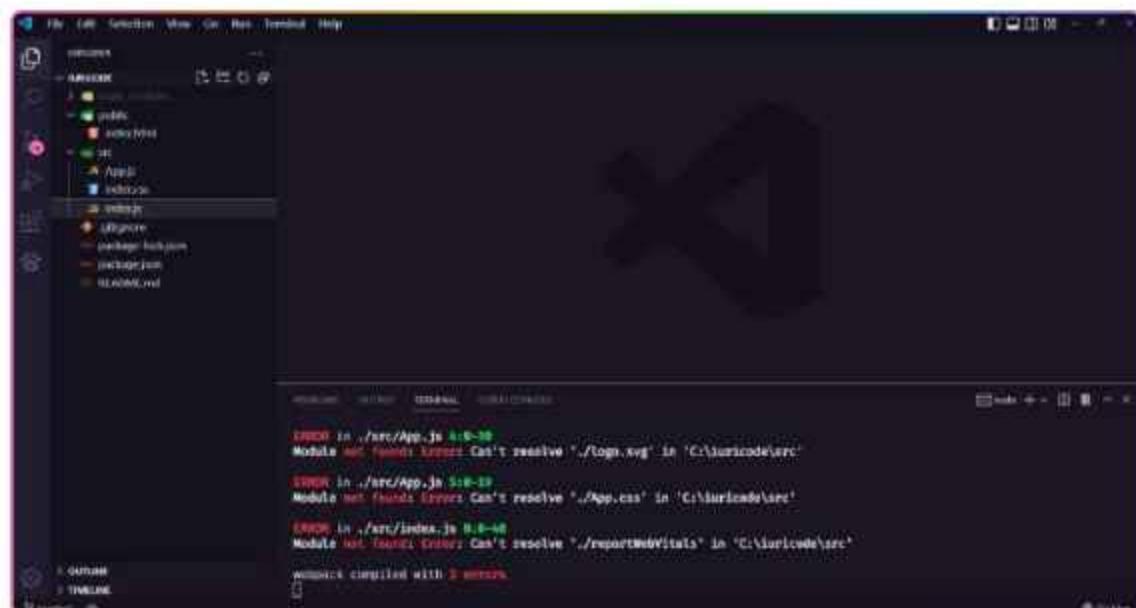
No topo do arquivo index.js, é importando o React, ReactDOM, index.css, App e serviceWorker. Ao importar o React, você está, na verdade, extraíndo código para converter o JSX em JavaScript. JSX são os elementos similares ao HTML.

"Mas o que é esse JSX, Iuri?"

Essa é uma extensão de sintaxe especial e válida para o React, seu nome vem do JavaScript XML. Normalmente, mantemos o código em HTML, CSS e JavaScript em arquivos separados. Porém no React, isso funciona de um modo um pouco diferente.

Nos projetos em React, não criamos arquivos em HTML separados, pois o JSX nos permite escrever uma combinação de HTML e JavaScript em um mesmo arquivo. Você pode, no entanto, separar seu CSS em outro arquivo.

Antes de trabalhar com o JSX, iremos tirar os erros de importações que estão aparecendo em seu projeto. Seu console deve estar com os seguintes alertas de erros:



```

    ERRORS:
    ERROR in ./src/App.js 4:10-18
    Module not found: Error: Can't resolve './login.svg' in 'C:\Users\jose\Documents\React\reactive\src'
    ERROR in ./src/App.js 5:10-18
    Module not found: Error: Can't resolve './App.css' in 'C:\Users\jose\Documents\React\reactive\src'
    ERROR in ./src/index.js 8:9-46
    Module not found: Error: Can't resolve './reportBrewitals' in 'C:\Users\jose\Documents\React\reactive\src'
    webpack compiled with 3 errors
  
```

Isso porque excluímos alguns arquivos porém ainda estamos chamando eles nos arquivos existentes.

- No arquivo index.css você pode excluir todo o código de dentro dele.
- No arquivo index.js iremos tirar a importação do reportWebVitals e o reportWebVitals() em nosso código.
- No arquivo App.js deixe como está, iremos trabalhar nele agora.

Como dito anteriormente, o JSX possui uma sintaxe muito semelhante ao HTML. O código abaixo (no arquivo App.js) demonstra claramente esta característica. Apesar de muito parecido, o código a seguir não é HTML e sim um trecho de código JSX.

```
● ● ●  
1 function App() {  
2   return <h1>Sou um título</h1>;  
3 }  
4  
5 export default App;
```

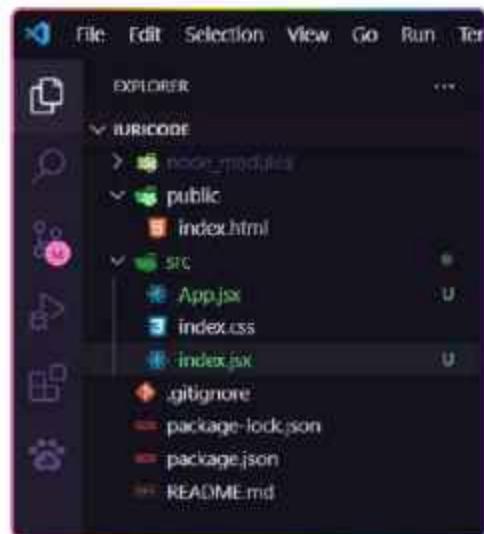
Deixe seu código no arquivo App.js dessa forma. Exclua as importações e os conteúdos de dentro do return.

Para melhor o processo de desenvolvimento podemos mudar (ou criar) os arquivos com o formato .js para .jsx

Alguns editores de código, como Visual Studio Code, podem vir mais preparados.

Por exemplo: se você utilizar .jsx, pode ser que tenha vantagens no autocomplete das tags e também no highlight do código.

Dessa forma sempre é bom mudar de .js para .jsx.



Após mudar o formato dos arquivos de .js para .jsx teremos o seguinte erro:

```
Module not found: Error: Can't resolve 'C:\juricode\src\index.js' in 'C:\juricode'
  in main
Module not found: Error: Can't resolve 'C:\juricode\src\index.js' in 'C:\juricode'

webpack compiled with 1 error.
```

Isso ocorreu por causa da mudança do formato, para resolver isso bastante parar o comando npm start e rodar novamente.

Uma coisa muito importante sobre o JSX é que sempre que utilizamos mais de uma tag HTML no retorno da função, precisamos englobadas em uma tag pai. No exemplo abaixo é uma demonstração incorreta.

```
function App() {
  return (
    <h1>Sou um título</h1>
    <p>Sou uma descrição</p>
  );
}
export default App;
```

Agora iremos corrigir a função englobando a tag H1 e P dentro de uma Div.

```
1 function App() {  
2   return (  
3     <div>  
4       <h1>Sou um título</h1>  
5       <p>Sou uma descrição</p>  
6     </div>  
7   );  
8 }  
9 export default App;
```

Porém nem sempre queremos que o retorno tenha uma tag pai englobando tudo. Para isso temos o Fragment do React, que nada menos é uma tag vazia/sem significado.

```
1 ...  
2 <>  
3   <h1>Sou um título</h1>  
4   <p>Sou uma descrição</p>  
5 </>  
6 ...
```

Você pode fazer dessa forma também (observe que importamos o React):

```
1 import React from "react";
2
3 function App() {
4   return (
5     <React.Fragment>
6       <h1>Sou um título</h1>
7       <p>Sou uma descrição</p>
8     </React.Fragment>
9   );
10 }
11
12 export default App;
```

Observação: para utilizar o Fragment é preciso da importação do React.

Componentes

No exemplo anterior foi mostrado a utilização do JSX, o exemplo nada mais é do que um componente no React.

No exemplo anterior foi mostrado a utilização do JSX, o exemplo nada mais é do que um componente no React.

Os componentes permitem você dividir a UI em partes independentes, reutilizáveis, ou seja, trata cada parte da aplicação como um bloco isolado, livre de outras dependências externas. Componentes são como as funções JavaScript. Eles aceitam entradas e retornam elementos React que descrevem o que deve aparecer na tela.

"E como isso é importante?"

Vou criar um simples exemplo. Imagine que temos um card que representa a apresentação de uma notícia. Sua estrutura poderá se parecida como essa:

```
1 <article>
2   
3   <div>
4     <h2>Título de um post</h2>
5     <p>Sou apenas uma pequena descrição</p>
6   </div>
7   <a href="www.iuricode.com/efront">Acessar post</a>
8 </article>
```

Até aqui está tudo bem. Mas imagine que queremos 6 (seis) cards iguais a esse. Normalmente com HTML iremos duplicar esse código em 6 (seis) vezes, correto?

Agora imagine que por algum motivo seja preciso adicionar uma tag span para sinalizar a data da postagem. Teríamos que mudar em todos! Pior ainda, imagine que existe este card em várias páginas em nossa aplicação. Concordo que iria demorar muito para dar manutenção neste código, por causa de uma simples alteração?

E como o React resolve isso? Com os componentes.

Para criar um componente é bem simples, basta você criar um novo arquivo, exemplo, Card.jsx e criar a estrutura de um componente.

```
● ● ●  
1 function Card() {  
2   return (  
3     Aqui vai o código do exemplo anterior  
4   );  
5 }  
6  
7 export default Card;
```

Após criar o arquivo basta chamar/importar em nosso arquivo App.js

```
1 import Card from './Card';
2
3 function App() {
4   return (
5     <div>
6       <Card />
7       <Card />
8       <Card />
9     </div>
10 );
11 }
12 export default App;
```

O que vale ser comentado aqui é que um componente sempre irá começar com a letra maiúscula, exemplo, Card, Footer, Menu... caso a primeira letra seja minúscula o React irá interpretar como se fosse uma tag HTML. Outra coisa importante é o './Card' é o caminho onde se encontra o arquivo. Nesse caso o arquivo se encontra na mesma raiz do arquivo App.js.

Você pode importar dizendo o formato do arquivo como: './Card.tsx'.

Lembra da pasta public? No exemplo do Card tivemos o seguinte trecho de código:

```
1 
```

Bem, tudo que tiver dentro da pasta public pode ser acessado apenas com o nome e formato do arquivo, diferente do HTML tradicional que teria que escrever o caminho das pasta até acessar o arquivo desejado.

Mas você deve estar se perguntando "Ok Iuri, mas se caso eu queira que cada componente do arquivo Card tenha um título, descrição e/ou imagem diferente das outras?"

A resposta para isso é props!

Props

Outro conceito importante dos componentes é a forma como se comunicam. O React tem um objeto especial, chamado de prop (que significa propriedade), que usamos para transportar dados de um componente para o outro. Isso é, as props são valores personalizados e tornam os componentes mais dinâmicos.

Vamos utilizar o componente Card já mostrado para ver como passar dados com as props.

Primeiro, precisamos definir uma prop no componente Card e atribuir um valor a ela (isso no arquivo App.jsx).



```
 1 import Card from './Card';
 2
 3 function App() {
 4   return (
 5     <div>
 6       <Card title="React" />
 7       <Card title="Front-end" />
 8       <Card title="iuricode" />
 9     </div>
10   );
11 }
12 export default App;
```

Como o componente Card é o elemento filho, precisamos definir as props no seu pai (App.jsx), para assim podermos receber os valores e obter o resultado simplesmente acessando a prop "title".

```
● ● ●  
1 function Card(props) {  
2   return (  
3     <article>  
4         
5       <div>  
6         <h2>{props.title}</h2>  
7         <p>Sou apenas uma pequena descrição</p>  
8       </div>  
9       <a href="www.iuricode.com">Acessar post</a>  
10      </article>  
11    );  
12  }  
13 export default Card;
```

Dessa forma conseguimos utilizar o código do componente Card e apresentar dados de forma dinâmica.

CSS Modules

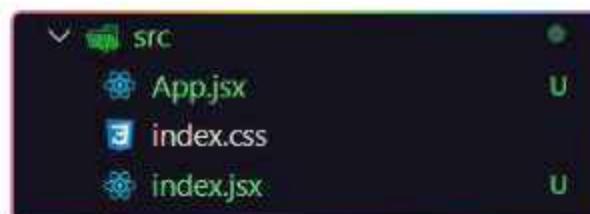
Quando criamos nossa aplicação com CRA, foram criados dois arquivos CSS (index.css e App.css) que são responsáveis por adicionar estilos na aplicação toda. Porém existe um problema com os classNames ao adicionar estilos dessa forma que foi feito pelo CRA.

O problema com os classNames ocorre da seguinte maneira. Imaginem um nome de className bastante utilizado por todos, no caso irei explicar utilizando o nome "title". Todos os lugares em que utilizamos o "title" precisamos criar um nome composto para que não haja globalidade entre os estilos, então caso formos utilizar dentro de um card, criaremos o "card-title", caso seja uma modal, será "modal-title", e cada vez ficará mais difícil para pensar num bom nome de className para cada componente que precise de um "title".

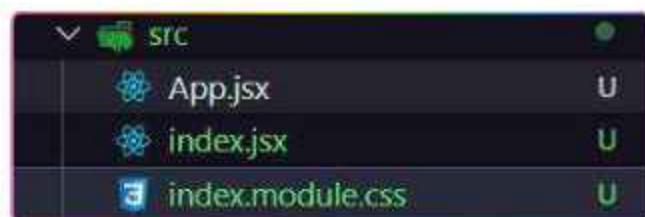
Porém temos a opção de criar estilos únicos para cada componente, utilizando os css-modules. Os css-modules são arquivos CSS em que os classNames são definidos localmente, isso significa que os estilos ali criados, só serão declarados dentro daquele escopo, e não globalmente, evitando conflitos entre estilos.

O primeiro passo é alterar o nome do seu arquivo de estilos para o seguinte formato: [name].module.css

Antes:



Depois:



Depois de fazer isso, deverá alterar seu import dos estilos e a utilização dos classNames para a seguinte forma:

```

1 import styles from "./index.module.css";
2
3 function App() {
4   return <h1 className={styles.title}>Hello</h1>;
5 }
6 export default App;

```

Perceba que dentro de className chamamos como {styles.title} diferente do CSS normal que seria "title". A palavra "styles" é opcional, você escolher outro nome no lugar, mas é o mais utilizado.

Observação: lembre-se de tirar o import do index.css dos outros arquivos, se não nossa aplicação não irá rodar.

Você deve estar se perguntando "como que isso funciona por baixo dos panos?". O css-modules cria um className único para cada local em que é utilizado.

Resultado final utilizando o CSS Modules:



Eventos

Manipular eventos com elementos React é muito semelhante ao tratamento de eventos em elementos DOM. Existem algumas diferenças sintáticas para as quais devemos atentar:

- Os eventos React são nomeados usando camelCase, em vez de minúsculas;
- Com o JSX, você passa uma função como manipulador de eventos, em vez de uma string;

Por exemplo, o evento HTML definimos assim:

```
1 <button onclick="ativarDesconto()">
2   Ativar Desconto
3 </button>
```

Será definido da seguinte forma no React:

```
1 <button onclick={ativarDesconto}>
2   Ativar Desconto
3 </button>
```

Vamos criar um novo componente bem simples para mostrar como escutar eventos no React.

No App.jsx defina o seu código a seguir:

```
● ● ●  
1 function App() {  
2   const exibirAviso = () => {  
3     alert("Cuidado!");  
5   };  
6  
7   return (  
8     <button onClick={exibirAviso}>Exibir aviso</button>  
9   );  
10 }  
11 export default App;
```

No exemplo acima, o atributo onClick é nosso manipulador de eventos e é adicionado ao elemento de destino para especificar a função a ser executada quando esse elemento for clicado.

O atributo onClick é definido para a função exibirAviso que exibe uma mensagem.

Assim, sempre que o botão for clicado, a função exibirAviso será chamada que, por sua vez, mostra a caixa de aviso.

Essa é uma forma bem simples de tratar eventos no React.

Hooks

Hooks foram adicionados ao React na versão 16.8.

Hooks permitem que componentes de função tenham acesso ao estado e outros recursos do React. Por causa disso, os componentes de classe geralmente não são mais necessários.

Embora Hooks geralmente substituem componentes de classe, não há planos para remover classes do React.

"Mas o que são Hooks?"

Basicamente, Hooks nos permitem "ligar" em recursos do React, como métodos de estado e ciclo de vida.

Aqui está um exemplo de um Hook. Não se preocupe se não fizer sentido. Já entrarei em mais detalhes.

usaState

```
1 import React, { useState } from "react";
2
3 function App() {
4   // Declaramos a nova variável de estado,
5   // que chamaremos de "count"
6   const [count, setCount] = useState(0);
7
8
9   return (
10    <div>
11      <p>Você clicou {count} vezes</p>
12      <button onClick={() => setCount(count + 1)}>
13          Click
14      </button>
15    </div>
16  );
17}
18
19 export default App;
```

Para usar o Hook useState (ou qualquer Hook), primeiro precisamos importá-lo em nosso componente.

```
1 import React, { useState } from "react";
```

Observe que estamos desestruturando o useState a partir do React pois é uma exportação nomeada.

"Mas o que o useState está fazendo exatamente?"

Aqui estamos usando o Hook useState para acompanhar o estado da aplicação.

useState aceita um estado inicial e retorna dois valores:

- O estado atual.
- Uma função que atualiza o estado.

O estado geralmente se refere aos dados ou propriedades da aplicação que precisam ser rastreados.

Depois de importar o useState do React. Inicialize o estado na parte superior do componente da função.

```
1 import React, { useState } from "react";
2
3 function App() {
4   const [color, setColor] = useState("vermelho");
5 }
6 export default App;
```

Observe que, novamente, estamos desestruturando os valores retornados de useState.

- O primeiro valor, color, é o nosso estado atual.
- O segundo valor, setColor, é a função que é usada para atualizar nosso estado.

Por fim, definimos o estado inicial como "vermelho".

Agora podemos incluir nosso estado em qualquer lugar em nosso componente.

"E como irei atualizar o estado atual?"

Para atualizar nosso estado, usamos nossa função de atualização de estado.

Nunca devemos atualizar o estado diretamente.

Ex: color = "vermelho" não é permitido.

Sempre iremos utilizar o segundo valor para adicionar o novo valor, e o primeiro valor para apresentar o valor atual. Exemplo:

Correto:

```
1 import React, { useState } from "react";
2
3 function App() {
4   const [color, setColor] = useState("vermelho");
5
6   return (
7     <div>
8       <h1>Minha cor favorita é {color}!</h1>
9       <button type="button"
10         onClick={() => setColor("azul")}> Mudar para azul
11       </button>
12     </div>
13   );
14 }
15 export default App;
```

Errado (erro está no button):

```
1 import React, { useState } from "react";
2
3 function App() {
4   const [color, setColor] = useState("vermelho");
5
6   return (
7     <div>
8       <h1>Minha cor favorita é {color}!</h1>
9       <button type="button"
10         onClick={() => color("azul")}> Mudar para azul
11       </button>
12     </div>
13   );
14 }
15 export default App;
```

O Hook useState pode ser usado para rastrear strings, números, booleanos, arrays, objetos e qualquer combinação destes.

Poderíamos criar vários Hooks de estado para rastrear valores individuais.

Errado (erro está no button):

```
1 import React, { useState } from "react";
2
3 function App() {
4   const [brand, setBrand] = useState("Ford");
5   const [model, setModel] = useState("Mustang");
6   const [year, setYear] = useState("1964");
7   const [color, setColor] = useState("red");
8 }
9 export default App;
```

Ou podemos usar apenas um estado e incluir um objeto.

```
1 const [car, setCar] = useState({
2   brand: "Ford",
3   model: "Mustang",
4   year: "1964",
5   color: "red",
6 });
```

useEffect

O useEffect é um Hook do React. Ele permite que você execute efeitos colaterais no seu código. Mas o que seriam esses efeitos colaterais? Buscar dados de uma API, mudar a DOM e cronômetros, por exemplo, são algumas opções de efeitos colaterais no seu código.

Na sua declaração, ele vai receber dois parâmetros, que funcionam assim:

```
1 useEffect(<Função>, <Dependência>)
```

Por fim, no trecho de código abaixo você vai ver o useEffect usado para mudar o título de uma página usando como base o estado da variável 'count'. Note que, como dependência, o useEffect vai utilizar o 'count' justamente porque o 'document.title' precisa ser executado novamente toda vez que o 'count' mudar de estado. Portanto, se não houvesse dependência, a função seria executada somente uma vez ao renderizar o componente.

Errado (erro está no button):

```
1 import React, { useState, useEffect } from "react";
2
3 function App() {
4   const [count, setCount] = useState(0);
5
6   useEffect(() => {
7     document.title = `Você clicou ${count} vezes`;
8   });
9
10  return (
11    <div>
12      <p>Você clicou {count} vezes</p>
13      <button
14        onClick={() => setCount(count + 1)}> Clique aqui
15      </button>
16    </div>
17  );
18 }
19 export default App;
```

Claro que esses não são os únicos Hooks disponíveis no React. Abaixo deixa a lista dos principais Hook: useState, useEffect, useContext, useRef, useReducer, useCallback e useMemo.

API

Agora que entendemos como funciona os Hooks useState e useEffect já podemos trazer um exemplo prático da sua utilização. Em nosso exemplo com os Hooks, teremos um componente que será responsável por listar os nossos repositórios do GitHub.

Iremos construir uma aplicação que lista os repositórios da nossa conta do GitHub, então precisarei acessar a API deles. Faremos uma requisição HTTP utilizando o fetch do JavaScript para buscar a lista de repositórios desta API. Vale ressaltar que o fetch gera efeitos colaterais em nosso código, pois ele é uma operação de input e output.

O Hook useEffect nos auxilia a lidar com os efeitos colaterais.

```
● ● ●  
1 import React, { useEffect } from "react";  
2  
3 function App() {  
4   useEffect(() => {  
5     async function carregaRepositorios() {  
6       const resposta = await fetch(  
7         "https://api.github.com/users/iuricode/repos"  
8     );  
9     ...  
10    }  
11  })  
12}  
13  
14 export default App;
```

```
10 ...
11   const repositorios = await resposta.json();
12   return repositorios;
13 }
14
15 carregaRepositorios();
16 }, []);
17 return <h1>Retorno da função</h1>;
18 }
19 export default App;
```

No exemplo acima, os efeitos colaterais são a chamada API. Este Hook recebe dois parâmetros: o primeiro é uma função que será executada quando o componente for inicializado e atualizado. Em nosso exemplo, este primeiro parâmetro é uma arrow function assíncrona, que faz uma requisição à API e guarda na const response, em formato de json, e, depois, na const repositorios. Já o segundo parâmetro indica em qual situação os efeitos colaterais irão modificar. No nosso caso, como queremos carregar a lista somente uma vez, passamos um array vazio [], pois ele garante a execução única.

Com os dados da API em mãos, agora só armazená-los em uma lista e depois exibi-los em tela. Para que possamos lidar com as mudanças de estado da nossa aplicação, iremos usar o hook useState. Para isso, usamos o hook desta forma:

```
1 import React, { useState, useEffect } from "react";
2
3 function App() {
4   const [repositorio, setRepositorio] = useState([]);
5
6   useEffect(() => {
7     async function carregaRepositorios() {
8       const resposta = await fetch(
9         "https://api.github.com/users/iuricode/repos"
10      );
11
12       const repositorios = await resposta.json();
13       return repositorios;
14     }
15     carregaRepositorios();
16   }, []);
17
18   return <h1>Retorno da função</h1>;
19 }
20 export default App;
```

Como a nossa função setRepository é a responsável por alterar o estado de repository, precisamos colocá-la dentro do escopo da função useEffect, porque ela é a responsável por pegar os dados que vão modificar o estado da nossa aplicação. Desta forma, fica assim:

```
● ● ●
1 import React, { useState, useEffect } from "react";
2
3 function App() {
4   const [repository, setRepository] = useState([]);
5
6   useEffect(() => {
7     async function carregaRepositories() {
8       const resposta = await fetch(
9         "https://api.github.com/users/iuricode/repos"
10      );
11
12       const repositories = await resposta.json();
13       setRepository(repositories);
14     }
15     carregaRepositories();
16   }, []);
17
18   return <h1>Retorno da função</h1>;
19 }
20 export default App;
```

A diferença dentro dessa função é que retiramos o return repositorios e adicionamos a função setRepository(), passando como parâmetro a constante repositorys, pois, como foi explicado, é essa função responsável por atualizar os estado da nossa aplicação.

Agora que já temos os dados dos repositórios vindo da API do GitHub e estamos atualizando o estado da nossa aplicação com eles, o último passo é exibir ele de forma dinâmica dentro da tag ul desta forma.

```
1 import React, { useState, useEffect } from "react";
2
3 function App() {
4   const [repository, setRepository] = useState([]);
5
6   useEffect(() => {
7     async function carregaRepositorys() {
8       const resposta = await fetch(
9         "https://api.github.com/users/iuricode/repos"
10     );
11
12     const repositorys = await resposta.json();
13     setRepository(repositorys);
14   }
15 }
```

```
1 ...
2     carregaRepositórios();
3 }, []);
4
5 return (
6     <ul>
7         {repositorio.map((repositorio) => (
8             <li key={repositorio.id}>{repositorio.name}</li>
9         ))}
10    </ul>
11 );
12 }
13 export default App;
```

Para que fosse possível gerar dinamicamente, utilizamos o método map() para poder percorrer o nosso array repositorio, que possui a lista de repositórios, e imprimir um a um na tela.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fazendo deploy e implementando versões móveis de sites, é comum que surjam muitas dúvidas para quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, treinamento, você também realizará sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **medo** de problemas para entrar na área de programação, este é o seu guia para superá-los. Com dicas em suas etapas, exercícios práticos, exercícios teóricos e muitos **ONLINE**, profissionais qualificados para orientar a aula que os resultados são:

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 7

React Router

Reactive咖啡提供一個新的模塊來管理應用的路徑。讓你可以在不同的頁面之間跳轉，而不需要重新載入整個頁面。這是一個非常有用的技術，可以讓你的應用更流暢和更易於管理。

* 6 TÓPICOS NESTE MÓDULO

Introdução

Em um site acessamos várias páginas, como por exemplo, página inicial, contatos e sobre. Para navegar entre as páginas de uma aplicação React precisaremos criar rotas, onde cada rota vai representar uma página. Para trabalhar com rotas no React, vamos utilizar a biblioteca React Router (em sua versão 6), é ele que nos auxiliará na criação da navegação.

O que é

O React Router é a biblioteca de roteamento mais popular do React. Quando trabalhamos com aplicativos de página única (SPAs) com várias exibições precisam ter um mecanismo de roteamento para navegar entre essas diferentes exibições sem atualizar toda a página. Isso pode ser tratado usando uma biblioteca de roteamento como o React Router.

Iniciando

Iremos criar uma aplicação do zero e depois adicionar o React Router.



```
npx create-react-app iuricode
```

```
npm i react-router-dom
```

A primeira coisa que faremos é englobar todas as diferentes rotas que teremos em nossa aplicação. Para isso iremos utilizar o BrowserRouter que armazena a localização atual na barra de endereço do navegador usando URLs limpas e navega usando a pilha de histórico integrada do navegador.

```
1 import React from "react";
2 import ReactDOM from "react-dom/client";
3 import "./index.css";
4 import App from "./App";
5 import { BrowserRouter } from "react-router-dom";
6
7 const root = ReactDOM.createRoot(
8   document.getElementById("root")
9 );
10 ...
```

```
7 root.render(  
8 <React.StrictMode>  
9   <BrowserRouter>  
10    <App />  
11  </BrowserRouter>  
12 </React.StrictMode>  
13 );
```

Agora toda a nossa aplicação está englobada no BrowserRouter. Dessa forma já podemos utilizar métodos do React Router em qualquer lugar da nossa aplicação.

Criando as rotas

Após englobar o BrowserRouter em nossa aplicação, a próxima etapa é definir suas rotas. Isso geralmente é feito no nível superior da sua aplicação, como no componente App, mas podemos criar em um arquivo separado. É o que faremos agora.

Iremos criar um arquivo novo como routes.jsx. Nele iremos colocar todas as nossas rotas que queremos. Este arquivo deverá se importado no index.jsx assim como o App.jsx. Lembrando que os dois devem estar dentro do BrowserRouter:

```
1 ...
2 import { BrowserRouter } from "react-router-dom";
3 import AppRoutes from "./routes";
4 import App from "./App";
5 ...
6 ...
7 <BrowserRouter>
8   <AppRoutes />
9   <App />
10 </BrowserRouter>
11 ...
```

Antes de configurar as rotas no arquivo routes.jsx iremos criar os arquivos que serão acessados na rota. Iremos criar uma pasta chamada "pages" e dentro dela iremos criar a página de Login, Profile e a Home.

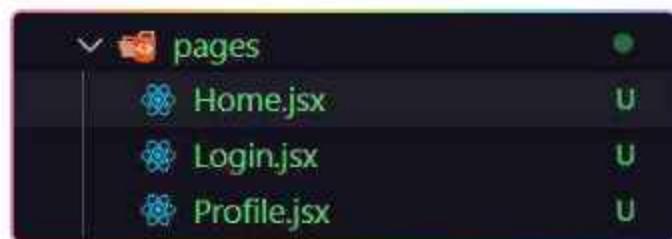
Todos eles terão a mesma estrutura, um componente React com um h1.

```

1 import React from "react";
2
3 const Home = () => {
4   return <h1>Home</h1>;
5 }
6
7 export default Home;

```

Iremos fazer o mesmo nos seguintes arquivos:



Agora já podemos configurar as nossas rotas.

Na pasta src iremos criar o arquivo routes.jsx iremos importar os componentes routes e route do react-router-dom.

```

1 import { Routes, Route } from "react-router-dom";

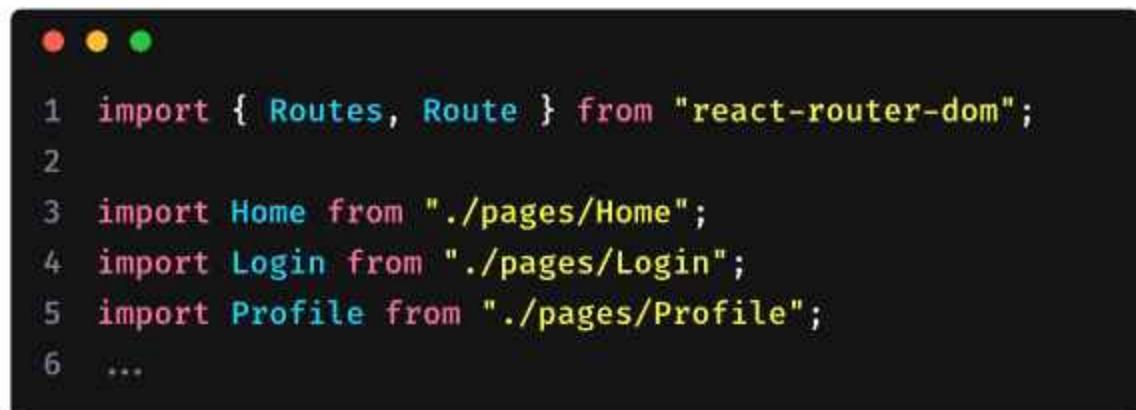
```

O componente Routes conterá todas as diferentes rotas possíveis que podem ser exibidas naquele segmento específico de uma página e pode ser visto como uma matriz de rotas.

Para definir rotas individuais que podem ser exibidas no componente Routes, usamos outro componente chamado Route.

O componente Route aceita duas props.

- path: este é o URL para o qual o link nos levará. Este suporte é semelhante ao suporte do componente Link que iremos falar no próximo tópico.
- element: contém o elemento que será renderizado quando a página navegar para essa rota.

A screenshot of a code editor window. At the top left, there are three colored circular icons: red, yellow, and green. The main area contains the following code:

```
1 import { Routes, Route } from "react-router-dom";
2
3 import Home from "./pages/Home";
4 import Login from "./pages/Login";
5 import Profile from "./pages/Profile";
6 ...
```

The code uses syntax highlighting where 'import' and 'Route' are in blue, 'Routes' is in green, and the component names 'Home', 'Login', and 'Profile' are in pink. The file extension '.js' is implied at the end of each import statement.

```
7 ...
8 const AppRoutes = () => {
9   return (
10   <Routes>
11     <Route path="/" element={<Home />} />
12     <Route path="login" element={<Login />} />
13     <Route path="profile" element={<Profile />} />
14   </Routes>
15 );
16 };
17
18 export default AppRoutes;
```

Como dito, perceba que dentro o “path” passamos o valor que irá enviado para a URL, então quando acessar o “login” será renderizado o componente de dentro do “element” dele, que no caso é o <Login />

Faça um teste, acesse: <http://localhost:3000/login> (No App.jsx podemos adicionar só um título (tag h1) que irá aparecer em todas as rotas da nossa aplicação).

Porém até o momento só estamos acessando as rotas passando um valor pela URL, para a experiência do usuário isso não é legal. Iremos agora criar um menu que será responsável pela navegação.

Navegação de links

Vamos criar alguns links de navegação que vão apontar para algumas rotas dentro do nosso routes.jsx. Essas rotas foram passadas para o index.jsx porque queremos que fiquem visíveis em todas as páginas. Para fazer isso, devemos importar o componente Link. Este é um componente de link (semelhante à tag <a>) que nos ajuda a navegar entre as rotas sem passar o valor direto na URL.

Para isso, iremos adicionar no App.jsx, utilizando o componente Link do React Router. Pois ao tentar navegar utilizando a tag A, a nossa página será toda recarregada novamente.

```
● ● ●
1 import React from "react";
2 import { Link } from "react-router-dom";
3
4 const App = () => {
5   return (
6     <nav>
7       <ul>
8         <li>
9           <Link to="/">Home</Link>
10        </li>
11      ...
12    )
13  }
14
15  export default App;
```

```
12 ...
13     <li>
14         <Link to="/login/efront">Login</Link>
15     </li>
16     <li>
17         <Link to="/profile">Profile</Link>
18     </li>
19     </ul>
20 </nav>
21 );
22 };
23
24 export default App;
```

Rota 404

A página de erro foi criada para que as rotas que não correspondam a nenhuma das rotas definidas acima sejam atribuídas à página de erro (essencialmente, esta será nossa página de erro 404 personalizada).

Primeira coisa que iremos fazer é criar uma rota dentro a pasta "pages", irei chamar ele de "Error.jsx".

```
1 import React from "react";
2
3 const Error = () => {
4   return <h1>Ops, página não encontrada!</h1>;
5 }
6
7 export default Error;
```

A única coisa que precisamos fazer é passar um * (asterisco) dentro do path no componente do Route.

```
1 ...
2 <Route path="*" element={<Error />} />
3 ...
```

Agora qualquer valor que for acessado na URL que não estiver configurado dentro do nosso arquivo routes.jsx será encaminhado para a página de erro 404 personalizada por nós.

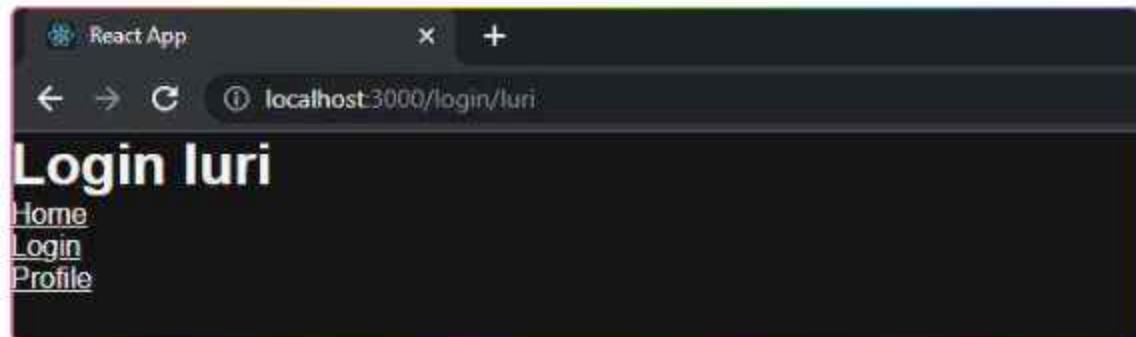
Rotas dinâmicas

Importaremos o useParams para a página Login para que possamos usá-lo para obter o parâmetro de URL e, em seguida, usar esse parâmetro para exibir ele na tela. Além disso iremos desestruturar para que possamos acessar diretamente o parâmetro slug e utilizá-lo dentro do nosso Login. Dessa forma:

```
1 import React from "react";
2 import { useParams } from "react-router-dom";
3
4 const Login = () => {
5   const { slug } = useParams();
6
7   return <h1>Login {slug}</h1>;
8 }
9
10 export default Login;
```

Agora basta adicionar "/:slug" no path do Login.

```
1 <Route path="login/:slug" element={<Login />} />
```



"Quando utilizamos uma rota dinâmica?"

Vamos supor que queremos renderizar um componente para livros individuais em nossa aplicação. Poderíamos desenvolver uma rota para cada livro, mas se tivermos centenas de livros, será impossível desenvolver todas essas rotas. Em vez disso, apenas precisamos de uma rota dinâmica que irá gerar essas rotas individuais.

Neste módulo entendemos a usabilidade do React Router, criando rotas, criamos rotas dinâmicas, navegação com links. A Biblioteca utilizada, além de auxiliar na construção dessa navegação de uma maneira mais objetiva, também é customizável, abrindo um leque de opções de uso que se encaixam na maneira como você vai desenvolver o seu projeto. Espero que tenha sido útil.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fazendo deploy e implementando versões móveis de sites, é comum que surjam muitas dúvidas para quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, treinamento você também realizará sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **problemas para entrar na área de programação**, este é o seu guia para começar a programar em suas redes sociais, empresas, serviços bancários e muito mais! O HTML, CSS, JavaScript, Python, Java, MySQL e muito mais!

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 8

TypeScript

Preparamos o café e preparamos a máquina para começar o módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes com os conhecimentos adquiridos através de um questionário no Módulo.

+ 7 TÓPICOS NESTE MÓDULO

Introdução

Como já dito, a tipagem dinâmica nos permite atribuir diferentes tipos de valores para uma variável, hora pode receber uma string, hora um number e isso pode prejudicar a nossa aplicação a medida que ela vai crescendo. Exemplo: imagina que você tem uma variável chamada "nome" que no decorrer da aplicação ela recebe um número, isso é aceitável no JavaScript, mas não é nossa intenção ao criar variável "nome".

```
● ● ●  
1 let nome = "Iuri";  
2 nome = 12;
```

Porém, isso já não acontece em linguagens que possuem tipagem estática, pois, nessa linguagens obrigatoriamente precisamos declarar o tipo do dado que a variável irá receber.

```
● ● ●  
1 let nome:string = "Iuri";  
2 nome = 12; // Teremos um erro no tipo
```

O que é

Pensando nesse problema a Microsoft desenvolveu o TypeScript para tiparmos os dados, porém de uma forma que mantém toda a sintaxe e funcionamento do JavaScript que já conhecemos. Mas acaba entrando uma questão: "Mas Iuri, o navegador não entende só JavaScript?". Exato! No final de todo o processo, o código que desenvolvemos utilizando TypeScript irá virar JavaScript comum, isso funciona da mesma maneira do Sass.

O TypeScript detecta quando passamos um valor diferente para uma variável declarada durante o desenvolvimento e avisa na IDE (no seu editor de código). Isso reflete num ambiente muito mais seguro enquanto o código está sendo digitado. Outra coisa interessante do TypeScript é a possibilidade de transpilar o seu código para uma versão antiga do JavaScript, isso faz com que seu código seja lido por todas as versões dos navegadores.

Iniciando

Para utilizarmos o TypeScript precisamos do Node.js instalado. Depois do Node, começaremos instalando o TypeScript globalmente em nossa máquina. Para isso daremos o seguinte comando:



```
npm install -g typescript
```

Agora com o TypeScript instalado, podemos utilizar o comando "tsc" no nosso terminal. Onde usaremos o tsc para executar o nosso arquivo chamado "app.ts". Obs: para um arquivo rode código TypeScript, ele precisa ter a extensão .ts



```
tsc app.ts
```

Perceba que agora um arquivo app.js foi criado na pasta raiz do nosso projeto.

Isso aconteceu porque o tsc transformou toda linguagem TypeScript em JavaScript.

Caso tivéssemos algum recurso que só existe no TypeScript, ele também seria traduzido de forma que o node pudesse entender.

Tipos

Tipos são valores que informamos como um complemento de uma variável, a fim de evitar que seja atribuído um valor diferente de variável.

String

String é um tipo de dados usado para armazenar dados de texto. Os valores de string são colocados entre aspas simples ou aspas duplas.

```
1 let nome:string = "Iuri";
2 // Ou
3 let nome:string = 'Iuri';
```

Template string

Template Strings são usados para incorporar expressões em strings.

Aqui, em vez de escrever uma string que é uma combinação de texto e variáveis com concatenações, podemos usar uma única instrução com back-ticks/crase(`). Os valores das variáveis são escritos como \${}.

```
1 let nome:string = "Iuri";
2 let frase:string = `${nome} é dev`;
```

Boolean

Os valores booleanos são utilizados para armazenar valores true ou false.

```
1 let dev:boolean = true;
```

Number

Todos os números são armazenados como números de ponto flutuante. Esses números podem ser decimais (base 10), hexadecimal (base 16) ou octal (base 8).

```
1 let idade:number = 22;
```

Array

Um array pode armazenar vários valores de diferentes tipos de dados sequencialmente usando uma sintaxe especial. Existem duas maneiras de declarar um array, ambos os métodos produzem a mesma saída.

1. Usando colchetes. Este método é semelhante a como você declararia array em JavaScript.



```
1 let nomes:string[] = ['Iuri', 'Eiji'];
```

2. Usando um tipo de array genérico, Array <elementType>.



```
1 let nomes: Array<string> = [
2   'Iuri', 'Eiji'
3 ]
```

Um array pode conter elementos de diferentes tipos de dados, para isso no TypeScript usamos uma sintaxe de tipo de array genérico, conforme é mostrado a seguir.

```
● ● ●  
1 let nomes: (string | number)[] = [  
2   'Iuri', 22, 'Eiji', 21  
3 ]  
4  
5 // Ou  
6  
7 let nomes: Array<string | number> = [  
8   'Iuri', 22, 'Eiji', 21  
9 ]
```

Tuple

O TypeScript introduziu um novo tipo de dados chamado Tuple. O tuple pode conter dois valores de diferentes tipos de dados, assim, eliminando a necessidade de declarar duas variáveis diferentes.

```
● ● ●  
1 let pessoa: [number, string] = [  
2   2000, 'Iuri'  
3 ]
```

Enum

Em palavras simples, enums nos permite declarar um conjunto de constantes nomeadas, ou seja, uma coleção de valores relacionados que podem ser valores numéricos ou strings. Enums são definidos usando a palavra-chave enum.

Existem três tipos de enums:

- Enum numérico
- Enum string
- Enum heterogêneo

Enum numérico

Enums numéricos são enums baseados em números, ou seja, armazenam valores de string como números.

No exemplo abaixo, temos um enum chamado Direcao. O enum tem quatro valores: Norte, Leste, Sul e Oeste.

```
1 enum Direcao {  
2     Norte,  
3     Leste,  
4     Sul,  
5     Oeste,  
6 }
```

Os Enums são sempre atribuídos a valores numéricos quando são armazenados. O primeiro valor sempre assume o valor numérico de 0, enquanto os outros valores no enum são incrementados em 1.

```
1 enum Direcao {  
2     Norte, // 0  
3     Leste, // 1  
4     Sul, // 2  
5     Oeste, // 3  
6 }
```

Enum string

Enums de string são semelhantes aos enums numéricos, exceto que os valores de enum são inicializados com valores de string em vez de valores numéricos.

Os benefícios de usar enums de string é que enums de string oferecem melhor legibilidade. Se tivéssemos que depurar um programa, seria mais fácil ler valores de string do que valores numéricos.

```
1 enum Direcao {  
2     Norte = "Norte",  
3     Leste = "Leste",  
4     Sul = "Sul",  
5     Oeste = "Oeste",  
6 }
```

A diferença entre enums numéricos e de string é que os valores de enum numéricos são incrementados automaticamente, enquanto os valores de enum de string precisam ser inicializados individualmente.

Enum heterogêneo

Enums heterogêneos são enums que contêm valores de string e numéricos.

```
1 enum Direcao {  
2     Norte = "OK",  
3     Leste = 1,  
4     Sul,  
5     Oeste,  
6 }
```

Any

Nem sempre temos conhecimento prévio sobre o tipo de algumas variáveis, especialmente quando existem valores inseridos pelo usuário em bibliotecas de terceiros. Nesses casos, precisamos de algo que possa lidar com conteúdo dinâmico. O tipo Any é útil aqui.

```
1 let nome:any = "Iuri";  
2 nome = 22; // Não teremos erro
```

Da mesma forma, você pode criar um array do tipo any [] se não tiver certeza sobre os tipos de valores que podem conter essa array.

A utilização do tipo any não é recomendado. Pois assim irá perder a mágica do TypeScript.

Void

O void é usado onde não há dados. Por exemplo, se uma função não retornar nenhum valor, você pode especificar void como tipo de retorno.

```
1 function mensagem():void {  
2   console.log("iuricode");  
3 }
```

Never

O tipo never é usado quando você tem certeza de que algo nunca acontecerá. Por exemplo, você escreve uma função que não retorna ao seu ponto final.

```
1 function mensagem():void {  
2   while(true) {  
3     console.log("iuricode");  
4   }  
5 }
```

No exemplo acima, a função `repeticao()` está sempre em execução e nunca atinge um ponto final porque o loop while nunca termina. Assim, o tipo `never` é usado para indicar o valor que nunca ocorrerá ou retornará de uma função.

Union

No exemplo abaixo, a variável "dev" é do tipo union. Portanto, você pode passar um valor de string ou um valor numérico. Se você passar qualquer outro tipo de valor, por exemplo, booleano, o compilador dará um erro.

```
● ● ●
1 let dev: (string | number);
2
3 dev = 123;
4 dev = 'iuri';
5 dev = true; // Teremos um erro
```

Interface

No TypeScript, temos as interfaces que são um conjunto de métodos e propriedades que descrevem um objeto. As interfaces desempenham a função de nomear esses tipos e são uma maneira poderosa de definir "contrato".

Isso significa que uma estrutura que implementa uma interface é obrigada a implementar todos os seus dados tipados.

Primeiro vamos criar um exemplo sem o uso de uma interface. E entender a necessidade do uso de uma interface:

```
1 function somar(x, y) {  
2   return x + y;  
3 }
```

Neste exemplo não garantimos qual é o tipo que a propriedade x e y irá receber (isso é, uma string, number, boolean..). Já com a interface, garantimos qual será o tipo da nossa propriedade. Com ela, criamos um "contrato", assim sabemos exatamente o que deve ser passado como parâmetro para as propriedades, e quem recebe o parâmetro também saberá.

Definindo uma Interface

Iremos declarar uma interface com a palavra-chave Pessoa. As propriedades e métodos que compõem a interface são adicionados dentro das chaves como key:value pares.

```
● ● ●  
1 interface Pessoa {  
2   nome: string;  
3   idade: number;  
4 }
```

Repare que as propriedades da interface devem ser separadas por ";".

Criando um objeto baseado

Depois de criado a interface Pessoa, podemos criar um objeto onde iremos passar o valor para as propriedades da interface.

```
● ● ●  
1 interface Pessoa {  
2   nome: string;  
3   idade: number;  
4 }  
5  
6 let dev: Pessoa = {  
7   nome: "Iuri",  
8   idade: 22,  
9 }
```

Propriedades opcionais

Nem sempre temos certeza que um dado será exibido ou lido em nossa aplicação, para isso temos a propriedade opcional do TypeScript, que são marcadas com um "?". Nesses casos, os objetos da interface podem ou não definir essas propriedades.

```
● ● ●

1 interface Pessoa {
2   nome: string;
3   sobrenome?: string;
4 }
5
6 let dev1: Pessoa = {
7   nome: "Iuri",
8   sobrenome: "Silva",
9 } // OK
10
11 let dev2: Pessoa = {
12   nome: "Eiji",
13 } // OK
```

No exemplo acima, a propriedade departamento está marcado com ?, então os objetos de Empregado podem ou não incluir esta propriedade.

Estendendo Interfaces

As interfaces podem estender uma ou mais interfaces, isso é, uma interface pode ter as propriedades declaradas em uma outra interface. Isso torna as interfaces e suas propriedades mais flexíveis e reutilizáveis.

```
1  interface Pessoa {
2    nome: string;
3    idade: number;
4  }
5
6  interface Pessoa2 extends Pessoa {
7    front: boolean;
8  }
9
10 let dev2: Pessoa2 = {
11   nome: "Eiji",
12   idade: 21,
13   front: true
14 }
```

No exemplo, a interface Pessoa2 estende a interface Pessoa. Portanto, os objetos de Pessoa2 devem incluir todas as propriedades e métodos da Pessoa interface.

TypeScript no React

Agora que aprendemos o tipos que temos no TypeScript e como utilizar ele no JavaScript nativo, iremos aplicar esses conhecimentos em uma aplicação feita com React.

Vamos utilizar nossa aplicação que utilizamos a API do GitHub:

```
 1 import React, { useState, useEffect } from "react";
 2
 3 function App() {
 4   const [repositorio, setRepositorio] = useState([]);
 5
 6   useEffect(() => {
 7     async function carregaRepositorios() {
 8       const resposta = await fetch(
 9         "https://api.github.com/users/iuricode/repos"
10       );
11
12       const repositorios = await resposta.json();
13       setRepositorio(repositorios);
14     }
15   ...
}
```

```
16
17  carregaRepositórios();
18 }, []);
19
20 return (
21   <ul>
22     {repositorio.map((repositorio) => (
23       <li key={repositorio.id}>{repositorio.name}</li>
24     )));
25   </ul>
26 );
27 }
28 export default App;
```

Instalando

A primeira coisa que iremos fazer é instalar o TypeScript, para isso daremos o seguinte comando dentro da nossa aplicação:

```
npm install typescript @types/node @types/react
@types/react-dom @types/jest
```

Para pular essas etapas de configuração você pode começar um projeto com TypeScript, isso é diferente do que adicionar em um já existente, pois tudo já vem configurado.

```
npx create-react-app my-app --template typescript
```

Após instalar, iremos criar um arquivo chamado tsconfig.json ele será responsável por configurar a forma que queremos que o TypeScript trabalha em nossa aplicação.

```
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "lib": [
5        "dom",
6        "dom.iterable",
7        "esnext"
8      ],
9      ...
}
```

```
 10 "allowJs": true,
 11 "skipLibCheck": true,
 12 "esModuleInterop": true,
 13 "allowSyntheticDefaultImports": true,
 14 "strict": true,
 15 "forceConsistentCasingInFileNames": true,
 16 "noFallthroughCasesInSwitch": true,
 17 "module": "esnext",
 18 "moduleResolution": "node",
 19 "resolveJsonModule": true,
 20 "isolatedModules": true,
 21 "noEmit": true,
 22 "jsx": "react-jsx"
 23 },
 24 "include": [
 25   "src"
 26 ]
 27 ]
 28 }
```

Você pode encontrar outros tipos de configuração com mais ou menos códigos no arquivo tsconfig.json. Mas essa é a configuração criada pelo próprio React.

TSX

Agora para que o TypeScript entendia que nossos arquivo contém código em TypeScript, precisamos mudar o formato deles de .jsx para .tsx.

Após fazer isso teremos alguns erros em nossa aplicação:

The screenshot shows a code editor with two files open:

- App.tsx:**

```

    import React from 'react';
    import { AgregadorRepositorio } from './repositorio';

    const App: React.FC = () => {
      const repositorio = new AgregadorRepositorio();
      const listaDeRepositorios = repositorio.lista();
      return (
        <ul>
          {listaDeRepositorios.map((repositorio) => (
            <li key={repositorio.id}>{repositorio.name}</li>
          ))}
        </ul>
      );
    }

    export default App;
  
```
- index.tsx:**

```

    import React from 'react';
    import App from './App';

    const rootElement = document.createElement('div');
    ReactDOM.render(, rootElement);
  
```

This file contains a TypeScript error:

```

    TS2339: Argument of type 'HTMLDivElement' is not assignable to parameter of type 'Element | DocumentFragment'.
    Type 'null' is not assignable to type 'Element | DocumentFragment'.
    Import App from './App'.
  
```

Basicamente ele está dizendo que alguns tipos de dados no arquivo App.tsx e index.tsx não foram definidos.

Primeiro iremos arrumar os erros no arquivo index.tsx

A screenshot of a code editor showing a TypeScript file named `index.tsx`. The code is a simple React application setup. Line 6 contains the problematic line: `const root = ReactDOM.createRoot(document.getElementById("root"));`. The `getById` method is highlighted in red, indicating a type error. A tooltip or status bar at the bottom shows the message "Type 'HTMLElement | null' is not assignable to type 'Element'.ts(2322)".

```

1 import React from "react";
2 import ReactDOM from "react-dom/client";
3 import "./index.css";
4 import App from "./App";
5
6 const root = ReactDOM.createRoot(document.getElementById("root"));
7 root.render(
8   <React.StrictMode>
9     <App />
10    </React.StrictMode>
11 );
12

```

Aqui o TypeScript está pedindo para tipar o retorno de `getElementById`. O tipo do retorno de `getElementById` é `HTMLElement`. A maneira certa de resolver isso é definir o tipo ou usar um “!”. Iremos definir o tipo do retorno (com “`as HTMLElement`”) para tirar o erro.

A screenshot of a code editor showing the same `index.tsx` file after the fix. The problematic line now includes the `as HTMLElement` cast: `document.getElementById("root") as HTMLElement`. The red highlighting and error message are no longer present, indicating the error has been resolved.

```

1 import React from "react";
2 import ReactDOM from "react-dom/client";
3 import "./index.css";
4 import App from "./App";
5
6 const root = ReactDOM.createRoot(
7   document.getElementById("root") as HTMLElement
8 );
9 root.render(
10   <React.StrictMode>
11     <App />
12    </React.StrictMode>
13 );
14

```

Agora iremos arrumar os erros no arquivo App.tsx. Primeiro iremos ver os erros que estão sendo alertados em nossa IDE:

```
return (
  <ul>
    {repositorio.map((repositorio) => (
      <li key={repositorio.id}>{repositorio.name}</li>
    )));
)
```

O TypeScript está reclamando que o "id" e "name" não foram tipados, dessa forma, o retorno da nossa API pode ser qualquer tipo de dados.

Para resolver isso iremos criar uma Interface e tipar o "id" e "name":

```
1 interface IRepository {
2   id: number;
3   name: string;
4 }
```

Agora é hora de chamar a Interface no retorno do nosso .map()

```

return (
  <ul>
    {repositorio.map((repositorio: IRepositoryos) => (
      <li key={repositorio.id}>{repositorio.name}</li>
    )))
  </ul>
);

```

Após realizar isso nossa aplicação irá rodar normalmente.

Uma outra forma de arrumar esse erro é chamar a nossa Interface no useState dessa forma:

```

function App() {
  const [repositorio, setRepositorio] = useState<IRepositoryos[]>([]);
  useEffect(() => {

```

Sei que essa foi uma forma básica de como tipar dados em aplicações React, mas com esse exemplo conseguimos ter uma noção melhor de como funciona o TypeScript.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fazendo deploy e implementando versões móveis de sites, é comum que surjam muitas dúvidas para quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, trazendo resultados melhores em suas competências de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **medo** de problemas para entrar na área de programação, este é o seu guia para superá-los! Com dicas em suas etapas, desde ideias, enunciados, códigos e testes, até o seu **CV** e **LinkedIn**, perfeitas e práticas para alcançar a categoria dos resultados.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 9

styled-components

Reserve o café e prepare-se para mergulhar nesse módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes com os conhecimentos adquiridos através de um questionário no Módulo.

+ 7 TÓPICOS NESTE MÓDULO

Introdução

A grande vantagem de se utilizar React como tecnologia de desenvolvimento é a sua componentização, isso nos ajuda a reaproveitar um pedaço de lógica em vários lugares da nossa aplicação. Com styled-components isso não é diferente, essa biblioteca funciona da mesma forma só que com estilos e utilizando o conceito CSS-in-JS. Além disso, a biblioteca styled-components permite que você escreva CSS simples para seus componentes sem se preocupar com colisões dos nomes de classes, assim, cada componente conhece apenas o seu CSS, uma mudança em um componente será refletida somente nele.

O que é

Basicamente, styled components é uma biblioteca que utiliza o conceito de CSS-in-JS, ou seja, que nos permite escrever códigos CSS dentro do JavaScript ao nível de componente na sua aplicação. Com ele, você pode criar websites bonitos e funcionais. Além disso, ganhar mais agilidade e precisão no desenvolvimento frontend.

Um pouco mais sobre CSS-inJS: o método de estilização CSS-in-JS aplica estilos CSS em componentes individualmente em lugar de aplicá-los no documento HTML como um todo.

Isso significa que a estilização CSS se aplica exclusivamente no escopo do componente e não no documento globalmente.

Vantagens

Styled Components foi criado pelos seguintes motivos:

- Automatização de CSS crítico: styled-components controlam automaticamente quais componentes e seus respectivos estilos serão renderizados em uma página.
- Eliminação de bugs devido a colisão de nomes de classes: styled-components geram nomes de classe exclusivos para seus estilos. Você não precisa se preocupar com duplicação, sobreposição ou erros de ortografia.
- Segurança na exclusão de CSS: pode ser difícil saber se um nome de classe é usado em algum lugar na sua base de código. styled-components tornam isso óbvio, pois cada estilo está ligado a um componente específico. Se o componente não for utilizado ou for excluído, toda sua estilização será excluída.

- Simplificação da estilização dinâmica: adaptar o estilo de um componente com base em props ou em um tema global é uma tarefa simples, intuitiva e não requer gerenciamento manual de dezenas de classes.
- Facilidade de manutenção: você nunca precisará procurar em diferentes arquivos para encontrar o estilo que se aplica ao seu componente; portanto, a manutenção é muito fácil, não importa o tamanho da sua base de código.

Iniciando

Já com o projeto React criado, iremos agora instalar o styled-components, para isso requer apenas um único comando para utilizar:



```
npm install styled-components
```

A documentação oficial do styled-components recomenda se você usa um gerenciador de pacotes como o yarn que oferece suporte ao campo "resolutions" no package.json. Isso ajuda a evitar problemas que surgem da instalação de várias versões do styled-component em seu projeto.

```
package.json

{
  "resolutions": {
    "styled-components": "^5"
  }
}
```

Aplicando estilos

Depois de instalar a biblioteca vamos criar nosso primeiro componente. No arquivo App.js dentro da pasta src iremos ter o seguinte código:

```
import styled from "styled-components";

const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

function App() {
  return <Title>Hello World!</Title>;
}

export default App;
```

O resultado será esse:

Hello World!

Calma, irei explicar o que aconteceu exatamente.

```
1 const Title = styled.h1`  
2   font-size: 1.5em;  
3   text-align: center;  
4   color: palevioletred;  
5 `;
```

No código acima criamos um componente chamado Title (você pode escolher qualquer nome para o componente), logo em seguida chamamos o styled da importação do styled-components:

```
1 import styled from "styled-components";
```

Após o styled dizemos qual é a tag do elemento, como no exemplo foi só um texto coloquei como h1. Depois de criado é só chamar o componente no retorno da função.

Obs: Lembra que falei que a biblioteca styled-components permite que você escreva CSS simples para seus componentes sem se preocupar com colisões dos nomes de classes? Se a gente inspecionar o elemento criado teremos a seguinte classe:

Hello World!

```
h1 {color: #D9EAD3;}
```

Estilos dinâmicos

O styled-components nos permite passar uma função como o valor do estilo, isto é, passar estilos baseado nas props, ao invés de adicionar classes ao nosso arquivo, como é feito tradicionalmente com CSS.

Iremos utilizar o mesmo exemplo e mudar de forma dinâmica.

```
1 const Title = styled.h1`  
2   font-size: 1.5em;  
3   text-align: center;  
4   color: ${props} => (  
5     props.iuricode ? "green" : "palevioletred"  
6   );  
7 `;
```

No exemplo acima (na propriedade color) fizemos uma verificação na props do componente Title. Caso tenha a propriedade "iuricode" iremos adicionar o cor verde (green) no elemento, caso não tenha iremos manter a mesma cor utilizada no primeiro exemplo.

Para um melhor entendimento irei renderizar dois componentes Title, dessa forma:

```
● ● ●  
1 function App() {  
2   return (  
3     <div>  
4       <Title iuricode>Hello World!</Title>  
5       <Title>Hello World!</Title>  
6     </div>  
7   );  
8 }
```

Como no primeiro componente chamado temos a prop "iuricode" teremos a seguinte renderização em nossa aplicação:

Hello World!

Hello World!

Estendendo estilos

O styled-components também permite estender estilos entre os componentes de forma bem simples.

```
1 const Title = styled.h1`  
2   font-size: 1.5em;  
3   text-align: center;  
4   color: palevioletred;  
5 `;  
6  
7 const Title2 = styled(Title)`  
8   font-size: 4.5em;  
9 `;
```

Dessa forma acabamos de criar um componente chamado "Title2" e adicionamos os mesmos estilos do componente "Title", porém aumentamos o tamanho da fonte. O resultado é esse:

Hello World!

Hello World!

Arquivos separados

"Okay Iuri, mas é estranho colocar os estilos no mesmo arquivo da lógica da aplicação!" É, de fato é mesmo! Para resolver isso, o styled-components nos dá a possibilidade de separar a lógica do estilo, isso é, criando arquivos separados.

Iremos criar um arquivo chamado "Title.js", nele iremos adicionar todos os estilos já utilizados nos exemplos anteriores. Porém, antes de "const" na criação do componente iremos colocar o "export" para assim exportar nosso arquivo de estilo no arquivo de lógica.

```
● ● ●
1 import styled from "styled-components";
2
3 export const Title = styled.h1`  

4   font-size: 1.5em;  

5   text-align: center;  

6   color: palevioletred;  

7 `;
```

Feito isso basta agora importar dessa forma no nosso arquivo App.js

```
1 import { Title } from "./Title.js";
```

Okay luri, mas toda vez que eu tiver um componente/elemento vou ter que importar ele assim? Se todos os elementos tiver dentro de um elemento pai não será preciso criar componente por componente no styled-components, pois ele funciona da mesma forma do Sass, basta chamar o elemento pai.

No exemplo a seguir temos uma tag span dentro do componente Title:

```
1 <Title>
2   Hello World! <span>texto maior</span>
3 </Title>
```

Para estilizar a tag span bata chamar ele dentro do componente Title, dessa forma:

```
● ● ●  
1 import styled from "styled-components";  
2  
3 export const Title = styled.h1`  
4   font-size: 1.5em;  
5   text-align: center;  
6   color: palevioletred;  
7  
8   span {  
9     font-size: 4.5em;  
10 }  
11 `;
```

Teremos o seguinte resultado porém sem precisar criar um componente para o span:

Hello World! **texto maior**

Estilos globais

Em muitos casos queremos aplicar estilos globais em nossa aplicação. Dentro do da pasta src crie um arquivo chamado globalStyles.js

Nele, usaremos uma função chamado `createGlobalStyle` do `styled-components` e adicionaremos alguns estilos globais, dessa forma:

```
1 import { createGlobalStyle } from "styled-components";
2
3 export const GlobalStyle = createGlobalStyle`
4   body {
5     margin: 0;
6     padding: 0;
7     font-family: Open-Sans, Helvetica, Sans-Serif;
8     background-color: #121212;
9   }
10 `;
```

Agora iremos importar o componente `GlobalStyle` no `index.js` para que seja aplicado em todos documentos.

```
1 import React from "react";
2 import ReactDOM from "react-dom/client";
3 import App from "./App";
4 import { GlobalStyle } from "./globalStyles.js";
5
6 ...
```

```
7  const root = ReactDOM.createRoot(  
8    document.getElementById("root")  
9  );  
10 root.render(  
11   <React.StrictMode>  
12     <GlobalStyle />  
13     <App />  
14   </React.StrictMode>  
15 );
```

Dessa forma todo estilo adicionado no arquivo globalStyles.js será aplicado de forma global em nossa aplicação.

Hello World! **texto maior**

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fazendo deploy e implementando versões móveis de sites, é comum que surjam muitas dúvidas para quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, treinamento você também realizará sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **problemas para entrar na área de programação**, este é o seu guia para começar a programar em suas redes sociais, empresas, serviços bancários e muito mais! O HTML, CSS, JavaScript, Python, Java, MySQL e muito mais!

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 01

Tailwind CSS

Reserve seu café e prepare-se para o maior módulo da Umbra! Neste módulo, ao final de todos os tópicos, você poderá iniciar suas conquistas com Tailwind CSS. Só é necessário que você responda a perguntas por meio de um questionário no Módulo.

+ 7 TÓPICOS NESTE MÓDULO

praticando

Introdução

Neste módulo, vamos aprender sobre um dos frameworks CSS que mais estão em alta no mercado, que tem como objetivo auxiliar na construção e design de páginas feita com HTML, React e outras bibliotecas JS.

O que é

Assim como o Bootstrap, Tailwind oferece a possibilidade de você criar layouts usando uma estrutura de CSS pronta. Isso permite que você otimize o tempo de criação de uma interface sem precisar fazer tudo manualmente.

O Tailwind é um framework baseado em utilidades e tem como prioridade a facilidade de customização. Estilizar elementos com Tailwind é quase como escrever estilizações inline (escrever CSS dentro do atributo style), só que com classes. Por isso ele é um framework focado em utilidades.

```
1 <p className="text-lg font-medium">
2   Sou um texto estilizado pelo Tailwind
3 </p>
```

Iniciando

O Tailwind normalmente recomenda que você instale por meio de um gerenciador de pacotes para obter a funcionalidade completa.

Já com o projeto React criado, iremos agora instalar o Tailwind e iniciar a configuração, para isso requer apenas um único comando para utilizar:

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

Agora vamos configurar os caminhos de arquivos que vão ter as classes do Tailwind.

```
1  /** @type {import('tailwindcss').Config} */
2
3  module.exports = {
4    content: ["./src/**/*.{js,jsx,ts,tsx}"],
5    theme: {
6      extend: {},
7    },
8    plugins: [],
9  };
```

Remove o CSS do arquivo index.css e adicione as diretivas do Tailwind.

```
1  @tailwind base;
2  @tailwind components;
3  @tailwind utilities;
```

Agora que temos o Tailwind adicionado na aplicação. Vamos começar adicionando um tamanho na fonte e uma cor em nosso parágrafo (<p>) no arquivo index.js onde contém o JSX da página inicial. Podemos fazer isso adicionando a classe text-3xl e text-red-500:

```
● ● ●  
1 const App = () => {  
2   return <p className="text-3xl text-red-500">Olá</p>  
3 }  
4  
5 export default App;
```

Por fim, execute o projeto com o comando:

```
● ● ●  
npm start
```

A partir de agora, você está com o Tailwind CSS instalado em um projeto Next.js e você pode começar a usar as classes no seu HTML.

Para saber os nomes das classes, você pode consultar a [documentação do Tailwind](#) ou utilizar ferramentas como o [Tailwind CSS IntelliSense](#) que facilita o desenvolvimento com o preenchimento automático.

Imagens da extensão Tailwind CSS IntelliSense recomendando as classes com seu preenchimento automático:

```

→ [
  assName="text-3xl text-red-500 font">Olá</p>;
    □ font-s... font-family: ui-sans-seri
    □ font-serif
    □ font-mono
    □ font-thin
    □ font-extralight
  App;
]

```

Responsividade

No Tailwind é utilizado a abordagem de mobile-first, similar ao que temos no Bootstrap, e isto quer dizer que quando um estilo é adicionado ao código, ele é aplicado da menor para a maior resolução.

Os breakpoints são organizados em 5 grupos e conseguimos atender a uma variedade bem ampla de dispositivos:

Prefixo	Resolução	CSS
sm	640px	@media (min-width: 640px) { ... }
md	768px	@media (min-width: 768px) { ... }
lg	1024px	@media (min-width: 1024px) { ... }
xl	1280px	@media (min-width: 1280px) { ... }
2xl	1536px	@media (min-width: 1536px) { ... }

Esses breakpoints são usados utilizando os prefixos como modificadores. Veja na prática:

```
1 const App = () => {
2   return (
3     <p className="text-red-500 md:text-indigo-900">
4       Oi
5     </p>
6   );
7 };
8 export default App;
```

Neste exemplo falei que o padrão da cor do texto é “text-red-500”, mas quando estiver com resolução maior ou igual a 768px, terá a cor “text-indigo-900”. Se você não conhece a abordagem de mobile-first faça um teste para entender melhor o seu funcionamento.

Estados dos elementos

Assim como temos modificadores para resoluções, temos como adicionar modificadores para estados dos elementos, tão simples como o exemplo anterior.

Neste exemplo quero alterar a cor do botão ao passar o mouse sobre o elemento (hover):

```
1 const App = () => {
2   return (
3     <p className="text-red-500 hover:text-indigo-900">
4       Oi
5     </p>
6   );
7 };
8 export default App;
```

O mais legal é que o Tailwind inclui modificadores para praticamente tudo que você precisa:

- Pseudo-classes como :hover, :focus, :first-child e :required.
- Pseudo-elementos como ::before, ::after, ::placeholder e ::selection.

Dark mode

O Dark Mode é uma opção que está se tornando cada vez mais comum criar uma versão escura do seu site para acompanhar o design padrão.

O Tailwind inclui uma variante "dark" que permite estilizar seu site de maneira diferente quando o modo escuro está ativado:

```
1 const App = () => {
2   return (
3     <button className="bg-white dark:bg-gray-900">
4       Sou um botão
5     </button>
6   );
7 };
8 export default App;
```

Assim como o estado do elemento e resolução, adicionar a variante dark é simples com Tailwind.

Diretivas

As diretivas são regras CSS ([at-rules](#)) personalizadas do framework para usar no CSS e ter acesso a funcionalidades especiais do Tailwind. As diretivas são três:

@tailwind

Esta é a principal diretiva do Tailwind, responsável por aplicar os estilos CSS, da base, de componentes, utilitários e variantes do framework. Quando criar um projeto do zero, você vai precisar registrar esta diretiva para conseguir usar as classes no HTML.

```
● ● ●  
1 @tailwind base;  
2 @tailwind components;  
3 @tailwind utilities;  
4 @tailwind variants;
```

@layer

Esta diretiva permite informar ao Tailwind a qual grupo o conjunto de estilos e classes fazem parte. Por exemplo, você pode querer mudar o padrão de fonte das tags Heading.

```
1 @layer base {  
2   h1 {  
3     @apply text-2xl;  
4   }  
5  
6   h2 {  
7     @apply text-xl;  
8   }  
9 }
```

@apply

Se você utilizar HTML para desenvolvimento, os elementos podem ficar lotados de classes, sem contar a repetição e a manutenção para vários componentes ao mesmo tempo. Por isso, o Tailwind permite que você crie classes personalizadas que recebem as classes de utilidade, e depois basta chamar as classes criadas que os componentes vão receber todas as propriedades selecionadas.

```
1 .botao-iuricode {  
2   @apply text-base font-medium p-3;  
3 }
```



```

1 <button className="botao-iuricode">
2   Sou um botão
3 </button>

```

Extra

Cores

O Tailwind CSS vem com um monte de cores pré-definidas. Cada cor tem um conjunto de variações diferentes, com a variação mais clara sendo 50 e a mais escura sendo 900.

Aqui está uma imagem de múltiplas cores e seus códigos HTML hexadecimal para ilustrar isto:



Esta sintaxe é a mesma para todas as cores em Tailwind, exceto para preto e branco, que são escritas da mesma maneira, mas sem o uso de números bg-black e bg-white.

Padding e Margin

O Tailwind CSS já tem um sistema de design que ajuda a manter uma escala consistente em todo design no site. Tudo que você tem que saber é a sintaxe para aplicar cada utilidade.

A seguir estão as utilidades para adicionar padding aos seus elementos:

- p: determina o preenchimento de todo o elemento
- py: determina padding padding-top e padding-bottom
- px: determina padding-left e padding-right
- pt: determina padding-top
- pr: determina padding-right
- pb: determina padding-bottom
- pl: determina padding-left

Para aplicá-los aos seus elementos, você teria que usar os números apropriados fornecidos pelo Tailwind:

```
1 <button className="p-4 bg-slate-400">  
2 Sou um botão  
3 </button>
```

As utilidades pré-definidas para padding e margin são muito similares. Você só tem que substituir o “p” por um “m”.

Sizing

O sizing é um utilitário que define a largura e altura de um elemento. As utilidades pré-definidas para padding e margin são muito similares. Você só tem que substituir o “w” por um “h”.

A seguir estão as utilidades para adicionar largura (e da mesma forma a altura) aos seus elementos:

w-0	width: 0px;
w-5	width: 1.25rem; /* 20px */
w-8	width: 2rem; /* 32px */
w-10	width: 2.5rem; /* 40px */
w-auto	width: auto;
w-1/6	width: 16.666667%;
w-1/4	width: 25%;
w-5	width: 33.333333%;
w-2/4	width: 50%;
w-full	width: 100%;
w-screen	width: 100vw;
w-min	width: min-content;
w-max	width: max-content;
w-fit	width: fit-content;

Lembre-se: não é errado sempre pesquisar para resolver um problema.

NENHUM desenvolvedor faz algo do zero, sempre a uma referência/pesquisa antes. Então está tudo bem não decorar, o importante é a entrega e solucionar o problema.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é um complemento ao meu curso de Desenvolvimento Frontend. Aprendendo a construir frontends, fazendo deploy e implementando versões móveis de sites, é comum que surjam muitas dúvidas para quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de comprovar sua aplicação, trazendo resultados melhores em suas competências de UX.

[QUERO COMPRAR TAMBÉM](#)

Se você está com **medo** de problemas para entrar na área de programação, este é o seu guia para superá-los. Com dicas em suas etapas, desde ideias, enunciados, códigos e testes, até o seu **CV** e **LinkedIn**, perfeitas e práticas para alcançar a categoria dos resultados.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 10

React Hook Form

Regras de validação e tratamento de erros no formulário. Saiba como usar o hook de tratamento de erros para tratar erros de validação de forma mais eficiente. Descubra como usar o hook de tratamento de erros para tratar erros de validação de forma mais eficiente.

* 8 TÓPICOS NESTE MÓDULO

Introdução

Os formulários são uma parte essencial de como os usuários interagem com sites. Validar os dados do usuário passados pelo formulário é uma responsabilidade crucial para um desenvolvedor. No entanto, devido à complexidade de alguns formulários, eles podem parecer intrusivos e afetar a experiência do usuário.

Para resolver esse problema, você aprenderá como usar a biblioteca React Hook Form para criar formulários excelentes no React sem usar render props complicados ou componentes de ordem superior (HOC).

O que é

React Hook Form é uma biblioteca que ajuda a validar formulários no React. É muito eficiente, adaptável e super simples de usar. Também é leve, sem dependências externas, o que ajuda os desenvolvedores a obter mais resultados enquanto escrevem menos código. Outra facilidade que ele traz é na melhora significativa de desempenho, já que a ação de renderização também pode ser controlada. Dessa forma, apenas as alterações de entradas são re-renderizadas, não o formulário inteiro.

Iniciando

Para iniciar com React Hook Form não tem segredo. A instalação requer apenas um único comando e você está pronto para começar.



```
npm install react-hook-form
```

Para começar iremos utilizar o exemplo disponível na documentação do React Hook Form, caso queria utilizar esse exemplo clique [aqui](#).



```
1 import { useForm } from "react-hook-form";
2
3 export default function App() {
4   const {
5     register,
6     handleSubmit,
7     watch,
8     formState: { errors },
9   } = useForm();
10
11 const onSubmit = (data) => console.log(data);
12 ...
```

```
13 ...
14 console.log(watch("example"));
15
16 return (
17   <form onSubmit={handleSubmit(onSubmit)}>
18     <input defaultValue="test"
19       {...register("example")}>
20
21     <input {...register("exampleRequired",
22       { required: true })}>
23
24     {errors.exampleRequired ??
25      <span>This field is required</span>}
26
27     <input type="submit" />
28   </form>
29 );
30 }
```

Essa é basicamente a estrutura de qualquer formulário. Onde contém as tags form, inputs e o botão de envio. Claro que teremos outras tags dentro de nosso formulários mas essas são as mais importantes.

Irei explicar o que está acontecendo no código nos seguintes tópicos.

Esse é um formulário simples sem utilizar React Hook Form. Agora iremos aplicar os recursos da biblioteca para criar as validações. Mas perceba que a forma que está não sabemos o que o usuário irá colocar em nosso input, ou se até mesmo ele irá colocar algo.

```
● ● ●  
1 export default function App() {  
2   return (  
3     <form>  
4       <input />  
5       <input type="submit" />  
6     </form>  
7   );  
8 }
```

Registrando campo

A biblioteca React Hook Form fornece o hook useForm para podemos utilizar os recursos para as validações. É dele que serão importados.

```
● ● ●  
1 import { useForm } from 'react-hook-form';
```

Após importar o hook iremos chamar os recursos dessa forma (isso já dentro do nosso componente React):

```
1 const { register } = useForm();
```

- register é uma função fornecida pelo hook useForm. Podemos atribuí-lo a cada campo de entrada (input) para que possamos rastrear as alterações no valor do campo de entrada.

Registro

Após chamar a função do hook já podemos adicionar o register em um campo de entrada. Para precisamos apenas chamar a função com o operador spread passando um nome exclusivo para cada register da seguinte forma:

```
1 <input {...register("nome")}>
```

Dessa forma o nosso formulário se encontra assim:

```
1 import { useForm } from "react-hook-form";
2
3 export default function App() {
4   const { register } = useForm();
5
6   return (
7     <form>
8       <input {...register("nome")}>
9       <input type="submit" />
10    </form>
11  );
12 }
```

Está tudo ok, mas ainda não falamos/chamamos a função que receberá os dados do formulário se a validação do formulário for bem-sucedida.

Para isso adicionamos a função onSubmit que é passada para o método handleSubmit assim:

```
1 ...
2 <form onSubmit={handleSubmit()}>
3 ...
```

Lembrando que o handleSubmit deve ser chamada no hook useForm.

```
1 const { register, handleSubmit } = useForm();
```

Agora vamos criar a função que irá mostrar o que foi digitado no input

```
1 const submitConsole = (data) => console.log(data);
```

Agora basta passar nossa função para dentro do método handleSubmit.

```
1 ...
2 <form onSubmit={handleSubmit(submitConsole)}>
3 ...
```

Dessa forma já está tudo pronto para ver o que foi digitado no campo e enviado para o console.log.

▶ {nome: 'Iuri'}

App.jsx:6

>

Aqui temos o nosso formulário completo até o momento:

```

1 import { useForm } from "react-hook-form";
2
3 export default function App() {
4   const { register, handleSubmit } = useForm();
5
6   const submitConsole = (data) => console.log(data);
7
8   return (
9     <form onSubmit={handleSubmit(submitConsole)}>
10    <input {...register("nome")} />
11    <input type="submit" />
12  </form>
13);
14}

```

Uma coisa bem simples que podemos fazer é adicionar um valor padrão para nosso input. Dependendo do formulário essa função pode ser muito útil e de grande ajuda na experiência do usuário. A função `defaultValue` que iremos usar não é importado do hook `useForm`, basta apenas chamar.

```

1 <input defaultValue="iuri" {...register("nome")} />

```

Aplicando validação

Agora que conseguimos enviar os dados de input no console do navegador, já pode criar uma validação para o mesmo.

Para adicionar validação, podemos passar um objeto para a função register como um segundo parâmetro como este:

```
1 <input ... register("nome", { required: true })>
2 <input ... register("senha",
3   { required: true, minLength: 6})>
4 />
```

Aqui estamos especificando a validação do campo como obrigatório.

Para o campo de senha, estamos especificando o campo obrigatório e a validação mínima de 6 caracteres.

Agora você deve estar se perguntando: "Ok luri, mas tudo isso da para fazer só com HTML sem precisar de uma biblioteca".

De fato! Mas tenha calma que ainda estamos no meio do caminho sobre validações em formulários.

Se a gente testar o código só com o required true em um campo vazio, apenas irá acontecer nada. Para a melhor experiência do usuário, quando ocorre um erro no preenchimento de um formulário damos uma instrução.

Para instruir o usuário iremos utilizar o objeto "erros" do hook useForm. Portanto iremos utilizar esse objeto para retornar uma mensagem caso o campo não seja preenchido.

Para isso iremos importar ele do hook dessa forma:

```
1 const { formState: { errors } } = useForm();
```

Agora para adicionar a validação temos que chamar nosso objeto "erros" e passar o nome do nosso campo que adicionamos na função register e logo em seguida a mensagem que queremos mostrar na tela.

```
1 <input {...register("nome", { required: true })} />
2 {errors.nome && <p>Coloque seu nome!</p>}
```

Como você pode ver, estamos recebendo erros de validação do campo de entrada assim que enviamos o formulário.

Mas ainda temos o campo de senha, que além de ter o preenchimento obrigatório, exigimos que tenha no mínimo 6 caracteres. Para criar uma validação para cada caso fazemos dessa forma:

```
1 <input {...register("senha",  
2   { required: true, minLength: 6 })}  
3 >  
4 {errors.senha ?? <p>Coloque sua senha!</p>}  
5 {errors.senha?.type === "minLength" ??  
6 <p>A senha deve ter no mínimo 6 caracteres</p>}
```

Assim como trabalhamos com mínimo de caracteres também podemos aplicar o máximo.

Por fim, a validação de padrão pode ser usada para definir o padrão regex para um campo de formulário.

O campo do formulário abaixo aceitará apenas letras minúsculas:

```
1 <input {...register("senha",  
2   {required: true, minLength: 6, pattern: /[a-z]/, })}  
3 >
```

Criando a mensagem do erro:

```
1 {errors.nome?.type === "pattern" && (
2   <p>Escreva em letras minúsculas!</p>
3 )}
```

Agora quando um usuário insere sua senha de usuário em letras maiúsculas, nossa validação irá disparar a mensagem de erro/aviso.

Criando componentes

Criar um componente de input é bem simples, única coisa que precisamos é passar o register e o nome que irei dar a ele pelo parâmetros.

```
1 import React from "react";
2 import { useForm } from "react-hook-form";
3
4 const Input = ({ label, register, required }) => {
5   return <input {...register(label, { required })}>
6 };
7 ...
```

```
1 ...
2 const App = () => {
3   const { register, handleSubmit } = useForm();
4   const onSubmit = (data) => console.log(data);
5
6   return (
7     <form onSubmit={handleSubmit(onSubmit)}>
8       <Input label="Nome" register={register} required />
9       <input type="submit" />
10    </form>
11  );
12};
13
14 export default App;
```

Watch

Este método observará o input e retornará seu valor. Isso é útil para renderizar o valor de entrada e para determinar o que renderizar por condição.

Para isso precisamos chamar esse método no hook useForm.

```
1 const { watch } = useForm();
```

Após isso basta apenas chama o método e passar o nome do register

```
1 console.log(watch("nome"));
```

Agora tudo que for digitado no campo nome irá aparecer no console.

Focus

Para uma experiência melhor para o usuário em alguns casos queremos que quando renderizar a página de formulário um campo ganhe um foque (focus) para ser digitado primeiro.

Para isso precisamos importar o setFocus do hook useForm.

```
1 const { setFocus } = useForm();
```

Agora só precisamos colocar o método setFocus junto com o nome do input dentro do hook useEffect.

```
1 useEffect(() => {
2   setFocus("nome");
3 }, [setFocus]);
```

Pronto!

Esses foram alguns conceitos básicos de como criar validações em seus formulários utilizando a biblioteca React Hook Form, mas não se limite somente a isso clique [aqui](#) e veja outras coisas que conseguimos fazer com essa maravilhosa biblioteca do mundo React.

Form Provider

Até agora aprendemos a como criar os campos e validar eles com o React Hook Form, mas temos um problema. Caso um dos nossos inputs seja um componente, isso é, esteja em um arquivo separado, não iremos conseguir aplicar a validação., pois o React Hook Form entende que o componente input não faz parte do formulário, e sim algo independente.

No código abaixo temos um exemplo de um formulário utilizando um componente de input mas sem utilizar o Form Provider:

```
1 import { useForm } from "react-hook-form";
2
3 function Senha() {
4     const { register } = useForm();
5     return <input {...register("senha")}>;
6 }
7
8 const onSubmit = (data) => console.log(data);
9
10 function App() {
11     const { register, handleSubmit } = useForm();
12
13     return (
14         <form onSubmit={handleSubmit(onSubmit)}>
15             <input {...register("nome")}>
16             <Senha />
17             <input type="submit" />
18         </form>
19     );
20 }
21
22 export default App;
```

Ao executar o código acima vemos que o valor digitado em nosso componente "Senha" não aparece no console, somente da tag input do "nome".

Para resolver isso utilizamos o Form Provider, que basicamente vai fazer que o formulário interprete que o componente Input faz parte do mesmo conteúdo do formulário.

Primeira coisa que temos que fazer é importar o FormProvider e o useFormContext do "react-hook-form".

```
1 import {  
2   useForm, FormProvider, useFormContext  
3 } from "react-hook-form";
```

Agora iremos mudar a chamada do nosso register do useForm para useFormContext. Dessa forma nosso componente Senha já está pronto.

```
1 function Senha() {  
2   const { register } = useFormContext();  
3   return <input {...register("senha")}>;  
4 }
```

No componente App iremos tirar a chamada do register e do handleSubmit no useForm. Iremos substituir eles pelo “methods” para assim todos os métodos fiquem no mesmo contexto.

```
1 const methods = useForm();
```

Agora iremos envolver todo nosso formulário dentro do FormProvider e dentro dele iremos chamar o “methods” que acabamos de chamar. Ele está passando todos os métodos para o contexto.

```
1 <FormProvider {...methods}>
2 ...
3 </FormProvider>
```

Após isso, basta adicionar o “methods” antes de chamar o register e o handleSubmit.

```
1 ...  
2 <form onSubmit={methods.handleSubmit(onSubmit)}>  
3   <input {...methods.register("nome")}>  
4 ...
```

Dessa forma o nosso formulário já consegue entender que o componente Input faz parte do formulário, pois todos os métodos estão dentro do mesmo contexto.

Esses foram alguns conceitos básicos de como criar validações em seus formulários utilizando a biblioteca React Hook Form, mas não se limite somente a isso clique [aqui](#) e veja outras coisas que conseguimos fazer com essa maravilhosa biblioteca do mundo React.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem se aprofundar na aprendizagem de front-end, fazer deploy, implementar corretamente de código e cultura ágil, além de outras palestras que desejam se tornar mais profissionais.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de conteúdos sobre design thinking, você também receberá sua compreensão de UX.

[QUERO COMPRAR TAMBÉM](#)

Este material tem **problemas para entrar na área de programação**, que são problemas que muitos iniciantes em suas primeiras horas, enfrentam, quando tentam entrar no mundo da programação. Porém, através desse material, você poderá ter uma visão clara das realidades que

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 12

Radix UI

Reserve o café e prepare-se para mergulhar nesse módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes com os conhecimentos adquiridos através de um questionário no Moodle.

* 6 TÓPICOS NESTE MÓDULO

Introdução

Neste módulo iremos aprender a utilizar a biblioteca de componentes Radix do React, mas calma, ela não é igual o Bootstrap ou outras bibliotecas de componentes. Com o Radix podemos utilizar componentes sem estilos e personalizar da forma que quisermos.

O que é

Como dito, o Radix é uma biblioteca de componentes de interface do usuário sem estilos, com foco em acessibilidade, personalização e experiência do desenvolvedor. Você pode usar esses componentes como a camada base do seu sistema e adicionar o estilo de preferência.

Iniciando

Iremos iniciar com o Radix com o exemplo utilizado na documentação com o componente Popover. O comando a seguir estamos instalando o componente Popover em nossa aplicação e não a biblioteca toda.



```
npm install @radix-ui/react-popover
```

Também iremos instalar a paleta de cores do Radix, que também é utilizada no exemplo do componente Popover.

```
npm install @radix-ui/colors
```

Pronto! Após as instalações já podemos importar o componente Popover em nossa aplicação. Primeira coisa que iremos fazer é importar e depois chamar o componente em nossa aplicação. Dessa forma:

```
1 import React from "react";
2 import * as Popover from "@radix-ui/react-popover";
3
4 const App = () => (
5   <h2>Radix</h2>
6 );
7
8 export default App;
```

Agora iremos chamar as peças do nosso componente Popover.

```
1 import React from "react";
2 import * as Popover from "@radix-ui/react-popover";
3
4 const App = () => (
5   <Popover.Root>
6     <Popover.Trigger>More info</Popover.Trigger>
7     <Popover.Portal>
8       <Popover.Content>
9         Some more info...
10        <Popover.Arrow />
11        </Popover.Content>
12      </Popover.Portal>
13    </Popover.Root>
14  );
15
16 export default App;
```

Dessa forma nosso componente Popover já está funcionando. Como pode ver nosso componente está totalmente sem estilos.

More info

More info

Some more info...

Aplicando estilos

Agora que já temos nosso componente funcionando iremos adicionar estilos a ele. Com Radix podemos estilar da forma e onde quisermos.

Agora iremos adicionar três classes:

```
1 import React from "react";
2 import * as Popover from "@radix-ui/react-popover";
3
4 const App = () => (
5   <Popover.Root>
6     <Popover.Trigger className="PopoverTrigger">
7       More info
8     </Popover.Trigger>
9     <Popover.Portal>
10       <Popover.Content className="PopoverContent">
11         Some more info...
12         <Popover.Arrow className="PopoverArrow" />
13       </Popover.Content>
14     </Popover.Portal>
15   </Popover.Root>
16 );
17
18 export default App;
```

Agora no index.css irei adicionar o seguinte código CSS:

```
1 .PopoverTrigger {  
2   display: inline-flex;  
3   align-items: center;  
4   justify-content: center;  
5   border-radius: 4px;  
6   padding: 0 15px;  
7   font-size: 15px;  
8   line-height: 1;  
9   font-weight: 500;  
10  height: 35px;  
11  background-color: white;  
12  color: var(--violet11);  
13  box-shadow: 0 2px 10px var(--blackA7);  
14  margin-top: 150px;  
15 }  
16  
17 .PopoverTrigger:hover {  
18   background-color: var(--mauve3);  
19 }  
20  
21 .PopoverTrigger:focus {  
22   box-shadow: 0 0 0 2px black;  
23 }
```

```
● ● ●  
24 .PopoverContent {  
25   border-radius: 4px;  
26   padding: 20px;  
27   width: 260px;  
28   font-size: 15px;  
29   line-height: 1;  
30   color: var(--violet11);  
31   background-color: white;  
32 }  
33  
34 .PopoverContent:focus {  
35   outline: none;  
36 }  
37  
38 .PopoverArrow {  
39   fill: white;  
40 }
```

Após isso iremos importar as cores que estamos usando em nosso código CSS do @radix-ui/colors, elas são: black, mauve e violet.

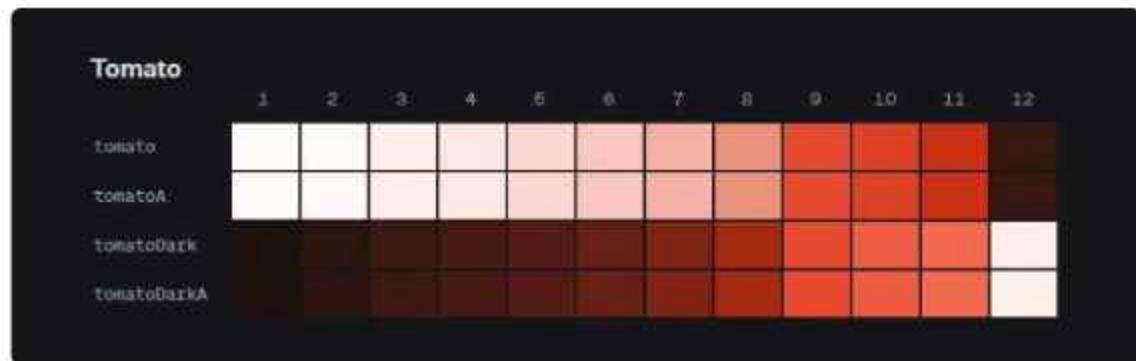
```
● ● ●  
1 @import "@radix-ui/colors/blackA.css";  
2 @import "@radix-ui/colors/mauve.css";  
3 @import "@radix-ui/colors/violet.css";
```

[More info](#)[Some more info...](#)

Cores

Okay, já entendemos como utilizar as cores da paleta de cores do Radix, basta importar e chamar elas em nosso código. Porém vale a pena abordar um pouco mais sobre elas, ainda mais se você estiver iniciando.

O Radix contém mais de 30 cores em sua paleta de cores com 12 escalas em cada uma delas. Veremos algumas:



Perceba em que nosso código importamos o "blackA" e não "black", dessa forma temos opacidade em nossa cor preta que no caso foi o "blackA7" utilizado na propriedade CSS box-shadow.



Ícones

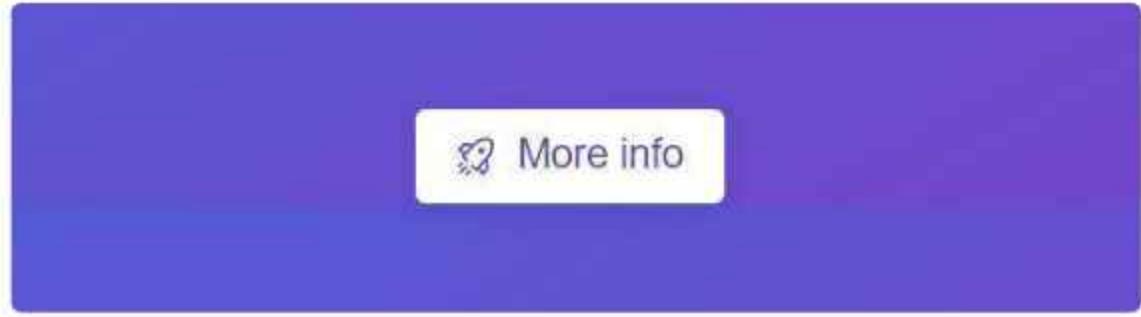
Antes de abordamos outros componentes iremos utilizar a biblioteca de ícones do Radix. O Radix Icons se apresenta como um "conjunto nítido de ícones 15x15 projetados pela equipe WorkOS." Todos os ícones estão disponíveis como componentes individuais que podem ser instalados por meio de um único pacote.

Simplesmente importe os ícones e você pode adicioná-los em sua aplicação como faria com qualquer outro componente do React.



Após instalar a biblioteca de ícone basta importar em nossa aplicação e chamar o mesmo. Para você ver o biblioteca de ícone clique [aqui](#).

```
1 ...
2 import { RocketIcon } from "@radix-ui/react-icons";
3
4 const App = () => (
5   ...
6   <Popover.Trigger className="PopoverTrigger">
7     <RocketIcon /> More info
8   </Popover.Trigger>
9   ...
10 );
11
12 export default App;
```



More info

Componentes

Já entendemos como utilizar o componente do Radix, mas para fixar o aprendizado iremos ver a utilização de mais um componente dessa biblioteca maravilhosa. Caso você queira ver a lista completa dos componentes disponíveis basta clicar [aqui](#).

Slider

O componente slider é uma entrada onde o usuário seleciona um valor dentro de um determinado intervalo.



Para utilizar este componente, e qualquer um outro, precisamos instalar ele em nossa aplicação e após isso importa-lo em nossa aplicação.



```
npm install @radix-ui/react-slider
```

```
1 ...
2 import * as Slider from "@radix-ui/react-slider";
3 ...
```

Após importar em nossa aplicação teremos o seguinte código do componente Slider:

```
1 ...
2 <form>
3   <Slider.Root
4     className="SliderRoot"
5     defaultValue={[50]}
6     max={100}
7     step={1}
8     aria-label="Volume"
9   >
10   <Slider.Track className="SliderTrack">
11     <Slider.Range className="SliderRange" />
12   </Slider.Track>
13   <Slider.Thumb className="SliderThumb" />
14 </Slider.Root>
15 </form>
16 ...
```

Agora iremos entender melhor o que cada peça do nosso componente Slider está fazendo. A sua anatomia consiste nos quatro elementos:

1. Root: contém todas as partes de um controle deslizante.
2. Track: a faixa que contém o Slider.Range.
3. Range: a parte do alcance que deve estar contida dentro Slider.Track.
4. Thumb: um lemento arrastável.

Aplicando CSS

Neste ponto, se você navegar para <http://localhost:3000> notará que a nossa aplicação está vazia. Isso ocorre porque os componentes Radix não aplicam nenhum estilo por padrão.

Agora veremos o processo de adição do CSS ao componente Slider.

Iremos criar as classes: SliderRoot, SliderTrack, SliderRange e SliderThumb. Além disso, iremos importar as cores blackA e violet.

O Root contém todas as partes de um controle deslizante e renderizará um input. Para ele iremos criar a classe "SliderRoot".

```
1 .SliderRoot {  
2   position: relative;  
3   display: flex;  
4   align-items: center;  
5   user-select: none;  
6   touch-action: none;  
7   width: 200px;  
8   height: 20px;  
9 }
```

O Track contém a parte do controle deslizante ao longo da qual o elemento arrastável fica. Para ele iremos adicionar a classe "SliderTrack".

```
10 .SliderTrack {  
11   background-color: var(--blackA10);  
12   position: relative;  
13   flex-grow: 1;  
14   border-radius: 100px;  
15   height: 3px;  
16 }
```

O Range representa o intervalo de valores que é selecionado. Para ele iremos adicionar a classe "SliderRange".

```
17 .SliderRange {  
18   position: absolute;  
19   background-color: white;  
20   border-radius: 100px;  
21   height: 100%;  
22 }
```

O Thumb é o elemento na trilha que o usuário pode mover para selecionar o valor. Para ele iremos adicionar a classe "SliderThumb".

```
23 .SliderThumb {  
24   display: block;  
25   width: 20px;  
26   height: 20px;  
27   background-color: white;  
28   box-shadow: 0 2px 10px var(--blackA7);  
29   border-radius: 10px;  
30 }
```

Iremos adicionar um hover e focos na classe SliderThumb.

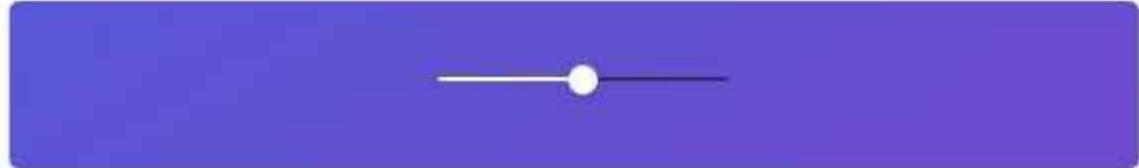
```
31 .SliderThumb:hover {  
32   background-color: var(--violet3);  
33 }  
34  
35 .SliderThumb:focus {  
36   outline: none;  
37   box-shadow: 0 0 0 5px var(--blackA8);  
38 }
```

Agora para finalizar iremos importar as cores.

```
1 @import "@radix-ui/colors/blackA.css";  
2 @import "@radix-ui/colors/violet.css";
```

Pronto! Com o mínimo de código, conseguimos criar um Slider acessível e funcional com estilo personalizado.

Este é o resultado final do nosso componente Slider:



Neste módulo desenvolvemos componentes personalizados, e incorporar recursos como acessibilidade em nossa aplicação React como a maravilhosa biblioteca de componentes Radix.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem se aprofundar na aprendizagem de front-end, fazer deploy, implementar corretamente de código e cultura ágil, além de outras palestras que desejam se tornar mais profissionais.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de conteúdos sobre design thinking, você também receberá sua compreensão de UX.

[QUERO COMPRAR TAMBÉM](#)

Desvendando os principais problemas para entrar na área de programação. Neste guia, ensino como iniciar seu caminho em suas primeiras horas, envolvendo conceitos básicos de programação, portfólio e outras dicas úteis para quem está a começar o seu caminho.

[QUERO COMPRAR TAMBÉM](#)



Módulo 1

Storybook

Reunião com o café expresso: tema da minicurso módulo. Lembrando que, ao final de cada módulo, você pode fazer testes e descobrimentos por meio de um questionário no Módulo.

* 6 TÓPICOS NESTE MÓDULO

Introdução

Neste módulo iremos utilizar um componente criado com styled-component e documentar sua variações de cenários diferentes antes de integrá-lo. Isso ajuda a garantir que seus componentes funcionem corretamente e parecem ótimos antes de serem implementados.

O que é

O Storybook é uma ferramenta muito útil para desenvolvedores de aplicações React. Ele permite criar e testar componentes de forma isolada, sem a necessidade de criar um aplicativo completo. Isso facilita o desenvolvimento e permite que os desenvolvedores criem componentes reutilizáveis e consistentes em suas aplicações.

Iniciando

Para começar a usar o Storybook, precisamos instalar o pacote npm do Storybook. Isso pode ser feito executando o seguinte comando:



```
npx storybook init
```

O Storybook examinará as dependências do seu projeto durante o processo de instalação e fornecerá a melhor configuração disponível.

O comando acima fará as seguintes alterações em seu ambiente local:

- Instalar as dependências necessárias.
- Configurar os scripts necessários para executar e construir o Storybook.
- Adicionar a configuração padrão do Storybook.
- Adicionar algumas stories padronizadas para você começar.
- Configurar a telemetria para nos ajudar a melhorar o Storybook.

```
Microsoft Windows [versão 10.0.22621.1365]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\luric\OneDrive\Área de Trabalho\storybook-components-npm-storybook-init

storybook init -- The simplest way to add a Storybook to your project.

• Detecting project type: ✓
• Adding Storybook support to your "Create React App" based project
```

Após a instalação iremos dar o seguinte comando:

```
npm run storybook
```

Talvez apareça uma mensagem pedindo a confirmação no processo de instalação do Storybook ou pedindo para você utilizar o npm7. Você pode confirmar o avanço digitando "y".

```
✓ Do you want to run the 'npm7' migration on your project? ... yes
  ✓ ran npm7 migration
  ✓ migration check successfully ran

To run your Storybook, type:
  npm run storybook

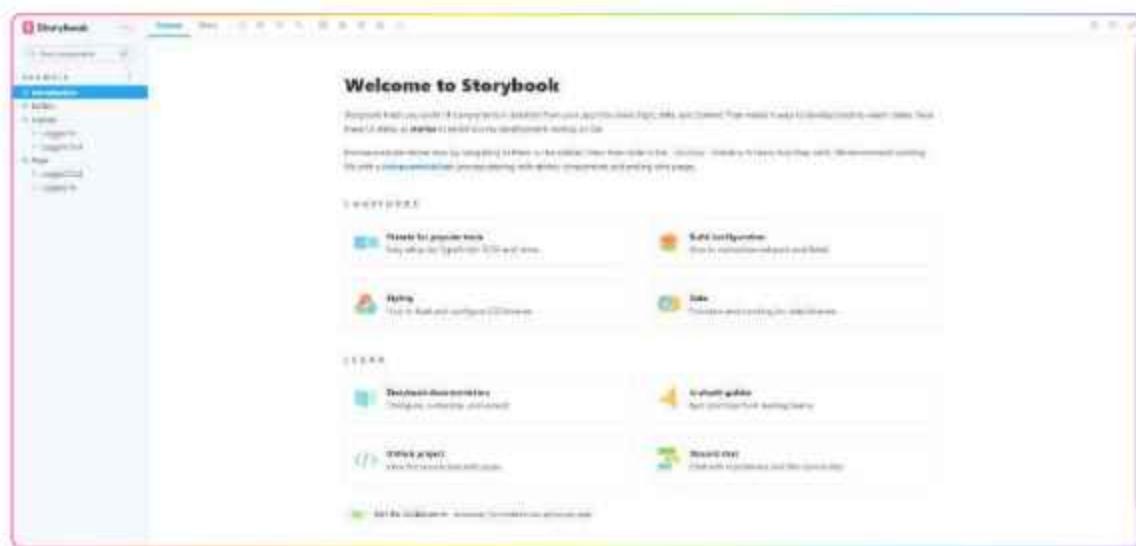
For more information visit: https://storybook.js.org
```

Ele iniciará o Storybook localmente e exibirá o endereço. Dependendo da configuração do seu sistema, ele abrirá automaticamente o endereço em uma nova guia do navegador e você será saudado por uma tela de boas-vindas.

```
info > using default Webpack setup
>>> [webpack-dev-middleware] wait until bundle finished
  38% building 8/16 entries 2/16 dependencies 8/2 modules
  info > Using cached manager
>>> [webpack-dev-middleware] wait until bundle finished: ./_webpack_hmr
99% done plugin webpack-hot-middleware[webpack].built preview visible in browser in 9:31 in 9987ms

Storybook 6.5.10 for React started
11 s for assets
Local: http://localhost:6006/
On your network: http://192.168.1.111:6006/
```

Caso ele não abra automaticamente o endereço em uma nova guia do navegador. Basta acessar a porta que o Storybook informará, no meu caso é o <http://localhost:6006/>



Perceba que em nossa aplicação foram adicionadas duas pastas: stories e .storybook. Além disso, temos o arquivo .npmrc para trabalhar com versões legados do npm.

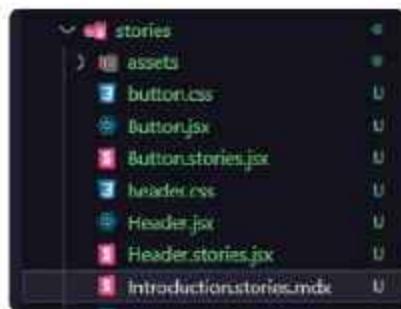
```

> storybook
> node_modules
> public
< src
  > stories
    App.js
    index.css
    index.js
    ignore
    .npmrc
    package-lock.json
    package.json
  
```

Estrutura do Storybook

Agora vamos entender melhor o que foi gerado pelo Storybook em nossa aplicação antes de documentar algo.

Perceba que dentro da pasta stories temos um arquivo chamado Introduction.stories.mdx que é exatamente a primeira página que aparece quando damos o comando npm run storybook no terminal.



Irei limpar este arquivo e deixar somente o seguinte código:

```

 1 import { Meta } from "@storybook/addon-docs";
 2
 3 <Meta title="Example/Introduction" />
 4
 5 # eFront
  
```

Após mudar o código no arquivo `Introduction.stories.mdx` iremos apagar os arquivos gerado pelo storybook na pasta `stories`, também iremos excluir a pasta “`assets`” de dentro de `stories`. Dessa forma, teremos somente o arquivo `Introduction` que modificamos.



Sua tela inicial deverá estar parecida com isso:



Agora estamos prontos para criar um componente e documentar ele no Storybook do zero.

Criando componente

Diferente do padrão do Storybook que adicionar os componentes na pasta stories, iremos adicionar nosso componente em uma pasta criada por nós. Dentro da pasta src irei criar uma pasta chamada “components”, e dentro de components teremos dois arquivos: Button.jsx e Button.stories.mdx



Dentro do arquivo Button.stories.mdx iremos colar o mesmo código do arquivo Introduction.stories.mdx porém iremos modificar alguns detalhes em seu código.

```

1 import { Meta } from "@storybook/addon-docs";
2
3 <Meta title="Components/Button" components={Button}/>
4
5 # Button

```

Mas tenha calma, ele ainda não irá aparecer em nossa documentação. Lembra que mudamos nosso componente da pasta stories para components? Então, agora precisamos avisar aos Storybook onde ele irá encontrar os componentes que serão documentados por ele.

Lembra a pasta .storybook gerada pelo Storybook na raiz da nossa aplicação? Basicamente nela temos nossa configuração do Storybook.

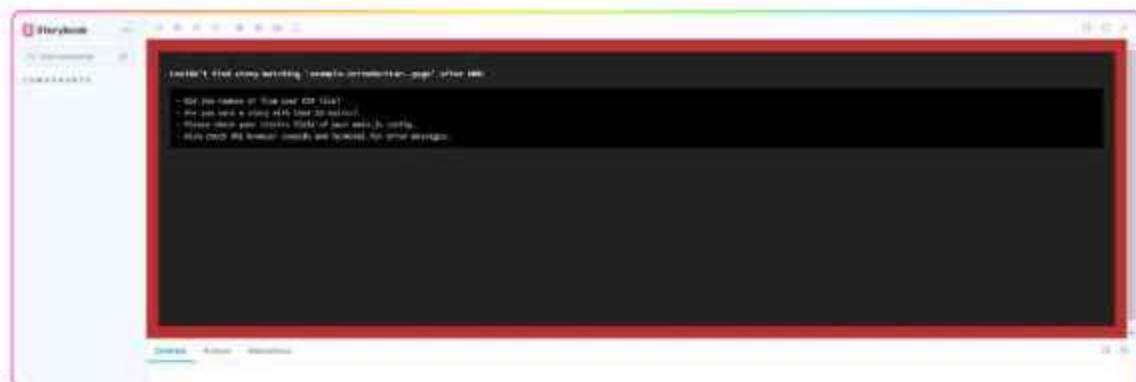


Dentro do arquivo main.js iremos mudar os seguintes como na imagem. Neste código estamos falando que nossos componentes estão em algum lugar de dentro da pasta src com o formato "stories".

```
module.exports = {
  "stories": [
    "../src/**/*.{stories.mdx}",
    "../src/**/*.{stories.@(js|jsx|ts|tsx)}"
  ],
}
```

Dependendo da versão do seu Storybook ele irá fazer toda essa configuração de mudança de pasta no main.js automaticamente.

Certamente você deve estar com o seguinte erro em sua aplicação:



Sempre que mudamos algo de dentro da pasta .storybook teremos que reiniciar a documentação com o comando `npm run storybook`.



Pronto! Após reiniciar o Storybook teremos a renderização do nosso arquivo `Button.stories.mdx` em nossa documentação.

Agora só falta criar o nosso botão. No arquivo Button.tsx teremos um código React retornando um simples botão:

```
 1 import React from "react";
 2
 3 const Button = () => {
 4   return <button>Sou um botão</button>;
 5 };
 6
 7 export default Button;
```

Documentando o componente

Agora iremos documentar o nosso componente. Primeira coisa que iremos fazer é importar e chamar o componente em nosso arquivo Button.stories.mdx.

```
 1 import { Meta } from "@storybook/addon-docs";
 2 import Button from "./Button";
 3
 4 <Meta title="Components/Button" components={Button} />
 5 # Button
 6 <Button />
```

Ao visualizar a documentação teremos a seguinte tela na aba Button:



Até aqui tudo bem, mas queremos adicionar as opções/recursos que o Storybook nos possibilita utilizar para deixar a documentação mais dinâmica. Para isso iremos importar os seguintes recursos:

```
● ● ●
1 import {
2   Meta, Story, Canvas, ArgsTable
3 } from "@storybook/addon-docs";
```

O primeiro que iremos utilizar é o ArgsTable. O ArgsTable gera automaticamente uma tabela de argumentos do componente. Essa tabela permitir que você altere os argumentos da story atualmente.

Iremos entender melhor na prática!

Para utilizar o ArgsTable apenas precisamos passar qual é o componente que desejamos listar os argumentos na tabela no parâmetro "of".

```

1 import {
2   Meta, Story, Canvas, ArgsTable
3 } from "@storybook/addon-docs";
4 import Button from "./Button";
5
6 <Meta
7   title="Components/Button" components={Button}
8 />
9
10 # Button
11
12 <ArgsTable of={Button} />
13
14 <Button />

```

Até o momento nossa documentação está da seguinte forma:



Aos utilizar a tabela de argumento precisamos utilizar o defaultProps para ter um valor padrão caso não seja informado. No arquivo Button.jsx iremos criar o parâmetro label que irá receber o valor de texto do nosso componente Button.

```

1 import React from "react";
2
3 const Button = ({ label }) => {
4   return <button>{label}</button>;
5 }
6
7 export default Button;
8
9 Button.defaultProps = {
10   label: "Sou um texto padrão",
11 };

```

Agora em nossa tabela de argumentos teremos nosso parâmetro label com seu valor default.



Perceba que na tabela na coluna Description nosso label está como unknown (desconhecido), isso por não falamos qual é o tipo do label.

Para isso iremos utilizar o PropTypes para a chegagem do tipo e dizer que nosso label é do tipo string. Para a sua utilização precisamos instalar o pacote via comando.

```
npm install prop-types
```

Após instalar basta importar e chamar em nosso componente Button.

```
1 import React from "react";
2 import PropTypes from "prop-types";
3
4 const Button = ({ label }) => {
5   return <button>{label}</button>;
6 }
7
8 export default Button;
9
10 ...
```

```
11 ...
12
13 Button.propTypes = {
14   label: PropTypes.string.isRequired,
15 };
16
17 Button.defaultProps = {
18   label: "Sou um texto padrão",
19 };
```

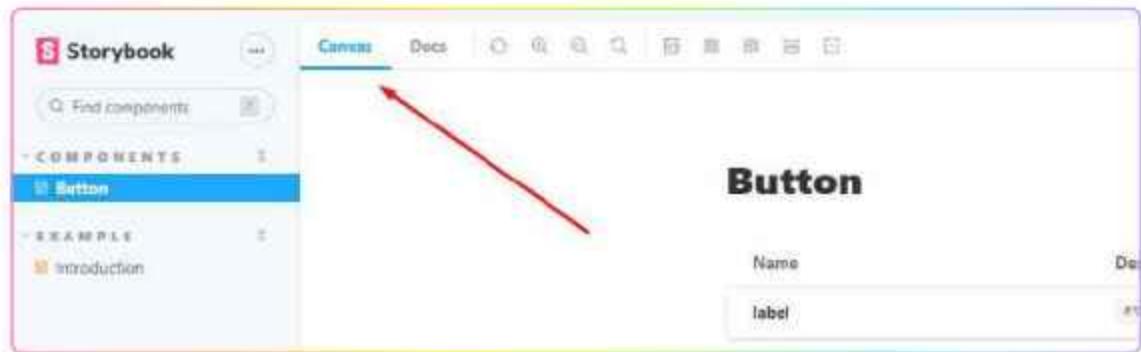
Assim como o defaultProps a utilização do PropTypes é muito simples então não entraremos em detalhes.

Agora se visualizarmos a documentação, na coluna Description teremos o tipo do label como string.



Agora iremos utilizar os outros recursos (Canvas e Story) que já importamos em nosso arquivo mdx.

O Canvas é um wrapper com uma barra de ferramentas que permite que você interaja com seu componente. Em nosso caso, ele será utilizado para mudar o texto.



Ao utilizar o Story abaixo do nosso Canvas teremos uma tabela com as opções de configuração do nosso componente. Então sempre que utilizarmos o Canvas iremos também ter um Story junto para interagir.

No arquivo `Button.stories.mdx` teremos o seguinte código utilizando os recursos citados:

```
1 <Canvas>
2   <Story
3     name="Story Button"
4     args={{
5       ...Button.defaultProps
6     }}
7   >
8   {((args) => <Button {...args} />)
9 </Story>
10 </Canvas>
```

Dentro do Story temos as propriedades name e args.

- name é a propriedade responsável por dar o nome ao Story do nosso componente.
- args fornece as entradas que nosso Story pode receber. Em nosso caso é ele o responsável por fornecer a entrada para mudar o texto.

Dessa forma nossa documentação ficará dessa forma:



Agora já podemos mudar o texto do nosso componente Button utilizando o Canvas do Storybook.

Entendendo o estrutura gerada

Agora que documentamos nosso componente Button fica mais fácil de entender o exemplo gerado pelo Storybook e começar a utilizar ele para documentar outros componentes.

Até o momento utilizamos arquivos no formato .mdx mas também é comum encontrar arquivo em .jsx para documentar componentes. O próprio Storybook utiliza .jsx nos exemplos gerado pelo comando npx storybook init e em sua documentação oficial utiliza .mdx. Então, não existe uma forma correta de utilizar, vai de acordo com suas pessoas da sua equipe ou empresa.

Veremos aqui o código gerado pelo Storybook (menos o CSS), veja que agora que aprendemos não é muito difícil de saber o que cada coisa faz.

Arquivo Button.stories.jsx:

```
● ● ●  
1 import React from "react";  
2 import { Button } from "./Button";  
3  
4 export default {  
5   title: "Components/Button",  
6   component: Button,  
7  
8   argTypes: {  
9     backgroundColor: { control: "color" },  
10    },  
11  };  
12  
13 const Template = (args) => <Button {...args} />;  
14  
15 export const Primary = Template.bind({});  
16  
17 Primary.args = {  
18   primary: true,  
19   label: "Button",  
20  };
```

Arquivo Button.jsx:

```
 1 import React from "react";
 2 import PropTypes from "prop-types";
 3 import "./button.css";
 4
 5 export const Button = ({
 6   primary, backgroundColor, size, label, ...props
 7 }) => {
 8   const mode = primary ? "storybook-button--primary"
 9     : "storybook-button--secondary";
10
11   return (
12     <button
13       className={[ "storybook-button",
14         `storybook-button--${size}`, mode].join(" ")}
15       style={backgroundColor && { backgroundColor }}
16       { ...props }
17     >
18       {label}
19     </button>
20   );
21 };
22 ...
```

```
25 ...
26 Button.propTypes = {
27   primary: PropTypes.bool,
28   backgroundColor: PropTypes.string,
29   size: PropTypes.oneOf(["small", "medium", "large"]),
30   label: PropTypes.string.isRequired,
31   onClick: PropTypes.func,
32 };
33
34 Button.defaultProps = {
35   backgroundColor: null,
36   primary: false,
37   size: "medium",
38   onClick: undefined,
39 };
```



Com este código é possível documentar variações de alguns componentes em nossa aplicação. Por mais que seja um exemplo utilizando um Button, a sua estrutura será a mesma para outros elementos no código.

Faça um teste você mesmo e se aprofunde no Storybook!

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem se aprofundar na aprendizagem de front-end, fazer deploy, implementar corretamente de código e cultura ágil, além de outras palestras que desejam se tornar mais profissionais.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de conteúdos sobre design thinking, você também receberá sua compreensão de UX.

[QUERO COMPRAR TAMBÉM](#)

Este material tem **problemas para entrar na área de programação**, que são problemas que muitos iniciantes em suas redes sociais, empresas e universidades enfrentam. São problemas que muitas vezes não conseguem ser resolvidos.

[QUERO COMPRAR TAMBÉM](#)



MÓDULO 14

Cypress

Reunião com o professor para tratar sobre o módulo. Lembrando que, ao final de todos os módulos, você pode fazer testes e questionários por meio de um questionário no Moodle.

→ 3 TÓPICOS NESTE MÓDULO

Introdução

Ouvimos muito falar sobre a importância de testar nosso código. Queremos ter feedback instantâneo sobre as mudanças que fazemos e ter confiança que o código funciona como planejado, sem precisar esperar ter uma aplicação pronta que possa ser validada. Para isso ser possível, temos diferentes tipos de testes automatizados que podemos implementar na nossa aplicação. Neste módulo iremos aprender a utilizar uma tecnologia muito utilizada por QAs, Cypress!

O que é

Cypress é um framework para automação de testes end-to-end, onde atualmente usa a linguagem JavaScript e roda em vários navegadores como Chrome, Firefox, Edge, entre outros. Com ele conseguimos simular um usuário interagindo na nossa aplicação e assim testar se as funcionalidades estão agindo corretamente.

Iniciando

Iremos utilizar uma aplicação pronta para iniciar os testes.

Para instalar o Cypress no projeto basta adicionar o seguinte comando:



```
npm install cypress
```

Após instalar iremos abrir o Cypress onde serão executados os testes.

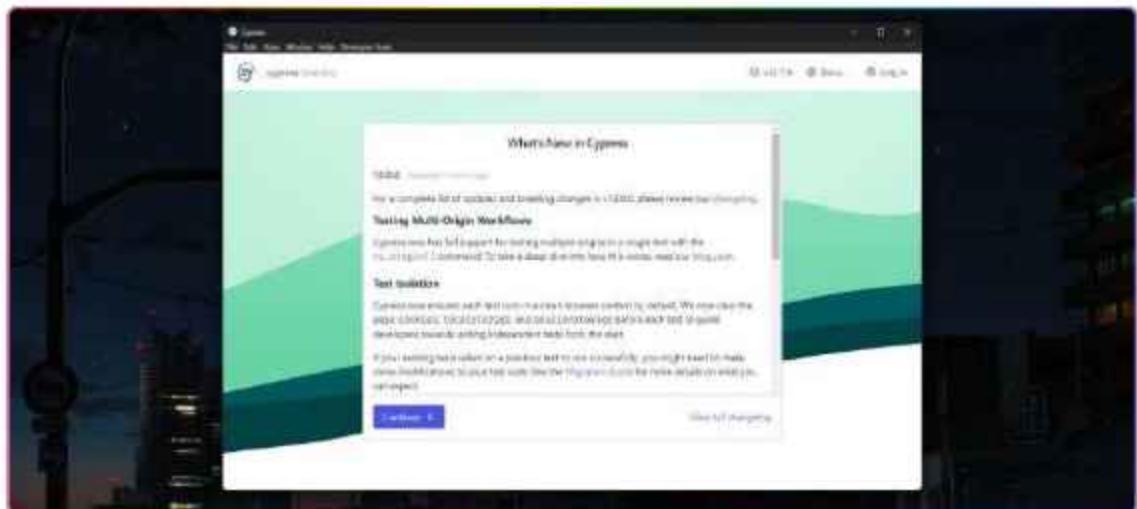


```
npx cypress open
```

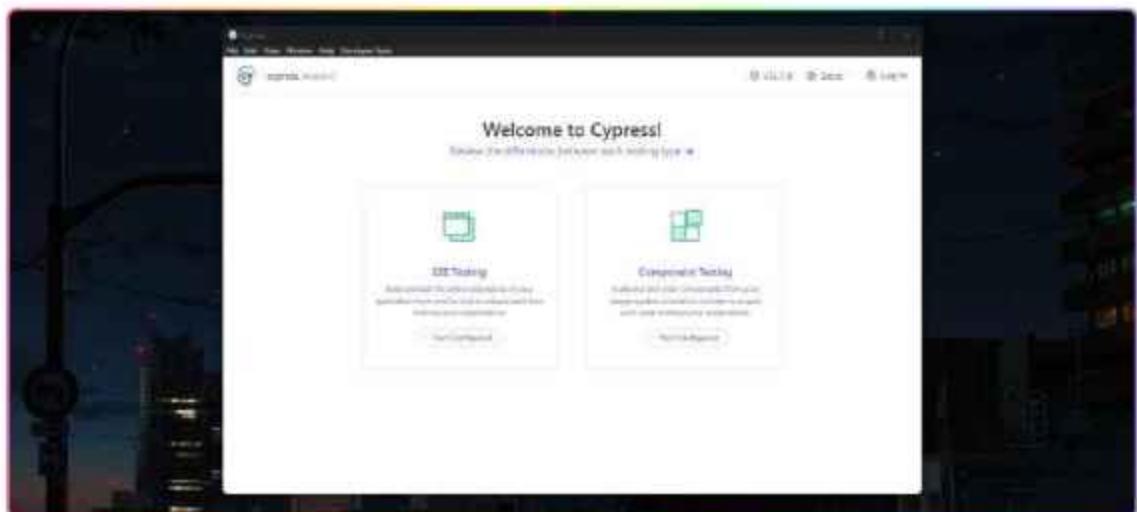
Agora entraremos em um processo de configuração do Cypress.

Se você estiver utilizando alguma tecnologia como o TypeScript talvez o Cypress solicite algum arquivo de configuração como o tsconfig.

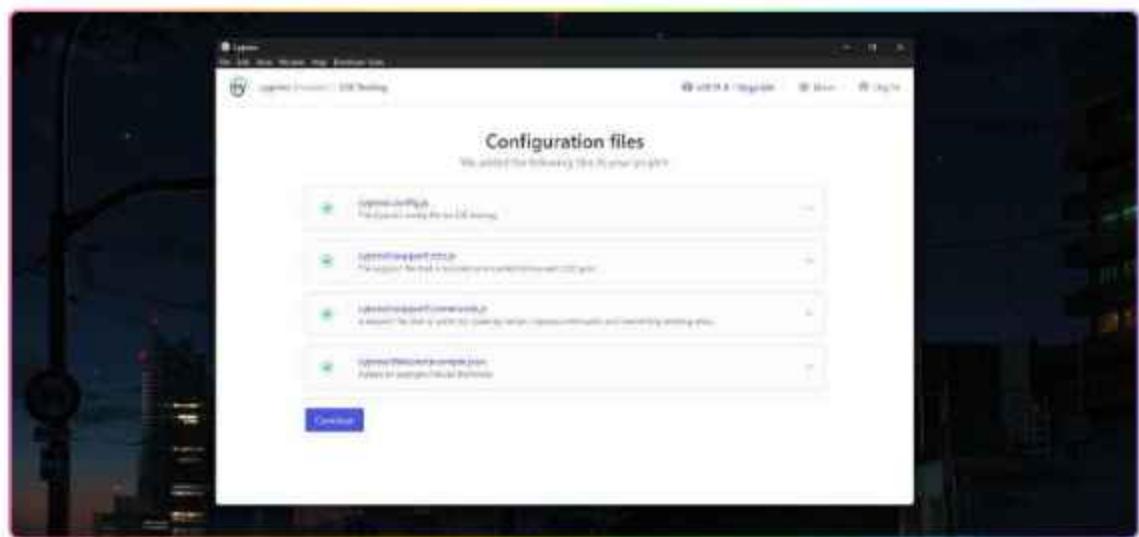
Após o comando cypress open irá abrir a seguinte tela falando sobre os novos recursos do Cypress e pedindo para você clicar em "Continue".



Após teremos uma tela de boas-vindas do Cypress solicitando para selecionar qual será o nosso tipo de teste. Iremos utilizar "E2E Testing".



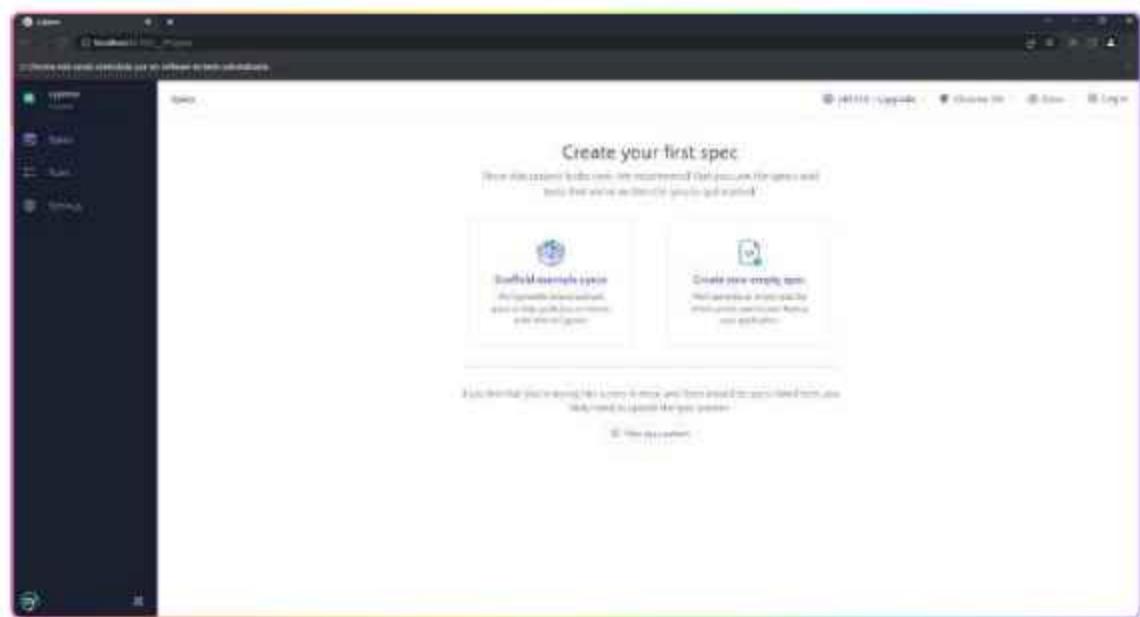
Depois de escolher E2E Testing ele irá adicionar alguns arquivos de configurações em nosso projeto para que o Cypress rode corretamente. Nesta etapa basta clicar em "Continue".



Na nova tela será solicitado para escolher qual navegador você quer rodar seus testes (podemos mudar depois).



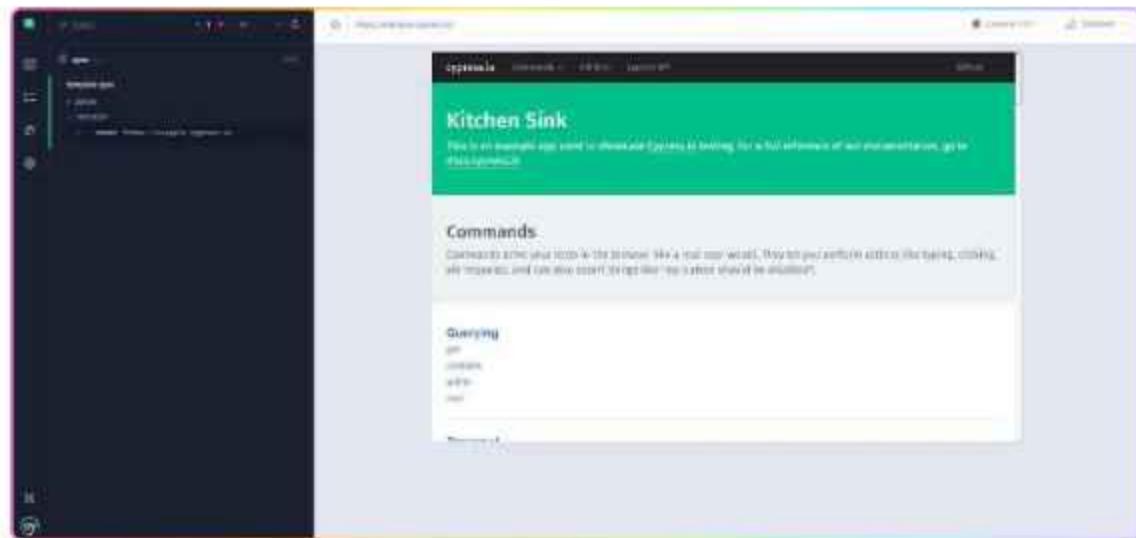
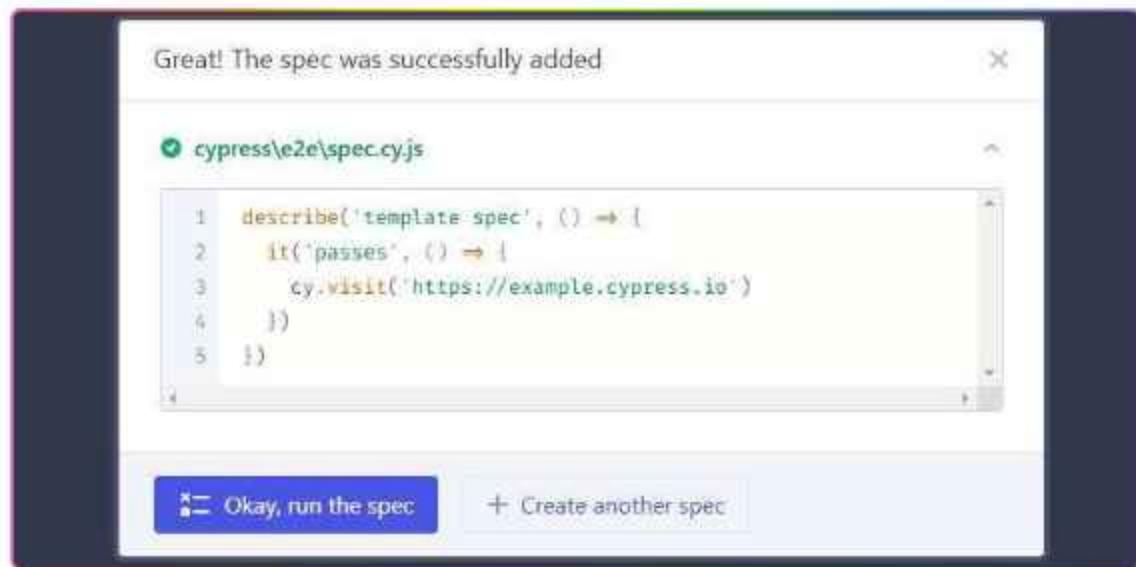
Na nova tela que aparece o Cypress está perguntando se começar com um exemplo de teste ou se vamos começar em um arquivo novo.
Selecione “Create new empty spec” para começar um arquivo do zero.



Na nova tela que se abre você poderá escolher o nome do arquivo.



Por último ele irá falar se você quer rodar um teste de exemplo. Basicamente este exemplo irá visitar uma URL do site de exemplo do Cypress, porém o mais legal de tudo é quando executamos o teste. Conseguimos um ter uma ideia de como funciona os testes no Cypress.



Você terá acesso ao código do teste na pasta cypress criado pelo próprio Cypress durante a configuração. Além desta pasta, o Cypress criou o arquivo cypress.config.js na raiz do nosso projeto.



```

DEPOERA
...
specify.js x
describe("template spec", () => {
  it("passes", () => {
    cy.visit("http://exemple.cypress.io");
  });
});

```

Já que citei nosso projeto, vamos ver de fato o que é ele antes de testar.

Basicamente nosso projeto é um formulário de cadastro onde o usuário irá digitar/selecionar algumas informações. É uma simples aplicação React utilizando o React Hook Form.

O que iremos fazer é adicionar essas informações utilizando o Cypress.



Aplicando testes no projeto

Primeira coisa que iremos fazer é utilizar o exemplo criado pelo Cypress e visitar a nossa aplicação (localhost). Porém, iremos utilizar o hook `beforeEach()` que será responsável por garantir que nossa aplicação seja carregada antes de iniciar os próximos testes que iremos escrever. Isso é feito se quisermos executar algumas etapas antes de cada caso de teste. Isso nos ajuda a obter estabilidade geral do teste.

Teremos o seguinte código utilizando o `beforeEach()`:

```
● ● ●  
1 describe("Cadastrando conta", () => {  
2   beforeEach(() => {  
3     cy.visit("http://localhost:3000/");  
4   });  
5 });
```

Dessa forma já conseguimos criar nosso próximo bloco de teste sem se preocupar com o carregamento da nossa aplicação. Os hooks `before` e `beforeEach` são muito utilizados quando estamos trabalhando com dados de uma API.

Antes de escrever o próximo bloco de teste vamos entender o que significa este `describe()`.

Basicamente o `describe()` serve para agrupar nossos casos de teste. O `describe()` recebe dois argumentos, o primeiro é o nome do grupo de teste e o segundo é uma função de retorno de chamada. Pode haver várias `describe()` no mesmo arquivo. Além disso, uma `describe()` pode ser declarada dentro de outra `describe()`.



```
1 describe("Descrevo o bloco de teste", () => {});
```

Em nosso próximo teste não iremos utilizar o `describe()` mas sim o `it()`.

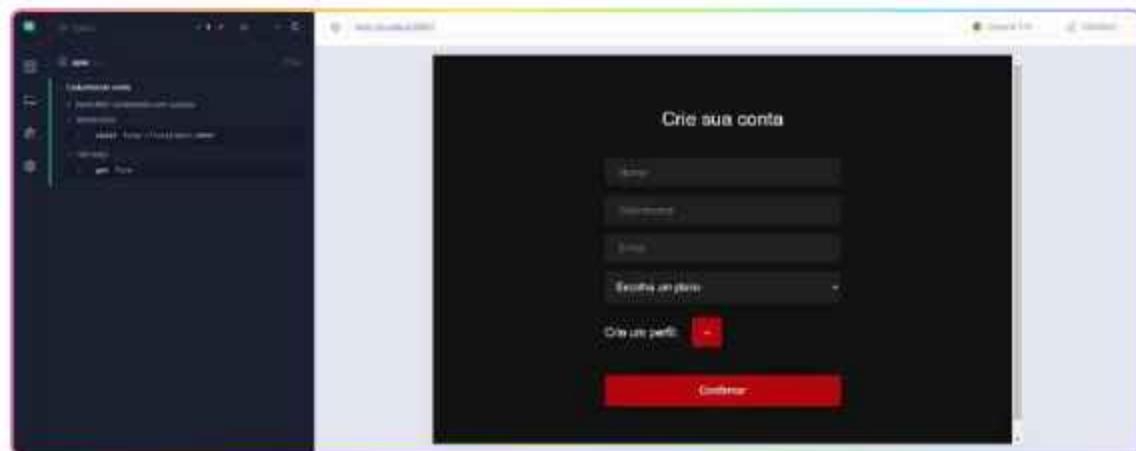
O `it()` é usado para um caso de teste individual. Ele recebe dois argumentos, uma string explicando o que o teste deve fazer e uma função de retorno de chamada que contém nosso teste real. Sua utilização é igual a do `describe()`, mas lembrando que o `describe` é utilizado para agrupar os testes e o `it` para teste individual.

O que iremos fazer agora é verificar se nosso formulário está sendo renderizado na nossa tela. Para isso iremos utilizar o `get()`.

```
1 describe("Cadastrando conta", () => {
2   beforeEach(() => {
3     cy.visit("http://localhost:3000/");
4   });
5
6   it("Formulário renderizado com sucesso", () => {
7     cy.get("form");
8   });
9 });
```

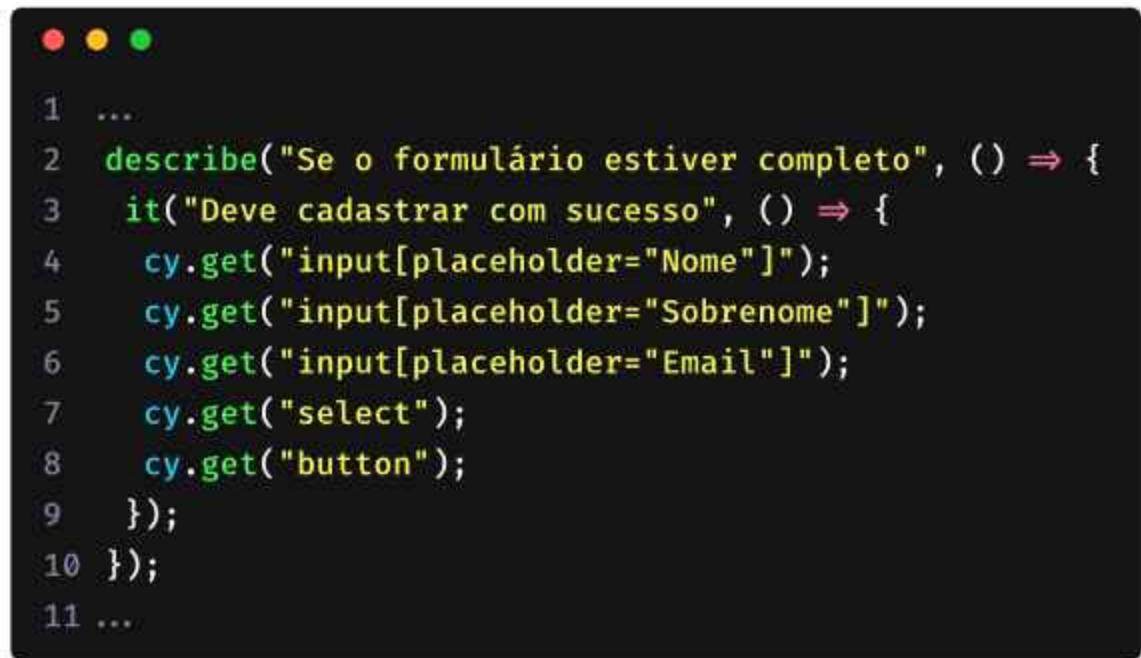
O `get()` é utilizado para obter um ou mais elementos em nossa DOM. Em nosso caso estamos verificando se temos o elemento "form".

Até o momento nossos testes estão assim:



Agora iremos para a melhor parte de utilizar o Cypress, interagir com a nossa aplicação utilizando os testes. Basicamente iremos preencher os campos e registrar o cadastro da conta utilizando os testes.

Vamos fazer isso por etapa, primeiro irei criar o teste utilizando os recursos já utilizado até aqui: describe, it e get.



```
1 ...
2 describe("Se o formulário estiver completo", () => {
3   it("Deve cadastrar com sucesso", () => {
4     cy.get("input[placeholder='Nome']");
5     cy.get("input[placeholder='Sobrenome']");
6     cy.get("input[placeholder='Email']");
7     cy.get("select");
8     cy.get("button");
9   });
10 });
11 ...
```

Se a gente rodar este teste não teremos nenhum erro, porém a única coisa que estamos fazendo é verificar se o elemento estão na tela.

Obs: para identificar cada campo de forma individual podemos chamar eles utilizando o placeholder.

Agora iremos adicionar as informações nos inputs da nossa aplicação. Para isso iremos utilizar o type() do Cypress para digitar em um elemento da DOM.

Abaixo temos um exemplo utilizando o type() no campo email, nome e sobrenome do formulário.

```
1 ...
2 cy.get("input[placeholder='Nome']")
3   .type("Iuri");
4 cy.get("input[placeholder='Sobrenome']")
5   .type("Silva");
6 cy.get("input[placeholder='Email']")
7   .type("iuricode@gmail.com");
8 ...
```

Obs: o type() é utilizado na mesma liga do get(), coloquei na linha de baixo pois iria transbordar do nosso bloco de código.

Agora iremos aplicar teste no elemento select e button.

No select iremos fazer com que o Cypress selecione a opção “Básico” e que ele procure um botão com o texto “Confirmar” e clique nele.

No select iremos utilizar o select() do Cypress que seleciona um elemento option de um elemento select. Sua utilização é bem simples:

```
1 ...
2 cy.get("select").select("Básico");
3 ...
```

Agora iremos fazer com que nosso teste clique no botão "Confirmar".

Para isso vamos utilizar dois recursos do Cypress: click() e contains().

- O contains() obtém o elemento que contém o texto.
- O click() simplesmente irá clicar em um elemento.

```
1 ...
2 cy.get("button").contains("Confirmar").click();
3 ...
```

A penúltima coisa que iremos fazer neste bloco de teste é adicionar um perfil para a conta cadastrada.

Teremos o seguinte código que fará a ação de adicionar um perfil.

```
1 ...
2 cy.get("#add-profile-btn").click();
3 cy.get("input[placeholder='Nome do perfil']")
4 .type("iuricode");
5 ...
```

A única novidade que temos neste código é o "#add-profile-btn". Ele está pegando o elemento que tem o ID "add-profile-btn". É igual a chamada de um ID no CSS.

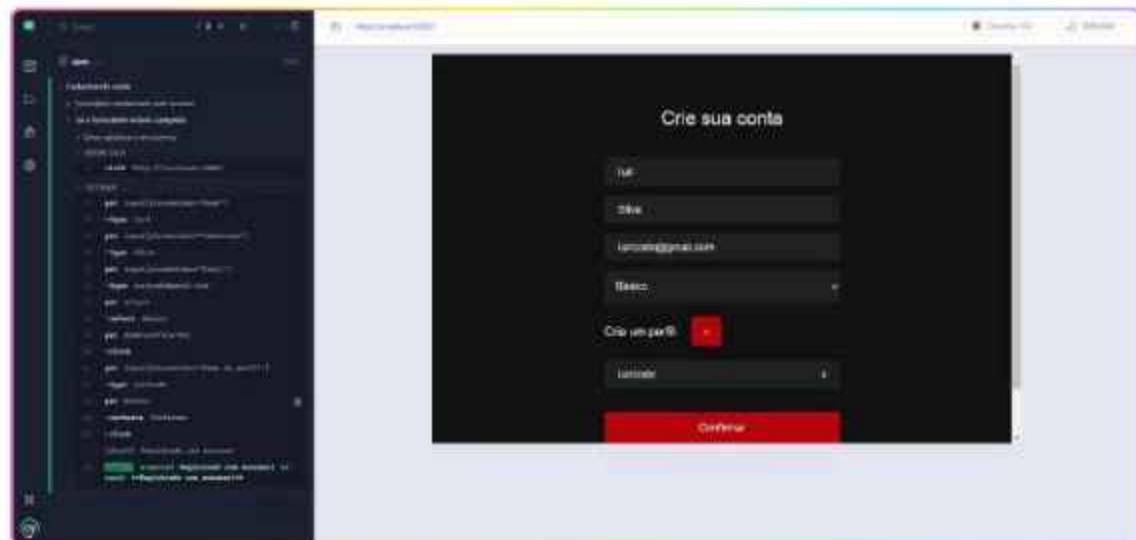
Agora por último, vamos adicionar um evento que dispara um alerta confirmando o cadastro da nossa conta.

```
1 ...
2 cy.on("window:alert", (str) => {
3   expect(str).to.equal("Registrado com sucesso!");
4 });
5 ...
```

O cy.on é responsável por capturar eventos, seja uma alerta, seja uma exceção, dentre outros. Essas capturas de eventos são essenciais para controlar o comportamento da aplicação e auxilia na depuração.

O cy.on irá receber como parâmetro, o tipo que é exibido no evento, se é uma string (str) ou um número (number), dentre outros. Já dentro da função do cy.on, é colocado a asserção esperada (expect), neste caso, que a mensagem do evento de alerta seja 'Registrado com sucesso!'. Caso seja uma mensagem diferente, o expect gerará um erro.

Dessa forma já criamos nosso primeiro teste com Cypress. Se tudo estiver certo teremos a seguinte mensagem no teste:



O código completo do nosso teste ficou assim:

```
● ● ●

1 describe("Cadastrando conta", () => {
2   beforeEach(() => {
3     cy.visit("http://localhost:3000/");
4   });
5
6   it("Formulário renderizado com sucesso", () => {
7     cy.get("form");
8   });
9
10  describe("Se o formulário estiver completo", () => {
11    it("Deve cadastrar com sucesso", () => {
12      cy.get("input[placeholder='Nome']").type("Iuri");
13      cy.get("input[placeholder='Sobrenome']")
14        .type("Silva");
15      cy.get("input[placeholder='Email']")
16        .type("iuricode@gmail.com");
17      cy.get("select").select("Básico");
18      cy.get("#add-profile-btn").click();
19      cy.get("input[placeholder='Nome do perfil']")
20        .type("iuricode");
21      cy.get("button").contains("Confirmar").click();
22    ...
23  });
24})
```

```
23    ...
24    cy.on("window:alert", (str) => {
25      expect(str).to.equal("Registrado com sucesso!");
26    });
27  });
28 });
29 });
```

Chegamos no final do módulo de Cypress!

Sei que foram abordado conceitos básicos sobre essa tecnologia de testes do front-end, mas sem dúvida após executar este passo a passo, você terá uma boa base de como utilizar Cypress em grandes projetos.

Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem se aprofundar na
aprendizagem de front-end, fazer deploy, implementar
corretamente de código e cultura ágil, além de outras paixões para
quem deseja se tornar um frontender.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a
criação de interfaces**. Além de muitos **exercícios**,
trabalhos, exercícios e também realizarem sua competência de UX.

[QUERO COMPRAR TAMBÉM](#)

Este material tem **problemas para entrar na área de
programação** que podem ser resolvidos em suas
máquinas. Pode usar, enviar para amigos e familiares. **ONLINE**,
permite a prática para quem está a campo das tecnologias.

[QUERO COMPRAR TAMBÉM](#)



Módulo 14

Next.js

Pronto para o café? Vamos ao módulo final! Lembrando que, ao final de todos os módulos, você pode fazer testes e decíduar se é hora de um questionário no Módulo.

+ 7 TÓPICOS NESTE MÓDULO

Introdução

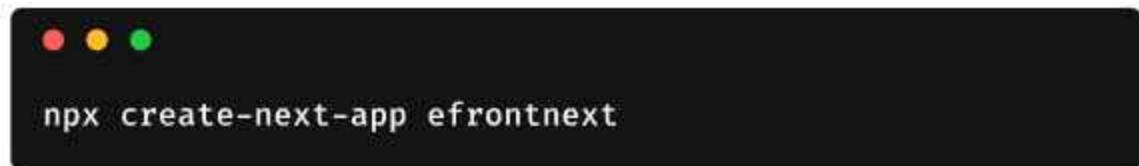
Neste módulo iremos falar do framework que inovou a forma de utilizar React, o Next.js! O Next.js permite que suas aplicações sejam renderizadas no lado do servidor (SSR), diminuindo o tempo de carregamento da aplicação, já que o esforço fica por conta do servidor, não do dispositivo do cliente, além de consumir menos recursos.

O que é

Next.js é um framework React com foco em produção e eficiência criado e mantido pela equipe da Vercel, o Next busca reunir diversas funcionalidades como renderização híbrida e estática de conteúdo, suporte a TypeScript, pre-fetching, sistema de rotas e entre outras que fazem o Next ser o queridinho da comunidade frontend.

Iniciando

A inicialização do Next é semelhante ao React. Dessa forma, é simples lembrar o processo de criar uma aplicação feita com Next.

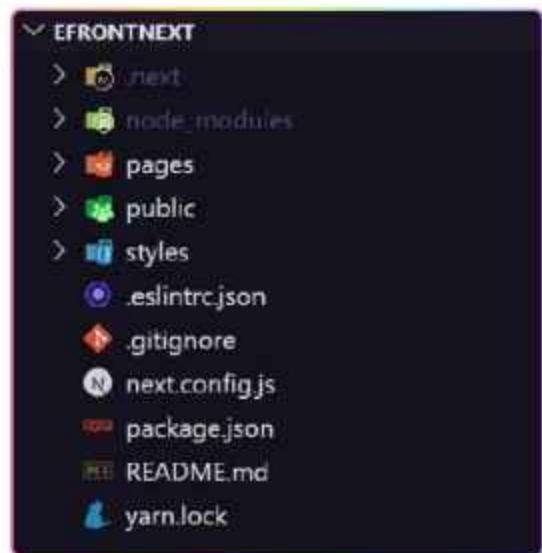
A dark-themed terminal window icon with three colored dots (red, yellow, green) at the top left.

```
npx create-next-app efrontnext
```

Veja que a única diferença está no “next” que no caso dos projetos feitos em React é escrito “react” no lugar.

Estrutura

Após rodar nosso comando e o Node criar nossa aplicação, teremos a seguinte estrutura de pastas e arquivos:



Diferente da estrutura do React, o Next gerou também as pastas `styles`, `pages`, `.next` e o arquivo `next.config.js`.

- Na pasta styles contém os arquivos de estilos.
- Na pasta pages você encontra arquivos que representam as rotas e páginas da sua aplicação (*já iremos falar sobre*).
- Na pasta .next é o local que nosso projeto é gerado.
- O arquivo next.config.js é onde ficará as configurações personalizadas do Next.js. Isso é, algumas tecnologias como o TypeScript precisam ser configurados para que funcionem no Next.

Agora que você entendeu a estrutura gerada, vamos rodar nossa aplicação feita pelo Next. Para isso basta dar o seguinte comando:

```
● ● ●  
npm run dev
```



Agora vamos verificar se a aplicação está sendo renderizada do lado do servidor (SSR).

"Okay Iuri, mas para que queremos que a nossa aplicação seja renderizada do lado do servidor?"

Quando criamos uma aplicação usando o Next, as páginas do site são renderizadas no lado do servidor. Isso faz com que o HTML seja entregue para o navegador.

E isso tem três grandes benefícios:

- O client não precisa instanciar o React para renderizar, o que torna o site mais rápido para os seus usuários.
- Os motores de busca indexarão as páginas sem a necessidade de executar o JavaScript do lado do cliente.
- Você pode ter metatags, adicionar imagens de visualização, personalizar o título e a descrição para qualquer uma de suas páginas compartilhadas no Facebook, Twitter, LinkedIn...

Quando acessamos um site feita pelo React a renderização é diferente.

Como toda renderização é feita pelo lado do cliente, para que o conteúdo da página seja “mostrado” precisamos acessar e esperar a renderização. Isso acaba dificultando a indexação dos navegadores, pois não é possível recomendar o conteúdo se ele não “existe” naquele momento.

Isso é diferente com o Next, pois todo o conteúdo (HTML) já está sendo apresentado no código-fonte da nossa página. Não é preciso esperar o JavaScript mostrar o conteúdo na página.

Mas não acredite em mim, vamos ver isso na prática!

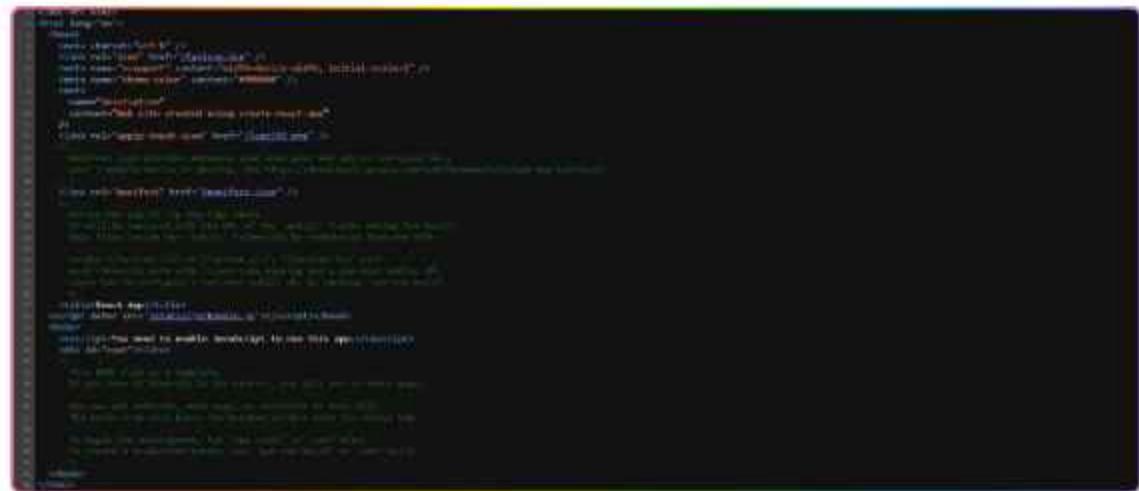
Acesse o localhost da nossa aplicação feita com Next. Após acessa, aperte as teclas Ctrl + U para acessar o código-fonte.



```
http://127.0.0.1:3000/
meta: <meta name="viewport" content="width=device-width, initial-scale=1" />
title: <title>Learn Next.js</title>
html: <html><head><script>document.write('Hello from a static Next.js page!')</script></head><body><h1>Learn Next.js</h1><p>This page was generated by the Next.js API Router. If you made it here, you're probably looking for the <a href="#">nextjs.org/router documentation. You can also check out the <a href="#">nextjs.org/docs documentation or <a href="#">nextjs.org/examples examples. If you're interested in learning about Next.js in an interactive course with quizzes, check out the <a href="#">nextjs.org/quizzes.</p></body></html>
```

Perceba que todo o texto (conteúdo) da nossa aplicação está sendo mostrada no código da página.

Agora irei fazer este mesmo processo em uma aplicação React.



The screenshot shows the Network tab of a browser's developer tools. A single request is listed for 'bundle.js' with a size of approximately 1.3 MB. The status is 'Success' and the response code is 200. The URL is 'http://localhost:3001/bundle.js'. The file type is 'application/javascript'. The content type is 'text/html; charset=UTF-8'. The last modified date is '2023-06-15T14:39:28.380Z'. The file was created on '2023-06-15T14:39:28.380Z' and updated on '2023-06-15T14:39:28.380Z'. The file is compressed with 'gzip'.

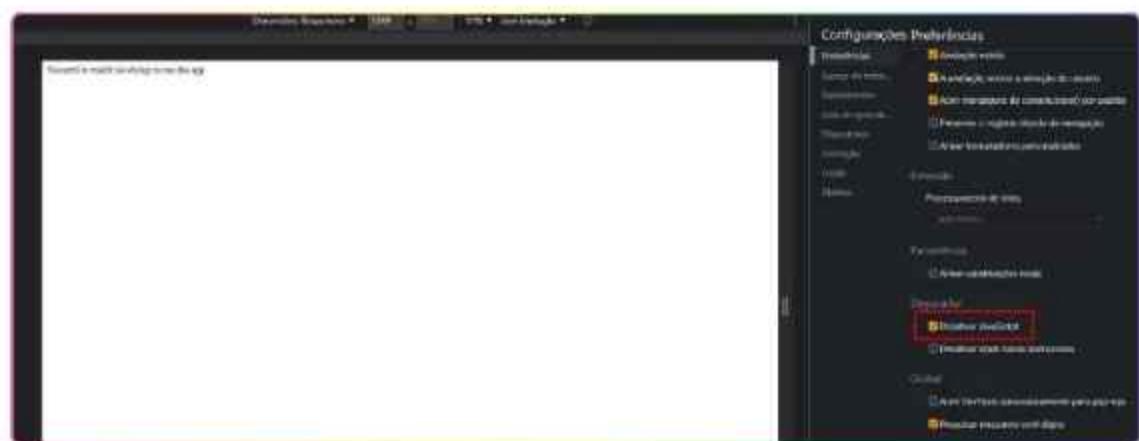
```

Request URL: http://localhost:3001/bundle.js
Request Method: GET
Status Code: 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1.3M
Last-Modified: Mon, 15 Jun 2023 14:39:28 GMT
Content-Encoding: gzip
Content-Type-Options: nosniff
Date: Mon, 15 Jun 2023 14:39:28 GMT
Etag: "1686880368380"
Last-Modified: Mon, 15 Jun 2023 14:39:28.380Z
X-Powered-By: Express
```

```

Na imagem acima todo nosso conteúdo está vindo do JavaScript através do ID "root". Dessa forma o navegador não consegue indexar o site.

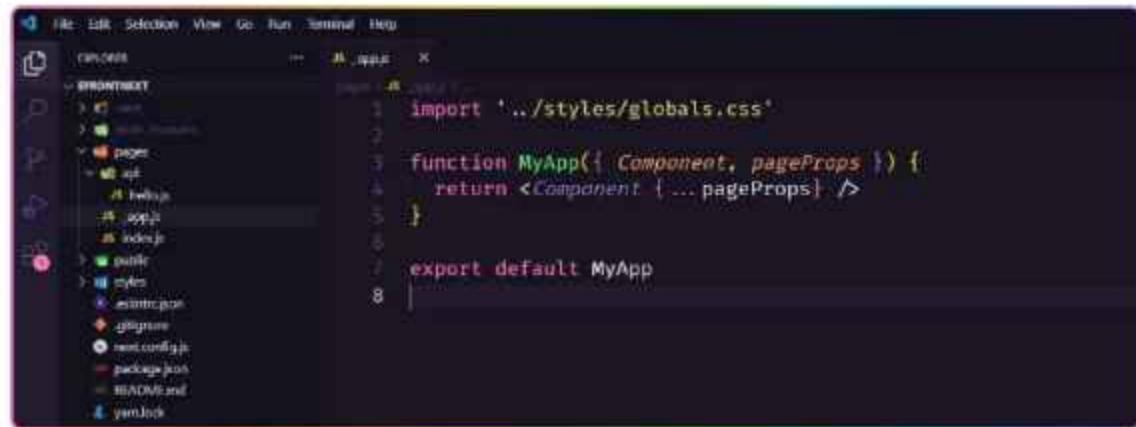
Para finalizar este assunto vamos fazer mais um teste!



Como pode ver, quando desabilitamos o JavaScript nossa aplicação React morre, pois todo conteúdo é criado pelo próprio JavaScript.

Para finalizar este tópico vamos abordar os arquivos existentes e um super importante que sempre iremos utilizar em nossas aplicações.

Na pasta pages temos o arquivo `_app.js`. O Next.js usa este arquivo para inicializar as páginas.



The screenshot shows a code editor with the file `_app.js` open. The file contains the following code:

```

import '../styles/globals.css'

function MyApp({ Component, pageProps }) {
 return <Component {...pageProps} />
}

export default MyApp

```

The code editor interface includes a sidebar showing the project structure, which includes `components`, `public`, `pages`, and other Next.js specific files like `next.config.js` and `package.json`.

A prop “Component” é a visualização da página que será renderizada. Podemos usar isso para passar nossas próprias props ou agrupar com componentes de layout/contexto. A prop “pageProps” são os props que cada página receberá quando renderizada.

Dentro da pasta pages temos a pasta api que contém o arquivo hello.js

```

File: /Users/joselopes/Documentos/Next.js/nextjs-api-hello/api/hello.js
1 // Next.js API route handler: https://nextjs.org/docs/api-routes/introduction
2
3 export default function handler(req, res) {
4 res.status(200).json({ name: 'John Doe' })
5 }
6

```

Esta rota fornece uma solução para criar sua API com Next.js, mas fique tranquilo que teremos um tópico para abordar este tema.

Por últimos vamos falar sobre o arquivo `_document.js`. Este arquivo não existe em nossa aplicação mas ela pode ser criada no mesmo diretório do arquivo `_app.js`.

```

File: /Users/joselopes/Documentos/Next.js/nextjs-api-hello/pages/_document.js
1 import { Html, Head, Main, NextScript } from "next/document";
2
3 export default function Document() {
4 return (
5 <Html>
6 <Head>
7 <body>
8 <Main/>
9 <NextScript/>
10 </body>
11 </Html>
12);
13}
14

```

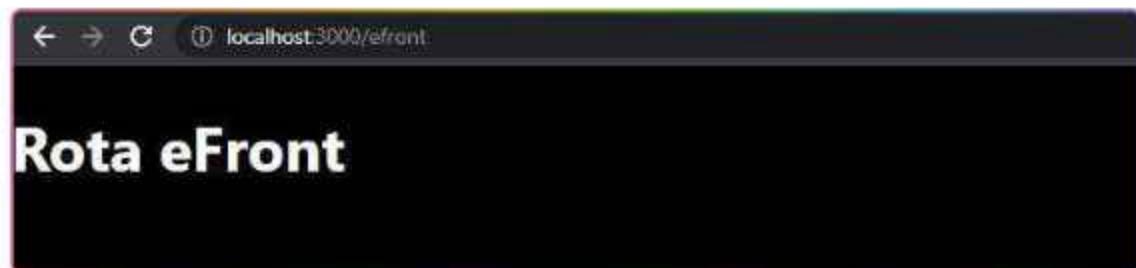
O arquivo `_document` é utilizado para definir o idioma, carregar fontes, carregar scripts antes da interatividade da página, coletar arquivos CSS como styled-components, entre algumas outras coisas.

## Rotas

Há muitos benefícios em usar o Next.js, mas um recurso muito útil é o sistema de roteamento de arquivos. Essa arquitetura simplificou o processo de criação de rotas.

Quando um arquivo é adicionado a pasta `pages`, automaticamente uma rota com o nome deste arquivo é gerada. Mas caso seja adicionado um arquivo CSS dentro de `pages` o Next.js irá apresentar um erro.

Veja no exemplo abaixo que o arquivo `efront.js` dentro da pasta `pages` fará com que a rota `http://localhost:3000/efront` chame este arquivo.



Dessa forma tudo que for componente, estilos CSS e entre outros, devem estar fora da pasta pages. Para não ficar criando várias pastas na raiz do projeto, podemos criar a pasta “src” e adicionar as rotas, styles e componentes dentro dele, pois o Next suporta/entende esta pasta.

“Mas se eu acessa uma rota que não existe em nossa aplicação?”

Você será direcionado a esta página:



Mas fique tranquilo que o Next criou uma solução para personalizar as rotas de erros de uma forma bem simples e fácil.

Para criar uma página 404 personalizada, você pode criar um arquivo 404.js. Este arquivo é gerado estaticamente no momento da compilação.



```
1 export default function Custom404() {
2 return <h1>Página não encontrada</h1>;
3 }
```

Também podemos criar para a rota caso aconteça um erro no servidor. Para isso basta criar o arquivo 500 e mudar o nome da função.

Para finalizar este tópico vamos falar sobre rotas dinâmicas no Next.

Rotas dinâmicas no Next.js são rotas que não definimos de forma estática. Um caso muito utilizado é ter uma lista de posts e quando o usuário clicar em um desses posts, nos direciona para essa rota, onde nessa rota possui o id e então fazemos uma requisição a alguma API trazendo os dados desse post.

Diferente da rota estática que já aprendemos onde o usuário deve acessar aquela exata rota com aquele exato nome para cair na página desejada em nossa aplicação.

As rotas dinâmicas são utilizadas quando necessitamos receber algum parâmetro através da URL.

Para definir rotas dinâmicas usamos colchetes nos nomes dos arquivos que queremos como rotas. Exemplo: [id].js. Esse id é o id do parâmetro que você irá receber na rota.



```
posts.js
1 export default function Posts() {
2 return <h1>Rota dinâmica</h1>;
3 }
```

Como podemos ver logo acima, temos um arquivo que contém o nome [id] na qual id será o parâmetro que vamos receber dessa rota.

Ao passar qualquer coisa na URL iremos entrar nesta rota dinâmica.

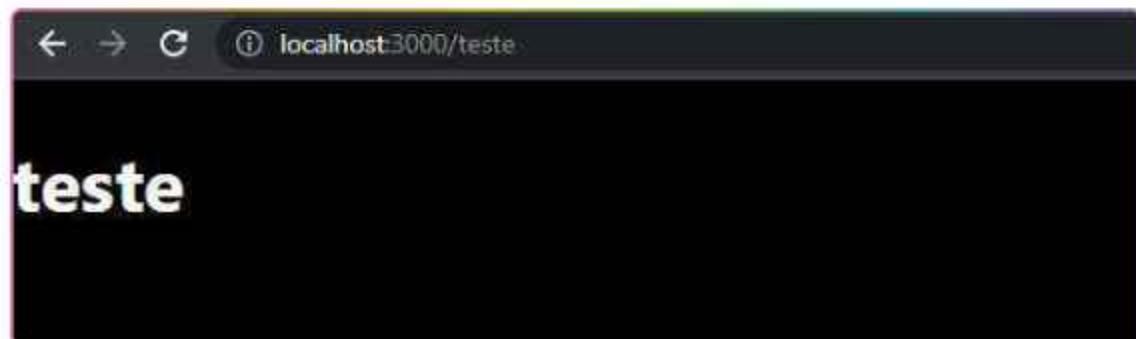


Para capturar o valor dessa rota acessada pelo usuário, podemos usar o hook `useParams`, que nos fornece uma propriedade chamada `query` que é onde ficará armazenada uma propriedade com o nome que escolhemos para o arquivo (que nesse caso foi `id`), e é nessa propriedade que podemos ter acesso a rota que o usuário acessou.



```
1 import { useRouter } from "next/router";
2
3 export default function Posts() {
4 const router = useRouter();
5 const parametro = router.query.id;
6
7 return <h1>{parametro}</h1>;
8 }
```

Então, agora se acessarmos qualquer rota, será mostrado pra gente na tela o que foi digitado na URL:



## API Reference

Como você viu no tópico anterior usamos um recurso do Next, o useRouter. Agora neste tópico iremos abordar os recursos mais utilizados pela comunidade.

O primeiro não entrando muito afundo pois já usamos ele, é o useRouter.

```
1 import { useRouter } from "next/router";
2
3 export default function Posts() {
4 const router = useRouter();
5 const parametro = router.query.id;
6
7 return <h1>{parametro}</h1>;
8 }
```

Utilizamos ele para acessar o objeto router dentro de qualquer componente.

Lembrando que Hooks não podem serem usados com classes.

O próximo que iremos abordar é o Link. Como você já pode imaginar o componente Link é utilizado para as transições do lado do cliente entre as rotas. Para isso exportado por next/link.

```
 1 import Link from "next/link";
 2
 3 function Home() {
 4 return (
 5
 6
 7 <Link href="/">Home</Link>
 8
 9
10 <Link href="/blog">Blog</Link>
11
12
13)
14 }
15
16 export default Home
```

O componente Link aceita vários props, o mais utilizado é o href. O href é o caminho para onde você seja navegar ao acessar o Link.

O próximo componente é o Head, ele é utilizado para anexar a tag head do HTML na página.

```
1 import Head from "next/head";
2
3 function IndexPage() {
4 return (
5 <div>
6 <Head>
7 <title>Título da página</title>
8 </Head>
9 <p>eFront</p>
10 </div>
11)
12 }
13
14 export default IndexPage
```

O title, meta ou quaisquer outros elementos (por exemplo script) precisam estar dentro do componente Head do Next.

O componente Image é uma solução moderna para utilizar imagens em aplicações com Next. É semelhante ao elemento <img/> HTML, mas tem uma grande diferença.

A principal diferença entre os dois é a otimização de imagem pronta para uso, isso ajuda demais no desempenho na nossa aplicação.

```
1 import Image from "next/image";
2 import profile from "../public/profile.png";
3
4 function Profile() {
5 return (
6 <Image
7 src={profile} alt="Foto usuário"
8 width={300}
9 height={300}
10 />
11)
12 }
13 export default Profile
```

O componente Image requer três tipos de propriedades em seu uso. Os props src, width e height.

As props src aceitam dois tipos de valores: um objeto de imagem local importado estaticamente ou uma string de caminho para uma URL de imagem externa. No exemplo anterior, importamos uma imagem estática da pasta public.

Para URL externas é necessária uma configuração para `next.config.js` fornecendo uma lista de nomes de host permitidos.

Os props `width` e `height` basicamente determinam quanto espaço uma imagem ocupa em uma página ou como ela é dimensionada em relação ao seu container. As props `width` e `height` podem representar a largura renderizada ou original da imagem, dependendo do valor de `layout`.

## Rotas API

Agora vamos entender melhor o que são Rotas API no Next.js.

Se você abrir a pasta `pages` no diretório raiz do projeto, deverá ver uma pasta `api` nela. A pasta `api` contém um exemplo de rota API nomeada `hello.js` com o seguinte conteúdo:

```
1 export default function handler(req, res) {
2 res.status(200).json({ name: 'John Doe' })
3 }
```

Dessa forma, qualquer arquivo dentro da pasta `pages/api` é mapeado e tratado como um endpoint de API.

Para que uma rota de API funcione, você precisa exportar uma função como padrão (request handler ) que então recebe os seguintes parâmetros:

- req: uma instância de http.IncomingMessage que também inclui alguns auxiliares integrados - ou middlewares - como req.body, req.query e entre outros, para analisar a solicitação recebida.
- res: uma instância de http.ServerResponse que inclui alguns métodos auxiliares do tipo Express.js, como res.send, res.body e entre outros.

Obs: da mesma forma que é criado uma rota dinâmica na pasta pages, é a forma para criar uma rota API dinâmica.

## Data Fetching

O Data Fetching no Next permite que você renderize seu conteúdo de maneiras diferentes, dependendo do caso de uso da sua aplicação. Isso inclui pré-renderização com renderização do lado do servidor (SSR) ou geração estática (SSG), e atualização ou criação de conteúdo em tempo de execução com regeneração estática incremental (ISR) .

O método `getStaticProps` pode ser usado dentro de uma página para buscar dados em tempo de construção, por exemplo, quando você executa `next build`. Depois que a aplicação for criado, ele não atualizará os dados até que outra compilação seja executada.

```
1 export async function getStaticProps(context) {
2 const res = await fetch(`https://.../data`)
3 const data = await res.json()
4
5 if (!data) {
6 return {
7 notFound: true,
8 }
9 }
10
11 return {
12 props: {},
13 }
14 }
```

Ele permite que a página seja gerada estaticamente e produzirá tempos de carregamento rápidos de todos os métodos de busca de dados.

O método `getServerSideProps` busca dados sempre que um usuário solicita a página. Ele buscará os dados antes de enviar a página ao cliente (ao contrário de carregar a página e buscar os dados no lado do cliente). Se o cliente fizer uma solicitação, os dados serão buscados novamente.

```
1 export async function getServerSideProps(context) {
2 const res = await fetch(`https://.../data`)
3 const data = await res.json()
4
5 if (!data) {
6 return {
7 notFound: true,
8 }
9 }
10
11 return {
12 props: {},
13 }
14 }
```

Os dados são atualizados cada vez que um cliente carrega a página, o que significa que estão atualizados a partir do momento em que visitam.

Chegamos ao fim no nosso módulo de Next.js. Aqui foram citados alguns conceito/recursos básicos no Next, apenas para te dar um caminho do que estudar quando for preciso utilizar ele amado framework do React.

# Adquira meus outros materiais com 10% de desconto!

UTILIZE O CUPOM: **FRONTIO**

Este material é destinado a pessoas que querem se aprofundar na aprendizagem de front-end, fazer deploy, implementar corretamente de código e cultura ágil, além de outras palestras que desejam se tornar mais profissionais.

[QUERO COMPRAR TAMBÉM](#)

Adquira este material e obterá um **guião de boas práticas para a criação de interfaces**. Além de conteúdos sobre design thinking, você também receberá sua compreensão de UX.

[QUERO COMPRAR TAMBÉM](#)

Desvendando os principais problemas para entrar na área de programação. Neste guia, ensino como iniciar seu caminho em suas primeiras horas, envolvendo conceitos básicos de programação, portfólio e outras dicas úteis para quem está a começar o seu caminho.

[QUERO COMPRAR TAMBÉM](#)

# Bônus

Receba café premium, biscoitos e muito mais! Lembrando que ao final de todos módulos, você pode tirar suas comodidades por meio de um questionário no Módulo.

6 TEMAS NESTE MÓDULO

## Introdução

Neste último módulo iremos abordar as novas tecnologias que estão surgindo no mundo frontend, mas não de uma forma aprofundada como nos módulos anteriores, o objetivo é apenas dar uma introdução para vocês saberem o que estudar depois de concluir todos os módulos.

### Vite

Surgiu uma ferramenta muito incrível na criação de projetos React, chamada [Vite](#). Basicamente, o Vite é um boilerplate (um padrão que você pode utilizar), assim como o `create-react-app` (CRA), que usamos para criar nossas aplicações React. No entanto, o Vite traz algumas vantagens e diferenças em comparação com o CRA. Será que a utilização do Vite no React é melhor do que o CRA? Vamos entender um pouco mais sobre esta ferramenta.

`Create-React-App` é uma ferramenta de linha de comando criada pelo Facebook que permite gerar um novo projeto React, e usa o `webpack` pré-configurado para desenvolvimento.

Há muito tempo ele se tornou uma parte integrante do ecossistema React, que elimina a necessidade de manter pipelines de construção complexos em seu projeto, permitindo que você se concentre na criação e não em configuração.



```
npx create-react-app efront
```

Recentemente, o Create-React-App deixou de ser a escolha padrão para configurar projetos React, de acordo com a documentação oficial, que agora recomenda outras opções como Next.js, Remix, Gatsby e Expo (para aplicativos nativos).

No entanto, o Vite tem sido apontado como o sucessor do CRA há algum tempo. É um construtor de projetos mais rápido e mais leve, tornando-se a nova escolha padrão para projetos React.

## Diferenças

Vamos dar uma olhada em algumas das principais diferenças entre essas duas ferramentas.

- Mais rápido e mais leve do que o Create-React-App- Usa o esbuild para compilação, que é significativamente mais rápido do que o webpack.
- Permite que os desenvolvedores construam e recarreguem projetos mais rapidamente.
- Projetado para trabalhar com outras tecnologias modernas, como o TypeScript e o Vue.js (Vite e Vue.js têm o mesmo criador, Evan You).
- Possui um ambiente de desenvolvimento mais moderno e atualizado.
- Permite importações dinâmicas com melhor suporte a tipos para otimização de carregamento de módulos.
- Permite que você escolha como deseja configurar seu projeto, dando aos desenvolvedores mais flexibilidade e controle.

Em resumo, o Vite se tornou a escolha padrão para projetos React. Ele é mais rápido, mais leve e mais flexível para trabalhar com outras tecnologias modernas.

## React Query

No módulo de JavaScript abordamos o Axios, que é uma forma de consumir dados da API, assim como o fetch que já é algo nativo da linguagem JavaScript para consumo de dados. Porém a melhor forma de consumir APIs em aplicações React é o React Query.

O [React Query](#) é uma das "ferramentas do momento" no ecossistema React que lida com data fetching de forma simples e prática. Ele permite fazer o fetching de dados, adicionar dados no cache, gerenciar invalidação de cache, lógicas de retry entre outras funcionalidades úteis de forma fácil, sem necessidade de lidar com lógicas complexas ou "tocar no estado global".

Se você ainda não consegue ver a vantagem de todos os itens citados, deixe-me dar um exemplo onde trabalharemos com uma lista de Pokémons.

## Instalação de Config

Em primeiro lugar, vamos instalar a React Query dentro de um projeto React:

```
yarn add react-query
```

O React Query precisa de uma configuração básica para funcionar. Para isso, você deve criar um QueryClient e envolvê-lo com o QueryClientProvider no nível mais alto da sua aplicação. Isso normalmente é feito no arquivo index.js ou App.js. Aqui está um exemplo:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {
4 QueryClient, QueryClientProvider
5 } from 'react-query';
6 import App from './App';
7
8 const queryClient = new QueryClient();
9
10 ReactDOM.render(
11 <React.StrictMode>
12 <QueryClientProvider client={queryClient}>
13 <App />
14 </QueryClientProvider>
15 </React.StrictMode>,
16 document.getElementById('root')
17);
```

Agora já podemos fazer a busca de dados.

## Buscando dados

O Hook useQuery é maneira principal de buscar dados no React Query.

```
 1 import { useQuery } from 'react-query';
 2 import axios from 'axios';
 3
 4 const fetchAllPokemon = async () => {
 5 const response = await axios.get(
 6 'https://pokeapi.co/api/v2/pokemon/'
 7);
 8 return response.data.results;
 9 }
10
11 function PokemonList() {
12 const {
13 data, isLoading, error
14 } = useQuery('pokemonList', fetchAllPokemon);
15 }
```

## Carregamento e erro

```
1 if (isLoading) return <p>Carregando ... </p>
2
3 if (error) return <p>Ocorreu um erro!</p>
```

- data: os dados retornados pela consulta, se a consulta for bem-sucedida.
- isLoading: um indicador booleano que indica se a consulta está atualmente sendo realizada.
- error: um objeto de erro, caso ocorra algum erro durante a consulta.

Para fazer a renderização utilizei o método map que tem a função de percorrer os dados retornados peça consulta.

Cada item de dados na matriz data é representado por pokemon, e para cada um desses itens, um elemento li é retornado com o nome do Pokémon (pokemon.name).

```
● ● ●
1 return (
2
3 {data.map((pokemon) => (
4 <li key={pokemon.name}>{pokemon.name}
5))}
6
7);
```

"Okay Iuri, mas o exemplo você está utilizando o Axios para consumir a nossa API".

O React Query e o Axios (e Fetch) são ferramentas diferentes com finalidades diferentes, e eles não substituem um ao outro. Ambos têm papéis distintos em uma aplicação React.

Geralmente, em uma aplicação React que faz solicitações HTTP para buscar dados de uma API externa, você usará o Axios para realizar essas solicitações. Em seguida, você pode usar o React Query para gerenciar o estado desses dados. Isso significa que eles são complementares, não substitutos.

## Conclusão

O React Query é uma das melhores maneiras de buscar, armazenar em cache e atualizar dados remotos. Precisamos apenas dizer à biblioteca onde você precisa buscar os dados, e ela tratará do cache, das atualizações em segundo plano e da atualização dos dados sem nenhum código ou configuração extra.

Ele também fornece alguns hooks e eventos para mutação e query para lidar com erros e outros estados dos efeitos colaterais, o que remove a necessidade de usar hooks como useState e useEffect e os substitui por algumas linhas com React Query.

## Node

No módulo de React, utilizamos o Node como base para a construção da aplicação, mas muitas vezes não exploramos detalhadamente o papel do Node neste contexto, e é exatamente que isso que faremos neste tópico. Abordaremos sobre o gerenciamento de pacotes.

O [Node.js](#) é muito mais do que apenas um ambiente de execução para JavaScript. Trata-se de um ambiente altamente eficiente, conhecido por sua capacidade de executar códigos no lado do servidor. Sua arquitetura orientada a eventos e assíncrona é crucial para lidar com múltiplas conexões simultâneas, tornando-o uma escolha ideal para aplicações web em tempo real, desenvolvimento de APIs e uma gama diversificada de outros casos de uso.

O diferencial do Node.js está na utilização do JavaScript tanto no front-end quanto no back-end, permitindo aos desenvolvedores trabalhar de forma mais coesa em todos os aspectos de um projeto. Essa unificação da linguagem simplifica o desenvolvimento, promovendo uma transição mais suave de códigos entre o cliente e o servidor.

## NPM

O NPM, sigla para Node Package Manager, é o gerenciador de pacotes padrão do Node.js. Em janeiro de 2017, o registro do npm listava mais de 350.000 pacotes, tornando-se o maior repositório de código de uma única linguagem no mundo. Este ecossistema diversificado garante que praticamente exista um pacote para quase todas as necessidades.

Inicialmente concebido para realizar o download e gerenciar dependências de pacotes no ambiente Node.js, o NPM evoluiu para se tornar uma ferramenta amplamente adotada no desenvolvimento.

As funcionalidades do NPM são variadas e abrangentes, indo muito além do simples gerenciamento de dependências.

### Instalando todas dependências

Ao criar a nossa aplicação React foi adicionado o arquivo package.json, ao rodar o comando:

A screenshot of a Mac OS X terminal window. The window has a dark background with three colored window control buttons (red, yellow, green) at the top left. The main area of the terminal shows the command 'npm install' typed in white text.

```
npm install
```

Ele vai instalar tudo que o projeto precisa, na pasta node\_modules.

## Instalando um único pacote

Instalar um pacote específico é simples. Basta executar o comando:

```
npm install <nome-do-pacote>
```

Normalmente, você verá mais opções adicionadas a esse comando:

- save: instala o pacote e adiciona uma entrada no campo dependencies do arquivo package.json.
- save-dev: instala o pacote e adiciona uma entrada no campo devDependencies do arquivo package.json.

A principal diferença entre eles é que os pacotes listados em devDependencies são ferramentas de desenvolvimento, como bibliotecas de testes, enquanto os pacotes em dependencies são os necessários para a aplicação em um ambiente de produção.

Quanto à atualização de pacotes, é igualmente simples.



```
npm update
```

O NPM busca em todos os pacotes por uma versão atualizada que satisfaça as restrições de versionamento especificadas.

Você também pode atualizar um único pacote específico utilizando:



```
npm update <nome-do-pacote>
```

Esses comandos básicos facilitam tanto a instalação quanto a atualização de pacotes no seu projeto, mantendo as dependências atualizadas e ajustadas às necessidades do ambiente de desenvolvimento e produção.

## Executando tarefas

Dentro do arquivo package.json, há um campo essencial chamado "scripts", destinado a especificar tarefas de linha de comando que podem ser executadas por meio do comando:

```
npm run <nome-da-tarefa>
```

Por exemplo:

```
{
 "scripts": {
 "start-dev": "node lib/server-development",
 "start": "node lib/server-production"
 }
}
```

Ao definir essas tarefas, como "start-dev" e "start" no exemplo, é possível executar comandos complexos ou de longa digitação com apenas um atalho de fácil memorização.

Assim, em vez de digitar comandos extensos e propensos a erros ou esquecimentos, é possível simplificar a execução utilizando:

```
npm run watch
npm run dev
npm run prod
```

Isso permite acionar prontamente tarefas como "watch", "dev" ou "prod" sem a necessidade de se lembrar dos comandos específicos ou comandos mais complexos, otimizando a gestão e execução de diferentes processos do projeto.

## Onde o NPM instala os pacotes?

Quando se instala um pacote usando o NPM, existem duas maneiras de fazê-lo:

1. Instalação Local: Ao digitar o comando `npm install`, por exemplo: `npm install react-query`, o pacote será instalado na estrutura de arquivos do diretório atual, dentro de uma subpasta denominada `node_modules`. Quando isso ocorre, o NPM adiciona uma entrada referente ao pacote `react-query` na propriedade "dependencies" do arquivo `package.json` presente na pasta onde o comando foi executado.

2. Instalação Global: Para uma instalação global, utiliza-se a flag `-g`, como em `npm install -g react-query` dessa forma você encontrará o pacote na propriedade “`devDependencies`” do arquivo `package.json`. Nesse caso, o npm não instala o pacote no diretório local, mas sim em uma localização global, que pode ser identificada usando o comando `npm root -g`.

- No macOS ou Linux, a localização geralmente é encontrada em `/usr/local/lib/node_modules`.
- No Windows, é comumente em C:  
`\Users\YOU\AppData\Roaming\npm\node_modules`.

## **package-lock.json**

Na versão 5, o NPM introduziu o arquivo `package-lock.json`, que descreve detalhadamente as dependências de um projeto. Seu propósito principal é manter um registro preciso das versões exatas de cada pacote instalado. Isso garante a total reprodução do ambiente, mesmo se os mantenedores dos pacotes fizerem atualizações.

Enquanto o arquivo `package.json` é comum e amplamente conhecido, ele permite a definição de faixas de versões para atualizações, usando a notação semver, por exemplo:

- Utilizando `~0.13.0`, você expressa o desejo de aceitar apenas atualizações de patch: `0.13.1` está dentro da margem, mas `0.14.0` não está.
- Com `^0.13.0`, você busca atualizar tanto para versões de patch quanto de minor: `0.13.1`, `0.14.0` e assim por diante.
- Se você especifica `0.13.0` exatamente, essa será a versão utilizada, sem variação.

Contudo, o problema reside na inconsistência gerada pelo `package.json`. Quando um projeto é replicado em uma nova máquina via `npm install`, se a sintaxe de versão utilizada permite atualizações de patch ou minor e uma nova versão é lançada, as versões instaladas podem ser diferentes, impactando a consistência do projeto.

Para resolver essa questão, o `package-lock.json` entra em cena. Este arquivo grava precisamente as versões instaladas de cada pacote, assegurando que o ambiente seja reproduzido de forma idêntica em diferentes instâncias do projeto, independentemente de atualizações feitas por terceiros. Assim, mesmo que atualizações de patch ou minor sejam lançadas, o projeto permanece estável e consistente.

O arquivo package-lock.json é crucial para manter a consistência do projeto e precisa ser commitado no repositório Git. Isso permite que outros colaboradores, caso o projeto seja público, ou para operações de deploy a partir do repositório Git, tenham acesso às versões exatas das dependências.

Quando você executa o comando npm update, as versões das dependências são atualizadas no arquivo package-lock.json.

Por fim, um exemplo ilustrativo seria a estrutura de um arquivo package-lock.json gerado ao executar npm install:

A screenshot of a terminal window on a Mac OS X system, indicated by the red, yellow, and green window control buttons at the top left. The terminal itself is black with white text. It displays the start of a JSON object, specifically a package-lock.json file. The visible code includes the opening brace '{', followed by several key-value pairs: 'name': 'naped', 'version': '0.1.0', 'lockfileVersion': 2, 'requires': true, and 'packages': { followed by three ellipsis characters '...'.

```
{
 "name": "naped",
 "version": "0.1.0",
 "lockfileVersion": 2,
 "requires": true,
 "packages": {
 ...
 }
}
```

```
node_modules/@ampproject/remapping": {
 "version": "2.2.0",
 "resolved": "https://.../",
 "integrity": "sha512-RMP1eZuhxH5Jc26w==",
 "peer": true,
 "dependencies": {
 ...
 },
 "engines": {
 "node": "≥6.0.0"
 }
},
...
}
```

## package.json

O package.json atua como um tipo de manifesto do seu projeto. Ele desempenha diversas funções, muitas vezes não necessariamente relacionadas entre si. É um repositório central de configurações para ferramentas, como por exemplo, o local onde o NPM e o Yarn armazenam os nomes e as versões de todos os pacotes instalados.

Aqui está um exemplo de um arquivo package.json:

```
{}
```

Está vazio! Não existem campos fixos obrigatórios que devem ser incluídos no arquivo package.json para uma aplicação. A única exigência é que ele siga o formato JSON. Caso contrário, programas que tentem acessar as propriedades de forma programática não terão sucesso.

No entanto, se você está criando um pacote Node.js que precisa ser distribuído no npm, as coisas mudam consideravelmente.

Nesse caso, é necessário preencher algumas propriedades que ajudarão outras pessoas a utilizar seu pacote.

Aqui está um exemplo de outro package.json:

```
{
 "name": "test-project"
}
```

Esse trecho define a propriedade "name", que identifica o nome da aplicação contido na mesma pasta do arquivo package.json está.

Aqui temos outro exemplo:

```
{
 "name": "test-project",
 "version": "1.0.0",
 "description": "React project",
 "main": "src/main.js",
 "private": true,
 "scripts": {
 ...
 },
 "dependencies": {
 ...
 },
 "devDependencies": {
 ...
 },
 "engines": {
 ...
 },
 "browserslist": [...]
}
```

Várias configurações estão em andamento aqui:

- name: Define o nome da aplicação.
- version: Indica a versão atual da aplicação.
- description: Oferece uma breve descrição sobre a aplicação.
- main: Define o ponto de entrada principal da aplicação.
- private: Ao ser definido como true, previne que a aplicação seja publicada acidentalmente no NPM.
- scripts: Define uma série de scripts node que podem ser executados na linha de comando.
- dependencies: Enumera os pacotes NPM instalados como dependências da aplicação.
- devDependencies: Enumera os pacotes NPM instalados como dependências de desenvolvimento.
- engines: Define as versões do Node.js nas quais essa aplicação é compatível.
- browserslist: Utilizado para determinar quais navegadores (e suas versões) terão suporte.

Todas essas propriedades são usadas tanto pelo NPM quanto por outras ferramentas para a gestão e execução da aplicação.

Chegamos ao fim no nosso módulo Bônus. Aqui foram citados algumas tecnologias que estão sendo bastante utilizadas pela comunidade de desenvolvedores pelo mundo todo. Assim que novas tecnologias surgirem e serem bastante requisitadas do mercado estarei adicionando neste módulo.

Mas fique tranquilo, estou sempre atualizando este ebook, e futuramente teremos novos tópicos e exemplos melhores em todos os módulos.

## i18n

"i18n" é uma abreviação para "internacionalização". É um processo usado para tornar um software fácil de ser adaptado para diferentes idiomas e regiões do mundo. Isso significa projetar o software de forma que ele possa exibir texto, datas, números e outros elementos de acordo com as preferências culturais de cada região. Dessa forma, mais pessoas ao redor do mundo podem usar o software em seu próprio idioma.

### Primeiro passo: Instalação do pacote i18n

```
● ● ●
npm install i18next react-i18next
```

### Segundo passo: Configuração do i18n

No seu arquivo de configuração/inicialização, importe e configure o i18N da seguinte maneira:

```
● ● ●
1 import i18n from 'i18next';
2 import { initReactI18next } from 'react-i18next';
```

```
● ● ●
3 import translationEN from './locales/en/translation.json';
4 import translationPT from './locales/pt/translation.json';
5
6 const resources = {
7 en: { translation: translationEN },
8 pt: { translation: translationPT }
9 };
10
11 i18n.use(initReactI18next).init({
12 resources,
13 lng: 'en', // Idioma padrão
14 fallbackLng: 'en', // Idioma de fallback
15 interpolation: {
16 escapeValue: false,
17 },
18 });
19
20 export default i18n;
```

Crie um diretório "locales" na raiz do seu projeto para poder adicionar os arquivos JSON para cada idioma.

## Terceiro passo: Configuração do JSON

Em cada arquivo de tradução, adicione as chaves e valores correspondentes para as traduções.

```
1 // src/locales/en/translation.json
2
3 {
4 "welcome": "Welcome to my app!",
5 "button": "Click me"
6 }
```

```
1 // src/locales/pt/translation.json
2
3 {
4 "welcome": "Bem-vindo ao meu app!",
5 "button": "Clique aqui"
6 }
```

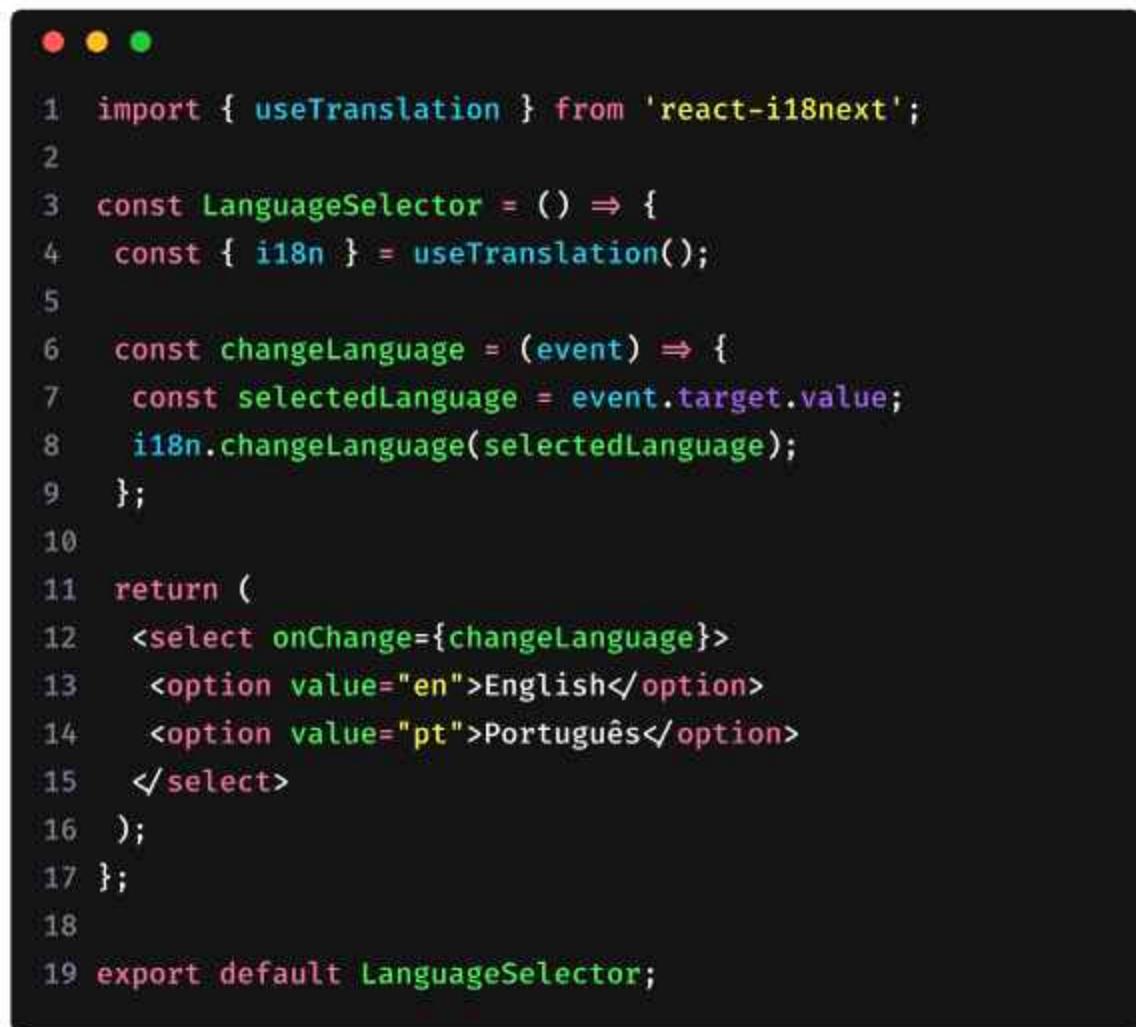
## Quarto passo: Adicionando no componente

Agora, você pode começar a usar o i18N em seus componentes React. Importe o pacote react-i18next e use o componente useTranslation para acessar as traduções.

```
1 import { useTranslation } from 'react-i18next';
2
3 const MyComponent = () => {
4 const { t } = useTranslation();
5
6 return (
7 <div>
8 <h1>{t('welcome')}</h1>
9 <button>{t('button')}</button>
10 </div>
11);
12};
13
14 export default MyComponent;
```

## Quinto passo: Trocando o idioma

Para permitir que os usuários troquem o idioma, você pode adicionar um seletor de idioma e um manipulador de eventos para atualizar o idioma selecionado.



```
 1 import { useTranslation } from 'react-i18next';
 2
 3 const LanguageSelector = () => {
 4 const { i18n } = useTranslation();
 5
 6 const changeLanguage = (event) => {
 7 const selectedLanguage = event.target.value;
 8 i18n.changeLanguage(selectedLanguage);
 9 };
10
11 return (
12 <select onChange={changeLanguage}>
13 <option value="en">English</option>
14 <option value="pt">Português</option>
15 </select>
16);
17 };
18
19 export default LanguageSelector;
```

## Dica extra

À medida que seu projeto cresce, você pode querer carregar as traduções dinamicamente para evitar o carregamento de todos os idiomas de uma só vez.

```
 1 import { useTranslation } from 'react-i18next';
 2
 3 const DynamicComponent = React.lazy(
 4 () => import('./DynamicComponent')
 5);
 6
 7 function App() {
 8 const { t } = useTranslation();
 9
10 return (
11 <div>
12 <h1>{t('welcome')}</h1>
13 <Suspense fallback={<div>Loading ... </div>}>
14 <DynamicComponent />
15 </Suspense>
16 </div>
17);
18 }
19
20 export default App;
```

## Conclusão

A internacionalização, representada pela sigla "i18n", é um componente essencial no desenvolvimento de software que visa tornar as aplicações acessíveis e adaptáveis a diversas culturas e idiomas ao redor do mundo. Ao implementar práticas de internacionalização, os desenvolvedores podem garantir que suas criações ofereçam uma experiência de usuário consistente e significativa, independentemente da localização geográfica do usuário. Em um cenário cada vez mais interconectado, a i18n desempenha um papel crucial na construção de pontes entre diferentes culturas.

## Swagger

As APIs são responsáveis por desempenhar papel fundamental na comunicação entre as mais diferentes aplicações e, apesar de estarem tão presentes nos softwares, elas são invisíveis aos usuários.

Dentro do universo das APIs existe um padrão conhecido como REST (Representational State Transfer). O conceito de REST se aplica à construção de aplicações que se comuniquem com um serviço ou servidor por meio da web, uma API REST utiliza o protocolo HTTP (Hypertext Transfer Protocol) para extrair, inserir, alterar e deletar dados nestes serviços/servidores.

Uma boa documentação de uma API REST é crucial para desenvolvedores entenderem claramente o comportamento.

Existem várias ferramentas que ajudam a criar uma boa documentação, mas a principal e mais utilizada nas empresas é o Swagger. Neste módulo ire apresentar o Swagger e como funciona a sua utilização (não iremos criar uma documentação).

## O que é

O Swagger é um software criado em 2011, que possui um conjunto de ferramentas construídas em torno de uma especificação chamada OpenAPI, que ajuda a projetar, construir, documentar e consumir APIs.

Seu objetivo é proporcionar uma maior simplificação do processo de desenvolvimento por meio de uma documentação eficiente. Com isso, é possível ter dados sobre o endpoint, parâmetros, respostas e exemplos de solicitações e respostas.

Em seu conjunto de ferramentas que tem como finalidade ajudar desenvolvedores a construir documentações APIs incluem:

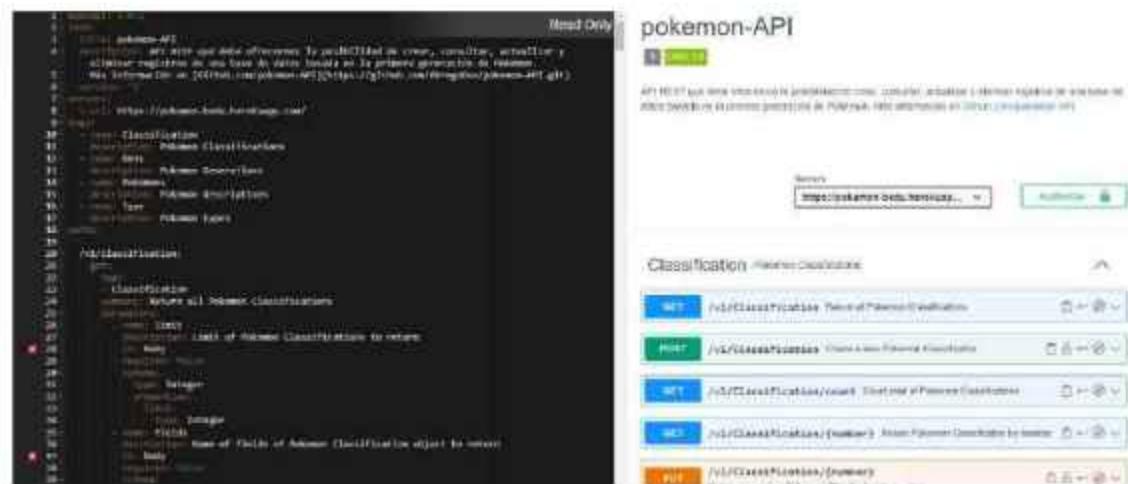
Swagger Editor: editor para escrita de especificações de APIs utilizando a especificação OpenAPI.

Swagger UI: ferramenta capaz de renderizar as especificações OpenAPI como documentação interativa.

Swagger Codegen: gerador de código a partir da documentação especificada via OpenAPI.

## Swagger Editor

O Swagger Editor é uma ferramenta que permite criar manualmente a documentação da API. Ele é mais simples do que parece e possui um conjunto de templates de documentos que servem como base para quem não deseja iniciar a documentação do 'zero'.



Ao lado esquerdo, podemos ver o código da documentação. No lado direito, é possível ver o resultado gerado a partir das informações contidas neste código. Ao editar o lado esquerdo, podemos conferir de forma simultânea o resultado no lado direito.

## Swagger UI

Com o Swagger UI, podemos criar documentações elegantes e acessíveis ao usuário, permitindo assim uma compreensão sobre a API, pois além de poder ver os endpoints e modelos das entidades com seus atributos e respectivos tipos, o Swagger UI possibilita que os usuários da API interajam com a API usando uma sandbox, que é uma plataforma de testes onde as aplicações podem ser alteradas sem interferir na aplicação em produção. Nela, os usuários podem executar todas as operações de mudanças experimentais que vão garantir o bom funcionamento da solução.

### Classification Pokemon Classifications



GET

/v1/Classification Return all Pokemon Classifications



POST

/v1/Classification Create a new Pokemon Classification



Cada barra colorida é uma rota da API. Nas barras podemos ver o caminho da rota, o método utilizado e alguns dos parâmetros necessários. Ao clicar em uma das barras ela se expande mostrando um exemplo de como cada parâmetro deve ser preenchido e a sua resposta.

## Swagger Codegen

O Swagger Codegen é um projeto muito interessante. A partir da especificação em YAML, ele gera automaticamente o "esqueleto" da API em diferentes linguagens, como Java, Python, C++, entre outras. Isso mesmo, com apenas algumas linhas de comando, você pode criar todo o código inicial da sua API na linguagem que desejar.

Você pode verificar quais versões o OpenAPI Specification suporta clicando [aqui](#).

## Conclusão

Este módulo introduziu brevemente algumas das ferramentas mais utilizadas no Swagger. No entanto, a comunidade do Swagger é vasta e abrange diversas outras ferramentas. Você pode conferir muitas delas em [Tools and Integrations](#). O ecossistema em torno do Swagger é rico, facilitando todo o processo de documentação e teste. Vale a pena considerar escrever suas próximas especificações de APIs com esse padrão.

## Next.js 13

O diretório 'app/' é uma nova funcionalidade que aprimora o roteamento e os layouts. Com 'app/', você pode compartilhar facilmente interfaces de usuário entre várias páginas, mantendo o estado e evitando re-renderizações.

As funcionalidades incluídas no 'app/' são:

- Layouts: Agora é possível definir layouts através do sistema de arquivos. Isso permite compartilhar a interface do usuário em várias páginas e manter a interatividade sem re-renderizações.
- Server Components: Introdução da nova arquitetura 'Server Components' do React, que utiliza tanto o servidor quanto o cliente de forma otimizada, resultando em aplicativos mais rápidos e altamente interativos.
- Streaming: É possível exibir estados de carregamento instantâneos e transmitir atualizações, melhorando a experiência do usuário.
- Data fetching: Um novo hook 'use' que permite o fetching em nível de componente.

## Layouts

O diretório 'app/' facilita a implementação de sistemas de layouts para interfaces complexas, permitindo a manutenção do estado entre as navegações, evitando re-renderizações desnecessárias e possibilitando a aplicação de padrões avançados de roteamento.

Agora, também é possível definir layouts por meio do sistema de arquivos. Esses layouts compartilham suas interfaces entre várias páginas, o que permite que, durante a navegação, o estado seja preservado e os layouts permaneçam interativos, sem a necessidade de serem renderizados novamente.



```
layout.jsx u. ×
src/app/blog/layout.jsx BlogLayout
1 export default function BlogLayout({ children }) {
2 return <section>{children}</section>;
3 }
```

## Server Components

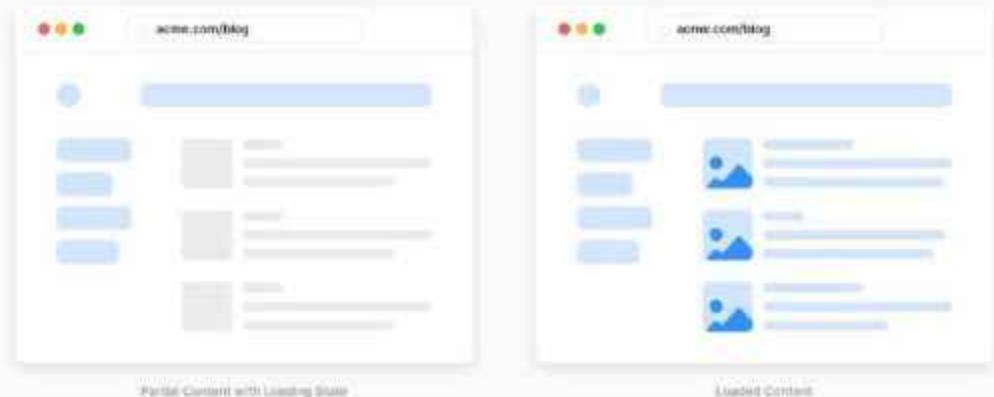
O diretório "app/" também traz suporte para a nova arquitetura de "Server Components" do React. Esses componentes permitem que o servidor e o cliente trabalhem em conjunto, resultando em carregamentos iniciais de página mais rápidos.

Além disso, o "runtime" do Next.js e do React é carregado de forma assíncrona e pode ser armazenado em cache, proporcionando melhorias progressivas no cliente.

Ao usar os Server Components, você reduz a quantidade de JavaScript enviada ao cliente, permitindo carregamentos iniciais de página mais rápidos.

## Streaming

O diretório "app/" possibilita a renderização progressiva e a transmissão aprimorada de unidades renderizadas na interface do usuário para o cliente.



Com "Server Components" e layouts aninhados, é possível renderizar instantaneamente partes da página que não exigem dados e exibir um indicador de carregamento para as partes que estão buscando dados. Isso permite que o usuário comece a interagir com a página antes que ela seja totalmente carregada.

## Data Fetching

O diretório "app/" apresenta uma nova maneira poderosa de buscar dados usando o "Suspense for Data Fetching" do React. O novo hook "use" substitui as APIs anteriores do Next.js, como "getStaticProps" e "getServerSideProps", alinhando-se com o futuro do React.

```
import { use } from "react";

async function getData() {
 const res = await fetch "...";
 const name: string = await res.json();

 return name;
}

export default function Page() {
 const name = use(getData());

 return "...";
}
```

Com o diretório "app/", é possível buscar dados dentro de layouts, páginas e componentes, permitindo o suporte para "streaming responses" do servidor. Isso resulta em experiências mais ricas e interativas no lado do cliente, com menos JavaScript sendo enviado por padrão.

## Turbopack

O Next 13 traz o Turbopack, o novo sucessor do Webpack baseado em Rust. Ele é significativamente mais rápido do que o Webpack e o Vite em testes realizados. O Turbopack suporta Server Components, TypeScript, JSX, CSS e outros recursos.

## @next/image

O Next 13 introduz um novo e poderoso componente chamado Image, que permite exibir imagens facilmente sem afetar o layout e otimizar os arquivos sob demanda para melhorar o desempenho. O componente de imagem envia menos JavaScript para o lado do cliente, é mais fácil de estilizar e configurar, é mais acessível e possui suporte para carregamento "lazy" nativo sem precisar de hidratação.

## @next/font

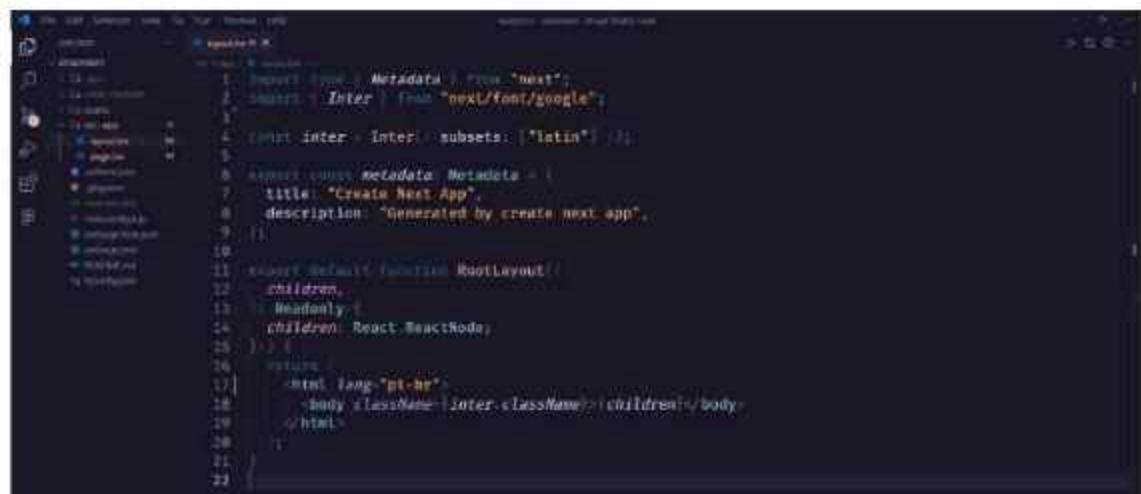
O Next 13 apresenta um novo sistema de fontes que otimiza automaticamente as fontes, incluindo fontes personalizadas, e remove solicitações de rede externa para melhorar a privacidade e o desempenho. O sistema de fontes também oferece auto-hospedagem automática integrada para qualquer arquivo de fonte, permitindo mudanças no layout sem esforço usando a propriedade CSS size-adjust.

## @next/link

O componente next/link não requer mais uma tag <a> como filho. Essa funcionalidade já era uma opção experimental na versão 12.2 e agora está habilitada como padrão. No Next 13, o componente <Link> renderizará uma tag <a>.

## Estrutura do Next 13

Ao criar uma aplicação com Next, iremos nos deparar com uma nova estrutura de pastas e arquivos. Como já dito, agora temos o diretório 'App', mas além dele foi adicionado o arquivo 'layout', que agora será utilizado para configuração de metadados. Vale informar que todos os metadados compartilhados neste arquivo serão apresentados em toda a aplicação, mas você pode criar metadados para páginas específicas.

A screenshot of a code editor showing the contents of a file named '\_app.js'. The code is written in JavaScript and defines a component named 'RootLayout'. The component has a 'children' prop and a 'Headonly' prop. The 'Headonly' prop contains a 'children' prop with a 'React.ReactNode' type. The code also includes imports for 'Metadata' from 'next' and 'Inter' from 'next/font/google'. Metadata is used to define a font stack with the font 'Inter' and subsets 'latin'. A title 'Create Next App' and a description 'Generated by create next app.' are also defined. The 'Headonly' component uses the 'Inter' font for its children. The file ends with a closing brace for the 'RootLayout' component.

```
1 import { Metadata, Inter } from "next";
2 import { Inter as inter, subsets: ["latin"] } from "next/font/google";
3
4 const metadata: Metadata = {
5 title: "Create Next App",
6 description: "Generated by create next app.",
7 }
8
9 export default function RootLayout({
10 children,
11 Headonly: {
12 children: React.ReactNode,
13 },
14 }: {
15 children: React.ReactNode;
16 Headonly: {
17 children: React.ReactNode;
18 };
19 }) {
20 return (
21
22 <!--
23 </body>
24
25 }
26
27
28
29
30
31
32
33
```

## Componentes do Cliente

Já sabemos que os componentes do cliente são renderizados no lado do cliente, mas agora, com essa nova atualização, todos os componentes colocados no diretório App são considerados componentes do servidor. Para designar um componente como componente do cliente, o Next agora exige que você utilize a diretiva 'use client' no topo do componente, pois os componentes do cliente não são renderizados no servidor.

Dê uma olhada no exemplo a seguir de um componente utilizando a diretiva 'use client':

```
1 *use client"; // Client component
2
3 import { useState } from "react";
4
5 const PostUpvoteButton = ({ upvotes, children }) => {
6 const [upvoteCount, setUpvoteCount] = useState(upvotes);
7 return (
8 <div>
9 {children}
10 <button onClick={() => setUpvoteCount(upvoteCount + 1)}>
11 {upvoteCount} Upvotes
12 </button>
13 </div>
14);
15};
16
17 export default PostUpvoteButton;
18 |
```

Por fim, algumas mudanças importantes foram feitas no Next.js 13, incluindo a atualização da versão mínima do React e do Node.js, a remoção de opções e funcionalidades obsoletas e a definição de novos navegadores suportados.

Essas são apenas algumas das principais novidades do Next 13. Essa versão promete melhorar a produtividade dos desenvolvedores e a experiência dos usuários, permitindo criar aplicativos web mais rápidos, interativos e eficientes.

# Obrigado!

Quero expressar minha sincera gratidão por ter adquirido este ebook. Foram meses de trabalho e dedicação para sua finalização. Tenho certeza que este ebook foi de grande importância em sua vida profissional, pois ele foi criado com muito amor, lofi e café.