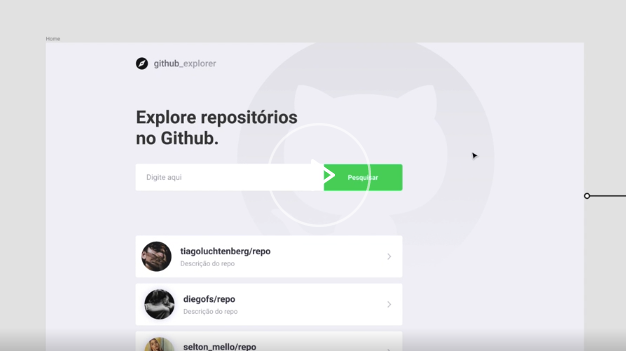
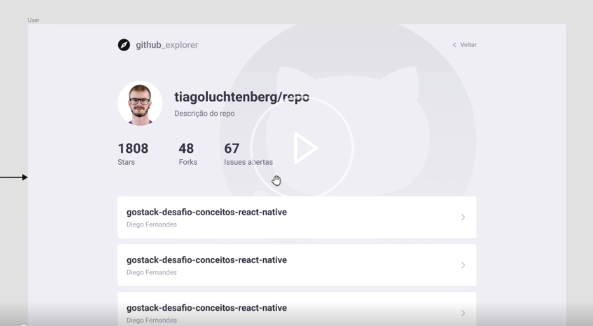
**Criando uma aplicação completa com React Native**

Utilizaremos uma API pública do GitHub (que será nosso backend), para criar uma aplicação com o nome de GitHub Explorer. Ela fará buscas nos repositórios do GitHub a partir de uma tela de consulta.

**Primeira tela (tela principal)**



**Segunda tela** (será aberta ao clicar em um dos elementos da lista da primeira tela)

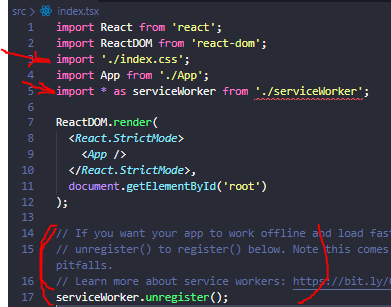


1. **Criando a estrutura básica do projeto**
   1. Na pasta onde deseja criar o seu projeto, digite o comando abaixo:

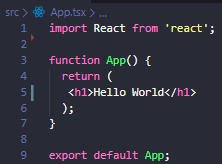
**create-react-app meu-github-explorer --template=typescript**

Este comando já cria a estrutura básica de um projeto React, já disponibilizando o uso do typeScript. Já vem com o webpak e o babel instalado.

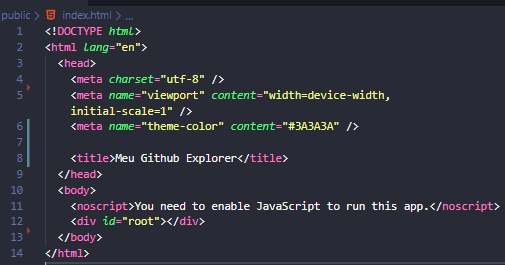
* 1. Abra a pasta do projeto no VsCode e em seguida façamos as seguintes alterações na estrutura do projeto:
     + Delete os arquivos: readme.md.
     + Dentro da pasta SRC, delete os arquivos: App.css, App.test.tsx, index.css, logo.svg e serviceWorker.ts
  2. Na pasta “SRC”, abra o arquivo “index.tsx” e remova os conteúdos marcados abaixo:



* 1. Ainda nesta pasta, deixe o arquivo “App.tsx”, com apenas este conteúdo:



* 1. Na pasta “public”, exclua os arquivos: favicon.ico, logo192.png, logo512.png e manifest.json.
  2. Ainda nesta pasta, abra o arquivo “index.html” e remova as linhas que fazem referência aos arquivos que foram deletados e mais alguns conteúdos. Ele deve ficar com o seguinte conteúdo:



1. **Inicializando nosso servidor**

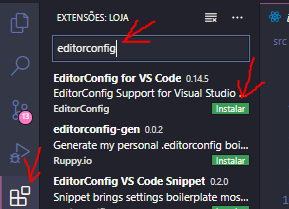
No terminal do VsCode digite o comando: **yarn start** (o browser será aberto automaticamente, com a mensagem Hello World, com a URL localhost:3000)



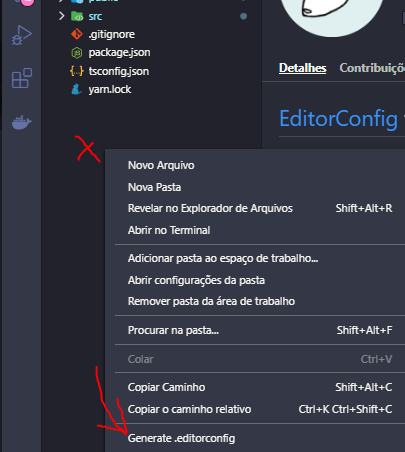
Quando executamos este comando (yarn start), o React executa o scritp que está dentro do arquivo “package.json”. Dentro deste arquivo também temos o script “build”, que é utilizado quando for necessário gerar o arquivo que será publicado no servidor. Dos scripts que estão neste arquivo, recomenda-se não utilizar o “eject”, porque ele vai gerar vários arquivos na estrutura do projeto, que só faria sentido se quisesse fazer uma customização mais complexa no projeto. É muito raro precisar fazer isso.

O arquivo “index.tsx”, da pasta “src” é responsável por pegar o conteúdo do arquivo “index.tsx” desta mesma pasta e jogar dentro do arquivo “index.html”, que está dentro da pasta “public”. Para conseguir executar este processo, o arquivo “index.tsx”, utiliza a biblioteca “react-dom”. O “App.tsx” é o componente principal da nossa aplicação.

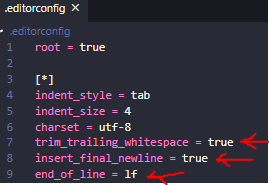
1. **Configurando nosso editor**
   1. Instale a extensão EditorConfig do VsCode



* 1. Clique o botão direito nesta área e selecione a opção “Generate .editorconfig”. Esta ação vai gerar este arquivo na pasta raiz do projeto.



* 1. Abra este arquivo e mantenha ele com o seguinte conteúdo:



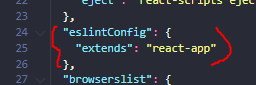
1. **Configurando o ESLint** (Obs.: *este recurso deixa o VsCode mais pesado*)

O que você vai ver aqui, são padrões de configuração utilizadas pelo professor, que aprimorou durante toda a sua experiência de desenvolvimento com React.

* 1. Instale o Eslint pelo terminal do VsCode, somente para o ambiente de desenvolvimento:

**yarn add eslint -D**

* 1. No arquivo “package.json”, remova a referência ao Eslint, que é uma configuração padrão.



* 1. No terminal execute o comando: yarn eslint –init (ele vai fazer uma série de perguntas)
* Na primeira pergunta selecione a última opção
* Na segunda pergunta selecione a primeira opção (javascript modules (import/export)
* A resposta da terceira pergunta é o React.
* Na quarta pergunta responda Yes para Typescript.
* Na quinta pergunta selecione “browser”
* Para a pergunta “user a popular style guide” selecione “airbnb”
* Na pergunta seguinte selecione a opção JSON
* Na pergunta seguinte responda NO (não utilizaremos o npm)
* Em seguida ele vai gerar uma lista de dependências. Selecione e copie esta lista



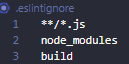
* 1. O próximo passo é, o terminal digite yarn add e cole aqui o conteúdo que você copiou. Antes de executar o comando, faça nele as seguintes alterações. Remova as partes sublinhadas e acrescente -D no final.



Este é o comando:

yarn add eslint-plugin-react@^7.19.0 @typescript-eslint/eslint-plugin@latest eslint-config-airbnb@latest eslint-plugin-import@^2.20.1 eslint-plugin-jsx-a11y@^6.2.3 eslint-plugin-react-hooks@^2.5.0 @typescript-eslint/parser@latest -D

* 1. Crie o arquivo “.eslintignore” na pasta raiz, com o seguinte conteúdo:



Com isso, o Eslint ignorará todos os arquivos “.js” em qualquer pasta do projeto. Ignorará todos os arquivos da pasta node\_modules e também os da pasta build (onde serão gerados nossos arquivos para o ambiente de produção).

* 1. Abra o arquivo “eslintrc.json” e faça as seguintes alterações:
     + Em extends, inclua: "plugin:@typescript-eslint/recomended"
     + Em plug-ins, inclua: "react-hooks",
     + Nas rules, acrescente o seguinte conteúdo:

"rules": {

        "react-hooks/rules-of-books": "error",

        "react-hooks/exhaustive-deps": "warn",

        "react/jsx-filename-extension": [1, {"extensions": [".tsx"]}],

        "import/prefer-default-export": "off",

"import/extensions": [

            "error",

            "ignorePackages",

            {

                "ts": "never",

                "tsx": "never"

            }

        ]

     }

* 1. Agora execute o seguinte comando**: yarn add eslint-import-resolver-typescript -D**
  2. Ainda, terminado as rules acrescente logo abaixo o “settings”:

"settings": {

        "import/resolver": {

            "typescript":{}

        }

    }

Isto possibilitará ao Eslint reconhecer o Typescript.

1. **Adicionando o Prettier ao projeto**

yarn add prettier eslint-config-prettier eslint-plugin-prettier -D

* 1. Acrescente estas duas linhas à tag “extends” do arquivo “.eslintrc.json”

"prettier/@typescript-eslint",

"plugin:prettier/recomended"

* 1. Na tag “plugins” acrescente também o prettier

"prettier"

* 1. E na tag “rules”, a seguinte linha:

"prettier/prettier": "error",

* 1. Crie o arquivo “prettier.config.js” na pasta raiz do projeto, com o seguinte conteúdo:

*module*.*exports* = {

    singleQuote: true,

    trailingComma: 'all',

    allowParens: 'avoid'

}

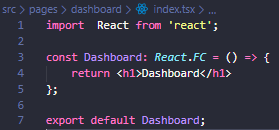
1. **Criando as rotas**

Nosso aplicativo terá duas telas, então criaremos a rota para fazer esta navegação. Utilizaremos a biblioteca “react-router-dom” para esta funcionalidade.

**yarn add react-router-dom**

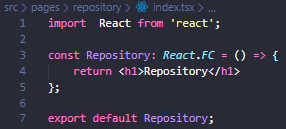
**yarn add @types/react-router-dom -D**

* 1. Crie a pasta “pages” dentro da pasta “src”.E dentro dela, a pasta “dashborad”, com o arquivo “index.tsx”, com o seguinte conteúdo:

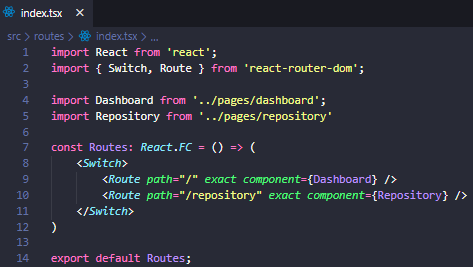


**Linha 3 –** criamos uma variável do tipo “React.FC”. FC é a abreviação de Function Component.

* 1. Ainda dentro da pasta “src”, crie a pasta “repository” e dentro dela o arquivo “index.tsx”, com o mesmo conteúdo do arquivo anterior, apenas modificando o nome Dashboard, para “Repository”:



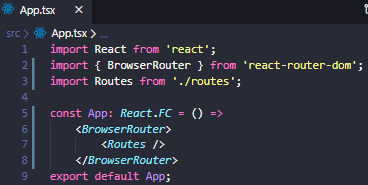
* 1. Crie a pasta “routes” dentro da pasta “src”. E dentro dela o arquivo “index.tsx”, com o seguinte conteúdo:



**Linha 8 –** Se não tivesse o “switch”, as duas páginas seriam exibidas na mesma tela.

**Linha 9** – considerando que a página “Dashboard” será a página principal (ou primeira página), ela deverá ser acessada somente com a barra após a URL. É necessário o termo “exact” para que o roteador reconheça, que a página principal será acessada digitando somente a barra no final da URL.

* 1. O Conteúdo do nosso arquivo App.tsx, agora deverá ficar da seguinte forma:



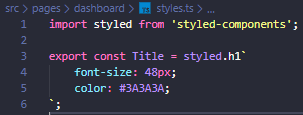
**Linha 5** – esta é uma forma abreviada de criar as funções da forma com fizemos anteriormente. Isso somente quando se trata de funções simples, sem conteúdo.

1. **Estilizando os componentes**

No terminal execute o comando: **yarn add styled-components**

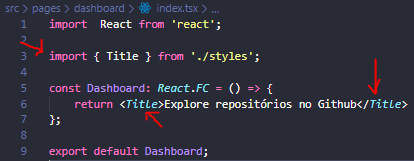
E também **yarn add @types/styled-components -D**

Esta biblioteca, isola os arquivos CSS por componente, de forma que o CSS de um componente não interfere no CSS de outro componente. Mas isso não impede de podermos criar também CSSs globais.

* 1. Criaremos um arquivo com o nome de “styles.ts” dentro da pasta “dashboard”. Se criássemos como “styles.css” ele se tornaria um css global na aplicação. O conteúdo inicial deste arquivo será o seguinte:

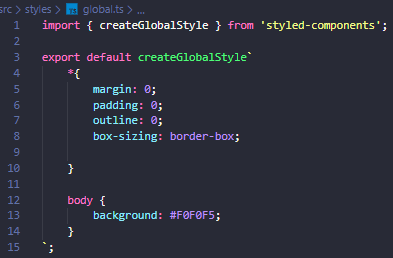
Para ficar assim com estas cores, é necessário instalar o plugin “styled-components” do vsCode.

* 1. Agora, no arquivo “index.ts”, da pasta “dashboard”, faça a importação deste arquivo de estilos e altere a tag <h1> para <Title>, que é o estilo que criamos na linha 3 acima.

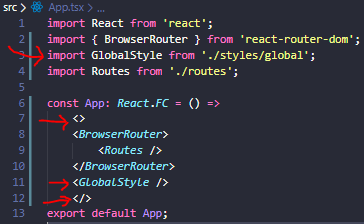


A utilização do “styled-components”, possibilita que o CSS no React, tenha a mesma sintaxe do CSS do html.

* 1. Criaremos agora nosso estilo css glogal. Na pasta “src”, crie a pasta “styles” e dentro dela o arquivo “global.ts”, com o seguinte conteúdo:



Este arquivo nós importaremos dentro do arquivo App.ts da nossa pasta “src”, que ficará da seguinte forma:



**Linha 3** – importamos o GlobalStyle que criamos acima.

**Linhas 7 e 12** – Criamos estas tag vazias, por uma necessidade do React manter o conteúdo sempre entre tags.

Linha 10 – Aplicando o GlobalStyle dessa forma, ele envolve todo o conteúdo da página.

A fonte utilizada no layout do projeto é a ‘Roboto’. Podemos pegá-la no fonts.google.com/specimen/Roboto

* 1. Selecione a Regular 400 e a bold 700, clique na aba “embebed” e copie o código.

<link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap" rel="stylesheet">

* 1. Cole este código no head do arquivo index.html, da pasta public.
  2. Acrescente esta fonte ao arquivo de css global. Ele deverá ficar da seguinte forma:



**Linha 2** – importamos a imagem que utilizemos como fundo de tela.

**Linha 14** – aplicamos a imagem ao fundo da tela, com 70% de margem esquerda e alinhada com o topo.

**Linha 19** – aplicamos a fonte “roboto” que importamos do fonts.google (caso ocorra problema ele aplicará a fonte sans-serif)

**Linha 22** – “root” é o ID da div da nossa tela principal. Utiliza-se o **max-width** para que a largura possa se adaptar à tela de um celular.

* 1. Colocar o logo do Github, no topo da tela.

No arquivo index.tsc, da pasta “dashboard”, faça o import da imagem ‘logo.svg’.

**Linha 2** – importamos a imagem da logo.

**Linha 7** – colocamos agora todo o conteúdo entre parênteses. Quando temos somente um elemento não precisamos dos parênteses. Antes tínhamos somente o <Title>.

**Linha 8** – Quando temos dois elementos, um embaixo do outro, precisamos ter a tag ‘fragments’ (<>), que fecha na linha 11.



* 1. No arquivo de estilos da pasta “dashboard”, acrescentamos mais duas linhas:

**Linha 6** – Limitamos a largura do h1 para 450px, de modo que tenha uma quebra de linha, exatamente como no layout.

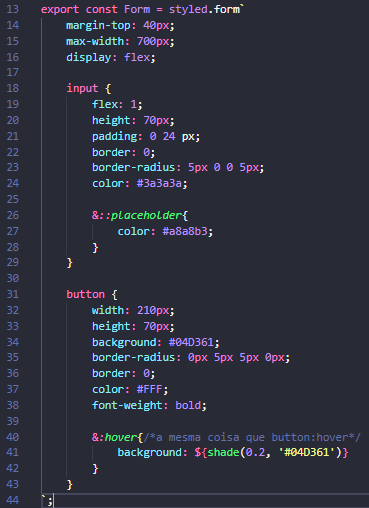
**Linha 7** – Ajuste de entrelinha.

1. **Acrescentando o campo de pesquisa na tela principal**



**Linha 4** – importamos o componente Form. Mas ante precisamos cria-lo, no arquivo “styles.ts” da pasta do dashboard.

**Linha 12** – Utilizamos o componente form. Observe que, por se tratar de um componente, utiliza-se letra maiúscula (Form)

* 1. Estilizando o campo de pesquisa
* Instale a biblioteca “polished”: **yarn add polished** (para aplicarmos sobreamento a um botão)
* Crie o componente Form dentro do arquivo de estilos da pasta dashboard, com o seguinte conteúdo:

**Linha 13** – criamos nosso componente Form. Dentro deste componente criamos o css dos seus elementos internos (input e button).

**Linha 16** – coloca um elemento ao lado do outro. No caso o input ao lado do button.

**Linha 19** – coloca a largura do input limitada à largura do button.

**Linha 26** – é a mesma coisa que input:placeholder

**Linha 40** – é a mesma coisa que button:hover

**Linha 41** – O recuros ${..} é a maneira de implementarmos javascript dentro do CSS. A função “shade()” é uma função da biblioteca “polished”, que acabamos de instalar.

* 1. Estilizando a lista de repositórios

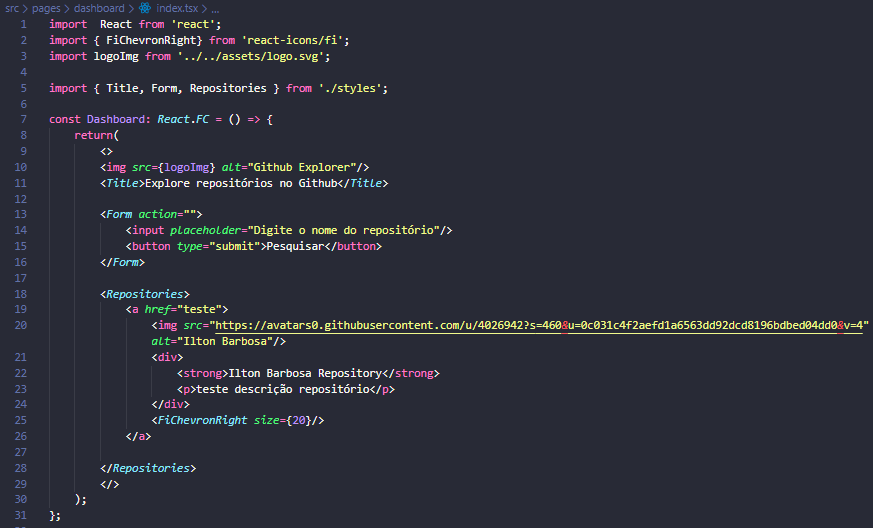
Observe que para cada item da nossa lista de repositórios, temos um ícone clicável à direita. Para implementarmos este ícone utilizaremos a biblioteca “react-icons”

**yarn add react-icons**

* Em seguida importamos esta biblioteca no arquivo index.tsx da pasta “dashboard”. Utilizaremos somente o elemento FiArrowRight desta biblioteca, que é justamente a seta para a direita.

import { FiChevronRight} from 'react-icons/fi';

* 1. Agora, nosso arquivo index.tsx, ficará da seguinte forma:

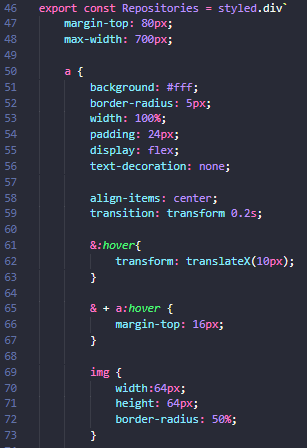


**Linha 2** – temos o import do react-icons/fi.

**Linha 18** – implementamos o componente Repositories. Lembrando que precisamos criar o estilo para este componente no arquivo de estilos da pasta dashboard.

**Linha 25** – implementação da seta para a direita, conforme está no nosso layout.

Por enquanto estamos utilizando aqui, somente dados estáticos, que serão substituídos por dados dinâmicos quando fizermos a integração com o nosso backend.

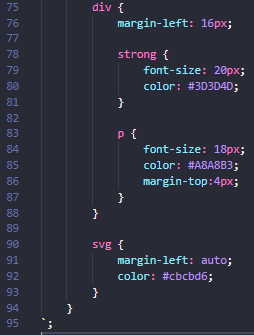
* 1.  Criando o estilo Repositories no arquivo de estilos da pasta dashboard.

**Linha 46** – criamos o componente de estilo “Repositories”.

**Linha 55** – display: flex para alinhamento dos elementos numa mesma linha.

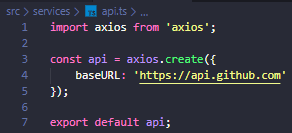
**Linha 59 e 62** – move o elemento suavemente para a direita, ao passar o mouse sob ele.

**Linha 65** – Se o elemento <a> for precedido de outro elemento <a>, ele acrescentará um espaço entre um e outro, de 16px. Este recurso é utilizado para listas de elementos.

 **Linha 75** – esta DIV é a que temos dentro do <a href> na lista de repositórios na nossa página principal.

**Linha 90** – é a seta para a direita que temos na lista de repositórios na página principal.

1. **Integrando o sistema com a API do Github**
   1. Instale o axios: **yarnn add axios**
   2. Crie a pasta “services” na raiz do projeto, e dentro dela o arquivo “api.ts”, com o seguinte conteúdo:



* 1. Importe este arquivo dentro do arquivo “index.tsx” da pasta “dashboard”.

import api from '../../services/api';

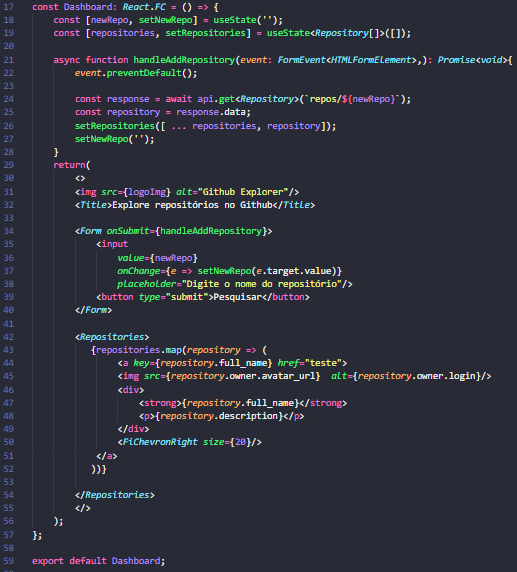
* 1. Importe também para este mesmo arquivo os recursos UseStat e FormEvent, do React que já temos na importação ( linha 1), para que possamos armazenar nossos repositórios em algum lugar.

import  React, {useState, FormEvent } from 'react';

* 1. Criamos também a interface que representará o repositório do github que utilizaremos na nossa aplicação para buscar dados. Desse repositório utilizaremos somente 3 atributos. Veja linhas 8 a 15.



O arquivo “index.tsx” da pasta “dashboard”, segue com o seguinte conteúdo:



**Linha 18** – criamos a variável (*constante*), newRepo e na estrutura dela temos a função “setNewRepo” que é utilizada para atribuirmos valor a esta variável (*Princípio da orientação a objeto, nunca setar diretamente um valor a uma variável*).

**Linha 19** – criamos também a variável “repositories” e estamos instanciando nela a classe “useState” da biblioteca React, que importamos na linha 1. Veja que estamos passando como parâmetro a interface que criamos na linha 8. Dessa forma estamos informando que a nossa variável “repositories” receberá conteúdo somente do tipo da nossa interface “repositories”. Veja que ela também tem a função setRepositories().

**Linha 21** – Esta é uma função assíncrona porque ela fará consultas a uma url externa, que é a API do github, que retornará os repositórios da consulta que faremos. Veja que esta função recebe como parâmetro um evento do formulário, que no caso seria quando o usuário clica no botão “submit”.

**Linha 22** – evita que ocorra um reload na página após clicar no botão submit.

**Linha 24** – faz a pesquisa no repositório do github a partir da palavra que o usuário digitou no campo de pesquisa. Veja que a palavra digitada pelo usuário ficou armazenada na variável “newRepo”. O valor foi armazenado nesta variável por meio da ação “onChange”, que temos no formulário (linha 37). Veja também que estamos utilizando o componente “api” que criamos na pasta servisse e que contem a url do repositório no github.

**Linha 25** – o conteúdo retornado da API do github é armazenado na variável “response.data”.

**Linha 26** – Estamos acrescentando o conteúdo retornado da API do github ao final do array “Repositories” que criamos na linha 19.

**Linha 27** – estamos limpando o campo de consulta do nosso formulário.

**Linha 34** – ao clicar no botão “submit”, o formulário executa a função “handleAddRepository” que criamos na linha 19.

**Linha 37** – a palavra digitada no campo de pesquisa é armazenada na variável newRepo.

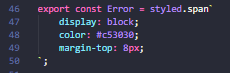
**Linha 42 a 54** – lista o resultado da consulta.

**Linha 50** – é a seta para a direita que temos na nossa lista de repositórios.

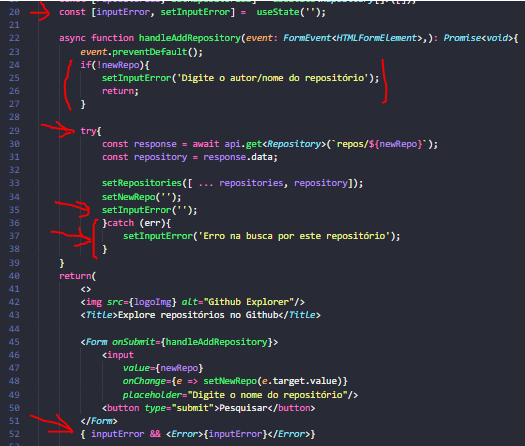
1. **Lidando com erros**

O que fazer se o usuário clicar no botão “submit” sem que tenha digitado nada no campo de pesquisa, ou tenha digitado um termo que não existe no repositório.

* 1. No arquivo de estilo da pasta “dashboard”, crie o seguinte estilo:

 A mensagem será exibida em uma tag <span>

* 1. No arquivo index da pasta Dashborad, importe este estilo (linha 6) e acrescente as seguintes alterações:



**Linha 20** – criamos a variável que vai armazenar a mensagem de erro.

**Linha 24** – se o usuário não digitar nada no campo do formulário, a variável “newRepo” estará vazia. Se não colocar o return da linha 26, a função continuará executando as próximas linhas.

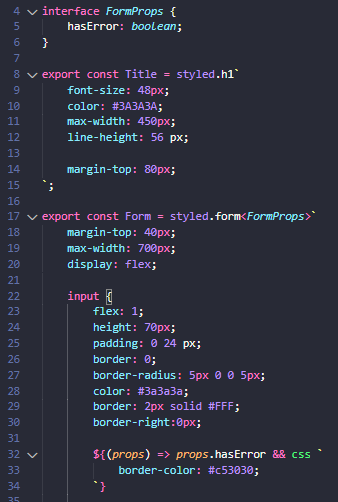
**Linhas 29 a 36** – colocamos estas linhas dentro de um try catch. Se ocorrer algum erro na pesquisa, a mensagem a ser exibida é a da **linha 37**.

**Linha 35** – limpamos a mensagem de erro após uma segunda consulta.

**Linha 52** - A mensagem de erro será exibida logo abaixo do campo de pesquisa, somente se a variável “inputError” não estiver vazia.

* 1. Aperfeiçoando um pouco mais os erros.

Se quisermos que a nossa caixa do formulário fique em destaque, indicando que o campo precisa ser preenchido, teremos que criar uma interface de erro no nosso CSS e passá-lo como parâmetro no nosso componente Form e em seguida implementa-la no nosso formulário.

**Linha 4** – criamos nossa interface com uma variável booleana.

**Linha 17** – passamos nossa interface como parâmetro para o nosso componente Form. Assim a variável “**hasError**” passou a ser uma propriedade (característica) do nosso formulário.

**Linha 29 e 30** – colocamos uma borda branca, que será alterada para vermelho quando o erro ocorrer.

**Linha 32** – ele verifica se o formulário possui alguma propriedade e se possuir verifica se ela é true. Se sim aplica a borda vermelha ao campo de pesquisa do nosso formulário. Mas para isso é necessário implementá-la no formulário, como veremos a seguir.

* 1. No nosso formulário inclua a propriedade “hasError”, da seguinte forma:

Estamos transformando a variável “inputError” no nosso arquivo “index.tsx” da pasta “dashboard”, para o tipo booleano. Se ela conter alguma mensagem de erro, então ela passará a ter o valor “true”. Dessa forma o campo input ficará com a borda vermelha.

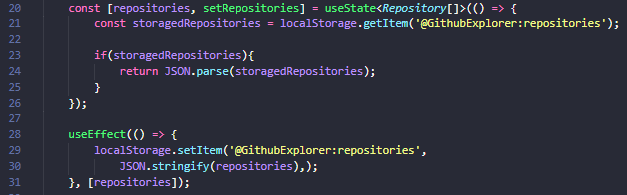
1. **Salvando o resultado da busca em um storage local**.

O storage local seria como um banco de dados textual ou simplesmente um cookie do browser, de forma que possa manter a lista de resultados das nossas pesquisas de repositórios do github.

* 1. Inclua o **useEffect** no import da linha 1

import  React, {useState, useEffect, FormEvent } from 'react';

* 1. Altere a declaração da variável **repositories**, para o seguinte:



**Linha 20** – Trocamos o array que tínhamos no final da linha, por uma função. Esta função cria um repositório local com o nome de ‘@GithubExplorer:repositories’ (ela é do tipo repositories).

**Linha 23** – Se o storage não estiver vazio ele faz a conversão inversa do JSON para array e armazena na variável repositores, da linha 20. É esta variável que exibe a lista de variáveis na tela.

**Linha 28** – o useEffect é um recurso que fica observando a variável ‘repositories’. Se ela sofrer alguma alteração no seu conteúdo, ele então atualiza o repositório local.

1. **Navegando de uma página para outra**
   1. Comecemos importando o componente Link da biblioteca **react-router-dom**

import { Link } from 'react-router-dom';

* 1. Trocamos agora, o nosso <a href> que tínhamos na linha 71, pelo seguinte:



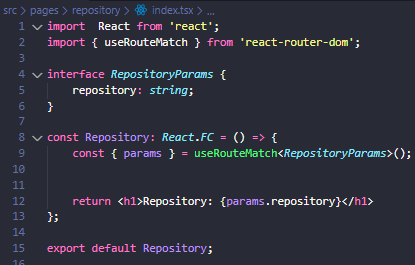
Somente trocar o “a” pela palavra “Link” e o “href” por “to” com este conteúdo aí, que é o parâmetro que vai ser lido/capturado pela próxima página, que vai mostrar detalhes do repositório clicado. Lembrando que também é necessário trocar o “**</a>**” por “**</Link>**”na linha 78.

* 1. Agora temos que informar à nossa rota de repositório, que ela agora recebe um parâmetro:



O sinal de “+” indica que tudo o que vier após a palavra “repository” faz parte dela, como parâmetro.

* 1. Alteraremos agora nosso arquivo “index.tsx” da pasta “page/repository”.



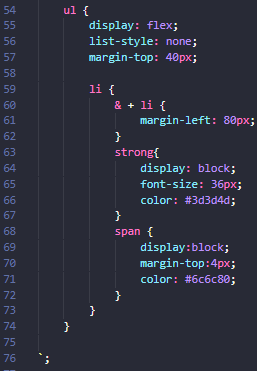
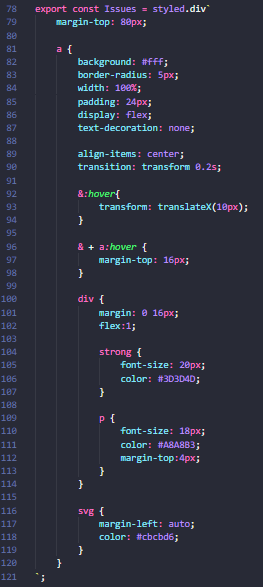
Criamos a interface e a colocamos como parâmetro do useRouteMatch (linha 9).

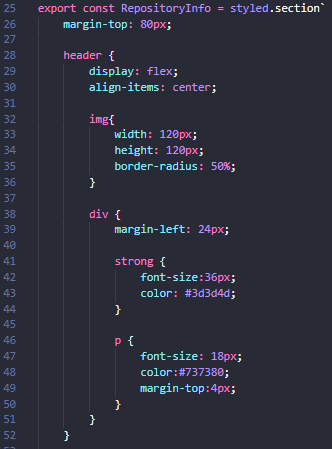
Agora na linha 12, já conseguimos exibir o título do repositório que clicamos, na página de detalhes do repositório. O próximo passo e exibir o restante das informações sobre o repositório.

1. **Criando a página de detalhes do repositório**

Primeiro montaremos toda a parte visual, com dados estáticos e depois que estiver tudo ok, substituiremos pelos dados dinâmicos.

* 1. Na pasta “page/repositor”, crie o arquivo styles.ts, com o seguinte conteúdo:

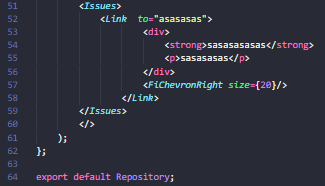




O conteúdo do estilo Issues foi praticamente copiado  
do CSS de Repositories, retirando somente a parte de imagem. O professor recomenda que quando a redundância de código acontece somente uma vez, não há necessidade de criar um componente à parte para ser reaproveitado.

* 1. O arquivo “index.tsx” da pasta “page/repository”, ficará da seguinte forma, por enquanto somente com dados estáticos:

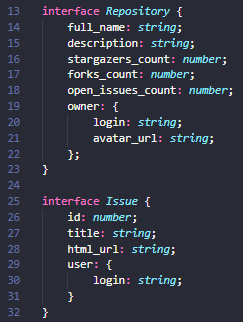




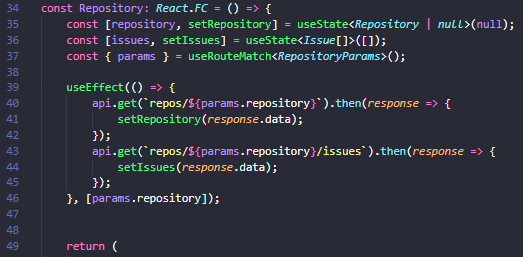
* 1. Integração desta página com a API do Github.
     + Importar a api

import api from '../../services/api';

* Criar as interfaces do conteúdo que virá do repositório do github.



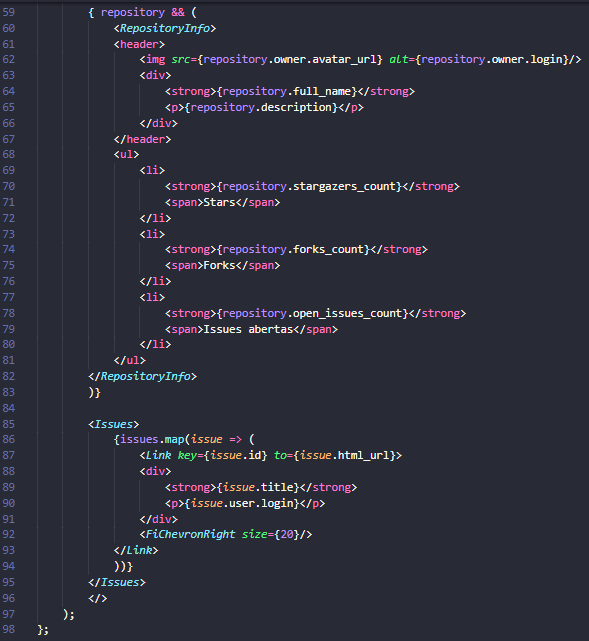
* Implementar o “useState” nas nossas variáveis.



Observe a diferença no uso do useState em “repository” e em “issues”. No caso a variável “issues” é um array. O **useRouteMatch** é para especificarmos qual o tipo de dado que será enviado/recebido via url/parâmetro.

O useEffect busca os dados de repositório e issues da api do Github. Essa é uma maneira de implementar a busca de dados em servidores externos, sem a necessidade de utilizar funções assíncronas. Este recurso não aceita o uso do “async”. O “**then**” tem praticamente a mesma função do “async”. O uso do udeEffect se aplica quando precisamos trazer conteúdos diferentes ao mesmo tempo, sem que um dependa do outro. No caso, o conteúdo de repository não depende do conteúdo de issues. Ambos estão sendo trazidos a partir da consulta por “params.repository”, que é o parâmetro enviado via URL, pela página principal.

* Abaixo do return (linha 49), temos o <header> que não sofreu alteração. E mais abaixo, temos a implementação da exibição dos dados do repositório e a lista de issues.



**Linha 59** – verifica se a variável “repository” não está vazia.

**Linha 60** – RepositoryInfo é o nosso componente css.

**Linha 86** – O recurso “map” funciona como um foreach.

**Fonte: Curso de React da Rocketseat**

**Maio/2020**