***NLW – Next Level Week. #01***

# Projeto: Ecoleta

## Justificativa

Na semana 01/06 a 05/06 começa a semana internacional do meio ambiente. O projeto desenvolvido é voltado para esta área.

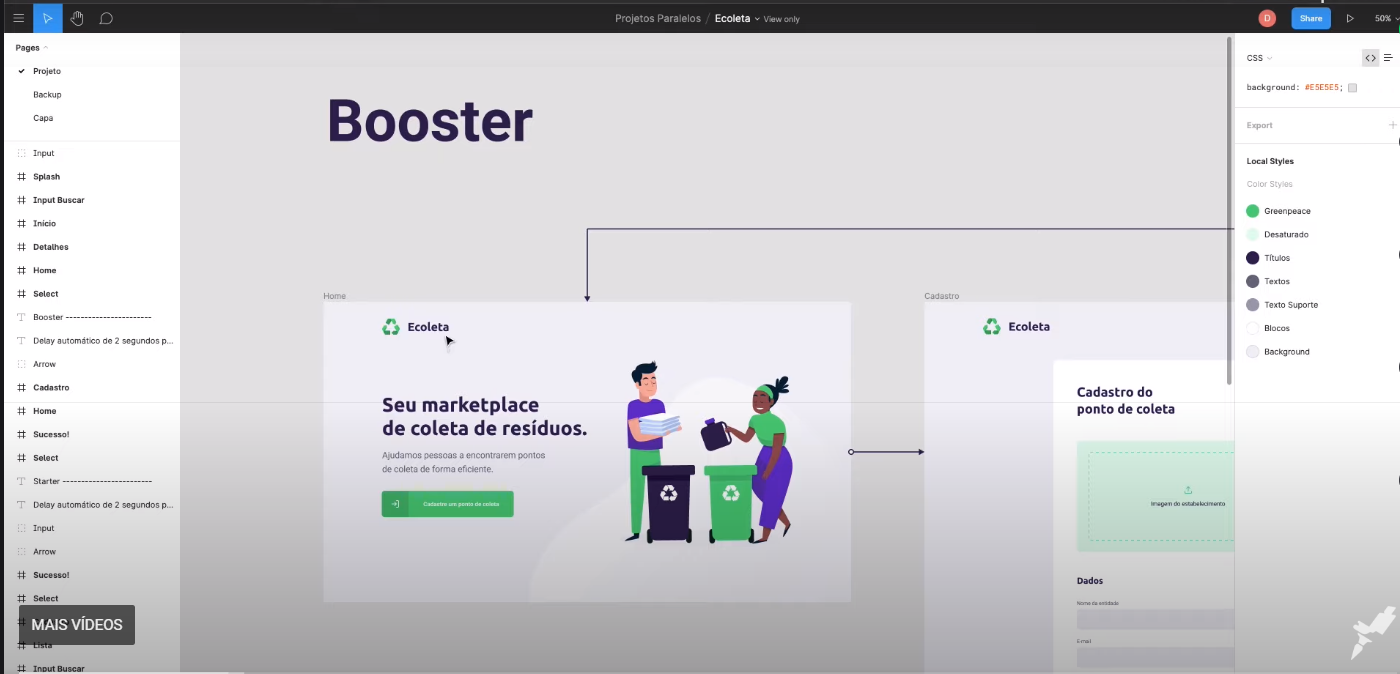
## Ideia

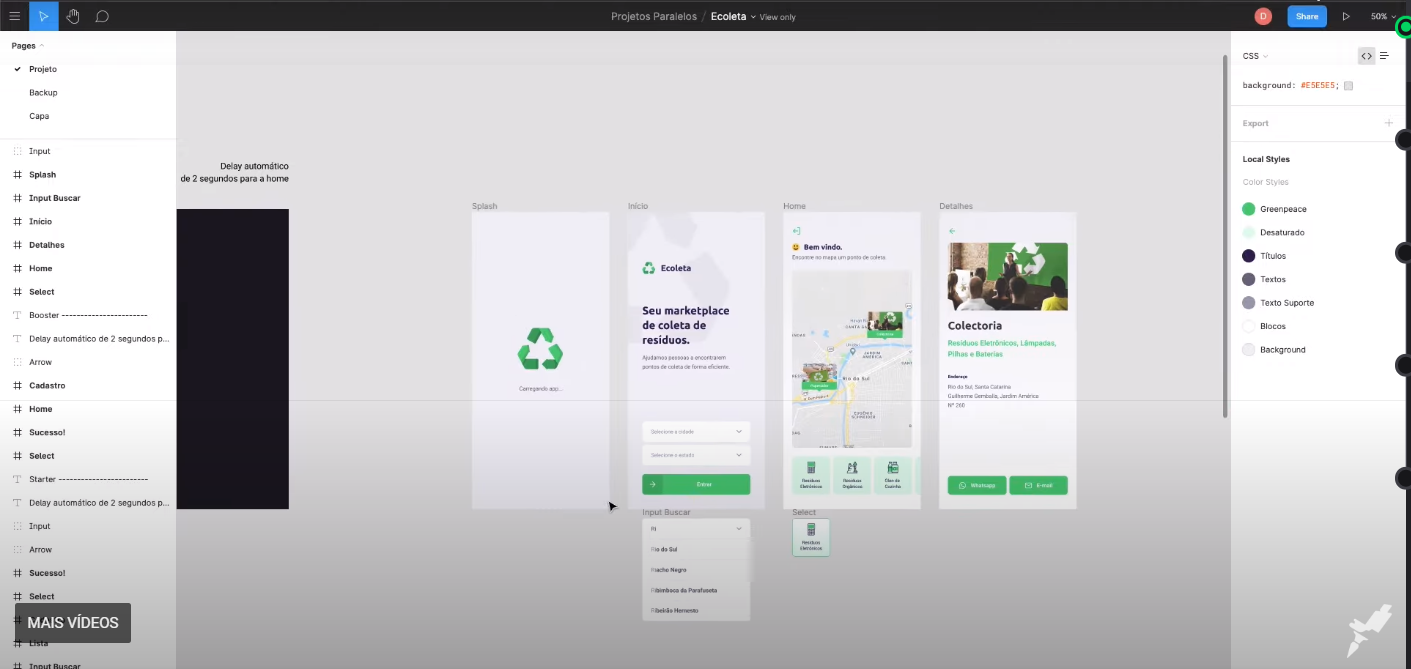
O Ecoleta é um marketplace (mapa, conexão) entre empresas/entidades que coletam resíduos, orgânicos ou inorgânicos às pessoas que precisam descartar esses resíduos.

## Desenvolvimento

Frontend (Web): entidades cadastrarem os pontos de coleta.

Frontend (Mobile): pessoas identificarem quais os locais fazem a coleta de materiais específicos de acordo com uma localização escolhida.



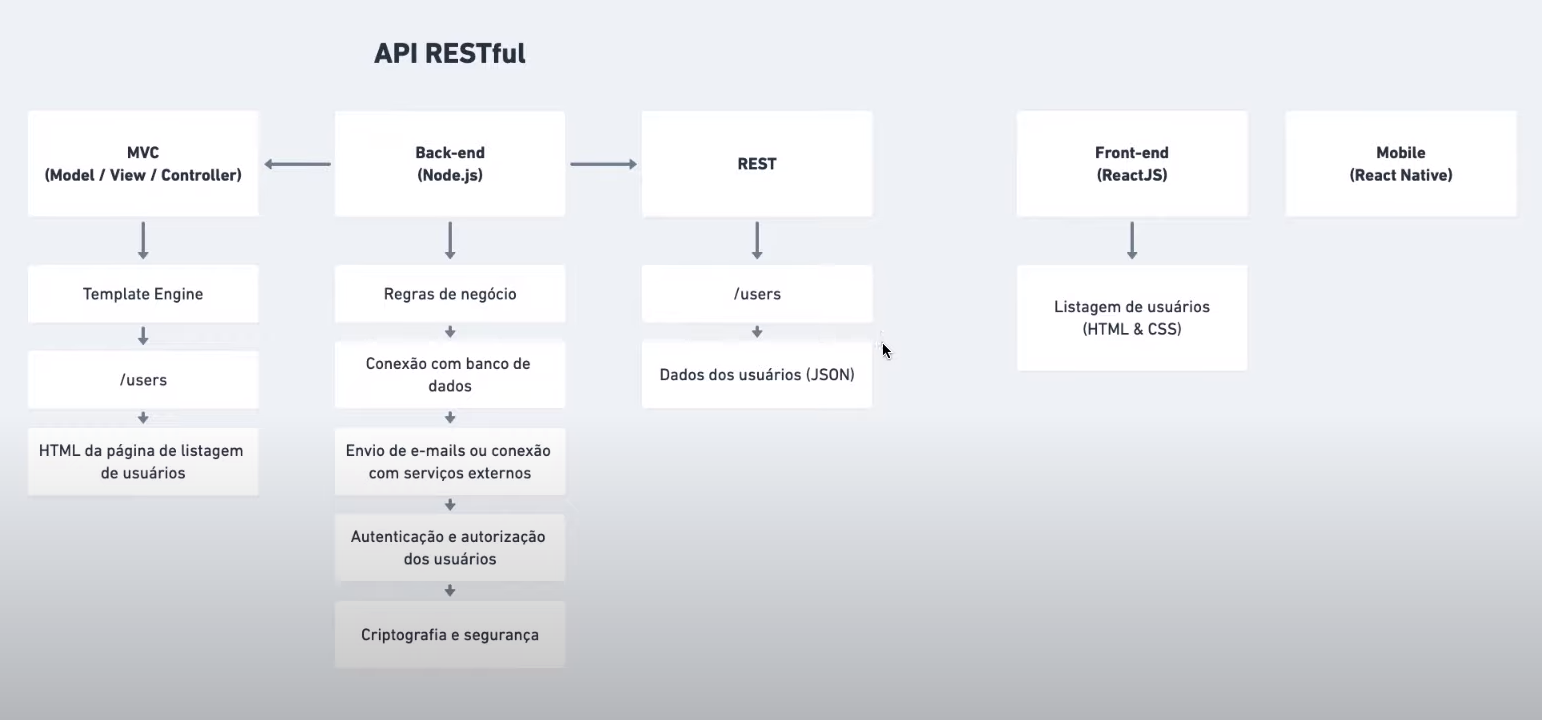


# Conceitos

## API RESTful

Restful: informa que a API (serviço desenvolvido/backend), segue as regras, está padronizado seguindo o REST. Ou seja, uma API que segue os psdrões REST é RESTful.

Seguindo esse conceito, há uma padronização no retorno dos dados, facilitando a independência de fron-end, ou seja, pode fazer uma troca simples entre web e mobile, por exemplo, sem ficar amarrado a uma única interface que irá mostrar os dados.

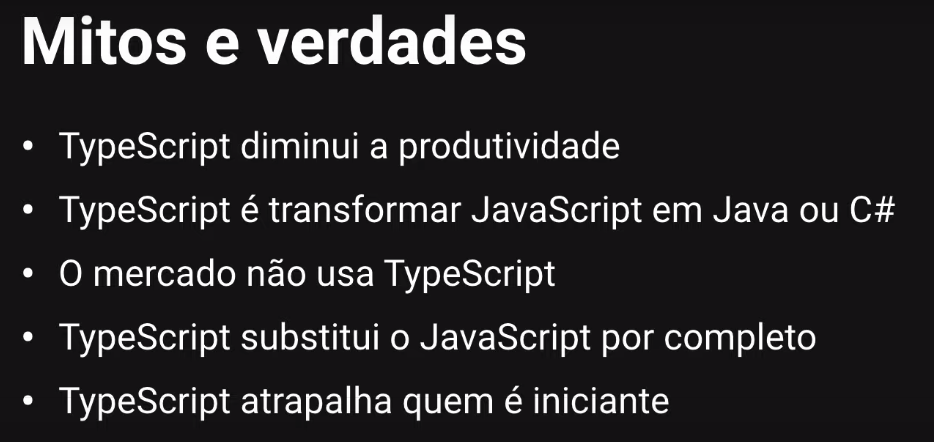


## TypeScript

É um superst (linguagem): javascript adicionado a possibilidade de adicionar tipagens.

Facilita o lado do desenvolver pois adiciona tipos para que a IDE consiga identificar quais os formatos das variáveis, argumentos das funções... (IntelliSense).





Link com os principais tipos: <https://github.com/typescript-cheatsheets/react-typescript-cheatsheet>

Instalação: npm install typescript -D

Criação do arquivo de configuração do TS: definição de quais features do typescript deseja/não deseja ser utilizado. Comando: npx tsc –init. (tsc: typescript).

Utiliza na extensão o .ts ao invés do .js.

Quando a aplicação rodar o código ts é convertido em js.

## Knex

Vantagens de se utilizar o Knex:

1. utiliza apenas JS na aplicação, dessa forma, há a presença da inteligência da IDE (IntelliSense).
2. Adaptação para qualquer banco de dados, ou seja, em situações de migração de banco de dados, o código continua o mesmo, pois foi utilizado o query builder.

## Path

Biblioteca para trabalhar com caminhos no node.

## Migrations

Recurso para criação das tabelas utilizando JS proporcionado pelo knex.

É o controle de versão do banco de dados, guarda o histórico do banco de dados. São arquivos que informam as alterações que serão feitas ao banco de dados.

Auxilia em projetos que possui mais de uma pessoa desenvolvendo e em projetos muito grandes.

Ex de migrations: criar, editar, excluir migrations.

## Banco de Dados - SQLite

SQL: SQLite. Para utilizar o SQLite não precisa instalar nada na máquina. Salva os dados em um arquivo .sqlite.

### Visualização das tabelas

Para visualizar as tabelas e registros inseridos no sqlite instale o pacoteno VSCode: SQLite.

Comando para abrir: Ctrl + Shift + P -> sqlite-database

## Conceitos utilizados no projeto

### Backend

* Banco de dados SQL: SQLite.
* Knex: query builder.
  + Migrations: criação das tabelas.
  + Seeds
  + Transaction
* Cors: informa quais URLs podem fazer acesso à API.

### Frontend – Web

* Páginas SPA.
* Leaflet: mapa de código aberto.
* Axios.
* API IBG – estados e cidades
* Localização do usuário

### Frontend – Mobile

* Mapas
* Axios
* Localização do usuário
* Permissão do usuário para acesso a localização
* Conexão com email (mail composer) e whatsapp (deep linking)

### Funcionalidades Avançadas

* Upload de Imagens
  + Crypto: biblioteca para geração de hash aleatório
  + React DropZone
* Validação de dados (backend) - celebrate
* Deploy

# Aula 1

## Criação do projeto - backend

1. Comando: npm init -y
2. Express: micro framework para lidar com rotas, interpreta a requisição e direcionar a execução de tarefas. Comando: npm install express.
3. Instalação do typescript: Comando: npm install typescript -D
4. Criação do arquivo de configuração do typescript. Comando: npx tsc –init.
5. Instalar o pacote para o node entender o typescript (converter para js) para executar. Comando: npm install ts-node -D.
6. Instalar o pacote para não precisar reiniciar o servidor a cada nova modificação. Comando: npm install ts-node-dev -D.
7. Criação da pasta src (**source**): concentra todo o código da aplicação.

## Rodar o projeto

Comando: npx ts-node src/server.ts / npx ts-node-dev src/server.ts

Npx: serve para executar um pacote instalado.

Adição do comando de rodar o projeto a um script para não ficar digitando o comando grande.

"scripts": {

    "dev": "ts-node-dev  --transpileOnly --ignore-watch node\_modules src/server.ts",

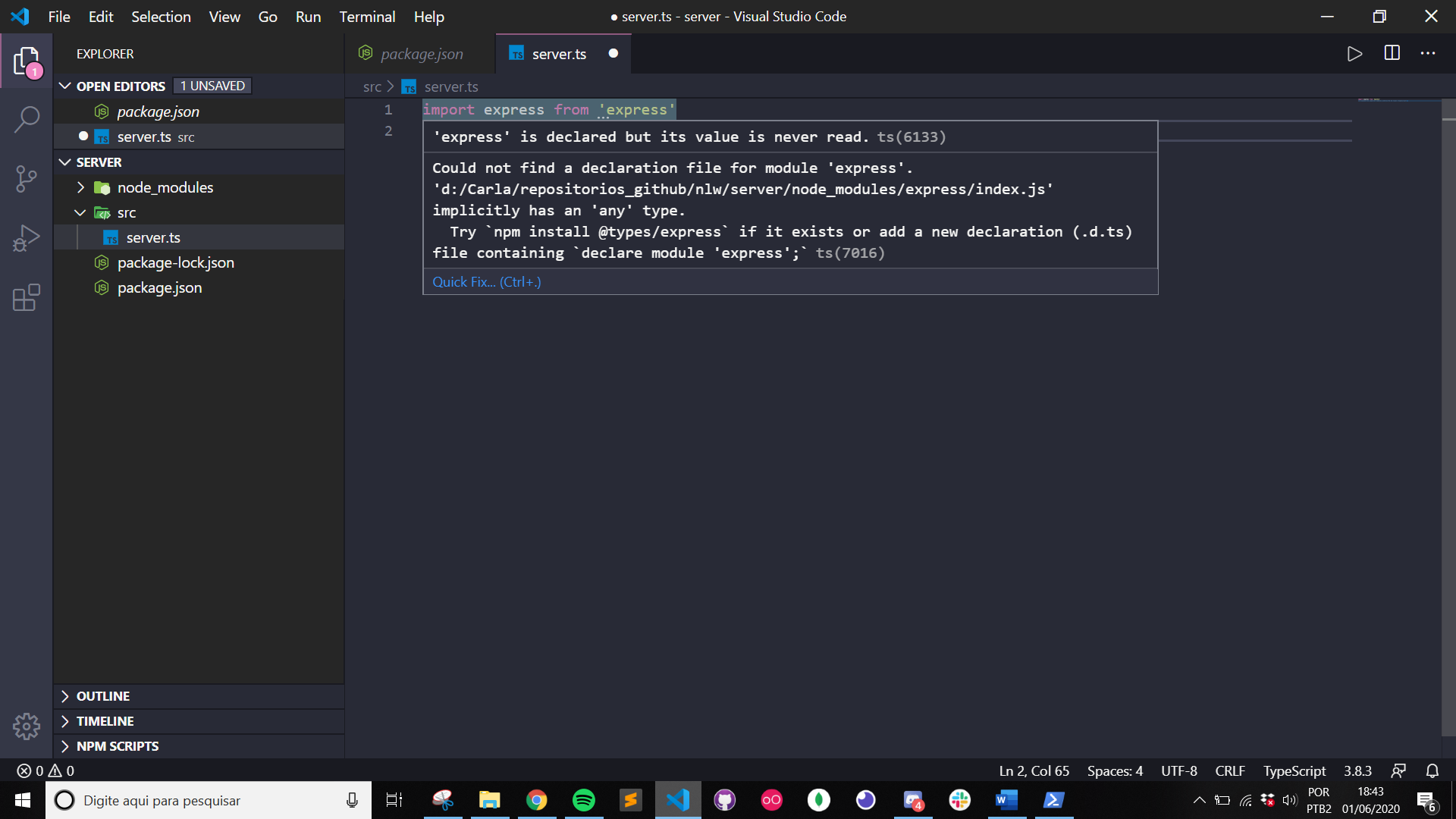
  },

Rodar agora com: npm run dev ou yarn dev.

### Projeto

1. Criação do arquivo server.ts (type script).
2. Importação do express. É necessário importar também a definição de tipos (typescript)

Nesses casos o próprio VSCode informa através de ... no nome da biblioteca importada:



Comando: npm install @types/express -D

1. Criação da aplicação

const app = express()

1. Criação das rotas.

//request: obter dados da requisição.

//response: devolver resposta para a requisição feita. Sempre deve proceder do return

app.get('/users', (*request*, *response*) => {

    console.log('Listagem de usuários.')

    return *response*.json([

        'Carla',

        'Patrícia',

        'Jéssica'

    ])

})

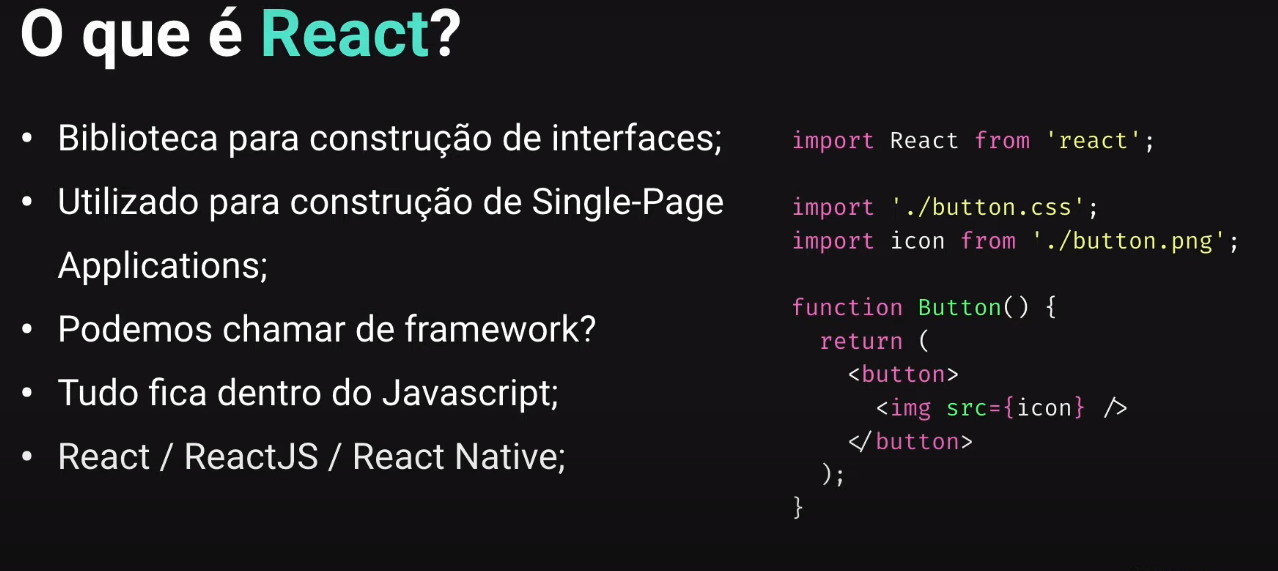
1. Definição da porta.

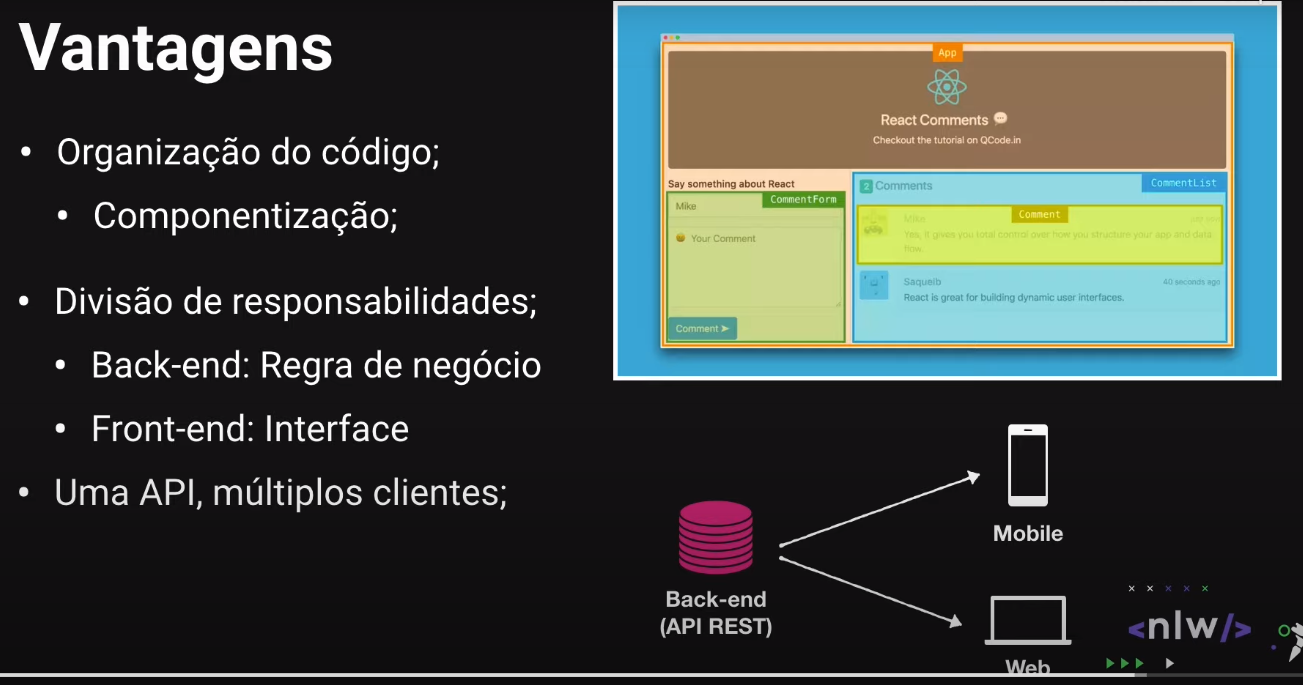
//defini qual a porta de execução da aplicação

app.listen(3333)

## Criação do Projeto - Frontend ReactJS

O react é uma biblioteca para a construção de interfaces. Pode ser considerado um framework devido ao ecossistema completo que se pode utilizar com a instalação da biblioteca.





## Criação do projeto

Comando: npx create-react-app web –template=typescript

## Rodar o projeto

Comando: npm start.

# Aula 2 - Backend

## Conceitos

* Rota: URL da aplicação, endereço completo. As rotas são semânticas, é possível entender a ação que será realizada.
* Recurso: entidade do sistema que se deseja acessar. Ex: /users.

## Rotas

const users = [

    'Carla',

    'Patrícia',

    'Jéssica'

]

//request: obter dados da requisição.

//response: devolver resposta para a requisição feita. Sempre deve proceder do return

//Query Params

app.get('/users', (*request*, *response*) => {

    const search = *String*(*request*.query.search)

    const filteredUsers = search ? users.filter(*user* => *user*.includes(search)): users

    return *response*.json(filteredUsers)

})

//Request Params

app.get('/users/:id', (*request*, *response*) => {

    const id = *Number*(*request*.params.id)

    const user = users[id]

    return *response*.json(user)

})

//Request Body

app.post('/users', (*request*, *response*) => {

    const data = *request*.body

    const user = {

        name: data.name,

        email: data.email

    }

    return *response*.json(user)

})

### Parâmetros

Para o express entender o corpo da requisição é necessário adicionar:

app.use(express.json())

const data = *request*.body

### Separação das Rotas em um arquivo específico

Arquivo: routes.ts

import express from 'express'

//desacopla as rotas do arquivo principal para outro arquivo

const routes = express.Router()

routes.get('/', (*request*, *response*) => {

    return *response*.json({ message: 'Hello World!'  })

})

export default routes

Arquivo principal:

import routes from './routes'

app.use(routes)

## Banco de Dados

SQL: SQLite. Para utilizar o SQLite não precisa instalar nada na máquina. Salva os dados em um arquivo .sqlite.

Knex: query builder: escreve todos os SQL em formato JS. Biblioteca que permite trabalhar com bancos de dados SQL com uma linguagem unificada para todos os bancos. Dessa forma, é possível fazer a migração entre os bancos sem complicações.

Documentação: <http://knexjs.org/>

### Configurando a Conexão

1. Criação da pasta database: conexão com o banco de dados, arquivos relacionados com o banco de dados.
2. Instalação do knex: npm install knex.
3. Criação do arquivo connection.ts: conexão com o banco.
4. Instalar o banco: npm install sqlite3.

Arquivo: connection.ts

import knex from 'knex'

//trabalhar com caminhos no node. Padroniza os caminhos independente do SO

import path from 'path'

//recebe um objeto com as configurações do banco de dados

//filename: qual o arquivo que será armazenado o arquivo do banco de dados

const connection = knex({

    client: 'sqlite3',

    connection: {

        //resolve: une caminhos

        //\_\_dirname: variável global: retorna o diretório do arquivo que está sendo executado.

        filename: path.resolve(\_\_dirname, 'database.sqlite')

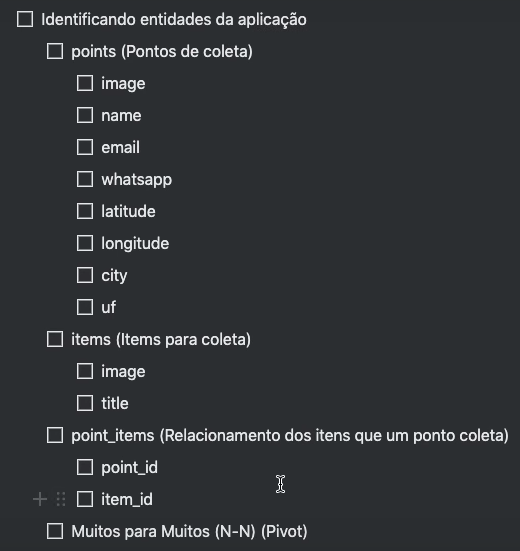
    }

})

export default connection

### Criação das tabelas

1. Identificar as entidades da aplicação.
   1. Tabela pivot: relacionamento N-N. Herda o nome das duas tabelas relacionadas.



1. Criação das tabelas

Poderia utilizar uma interface visual para criação das tabelas, porém, o knex abre a possibilidade de utilizar as migratios, que permite criar as tabelas utilizando JS.

Criação:

Criação de uma nova pasta dentro de database: database/migrations.

A ordem das migrations na pasta é a ordem em que elas serão executadas no banco.

* Criação da tabela points: 00\_create\_points.ts

//importa o tipo da variável

import Knex from 'knex'

//exporta duas funções

//recebe a instância de acesso ao banco de dados: knex

//realiza as alterações no banco

//utiliza o typescript para informar qual o formato da variável e ter acesso a todos os métodos existentes

export async function up(*knex*: *Knex*) {

    //função que recebe a referência para a tabela para a criação dos campos

    return *knex*.schema.createTable('points', *table* => {

*table*.increments('id').primary()

        //salva a referência para a imagem: nome do arquivo

*table*.string('image').notNullable()

*table*.string('name').notNullable()

*table*.string('email').notNullable()

*table*.string('whatsapp').notNullable()

*table*.decimal('latitude').notNullable()

*table*.decimal('longitude').notNullable()

*table*.string('city').notNullable()

*table*.string('uf', 2).notNullable()

    })

}

//desfaz as alterações no banco

export async function down(*knex*: *Knex*) {

    return *knex*.schema.dropTable('points')

}

* Criação da tabela items: 01\_create\_items.ts

import Knex from 'knex'

export async function up(*knex*: *Knex*) {

    return *knex*.schema.createTable('items', *table* => {

*table*.increments('id').primary()

*table*.string('image').notNullable()

*table*.string('title').notNullable()

    })

}

export async function down(*knex*: *Knex*) {

    return *knex*.schema.dropTable('items')

}

* Criação da tabela pivot: 02\_create\_point\_items.ts

import Knex from 'knex'

export async function up(*knex*: *Knex*) {

    return *knex*.schema.createTable('point\_items', *table* => {

*table*.increments('id').primary()

*table*.integer('point\_id')

            .notNullable()

            .references('id')

            .inTable('points')

*table*.integer('item\_id')

            .notNullable()

            .references('id')

            .inTable('items')

    })

}

export async function down(*knex*: *Knex*) {

    return *knex*.schema.dropTable('point\_items')

}

Executar as migrations: Criação do arquivo knexfile.ts na raiz do projeto.

import path from 'path'

//configurações

//não pode utilizar a sintaxe export default, pois o Knex não suporta

*module*.*exports* = {

    client: 'sqlite3',

    connection: {

        filename: path.resolve(\_\_dirname, 'src', 'database', 'database.sqlite')

    },

    //defini a localização da pasta das migrations

    migrations: {

        directory: path.resolve(\_\_dirname, 'src', 'database', 'migrations')

    }

}

Comando para rodar a migration: npx knex migrate:latest –knexfile knexfile.ts migrate:latest.

Knexfile: informa o caminho para o arquivo knexfile.ts

Definição para aceitar a inserção de valores nulos (arquivos: connection.ts e knexfile.ts).

useNullAsDefault: true

Criação do script para não rodar o comando muito grande:

"scripts": {

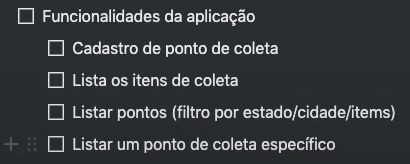
    "dev": "ts-node-dev src/server.ts",

    "knex:migrate": "knex migrate:latest -knexfile knexfile.ts migrate:latest"

  },

Passa a rodar: npm run knex:migrate.

## Funcionalidades



### Cadastro de Items

#### Items pré-cadastrados no sistema

Funcionalidade do Knex: Seed Files.

Criação de uma nova pasta no src/database: seeds.

Arquivo: create\_items.ts

import Knex from 'knex'

export async function seed(*knex*: *Knex*) {

    await knex('items').insert([

        { title: 'Lâmpadas', image: 'lampadas.svg' },

        { title: 'Pilhas e Baterias', image: 'baterias.svg' },

        { title: 'Papéis e Papelão', image: 'papeis-papelao.svg' },

        { title: 'Resíduos Eletrônicos', image: 'eletronicos.svg' },

        { title: 'Resíduos Orgânicos', image: 'organicos.svg' },

        { title: 'Óleo de Cozinha', image: 'oleo.svg' },

    ])

}

Configuração do arquivos knexfile.ts:

//defini a localização da pasta das seeds

sedds: {

    directory: path.resolve(\_\_dirname, 'src', 'database', 'seeds')

},

Comando para executar:

Configuração do script:

"scripts": {

    "dev": "ts-node-dev src/server.ts",

    "knex:seed": "knex --knexfile knexfile.ts seed:run",

    "knex:migrate": "knex --knexfile knexfile.ts migrate:latest"

  },

Rodar: npm run knex:seed

#### Adição das Imagens

Criação de uma nova pasta na raix do projeto com todas as imagens .svg: uploads.

##### Criação da rota de acesso

Função específica do express para servir arquivos estáticos, que há a necessidade de acesso direto. Ex: imagens, pdf, arquivos word.

//função específica do express para servir arquivos estáticos

app.use('/uploads', express.static(path.resolve(\_\_dirname, '..' , 'uploads')))

### Listar todos os itens da base de dados

Processo de transformar as informações da forma que é necessária/mais acessível para visualização do cliente final: serialização de dados.

routes.get('/items', async (*request*, *response*) => {

    //importação da conexão com o banco de dados

    const items = await knex('items').select('\*')

    const serializedItems = items.map(*item* => {

        return {

id: *item*.id,

            title: *item*.title,

            image\_url: `http://localhost:3333/uploads/${*item*.image}`

        }

    })

    return *response*.json(serializedItems)

})

### Pontos de Coleta

Utilizando transações para garantir que se a segunda inserção (relacionamento points\_items) der erro, a primeira não seja persistida (executa o rollback).

//transação - transaction

    const trx = await knex.transaction()

apenas trocar o knex por trx

e adicionar após as transações:

await trx.commit()

Código:

routes.post('/points', async (*request*, *response*) => {

    const {

        name,

        email,

        whatsapp,

        latitude,

        longitude,

        city,

        uf,

        items

    } = *request*.body

    //transação - transaction

    const trx = await knex.transaction()

    //o insert retorna os ids da inserção realizada

    const insertedIds = await trx('points').insert({

        image: 'image-fake',

        name,

        email,

        whatsapp,

        latitude,

        longitude,

        city,

        uf

    })

    const point\_id = insertedIds[0]

    const pointItems = items.map((*item\_id*: *number*) => {

        return {

            item\_id,

            point\_id

        }

    })

    await trx('point\_items').insert(pointItems)

await trx.commit()

    return *response*.json({ sucess: true })

})

### Desacoplamento do código

Criação da pasta controllers no src: 1 controller para cada entidade.

Métodos Padrão Controller:

* Idex: listagem
* Show: exibir único registro
* Create (store)
* Update
* Delete (destroy)
* PointsController

import { Request, Response } from 'express'

import knex from '../database/connection'

class PointsController {

    async create (*request*: *Request*, *response*: *Response*) {

        const {

            name,

            email,

            whatsapp,

            latitude,

            longitude,

            city,

            uf,

            items

        } = *request*.body

        //transação - transaction

        const trx = await knex.transaction()

        const point = {

            image: 'image-fake',

            name,

            email,

            whatsapp,

            latitude,

            longitude,

            city,

            uf

        }

        //o insert retorna os ids da inserção realizada

        const insertedIds = await trx('points').insert(point)

        const point\_id = insertedIds[0]

        const pointItems = items.map((*item\_id*: *number*) => {

            return {

                item\_id,

                point\_id

            }

        })

        await trx('point\_items').insert(pointItems)

await trx.commit()

        //...: spread operator: pega todas as informações do objeto para retornar dentro de outro

        return *response*.json({

            id: point\_id,

            ...point

        })

    }

}

export default PointsController

* ItemsController

import { Request, Response } from 'express'

import knex from '../database/connection'

class ItemsController {

    //index: listagem

    async index (*request*: *Request*, *response*: *Response*) {

        //importação da conexão com o banco de dados

        const items = await knex('items').select('\*')

        const serializedItems = items.map(*item* => {

            return {

                id: *item*.id,

                title: *item*.title,

                image\_url: `http://localhost:3333/uploads/${*item*.image}`

            }

        })

        return *response*.json(serializedItems)

    }

}

export default ItemsController

* Arquivo de rotas modificado:

import express from 'express'

import PointsController from './controllers/PointsController'

import ItemsController from './controllers/ItemsController'

//desacopla as rotas do arquivo principal para outro arquivo

const routes = express.Router()

const pointController = **new** PointsController()

const itemController = **new** ItemsController()

routes.get('/items', itemController.index)

routes.post('/points', pointController.create)

export default routes

## Conceitos para avançar:

* Service Pattern
* Repository Pattern (Data Mapper) – banco de dados

### Listagem ponto de coleta específico

class PointsController {

    async show (*request*: *Request*, *response*: *Response*) {

        const { id } = *request*.params

        const point = await knex('points').where('id', id).first()

        if ( !point ) {

            return *response*.status(400).json({ message: 'Point not found.' })

        }

        const items = await knex('items')

            .join('point\_items', 'items.id', '=', 'point\_items.item\_id')

            .where('point\_items.point\_id', id)

            .select('items.title')

        return *response*.json({ point, items })

    }

Rotas:

routes.get('/points/:id', pointController.show)

### Listagem Pontos utilizando filtros (estado, cidade, items)

class PointsController {

    async index (*request*: *Request*, *response*:*Response*) {

        const { city, uf, items } = *request*.query

        const parsedItems = *String*(items)

            .split(',')

            .map(*item* => *Number*(*item*.trim()))

        //whereIn: busca pelo menos um dos parâmetros enviados

        const points = await knex('points')

            .join('point\_items', 'points.id', '=' , 'point\_items.point\_id')

            .whereIn('point\_items.item\_id', parsedItems)

            .where('city', *String*(city))

            .where('uf', *String*(uf))

            .distinct()

            .select('points.\*')

        return *response*.json(points)

    }

Rotas:

routes.get('/points', pointController.index)

## CORS

CORS (Cross Origin Resource Sharing)

Defini na API quais endereços externos podem fazer acesso.

Instalação: npm install cors

Instalação do tipo typescript: npm install @types/cors -D

Modificação no arquivo server.ts:

import cors from 'cors'

//permite o acesso de todas as URLs

app.use(cors())

Em ambiente de produção tem como definir o domínio que irá acessar:

app.use(cors({

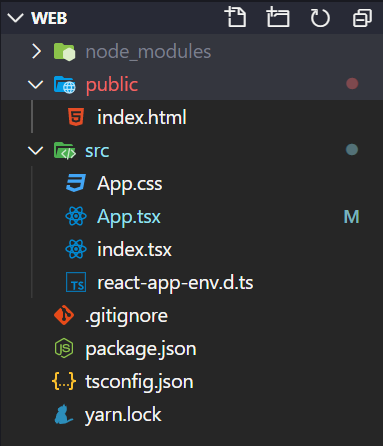
    origin: 'www.'

}))

# Aula 3 – Frontend Web

## Limpar a estrutura da aplicação

Manter apenas os arquivos:



Modificar:

Index.html:

<!DOCTYPE *html*>

<html *lang*="en">

  <head>

    <meta *charset*="utf-8" />

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1" />

    <meta *name*="theme-color" *content*="#34CB79" />

    <title>Ecoleta</title>

  </head>

  <body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div *id*="root"></div>

  </body>

</html>

Index.tsx:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>,

  document.getElementById('root')

);

App.tsx

import React from 'react';

import './App.css';

function App() {

  return (

    <h1>Hello World!</h1>

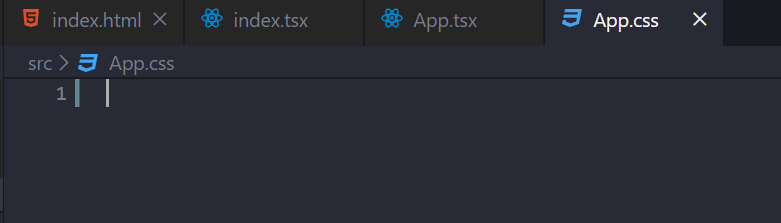
  );

}

export default App;

App.css

Arquivo vazio:



## Conceitos

### Index e id Root

O React é uma biblioteca pelo JS, toda a interface vista em tela é montada pelo JS.

Todo o conteúdo do projeto é mostrado através da id root.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

//render: renderiza (mostra em tela) o APP dentro do element 'root'

ReactDOM.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>,

  document.getElementById('root')

);

### JSX

Possibilidade de escrever HTML dentro do Java Script. JSX: HTML (sintaxe do XML) + JS.

Extensão do arquivo: .tsx.

### Componentes

Componentes: separação da aplicação em blocos que podem ser reutilizados.

Exemplo: Header:

//é sempre necessário importar o React em componentes

//todos componentes começam com a letra maiúscula

import React from 'react'

function Header() {

    return (

        <div>

            <h1>Ecoleta</h1>

        </div>

    )

}

export default Header

App.tsx:

import React from 'react';

import './App.css';

import Header from './Header'

function App() {

  return (

    <*Header* />

  );

}

export default App;

### Propriedade

Atributos que são enviados para o componente.

#### Interface

É a forma utilizada para definir a tipagem de um objeto. Dessa forma, há um controle das informações que são/não são obrigatórias, quais informações o componente recebe/não recebe.

* Informação **não** obrigatória: com ?
* Informação obrigatória: sem ?

#### React.FC

FC – Function Component: é um tipo genérico que informa que o componente está escrito em formato de função. Passa como parâmetro as propriedades criadas na interface do componente.

Header.tsx

import React from 'react'

interface *HeaderProps* {

    title: *string*

    //title?: string

}

//transforma o componente escrito em formato de função para constante - arrow function

//para que seja possível passar as propriedades

//coloca as chaves para trabalhar com JS em HTML

const Header: *React*.*FC* <*HeaderProps*> = (*props*) => {

    return (

        <div>

            <h1>{*props*.title}</h1>

        </div>

    )

}

export default Header

App.tsx

import React from 'react';

import './App.css';

import Header from './Header'

function App() {

  return (

    <div>

      <*Header* *title* = 'Página 1' />

      <*Header* *title* = 'Página 2' />

      <*Header* *title* = 'Página 3' />

      <*Header* *title* = 'Página 4' />

    </div>

  );

}

export default App;

### Estado e imutabilidade

Estado: informações mantidas pelo componente. Sempre que o valor do estado é alterado, ele é refletido na interface. Assim, se o estado for utilizado também em outros componentes distintos, o valor é refletido em toda aplicação em tempo real.

Imutabilidade: não pode alterar a informação de um estado de forma direta, é necessário criar novo valor para o estado com as modificações desejadas.

useState retorna dois parâmetros:



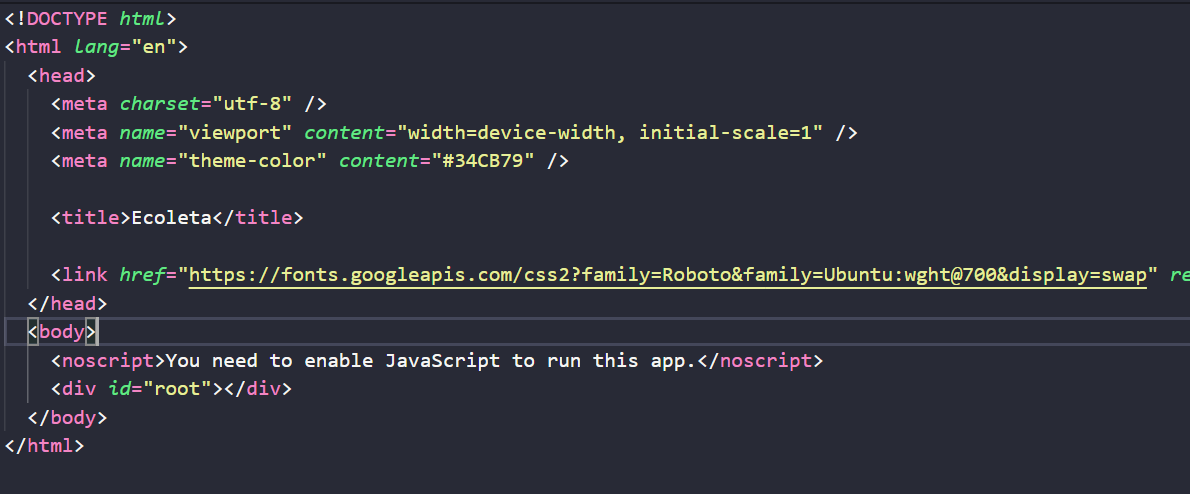


## Desenvolvimento

### Adição de fontes

Site: google fonts. Selecione as fontes desejadas e utilize o link gerado no index.html





### Páginas

#### Landing Page

Criação da pasta src/pages.

Para cada página há uma pasta separada para melhor organização e isolamento do código.

Pages/Home

#### Imagens

Todas as imagens da aplicação devem ser colocadas na pasta: src/assets.

#### Ícones

Pacote do React para ícones: npm install react-icons

Utiliza as fontes mais famosas: Font Awesome, Material Icons, Feather Icons.

#### Mudança entre as Páginas

Sistema de roteamento para fazer a transição entre as páginas.

Instalação: npm install react-router-dom

Instalação do typescript: npm install @types/react-router-dom -D

Criação do arquivo de rotas: src/routes.tsx

import React from 'react'

import { Route, BrowserRouter } from 'react-router-dom'

import Home from './pages/Home'

import CreatePoint from './pages/CreatePoint'

//cada página da aplicação é um Route

//path: qual o endereço que acessa

//exact: faz a verificação exata da url, se não verifica apenas o ínicio da URL e quando da match

//já direciona

const Routes = () => {

    return (

        <*BrowserRouter*>

            <*Route* *component*={Home} *path*='/' *exact* />

            <*Route* *component*={CreatePoint} *path*='/create-point' />

        </*BrowserRouter*>

    )

}

export default Routes

Mudança no App.tsx:

import React from 'react';

import './App.css';

import Routes from './routes'

function App() {

  return (

    <*Routes* />

  );

}

export default App;

##### Páginas SPA

Não faz o carregamento completo da página. Assim, a performance da aplicação aumenta (mais performática), pois o HTML, CSS, código JS é mantido entre as telas evitando o recarregamento dos dados na rede.

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

Trocar a por link e href por to

<*Link* *to*="/create-point">

                        <span>

                            <*FiLogIn* />

                        </span>

                        <strong>Cadastre um ponto de coleta</strong>

                    </*Link*>

#### Create Point

##### HTML

* Fieldset: Agrupar elementos em um formulário.
  + Legend: rótulo para o fieldset.

#### Armazenando dados dos inputs

const [formData, setFormData] = useState({

        name: '',

        email: '',

        whatsapp: ''

    })

function handleInputChange(*event*: *ChangeEvent*<*HTMLInputElement*>) {

        console.log(*event*.target.name, *event*.target.value)

        const { name, value } = *event*.target

        //mantém os dados que já possue e só modifica a alteração

        //se alterar direto as informações dos outros campos não alterados serão perdidas

        //[] - inserir o nome de uma variável

        setFormData({ ...formData, [name]: value })

    }

Para cada input adicionar o onChange

<input

*type*="text"

*name*="name"

*id*="name"

*onChange*={handleInputChange}

                        />

#### Armazenando dados da lista

const [selectedItems, setSelectedItems] = useState<*number*[]>([])

function handleSelectItem(*id*: *number*) {

        //findIndex: retorna um número >= 0 se já existir else -1

        const alreadySelected = selectedItems.findIndex(*item* => *item* === *id*)

        if (alreadySelected >= 0) {

            //pega todos os itens que não é aquele informado

            const filteredItems = selectedItems.filter(*item* => *item* !== *id*)

            setSelectedItems(filteredItems)

        } else {

            setSelectedItems([ ...selectedItems, *id* ])

        }

        console.log(selectedItems)

    }

Muda a classe se tiver selecionado

<ul *className*='items-grid'>

                        {items.map(*item* => (

                            <li

*key*={*item*.id}

*onClick*={() => handleSelectItem(*item*.id)}

*className*={selectedItems.includes(*item*.id) ? 'selected' : ''}

                            >

                                <img *src*={*item*.image\_url} *alt*={*item*.title}/>

                                <span>{*item*.title}</span>

                            </li>

                        ))}

                    </ul>

#### Submit do formulário

Remover o reload da página:

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

<form *onSubmit*={handleSubmit}>

async function handleSubmit(*event*: *FormEvent*) {

        //remove o reload da página

*event*.preventDefault()

        const { name, email, whatsapp } = formData

        const uf = selectedUf

        const city = selectedCity

        const [ latitude, longitude ] = selectedPosition

        const items = selectedItems

        const data = {

            name,

            email,

            whatsapp,

            uf,

            city,

            latitude,

            longitude,

            items

        }

        await api.post('points', data)

        alert('Ponto de coleta cadastrado!')

        //redireciona o usuário para a página anterior

        history.push('/')

    }

### Leaflet

Mapa de código aberto.

Link: <https://leafletjs.com/>

Biblioteca que permite integrar o mapa com o projeto: react-leaftlet.

Documentação: <https://react-leaflet.js.org/>

Instalação: npm install leaflet react-leaflet

Npm install @types/react-leaflet -D

1. Importação do css do leaftlet. Link: <https://leafletjs.com/examples/quick-start/>

#### Inserindo o mapa:

* Center/position: localização inicial: latitude/longitude.
* TileLayer: design.

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

<*Map* *center*={[ -20.5206602, -43.7046644 ]} *zoom*={15}>

                        <*TileLayer*

*attribution*='&amp;copy <a href="http://osm.org/copyright">OpenStreetMap</a> contributors'

*url*="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"

                        />

                        <*Marker* *position*={[ -20.5206602, -43.7046644 ]} />

                    </*Map*>

#### Selecionando posição no mapa

import { LeafletMouseEvent } from 'leaflet'

const [selectedPosition, setSelectedPosition] = useState<[*number*, *number*]>([0, 0])

function handleMapClick(*event* : *LeafletMouseEvent*) {

        setSelectedPosition([

*event*.latlng.lat,

*event*.latlng.lng

        ])

    }

<*Marker* *position*={selectedPosition} />

#### Carregamento inicial da posição com base na localização do usuário

const [initialPosition, setInitialPosition] = useState<[*number*, *number*]>([0, 0])

Utiliza a API do próprio navegador.

useEffect(() => {

        //navigator: variável global

        navigator.geolocation.getCurrentPosition(*position* => {

            const { latitude, longitude } = *position*.coords

            setInitialPosition([ latitude, longitude ])

        })

    }, [])

<*Map* *center*={initialPosition} *zoom*={15} *onClick*={handleMapClick}>

### Consumindo endpoint da API

#### Axios

Permite fazer requisições para o backend. Vantagem: definição da baseURL: url que irá se repetir em todas as requisições, assim, quando colocar a API em produção, ou trocar o endereço basta trocar a URL em apenas um lugar. Dessa forma, é mais vantajoso utilizar o axios do que a API fetch padrão do navegador para fazer requisições.

Instalação: npm install axios

Criação da pasta src/services: obtenção de recursos externos

Arquivo: api.ts

import axios from 'axios'

const api = axios.create({

    baseURL: 'http://localhost:3333'

})

export default api

#### useEffect

Função do react para fazer o carregamento inicial apenas uma vez e não todas as vezes que o componente sofrer alteração.

//[] - 1 única vez quando o componente for carregado

//[estado] - executa toda vez que o estado específico for alterado

//não é possivel utilizar async await no useEffect, deve tratar a promisse com then()

//then: assim que tiver a resposta

useEffect(() => {

        api.get('items').then(*response* => {

            setItems(*response*.data)

        })

    }, [])

#### Retornando dados

Sempre ao utilizar array ou objeto, é necessário informar qual o tipo da variável.

interface *Item* {

    id: *number*,

    title: *string*,

    image\_url: *string*

}

const [items, setItems] = useState<*Item*[]>([])

<ul *className*='items-grid'>

                        {items.map(*item* => (

                            <li *key*={*item*.id}>

                                <img *src*={*item*.image\_url} *alt*={*item*.title}/>

                                <span>{*item*.title}</span>

                            </li>

                        ))}

                    </ul>

### API IBGE

Buscar estados e cidades pela API do IBG.

Link: <https://servicodados.ibge.gov.br/api/docs/localidades?versao=1#api-Municipios-estadosUFMunicipiosGet>

#### UFs

Nova URL: importar axios

import axios from 'axios'

interface *IBGEUFResponse*{

    sigla: *string*

}

useEffect(() => {

        axios.get<*IBGEUFResponse*[]>('https://servicodados.ibge.gov.br/api/v1/localidades/estados').then(*response* => {

            const ufInitials = *response*.data.map(*uf* => *uf*.sigla)

            setUfs(ufInitials)

        })

    }, [])

<select *name*="uf" *id*="uf">

                                <option *value*="0">Selecione uma UF</option>

                                {ufs.map(*uf* => (

                                    <option *key*={*uf*} *value*={*uf*}>{*uf*}</option>

                                ))}

                            </select>

#### Cidades

As cidades serão carregadas sempre que a UF mudar.

import React, { useEffect, useState, ChangeEvent } from 'react'

value: sempre que o estado for modificado ele deve refletir nessa parte do código

<select

*name*="uf"

*id*="uf"

*value*={selectedUf}

*onChange*={handleSelectUf}

                            >

                                <option *value*="0">Selecione uma UF</option>

                                {ufs.map(*uf* => (

                                    <option *key*={*uf*} *value*={*uf*}>{*uf*}</option>

                                ))}

                            </select>

Define o tipo do evento como ChangeEvent (Mudança no evento) do elemento select do HTML.

function handleSelectUf(*event*: *ChangeEvent*<*HTMLSelectElement*>) {

        const uf = *event*.target.value

        setSelectedUf(uf)

    }

Executa sempre que a UF mudar

useEffect(() => {

        if (selectedUf === '') return

        axios.get<*IBGECityResponse*[]>

            (`https://servicodados.ibge.gov.br/api/v1/localidades/estados/${selectedUf}/municipios`)

            .then(*response* => {

                const cityNames = *response*.data.map(*city* => *city*.nome)

                setCities(cityNames)

        })

    }, [selectedUf])

function handleSelectCity(*event*: *ChangeEvent*<*HTMLSelectElement*>) {

        const city = *event*.target.value

        setSelectedCity(city)

    }

<select

*name*="city"

*id*="city"

*value*={selectedCity}

*onChange*={handleSelectCity}

                            >

                                <option *value*="0">Selecione uma cidade</option>

                                {cities.map(*city* => (

                                    <option *key*={*city*} *value*={*city*}>{*city*}</option>

                                ))}

                            </select>

# Aula 4 – Frontend – Mobile

## Instalações

Expo: link com os principais problemas da rocketseat: <https://github.com/Rocketseat/expo-common-issues>

### Expo CLI

Executar comandos expo pelo terminal.

Comando: npm install -g expo cli

### Criação do Projeto

Comando: expo init mobile

Opção: blank (TypeScript)

Depois da instalação entre na pasta do projeto e rode: npm start.

### Execução

#### Celular

Baixar o app expo. Para rodar deve estar na mesma rede.

Conexão: LAN. Se não abrir clique no + e insira manualmente a URL.

Tunnel: cria um endereço acessível online.

Abrir o app: expo start

## Conceitos

### Diferenças entre React Native e React JS

|  |  |  |
| --- | --- | --- |
|  | React Native | React JS |
| Tags | Tags dos elementos do React Native. Tudo é View. | Tags HTML |
| Estilização | * Propriedade styles disponível em todos os elementos utilizando CSS. Única diferença é que a separação por hífen passa a ser por camealCase. * Envio de um objeto JS, assim, deve-se colocar aspas. * Tem o padrão display flex. * Não há a herança de estilos: não é possível estilizar o elemento Text de forma geral que irá se repetir para todas as utilizações. Deve-se criar estilos específicos. * Não há a cascata: o estilo de um elemento pai não será refletido no elemento filho. | Cascata. |

Os demais elementos do ReactJS são os mesmos para o React Native: propriedade, estado, componentização, imutabilidade – é o mesmo JS.

## Desenvolvimento

1. Criação da pasta src.

Para retornar dois elementos adjacentes é necessário ter um elemento que os englobe: view ou fragment (não reproduz elemento em tela): <> </>..

### Páginas

Criação da pasta src/pages

#### Home

Pasta pages/Home

Definição da cor para a status bar

import { StatusBar } from 'react-native'

<*StatusBar* *barStyle*="dark-content" *backgroundColor*="transparent" *translucent*/>

backgroundColor e translucent (a status bar fica por cima do conteúdo, assim, é possível adicionar conteúdo por baixo da status bar(cor)) são opções para o android.

### Imagens

Sempre as imagens no react native (aplicações mobile) devem ser exportadas em 3 tamanhos diferentes. Pois, os dispositivos possuem densidades de pixels diferentes: casos em que cabem mais pixels na tela de um dispositivo quando comparado a outro.

Pasta: src/assets.

1. Original
2. 2x tamanho original
3. 3x tamanho original

No momento de importar a logo não precisa colocar os 3 tipos, ele já identifica e importa de acordo com o dispositivo.

Importação e uso para projetos typescript:

import { View, Image } from 'react-native'

<*Image* *source*={require('../../assets/logo.png')} />

### Fontes

Documentação: <https://github.com/expo/google-fonts?utm_campaign=React%2BNative%2BNow&utm_medium=web&utm_source=React_Native_Now_65>

Instalação:

expo install expo-font @expo-google-fonts/nome\_da\_fonte expo-font

expo install expo-font @expo-google-fonts/ubuntu @expo-google-fonts/roboto

import { AppLoading } from 'expo'

import { Roboto\_400Regular, Roboto\_500Medium } from '@expo-google-fonts/roboto'

import { Ubuntu\_700Bold, useFonts } from '@expo-google-fonts/ubuntu'

const [fontsLoaded] = useFonts({

    Roboto\_400Regular,

    Roboto\_500Medium,

    Ubuntu\_700Bold

  })

  //AppLoading: fica mostrando o carregamento

  if (!fontsLoaded) return AppLoading

### Navegação

#### React Navigation

Instalação: npm install @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

npm install @react-navigation/stack

* StackNavigator: navegação em pilhas. A navegação é feita através de botões e as telas anteriores permanecem em pilhas, ou seja, o usuário consegue voltar.
* Navegação por abas: abas em cima ou embaixo da tela.
* Navegação por Drawer: menu que abre da esquerda para a direita.

#### Criação das Rotas

Criação do arquivo src/routes.tsx

import React from 'react'

import { NavigationContainer } from '@react-navigation/native'

import { createStackNavigator } from '@react-navigation/stack'

import Home from './pages/Home'

import Points from './pages/Points'

import Detail from './pages/Detail'

//criação do roteamento

const AppStack = createStackNavigator()

const Routes = () => {

    //NavigationContainer: definição do comportamento das rotas

    //Screen: utilizado para cada tela

    //headerMode: remove o header da tela

    //coloca um background que será aplicado em todas as telas

    return (

        <*NavigationContainer*>

            <*AppStack.Navigator*

*headerMode*="none"

*screenOptions*={{

                    cardStyle: {

                        backgroundColor: '#f0f0f5'

                    }

                }}

            >

                <*AppStack.Screen* *name*="Home" *component*={Home} />

                <*AppStack.Screen* *name*="Points" *component*={Points} />

                <*AppStack.Screen* *name*="Detail" *component*={Detail} />

            </*AppStack.Navigator*>

        </*NavigationContainer*>

    )

}

export default Routes

importação no arquivo principal:

import Routes from './src/routes'

<*Routes* />

#### Colocando a navegação nos botões

import { useNavigation } from '@react-navigation/native'

const navigation = useNavigation()

    function handleNavigateToPoints() {

      navigation.navigate('Points')

    }

<*RectButton* *style*={styles.button} *onPress*={handleNavigateToPoints}>

### Botões

RectButton: Botões retangulares que utilizam cor de fundo.

import { RectButton} from 'react-native-gesture-handler'

<*View* *style*={styles.footer}>

              <*RectButton* *style*={styles.button} *onPress*={() => {}}>

                <*View* *style*={styles.buttonIcon}>

                  <*Text*>

                    <*Icon* *name*="arrow-right" *color*="#FFF" *size*={24} />

                  </*Text*>

                </*View*>

                <*Text* *style*={styles.buttonText}>Entrar</*Text*>

              </*RectButton*>

            </*View*>

### Ícones

Não precisa de instalação, basta apenas importar.

import { *Feather* as Icon } from '@expo/vector-icons'

<*Icon* *name*="arrow-right" *color*="#FFF" *size*={24} />

#### Points

Constants

Instalação: expo install expo-constants

import Constants from 'expo-constants'

Botao de voltar: TouchableOpacity: botão quando clicado diminui a opacidade.

##### Mapa

Instalação: expo install react-native-maps

import MapView from 'react-native-maps'

<*View* *style*={styles.mapContainer}>

                    <*MapView* *style*={styles.map} />

                </*View*>

##### Imagens SVG

React Native por padrão não reconhece svg.

Instalação: expo install react-native-svg

import { SvgUri } from 'react-native-svg'

<*SvgUri* *width*={42} *height*={42} *uri*="http://192.168.2.108:3333/uploads/lampadas.svg" />

##### Container dos Items

<*View* *style*={styles.itemsContainer}>

                <*ScrollView*

*horizontal*

*showsHorizontalScrollIndicator*={false}

*contentContainerStyle*={{ paddingHorizontal: 20 }}

                >

                  <*TouchableOpacity* *style*={styles.item} *onPress*={() => {}}>

                    <*SvgUri* *width*={42} *height*={42} *uri*="http://192.168.2.108:3333/uploads/lampadas.svg" />

                    <*Text* *style*={styles.itemTitle}>Lâmpadas</*Text*>

                  </*TouchableOpacity*>

###### Mudança do endereço do server

Mudar o endereço pois estava como localhost, o app não consegue acessar: endereço da máquina que aparece no expo.

Server/itemsController

image\_url: `http://192.168.2.108:3333/uploads/${*item*.image}`

### Consumindo dados da API

Criação da pasta src/services, arquivo api.ts:

#### Axios

Intsalaçaõ: npm install axios

 import axios from 'axios'

 const api = axios.create({

     baseURL: 'http://192.168.2.108:3333'

 })

 export default api

#### Localização do usuário

Acesso a geolocalização

Instalação: npm install expo location

import \* as Location from 'expo-location'

const [initialPosition, setInitialPosition] = useState<[*number*, *number*]>([0, 0])

useEffect(() => {

      async function loadPosition() {

        //permissão do usuário para acessa a localização

        //status: usuário deu ou não permissão

        const { status } = await Location.requestPermissionsAsync()

        if (status !== 'granted') {

*Alert*.alert('Oooops', 'Precisamos de sua permissão para obter a localização')

          return

        }

        const location = await Location.getCurrentPositionAsync()

        const { latitude, longitude } = location.coords

        console.log(latitude, longitude)

        setInitialPosition([

          latitude,

          longitude

        ])

      }

      loadPosition()

    }, [])

#### Acessando informações da rota ao navegar para outra

function handleNavigateToDetail(*id*: *number*) {

      //as informações enviadas no objeto serão reconhecidas como parâmetro na próxima rota

      navigation.navigate('Detail', { point\_id: *id* })

    }

*onPress*={() => handleNavigateToDetail(*point*.id)}

Página que usará os dados:

import { useNavigation, useRoute } from '@react-navigation/native'

const route = useRoute()

const routeParams = route.params as *Params*

### Conexão com Email e Whatsapp

#### Email

Mail Compose. Documentação: <https://docs.expo.io/versions/latest/sdk/mail-composer/>

Instalação: expo install expo-mail-composer

import \* as MailComposer from 'expo-mail-composer';

function handleComposeMail() {

      //envio das opções

      //subject: assunto do email

      //recipients: para quem será enviado

      //body: corpo da mensagem

      MailComposer.composeAsync({

        subject: 'Interesse na coleta de resíduos',

        recipients: [data.point.email],

      })

    }

<*RectButton* *style*={styles.button} *onPress*={handleComposeMail}>

#### WhatsApp

Linking: abre links externos (endereços no browser) e links internos (aplicativos do celular, através do deep linking).

import { Linking } from 'expo'

function handleWhatsapp() {

      Linking.openURL(`whatsapp://send?phone=${data.point.whatsapp}&text=Tenho interesse sobre coleta de resíduos`)

    }

<*RectButton* *style*={styles.button} *onPress*={handleWhatsapp}>

#### Consumindo dados externos

Para fazer select no react native: <https://www.npmjs.com/package/react-native-picker-select>

Input:

import { View, Image, StyleSheet, Text, ImageBackground, TextInput, KeyboardAvoidingView, Platform } from 'react-native'

<*TextInput*

*style*={styles.input}

*placeholder*="Digite a UF"

*maxLength*={2}

                  //deixa todos os caracteres em caixa alta

*autoCapitalize*="characters"

                  //retirar o auto corrigir

*autoCorrect*={false}

*value*={uf}

                  //o parâmetro que recebe é o mesmo passado para a função: pode renomear

                  //onChangeText={text => setUf(text)}

*onChangeText*={setUf}

                />

                <*TextInput*

*style*={styles.input}

*placeholder*="Digite a cidade"

*autoCorrect*={false}

*value*={city}

*onChangeText*={setCity}

                />

Subir o conteúdo quando ativar o teclado: tag por volta de todo o conteúdo. Para conteúdos que não subirem coloque um tag view vazia por volta deles.

<*KeyboardAvoidingView*

*style*={{ flex: 1 }}

*behavior*={Platform.OS === 'ios' ? 'padding' : undefined}>

function handleNavigateToPoints() {

      navigation.navigate('Points', {

        uf,

        city

      })

    }

# Funcionalidades Avançadas

## Upload de Imagens

### Backend

Instalação da extensão: npm install multer

Instalação dos tipos: npm install @types/multer -D

fileFilter: Aplicar restrição de quais arquivos podem fazer upload: tipo do arquivo(png, pdf), tamanho dos arquivos.

Documentação: <https://github.com/expressjs/multer>

Criação da pasta src/config/multer.ts

import multer from 'multer'

import path from 'path'

import crypto from 'crypto'

export default {

    //destination: localização da pasta onde serão salvos os arquivos enviados

    //filename:

    //gerar nome único para os arquivos: crypto

    storage: multer.diskStorage({

        destination: path.resolve(\_\_dirname, '..', '..', 'uploads'),

        filename(*request*, *file*, *callback*) {

            const hash = *crypto*.randomBytes(6).toString('hex')

            const fileName = `${hash}-${*file*.originalname}`

            //primeiro parâmetro seria o erro. colocar um try catch para situações que teria erro

            callback(null, fileName)

        }

    })

}

Modificações arquivo routes.ts

//biblioteca de upload de imagens

import multer from 'multer'

//configuraçaõ da bibioteca

import multerConfig from './config/multer'

const upload = multer(multerConfig)

//single: receber apenas 1 arquivo

//mais de um arquivo: upload.array()

//nome\_do\_campo

routes.post('/points', upload.single('image'), pointController.create)

Insomnia:

Passa a utilizar o formData para envio do nome do arquivo na requisição.

Multipart Form: tipo de requisição que aceita arquivos

### Modificações no WEB

React Dropzone: criação de uma área para jogar arquivos.

Documentação: <https://react-dropzone.js.org/>

<https://github.com/react-dropzone/react-dropzone>

Instalação: npm install react-dropzone

Criação da pasta src/componentes/Dropzone/index.tsx

import React, {useCallback, useState} from 'react'

import {useDropzone} from 'react-dropzone'

import { FiUpload } from 'react-icons/fi'

import './styles.css'

//define a função que será recebida

interface *Props* {

  onFileUploaded: (*file*: *File*) => *void*

}

const Dropzone: *React*.*FC*<*Props*> = ({ *onFileUploaded* }) => {

  const [selectedFileUrl, setSelectedFileUrl] = useState('')

  const onDrop = useCallback(*acceptedFiles* => {

    //0: apenas 1 arquivo

    const file = *acceptedFiles*[0]

    //criação da URL:

    const fileUrl = *URL*.createObjectURL(file)

    setSelectedFileUrl(fileUrl)

    onFileUploaded(file)

  }, [onFileUploaded])

  //accept: 'image/\*': aceita qualquer tipo de imagem

  const { getRootProps, getInputProps } = useDropzone({

    onDrop,

    accept: 'image/\*'

  })

  //receber mais de um arquivo: adicionar a propriedade multiple no input

  return (

    <div *className*="dropzone" {...getRootProps()}>

      <input {...getInputProps()}  *accept*= "image/\*" />

      { selectedFileUrl

        ? <img *src*={selectedFileUrl} *alt*="Point thumbnail" />

        : (

          <p>

            <*FiUpload* />

            Imagem do estabelecimento

          </p>

        )

      }

    </div>

  )

}

export default Dropzone

Utilização no arquivo CreatePoint/index.tsx

#### Comunicação de informação entre componente filho e pai

Cria a propriedade. Envia uma função para o componente filho da árvore.

Assim, você tem o envio de uma informação que está disponível apenas para o componente filho, ser acessível no componente pai.

import DropZone from '../../components/Dropzone'

const [selectedFile, setSelectedFile] = useState<*File*>()

async function handleSubmit(*event*: *FormEvent*) {

        //remove o reload da página

*event*.preventDefault()

        const { name, email, whatsapp } = formData

        const uf = selectedUf

        const city = selectedCity

        const [ latitude, longitude ] = selectedPosition

        const items = selectedItems

        //FormData: classe global do JS para o envio do multipart/form-data

        const data = **new** *FormData*()

        data.append('name', name)

        data.append('email', email)

        data.append('whatsapp', whatsapp)

        data.append('uf', uf)

        data.append('city', city)

        data.append('latitude', *String*(latitude))

        data.append('longitude', *String*(longitude))

        data.append('items', items.join(','))

        if (selectedFile) data.append('image', selectedFile)

        await api.post('points', data)

        alert('Ponto de coleta cadastrado!')

        //redireciona o usuário para a página anterior

        history.push('/')

    }

<*DropZone* *onFileUploaded*={setSelectedFile} />

## Validação da entrada de dados

Frontend: Yup: <https://github.com/jquense/yup>

Backend: celebrate – para express

Projetos sem express: joi schema validation. Documentação: <https://github.com/hapijs/joi>

Documentação: <https://github.com/arb/celebrate>

Instalação: npm install celebrate

Tipos: npm install @types/hapi\_\_joi -D

É possível validar o body, query params, params, headers, cookies..

É possível configurar a mensagem de retorno.

Normalmente é isolado em outro arquivo.

Arquivo routes.ts

//validação dos dados

import { celebrate, Joi } from 'celebrate'

routes.post(

    '/points',

    upload.single('image'),

    celebrate({

        body: Joi.object().keys({

            name: Joi.string().required(),

            //email: verifica se tem o @ com domínio

            email: Joi.string().required().email(),

            whatsapp: Joi.number().required(),

            latitude: Joi.number().required(),

            longitude: Joi.number().required(),

            city: Joi.string().required(),

            uf: Joi.string().required().max(2),

            //possível utilizar regex()

            items: Joi.string().required()

        })

    }, {

        //verifica todos os campos, e não para a execução no primeiro erro que enocntrar.

        //dessa forma, retorna todos os campos que deram erro

        abortEarly: false

    }),

    pointsController.create

)

Arquivo server.ts

import { errors } from 'celebrate'

//no final antes do listen

//retorno dos erros para o frontend para nao dar erro 500 (servidor) e sim bad request (400)

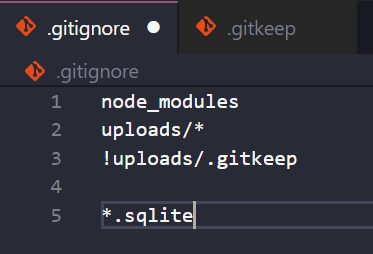
app.use(errors())

## Envio projeto GitHub

Para enviar os arquivos para o GitHub deve sempre criar o arquivo .gitignore e adicionar o node\_modules para não ser enviado para o github.

Criação do arquivo .gitkeep para subir a pasta uploads.

Ignora todos os arquivos com extensão sqlite



Como dar commit pela linha de comando:

1. Crie o repositório na web.
2. Entre na pasta do projeto e digite: git init
3. Comando: Git add .
4. Comando: Git commit -m “App”
5. Copie a URL do github: git remote add origin ul\_repositorio
6. Comando: Git push origin master

## Deploy

Tem vídeos no youtube da rocketseat ensinando como fazer : heroku, digital ocean.

### Mobile

Lojas de aplicativos: Google Play (aprox.. 30 dólares uma única vez, sem expiração), App Store (aprox.. 99 dólares por ano, plano PF – pessoa física).

### Backend

#### Heroku

**Foco**: aplicação de Teste: mostrar para as outras pessoas como ficou. NÃO utilizar o plano gratuito para entrar em produção.

Faz o gerenciamento da escalabilidade – plano a partir de 7 dólares, porém, é mais vantajoso utilizar a digital ocean para esse caso.

#### Digital Ocean

**Foco**: não possui muitos usuários utilizando.

Plano previsíveis: sabe de antemão o gasto (cálculo do servidor..).

#### AWS / Google Cloud / Microsoft Azure

**Foco**: grande número de usuários, aplicações grandes.

Desejável: existir uma pessoa para fixar responsável só pelo deploy – SRE.

### Frontend

#### Netlify / Vercel

Link: <https://www.netlify.com/pricing/>

Plano gratuito que é acessível para a grande maioria das aplicações.

Para fazer o deploy basta apenas colocar o link do repositório do git que ele faz o deploy automático.

Vercel: <https://vercel.com/>

#### Amazon S3 / Google Cloud Storage

Aplicações grandes.

Amazon S3 / Google Cloud Storage: para hospedar arquivos estáticos (projeto React – HTML, CSS, JS).

# VSCode

