



# GraphQL

Material Complementar



# Marianne Salomão

Cloud Engineer na IBM  
Brasil, desenvolvedora de  
software full - stack e  
instrutora de TI.

#PraCegoVer: Fotografia da  
autora Marianne Salomão



[linkedin.com/in/mariannesalomao](https://linkedin.com/in/mariannesalomao)



[github.com/mariannesalomao](https://github.com/mariannesalomao)

# Introdução

**GraphQL** é uma linguagem de consulta e ambiente de execução voltada a servidores para as APIs cuja prioridade é fornecer **exatamente os dados** que os clientes solicitam e nada além.

## **DICA:**

Durante a leitura desta apostila iremos estudar vários códigos, utilize a ferramenta de zoom para uma melhor visualização..



# 01. O que é GraphQL?

É simplesmente uma linguagem de consulta, que foi desenvolvida para tornar as APIs mais rápidas, flexíveis e intuitivas para os desenvolvedores.

O **GraphQL** proporciona aos profissionais responsáveis pela manutenção das APIs flexibilidade para adicionar ou preterir campos, sem afetar as consultas existentes. Os desenvolvedores podem criar APIs com o método que quiserem, pois a especificação do GraphQL assegura que elas funcionem de maneira previsível para os clientes.



## 02. Termos

Os desenvolvedores de API usam o GraphQL para criar um **esquema (schema)** para descrever todos os dados disponíveis para consulta pelos clientes por meio do serviço em questão.

Um esquema do GraphQL é composto por tipos de objeto que definem os objeto que podem ser solicitados e quais campos eles terão.

Conforme as consultas (queries) são recebidas, o GraphQL as valida de acordo com o esquema. Em seguida, o GraphQL executa as consultas validadas.



## 03. Consulta do GraphQL

O primeiro exemplo mostra como um cliente pode construir uma consulta do GraphQL, solicitando à API que retorne campos específicos no formato determinado:

```
{  
  me {  
    name  
  }  
}
```



## 03. Consulta do GraphQL

Uma API GraphQL retornaria um resultado como o abaixo no formato JSON:

```
{  
  "me": {  
    "name": "Dorothy"  
  }  
}
```





## 03. Consulta do GraphQL

Um cliente também pode transmitir argumentos como parte da consulta do GraphQL, como neste exemplo:

```
{  
  human(id: "1000") {  
    name  
    location  
  }  
}
```





## 03. Consulta do GraphQL

O resultado:

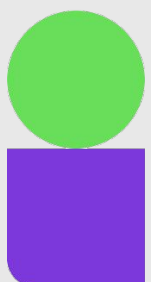
```
{  
  "data": {  
    "human": {  
      "name": "Dorothy,  
      "location": "Kansas"  
    }  
  }  
}
```



## 03. Consulta do GraphQL

A partir daí, as coisas ficam mais interessantes. Com o GraphQL, os usuários podem definir fragmentos reutilizáveis e atribuir variáveis.

Imagine que você precisa solicitar uma lista de IDs e depois uma série de registros de cada ID. Com o GraphQL, é possível elaborar uma consulta que extraia todos os dados que você quer em uma única chamada de API.

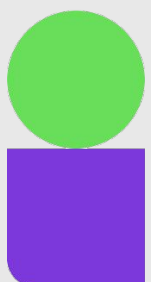


## 03. Consulta do GraphQL

Portanto, esta consulta:

```
query HeroComparison($first: Int = 3) {  
  leftComparison: hero(location: KANSAS) {  
    ...comparisonFields  
  }  
  rightComparison: hero(location: OZ) {  
    ...comparisonFields  
  }  
}
```

```
fragment comparisonFields on Character {  
  name  
  friendsConnection(first: $first) {  
    totalCount  
    edges {  
      node {  
        name  
      }  
    }  
  }  
}
```

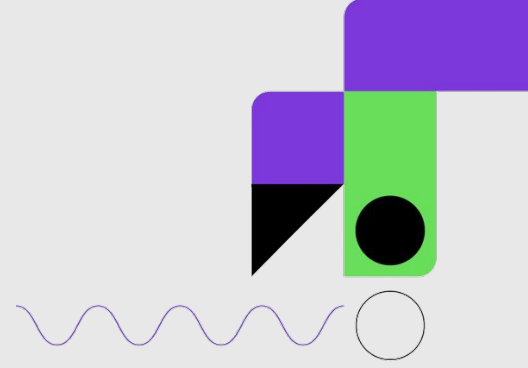


**Pode produzir o seguinte resultado:**

```
{
  "data": {
    "leftComparison": {
      "name": "Dorothy",
      "friendsConnection": {
        "totalCount": 4,
        "edges": [
          {
            "node": {
              "name": "Aunt Em"
            }
          },
          {
            "node": {
              "name": "Uncle Henry"
            }
          },
          {
            "node": {
              "name": "Toto"
            }
          }
        ]
      }
    },
    "rightComparison": {
      "name": "Wizard",
      "friendsConnection": {
        "totalCount": 3,
        "edges": [
          {
            "node": {
              "name": "Scarecrow"
            }
          },
          {
            "node": {
              "name": "Tin Man"
            }
          },
          {
            "node": {
              "name": "Lion"
            }
          }
        ]
      }
    }
  }
}
```



## 04. Projeto

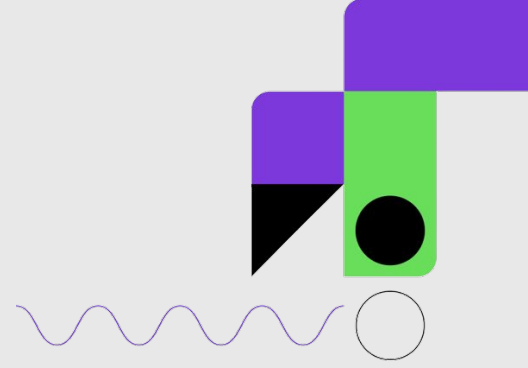


O primeiro passo será criar um novo projeto.

Para isso, escolha uma nova pasta no seu computador, abra um terminal, navegue até esse diretório e execute o comando abaixo:

```
npm init -y
```

## 04. Projeto



O resultado desse comando será um arquivo chamado **package.json** com as informações default do seu projeto, como: nome, versão, etc.

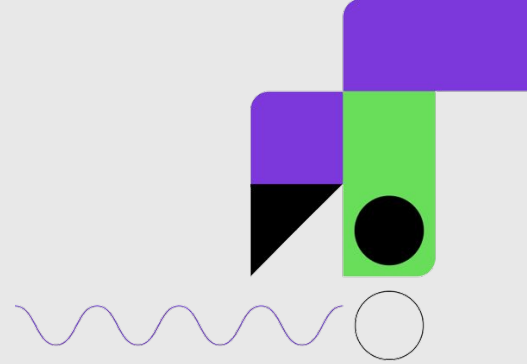
O próximo passo será importar os pacotes do portal NPM para o nosso projeto. Ainda com o seu terminal aberto e dentro da pasta que você executou no passo anterior, execute o comando abaixo:

```
npm install express
```

```
graphql express-graphql
```

```
--save
```

## 04. Projeto



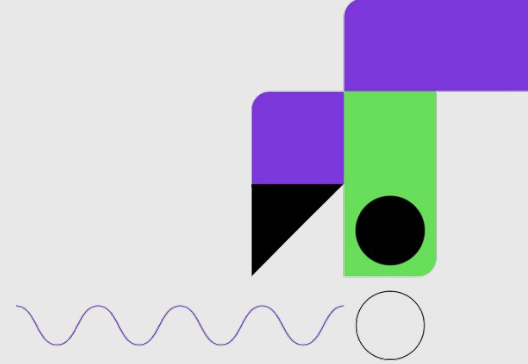
Esse comando deve criar uma nova pasta na sua estrutura, chamado **nome\_modules** com os pacotes que acabamos de importar no nosso projeto.

Agora vamos criar um novo arquivo para criarmos as nossas rotas.

Para isso eu irei criar um chamado **app.js**. Abra esse arquivo em um editor de textos e atualize ele com o trecho de código a seguir:



# 04. Projeto



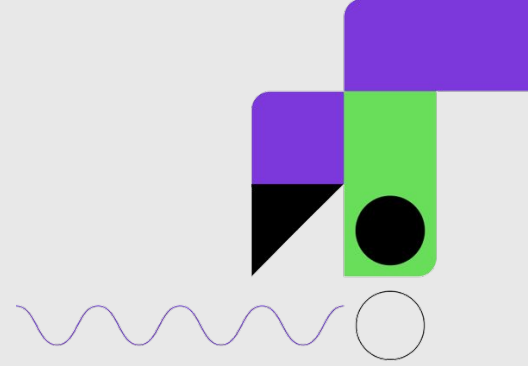
```
var express = require('express');
var express_graphql =
require('express-graphql');
var { buildSchema } = require('graphql');

// Schema
var schema = buildSchema(`
  type Query {
    message: String
  }
`);

// Mapeamento
var root = {
  message: () => 'Hello World!'
};

var app = express();
app.use('/graphql', express_graphql({
  schema: schema,
  rootValue: root,
  graphql: true
}));
app.listen(3000, () => console.log('Express
GraphQL Server Now Running On
localhost:3000/graphql'));
```

## 04. Projeto



Analizando o trecho de código anterior, nós temos:

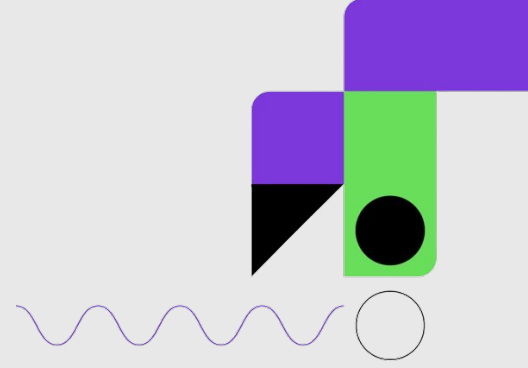
- **01 até a 03:** estamos importando os pacotes `express` e `graphql`
- **04 até a 09:** estamos criando um simples GraphQL schema
- **10 até a 13:** estamos criando um root resolver – ele contém o mapeamento do retorno do nosso `buildSchema`
- **15 até a 21:** estamos criando um server com uma rota para `graphql`. Nessa parte nós temos três pontos importantes:

1º mapeamento dos nossos schemas.

2º os nossos mapeamentos.

3º **`graphqliql`** (quando essa flag está como `true`, nós habilitamos o modo interativo do GraphQL no browser).

## 04. Projeto



Agora execute o comando **node + (nome que você escolheu para o seu arquivo index)**, aqui ficou **node app.js**.

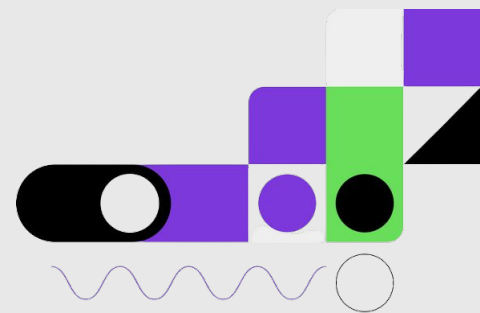
E você verá a frase "Hello Word" ser renderizada na tela.

## 05. Monorepo

O Monorepo como o próprio nome diz, seria um único repositório para manter o código de diversos projetos.

Em um ambiente de trabalho, na qual há diversos projetos separados que são integrados, é comum os desenvolvedores precisarem criar mocks, ou precisar carregar diversos projetos para realizar algumas atividades simples.

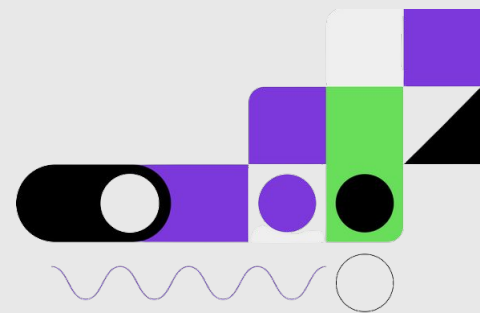
**Por exemplo:** Você possui um projeto que é responsável por realizar o Login. Esse projeto está em 10 outros projetos. Foi encontrado uma vulnerabilidade nesse projeto de Login. Com um Monorepo o ajuste seria bem mais simples.



## 05. Monorepo

### Vantagens:

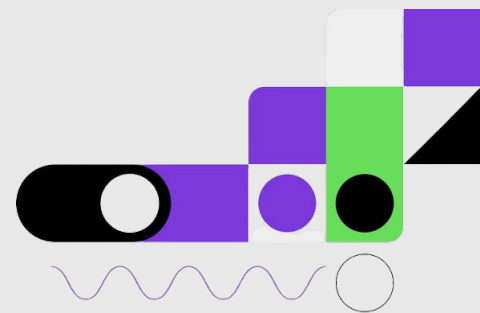
- Com a utilização de Monorepo, os diversos projetos estarão em apenas um único git, isso ajudará na reutilização de código.
- Com Monorepo é mais simples fazer um build de todos os projetos que foram alterados.
- É mais simples manter um único estilo visual para os diversos projetos, visto que é possível ter um projeto em comum para lidar com css e imagens.
- A integração de um novo membro no time tende a ser algo mais rápido e tranquilo, pois o mesmo não precisará baixar e configurar diversos projetos separados.



## 05. Monorepo

### Desvantagens:

- Criar a cultura de utilizar um único repositório.
- Com base no crescimento do projeto, ele se torna mais custoso para realizar o clone.
- Não há uma forma de criar segurança entre os projetos, todos os desenvolvedores podem trabalhar em todos os projetos.
- Dependendo do sistema de Git, é bom avaliar o tamanho máximo do histórico de arquivos permitido, para assegurar que utilizar um Monorepo não será algo custoso.





## 06. Projeto (back-end)

O projeto é um website CRUD sobre frutas onde podemos gerenciar os dados fazendo as operações CRUD.



**O site será feito em React.js e o servidor será em Node.js.**

---





## 06. Projeto (back-end)

Para começar o projeto crie uma pasta chamada **fruits** e dentro dela crie uma pasta **backend**.

Abra um terminal de comandos e navegue para a pasta **fruits/backend** e execute os comandos a seguir:

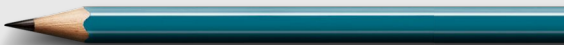
---

```
npm init -y
```



## 06.1 Dependências

```
npm i graphql apollo-server mongoose  
dotenv
```

- **graphql:** Montar shemas e executar queries GraphQL 
- **apollo-server:** Montar servidor GraphQL
- **mongoose:** Modelar dados do banco e conectar ao banco MongoDB
- **dotenv:** Configurar variáveis de ambiente para MongoDB



## 06.2 Servidor

Um servidor GraphQL contém principalmente **definições de tipo e resolvers**.

Além disso, para configurar o acesso ao banco de dados e o schema dos dados do banco, teremos mais um item chamado **Models**. Dessa forma nosso servidor terá 3 principais itens:

- 
- **Definições de tipo:** Modelar dados com schemas;
  - **Resolvers:** Definir como buscar e alterar os dados;
  - **Models:** Definir dados e abrir interface para dados no banco.



## 06.2 Servidor

Na pasta **backend**, crie uma nova pasta chamada **src**.

Em **backend/src** crie um arquivo **index.js** e adicione o código a seguir:

Caminho: **backend/src/index.js**

---



## 06.2 Servidor

```
const { ApolloServer } =  
require("apollo-server")  
  
const typeDefs =  
require("./typeDefs")  
  
const resolvers =  
require("./resolvers")  
  
const server = new ApolloServer({  
  typeDefs, resolvers })  
  
server  
  .listen()  
  .then(({ url }) =>  
    console.log(`Server rodando na  
    ${url}`))  
  .catch(error => console.log("Falha:  
    ", error))
```

---

## 06.2 Definições de Tipos

Nessa parte vamos criar as definições de tipo. As definições de tipo consistem de 3 items:

- **Types:** Modelar dados com schemas
- **Query:** Definir queries e associar aos tipos criados
- **Mutation:** Definir mutation e associar aos tipos criados

Para criarmos as definições de tipo crie uma pasta chamada **typeDefs em backend/src** e crie os arquivos de acordo com a estrutura a seguir:

**Caminho:** backend/src

```
typeDefs
├── index.js
├── mutation.js
├── query.js
└── types.js
```



## 06.3 Types

Vamos começar criando os tipos (**types**) de dados que teremos na API GraphQL. É nessa parte que se modela os dados.



Em **typeDefs/types.js** cole o código a seguir:

---

**Caminho:**

backend/src/typeDefs/types.js

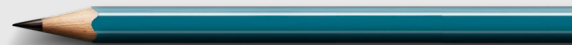




## 06.3 Types

```
const { gql } = require("apollo-server")
```

```
const types = gql`  
  type Fruit {  
    id: ID!  
    name: String  
    nutritions: Nutritions  
  }  
  
  type Nutritions {  
    calories: String  
    sugar: String  
  }  
`
```



---

```
module.exports = types
```



## 06.3 Types

Teremos apenas dois tipos de dados, um para frutas e uma para nutrições que incluem algumas informações sobre a fruta.



E como pode perceber o tipo **Nutrititions** na verdade é **usado dentro do tipo Fruit** de forma combinada. GraphQL permite fazer combinações de tipos e isso resulta em uma resposta unificada com os dados num mesmo JSON:



## 06.3 Types

```
{
```

```
  "id": "123",
```

```
  "name": "Banana",
```

```
  "nutritions": {
```

```
    "calories": "96",
```

```
    "sugar": "17.2"
```

```
  }
```

```
}
```



---



## 06.4 Query

Agora vamos montar as queries. Para configurar uma query basta dar um nome e atribuir a um type. Aqui, teremos duas queries: uma que retorna um array com todas as frutas e uma que retorna uma fruta a partir de seu id.

---

Em **typeDefs/query.js** cole o código:

**Caminho:**

backend/src/typeDefs/query.js



## 06.4 Query

```
const { gql } =
```

```
require("apollo-server")
```

```
const query = gql`
```

```
  type Query {
```

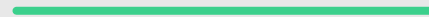
```
    fruits: [Fruit]
```

```
    fruit(id: ID!): Fruit
```

```
  }
```

```
`
```

```
module.exports = query
```





## 06.4 Query

Pronto as queries estão definidas.

No trecho de código acima dissemos ao servidor que a query chamada `fruits` deve retornar um array com dados do tipo **Fruit** (definido no passo anterior) e a query chamada `fruit` deve ter um parâmetro chamado `id` com um dado do tipo `ID` (já existente em GraphQL) e retornar um item do tipo `Fruit`.

A exclamação **(!)** ao final do tipo `ID` define o campo como **obrigatório**.



## 06.5 Mutation

Agora resta definir as mutations. Essa parte é bem parecida com as definições de queries. Mutations são queries que fazem alteração nos dados.

Em **typeDefs/mutation.js** cole o seguinte código:

**Caminho:**

backend/src/typeDefs/mutation.js





## 06.5 Mutation

```
const { gql } = require("apollo-server")

const mutation = gql`

  type Mutation {
    createFruit(fruit: FruitInput): Fruit
    updateFruit(id: String, fruit: FruitInput): Fruit
    deleteFruit(id: String): Fruit
  }

  input FruitInput {
    name: String!
    nutritions: NutritionsInput!
  }

  input NutritionsInput {
    sugar: String!
    calories: String!
  }
`

module.exports = mutation
```



## 06.5 Mutation

Para concluir, no arquivo **typeDefs/index.js** cole o seguinte código:

**Caminho:** backend/src/typeDefs/index.js

```
const query = require("./query")
const mutation = require("./mutation")
const types = require("./types")

const typeDefs = [query, mutation,
types]

module.exports = typeDefs
```



## 06.6 Models

Antes de definir os **resolvers**, vamos criar os **models** para que possamos modelar os dados ao banco e também usar como interface para conectar a eles. Na verdade teremos apenas uma model: a de frutas.

Dentro da pasta src crie uma pasta chamado models.

Dentro dela crie um arquivo Fruit.js.

Neste arquivo cole o código:

**Caminho:**

backend/src/models/Fruit.js



## 06.6 Models

```
const mongoose = require("mongoose")
```

```
const FruitSchema = mongoose.Schema({  
  name: String,  
  nutritions: {  
    sugar: String,  
    calories: String,  
  },  
})
```

```
module.exports =
```

```
mongoose.model("Fruit", FruitSchema)
```



## 06.7 Resolvers

**Resolvers** é onde associamos quais ação queries e mutations devem tomar. No nosso caso as ações são fazer chamadas ao banco de dados. Desta maneira, usamos **models** para implementar funções do banco de dados uma vez que são as models que fazem a interface com banco.

Dentro da pasta src crie uma pasta chamado resolvers.

Dentro dela crie um arquivo fruitResolver.js. Neste arquivo cole o seguinte código:


**Caminho:**

backend/src/resolvers/fruitResolver.js



## 06.7 Resolvers

```
const Fruit = require("../models/Fruit")
const fruitResolver = {
  Query: {
    fruits() {
      return Fruit.find()
    },
    fruit(_, { id }) {
      return Fruit.findById(id)
    },
  },
  Mutation: {
    createFruit(_, { fruit }) {
      const newFruit = new Fruit(fruit)
      return newFruit.save()
    },
    updateFruit(_, { id, fruit }) {
      return Fruit.findByIdAndUpdate(id, fruit, { new: true })
    },
    deleteFruit(_, { id }) {
      return Fruit.findByIdAndRemove(id)
    },
  },
}
module.exports = fruitResolver
```



---



## 06.8 Conectando ao Banco

Para conectar ao banco usaremos **DotEnv** para armazenar as credenciais do banco. Essa biblioteca permite criar variáveis de ambiente onde podemos ter as credenciais do banco ao invés de tê-las no código o que comprometeria as informações deixando disponíveis a qualquer pessoas com acesso ao código, por exemplo, num repositório git.

Dentro da pasta crie um arquivo backend crie um arquivo chamado .env:

**Caminho:** backend/.env



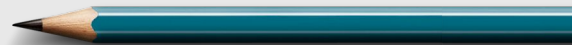
## 06.8 Conectando ao Banco

```
DB_HOST=localhost:27017
```

```
DB_USER=seu_nome
```

```
DB_PASS=sua_senha
```

```
DB_NAME=fruits
```



---





## 06.8 Conectando ao Banco

Agora modifique o arquivo `src/index.js` para adicionar a conexão ao banco:

**Caminho:**

`backend/src/index.js`





## 06.8 Conectando ao Banco

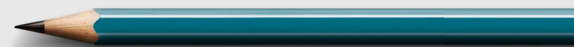
```
require("dotenv").config()
const mongoose = require("mongoose")
const { ApolloServer } = require("apollo-server")
const typeDefs = require("./typeDefs")
const resolvers = require("./resolvers")
// Database
const db = {
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  pass: process.env.DB_PASS,
  name: process.env.DB_NAME,
}
const dbUri =
`mongodb+srv://${db.user}:${db.pass}@${db.host}/${db.name}?retryWrites=true &w
=majority`
const dbOptions = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useFindAndModify: false,
}
mongoose
  .connect(dbUri, dbOptions)
  .then(() => console.log("Database connected"))
  .catch(error => console.log("Database failed: ", error))

// GraphQL
const server = new ApolloServer({ typeDefs, resolvers })
server
  .listen()
  .then(({ url }) => console.log(`Server ready at ${url}`))
  .catch(error => console.log("Server failed: ", error))
```



## 06.9 Testando a API

Para conseguir iniciar o servidor a partir do **npm**, abra o arquivo `package.json` e insira o seguinte script:



**Caminho:** backend/package.json

---

```
{  
  ...  
  "scripts": {  
    "start": "node src/index.js"  
  }  
  ...  
}
```



## 06.9 Testando a API

No terminal, navegue até o diretório backend e inicie o servidor:




```
npm start
```

---

## 06.10 Executando o Queries

Como ainda não existe nenhum dado no banco podemos começar testando a mutation de criação de frutas:

```
mutation createFruit {  
  createFruit(  
    fruit: {  
      name: "Maçã"  
      nutritions: { sugar: "2.3", calories:  
"52" }  
    }  
  ) {  
    name  
    nutritions {  
      sugar  
      calories  
    }  
  }  
}
```

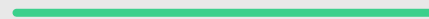
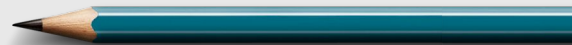


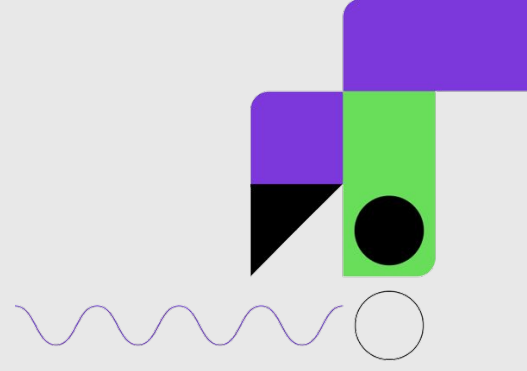


## 06.10 Executando o Queries

Depois buscar com a query de busca de frutas:

```
query getFruits {  
  fruits {  
    id  
    name  
    nutritions {  
      sugar  
      calories  
    }  
  }  
}
```



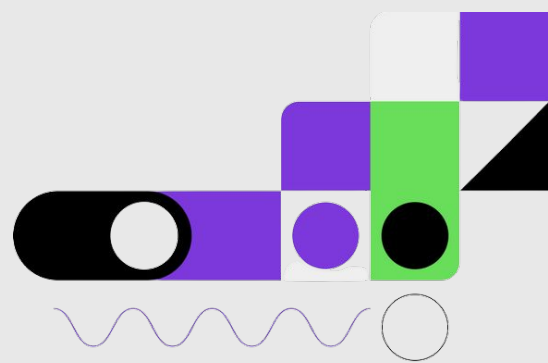


# 07. Projeto (Front-End)



## 07.1 Iniciando

A proposta é um website sobre frutas onde podemos gerenciar os dados fazendo as operações CRUD. O site será feito em React.js e o servidor em Node.js. Nesse tutorial desenvolvemos o frontend em **React.js**.





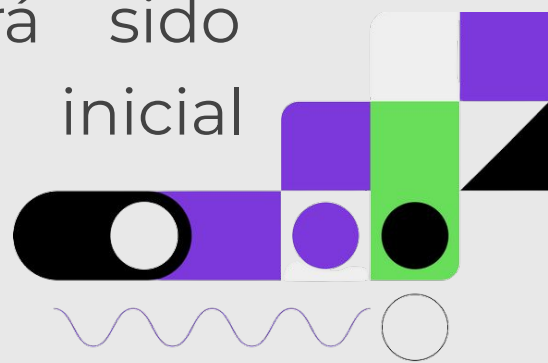
## 07.1 Iniciando

Aqui devemos continuar dentro da pasta **fruits** de onde começamos no tutorial anterior. Dentro dela, execute o seguinte comando para iniciar um projeto react:

```
npx create-react-app frontend
```

Quando terminado o processo, uma pasta frontend terá sido criada com a aplicação inicial React.js:

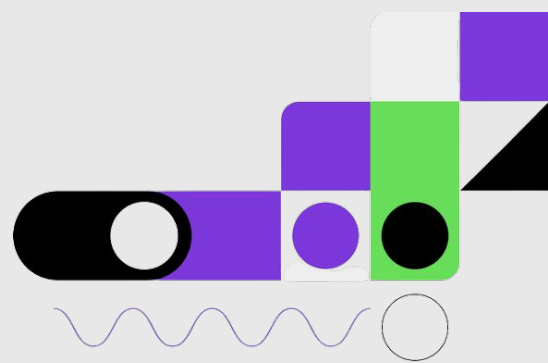
```
📦 fruits
├── 📁 backend
├── 📁 frontend
└── ...
```



## 07.1 Iniciando

Abra um terminal de comandos e navegue para a pasta fruits/frontend. Verifique que funcionou executando:

```
npm start
```

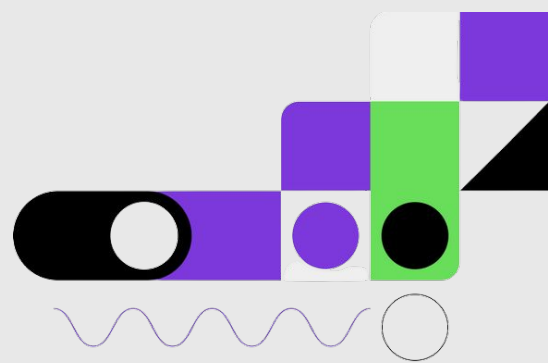


## 07.1 Iniciando

Começando na pasta public, abra index.html e deixe dessa maneira:

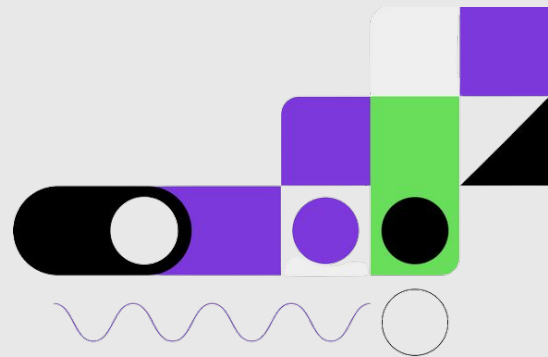
**Caminho:**

frontend/public/index.html



# 07.1 Iniciando

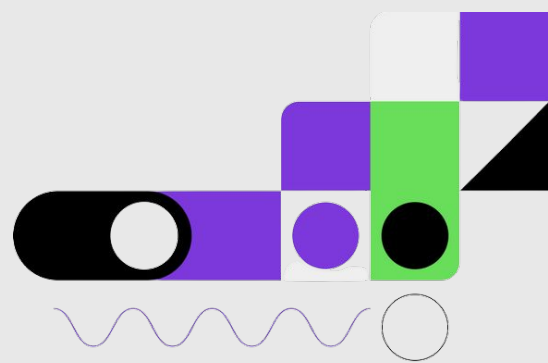
```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8" />
    <link rel="icon"
href="%PUBLIC_URL%/favicon.ico " />
    <meta
      name="viewport "
      content="width=device-width,
initial-scale=1 "
    />
    <meta
      name="description "
      content="Um app sobre informações
nutricionais de frutas. "
    />
    <title>Frutas</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this
app.
    </noscript>
    <div id="root"></div>
  </body>
</html>
```



## 07.1 Iniciando

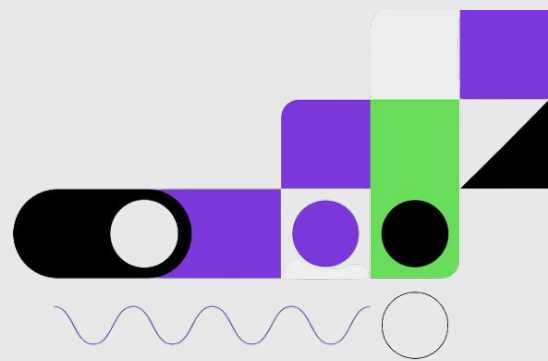
Agora, vamos adicionar os estilos que serão usado nesta aplicação. Na pasta src, substitua os conteúdos de index.css e App.css com os seguintes conteúdos:

**Caminho:** frontend/src/index.css



# 07.1 Iniciando

```
body {  
  margin: 0;  
  font-family: -apple-system, BlinkMacSystemFont, "Segoe  
UI",  
  "Roboto", "Oxygen", "Ubuntu", "Cantarell", "Fira Sans",  
  "Droid Sans", "Helvetica Neue", sans-serif;  
  -webkit-font-smoothing: antialiased;  
  -moz-osx-font-smoothing: grayscale;  
}  
input,  
button {  
  padding: 10px;  
  font-size: calc(10px + 1vmin);  
}  
button:hover {  
  cursor: pointer;  
}  
ul {  
  list-style: none;  
  margin: 20px 0;  
  padding: 0;  
}  
li {  
  display: flex;  
  justify-content: space-between;  
  align-items: baseline;  
  padding: 10px;  
  margin: 10px;  
}
```



# 07.1 Iniciando

**Caminho:** frontend/src/App.css

```
.App {
  text-align: center;
}

.App-header {
  background-color: #282c34;
  color: white;
  position: absolute;
  top: 10%;
  right: 0;
  width: 100vw;
}

.App-header h1 {
  margin: 0;
  padding: 20px;
}

.App-body {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-viewbox {
  position: relative;
}

.App-close-btn {
  position: absolute;
  top: -100px;
  right: -100px;
}

.App-close-btn button {
  background: none;
  border: 0;
  color: white;
  font-size: calc(10px + 2vmin);
}

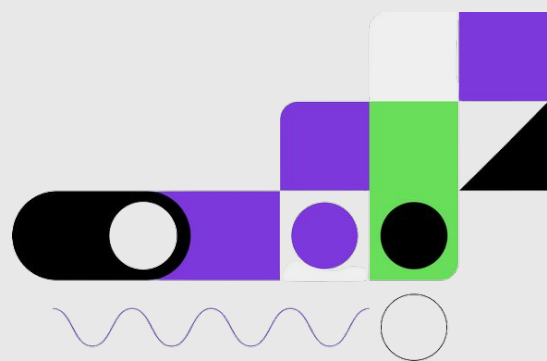
.App-btn {
  max-width: 120px;
  width: 100%;
}

.App-btn.secondary {
  background: transparent;
  border: 2px solid white;
  color: white;
}

.App-item-actions {
  margin-left: 40px;
}

.App-item-actions a {
  margin: 0 10px;
  background: none;
  text-decoration: none;
}

.App-item-actions a:hover {
  cursor: pointer;
}
```



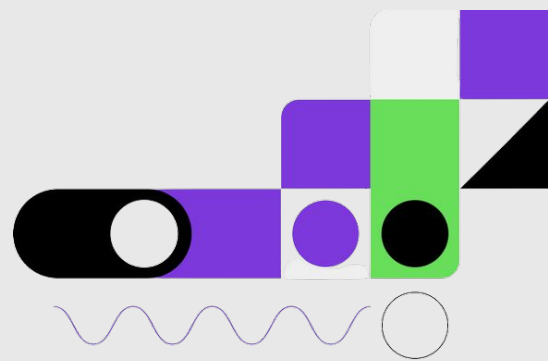
## 07.1 Iniciando

Estilos adicionados. Agora vamos a pasta `index.js` dentro de `src` e certificar que o arquivo está como a seguir:

**Caminho:** `frontend/src/index.js`

```
import React from "react"
import ReactDOM from "react-dom"
import "./index.css"
import App from "./App"

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
)
```





## 07.1 Iniciando

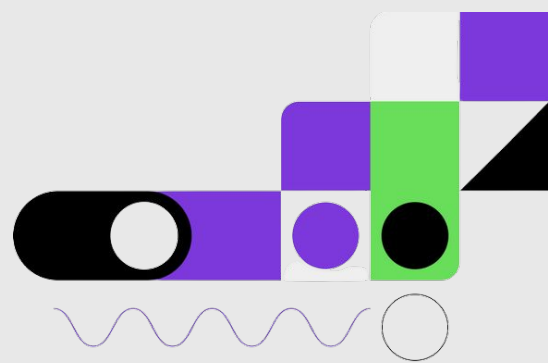
E agora, o último arquivo a ser checado antes de começarmos com a aplicação. Deixe `src/App.js` da seguinte maneira:

**Caminho:** `frontend/src/App.js`

```
import React from "react"
import "../App.css"

function App() {
  return (
    <div className="App">
      <div className="App-header">
        <h1>Frutas</h1>
      </div>
      <div className="App-body"></div>
    </div>
  )
}

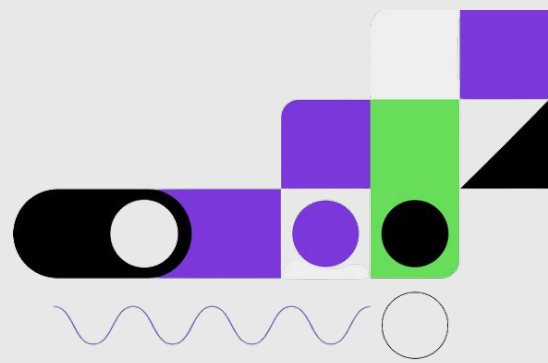
export default App
```



## 07.2 Configurando Rotas

Para facilitar a navegação entre rotas, vamos usar a biblioteca **React router**. Instale-a com o comando:

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



## 07.2 Configurando Rotas

Dentro da pasta src crie um arquivo chamado routes.js e inicie as rotas dessa maneira:

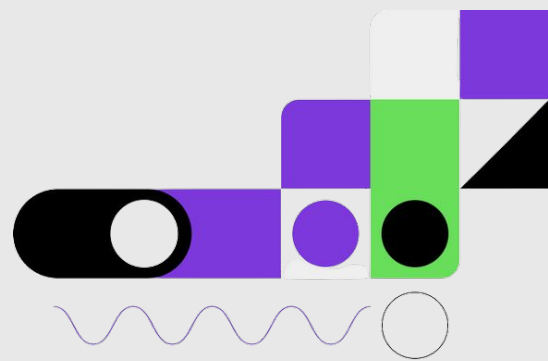
**Caminho:** frontend/src/routes.js

```
import React from "react"
import {
  BrowserRouter as Router,
  Switch,
  Route,
} from "react-router-dom"
```

```
import Fruits from "../components/Fruits"
```

```
const Routes = () => (
  <Router>
    <Switch>
      <Route exact path="/">
        <Fruits />
      </Route>
    </Switch>
  </Router>
)
```

```
export default Routes
```



## 07.2 Configurando Rotas

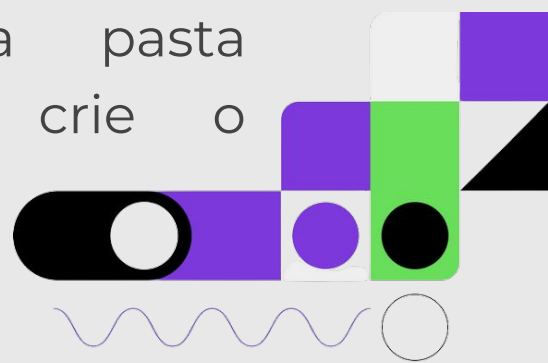
A propriedade `path` indica em qual caminho da aplicação aquele componente será exibido, no caso de `Fruits`, este será exibido na home da aplicação.

Agora, vamos criar o componente `Fruits.js` que está sendo chamado no arquivo de rotas. Esse componente mostrará uma lista de frutas assim como as ações de exibir, editar e excluir de cada fruta.

Dentro de `src`, crie uma pasta `components`. Dentro desta, crie o componente de frutas:

### **Caminho:**

`frontend/src/components/Fruits.js`



## 07.2 Configurando Rotas

```
import React from "react"
import { Link } from "react-router-dom"
const FruitsList = () => {
  return (
    <>
      <ul>
        <li>
          <span>Banana</span>
          <div className="App-item-actions">
            <Link>
              <span role="img" aria-label="visualizar">

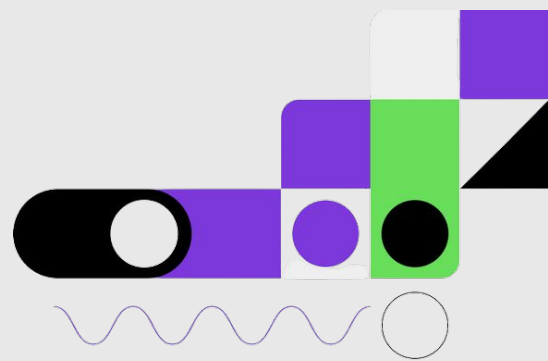
              </span>
            </Link>
            <Link>
              <span role="img" aria-label="editar">

              </span>
            </Link>
            <Link>
              <span role="img" aria-label="excluir">

              </span>
            </Link>
          </div>
        </li>
      </ul>

      <p>
        <Link>
          <button>Nova Fruta</button>
        </Link>
      </p>
    </>
  )
}
```

export default FruitsList



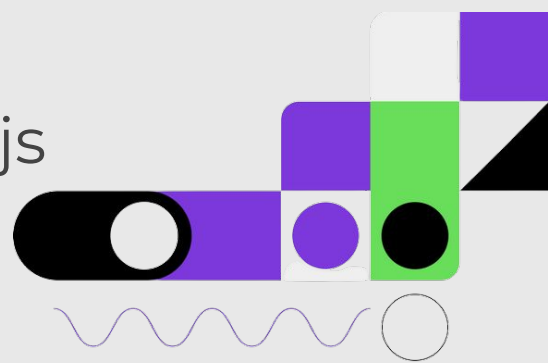
## 07.2 Configurando Rotas

Por enquanto adicionamos uma lista com apenas uma fruta.

Também criamos Link ao redor dos botões, mas não apontamos para nenhuma rota, nesse momento. Faremos isso mais a frente.

Agora, vá até App.js e inclua a rota criada:

**Caminho:** frontend/src/App.js



## 07.2 Configurando Rotas

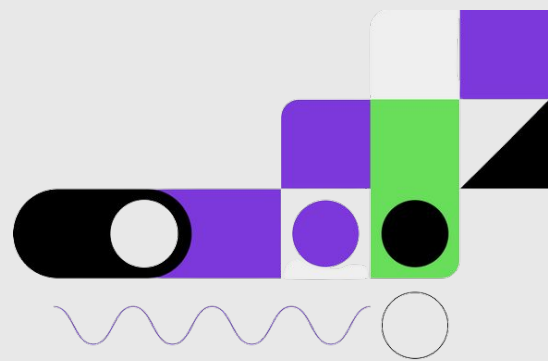
```
import React from "react"

import "./App.css"

import Routes from "./routes"

function App() {
  return (
    <div className="App">
      <div className="App-header">
        <h1>Frutas</h1>
      </div>
      <div className="App-body">
        <Routes />
      </div>
    </div>
  )
}

export default App
```

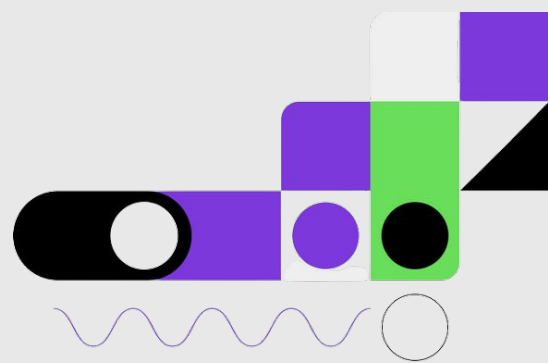


## 07.3 Conectando à API

Vamos começar instalando as dependências para usar apollo client.

**Nota:** Aqui estamos usando apollo client na versão 3.

```
npm i @apollo/client graphql
```

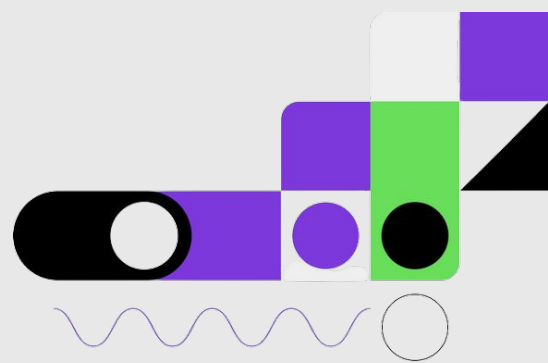




## 07.3 Conectando à API

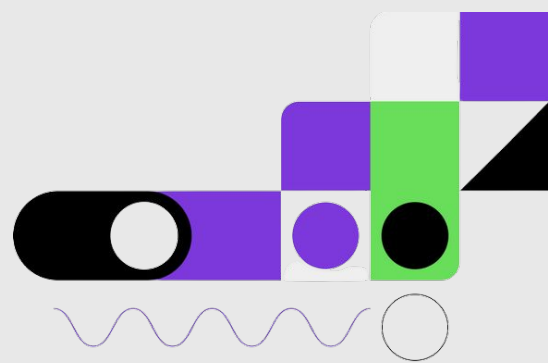
Agora, efetuamos a conexão usando a URL da API no backend. Como estamos desenvolvendo tudo localmente, vamos fornecer a URL local do backend que serve na porta 4000.

**Caminho:** frontend/src/App.js



## 07.3 Conectando à API

```
import React from "react"
import {
  ApolloProvider,
  ApolloClient,
  InMemoryCache,
} from "@apollo/client"
import "./App.css"
import Routes from "./routes"
const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache(),
})
function App() {
  return (
    <ApolloProvider client={client}>
      <div className="App">
        <div className="App-header">
          <h1>Frutas</h1>
        </div>
        <div className="App-body">
          <Routes />
        </div>
      </div>
    </ApolloProvider>
  )
}
export default App
```

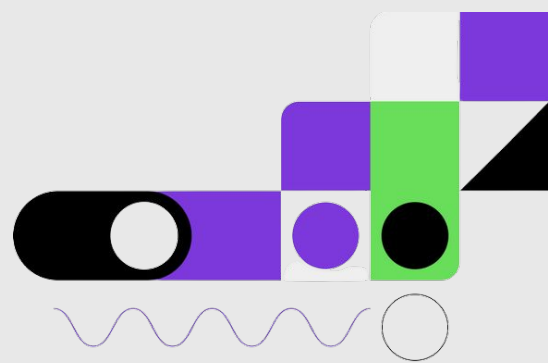


## 07.3 Conectando à API

Agora vamos voltar ao componente `Fruits.js` e popular o componente com dados vindos da API usando o Apollo client.

### **Caminho:**

`frontend/src/components/Fruits.js`



## 07.3 Conectando à API

```
import React from "react"
import { gql, useQuery } from "@apollo/client"
import { Link } from "react-router-dom"
export const GET_FRUITS = gql`
{
  fruits {
    id
    name
  }
}
`

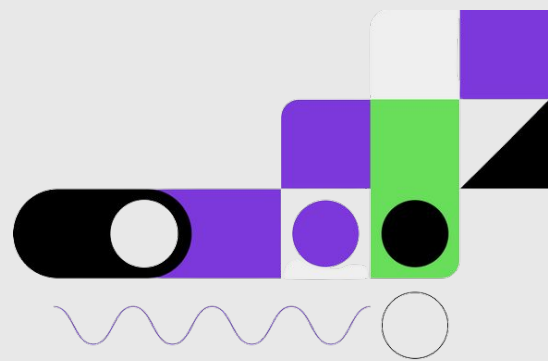
const FruitsList = () => {
  const { loading, error, data } = useQuery(GET_FRUITS)
  if (loading) return <p>Loading...</p>
  if (error) return <p>Error :(</p>
  return (
    <>
      <ul>
        {data.fruits &&
          data.fruits.map(({ name, id }) => (
            <li key={id}>
              <span>{name}</span>
              <div className="App-item-actions">
                <Link to={`/${fruit}/${id}`}>
                  <span role="img" aria-label="visualizar">

                  </span>
                </Link>
                <Link to={`/${editFruit}/${id}`}>
                  <span role="img" aria-label="editar">

                  </span>
                </Link>
                <Link to={`/${deleteFruit}/${id}`}>
                  <span role="img" aria-label="excluir">

                  </span>
                </Link>
              </div>
            </li>
          )))
      </ul>
      <p>
        <Link to="/createFruit">
          <button>Nova Fruta</button>
        </Link>
      </p>
    </>
  )
}

export default FruitsList
```



## 07.3 Conectando à API

E simples assim, fizemos a query e populamos o componente com dados da API. Ainda fizemos um retorno simples ao usuário com feedback de loading e de erro, caso ocorra algum.

Além disso, de antemão, apontamos rotas para cada ação CRUD relacionada à frutas. Vamos, agora, criar os componentes para cada ação para depois conectar cada rota à seu respectivo componente.

## 07.4 Crud-Create

**Caminho:** frontend/src/components/CreateFruit.js

# 07.5 Crud-Read

Agora, criaremos o componente de visualização.

**Caminho:** frontend/src/components/Fruit.js

```
import React from "react"
import { gql, useQuery } from "@apollo/client"
import { useParams, Link } from "react-router-dom"

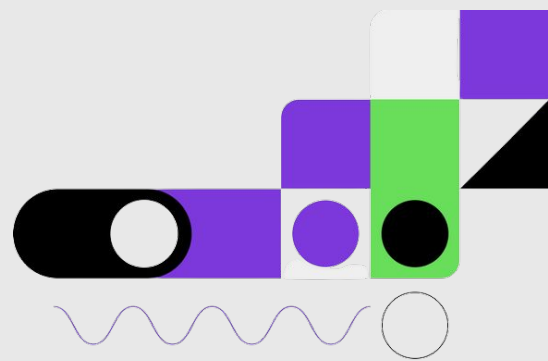
export const GET_FRUIT_BY_ID = gql`
  query GetFruit($id: ID!) {
    fruit(id: $id) {
      id
      name
      nutritions {
        sugar
        calories
      }
    }
  }
`

const Fruit = () => {
  const { id } = useParams()
  const { loading, error, data } = useQuery(GET_FRUIT_BY_ID, {
    variables: { id },
  })

  if (loading) return <p>Loading...</p>
  if (error) return <p>Error :(</p>

  return (
    <div className="App-viewbox">
      <p>
        <strong>Fruta: </strong>
        {data.fruit.name}
      </p>
      <p>
        <strong>Açúcar: </strong>
        {data.fruit.nutritions.sugar}g
      </p>
      <p>
        <strong>Calorias: </strong>
        {data.fruit.nutritions.calories}kcal
      </p>
      <p className="App-close-btr">
        <Link to="/">
          <button>✕</button>
        </Link>
      </p>
      <p>
        <Link to={` /editFruit/${id}`}>
          <button>Editar</button>
        </Link>
      </p>
    </div>
  )
}

export default Fruit
```



# 07.6 Crud-Update

E, para o componente de edição:

**Caminho:** frontend/src/components/EditFruit.js

```
import React from "react"
import { gql, useQuery, useMutation } from "@apollo/client"
import { useParams, Link, useHistory } from "react-router-dom"
import { GET_FRUIT_BY_ID } from "../Fruit"

const UPDATE_FRUIT = gql`
  mutation UpdateFruit(
    $id: String!
    $name: String
    $sugar: String
    $calories: String
  ) {
    updateFruit(
      id: $id
      fruit: {
        name: $name
        nutritions: { sugar: $sugar, calories: $calories }
      }
    ) {
      id
      name
      nutritions {
        calories
        sugar
      }
    }
  }
`

const EditFruit = () => {
  const { id } = useParams()
  const history = useHistory()

  const { loading, error, data } = useQuery(GET_FRUIT_BY_ID, {
    variables: { id },
  })

  const [updateFruit, { error: mutationError }] = useMutation(
    UPDATE_FRUIT,
    {
      onCompleted() {
        history.push('/')
      },
    }
  )

  if (loading) return <p>Loading...</p>
  if (error || mutationError) return <p>Error :(</p>

  let nameInput
  let sugarInput
  let caloriesInput

  return (
    <div>
      <form
        className="App-viewbox"
        onSubmit={e => {
          e.preventDefault()

```

```
      updateFruit({
        variables: {
          id: data.fruit.id,
          name: nameInput.value,
          sugar: sugarInput.value,
          calories: caloriesInput.value,
        },
      })
    }
  }
  >
  <p>
    <label>
      Fruta
    <br />
    <input
      type="text"
      name="name"
      defaultValue={data.fruit.name}
      ref={node => {
        nameInput = node
      }}
    />
  </label>
</p>
  <p>
    <label>
      Açucar (g)
    <br />
    <input
      type="text"
      name="sugar"
      defaultValue={data.fruit.nutritions.sugar}
      ref={node => {
        sugarInput = node
      }}
    />
  </label>
</p>
  <p>
    <label>
      Calorias
    <br />
    <input
      type="text"
      name="calories"
      defaultValue={data.fruit.nutritions.calories}
      ref={node => {
        caloriesInput = node
      }}
    />
  </label>
</p>
  <p className="App-close-btn">
    <Link to="/">
      <button type="button">✖</button>
    </Link>
  </p>
  <p>
    <button className="App-btn" type="submit">
      Salvar
    </button>
  </p>
</form>
</div>
)
}

export default EditFruit
```



# 07.7 Crud-Delete

E, pra finalizar, criaremos o componente de deleção de fruta.

**Caminho:** frontend/src/components/DeleteFruit.js

```
import React from "react"
import { gql, useQuery, useMutation } from
"@apollo/client "
import { useParams, Link, useHistory } from
"react-router-dom "
import { GET_FRUITS } from "../Fruits"
import { GET_FRUIT_BY_ID } from "../Fruit"

const DELETE_FRUIT = gql`
  mutation DeleteFruit($id: String) {
    deleteFruit(id: $id) {
      id
      name
      nutritions {
        calories
        sugar
      }
    }
  }
`

const DeleteFruit = () => {
  const history = useHistory()
  const { id } = useParams()

  const { loading, error, data } = useQuery(GET_FRUIT_BY_ID, {
    variables: { id },
  })

  const [deleteFruit, { error: mutationError }] =
    useMutation(
      DELETE_FRUIT,
      {
        update(cache) {
          const { fruits } = cache.readQuery({ query:
            GET_FRUITS })

          const deletedIndex = fruits.findIndex(
            fruit => fruit.id === id
          )

          const updatedCache = [
            ...fruits.slice(0, deletedIndex),
            ...fruits.slice(deletedIndex + 1, fruits.length),
          ]

          cache.writeQuery({
            query: GET_FRUITS,
            data: {
              fruits: updatedCache,
            },
          })
        },
        onCompleted() {
          history.push(`/${id}`)
        },
      },
    )

  if (loading) return <p>Loading...</p>
  if (error || mutationError) return <p>Error :(</p>
```

```
    return (
      <div>
        <form
          className="App-viewbox "
          onSubmit={e => {
            e.preventDefault()

            deleteFruit({
              variables: { id },
            })
          }}
        >
          <p>
            Excluir <strong>{data.fruit.name}</strong>?
          </p>
          <p className="App-close-btn ">
            <Link to="/">
              <button>✖</button>
            </Link>
          </p>
          <p>
            <button className="App-btn " type="submit ">
              Excluir
            </button>
          </p>
        </form>
      </div>
    )
  }

  export default DeleteFruit
```

# 07.8 Ligando Rotas

Agora para finalizar, ligamos cada rota à seu componente.

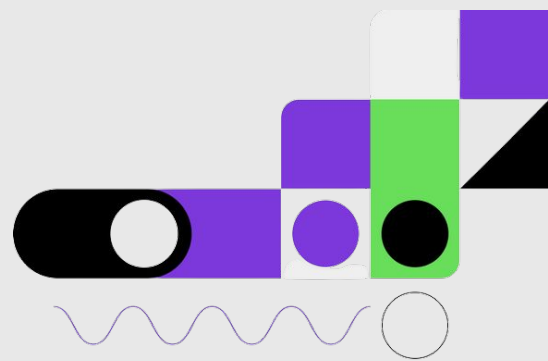
**Caminho:** frontend/src/routes.js

```
import React from "react"
import {
  BrowserRouter as Router,
  Switch,
  Route,
} from "react-router-dom"

import Fruits from "../components/Fruits"
import Fruit from "../components/Fruit"
import CreateFruit from "../components/CreateFruit"
import EditFruit from "../components/EditFruit"
import DeleteFruit from "../components/DeleteFruit"

const Routes = () => (
  <Router>
    <Switch>
      <Route exact path="/">
        <Fruits />
      </Route>
      <Route path="/fruit/:id">
        <Fruit />
      </Route>
      <Route path="/createFruit">
        <CreateFruit />
      </Route>
      <Route path="/editFruit/:id">
        <EditFruit />
      </Route>
      <Route path="/deleteFruit/:id">
        <DeleteFruit />
      </Route>
    </Switch>
  </Router>
)

export default Routes
```





# Fechamento

Vimos diversos assuntos até aqui, ainda há muito a ser abordado. Esse material pode servir como consulta e guia de estudos.

Até a próxima pessoal!