Documentação AutoPark

Alex De Abreu
Senac-RN / Zona Norte

Documentação Detalhada do Arquivo package.json

O arquivo package.json serve como o ponto central de configuração para o seu projeto Node.js e Electron. Ele inclui informações sobre o seu projeto, como nome, versão, scripts, dependências, e devDependencies.

1. Campos Básicos

- **name**:"parkingsystem"
  Define o nome do projeto. Esse nome é utilizado em pacotes, logs e outras identificações internas.

- **version**:"1.0.0"
  Indica a versão atual do seu projeto. A versão segue o padrão de versionamento semântico (semver), onde 1.0.0 significa a primeira versão estável.

- **main**: "main.js"
  Define o arquivo de entrada principal da aplicação. Quando o Electron é iniciado, ele utiliza esse arquivo para iniciar o aplicativo.

2. Scripts

- **start**:"concurrently \"nodeserver.js\" \"electron.\""
  Este script usa o pacote concurrently para rodar múltiplos processos ao mesmo tempo. No caso, ele inicia o servidor Node.js (server.js) e, ao mesmo tempo, inicia a aplicação Electron (electron .).

    - "node server.js": Inicia o backend Express.

    - "electron .": Inicia a aplicação Electron usando o arquivo main.js.

3. Dependências

As dependências listadas são necessárias para o funcionamento da aplicação:

- **cors**:"^2.8.5"
  Middleware que habilita o CORS (Cross-Origin Resource Sharing). É utilizado no backend para permitir que a aplicação web carregue recursos de diferentes origens, o que é útil para aplicações que fazem requisições entre servidores diferentes.

- **electron**:"^31.3.1"
  Framework que permite criar aplicações desktop multiplataforma usando tecnologias web. É a base para o desenvolvimento da interface gráfica da aplicação.

- **express**:"^4.19.2"
  Framework minimalista para construir servidores web em Node.js. É utilizado para criar o backend da aplicação, fornecendo APIs RESTful que são consumidas pelo frontend.

- **pg**:"^8.12.0"
  Cliente PostgreSQL para Node.js. Permite que o servidor Node.js interaja com um banco de dados PostgreSQL, realizando consultas, inserções, atualizações, etc.

4. DevDependencies

As devDependencies são pacotes necessários apenas para desenvolvimento e construção do projeto:

- **concurrently**:"^8.2.2"
  Utilitário para executar múltiplos comandos de shell simultaneamente. No contexto do seu projeto, ele é usado para iniciar o servidor Node.js e o Electron ao mesmo tempo.

- **electron-packager**:"^17.1.2"
  Ferramenta para empacotar aplicações Electron em executáveis. Isso permite que você distribua sua aplicação como um arquivo executável independente para diferentes sistemas operacionais.

Documentação Detalhada do Arquivo main.js

O arquivo main.js controla o ciclo de vida da aplicação Electron e define como a interface gráfica será apresentada ao usuário.

1. Importação de Módulos

- **app**: O módulo principal do Electron. Gerencia o ciclo de vida da aplicação.
- **BrowserWindow**: Cria e gerencia janelas da aplicação.
- **Menu**: Permite a criação de menus personalizados.
- **nativeTheme**: Controla o tema da aplicação (claro ou escuro).
- **path**: Módulo nativo do Node.js para manipulação de caminhos de arquivos e diretórios.
- **spawn**: Método do módulo child_process que cria um processo filho para rodar comandos externos, como iniciar o servidor Node.js.

2. Função startServer

Essa função é responsável por iniciar o servidor backend (server.js):

```
function startServer(){
 const server = spawn('node', ['server.js']);
 server.stdout.on('data', (data) => {
  console.log(`Servidor Node.js: ${data}`);
 });
 server.stderr.on('data', (data) => {
  console.error(`Erro no servidor Node.js: ${data}`);
 });
 return server;
}
```

- **spawn('node', ['server.js'])**: Inicia o server.js em um processo separado. Isso é essencial para manter o servidor Node.js rodando junto com o Electron sem bloquear a interface gráfica.
- **Eventos stdout.on('data') e stderr.on('data')**: Capturam e exibem saídas normais e erros do servidor no console, facilitando o debug.

3. Função createWindow

Essa função cria e configura a janela principal da aplicação Electron:

```
function createWindow() {
 nativeTheme.themeSource = 'dark'; // Define o tema da janela
 const mainWindow = new BrowserWindow({
  width: 800,
  height: 600,
  resizable: false,
```

```
  webPreferences: {
   preload: path.join(__dirname, 'preload.js'),
   contextIsolation: true,
   enableRemoteModule: false,
   nodeIntegration: false,
  }
 });

 mainWindow.maximize();
 mainWindow.loadURL('http://localhost:3000/'); // Carrega a interface web hospedada pelo backend
 mainWindow.webContents.on('did-finish-load', () => {
  console.log('Página carregada');
 });
 const template = [
  {
   label: 'MENU',
   submenu: [
    { label: 'Recarregar', accelerator: 'CmdOrCtrl+R', click: () => mainWindow.reload() },
    { label: 'Fechar', accelerator: 'CmdOrCtrl+W', click: () => app.quit() },
    { label: 'Imprimir', accelerator: 'CmdOrCtrl+P', click: () => mainWindow.webContents.print() },
   ]
  }
 ];

 const menu = Menu.buildFromTemplate(template);
 Menu.setApplicationMenu(menu);
}
```

- **nativeTheme.themeSource = 'dark';**: Define o tema escuro para a janela, fornecendo uma experiência visual consistente.
- **Propriedades de BrowserWindow**:
    - **width** e **height**: Define o tamanho da janela (800x600).
    - **resizable: false**: Impede que o usuário redimensione a janela.
    - **webPreferences**: Configurações de segurança, como isolamento de contexto (contextIsolation: true), desativação do módulo remoto e integração com o Node.js.
- **mainWindow.maximize();**: Abre a janela já maximizada.
- **mainWindow.loadURL('http://localhost:3000/');**: Carrega o frontend da aplicação hospedado no servidor backend.
- **Criação do Menu**: Cria um menu customizado com opções de recarregar, fechar e imprimir.
-

4. Gerenciamento do Ciclo de Vida da Aplicação

O ciclo de vida da aplicação é gerido por eventos do módulo app:

- **app.whenReady()**: Executa a função startServer() e createWindow() quando a aplicação está pronta. Além disso, garante que a janela seja reaberta se fechada no macOS.
- **app.on('before-quit', () => { server.kill(); });**: Garante que o processo do servidor seja encerrado corretamente quando a aplicação for fechada.
- **app.on('window-all-closed')**: Fecha a aplicação se todas as janelas forem fechadas, exceto no macOS, onde o padrão é manter a aplicação aberta.

Documentação do Servidor Express para o Sistema de Estacionamento

Server.js

Descrição Geral

Este servidor Express foi desenvolvido para gerenciar um sistema de estacionamento, permitindo a interação com um banco de dados PostgreSQL. Ele fornece funcionalidades para registrar a saída de veículos e remover registros do banco de dados, além de servir o frontend para o usuário.

Estrutura do Código

1. Importação de Módulos

```
const express = require('express');
const cors = require('cors'); // Middleware para permitir requisições cross-origin
const { Pool } = require('pg');
const app = express();
const path = require('path');
```

- **express**: Framework para criação de servidores HTTP.
- **cors**: Middleware para habilitar o Cross-Origin Resource Sharing (CORS).
- **pg**: Cliente para interagir com o banco de dados PostgreSQL.
- **path**: Módulo do Node.js para manipulação de caminhos de diretórios e arquivos.

2. Configuração do Middleware

```
app.use(cors());
app.use(express.json());
```

```
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')));
```

- **app.use(cors())**: Permite que a API receba requisições de diferentes origens.
- **app.use(express.json())**: Habilita o parsing de JSON para requisições POST e PUT.
- **app.use(express.urlencoded({ extended: true }))**: Permite o parsing de formulários HTML.
- **app.use(express.static(path.join(__dirname, 'public')))**: Configura o servidor para servir arquivos estáticos (HTML, CSS, JS).

3. Configuração da Conexão com o PostgreSQL

```
const pool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'autoPark',
  password: 'senha do seu bd',
  port: 5432,
});
```

- **Pool**: Configura o pool de conexões com o banco de dados PostgreSQL.
- **Configurações do pool**:
  - user: Nome de usuário do banco de dados.
  - host: Host do banco de dados.
  - database: Nome do banco de dados.

- password: Senha do banco de dados.
- port: Porta na qual o PostgreSQL está rodando.

4. Tratamento de Erros na Conexão com o PostgreSQL

```
pool.on('error', (err, client) => {
  console.error('Erro na conexão com o PostgreSQL:', err);
  process.exit(-1);
});
```

- **pool.on('error')**: Escuta erros na conexão com o PostgreSQL e encerra a aplicação em caso de falha crítica.

5. Rota Principal

```
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});
```

- **GET /**: Rota principal que serve o arquivo index.html da pasta public.

6. Rota para Registrar a Saída do Veículo

```
app.post('/registrarSaida', async (req, res) => {
  const { placa, veiculo, entrada, saida, valorPago } = req.body;
  try {
    const query = `
      INSERT INTO veiculos (placa, veiculo, entrada, saida, valor_pago) VALUES ($1, $2, $3, $4, $5) `;
```

```
    const values = [placa, veiculo, entrada, saida, valorPago];
    await pool.query(query, values);
    res.status(200).json({ message: 'Dados salvos com sucesso!' });
  } catch (err) {
    console.error('Erro ao salvar no banco de dados:', err);
    res.status(500).json({ error: 'Erro ao salvar no banco de dados' });
  }
});
```

- **POST /registrarSaida**: Recebe os dados do veículo e os insere no banco de dados.
- **Parâmetros**:
  - placa: Placa do veículo.
  - veiculo: Tipo de veículo.
  - entrada: Hora de entrada.
  - saida: Hora de saída.
  - valorPago: Valor pago pelo estacionamento.
- **Resposta**: Retorna uma mensagem de sucesso ou um erro.
-

7. Rota para Deletar Veículo do Banco de Dados

```
app.delete('/removerVeiculo/:placa', async (req, res) => {
  const { placa } = req.params;
  try {
```

```
    const query = `
      DELETE FROM veiculos WHERE placa = $1
    `;
    await pool.query(query, [placa]);
    res.status(200).json({ message: 'Registro deletado com sucesso!' });
  } catch (err) {
    console.error('Erro ao deletar no banco de dados:', err);
    res.status(500).json({ error: 'Erro ao deletar no banco de dados' });
  }
});
```

- **DELETE /removerVeiculo/:placa**: Remove um veículo do banco de dados usando a placa como identificador.
- **Parâmetro de URL**:
    - placa: Placa do veículo a ser removido.
- **Resposta**: Retorna uma mensagem de sucesso ou um erro.

8. Inicialização do Servidor

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
});
```

- **PORT**: Define a porta do servidor, utilizando a variável de ambiente PORT ou a porta 3000 por padrão.
- **app.listen()**: Inicia o servidor e escuta as requisições na porta definida.
-

Documentação do Arquivo preload.js

Descrição Geral

O arquivo preload.js é utilizado em aplicações Electron para expor APIs seguras ao processo de renderização (frontend), permitindo a comunicação entre o frontend e o backend sem comprometer a segurança da aplicação. O uso do contextBridge em conjunto com o ipcRenderer garante que a comunicação seja controlada e limitada às funções explicitamente permitidas.

Estrutura do Código

1. Importação de Módulos

```
const { contextBridge, ipcRenderer } = require('electron');
```

- **contextBridge**: Módulo do Electron que permite expor APIs seguras ao processo de renderização.
- **ipcRenderer**: Módulo do Electron usado para comunicação assíncrona entre o processo de renderização e o processo principal via IPC (Inter-Process Communication).
-

2. Expondo a API Segura

```
contextBridge.exposeInMainWorld('electronAPI', {
  printPage: () => ipcRenderer.send('print-page'),
  reloadPage: () => ipcRenderer.send('reload-page'),
  closeApp: () => ipcRenderer.send('close-app'),
});
```

contextBridge.exposeInMainWorld()

- **Função**: Exponha uma API no window do processo de renderização (renderer process) de forma segura, impedindo o acesso irrestrito ao Node.js no frontend.

- **Argumentos**:
    - **Nome da API**: 'electronAPI'.
    - **Objeto API**: Um objeto que contém as funções seguras que o processo de renderização pode invocar.

Métodos Expostos:

1. **printPage()**
    - **Descrição**: Envia uma solicitação ao processo principal para imprimir a página atual.
    - **IPC Channel**: 'print-page'
2. **reloadPage()**
    - **Descrição**: Envia uma solicitação ao processo principal para recarregar a página atual.
    - **IPC Channel**: 'reload-page'
3. **closeApp()**
    - **Descrição**: Envia uma solicitação ao processo principal para fechar a aplicação.
    - **IPC Channel**: 'close-app'

3. Segurança

O uso do contextBridge em conjunto com o ipcRenderer no Electron é uma prática recomendada para garantir que o processo de renderização (que pode ser exposto a conteúdo não confiável) tenha acesso limitado e controlado às funcionalidades sensíveis da aplicação. Apenas as funções explicitamente expostas estão disponíveis no contexto global (window.electronAPI), mantendo o restante da aplicação isolada e protegida.

Documentação da Funcionalidade de Login.js

Descrição Geral

Este script JavaScript implementa uma funcionalidade básica de autenticação em uma página web. Ele captura as entradas do usuário (e-mail e senha) e as valida contra credenciais pré-definidas. Se a validação for bem-sucedida, o usuário é redirecionado para outra página. Caso contrário, uma mensagem de erro é exibida.

Estrutura do Código

1. Evento DOMContentLoaded

```
document.addEventListener('DOMContentLoaded', function() {
```

- **Função**: Garante que o DOM esteja completamente carregado e pronto antes de qualquer manipulação.
- **Descrição**: A função anônima passada para o DOMContentLoaded é executada assim que o documento é carregado, mas antes que todos os recursos, como imagens, estejam completamente carregados.

2. Adicionar Ouvinte de Evento ao Botão Submit

```
document.getElementById('submit').addEventListener('click', function(event) {
    event.preventDefault();
```

- **Descrição**: Adiciona um ouvinte de evento ao botão com o ID submit, que será acionado quando o botão for clicado.
- **event.preventDefault()**: Impede o comportamento padrão do botão submit, evitando o recarregamento da página.

3. Captura e Validação das Entradas do Usuário

```
const email = document.getElementById('email').value;

const password =
document.getElementById('password').value;
```

- **Descrição**: Captura os valores dos campos de e-mail e senha inseridos pelo usuário.

4. Autenticação Simples

```
if (email === 'admin' && password === 'admin') {

  window.location.href = 'registro.html';

} else {

  alert('Usuário ou senha incorretos');

}
```

- **Descrição**:
  - **Autenticação**: Verifica se o e-mail e a senha correspondem a 'admin'.
  - **Redirecionamento**: Em caso de sucesso, o usuário é redirecionado para a página registro.html.
  - **Mensagem de Erro**: Se as credenciais estiverem incorretas, exibe um alerta com a mensagem 'Usuário ou senha incorretos'.

Considerações de Segurança

Esta implementação de autenticação é simplista e não segura para aplicações em produção. Em um ambiente real, credenciais e lógica de autenticação devem ser gerenciadas de forma segura em um backend, utilizando hashing de senhas, HTTPS, e outros padrões de segurança.

Melhorias Futuras

- Implementação de autenticação via backend para maior segurança.
- Uso de um banco de dados para gerenciar usuários e permissões.
- Implementação de feedback visual ao usuário em vez de usar alert.

Documentação do estacionamento.js

Visão Geral

Este arquivo gerencia a operação de um estacionamento, incluindo a entrada e saída de veículos, cálculo de tarifas, e atualização de vagas disponíveis. Ele faz uso de APIs do navegador para manipulação do DOM, armazenamento de dados no localStorage, e comunicação assíncrona com um servidor backend. O arquivo também calcula o valor devido pelo estacionamento com base no tempo de permanência**.**

Importação de Módulos

JavaScript Nativo: O arquivo não faz uso de módulos externos, mas depende de APIs nativas do JavaScript para manipulação de datas, DOM, e armazenamento local.

Funções

Inicialização e Configurações Iniciais

Define as tarifas e configurações iniciais, carregando-as do localStorage ou definindo valores padrão:

const tarifaMinima = localStorage.getItem('tarifaMinima') || 5;

const tarifaHora = localStorage.getItem('tarifaHora') || 10;

const tarifaDia = localStorage.getItem('tarifaDia') || 50;

calcularValorEstacionamento(entrada, saida, tarifaMinima, tarifaHora, tarifaDia)
Calcula o valor a ser cobrado com base no tempo de permanência do veículo no estacionamento:

function calcularValorEstacionamento(entrada, saida, tarifaMinima, tarifaHora, tarifaDia) {

// Implementa a lógica para calcular o valor total

}

adicionarNaTabela(veiculo)
Insere os dados do veículo na tabela de veículos estacionados exibida na interface do usuário:

function adicionarNaTabela(veiculo) {

// Manipulação da tabela no DOM

}

salvarNoLocalStorage(veiculo)
Armazena os dados de um veículo no localStorage do navegador:

function salvarNoLocalStorage(veiculo) {

// Salvamento de dados no localStorage

}

registrarEntrada()
Captura os dados de entrada do veículo, como placa e horário de entrada, e os armazena no localStorage e na tabela de veículos estacionados:

function registrarEntrada() {

// Coleta e armazena os dados do veículo na entrada

}

registrarSaida(placa)
Registra a saída de um veículo, remove-o da lista de veículos estacionados, calcula o valor devido, e envia os dados para o servidor:

```
async function registrarSaida(placa) {
  // Processamento de saída e comunicação com o backend
}
```

enviarDadosParaBanco(veiculoSaida)
Envia os dados de saída de um veículo para o banco de dados através de uma chamada HTTP POST:

```
async function enviarDadosParaBanco(veiculoSaida) {
  // Comunicação com o servidor usando fetch API
}
```

atualizarTabela()
Atualiza a tabela de veículos estacionados na interface do usuário, removendo veículos que já saíram:

```
function atualizarTabela() {
  // Atualiza a exibição da tabela de veículos estacionados
}
```

resetarDados()
Função que permite a limpeza total dos dados do estacionamento, incluindo a remoção de todos os registros do localStorage e a reinicialização da tabela:

```
function resetarDados() {
  // Limpeza dos dados e reinicialização da tabela
}
```

atualizarTarifas()
Permite a atualização das tarifas de estacionamento através de um painel de administração:

```
function atualizarTarifas() {
  // Atualiza as tarifas armazenadas no localStorage
}
```

Outras Funcionalidades

Controle de Vagas Disponíveis

Monitora em tempo real a quantidade de vagas disponíveis no estacionamento e atualiza a interface do usuário conforme os veículos entram e saem.

Persistência de Dados

Todos os dados de veículos, como placa e horários, são persistentemente armazenados no localStorage, garantindo que as informações não sejam perdidas mesmo após o recarregamento da página.

Comunicação com Backend

O arquivo se comunica com um servidor backend para registrar saídas de veículos, usando a API fetch para enviar dados e receber confirmações.

Validação de Entrada

Verifica se a placa do veículo é válida antes de permitir o registro da entrada ou saída, garantindo que os dados estejam corretos antes de serem processados.

Documentação do relatorio.js

Visão Geral

Este arquivo é responsável por gerenciar o relatório de veículos que utilizaram o estacionamento. Ele carrega os dados armazenados no localStorage e permite a exportação, impressão, e exclusão de registros.

Importação de Módulos

JavaScript Nativo: Utiliza apenas funcionalidades nativas do JavaScript, como manipulação do DOM e armazenamento local.

Funções

carregarRelatorio()

Carrega os dados do relatório do localStorage e os exibe na interface do usuário.

function carregarRelatorio() {

// Carregamento de dados do localStorage e manipulação do DOM

}

exportarParaCSV()

Exporta os dados do relatório para um arquivo CSV.

function exportarParaCSV() {

// Geração e download de arquivo CSV

}

deletarVeiculo(placa)

Remove um veículo do relatório, deletando-o tanto do localStorage quanto do banco de dados.

async function deletarVeiculo(placa) {

// Remoção de dados locais e comunicação com o backend

}

Outras Funcionalidades

Possui funções para navegação entre páginas de administração e de relatório, bem como para a manipulação dos dados de relatório, como a remoção e exportação dos dados.

Documentação admin.js

Descrição Geral

Este script JavaScript permite a configuração e armazenamento das tarifas mínimas, por hora e por dia, e da quantidade total de vagas de estacionamento utilizando o localStorage. O usuário pode atualizar ou resetar as tarifas e a quantidade de vagas através de um formulário na interface.

Estrutura do Código

1. Inicialização e Carregamento do DOM

```
document.addEventListener("DOMContentLoaded", () => {
```

- **Função**: O código é executado quando o DOM está completamente carregado e pronto.
- **Descrição**: Garante que todos os elementos HTML necessários estejam disponíveis antes da execução do restante do script.

2. Carregar Tarifas do localStorage

```
let tarifaMinima =
parseFloat(localStorage.getItem("tarifaMinima")) || 3;

let tarifaHora =

parseFloat(localStorage.getItem("tarifaHora")) || 6;

let tarifaDia =

 parseFloat(localStorage.getItem("tarifaDia")) || 100;
```

- **Descrição**: As tarifas são carregadas do localStorage, ou, se não existirem, valores padrão são definidos.

3. Atualizar a Exibição das Tarifas

```
tarifaMinimaAtual.innerText = `R$: ${tarifaMinima},00`;
tarifaHoraAtual.innerText = `R$: ${tarifaHora},00`;
tarifaDiaAtual.innerText = `R$: ${tarifaDia},00`;
```

- **Descrição**: As tarifas atuais são exibidas na interface do usuário.

4. Manipulação do Formulário de Tarifas

```
tarifaForm.addEventListener("submit", (event) => {
 event.preventDefault();
 tarifaMinima = parseFloat(document.getElementById("tarifaMinima").value);
 tarifaHora = parseFloat(document.getElementById("tarifaHora").value);
 tarifaDia = parseFloat(document.getElementById("tarifaDia").value);

 localStorage.setItem("tarifaMinima", tarifaMinima);
 localStorage.setItem("tarifaHora", tarifaHora);
 localStorage.setItem("tarifaDia", tarifaDia);

 tarifaMinimaAtual.innerText = `R$: ${tarifaMinima},00`;
 tarifaHoraAtual.innerText = `R$: ${tarifaHora},00`;
 tarifaDiaAtual.innerText = `R$: ${tarifaDia},00`;

 tarifaForm.reset();
});
```

- **Descrição**:
    - **Captura**: Obtém as novas tarifas inseridas pelo usuário.
    - **Armazenamento**: Salva as novas tarifas no localStorage.
    - **Atualização**: Atualiza a interface do usuário com os novos valores.
    - **Reset**: Limpa o formulário após a submissão.

5. Resetar Tarifas para Valores Padrão

```
const resetDataButton =
document.getElementById("resetData");

resetDataButton.addEventListener("click", () => {
 localStorage.removeItem("tarifaMinima");
 localStorage.removeItem("tarifaHora");
 localStorage.removeItem("tarifaDia");
 tarifaMinima = 3;
 tarifaHora = 6;
 tarifaDia = 100;

 tarifaMinimaAtual.innerText = `R$: ${tarifaMinima},00`;
 tarifaHoraAtual.innerText = `R$: ${tarifaHora},00`;
 tarifaDiaAtual.innerText = `R$: ${tarifaDia},00`;

 alert("Dados resetados com sucesso.");
});
```

- **Descrição**:
    - **Remoção**: Apaga as tarifas armazenadas no localStorage.
    - **Reset**: Reverte as tarifas para os valores padrão.

- **Confirmação**: Exibe uma mensagem de sucesso ao usuário.

6. Manipulação do Formulário de Vagas

```
const vagasForm = document.getElementById("vagasForm");
vagasForm.addEventListener("submit", (event) => {
 event.preventDefault();
 const totalVagas = parseInt(document.getElementById("totalVagas").value);
 localStorage.setItem("totalVagas", totalVagas);
 alert("Quantidade de vagas salva com sucesso!");
 vagasForm.reset();
});
```

- **Descrição**:
  - **Captura**: Obtém o número total de vagas inserido pelo usuário.
  - **Armazenamento**: Salva o número de vagas no localStorage.
  - **Confirmação**: Exibe uma mensagem de sucesso ao usuário.

Considerações Finais

O uso do localStorage permite que as configurações de tarifas e vagas sejam persistidas mesmo após o fechamento do navegador, facilitando a gestão do estacionamento.

Documentação dos Arquivos HTML do Sistema de Gerenciamento de Estacionamento

1. index.html

- **Descrição**: Página de login inicial do sistema, responsável pela autenticação do usuário. Após um login bem-sucedido, o usuário é redirecionado para a página de registro.

- **Componentes principais**:
  - **Formulário de Login**: Inclui campos para o email e senha.
  - **Validação Simples**: Se o email e a senha corresponderem a 'admin', o usuário é redirecionado para registro.html.

2. registro.html

- **Descrição**: Página principal do sistema onde são registradas as entradas e saídas de veículos.

- **Componentes principais**:
  - **Formulário de Entrada de Veículos**: Permite registrar a placa, veículo e horário de entrada.
  - **Formulário de Saída de Veículos**: Permite registrar a saída do veículo e calcular o valor a ser pago.
  - **Links de Navegação**: Links para acessar o relatório e a página do administrador.
  - **Tabela de Veículos Estacionados**: Mostra os veículos atualmente estacionados e suas informações.

- **Exibição do Valor a Pagar**: Calcula e exibe o valor que o cliente deve pagar baseado no tempo de permanência.

3. admin.html

- **Descrição**: Página de administração para configurar tarifas e gerenciar vagas de estacionamento.

- **Componentes principais**:
  - **Formulário de Configuração de Vagas**: Permite definir o número total de vagas disponíveis.
  - **Formulário de Configuração de Tarifas**: Permite definir as tarifas mínimas, por hora e por dia. As tarifas são salvas no localStorage e exibidas na tabela de tarifas atuais.
  - **Botão de Resetar Dados**: Permite resetar as tarifas para os valores padrão.
  - **Menu de Navegação**: Facilita o acesso à página de relatórios e à página de registro.

4. relatorio.html

- **Descrição**: Página que exibe um relatório com os veículos que passaram pelo estacionamento.

- **Componentes principais**:
  - **Tabela de Relatórios**: Exibe a placa, veículo, horário de entrada, saída e o valor pago por cada veículo registrado.
  - **Botões de Ação**: Possui botões para voltar, acessar a página do administrador, imprimir o relatório e exportá-lo para CSV.

    - **Seção de Deleção de Veículo**: Permite excluir um veículo do relatório usando a placa.

Tecnologias Utilizadas

- **HTML5**: Utilizado para estruturar as páginas web.
- **CSS3**: Utilizado para o design e layout das páginas, utilizando arquivos separados para estilos específicos de cada página.
- **JavaScript**: Usado para a manipulação de eventos, controle de formulário, e interação com o DOM.
- **Electron**: Framework utilizado para construir a aplicação desktop baseada em web.
- **Express.js**: Framework Node.js utilizado para criar o servidor que interage com o banco de dados PostgreSQL.
- **PostgreSQL**: Banco de dados relacional usado para armazenar as informações de entrada e saída dos veículos.
- **localStorage**: Usado para persistir dados de configuração, como tarifas e vagas disponíveis.

Considerações Finais

Este sistema foi desenvolvido para gerenciar o fluxo de veículos em um estacionamento, facilitando o controle de entrada, saída e cálculo de tarifas. As funcionalidades são distribuídas em diferentes páginas, cada uma com seu propósito específico, garantindo uma interface de usuário limpa e organizada.