Next Level Week #3 - WhySoFast

Repositório : GitHUB - https://github.com/whysofast/happy-nextlevelweek3

https://www.figma.com/file/3egzBgeJy5iNerujSFUCCr/Happy-Web-(Copy)?node-id=0%3A1

https://www.figma.com/file/0hODG8g0zPyIZeZvCa3ev2/Happy-Mobile-(Copy)

- Por que React?

  SPA - Single Page Application

- `yarn create react-app web -- template typescript`
  - `yarn start`
- Por que typescript ?

```
import React from "react";

interface TitleProps {
  text: string;
}

function Title(props: TitleProps) {
  return <h1>{props.text}</h1>;
}

function App() {
  return (
    <div className="App">
      <Title text="Oi, eu sou o Goku !" />
    </div>
  );
}

export default App;
```

▼ Criando a home Page - Landing.tsx

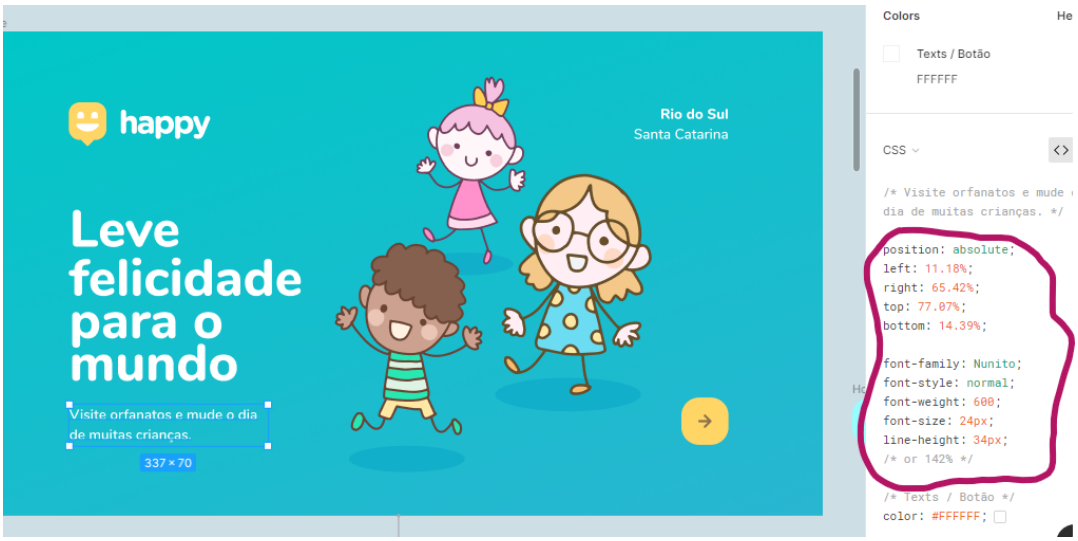
- Na importaçao do css, será feito pelo index.tsx ou App.tsx e não feito pelo HTML.

```
#page-landing {
    width: 100vw;
    height: 100vh;
  background: linear-gradient(329.54deg, #29b6d1 0%, #00c7c7 100%);
}

  display: flex;
  /* Alinha horizontal e verticalmente */
  justify-content: center;
  align-items: center;
}

#page-landing .content-wrapper {
  /* Pra servir de referencia para outros elementos */
```

```
  position: relative;
  width: 100%;
  max-width: 1100px;

  height: 100%;
  max-height: 600px;

  display: flex;
  align-items: flex-start;
  /* column pq os elementos ficam arrumados um em cima do outro, verticalmente */
  flex-direction: column;
  /* Joga um pra cima, um pro meio e outro pra baixo, espalha o conteudo igulamente */
  justify-content: space-between;

  /* no-repeat serve para não ficar aparecendo a imagem varias vezes */
  /* 80% é a posição no eixo x */
  /* center é a posição no eixo y */
  background: url(../../images/landing.svg) no-repeat 80% center;
  /* Para garantir que o background não vai ficar cortado em algum lugar; */
  background-size: contain;
}

#page-landing .content-wrapper main {
  /* Para forçar a quebra de linha dentro do main, que serve pro <h1> e pro <p> */
  max-width: 350px;
}

#page-landing .content-wrapper main h1 {
  font-size: 76px;
  font-weight: 900;
  /* Espaçamento entre linhas */
  line-height: 70px;
}

#page-landing .content-wrapper main p {
  margin-top: 40px;
  font-size: 24px;
  line-height: 34px;
}

/* Joga a location pra cima direita */
.content-wrapper .location {
  position: absolute;
  right: 0;
  top: 0;

font-size: 24px;
  line-height: 34px;

  display: flex;
  flex-direction: column;

  text-align: right;
}

/* Joga a seta pra baixo direita */
.content-wrapper .enter-app {
  position: absolute;
  right: 0;
  bottom: 0;

width: 80px;
  height: 80px;
  background: #ffd666;
  border-radius: 30px;

  display: flex;
  align-items: center;
  /* Alinha todo mundo centralizado um por cima do ioutro */
  justify-content: center;

  transition: background-color 0.2s;
```

```
}

.content-wrapper .enter-app:hover {
  background: #96feff;
}
```

Boa parte desses valores vieram do próprio figma



Para inserir icones, `yarn add react-icons`, que tem várias bibliotecas tipo o FontAwesome(`fa`), MaterialDesign(`md`), FeatherIcons(`fi` → que sera utilizado)

`import {FiArrowRight} from 'react-icons/fi'`

<FiArrowRight size={26} color="rgba(0,0,0,0.6)" />

▼ react-router-dom → Gerenciar rotas

yarn react-router-dom

novo arquivo de rotas criado como um componente → routes.tsx

Num primeiro import {} from 'react-router-dom', o typescript vai reclamar pois ele ainda n conhece as tipagens desta biblioteca. Dessa forma, devemos `yarn add @types/react-router-dom -D`

import { BrowserRouter, Switch, Route } from "react-router-dom";

- o BrowserRouter precisa estar envolta de todas as rotas criadas
  - e pra cada rota, é interessante ter-se uma "page" como componente

```
<BrowserRouter>
      <Route path="/" component={Landing} />
</BrowserRouter>
```

Na rota root ("/"), o componente Landing será acessado. Portanto, sendo o Landing um "página componente" :

```
import React from "react";
import { FiArrowRight } from "react-icons/fi";
```

```
import "../styles/global.css";
import "../styles/pages/landing.css";
import logoImg from "../images/Logo.svg";


function Landing() {
  return (
    <div id="page-landing">
      <div className="content-wrapper">
        <img src={logoImg} alt="Happy" />

        <main>
          <h1>Leve felicidade para o mundo</h1>
          <p>Visite orfanatos e mude o dia de muitas crianças.</p>
        </main>

        <div className="location">
          <strong>Salvador</strong>
          <span>Bahia</span>
        </div>

        <a href="" className="enter-app">
          <FiArrowRight size={26} color="rgba(0,0,0,0.6)" />
        </a>
      </div>
    </div>
  );
}

export default Landing;
```

Ao se ter mais de uma rota no BrowserRoutes

```
<BrowserRouter>
      <Route path="/" component={Landing} />
      <Route path="/app" component={OrphanagesMap} />
</BrowserRouter>
```

o react renderiza ambos ao mesmo tempo, pois ele não verifica igualdade e sim se os caminhos estão presentes. locahost:3000 / app, contem o "/" e contem o "app", então isso acontece :

Para que apenas uma página seja renderizada, utiliza-se o "exact" como parametro do Route

```
<BrowserRouter>
      <Route path="/" exact component={Landing} />
      <Route path="/app" component={OrphanagesMap} />
</BrowserRouter>
```

Já o **Switch**, faz com que apenas uma rota seja renderizada por tela

Se o botão com o <a> tiver um href="localhost:3000/app" do jeito que está, a aplicação será totalmente carregada do 0. Para reaproveitar, utiliza-se o **Link** do react-router-dom

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

E onde havia o anchor tag

```
<a href="" className="enter-app">
          <FiArrowRight size={26} color="rgba(0,0,0,0.6)" />
</a>
```

Será substituido por

```
<Link to="/app" className="enter-app">
          <FiArrowRight size={26} color="rgba(0,0,0,0.6)" />
</Link>
```

A diferença, pra uma aplicação simples, desse tamanho é de 2.6 mb de recursos contra 178 Bites. Tremenda.

▼ Criando a tela de mapas - OrphanagesMap.tsx

- SideBar
  - Utilizamos o <aside> </aside>

```
import React from "react";
import mapMakerImg from "../images/map-marker.svg";
import { Link } from "react-router-dom";
import { FiPlus } from "react-icons/fi";

import "../styles/pages/orphanages-map.css";

function OrphanagesMap() {
  return (
    <div id="page-map">
      <aside>
        <header>
          <img src={mapMakerImg} alt="Happy" />

          <h2>Escolha um orfanato no mapa</h2>
          <p>Muitas crianças estão esperando a sua visita :)</p>
        </header>

        <footer>
          <strong>Salvador</strong>
          <span>Bahia</span>
        </footer>
      </aside>

      <div></div>
      <Link to="" className="create-orphanage">
        <FiPlus size={32} color="#fff" />
      </Link>
    </div>
  );
}

export default OrphanagesMap;
```

```
----------> orphanages-map.css
#page-map {
  width: 100vw;
  height: 100vh;

  position: relative;
  display: flex;
}

#page-map aside {
  width: 440px;
  background: linear-gradient(329.54deg, #29b6d1 0%, #00c7c7 100%);
  padding: 80px;

  display: flex;
  flex-direction: column;
  justify-content: space-between;
}

#page-map aside h2 {
  font-size: 40px;
  font-weight: 800;
  line-height: 42px;
  margin-top: 64px;
}

#page-map aside p {
  line-height: 28px;
  margin-top: 24px;
}

#page-map aside footer {
  display: flex;
  flex-direction: column;

  line-height: 24px;
}

#page-map aside footer strong {
  font-weight: 800;
}

#page-map .create-orphanage {
  position: absolute;
  right: 40px;
  bottom: 40px;

  width: 64px;
  height: 64px;
  background: #15c3d6;
  border-radius: 20px;

  display: flex;
  justify-content: center;
  align-items: center;

  transition: background-color 0.2s;
}

#page-map .create-orphanage:hover {
  background: #17d6eb;
}
```

Para adicionar o mapa, usaremos o

`yarn add leaflet react-leaflet @types/react-leaflet`

import { Map, TileLayer} from 'react-leaflet'

import "leaflet/dist/leaflet.css"

Para renderizar um mapa:

```
<Map
  center={[-12.9890498, -38.4630524]}
  zoom={15}
  style={{ width: "100%", height: "100%" }}
>
  <TileLayer url="https://a.tile.openstreetmap.org/{z}/{x}/{y}.png" />
</Map>
```

Esse TileLayer do openstreetmap, porém é fei bagarai.

Outro gratuito e mais bonitin é o mapbox

Para isso, é necessário guardar um token em variaveis de ambiente

REACT_APP_MAPBOX_TOKEN=ASLDFLASDFLASDFLDAS

```
 <TileLayer
          url={`https://api.mapbox.com/styles/v1/mapbox/light-v10/tiles/256/{z}/
{x}/{y}@2x?access_token=${process.env.REACT_APP_MAPBOX_TOKEN}`}
        />
```

▼ Back-end

▼ Setup inicial

Será criada uma pasta separada para o backend, no mesmo diretório da web(front-end)

Dentro da pasta, `yarn init -y` → Cria um package.json

`yarn add express @types/express`

Criar o boilerplate padrão do express:

```
import express from "express";
const app = express();

const port = 3333;
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

Porém , o node ainda não entende typescript, portanto precisamos instalá-lo como dependencia de desenvolvimento

`yarn add typescript -D`

Além disso, devemos inicializar o typescript

`yarn tsc --init`

Que cria um tsconfig.json

E nele , por enquanto, so trocamos o target para es2017

E para executar o projeto com node e typescript tambem como dependencia de desenvolvimento

`yarn add ts-node-dev -D`

Dentro do package.json, criaremos uma chave "scripts" para poder usar scripts personalizados

```
"scripts": {
    "dev": "ts-node-dev src/server.ts"
  },
```

Agora, se executarmos `yarn dev`, ele vai executar o código "ts-node-dev src/server.ts"

- O --transpile-only faz com que o typescript não verifique por erros, apenas compile o código. Deixamos isso a cargo do editor de código
- O --ignore-watch node_modules ignora mudanças na pasta node_modules, acelerando MUITO, a compilação.
  - Inclusive, se adicionarmos no settings.json do vscode um file_exclude e dentro dele o "node_modules" ele tira da visualização do vscode. nice do nice

E agora a aplicação já ta funfando 🙂

▼ Rotas, parâmetros e métodos HTTP

```
app.get("/users", (req, res) => {
  return res.send("Hello World");
});
```

Esse get funciona belezinha, tranquilo, ok. Porém como se trata de uma API REST, mandaremos um JSON ao invés de uma página. Assim sendo, faremos:

```
app.get("/users", (req, res) => {
  return res.json({ message: "Hello World" });
});
```

- Métodos HTTP
  - Get → Buscar info
    - É o método utilizado pelos navegadores.
  - Post → Criar info
  - Put → Editar info
  - Delete → Del info
- Query params
  - localhost:3333/users?search=fast&page=2
  - Para acessá-los, basta chamar `request.query`

    { search: 'fast', page: '2' }
- Route params
  - localhost:3333/users/1
    - Identifica qual usuário a ser utilizado

- `request.params` com app.get("/users**/:id**", (req, res)...

  { id: '1' }

- Body
  - Corpo da requisição
    - Informações mais complexas, vindas geralmente de forms
    - Para ler JSON bodys, precisamos do famoso:
      - app.use(express.json());
      - `req.body`

        { testando: 'Com a testa' }
-

▼ Criando o banco de dados - SQLite

`yarn add typeorm sqlite3`

Poderíamos usar o Driver nativo, o Query builder ou o **ORM**

- Criaremos uma pasta 'database' para tudo que for relacionado com o banco de dados dentro da 'src' e dentro dela, um **database.sqlite**
- Além disso, dentro da raiz um arquivo chamado **ormconfig.json**

```
{
    "type": "sqlite",
    "database": "./src/database/database.sqlite"
}
```

- Novamente, na pasta 'database', criaremos um **connection.ts**, que fará a conexão com o banco de dados.

```
import { createConnection } from "typeorm";

createConnection();
```

- E devemos importar esse connection para o nosso server.

  `import"./database/connection";`
- Agora precisamos criar nossa tabela de banco de dados
  1. Primeiramente, criamos uma pasta 'migrations' dentro da pasta database
  2. Adicionar nos scripts do package.json
     1. `"typeorm": "ts-node-dev ./node_modules/typeorm/cli.js"`
     2. sabemos que deu certo se rodarmos um "yarn typeorm" e ele exibir o cli.js na coluna de comandos
  3. No arquivo ormconfig.json, adicionamos um

     `"migrations" : [ ".src/database/migrations/*.ts" ]`

     Isso informa que todos os arquivos **.ts** na pasta serão do tipo migração

     `"cli": { "migrationsDir": "./src/database/migrations" }`

Onde indicamos onde o typeorm deve criar novas migrations

`"entities": ["./src/models/*.ts"]`

Dizer onde estão os models

4. Agora criaremos uma migration
    1. `yarn typeorm migration:create -n create_orphanages`
    2. O arquivo criado tem um timestamp seguido do nome da migration criada
    3. O método **up** realiza as alterações que desejamos dentro do bancos de dados,criar uma nova tabela, um campo, deletar algum campo
    4. O método **down** DESFAZ o que foi feito no UP, famoso ctrl+z
5. Após importar o "Table" do typeorm no arquivo de migração,faremos dentro do método UP:

```
public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.createTable(
      new Table({
        name: "orphanages",
        columns: [
          {
            name: "id", //Essa coluna vai ser gerada automaticamente
            type: "integer",
            unsigned: true, //não pode ser um negativo
            isPrimary: true, //Primary key
            isGenerated: true, //Essa coluna é gerada automaticamente
            generationStrategy: "increment", //1,2,3,4,5,6,7...
          },
          {
            name: "name",
            type: "varchar", // string curto de até 200 caracteres
          },
          {
            name: "latitude",
            type: "decimal",
            scale: 10, //numeros dps da virgula
            precision: 2, //numeros antes da virgula
          },
          {
            name: "longitude",
            type: "decimal",
            scale: 10,
            precision: 2,
          },
          {
            name: "about",
            type: "text",
          },
          {
            name: "instructions",
            type: "text",
          },
          {
            name: "open_on_weekends",
            type: "boolean",
            default: false,
          },
        ],
      })
    );
  }
```

6. No método DOWN, a gente vai fazer o contrário de tudo que foi feito no UP

```
public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.dropTable("orphanages");
  }
```

7. Por fim, para executar a migration e criar os bagulho,
    1. `yarn typeorm migration:run`

Para gerenciar esse sqlite, vamos utilizar o beekeeper
https://www.beekeeperstudio.io/download/?platform=portable

Para cada tabela no banco de dados haverá um model, que é a representaçção daquela tabela como uma classe dentro do código

Para isso, criaremos uma pasta "**models**" dentro da src e dentro dela, um **Orphanage.ts**

Como esquecemos de criar um dos campos na migration, faremos um revert para desfazer a caca e fazer de um jeito melhor

`yarn typeorm migration:revert`

Dentro deste Orphanage.ts, iremos exportar uma class

```
export default class Orphanage {
  id: number;

  name: string;
  latitude: number;
  longitude: number;
  about: string;
  instructions: string;
  opening_hours: string;
  open_on_weekends: boolean;
}
```

E agora que já criamos os campos , precisamos integrar com o typeorm

alterando o tsconfig.json

`"strictPropertyInitialization": false,`

`"experimentalDecorators": true,`
`"emitDecoratorMetadata": true,`

Em seguida, utilizaremos os decorators no Orphanage.ts

`import {Entity, Column, PrimaryGeneratedColumn} from 'typeorm';`

Decoraremos a classe com o @Entity ('orphanages')

o id com @PrimaryGeneratedColumn('increment')

e os demais valores, @Column()

```
import { Entity, Column, PrimaryGeneratedColumn } from "typeorm";

@Entity("orphanages")
export default class Orphanage {
  @PrimaryGeneratedColumn("increment")
  id: number;
  @Column()
```

```
  name: string;
  @Column()
  latitude: number;
  @Column()
  longitude: number;
  @Column()
  about: string;
  @Column()
  instructions: string;
  @Column()
  opening_hours: string;
  @Column()
  open_on_weekends: boolean;
}
```

Agora podemos criar rotas no server.ts

1. Como iremos criar novos orfanatos → POST

```
app.post("/orphanages", (req, res) => {
  const {
    name,
    latitude,
    longitude,
    about,
    instructions,
    opening_hours,
    open_on_weekends,
  } = req.body
  return res.json({ message: "Hello World" });
});
```

   Inicialmente somente desestruturamos os dados do body e armazenamos nas variáveis

2. Para inserirmos esses dados no banco de dados,
   1. `import {getRepository} from 'typeorm'`
   2. importar nosso model: `import Orphanage from './models/Orphanage'`
   3. Armazenar numa variável: `const orphanagesRepository = getRepository(Orphanage);`
   4. A partir desse momento, `orphanagesRepository` já possui todos os métodos do tipo create, find, findOne, delete ...

```
const orphanage = orphanagesRepository.create({
    name,
    latitude,
    longitude,
    about,
    instructions,
    opening_hours,
    open_on_weekends,
  });

  orphanagesRepository.save(orphanage);
```

   Dessa forma, então, os dados são armazenados numa variavel orphanage e em seguida, salvos atraves do orphanagesRepository.save

Porem, o save é ASSÍNCRONO, PORTANTO USAREMOS UM **await** para aguardar o salvamento e só daí prosseguir para a linha seguinte e para isso, devemos tornar a função assíncrona → **async**

```
app.post("/orphanages", async (req, res) => {
  const {
    name,
    latitude,
    longitude,
    about,
    instructions,
    opening_hours,
    open_on_weekends,
  } = req.body;

  const orphanagesRepository = getRepository(Orphanage);

  const orphanage = orphanagesRepository.create({
    name,
    latitude,
    longitude,
    about,
    instructions,
    opening_hours,
    open_on_weekends,
  });

  await orphanagesRepository.save(orphanage);

  return res.json({ message: "Hello World" });
});
```

Por fim, ao criar um objeto, é interessante passar o status 201, informando que deu tudo certo. Portanto:

`return res.status(201).json(orphanage);`

▼ Organizando e criando rotas

Primeiramente, cria-se um arquivo exclusivo para as rotas dentro da src e arrastamos as rotas criadas pra la e corrigir os erros

1. Importar o Router do express
2. instanciá-lo em alguma variável - routes
3. trocar app por routes
4. exportar routes como default

```
import {Router} from 'express'
import { getRepository } from "typeorm";
import Orphanage from "./models/Orphanage";

const routes = Router()
routes.post("/orphanages", async (req, res) => {
    const {
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
    } = req.body;
```

```
    const orphanagesRepository = getRepository(Orphanage);

    const orphanage = orphanagesRepository.create({
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
    });

    await orphanagesRepository.save(orphanage);

    return res.status(201).json(orphanage);
  });

export default routes;
```

Por fim, importamos essas rotas no server.ts

`import routes from './routes'`

e depois do "app.use(express.json());"

`app.use(routes);`

ARQUITETURA MVC

- Model
  - Representação de uma tabela no db, de uma dado
- View
  - Como as coisas são visualizadas e ficam disponíveis pro front end
- Controller
  - Onde fica a lógica das rotas

Portanto, criaremos uma pasta controllers e dentro, um OrphanagesController.ts

Inicialmente, exportaremos um objeto e dentro dele, criaremos um método assincrono de CRIAÇÃO de um orfanato, portanto, jogaremos de novo a lógica que estava na rota, no controller.

dentro do create, passaremos o req,res como parametros, porém como é um arquivo externo, que n ta conectado ao express, precisamos dizer que o req,res é Request e Response do express.

OrphanagesController.ts

```
import { Request, Response } from "express";
import { getRepository } from "typeorm";
import Orphanage from "../models/Orphanage";

export default {
  async create(req: Request, res: Response) {
    const {
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
```

```
    } = req.body;

    const orphanagesRepository = getRepository(Orphanage);

    const orphanage = orphanagesRepository.create({
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
    });

    await orphanagesRepository.save(orphanage);

    return res.status(201).json(orphanage);
  },
};
```

Por fim, importaremos o controller no routes e no lugar da função toda com async , colocaremos simplesmente OrphanagesController.create

```
import { Router } from "express";
import OrphanagesController from "./controllers/OrphanagesController";

const routes = Router();
routes.post("/orphanages", OrphanagesController.create);

export default routes;
```

Agora o código ta abstrato p um cacete, mas bem bem organizado.

▼ Listando orfanatos

Adicionaremos um método assíncrono 'index' dentro do nosso controller.

Este método deve vir separado por vírgula dos outros já criados, pois são métodos listados em um objeto maior (OrphanagesController.ts)

```
async index(req: Request, res: Response) {
    const orphanagesRepository = getRepository(Orphanage);

    const orphanages = await orphanagesRepository.find();
  },
```

Como só queremos listar todos, não há nenhum query dentro do find. Caso fosse quisto, bastaria fazer um find({queries})

Criado o método, basta criar a nova rota em routes.ts

```
routes.get("/orphanages", OrphanagesController.index);
```

▼ Detalhe de um orfanato

O método sera get, porém com um route param :id para pegar um orfanato em específico e o nome do método será show

Para pegarmos o id dentro do método show, utilizamos o req.params, que pode retornar um ou mais params a depender do que o usuário passar. Para pegarmos apenas o id , utilizamos a sintaxe de desconstrução

`const { id } = req.params;`

Para encontrar UM , utilizamos o findOneOrFail(id)

```
async show(req: Request, res: Response) {

    const { id } = req.params;
    const orphanagesRepository = getRepository(Orphanage);

    const orphanage = await orphanagesRepository.findOneOrFail(id);

    return res.json(orphanage);
  },
```

▼ Upload de imagens

Para armazenar essas imagens, não é uma boa prática colocar o arquivo em si no banco de dados e sim o nome dele.

Portanto, criaremos uma nova tabela para armazenar esses nomes

`yarn typeorm migration:create -n create_images`

Relacionamento de UM pra MUITOS

Um orfanato, tem muitas imagens

Por isso precisamos de um id

```
{
            name: "id", //Essa coluna vai ser gerada automaticamente
            type: "integer",
            unsigned: true, //não pode ser um negativo
            isPrimary: true, //Primary key
            isGenerated: true, //Essa coluna é gerada automaticamente
            generationStrategy: "increment", //1,2,3,4,5,6,7...
          },
          {
            name: "path",
            type: "varchar",
          },
          {
            name: "orphanage_id",
            type: "integer",
          },
```

Como essa nova tabela está relacionada com outra, precisamos de uma ForeignKey

Nela indicamos qual coluna da nova tabela está relacionada(columnNames), qual outra tabela está relacionada(referencedTableName) e qual coluna da tabela relacionada será utilizada(referencedColumnName)

Alem disso, podemos passar outras opcoes como onUpdate e onDelete, que dizemos o que deve acontecer com os dados novos caso o parametro relacionado sejam atualizado ou deletado.

onUpdate: 'cascade' → altera automaticamente caso o parametro mude

onDelete: 'cascade' → deleta automaticamente caso o parametro mude

Por fim, precisamos setar o down novamente

```
import { query } from "express";
import { MigrationInterface, QueryRunner, Table } from "typeorm";

export class createImages1602645550882 implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.createTable(
      new Table({
        name: "images",
        columns: [
          {
            name: "id", //Essa coluna vai ser gerada automaticamente
            type: "integer",
            unsigned: true, //não pode ser um negativo
            isPrimary: true, //Primary key
            isGenerated: true, //Essa coluna é gerada automaticamente
            generationStrategy: "increment", //1,2,3,4,5,6,7...
          },
          {
            name: "path",
            type: "varchar",
          },
          {
            name: "orphanage_id",
            type: "integer",
          },
        ],
        foreignKeys: [
          {
            name: "ImageOrphanage",
            columnNames: ["orphanage_id"],
            referencedTableName: "orphanages",
            referencedColumnNames: ["id"],
            onUpdate: "CASCADE",
            onDelete: "CASCADE",
          },
        ],
      })
    );
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.dropTable("images");
  }
}
```

`yarn typeorm migration:run`

Para lidar com update de arquivos, utilizaremos uma biblioteca : multer

`yarn add multer @types/multer`

Para configurarmos como os uploads serão feitos dentro da aplicação, na src, criaremos uma pasta **config** e dentro um arquivo **upload.ts**.

importaremos o multer e exportaremos um objeto.

A primeira configuração é o storage, que dizemos onde será feito o armazenamento, pode ser num CDN amazons3, googlecloud etcc. Usaremos o diskStorage.

Para facilitar a utilização dos caminhos, utilizaremos outra biblioteca → path, que especifica o caminho independente se sistema operacional ou os karai.

path.join(__dirname, '..','..','uploads')

```
import multer from 'multer'
import path from 'path'

export default {
    storage: multer.diskStorage({
        destination: path.join(__dirname, '..','..','uploads'),
        filename: (request, file, cb) =>{
            const fileName = `${Date.now()}-${file.originalname}`;

            cb(null,fileName);
        },
    })
};
```

A outra configuração, filename, recebe por parametro um request, um file e uma callbackfunction (cb)

Para evitar que dois uploads de imagem com o mesmo nome seja feito e uma sobrescreve a outra, adicionamos um timestamp antes do nome do arquivo

A callback function então é chamada. Caso haja erro, "null", caso não , passe o nome do arquivo - fileName.

Configurado o multer, agora iremos importá-lo dessa vez no nosso routes.ts e tambem as nossas configs

```
import multer from 'multer';
import uploadConfig from './config/upload'
```

Criamos uma instancia do multer numa variável upload, passando as configs como parametro

```
const upload = multer( uploadConfig );
```

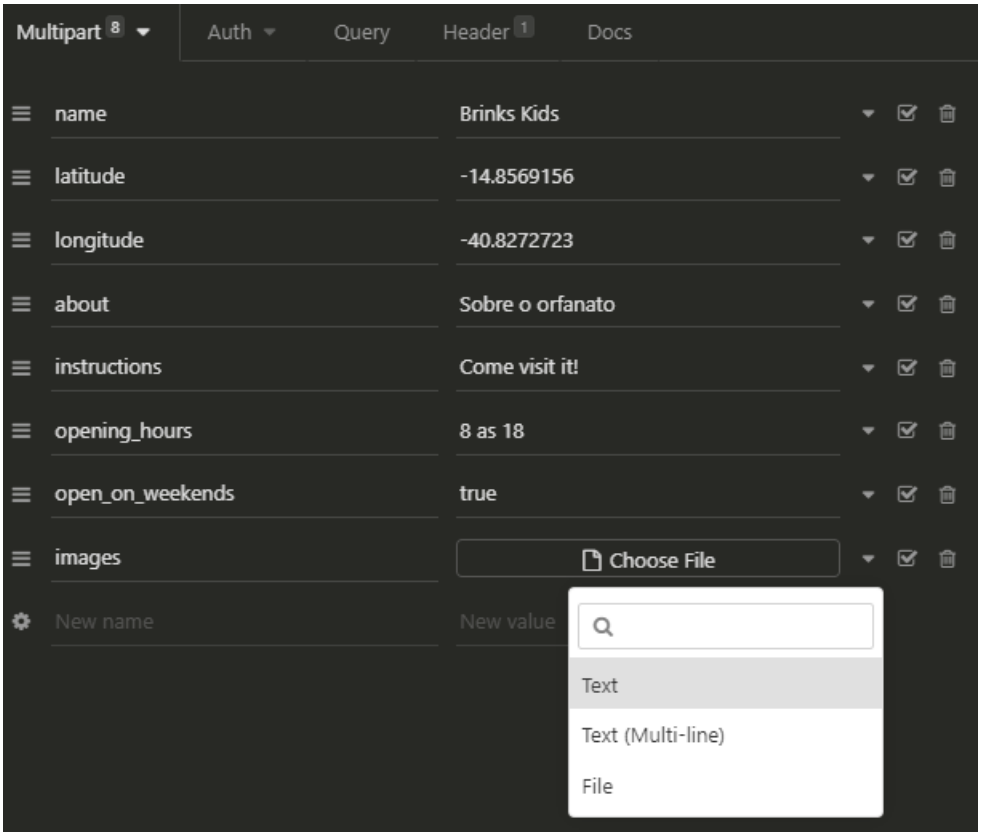
Agora, na rota em que queremos fazer upload de imagens iremos chamar o upload.any/array/fields/none/single, e utilizaremos o array porque queremos receber vários arquivos

e como parametro, o nome do campo que iremos passar os dados

```
routes.post("/orphanages",upload.array('images'), OrphanagesController.create);
```

Agora o upload já funciona localmente, ainda não configuramos para o banco de dados.

Para testar , podemos fazer uma requisição. Porém, para fazer requisições com uploads, o JSON não suporta imagens, então usaremos o MultipartForm



detalhe para o 'images', que foi o nome do campo que escolhemos em upload.array('images')

e podemos enviar qtos images forem necessários

Para cadastrar as imagens no banco de dados, precisamos primeiro criar o model dela

models/ → Image.ts

```
import { Entity, Column, PrimaryGeneratedColumn } from "typeorm";

@Entity("images")
export default class Orphanage {
  @PrimaryGeneratedColumn("increment")
  id: number;
  @Column()
  path: string;

}
```

o orphanage_id não precisa ficar no model. Poderia? poderia. Mas tem um jeito mais legal:

No model a ser relacionado, no nosso caso, Orphanage, importamos mais um cara do typeorm → 'OneToMany', um orfanato para muitas imagens

criamos um novo campo chamado 'images' MAS SEM O @Column, pois ele não ta no banco de dados e damos a ele o tipo dele será o model do Image.ts criado acima, que deve ser importado

`images: Image[]` → array de imagens

e decoramos ele com o @OneToMany(), que recebe uma função que diz qual o tipo do retorno

()⇒Image

e o segundo parametro, é a partir de um dado (image) recebido, qual o campo desse dado que retorna a relação inversa (image.orphanage), que criaremos agora

no Image.ts, importaremos o ManyToOne - inverso - e criamos o campo 'orphanage', do tipo igual ao model criado, então Orphanage, mas que dessa vez não é um array e adicionamos o decorador @ManyToOne tal qual o outro.

E outro decorador, @JoinColumn(), que recebe um objeto que indica qual o nome da coluna que está relacionado

No fim, os dois models ficam assim :

```
// Orphanage.ts
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  OneToMany,
  JoinColumn,
} from "typeorm";
import Image from "./Image";
@Entity("orphanages")
export default class Orphanage {
  @PrimaryGeneratedColumn("increment")
  id: number;
  @Column()
  name: string;
  @Column()
  latitude: number;
  @Column()
  longitude: number;
  @Column()
  about: string;
  @Column()
  instructions: string;
  @Column()
  opening_hours: string;
  @Column()
  open_on_weekends: boolean;

  @OneToMany(() => Image, (image) => image.orphanage)
  @JoinColumn({ name: "orphanage_id" })
  images: Image[];
}
```

```
// Image.ts
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  ManyToOne,
  JoinColumn,
} from "typeorm";
import Orphanage from "./Orphanage";

@Entity("images")
export default class Image {
  @PrimaryGeneratedColumn("increment")
  id: number;

  @Column()
```

```
  path: string;

  @ManyToOne(() => Orphanage, (orphanage) => orphanage.images)
  @JoinColumn({ name: "orphanage_id" })
  orphanage: Orphanage;
}
```

Além disso, no OneToMany, pode-se passar um terceiro parametro de configuração, em formato de objeto:

```
@OneToMany(() => Image, (image) => image.orphanage, {
    cascade: ['insert','update']
  })
```

esse cascade, que significa que irá fazer automaticamente, inserir ou atualizar as imagens relacionadas aquele objeto, no nosso caso, um orfanato

Por fim, para armazenarmos o nome das imagens upadas, podemos chamar o req.files durante a criação de um orfanato. o files retorna vários campos como fieldname, originalname, filename, path...

Então, no nosso método async create,

`const requestImages = req.files as Express.Multer.File[];`

o as Express.Multer.File[] força dizer que o req.files é um array de arquivos. Provavelmente um bug do multer

`const images = requestImages.map(( image ) => { return { path: image.filename }; });`

iteramos todas as imagens upadas e retornamos apenas o nome delas no campo path

e finalmente adicionamos o campo images no orphanage

```
async create(req: Request, res: Response) {
    const {
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
    } = req.body;

    const orphanagesRepository = getRepository(Orphanage);

    const requestImages = req.files as Express.Multer.File[];

    const images = requestImages.map((image) => {
      return { path: image.filename };
    });

    const orphanage = orphanagesRepository.create({
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
      images,
    });
```

```
    await orphanagesRepository.save(orphanage);

    return res.status(201).json(orphanage);
  },
};
```

E agora ta tudo setado 😂

▼ Trabalhando com as views

Inicialmente, para que as imagens upadas sejam vistas ao procurar os orfanatos, precisamos passar no find ou findOneOrFail um objeto com relations: ['images'] pra ele saber quais campos tem relação.

```
const orphanages = await orphanagesRepository.find({
      relations: ["images"],
    });

const orphanage = await orphanagesRepository.findOneOrFail(id, {
      relations: ["images"],
    });
```

Criamos dentro da src uma pasta views

E por que ?

as views determinam como os dados estarão visíveis para o front-end

Criamos então uma orphanages_view.ts onde será exportado :

o método render() que tratamos como retornar o orfanato pro front-end e quais dados retornar

```
// orphanages_view.ts
import Orphanage from "../models/Orphanage";

export default {
  render(orphanage: Orphanage) {
    return {
      id: orphanage.id,
      name: orphanage.name,
      latitude: orphanage.latitude,
      longitude: orphanage.longitude,
      about: orphanage.about,
      instructions: orphanage.instructions,
      opening_hours: orphanage.opening_hours,
      open_on_weekends: orphanage.open_on_weekends,
    };
  },
};
```

E agora de volta no Controller, ao inves de retornar o res.json(orphanage), retornaremos res.json(orphanageView.render(orphanage))

E ao mandar uma requisição, ele retorna só que setamos na view.

Importante p cacete, pois não precisamos alterar no controller

Beleza, mas esse render que fizemos só funciona para 1 orfanato. Queremos para muitos também, logo criaremos um `renderMany(orphanages: Orphanage[])` →[ ] de array e vamos retornar através de um map.

```
renderMany(orphanages: Orphanage[]){
      return orphanages.map(orphanage => this.render(orphanage))
  }
```

o que renderzira cada orfanato usando o método render criado acima.

O "this" se referente ao objeto maior, que é anônimo neste caso, e o método render faz parte dele.

Para mostrar as imagens, agora, criaremos outro view → images_view.ts

Só que ao inves de mostrarmos o id e o path, como já estava acontecendo, vamos exibir um url

```
import Image from "../models/Image";

export default {
  render(image: Image) {
    return {
      id: image.id,
      url: `http://localhost:3333/uploads/${image.path}`,
    };
  },

  renderMany(images: Image[]) {
    return images.map((image) => this.render(image));
  },
};
```

E no Orphanages_view.ts, vamos passar esse Image_view a ser exibido

```
import Orphanage from "../models/Orphanage";
import images_view from "./images_view";

export default {
  render(orphanage: Orphanage) {
    return {
      id: orphanage.id,
      name: orphanage.name,
      latitude: orphanage.latitude,
      longitude: orphanage.longitude,
      about: orphanage.about,
      instructions: orphanage.instructions,
      opening_hours: orphanage.opening_hours,
      open_on_weekends: orphanage.open_on_weekends,
      images: images_view.renderMany(orphanage.images),
    };
  },

  renderMany(orphanages: Orphanage[]) {
    return orphanages.map((orphanage) => this.render(orphanage));
  },
};
```

Por fim, para visualizar no url criado, vamos criar uma rota no server.ts utilizando o path pra dar um helps nesse caminho

```
app.use('/uploads', express.static(path.join(__dirname, '..','uploads')))
```

É claro que a imagem só ta no ambiente de desenvlvimento (localhost:3333), mas é interessante usar variaveis de ambiente na hora de dar deploy

▼ Tratando erros

Os métodos assincronos tem dificuldade de lidar com erros durante a execução

1. Para tal, usaremos a biblioteca express-async-errors

   yarn add express-async-errors

2. Importaremos no nosso server.ts logo apos o express

   `import "express-async-errors";`

Para deixar o erro mais legível, criaremos um exception handler

1. Criamos uma pasta na src → errors e um arquivo handler.ts
2. importamos do express o ErrorRequestHandler
    1. import {ErrorRequestHandler} from 'express'
    2. criamos uma variavel errorHandler e atribuimos a tipagem de ErrorRequestHandler a ela
    3. e essa variavel receber uma função que tem por parametros error,req,res,next e retorna
        1. console.error(error) → só pro dev
        2. return res.status(500).json({message: 'Internal Server Error'})
    4. Exportamos default

```
import { ErrorRequestHandler } from "express";

const errorHandler: ErrorRequestHandler = (error, req, res, next) => {
  console.error(error);

  return res.status(500).json({ message: "Internal Server Error" });
};

export default errorHandler;
```

E agora no server, podemos usá-lo após importar.

▼ Validação dos dados

Pra validar, utilizaremos o módulo yup

`yarn add yup @types/yup`

Para importar no controller

`import * as Yup from 'yup'`

inicialmente, vamos retirar os dados do orfanato de dentro do create e armazenar numa variavel 'data'

Criaremos um schema, que dirá quais campos temos na hora de inserir uma string e os tipos

```
const schema = Yup.object().shape({
      name: Yup.string().required(),
      latitude: Yup.number().required(),
      longitude: Yup.number().required(),
      about: Yup.string().required(),
      instructions: Yup.string().required(),
      opening_hours: Yup.string().required(),
      open_on_weekends: Yup.boolean().required(),
      images: Yup.array(
        Yup.object().shape({
          path: Yup.string().required(),
        })
      ).required(),
    });
```

E por fim , pra fazer a validação mesmo,

```
await schema.validate(data, {
      abortEarly: false,
    });
```

o abortEarly faz com que ele retorne tudo que deu merda, e não simplesmente parar no primeiro possível erro

Para exibir erros de validação, criamos outro handler no errorHandler

importamos a ValidationError da yup

```
import { ErrorRequestHandler } from "express";
import { ValidationError } from "yup";

interface ValidationErrors {
  [key: string]: string[];
}

const errorHandler: ErrorRequestHandler = (error, req, res, next) => {
  if (error instanceof ValidationError) {
    let errors: ValidationErrors = {};

    error.inner.forEach((err) => {
      errors[err.path] = err.errors;
    });

    return res.status(400).json({ message: "Validation fails", errors });
  }

  console.error(error);

  return res.status(500).json({ message: "Internal Server Error" });
};

export default errorHandler;
```

▼ Instalar o cors

Por fim, PRECISAMOS do cors para liberar que o front-end acesse o back-end

`yarn add cors @types/cors`

`import cors from "cors";`

`app.use(cors());`

Ainda vale uma configuração adicional pra dizer o caminho da aplicação, mas assim tá permitindo tudo de tudo.

▼ Finalizando o front-end

▼ Finalizando a página do mapa

Para colocar uma marcação no mapa, utilizaremos o Marker do reactLeaflet

E para utilizar um ícone, importamos o Leaftlet, sem ser do reactLeaflet

`import Leaflet from 'leaflet'`

e definimos um ícone utilizado, que no nosso caso ja foi importado : mapMarkerImg

```
const mapIcon = Leaflet.icon({
  iconUrl: mapMarkerImg,
})
```

Para configurar o centro do ícone como a ponta do pin, temos que mudar o iconAnchor do mapIcon

```
const mapIcon = Leaflet.icon({
  iconUrl: mapMarkerImg,
  iconSize: [58, 68],
  iconAnchor: [29, 68],
  popupAnchor: [170,2],
});
```

E então basta adicionar o Marker dentro do componente Map que o ícone já aparece

```
<Marker
  position={[-12.9890498, -38.4630524]}
  icon={mapIcon}
/>
```

Para adicionar um Popup ao clicar no icone,

importamos do reactLeaflet → Popup e dentro do Marker

```
<Marker position={[-12.9890498, -38.4630524]} icon={mapIcon}>
          <Popup
            closeButton={false}
            minWidth={240}
            maxWidth={240}
            className="map-popup"
          >
            Brinks Kids Bro
            <Link to="">
              <FiArrowRight size={20} color="#fff" />
            </Link>
          </Popup>
        </Marker>
```

closeButton false desabilita o 'x' de fechar

e min e maxWidth seta um tamanho fixo, ja que são iguas

Para deixar o popup lateral, adicionamos um popAnchor nas configs do icon e criamos uma className="map-popup", do proprio leaflet

Com isso , podemos personalizar o css dele em orphanages-map.css

Para o conteudo

```
#page-map .map-popup .leaflet-popup-content-wrapper {
  background: rgba(255, 255, 255, 0.8);
  border-radius: 20px;
}

#page-map .map-popup .leaflet-popup-content {
  color: #29b6d1;
  font-size: 20px;
  font-weight: bold;
  margin: 8px 12px;

  display: flex;
  justify-content: space-between;
  align-items: center;
}
```

Para o Link (apesar de ser Link, utiliza-se o 'a')

```
#page-map .map-popup .leaflet-popup-content a {
  width: 40px;
  height: 40px;
  background: #15c3d6;
  box-shadow: 17.2868px, 27.6589px, 41.4884px rgba(23, 142, 166, 0.16);
  border-radius: 12px;

  display: flex;
  justify-content: center;
  align-items: center;
}

#page-map .map-popup .leaflet-popup-tip-container {
  display: none;
}
```

▼ Adicionando as novas páginas e rotas

importando nos novos .tsx e chamando nas duas rotas :

```
<Route path="/orphanage/create" component={CreateOrphanage} />
        <Route path="/orphanage/:id" component={Orphanage} />
```

▼ Abstraindo componentes

Criamos uma pasta components em src e em styles

▼ Sidebar

Criamos um arquivo Sidebar.tsx pra tirar o componente da page 'CreateOrphanage' e jogamos dentro deste arquivo.

Além disso, tiramos os styles do create-orphanage.css e orphanage.css referentes e jogamos em um novo arquivo sidebar.css

E nas paginas Orphanage e CreateOrphanage e chamamos o componente <Sidebar /> Meczada ada

▼ Conectando backend com front end

Uma possibilidade seria usar o fetch, nativo. Mas iremos de axios :D

- `yarn add axios`

  No axios , conseguimos definir uma baseurl, tipo localhost:3333/

  Então criaremos uma pastas services e dentro dela, um api.ts, que fará nossa conexão

Nest api.ts, importaremos o axios e definiremos uma variavel api e dentro dela, o nosso baseURL, que será o utilizado pelo nosso **BACKEND** e a exportaremos

```
const api = axios.create({
    baseURL:  'http://localhost:3333'
})

export default api;
```

Bom, agora precisamos que o mapa renderize a cada alteração feita no backend.

Para isso, utilizaremos hookes

Por exemplo, o **useEffect(()⇒{},[])→** que executa assim que um componente é renderizado.

O **primeiro** é qual ação será executada. Se retornarmos uma função dentro desse escopo, como por exemplo: **useEffect(() ⇒ {return deleteCards}, [])**, essa função será executada única e exclusivamente quando o seu componente for desmontado.

o **segundo** , quando ela será executada.

por ser um vetor, o 'quando' será quando alguma das variáveis dentro desse vetor tiver o seu valor alterado.

Além disso, utilizaremos os estados → **useState.** O useState, por padrão, retorna um estado para uma variável e o seu set, por exemplo

```
const [number, setNumber] = useState(3);
```

Neste caso, o number começa com o valor 3 e toda vez que chamarmos , ele será 3. A não ser que utilizemos o 'setNumber'

setNumber(5)

Dessa forma, number passa a ser 5.

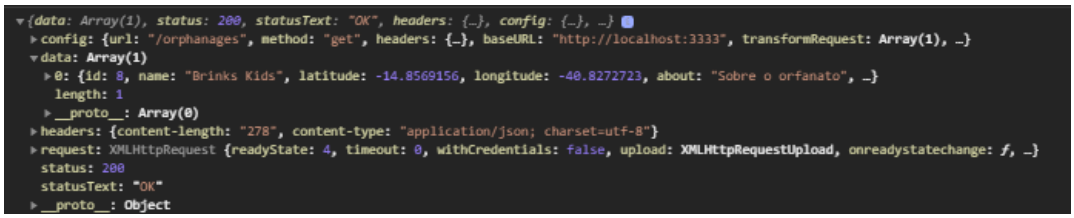
O valor anterior, 3, ainda pode ser acessado, mas veremos isso em breve.

Para utilizar os dois, importaremos do próprio react

Portanto, se fizermos a combinação dentro do nosso OrphanagesMap(), cada vez que esse componente for carregado, o useEffect será chamado e por conseguinte, o useState, atualizando ou não o estado das nossas variavel(eis) de interesse.

```
const [orphanages, setOrphanages] = useState([]);

  console.log(orphanages);

  useEffect(() => {
    api.get("/orphanages").then((response) => {
      console.log(response);

      setOrphanages(response.data);
    });
  }, []);
```

1. Criou-se uma variavel de estado 'orphanages' que inicia como um array vazio
2. O useEffect com a array vazia sendo passada como segundo argumento, será chamada sempre que o componente for montado, ele não executará a função nos renders posteriores, somente na montagem do componente que, no nosso caso, é o OrphanagesMap. Se tivéssemos qualquer variável dentro dos [], o useEffect seria chamado sempre que alguma dessas variáveis sofresse alterações.
3. A api está sendo mediada pelo axios e importada, logicamente. Quando damos um .get('/RotaDeInteresse'), ele acessa a rota do baseURL que setamos anteriormente. Para colher o retorno dessa rota na api, usamos o '.then(resposta ⇒ {})' e resposta vai armazenar o retorno da api na rota especificada. Foda.
4. Armazenamos na nossa variável de estado(orphanages) a resposta da nossa api através do método setOrphanages. a response possui vários campos, mas neste caso só estamos interessados no '.data'



▼ Dados dinâmicos

Agora, para deixar nossos orfanatos cadastrados aparecerem de forma dinâmica, iremos utilizar a variável de estado 'orphanages' e percorreremos todos os orfanatos existentes

Para isso, onde estamos exibindo um orfanato já fixo com o Marker, descrito no código, colocaremos um 'orphanages.map()' envolta dele e trocaremos os parametros que são mostrados.

```
{orphanages.map((orphanage) => {
          return (
            <Marker
              key={orphanage.id}
              position={[orphanage.latitude, orphanage.longitude]}
              icon={mapIcon}
            >
              <Popup
                closeButton={false}
                minWidth={240}
                maxWidth={240}
                className="map-popup"
              >
                {orphanage.name}
                <Link to={`/orphanage/${orphanage.id}`}>
                  <FiArrowRight size={20} color="#fff" />
                </Link>
              </Popup>
            </Marker>
          );
        })}
```

o key é uma propriedade que cada componente deve ter para identificar diferentes componentes, e deve ser UNICO, portanto, colocamos ele como sendo o próprio id do orfanato.

A grande fita é: o typescript ainda não sabe que diabos é um 'orphanage', precisamos dizer pra ele o que é, podemos verificar isso com o "never". Detalhe, aqui não precisamos tipar tuuuuuuuuuuuuuuuuuuudo que temos disponível. Apenas o que iremos usar.

Além disso, a tipagem é feita no armazenamento, então passaremos a nossa interface já na declaração dos estados

```
interface Orphanage {
  id: number;
  latitude: number;
  longitude: number;
  name: string;
}

const [orphanages, setOrphanages] = useState<Orphanage[]>([]);
```

Passamos para o useState um 'generic' → <>

Estamos dizendo pra o useState, que a lista que ele vai gerir é uma lista do tipo Orphanage.

E como é uma lista, colocamos o **[ ] na frente.**

Agora, nossa variável orphanage tem tipos 😇

Feito isso, para cada novo orfanato que cadastrarmos, já vai tá funfante.

Para a página de detalhe de um orfanato - Orphanage.tsx

Podemos usar o mesmo useEffect acima, importar a api, e o useState com pequenas alterações e criar a interface SOMENTE COM O NECESSÁRIO(quase tudo xD)

```
interface Orphanage {
  latitude: number;
  longitude: number;
  name: string;
  about: string;
  instructions: string;
  opening_hours: string;
  open_on_weekends: string;
images: Array<{
    id:number;
    path: string;
  }>
}
```

No detalhe para o images:

como ele é um array de infos, podemos usar o Array<{path: string}> ou {path: string}[]

o id:number é importante pois o usaremos para atribuir a uma key , ja que usaremos o map e todo elemento num map precisa deu ma key unica

nosso useState tem algumas alterações:

```
const [orphanage, setOrphanage] = useState<Orphanage>();
```

Como agora pegaremos apenas 1 orfanato, orphanage deixa de ser uma string e colocamos no singular

A rota do nosso api.get agora precisará de um id, e para isso utilizaremos o useParams do 'react-router-dom'

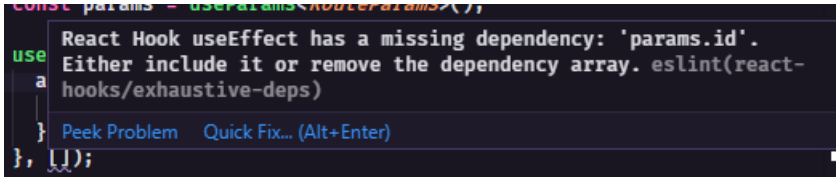
```
interface RouteParams {
  id: string;
}

export default function Orphanage() {
  const [orphanage, setOrphanage] = useState<Orphanage>();

  const params = useParams<RouteParams>();

  useEffect(() => {
    api.get(`/orphanages/${params.id}`).then((response) => {
      setOrphanage(response.data);
    });
  }, [params.id]);
```

De cara, precisamos tipar nosso useParams, pra dizer que aquele id que estamos pegando é uma string

Por fim, os params.id estão dentro das chaves do useEffect. Porque ?

Pq sempre que utilizarmos um link de dentro da pagina que altere esse params.id, queremos que o componente recarregue

Repare que o proprio typescript reclama :



Então 99% das vezes que utilizarmos uma variavel no nosso useEffect, ela será monitorável e estara dentro dos colchetes [ ].

Detalhe importante:

A lista de orfanatos começa vazia, então precisamos adicionar uma tratativa para isso

```
if (!orphanage) {
    return <p>Carregando ...</p>;
  }
```

Não é a melhor prática, geralmente se põe um spinner, um shimmer effect, mas é o que teremos p hoje

Agora substituimos tudo que antes era fixo por 'orphanages.ALGUMACOISA'

Para o card de Atende aos fins de semana, utilizamos

AlgoVerdadeiro ? faça isso : senão isso

'faça isso' no nosso caso é renderizar o cartão verde

'senão isso', o cartão vermelho. e criamos a classe dont-open , estilizando-a no css respectivo.

```
{orphanage.open_on_weekends ? (
                <div className="open-on-weekends">
                  <FiInfo size={32} color="#39CC83" />
                  Atendemos <br />
                  fim de semana
                </div>
              ) : (
                <div className="open-on-weekends dont-open">
                  <FiInfo size={32} color="#FF6690" />
                  Não atendemos <br />
                  fim de semana
                </div>
              )}
```

Para deixar as imagens dinamicas, fazemos um map

```
{orphanage.images.map(image => {
            return (
              <button key={image.id} className="active" type="button">
              <img
                src={image.url}
                alt={orphanage.name}
              />
            </button>
            );
          })}
```

lembrando: o key é necessário toda vez que for feito um map em react.

Pra que o mapa vá para o ponto pelo google maps:

```
<footer>
                <a
                  target="_blank"
                  rel="noopener noreferrer"
                  href={`https://www.google.com/maps/dir/?api=1&destination=${orphana
ge.latitude},${orphanage.longitude}`}
                >
                  Ver rotas no Google Maps
                </a>
              </footer>
```

Abre em outra guia

Inicia com about:blank na guia nova

Por fim, pra que o usuário clique em alguma das imagens e ela se torne a imagem ativa

Criaremos outro estado

```
const [activeImageIndex, setActiveImageIndex] = useState(0);
```

Pelo fato de iniciarmos o estado com um valor (0), o typescript já entende que a tipagem dele é um número

E passamos o activeImageIndex na img que é exibida

```
<div className="orphanage-details">
          <img src={orphanage.images[activeImageIndex].url} alt={orphanage.name}
 />
```

E por fim, atribuimos uma propriedade para os botões de onClick

```
{orphanage.images.map((image, index) => {
               return (
                 <button
                   key={image.id}
                   className={activeImageIndex === index ? "active" : ""}
                   type="button"
                   onClick={() => {
                     setActiveImageIndex(index);
                   }}
                 >
                   <img src={image.url} alt={orphanage.name} />
                 </button>
               );
             })}
```

detalhe para o **index** passado como segundo parametro no map

se a imagem clicada, ou seja, com indice certo for a selecionada, fica ativa, se não, fica fosca

▼ Página de cadastro de um novo orfanato

▼ Pegar posição clicada no mapa

Temos que atribuir um onClick para o <Map> e criar uma função a ser disparada quando esse clique acontecer - HandleMapClick

Essa função recebe um event que vai ser do type LeafletMouseEvent, e que possui uma propriedade latlng, que é latitude e longitude

Então, criaremos um estado para a posição clicada

```
const [position, setPosition] = useState({latitude:0 ,longitude:0})
```

```
const [position, setPosition] = useState({ latitude: 0, longitude: 0 });

function HandleMapClick(event: LeafletMouseEvent) {
  const { lat, lng } = event.latlng;

  setPosition({
    latitude: lat,
    longitude: lng,
  });
}
```

Fazemos uma desestruturação pra pegar lat e lng e passamos como parametros para o 'position' atraves do set.

E por fim, fazemos um "if", renderizando o map apenas se a posição for diferente de 0 (sempre né)

```
{position.latitude !== 0 && (
                <Marker
                  interactive={false}
                  icon={happyMapIcon}
                  position={[position.latitude, position.longitude]}
                />
              )}
```

▼ Pegar info do formulário

Existem várias formas de tratar formulários. A mais clássica é criar uma estado pra cada campo

```
const [name, setName] = useState('');
```

E no html desse input, adicionamos um value={name} e um onChange={event => setName(event.target.value)}

Por fim, precisamos criar uma função pra segurar estes valores : handleSubmit e atribuimos essa função ao onSubmit no form

```
function HandleSubmit(event: FormEvent){
    event.preventDefault();

    console.log(
      position,
      name,
      about,
      instructions,
      opening_hours
    )
  }
```

O event : FormEvent serve para chamarmos este preventDefault, que impede que a pagina recarregue automatico quando um formulário é submetido

```
<form onSubmit={HandleSubmit} className="create-orphanage-form">
```

Além disso, precisamos criar um estado para o botão e iniciamos ele num valor padrão

```
const [open_on_weekends, setOpen_on_weekends] = useState(true);
```

E nos nossos botões, vamos monitorar se open_on_weekends é true ou false para exibir o botão como ativo ou não

```
<button
                  type="button"
                  className={open_on_weekends ? "active" : ""}
                  onClick={() => setOpen_on_weekends(true)}
                >
                  Sim
                </button>
                <button
                  type="button"
                  className={!open_on_weekends ? "active" : ""}
                  onClick={() => setOpen_on_weekends(false)}
                >
                  Não
</button>
```

Por ultimo , o upload de imagens

Tomar cuidado para que o botão de upload não dê submit no form, então atribuir um type="button" a ele.

No nosso css, criamos um

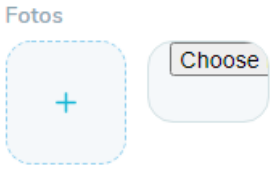
```
form.create-orphanage-form .input-block .images-container {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  grid-gap: 16px;
}

form.create-orphanage-form .input-block .images-container .new-image {
  width: 100%;
  height: 96px;
  background: #f5f8fa;
  border: 1px dashed #96d2f0;
  border-radius: 20px;
  cursor: pointer;

  display: flex;
  justify-content: center;
  align-items: center;
}
```

Agora precisamos criar uma maneira de quando clicar no botão '+', ele ser um input

a mais fácil é trocar o button para label e não precisa do type=button mais e adicionamos um htmlFor='image[]'

e embaixo criamos um input com o type='file' e id='image**[]**' , com o [] pois pode receber mais de uma imagem e 'multiple'

Isso já é suficiente pra funcionar, porém o botão de 'choose' fica aparecendo




```
<label htmlFor="image[]" className="new-image">
                  <FiPlus size={24} color="#15b6d6" />
                </label>
              </div>

              <input multiple type="file" id="image[]" />
```

Pra contornarmos isso, criamos um style específico pra esse input[type=file] com display none e jogamos o input pra fora do div images.container

```
form.create-orphanage-form .input-block .images-container {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  grid-gap: 16px;
}

form.create-orphanage-form .input-block .images-container .new-image {
  width: 100%;
  height: 96px;
  background: #f5f8fa;
  border: 1px dashed #96d2f0;
  border-radius: 20px;
  cursor: pointer;

  display: flex;
```

```
  justify-content: center;
  align-items: center;
}

form.create-orphanage-form .input-block input[type="file"] {
  display: none;
}
```

E para de fato adicionar a funcionalidade, criaremos um 'onChange' nesse input que chame um handleSelectImages

A gente não faz ideia do que vem dentro dele por enquanto, então passamos um event e damos um console.log só pra ver da qualé e um type any só pra o tsc n reclamar

```
function HandleSelectImages(event : any) {
    console.log(event);
  }
```

Nesse caso, veio um SyntheticEvent - (No idea)

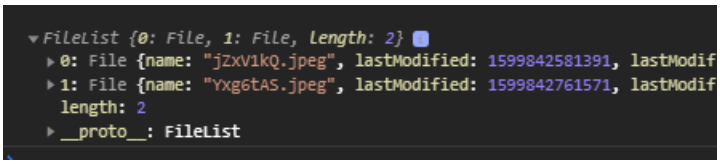
Então, podemos ver lá no onChange do input qual o tipo

Então o vemos que o tipo dele é ChangeEvent<HTMLInputElement>

E precisamos importar esse ChangeEvent do react

Dessa forma, conseguimos acessar ao event.target.files

E agora deu meczada, familia

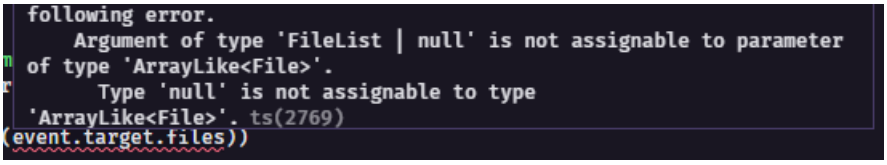


São esses files que enviamos ao backend e que mostramos no preview tambem

Então criaremos um state pra segurar essas images e precisamos dizer que ele será do tipo File Array

```
const [images, setImages] = useState<File[]>([]);
```

Agora, dentro do nosso Handle, iremos primeiro verificar se o usuario já inseriu alguma imagem, por que se não fizermos isso, o tipo pode ser FileList ou null



if (!event.target.files){return}

Nesse caso, se tiver vazio, ele quebra a função e não executa mais nada, por isso o return vazio

Além disso, precisamos 'converter' o FileList pra Array, então usaremos o Array.from

```
function HandleSelectImages(event: ChangeEvent<HTMLInputElement>) {
    if (!event.target.files) {
      return;
    }

    setImages(Array.from(event.target.files));
  }
```

Para o preview, vamos criar outro state, que será uma array de strings, pois usaremos o createObjectURL, que podemos passar um blob e ele retorna uma URL pra fazer preview

```
const [previewImages, setPreviewImages] = useState<string[]>([]);
```

No fim nosso Handle fica assim

```
function HandleSelectImages(event: ChangeEvent<HTMLInputElement>) {
    if (!event.target.files) {
      return;
    }
    const ChosenImages = Array.from(event.target.files);

    setImages(ChosenImages);

    const ChosenImagesPreview = ChosenImages.map(img => {
      return URL.createObjectURL(img)
    })

    setPreviewImages(ChosenImagesPreview)
```

Mas tá, que merda isso faz ? Pegamos as imagens selecionadas para upload(*ChosenImages*), iteramos sobre esse vetor e retornamos o URL criado de cada um dos arquivos lá dentro (*URL.createObjectURL*) e passamos isso para nosso estado previewImages

E agora, onde enviamos nossas imagens, vamos mostrar os preview junto.

Antes de renderizar o input, vamos mapear as imagens preview,

```
{previewImages.map((image) => {
                  return <img key={image} src={image} alt={name} />;
```

```
              })}
```

e criamos um html img pra exibir esse url criado. foda.

detalhe para o key={image} que como tem que ser unico, esse url já serve como, então ta mec

Tá funcionando, mas as imagens tão chegando nos tamanhos originais.

Para corrigir isso, um teco de css.

```
form.create-orphanage-form .input-block .images-container img {
  width: 100%;
  height: 96px;
  object-fit: cover;
  border-radius: 20px;
}
```

Detalhe pro object-fit:cover, que coloca o que dá da imagem sem esticar

Agora, finalmente, já temos todas as informações disponíveis no form

Para enviar essas infos no nosso HandleSubmit, como tem arquivos de imagem, não podemos usar JSON. ENtão utilizaremos um FormData(), que equivale ao MultiPartForm

```
const data = new FormData();
```

Então teremos de setar cada um desses campos manualmente

```
data.append("name", name);
    data.append("about", about);
    data.append("latitude", String(latitude));
    data.append("longitude", String(longitude));
    data.append("instructions", instructions);
    data.append("opening_hours", opening_hours);
    data.append("open_on_weekends", String(open_on_weekends));
    images.map((image) => {
      data.append("images", image);
    });
```

E chamamos api pra fazer o cadastro

Aqui temos duas opções

1. utilizar o api.post com o then

```
api.post("/orphanages", data).then(() => {
      alert("Cadastro realizado com sucesso!");
    });
```

2. tornar a função HandleSubmit async e dar um await no api.post

```
async function HandleSubmit(event: FormEvent) {
    event.preventDefault();
```

```
    const { latitude, longitude } = position;

    const data = new FormData();

    data.append("name", name);
    data.append("about", about);
    data.append("latitude", String(latitude));
    data.append("longitude", String(longitude));
    data.append("instructions", instructions);
    data.append("opening_hours", opening_hours);
    data.append("open_on_weekends", String(open_on_weekends));
    images.map((image) => {

// images.map((image) => {
    //   data.append("images", image);
    //   return ''

    images.forEach((image) => {
      data.append("images", image);
    });

    // api.post("/orphanages", data).then(() => {
    //   alert("Cadastro realizado com sucesso!");
    // });

    await api.post("/orphanages", data)
    alert("Cadastro realizado com sucesso!");

  }
```

Por que forEach e não map? Pq o map espera um **return**, e o eslint vai reclamar disso. Já o forEach, itera sempre esperar nada.

E por fim, redirecionamos o usuário pra algum lugar.

Para isso, usaremos o useHistory()

`const history = useHistory();`

e dentro do useHistory, podemos utilizar um método "push"

```
history.push('/app')
```

End game

Resolvendo o bug do open_on_weekends

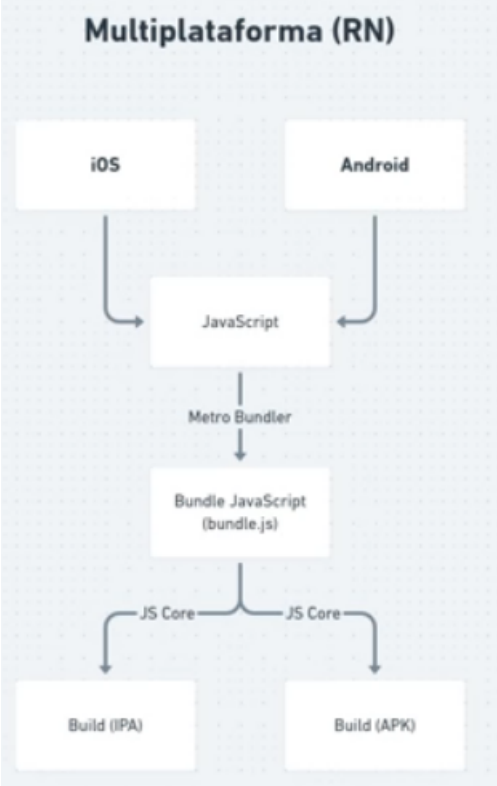
Ao enviar o dado da api, o frontend tá recebendo o dado com uma string "true" ou "false". Para fazer um lifehack rollercoaster mindblowing, na hora de armazenar o data, a gente vai transformar ele pra booleano lá no Controller do backend

```
const data = {
      name,
      latitude,
      longitude,
      about,
      instructions,
      opening_hours,
      open_on_weekends: open_on_weekends==='true',
      images,
    };
```

No fim das contas, isso retorna true ou false, a depender da info na string

▼ Mobile - React Native + Expo

Solução multiplataforma



Continua sendo uma aplicação nativa

Expo é como se fosse um framework com coisas prontinhas para utilizar e ele praticamente SIMULA o aplicativo mobile. foda.

Limitações e possibilidades : expo.canny.io

Para utilizar o expo e criar aplicações, precisamos instalar a CLI dele

```
yarn global add expo-cli
```

Pra começar um proejto

```
expo init mobile
```

Existem alguns templates, mas no nosso caso será o blank Typescript

E agora já podemos abrir a pasta mobile dentro do vscode

Com o expo instalado no celular , é só ler o QRCode e dar um Run on Android device/emulator

Além disso, podemos desenvolver um PWA. Um PWA é a aplicação acessada pela web - Progressive Web App

Com tudo pronto, `yarn start` 🙂

- Primeiro diferença : Não tem , <p> etc. Nós utilizamos os componentes de dentr do react-native
- A div é um View, mas qqr coisa é uma View, qualquer bloco
- E tudo que for texto, é Text. Não tem h1, h3, p
- Se quisermos estilizar algo, tem que ser diretão na linha, criando uma tag style={styles.algumaCoisa}. Não existe style de container.
  - E embaixo, nos styles, fazemos os estilos do 'algumaCoisa'
- Cada background, padding é específico
  - backgroundColor
  - paddingTop

  E tudo em camelCase

- TUDO JÁ ESTÁ EM DISPLAY FLEX e flexDirection COLUMN

▼ Criando a tela inicial

Pra usar o mapa, importaremos do react-native-maps o MapView

`import MapView from 'react-native-maps'`

O componente MapView pede

style = {styles.map}, que serão setados nos styles

um provider, que escolhemos o google

e uma latitude, longitude e zoom de cada um deles

```
<MapView
        provider={PROVIDER_GOOGLE}
        style={styles.map}
        initialRegion={{
          latitude: -28.1092052,
          longitude: -49.6401092,
          latitudeDelta: 0.008,
          longitudeDelta: 0.008,
        }}
      />
```

E os styles do map

```
map: {
    width: Dimensions.get("window").width,
    height: Dimensions.get("window").height,
  },
```

Uma coisa interessante é pegar o ícone do figma em 3 tamanhos: 1x 2x e 3x, por que os dispositivos tem densidade de pixel diferentes e o react native escolhe a melhor opção pra não ficar ruim. O nome **PRECISA** ter o @2x e @3x

Pra isso, novamente, criamos a src, e a pasta images pra colocar as imagens lá

De cara, ao importar o png, o typescript reclama

> Para contornar isso, criamos uma pasta @types no nosso src pra criar arquivos de DEFINIÇÃO de tipos : **index.d.ts**
>
> Nesse index, podemos declarar os arquivos de extensão .png com `declare module "*.png";`

Para utilizar o marker, importaremos do react-native-maps e colocaremos dentro do MapView :

```
<Marker
          icon={mapMarker}
          coordinate={{
            latitude: -28.1092052,
            longitude: -49.6401092,
          }}
        />
```

O "popout" do react é o **Callout no react native**

```
<Callout>
            <Text>Teste</Text>
          </Callout>
```

Então o View > MapView > Marker > Callout > Text

Pra estilizar esse Text do Callout, vamos criar um outro View dentro do Callout com o style={styles.calloutContainer} e dentro do View, o Text com o style.

> O tooltip={true} tira toda a estilização padrão. Opcionalmente poderia ser somente 'tooltip' que já teria o valor 'true' por padrão.
>
> onPress determinar o que vai acontecer quando clicarmos no Callout

```
<Callout tooltip={true} onPress={() => {
              alert("iu");
            }}>
  <View style={styles.calloutContainer}>
    <Text style={styles.calloutText}>Teste</Text>
  </View>
</Callout>
```

Os estilos do View e Text ficam:

```
calloutContainer: {
    width: 160,
```

```
    height: 46,
    paddingHorizontal: 16,
    backgroundColor: "rgba(255,255,255,0.8)",
    borderRadius: 16,
    justifyContent: "center",
  },
  calloutText: { color: "#0089a5", fontSize: 14 },
```

Pra criar o rodapé, vamos criar uma View embaixo do MapView

```
<View style={styles.footer}>
  <Text style={styles.footerText}> n orfanatos encontrados</Text>
  <TouchableOpacity style={styles.createOrphanageButton} onPress={()=>{}}>

  </TouchableOpacity>
</View>
```

O TouchableOpacity é uma das várias formas de criar um botão em react native. Nesse caso, ele perde um pouco de opacidade quando clicado. Também tem propriedade 'onPress'

Pra colocarmos um icon, importamos a Feather do @expo/vector-icons

`import {Feather} from '@expo/vector-icons'`

e usamos o 'plus'. Todos os ícones do Feather e nomes estão em feathericons.com

```
<Feather name="plus" size={20} color="#fff" />
```

Por fim, criamos as estilizações ditas acima

```
footer: {
    position: "absolute",
    left: 24,
    right: 24,
    bottom: 32,

    backgroundColor: "#fff",
    borderRadius: 20,
    height: 56,
    paddingLeft: 24,

    flexDirection: "row",
    justifyContent: "space-between",
    alignItems: "center",
  },

  footerText: {
    color: "#8fa7b3",
  },

  createOrphanageButton: {
    width: 56,
    height: 56,
    backgroundColor: "#15c3d6",
    borderRadius: 20,

    justifyContent: "center",
    alignItems: "center",
```

```
  elevation: 3, //SOMBRA
  },
});
```

Para utilizar fontes customizadas, utilizaremos o expo-google-fonts

`expo install @expo-google-fonts/nunito expo-font`

Isso só precisamos fazer 1 vez por fonte !

Pra usar as fontes, importaremos o useFonts do expo-font

`import {useFonts} from 'expo-font';`

E pra fonte em específico,

`import {Nunito_600SemiBold,Nunito_700Bold,Nunito_6800ExtraiBold} from '@expo-google-fonts/nunito'`

Por fim, chamamos

```
const [fontsLoaded] = useFonts({
    Nunito_600SemiBold,
    Nunito_700Bold,
    Nunito_800ExtraBold,
  });
```

Porém, esse carregamento de fontes é demorado. Então faremos um "loading..."

O fontsLoaded retornam true ou false se as fontes já tiverem sido carregadas ou não, então

```
if (!fontsLoaded) {
    return null;
  }
```

E agora já podemos adicionar as fontFamily no styles

```
footerText: {
    fontFamily: 'Nunito_700Bold',
    color: "#8fa7b3",
  },
```

▼ Fazendo roteamento

Para criar as rotas, vamos usar o react navigation

- `yarn add @react-navigation/native`
- `expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view`
- `yarn add @react-navigation/stack`

Existem 3 tipos de navegação :

stack - navegação em pilha, com botões

tab navigation - em abas

draw - puxa uma aba lateral

Criaremos um routes.tsx uma pasta pages, que conterá nosso OrphanagesMap.tsx

Jogaremos o View principal inteiro lá pra dentro e importamos o que é necessário

E no nosso routes.tsx , importaremos o navigation native e stack

Criamos uma instancia do createStackNavigator, que retorna duas coisas

Um navigator ,que é o navegador que fica por volta todas as telas

Um screen que é cada tela

```
import React from "react";

import { NavigationContainer } from "@react-navigation/native";
import { createStackNavigator } from "@react-navigation/stack";

const { Navigator, Screen } = createStackNavigator();

export default function Routes() {
  return (
    <NavigationContainer>
      <Navigator>
        <Screen name="OrphangesMap" component={OrphanagesMap} />
      </Navigator>
    </NavigationContainer>
  );
}
```

TOdas as rotas tem que tá dentro do Container com o Navigator dentro

E pra cada tela, um Screen, indicado qual o componente "rota"

Importamos as rotas devolta no App.tsx

`import Routes from './src/routes'`

e chamamos o Routes pra ser retornado

Detalhe para as fontes: Eles são importadas e chamadas no App.tsx, mas utilizados no OrphanagesMap.tsx

Aparentemente não tem problemas

Primeiramente, por utilizarmos uma navegação do tipo stack, um header é criado com o nome da rota. Pra desabilitar pra todas as rotas, chammos um screenOptions={{ headerShown: false}} no Navigator. Pra desabilitar rota por rota, é só colocar em cada screen.

```
<NavigationContainer>
  <Navigator screenOptions={{ headerShown: false }}>
    <Screen name="OrphangesMap" component={OrphanagesMap} />
```

```
    <Screen name="OrphanagesDetails" component={OrphanageDetails} />
  </Navigator>
</NavigationContainer>
```

Pra testar se a rota ta funcionando, criamos um componente só com um view vazio e no onPress do callout do detalhe dos orfanatos, chamaremos uma função HandleNavigateOrphanageDetails

`<Callout tooltip={true} onPress={HandleNavigateOrphanageDetails}>`

```
const navigation = useNavigation();
  function HandleNavigateOrphanageDetails() {
    navigation.navigate("OrphanageDetails");
  }
```

O navigation vem do navigation native

E tá plonto o sovetinho

▼ Fazendo o fluxo de cadastro

Como nosso cadastro será feito em mais de uma página, podemos criar várias páginas ou criar uma pasta que vai conter todas as paginas, que é o que faremos.

Na pasta de pages, vamos criar outra pasta CreateOrphanages

Criamos as paginas componente normalmente, OrphanageData e SelectMapPosition com o boilerplate padrão

```
import React from "react";
import { View } from "react-native";

export default function OrphanageData() {
  return <View />;
}
```

E importamos no routes.tsx

```
import SelectMapPosition from "./pages/CreateOrphanage/SelectMapPosition";
import OrphanageData from "./pages/CreateOrphanage/OrphanageData";
```

E chamamaos dentro do nosso Navigator

```
<Navigator screenOptions={{ headerShown: false }}>
        <Screen name="OrphangesMap" component={OrphanagesMap} />
        <Screen name="OrphanageDetails" component={OrphanageDetails} />

        <Screen name="SelectMapPosition" component={SelectMapPosition} />
        <Screen name="OrphanageData" component={OrphanageData} />
      </Navigator>
```

Agora na página que iremos puxar o redicionamento, no caso a OrphanagesMap, vamos criar um outro navigation HandleNavigateToCreateOrphanage, gigante mesmo, f***-se

```
function HandleNavigateToCreateOrphanage() {
    navigation.navigate("SelectMapPosition");
  }
```

E chamamos essa função no TouchableOpacity, o do "+"zinho

```
<TouchableOpacity
          style={styles.createOrphanageButton}
          onPress={HandleNavigateToCreateOrphanage}
        >
          <Feather name="plus" size={20} color="#fff" />
        </TouchableOpacity>
```

Como o header de várias telas é igual, iremos reaproveitá-lo :)

Pra isso,no nosso Navigator, manteremos o headerShown false e cardStyle: {backgroundColor: "#f2f3f5"}

E agora vamos criar o header customizado

Portanto, finalmente, pasta components

```
import React, { View, StyleSheet, Text } from "react-native";

export default function Header() {
  return (
    <View style={styles.container}>
      <Text style={styles.title}></Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    padding: 24,
    backgroundColor: "#f9fafc",
    borderBottomWidth: 1,
    borderColor: "#dde3f0",
    paddingTop: 44,

    flexDirection: "row",
    justifyContent: "space-between",
    alignContent: "center",
  },

  title: {
    fontFamily: "Nunito_600SemiBold",
    color: "#8fa7b3",
    fontSize: 16,
  },
});
```

E agora podemos chamar o nosso header nas paginas que terão header

Pra isso, na configuração da Screen que queremos, vamos passar um options com HeaderShown true e vamos substituir o header padrão com o criado

```
<Screen
          name="OrphanageDetails"
          component={OrphanageDetails}
```

```
          options={{
            headerShown: true,
            header: () => <Header title="Orfanato" />,
          }}
        />
```

e o title é uma propriedade customizada que criaremos no Header.tsx

E importamos o Header dos nossos components. top.

Pra criar o props.title, como estamos em typescript, precisamos de um HeaderProps 😂

```
interface HeaderProps {
  title: string;
}

export default function Header(props: HeaderProps) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>{props.title}</Text>
    </View>
  );
}
```

E para o botão de voltar, e fechar

```
<BorderlessButton>
        <Feather name="arrow-left" size={24} color="#15b6d6" />
      </BorderlessButton>

      <Text style={styles.title}>{props.title}</Text>

      <BorderlessButton>
        <Feather name="x" size={24} color="#ff669d" />
      </BorderlessButton>
```

E pra adicionar uma função de navegar aos botões, useNavigation denovo, criamos a variavel navigation, a função de navegar e atribuimos À propriedade onPress do botao ou colocamos direto

```
<BorderlessButton onPress={navigation.goBack}>
```

```
<BorderlessButton onPress={handleGoBackToHomepage}>
```

Detalhe que o goBack é builtin e volta pra ultima pagina acessada.

já o handle, foi criado e leva pra uma rota em especifico.

```
function handleGoBackToHomepage() {
    navigation.navigate("OrphanagesMap");
  }
```

Pra tirar o botão de algumas pagina específica,vams criar uma props showCancel booleana opcional. Então adicionamos isso no Header que queremos desabilitar o

'x'zinho.

```
options={{
            headerShown: true,
            header: () => <Header showCancel={false} title="Orfanato" />,
          }}
```

Pra setar, precisamos incluir no nosso HeaderProps

```
interface HeaderProps {
  title: string;
  showCancel?: boolean;
}
```

E pra tratar :

```
{showCancel ? (
        <BorderlessButton onPress={handleGoBackToHomepage}>
          <Feather name="x" size={24} color="#ff669d" />
        </BorderlessButton>
      ) : (
        <View />
      )
```

Detalhe pra o <View /> em branco, que serve pra não deslocar o texto pra direita caso não haja o botão

Outro detalhe pra o props desestruturado →{ title, showCancel = true }

ScrollView é um mesmo View, com scroll.

Se tiver um 'horizontal', é um scroll horizontal

o pagingEnabled deixa fixo por imagem, não fica na metade

▼ Pegando os dados do form e jogando pro backend

Ligamos o server

instalamos o axios novamente

→ pasta services → api.ts

```
import axios from "axios";

const api = axios.create({
  //   baseURL: "http://localhost:3333",
  baseURL: "http://192.168.0.107:3333",
});

export default api;
```

Criamos um estado pro orphanages, useEffect pra executar a cada render, e dentro, um api,get pra chamar a api na rota especificada

```
const [orphanages, setOrphanages] = useState<Orphanage[]>([])
useEffect(() => {
    api.get("/orphanages").then((response) => {
      setOrphanages(response.data);
    });
  }, []);
```

Além disso, pra exibir o detalhe do orfanato correto, passamos o id dele como parametro e pra não quebrar o onPress do botão, que só aceita uma função e não a execução dela, fazemos um ()⇒ função() pra contornar. Pro gamer movement

```
function HandleNavigateToOrphanageDetails(id: number) {
    navigation.navigate("OrphanageDetails", { id });
  }
```

```
onPress={() => HandleNavigateToOrphanageDetails(orphanage.id)}
```

Pra recuperar esse 'id' na página navegada, que é o OrphanageDetails, vamos utilizar um useRoute da navigation native

```
interface OrphanageDetailsRouteParams {
  id: number;
}

export default function OrphanageDetails() {
  const route = useRoute();
  const params = route.params as OrphanageDetailsRouteParams; // Object {
  "id": 19,
}
```

então conseguimos pegar o nosso id no outro componente 😌

Chamamos o useEffect

```
useEffect(() => {
    api.get(`/orphanages/${params.id}`).then((response) => {
      setOrphanage(response.data);
    });
  }, [params.id]);
```

Eqto espera carregar os dados

```
if (!orphanage) {
    return <View>Carregany.</View>;
  }
```

E agora só sair colocando os orphanage.INFO nos campos

Pra puxar as imagens

```
{orphanage.images.map((image) => {
            return (
              <Image
                key={image.id}
                style={styles.image}
                source={{
                  uri: image.url,
                }}
              />
            );
          })}
```

Atender no fim de semana

```
{orphanage.open_on_weekends ? (
            <View style={[styles.scheduleItem, styles.scheduleItemGreen]}>
              <Feather name="info" size={40} color="#39CC83" />
              <Text style={[styles.scheduleText, styles.scheduleTextGreen]}>
                Atendemos fim de semana
              </Text>
            </View>
          ) : (
            <View style={[styles.scheduleItem, styles.scheduleItemRed]}>
              <Feather name="info" size={40} color="#ff669d" />
              <Text style={[styles.scheduleText, styles.scheduleTextRed]}>
                Não atendemos fim de semana
              </Text>
            </View>
          )}
```

Ver rotas no google , vamos usar o 'deeplinking' que é a conexão de um app pra outro

E podemos usar o mesmo link do web

Pdemos trocar o View por um TouchableOpacity e um onPress handleOpenGoogleMapRoutes

Dentro do handle, vamos usar o Linking, que é a forma de abrir um link

```
function handleOpenGoogleMapRoutes() {
    Linking.openURL(
      `https://www.google.com/maps/dir/?api=1&destination=${orphanage?.latitude},${orphanag
e?.longitude}`
    );
  }
```

Por fim, criar um orfanato com os dados né ...

- Pegar a localização clicada no mapa → SelectMapPosition.tsx

  Para isso, criamos um estado position, já iniciado com um valor do tipo number, assim não precisamos tipar

```
const [position, setPosition] = useState({ latitude: 0, longitude: 0 });
```

Adicionamos um onPress ao MapView e criamos o handle com um event do tipo MapEvent do proprio react-native-maps e setamos a position como um event.nativeEvent.coordinate, que é do tipo LatLng, igual ao nosso tipo number number

```
function handleSelectMapPosition(event: MapEvent) {
    setPosition(event.nativeEvent.coordinate);
  }
```

E aí só mostramos o Marker caso a position.latitude seja diferente de 0, ou seja, se tivermos clicado no mapa com aquela notação foda do boolean && boolean

```
{position.latitude !== 0 && (
          <Marker
            icon={mapMarkerImg}
            coordinate={{
              latitude: position.latitude,
              longitude: position.longitude,
            }}
          />
        )}
```

E fazemos o mesmo pro botão 'proximo'

```
{position.latitude !== 0 && (
        <RectButton style={styles.nextButton} onPress={handleNextStep}>
          <Text style={styles.nextButtonText}>Próximo</Text>
        </RectButton>
      )}
```

Agora precisamos passar a coordenada para a próxima tela como parâmetro, então no handle que navega pra ela, passamos o position

```
function handleNextStep() {
    navigation.navigate("OrphanageData", { position });
  }
```

Novamente, pra recuperar um dado passado de um componente para o outro, useRoute() e recuperamos o position do params

```
export default function OrphanageData() {
  const route = useRoute();
  console.log(route.params);
```

Daí como não podemos simplesmente chamar o route.params.Object(denovo)

```
interface OrphanageCoordinateRouteParams {
  position: {
    latitude: number;
    longitude: number;
  };
}

export default function OrphanageData() {
```

```
const route = useRoute();
const params = route.params as OrphanageCoordinateRouteParams;
console.log(params.position.latitude);
```

Finalmente, pra pegar os dados do form

Criamos os estados

```
const [name, setName] = useState("");
  const [about, setAbout] = useState("");
  const [instructions, setInstructions] = useState("");
  const [opening_hours, setOpening_hours] = useState("");
  const [open_on_weekends, setOpen_on_weekends] = useState(true);
```

E pra cada input, novamente, atribuimos um value = nameDoimput e um onChange**Text** = {text => setName(text)} ou simplesmente {setName}

O unico diferente é o Switch, que usa o onValueChange

```
<Text style={styles.label}>Horario de visitas</Text>
      <TextInput
        style={styles.input}
        value={opening_hours}
        onChangeText={setOpening_hours}
      />

      <View style={styles.switchContainer}>
        <Text style={styles.label}>Atende final de semana?</Text>
        <Switch
          thumbColor="#fff"
          trackColor={{ false: "#ccc", true: "#39CC83" }}
          value={open_on_weekends}
          onValueChange={setOpen_on_weekends}
        />
      </View>
```

Agora no nosso botão Cadastrar, o RectButton, passamos um handleCreateOrphanage no onPress

Pra testar se ta funfnado, vamos dar um log no que já tem pronto

```
function handleCreateOrphanage() {
    let { latitude, longitude } = params.position;

    console.log({
      name,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
      latitude,
      longitude,
    });
  }
```

E ta faltando só o upload de imagens

Para isso, vamos usar um módulo do expo

```
import * as ImagePicker from "expo-image-picker";
```

E importamos todas as funções com o *

É só uma outra forma pra não ter que ficar importando separado com os {a,b,c}. É bom pra quando tiver muitos, mas imagino que no fim, seja mais vantajoso declarar tudo que ta sendo importado um a um

Criaremos um handleSelectImages que será chamada quando clicar no botão de adicionar fotos

Primeiro, precisamos pedir permissão ao usuário pra acessar a galerua

```
async function handleSelectImages() {
    const { status } = await ImagePicker.requestCameraRollPermissionsAsync();
  }

  if (status !== "granted") {
    alert("Você precisa conceder acesso !");
    return;
  }
```

E por conta do await, precisamos tornar ela uma function assincrona

Armazenamos a permissão em status, e caso o usuário não tenha permitido, , emite um alerta e quebra função com o return

Se tudo der certo na permissão,

```
const result = await ImagePicker.launchImageLibraryAsync({
      allowsEditing: true,
      quality: 1,
      mediaTypes: ImagePicker.MediaTypeOptions.Images,
    });
```

Permitimos a edição da foto pelo usuário, com crop e etc

A qualidade na melhor config→1

E só permitimos a adição de Images. Outros tipos de arquivo não serão permitidos

Verificamos se o usuário cancelou o upload. Se não, armazenamos o uri, Se sim, quebra a handle

```
if (result.cancelled){
      return;
    }

    const { uri } = result
```

Pra fazer o preview, vamos criar um estado como uma array de strings

```
const [images, setImages] = useState<string[]>([])
```

E aqui, como precisamos dos valores anteriores, caso queiramos adicionar mais de uma imagem , precisamos dos previewValues, ou seja, na hora do set, fazer um spread com o que tinha antes e depois passar o valor que queremos setar

```
const { uri } = result;
    setImages([...images, uri]);
```

E pra mostrar o preview, criamos uma View nova abaixo do título

```
<View style={styles.uploadedImagesContainer}>
        {images.map((image) => {
          return (
            <Image
              key={image}
              source={{ uri: image }}
              style={styles.uploadedImage}
            />
          );
        })}
      </View>
```

E as estilizações

```
uploadedImagesContainer: {
    flexDirection: "row",
  },
  uploadedImages: {
    width: 64,
    height: 64,
    borderRadius: 20,
    marginBottom: 32,
    marginRight: 8,
  },
```

Por fim, pra enviar isso pro banco de dados, vamos usar de novo o FormData

```
async function handleCreateOrphanage() {
    let { latitude, longitude } = params.position;

    console.log({
      name,
      about,
      instructions,
      opening_hours,
      open_on_weekends,
      latitude,
      longitude,
    });
```

```
    const data = new FormData();

    data.append("name", name);
    data.append("about", about);
    data.append("latitude", String(latitude));
    data.append("longitude", String(longitude));
    data.append("instructions", instructions);
    data.append("opening_hours", opening_hours);
    data.append("open_on_weekends", String(open_on_weekends));

    images.forEach((image, index) => {
      data.append("images", {
        name: `image_${index}.jpg`,
        type: "image/jpg",
        uri: image,
      } as any);
    });

    await api.post("/orphanages", data);

    navigation.navigate("OrphanagesMap");
  }
```

Detalhe para o append de images, que tem que ser um por um, e ele precisa de trÊs informações,: name, type e uri

o tipo ta definido como as any, por conta de um problema do react-native, mas será resolvido pela biblioteca em breve

Pra finalizar, vamos usar no map o useFocusEffect, que vai renderizar sempre que a tela for 'focada' no lugar do useEffect.