Projeto Classificatório – Rocky

Olá, meu nome é João Guilherme Zanardo e atualmente tenho 17 anos. Por meio deste arquivo, irei mostrar o sistema que desenvolvi com base nos requisitos que vocês pediram.

Conforme descrito no documento explicativo do sistema, deveríamos criar um sistema que formata dois bancos de dados corrompidos em formato JSON e, após formatados, era necessário ir à plataforma **sqliteonline.com** para criar uma tabela única que contenha todos os dados necessários para formar um relatório de vendas por meio de códigos SQL.

No entanto, decidi fazer de uma maneira diferente. Meu sistema executa todas essas tarefas automaticamente. Em resumo, ele:

1. Corrige as databases corrompidas.
2. Cria arquivos JSON para armazenar as databases corrigidas, e os guarda em uma pasta específica.
3. Unifica as duas databases corrigidas, criando uma única database que contenha todos os dados necessários para compilar um relatório de vendas completo.
4. Converte a base de dados unificada para o formato CSV e salva o arquivo resultante na área de trabalho do cliente.
5. Cria planilhas no Excel contendo todos os detalhes de faturamento de cada carro e marca, em seguida, salva-as na Área de trabalho do cliente.

Em resumo, essas são as funcionalidades do meu sistema. Achei que seria interessante adicionar algumas funcionalidades novas, como a geração de planilhas que mostram o faturamento de cada carro e marca. Isso permite que o usuário final tenha acesso a todos os detalhes de uma database que antes estava corrompida e foi corrigida.

Apenas para concluir, gostaria de informar que utilizei Node.js com TypeScript em vez de JavaScript. Já faz algum tempo que parei de usar JavaScript e migrei para o TypeScript, e achei interessante trazê-lo para este projeto. Também não dei muita importância para a arquitetura do sistema, acredito que isso não seja um problema, já que não há necessidade de tamanha complexidade. Em breve, explicarei detalhadamente o sistema automático que desenvolvi.

Biblioteca e ferramentas:

- SQL
- NPM
- Typescript
- Git
- @types/ node
- @types/json2csv
- @aternus/csv-to-xlsx

Funções criadas:

- formatCorruptedString
- formatBrokenDB1
- formatBrokenDB2
- formatter
- join
- createSoldBrandsDB
- createSoldCarsDB
- jsonToCsv

Explicando cada função

- formatCorruptedString

```
export const formatCorruptedString = (str: string): string => {
    for (const letter of str) {
        if (letter === 'æ') {
            str = str.replace('æ', 'a');
        }
        if (letter === "ø") {
            str = str.replace('ø', 'o');
        }
    }

    return str;
}
```

A função **formatCorruptedString** recebe uma **string** como parâmetro. Em seguida, um loop é executado para obter todas as letras da string. Verifica-se se alguma dessas letras é igual a **'æ'** ou **'ø'**. Se houver, essas letras são substituídas pela letra correta (**'æ'** é substituído por 'a' e **'ø'** é substituído por 'o'). A função retorna a string corrigida.

- formatBrokenDB1

```
export const formatBrokenDB1 = (brokenDB1: BrokenDB1): Car[] => {
    const carsDB: Car[] = [];

    for (const car of brokenDB1) {
        car.nome = formatCorruptedString(car.nome);
        car.vendas = Number(car.vendas);

        carsDB.push({
            data: car.data,
            id_marca_: car.id_marca_,
            nome: car.nome,
            vendas: car.vendas,
            valor_do_veiculo: car.valor_do_veiculo
        });
    }

    return carsDB;;
};
```

Essa função incialmente cria um array, a qual é nomeada **CarsDB** já que a primeira database se trata de carros. Logo após, efetua um loop sobre a database corrompida. Dentro desse loop, é setado o nome do carro como sendo o retorno da função **formatCorruptedString**, já que ela corrige o nome do carro que esteja corrumpido.

Nessa parte do processo, é setado a propriedade **vendas** para o tipo number. Mesmo que a maioria já sejam números, aqueles que eram em formato string passaram a ser number também, resolvendo mais um problema da database corrompida.

Feito isso, é efetuado um push no **carsDB**, colocando nele os dados corrigidos, com os **nomes** corretos e com as **vendas** em formado number.

Após finalizado o loop, a função retorna o **carsDB**, um JSON que contêm os dados da primeira database corrigidos.

- formatBrokenDB2

```
export const formatBrokenDB2 = (brokenDB2: BrokenDB2): Brand[] => {
    const brandsDB: Brand[] = [];

    for (const brand of brokenDB2) {
        brand.marca = formatCorruptedString(brand.marca);
        brandsDB.push({
            id_marca: brand.id_marca,
            marca: brand.marca
        });
    }

    return brandsDB;
};
```

A função **formatBrokenDB2** é executada da mesma maneira que a função **formatBrokenDB1**. É criado um novo array com o nome **"brandsDB"** para armazenar os dados da segunda database corrompida, que contém o **ID** e o nome da marca.

Após a criação do array, cada marca é corrigida utilizando a função **formatCorruptedString**, que foi mencionada anteriormente como responsável por corrigir strings corrompidas.

Ao final do loop, os dados corrigidos são adicionados ao array **brandsDB**, que é retornado como resultado da função

- formatter

```
export const formatter = (brokenDB1: BrokenDB1, brokenDB2: BrokenDB2):
FormatterReturn => {
    const carsDB = formatBrokenDB1(brokenDB1);
    const brandsDB = formatBrokenDB2(brokenDB2);

    return {carsDB, brandsDB};
};
```

A função **formatter** é na realidade bem simples, ela apenas recebe as duas databases corrompidas e chamas as duas funções que iram corrigi-las. Após isso é retornanado os dois arquivos JSON, que são as duas databases com os dados corrigidos.

- join

```
export const join = (carsDB: Car[], brandsDB: Brand[]): FinalDB => {
    let finalDB: FinalDB = [];

    carsDB.map(car => {
        brandsDB.map(brand => {
            if (car.id_marca_ === brand.id_marca) [
                finalDB.push({
                    data: car.data,
                    marca: brand.marca,
                    nome: car.nome,
                    vendas: car.vendas,
                    valor: car.valor_do_veiculo
                })
            ]
        });
    });

    return finalDB;
}
```

A função **join** é executada como uma substituição da função JOIN do SQL, a qual possibilita a junção duas ou mais tabelas.

O objetivo dessa função é retornar um JSON que substitua, em **carsDB,** a propriedade **id_marca_** por “**marca”**, e ao invés de colocar o **id** da marca, colocar o **nome** dela.

A função recebe dois parâmetos, **carsDB** e **brandsDB**, que são as databases corrigidas.

Em seguida é executado um loop no **carsDB**, seguido por outro loop no **brandsDB**. Efetuando esse segundo loop, é possível encontrar o nome exato da marca de cada carro, para que em seguida seja possível colocar dentro do array **finalDB** todas as informações do carro com o nome da sua marca.

Por fim, o **finalDB**, que contém as informações necessárias para gerar o relatório de vendas, é retornado como um JSON. Agora, apenas basta formatar esse JSON para CSV.

* join com SQL

```
SELECT
cars.data, brands.marca, cars.nome, cars.vendas, cars.valor_do_veiculo as
valor
FROM cars
INNER JOIN
brands
ON
cars.id_marca_ = brands.id_marca
```

Como citei no inicio do documento, decidi desenvolver a função **‘join’** para deixar tudo de forma automatizada e não precisar ir ao **sqliteonline** juntar as

duas databases. Porém, aqui também estarei colocando o código **SQL** que eu utilizaria (já que isso foi o solicitado desde o início) para montar a unificação das tabelas e gerar a tabela perfeita para o relatório de vendas.

- createSoldBrandsDB

```
export const createSoldBrandsDB = (finalDB: FinalDB): Soldbrand[] => {
    const soldBrands: Soldbrand[] = [];
    for (const car of finalDB) {
        let exists = false;
        if (soldBrands.length) {
            for (const brand of soldBrands) {
                if (car.marca === brand.nome) {
                    brand.vendas += car.vendas;
                    brand.total += car.valor * car.vendas;
                    exists = true;
                }
            }
        }

        if (!exists) {
            soldBrands.push({
                nome: car.marca,
                vendas: car.vendas,
                total: car.vendas * car.valor
            })
        }
    }

    return soldBrands;
}
```

Esse código exporta uma função chamada **createSoldBrandsDB** que recebe como parâmetro uma base de dados chamada **finalDB**. A função processa essa base de dados e retorna outra base de dados chamada **soldBrands** que contém informações sobre as vendas de cada marca de carro.

O processo da função consiste em iterar sobre cada carro na base de dados **finalDB**, verificar se já existe uma entrada para a marca do carro na base de dados **soldBrands**, caso exista, atualiza o número de vendas e o valor total de vendas daquela marca. Caso não exista uma entrada para aquela marca, adiciona uma nova entrada na base de dados **soldBrands** com as informações de nome da marca, número de vendas e valor total de vendas.

Ao final, a função retorna a base de dados **soldBrands** com todas as informações processadas.

- createSoldCarsDB

```
export const createSoldCarsDB = (finalDB: FinalDB): SoldCar[] => {
    const soldCars: SoldCar[] = [];
    for (const car of finalDB) {
        let exists = false;
        if (soldCars.length) {
            for (const i of soldCars) {
                if (i.nome === car.nome) {
                    i.vendas += car.vendas;
                    i.total += car.valor * car.vendas;
                    exists = true;
                }
            }
        }
        if (!exists) {
            soldCars.push({
                marca: car.marca,
                nome: car.nome,
                vendas: car.vendas,
                total: car.valor * car.vendas
            });
        }
    }

    return soldCars;
}
```

Aqui temos uma função que recebe uma matriz de objetos **FinalDB** e retorna uma matriz de objetos **SoldCar**. A função é responsável por criar um novo banco de dados de carros vendidos a partir do banco de dados final.

A função começa criando uma nova matriz vazia de **SoldCar**, onde os dados dos carros vendidos serão armazenados. Em seguida, um loop for é iniciado em cada carro presente no banco de dados final.

O loop for então verifica se o carro atual já existe no banco de dados de carros vendidos (**soldCars**) ou não. Se o carro já existe, então as vendas e o total de vendas são adicionados ao objeto correspondente no banco de dados de carros vendidos. Se o carro não existe, ele é adicionado ao banco de dados de

carros vendidos como um novo objeto **SoldCar** com as informações da marca, nome, vendas e total de vendas.

Finalmente, a função retorna o banco de dados de carros vendidos atualizado.

- jsonToCsv

```
import json2csv from "json2csv";

export const jsonToCsv = (json: any): string => {
    return json2csv.parse(json)
}
```

A função **jsonToCsv**, em resumo, converte arquivos JSON para CSV. Para implementar essa função, é necessário instalar e importar a biblioteca **json2csv**, que de acordo com sua própria documentação é um '**Fast and highly configurable JSON to CSV' (JSON para CSV rápido e altamente configurável)**.

O link para a documentação é: **https://juanjodiaz.github.io/json2csv/#/.**

A função recebe apenas um parâmetro, que é o JSON que se deseja formatar. Depois de receber o JSON, a função utiliza o método **'parse'** da biblioteca **json2csv** para converter o JSON para CSV e, em seguida, retorna o CSV já formatado.

Explicando como tudo é executado

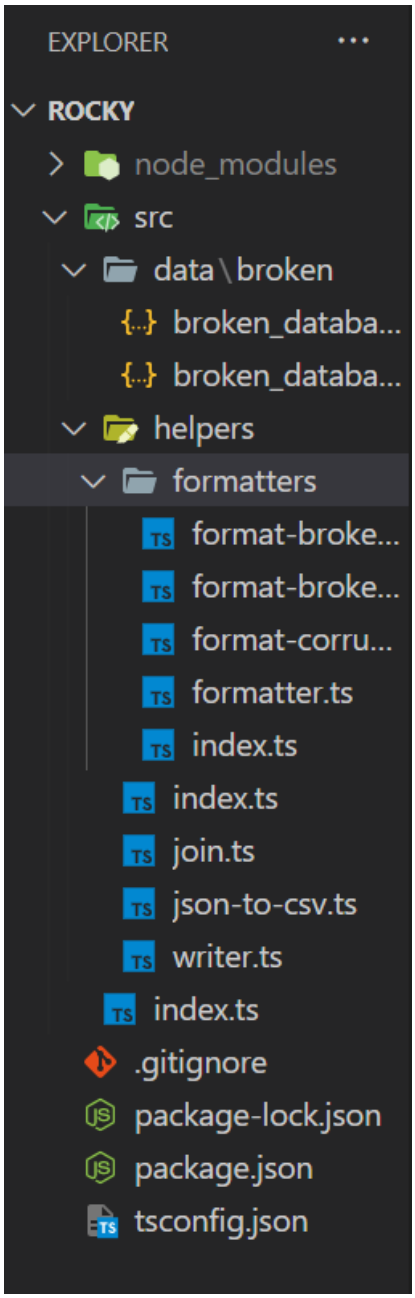
Antes de explicar como tudo é executado, gostaria de mostrar como montei a estrutura de pastas. Vou incluir uma imagem que demonstra como o projeto está organizado, mas não explicarei cada arquivo ou pasta. Vou focar apenas na pasta **"src"**, que é a principal para o nosso projeto.

A pasta **"src"** contém 2 **subpastas** e um **arquivo**:

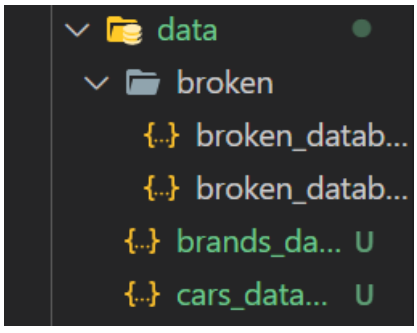
"data" - subpasta

"helpers" - subpasta

"index.ts" - arquivo principal do projeto



- data folder

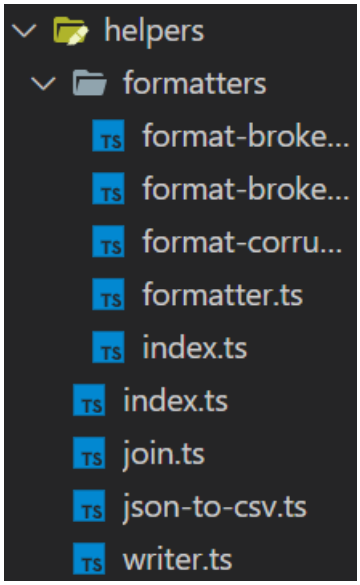


Dentro da pasta **"data**", temos outra pasta chamada **"broken"**.

Nesta pasta estão localizadas as bases de dados corrompidas que foram disponibilizadas para correção.

Também é aqui, na pasta **"data"**, que os dois arquivos JSON das bases de dados já corrigidas são armazenados, porém, obviamente, fora da pasta **"broken"**.

- helpers folder



A pasta **"helpers"** foi criada para armazenar todos os helpers do projeto. Na minha visão, todas as funções do sistema são helpers e, por isso, todas elas estão dentro da pasta **"helpers"**.

Além disso, criei uma subpasta chamada **"formatters"** dentro de **"helpers"**, onde se encontram todas as funções responsáveis por corrigir as databases corrompidas.

Em resumo, a pasta **"helpers"** armazena as funções que executam as tarefas essenciais do sistema.

- index.ts

```
import BrokenDB1 from './data/broken/broken_database_1.json';
import BrokenDB2 from './data/broken/broken_database_2.json';
import { jsonToCsv, formatter, join } from './helpers';
import os from 'os';
import { convertCsvToXlsx } from '@aternus/csv-to-xlsx';
import fs from 'fs';
import { createSoldCarsDB } from './helpers/create-sold-cars-database';
import { createSoldBrandsDB } from './helpers/create-sold-brands-database';
import path from 'path';

// Fix corrupted data.
const { carsDB, brandsDB } = formatter(BrokenDB1, BrokenDB2);

// Unify "carsDB" and "brandsDB" { data, marca, nome, vendas, valor }
const finalDB = join(carsDB, brandsDB);

// Create "soldCars" and "soldBrands" - JSON.
const soldCars = createSoldCarsDB(finalDB);
const soldBrands = createSoldBrandsDB(finalDB);

// Write "carsDb" and "brandsDB" inside "./data".
fs.writeFileSync(path.join(__dirname, 'data/cars_database.json'), JSON.
stringify(carsDB));
fs.writeFileSync(path.join(__dirname, 'data/brands_database.json'), JSON.
stringify(brandsDB));

// Write "finalDB" as CSV in user´s Desktop
fs.writeFileSync(path.join(os.homedir(),'Desktop/database.csv'), jsonToCsv(
finalDB));

//Create "temp" folder.
fs.mkdirSync(path.join(__dirname, 'temp'));

// Write "soldCars" and "soldBrands" as CSV in "./temp".
fs.writeFileSync(`${__dirname}/temp/sold_cars.csv`, jsonToCsv(soldCars));
fs.writeFileSync(`${__dirname}/temp/sold_brands.csv`, jsonToCsv(soldBrands));
fs.writeFileSync(path.join(__dirname, 'temp/sold_cars.csv'), jsonToCsv(
soldCars));
fs.writeFileSync(path.join(__dirname, 'temp/sold_brands.csv'), jsonToCsv(
soldBrands));

// Verify if "carros-vendas".xlsx and marcas-vendas.xlsx already exists.
// If they do not exist, write "soldCars" and "soldBrands" as XLSX files to t
he user's desktop.
if (
    !fs.existsSync(path.join(os.homedir(), 'desktop/carros-vendas.xlsx'))
    || !fs.existsSync(path.join(os.homedir(), 'desktop/marcas-vendas.xlsx'))
) {
    convertCsvToXlsx(path.join(__dirname, 'temp/sold_cars.csv'), path.join(os
.homedir(), 'desktop/carros-vendas.xlsx'));
    convertCsvToXlsx(path.join(__dirname, 'temp/sold_brands.csv'), path.join(
os.homedir(), 'desktop/marcas-vendas.xlsx'));
}

// delete "temp" folder.
fs.rmSync(path.join(__dirname, 'temp/sold_cars.csv'));
fs.rmSync(path.join(__dirname, 'temp/sold_brands.csv'));
fs.rmdirSync(path.join(__dirname, 'temp'));
```

O arquivo **“index.ts”** começa importando dois arquivos JSON que estão armazenados na pasta **./data/broken**. Esses arquivos representam os dois bancos de dados corrompidos, que precisam ser formatados antes de serem usados. O código utiliza a função **formatter** da importação **./helpers** para formatar esses bancos de dados, criando assim dois novos objetos **carsDB** e **brandsDB**.

Em seguida, o código utiliza a função **join** da importação **./helpers** para unir os dois bancos de dados em um único objeto **finalDB**.

Depois disso, utiliza-se as funções **createSoldCarsDB** e **createSoldBrandsDB** da importação **./helpers** para criar dois novos objetos **soldCars** e **soldBrands**, que representam as vendas realizadas de carros e marcas, respectivamente, com base no banco de dados unificado.

O código então escreve os objetos **carsDB** e **brandsDB** em arquivos JSON na pasta **./data**.

Em seguida, utiliza-se a função **jsonToCsv** da importação **./helpers** para converter o objeto **finalDB** em um arquivo CSV e escreve esse arquivo na área de trabalho do usuário.

O código cria uma pasta **temp** e escreve os objetos **soldCars** e **soldBrands** em arquivos CSV nessa pasta.

Em seguida, verifica-se se os arquivos **carros-vendas.xlsx** e **marcas-vendas.xlsx** existem na área de trabalho do usuário. Se não existirem, o código utiliza a função **convertCsvToXlsx** da importação **@aternus/csv-to-xlsx** para criar esses arquivos a partir dos arquivos CSV na pasta **temp**.

Finalmente, o código exclui os arquivos CSV na pasta temp, exclui a pasta temp e termina sua execução.

- scripts

```
"scripts": {
  "start": "node dist/index",
  "build": "tsc"
},
```

Estes são os scripts que implementei no projeto e estão presentes no arquivo **package.json**.

O primeiro script é chamado de **start** e utiliza o comando node **dist/index** para iniciar a aplicação após ela ter sido compilada. A expressão **node dist/index**

indica que o Node deve executar o arquivo **index.js** alocado na pasta **dist**. Geralmente, a pasta **dist** contém o código compilado da aplicação, que foi gerado a partir de um código-fonte escrito em **TypeScript**.

O segundo script é chamado de **build** e utiliza o comando **tsc** para compilar o código TypeScript da aplicação. O comando **tsc** é responsável por converter o código **TypeScript** em **JavaScript** que possa ser executado pelo Node. Ao executar o script **build**, o compilador **TypeScript** (tsc) percorre o código-fonte do projeto e gera um novo código **JavaScript** na pasta **dist**, que pode ser executado pelo script **start**.

Em resumo, o script **build** é responsável por compilar o código **TypeScript** da aplicação, enquanto o script **start** é responsável por iniciar a aplicação compilada.

Finalizando

Com grande satisfação, apresento o sistema que desenvolvi para concorrer à vaga de estágio na **Rocky**. O processo de criação foi desafiador, mas com dedicação e esforço, consegui concluir todas as etapas do projeto.

Durante o desenvolvimento do sistema, pude aprimorar minhas habilidades técnicas, especialmente em programação e análise de dados.

Estou animado para continuar aprendendo e trabalhar com a equipe da empresa, caso seja selecionado para a vaga de estágio. Mais uma vez, agradeço pela oportunidade de participar deste desafio e espero ter demonstrado meu potencial como profissional.

Linkedin: *https://www.linkedin.com/in/jo%C3%A3o-zanardo-14abb9203/*

GitHub do projeto: *https://github.com/JoaoZanardo/rocky*