Documentação do projeto CRUD Rentcars: Programa de estágio 2023

Sumário

Candidato: Ricardo Farias

1 - Execução e ambiente do projeto

Para não existir problemas com versões de ambiente, o projeto será executado via docker. Na raiz do projeto há um arquivo chamado **docker-compose.yaml** responsável por subir a aplicação. Segue os detalhes do docker-compose abaixo:

docker-compose.yaml

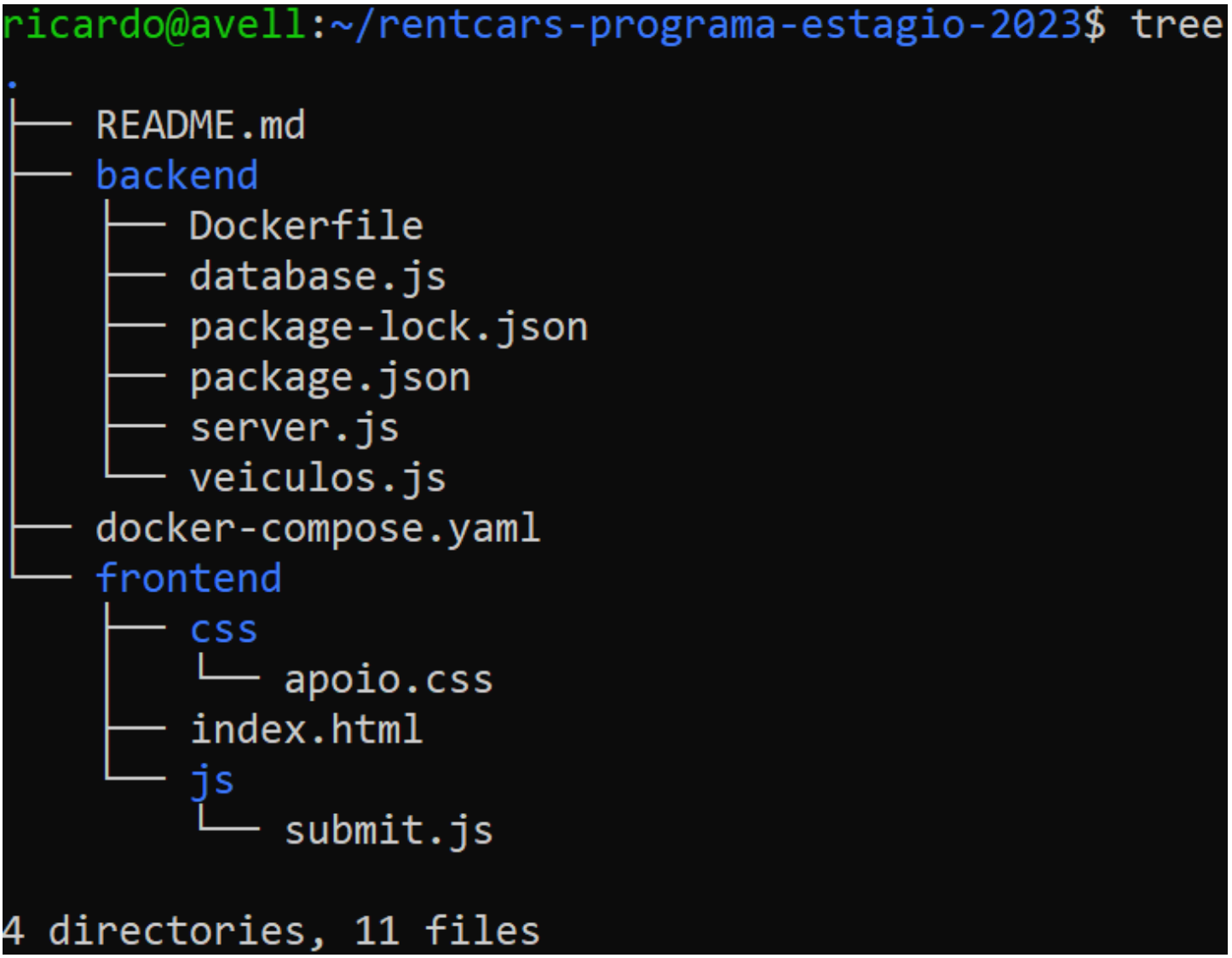
```yaml
version: '3'
services:
  backend:
    build: ./backend
    ports:
      - "3000:3000"
    networks:
      rent-network:
        ipv4_address: 172.18.0.2
    depends_on:
      mysql:
        condition: service_healthy

  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: ANSKk08aPEDbFjDO
      MYSQL_DATABASE: testing
    ports:
      - "3307:3306"
    networks:
      rent-network:
        ipv4_address: 172.18.0.3
    healthcheck:
      test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost"]
      timeout: 20s
      retries: 10

  frontend:
    image: nginx
    volumes:
      - ./frontend:/usr/share/nginx/html
    ports:
      - "8080:80"
    networks:
      rent-network:
        ipv4_address: 172.18.0.4

networks:
  rent-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/16
```

Nele há essencialmente **três serviços**, o do **backend**, do **banco de dados mysql** e do **frontend**. Para facilitar a comunicação entre os containers e fixar o IP para facilitar a conexão do banco de dados MySQL com o backend, foi criada uma rede para os containers (última declaração no arquivo). Também é possível observar que o backend só executa quando o banco de dados MySQL já está no ar, para que não ocorra falhas e impeça que o serviço do backend suba.

Então, para executar, basta ter o docker e o docker-compose instalado, que pode ser obtido por meio do link: https://docs.docker.com/engine/install/ubuntu/ (Para ubuntu nesse caso, mas há para outras distribuições). Com o docker compose instalado, na raiz do projeto, insira no terminal: **docker compose up**

A aplicação irá subir, e assim que concluído, o frontend estará disponível no **localhost:8080**, o backend no **localhost:3000** o banco de dados no **localhost:3307**

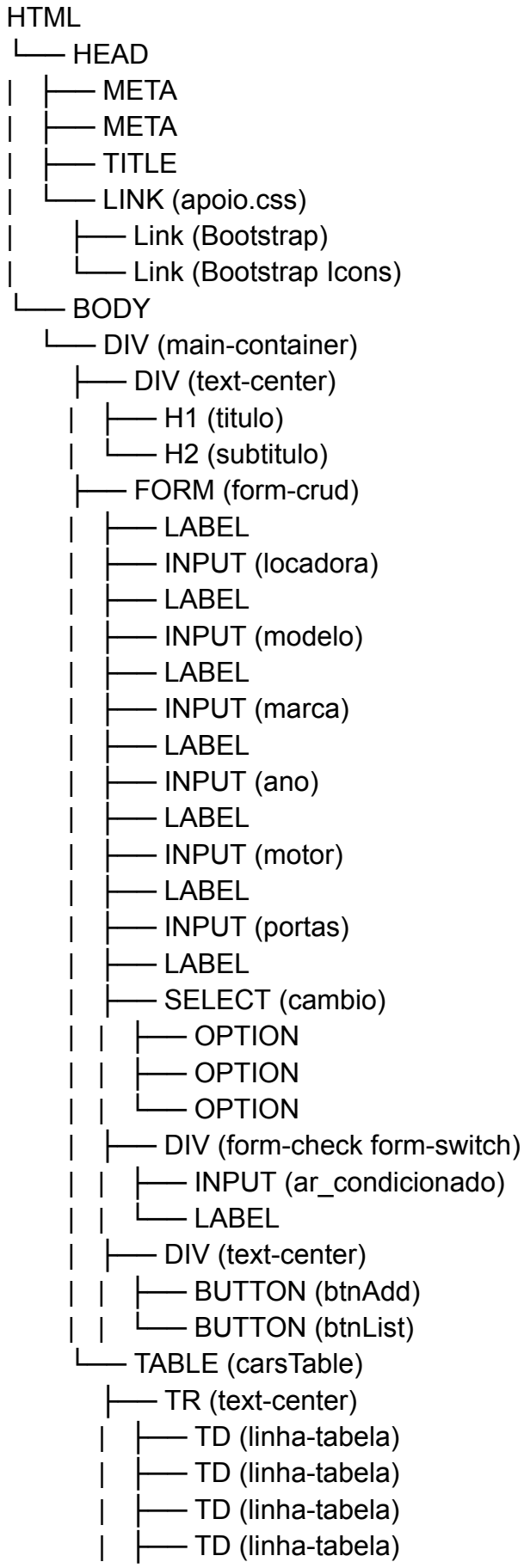
2 - Estrutura do projeto e hierarquia de diretórios



O projeto foi dividido em duas pastas, **backend e frontend.** No frontend ainda há mais duas pastas para separar scripts e estilos, que são, respectivamente, as pastas js e css. Já no backend, além dos arquivos de funcionamento da aplicação, há também um Dockerfile, que tem a intenção de prover um ambiente comum para a aplicação do backend executar.

3 - Detalhes de implementação do projeto

3.1 - **Front-end:** O front-end foi construído com HTML, CSS e Javascript. O **index.html**, que é a página inicial do usuário, tem uma estrutura bastante simples, que pode ser facilmente entendida por meio da árvore DOM abaixo:



```
    |   ├── TD (linha-tabela)
    |   ├── TD (linha-tabela)
    |   ├── TD (linha-tabela)
    |   ├── TD (linha-tabela)
    |   ├── TD (linha-tabela)
    |   └── TD (linha-tabela)
    └── SCRIPT (submit.js)
```

Como é possível observar, ela é constituída basicamente de um forms e uma tabela. Com relação ao css, é feito o uso do bootstrap para estilização simples do formulário e botões e ainda há um arquivo chamado **apoio.css** para alguns ajustes pontuais no estilo da página.

3.1.1 - **Script do frontend (submit.js):** O script é simples e possui duas funções **fillTable()** e **addEvents()**.

3.1.2 - **Função addEvents():** A função addEvents é responsável por adicionar diversos eventos aos elementos HTML da página, incluindo o formulário de envio, botão de listagem, botões de exclusão e botões de atualização. Vou descrever o que cada parte dessa função faz:

Obtenção dos Elementos HTML (Linhas [90, 93]): Obtém os elementos HTML usando document.getElementById. Esses elementos incluem o botão "Listar" (btnList), o formulário (form-crud), a tabela (carsTable), e uma variável endpoint, que já armazena o endereço para onde vamos realizar as requisições, sendo esta última, apenas uma medida para melhorar a legibilidade.

Evento de Envio do Formulário (Linhas [95, 125]): Adiciona um evento de escuta ao formulário para a ação de envio (submit). Quando o formulário é enviado, a função é chamada. Usa evt.preventDefault() para evitar o comportamento padrão de envio do formulário. Obtém os dados do formulário usando new FormData(form). Obtém o valor do cambio separadamente já que ele não faz parte do formulário. Converte o campo "ar_condicionado" em um valor booleano para compatibilidade com o backend. Envia os dados do formulário para o servidor usando a função fetch com o método POST. Em caso de sucesso, chama a função fillTable para atualizar a tabela com os dados recebidos.

Evento de Listagem (Linhas [127, 139]): Adiciona um evento de escuta ao botão "Listar" (btnList). Quando o botão é clicado, a função é chamada. Realiza uma requisição fetch para o servidor (provavelmente para listar os dados) e chama a função fillTable para preencher a tabela com os dados recebidos.

Evento de Exclusão (Linhas [141, 161]): Adiciona um evento de escuta à tabela (table). Quando ocorre um clique na tabela, a função é chamada. Verifica se o ID do elemento clicado inclui "btnDelete," o que indica que um botão de exclusão foi clicado. Se for o caso, extrai o ID do elemento clicado e realiza uma requisição fetch para excluir o registro correspondente no servidor. Após a exclusão, chama fillTable para atualizar a tabela com os dados restantes.

Evento de Atualização (Linhas [163, 266]): Verifica se o ID do elemento clicado inclui "btnUpdate," o que indica que um botão de atualização foi clicado. Se for o caso, obtém os

dados da linha correspondente à qual o botão de atualização pertence. Preenche os campos do formulário com esses dados para permitir a edição. Remove os botões de "Adicionar" e "Listar" e adiciona um botão de "Atualizar". Quando o botão "Atualizar" é clicado, os dados do formulário são enviados ao servidor usando o método PUT para atualização. Após a atualização, a página é recarregada (location.reload(true)) para exibir as alterações na tabela. Essa função é um pouco mais trabalhosa, visto que foi necessário realizar mais manipulação dos elementos da página

3.1.3 - **Função fillTable():** A função fillTable é responsável por preencher uma tabela HTML com dados provenientes de um array de objetos JavaScript. Aqui está uma explicação passo a passo do que essa função faz:

Limpa a tabela atual (Linhas [3, 8]): Obtém a referência à tabela existente com o ID "carsTable". Usa um loop for para iterar pelas linhas da tabela, começando da última linha até a segunda linha (a primeira linha contém os cabeçalhos da tabela). Remove as linhas existentes da tabela usando deleteRow(i). Isso efetivamente limpa a tabela para preenchimento futuro.

Define os dados interessantes (Linhas [10, 20]): Cria um array chamado interestingData que contém os nomes das propriedades de objetos que devem ser exibidas na tabela.

Itera sobre os dados de resultado (Linhas [22, 85]): O código entra em um loop for para iterar pelos objetos no array result, que contém os dados a serem exibidos na tabela.

Cria uma nova linha na tabela (Linhas [24, 25]): Cria um novo elemento <tr> (linha) para a tabela. Define a classe da linha como 'text-center' para alinhar o conteúdo ao centro.

Preenche as células da linha com dados (Linhas [26, 40]): Dentro do loop for, outro loop for...in é usado para iterar pelas propriedades do objeto a ser exibido (como "id", "locadora", "modelo", etc.). Cria uma nova célula (<td>) para cada propriedade. Verifica se o valor da propriedade é true ou false e exibe "Sim" ou "Não" em vez de true ou false, pois é mais cômodo para o caso do ar condicionado. Enfim, define o conteúdo da célula com o valor da propriedade atual.

Cria uma tabela interna nas células da linha e adiciona os botões Atualizar e Excluir (Linhas [57, 84]): Cria uma tabela interna (<table>) dentro de cada célula da linha para organizar os botões de "Atualizar" e "Excluir". Em seguida, cria elementos HTML para os botões de "Atualizar" e "Excluir" dentro da tabela interna. Define as classes e IDs apropriados para os botões. Define os rótulos dos botões e inclui ícones do Bootstrap para melhorar a apresentação.
Anexa os botões de "Atualizar" e "Excluir" à célula interna da tabela.
Anexa a tabela interna à célula da linha:
Anexa a tabela interna, contendo os botões, à célula da linha.
Anexa a linha à tabela principal:
Obtém a referência à tabela principal com o ID "carsTable".
Anexa a linha preenchida com os dados à tabela principal.
No geral, essa função é usada para preencher uma tabela HTML dinamicamente com os dados do array de objetos result. Ela também cria botões "Atualizar" e "Excluir" para cada linha da tabela, tornando-a interativa para o usuário.

3.2 - **Back-end:** Para o backend temos essencialmente 3 arquivos, o **database.js**, responsável por fazer a conexão com o banco de dados utilizando o ORM Sequelize. Temos ainda, **veiculo.js,** para construir o modelo de dados compatível com tabela no banco de dados, e por fim, **server.js,** que contém a implementação do servidor usando o framework express js.

3.2.1 - **database.js:** Essa implementação está configurando uma conexão com um banco de dados MySQL usando a biblioteca Sequelize. Nela, começamos importando o sequelize e em seguida, já criando a conexão nas linhas de 5 até 9. Há um teste de conexão com o banco nas linhas de 11 à 19. Se a conexão for bem sucedida, uma instância do Sequelize é exportada para que possa ser usada em outros módulos da aplicação.

3.2.2 - **veiculos.js:** Representa a definição de um modelo de dados usando a biblioteca Sequelize para interagir com um banco de dados. Ele define como os dados de veículos serão estruturados e armazenados no banco de dados. Em seguida, exporta o modelo para que possa ser usado em outros lugares da aplicação. Quando você cria uma instância do modelo e a sincroniza com o banco de dados, isso resultará na criação da tabela 'veiculos' no banco de dados com a estrutura definida no modelo. Esse modelo pode então ser usado para interagir com os dados de veículos no banco de dados.

3.2.3 - **server.js:** O código é um exemplo de um servidor Node.js que fornece uma API RESTful para gerenciar veículos. O código ocorre da seguinte forma;

Importações e definções (Linhas [10, 13]): As primeiras linhas apenas realizam a importação do express e do modelo Veiculos, bem como a definição da variável app como instância do expres e a porta com o valor 3000.

Configuração do Middleware (Linha 16): Essa linha configura o Express para tratar o corpo das requisições como JSON. Isso é útil para analisar dados JSON enviados nas solicitações POST e PUT.

Configuração do CORS (Linha [18, 34]): O código define as configurações de CORS (Cross-Origin Resource Sharing) para permitir que diferentes origens acessem a API. Ele permite qualquer origem ('Access-Control-Allow-Origin': '*') e especifica quais métodos e cabeçalhos são permitidos. Como eu só fiz a definição do servidor do frontend ao final do desenvolvimento, por ter sido uma forma mais fácil e prática que encontrei de entregar o produto, deixei o Access-Control com ‘*’ para facilitar no desenvolvimento

Rota Raiz (Linhas [36, 38]):
Esta rota raiz simplesmente retorna uma mensagem indicando que a API está em execução.

Endpoints da API (Linhas [40, 128]): Os seguintes endpoints são definidos para gerenciar veículos:

GET /veiculos: Retorna a lista de todos os veículos.
GET /veiculos/:id: Retorna os detalhes de um veículo específico com base no ID.
POST /veiculos: Cria um novo veículo.
PUT /veiculos/:id: Atualiza os detalhes de um veículo existente com base no ID.
DELETE /veiculos/:id: Exclui um veículo com base no ID.

Cada um desses endpoints é tratado com funções assíncronas que usam o modelo Veiculos para realizar operações no banco de dados. Essas operações incluem buscar todos os veículos, buscar um veículo específico, criar um novo veículo, atualizar um veículo existente e excluir um veículo.

Iniciar o Servidor (Linhas [130, 132]): O servidor é iniciado e começa a escutar na porta 3000. Quando o servidor está em execução, ele imprime uma mensagem no console informando a porta em que está ouvindo.

Em resumo, esse código cria um servidor Node.js que fornece uma API para gerenciar veículos. Ele permite que os clientes realizem operações CRUD nos dados de veículos por meio de solicitações HTTP. O modelo Veiculos é usado para interagir com o banco de dados e realizar operações relacionadas aos veículos.

4 - Testes

Infelizmente ainda não há implementação dos testes