# *SpotEasy –* Sistema de controle de estacionamento

# Descrição do sistema

O Sistema de Gerenciamento de Estacionamento foi desenvolvido para facilitar a administração de estacionamentos, permitindo o registro de veículos, controle de entradas e saídas, cálculo de tarifas e geração de relatórios. O sistema é implementado em JavaScript e segue uma arquitetura modular para fácil manutenção e escalabilidade.

# Público-alvo

Este sistema foi projetado para ser utilizado por operadores de estacionamento, gerentes e administradores que precisam monitorar e controlar o fluxo de veículos dentro de um estacionamento.

# Objetivo:

O sistema de gerenciamento de estacionamento tem como objetivo registrar a entrada e saída de veículos, controlando o tempo de permanência e calculando a tarifa de acordo com as regras definidas. Além disso, o sistema deve fornecer informações sobre a ocupação das vagas em tempo real e histórico.

# Funcionalidades:

## Entrada de veículos

O sistema deve permitir o registrar a entrada de Veículos:

O sistema deve permitir registrar a placa do veículo;

O sistema deve permitir identificar o tipo de veículo (carro, moto, caminhão, etc.);

O sistema deve permitir registrar a data e hora da entrada;

O sistema deve permitir anotar informações adicionais, se necessário (modelo, cor, etc.);

O sistema deve permitir emitir um tíquete de entrada com código único;

## Saída de Veículos:

O sistema deve permitir registrar a placa do veículo na saída;

O sistema deve permitir validar o tíquete de entrada;

O sistema deve permitir calcular a tarifa de acordo com o tempo de permanência e tipo de veículo;

O sistema deve permitir registrar a data e hora da saída;

O sistema deve permitir, permitir a saída do veículo mediante pagamento da tarifa;

O sistema deve permitir registrar o pagamento (dinheiro, cartão, etc.);

O sistema deve permitir emitir comprovante de pagamento.

## Controle de Vagas:

O sistema deve permitir monitorar a ocupação das vagas em tempo real;

O sistema deve permitir indicar vagas disponíveis e ocupadas;

O sistema deve permitir registrar histórico de ocupação das vagas.

## Relatórios e Estatísticas:

O sistema deve permitir gerar relatórios de entradas e saídas de veículos;

O sistema deve permitir apresentar dados sobre a arrecadação;

O sistema deve permitir indicar o índice de ocupação das vagas;

O sistema deve permitir identificar horários de pico de movimento;

## Regras de Negócio:

### Tarifas:

O sistema deve permitir definir diferentes tarifas por tipo de veículo (carro, moto, caminhão, etc.);

# Estrutura de Diretórios e Arquivos do Sistema "SpotEasy"

PROJETO-SPOTEASY/

├── node\_modules/ # Diretório contendo os pacotes de dependências do Node.js

├── prisma/ # Diretório contendo o esquema do banco de dados e configurações do Prisma

│ └── schema.prisma # Arquivo de esquema do Prisma, definindo a estrutura do banco de dados

├── src/ # Diretório contendo os arquivos principais da aplicação

│ ├── controllers/ # Diretório com os controladores (lógica de negócios)

│ │ └── controller.js # Arquivo que contém a lógica de controle da aplicação

│ ├── public/ # Diretório contendo os arquivos públicos acessíveis ao usuário

│ │ ├── assets/ # Recursos estáticos (imagens, fontes, etc.)

│ │ ├── css/ # Arquivos de estilo CSS

│ │ │ ├── photon.css # Arquivo de estilo Photon

│ │ │ ├── photon.min.css# Arquivo de estilo Photon (versão minimizada)

│ │ │ └── styles.css # Arquivo de estilo geral da aplicação

│ │ ├── fonts/ # Fontes utilizadas na aplicação

│ │ └── js/ # Scripts JavaScript públicos

│ │ ├── exportarCSV.js# Script para exportar dados para CSV

│ │ ├── script-mapa.js# Script para manipulação do mapa

│ │ └── script.js # Script principal da aplicação

│ │ └── index.html # Página principal (HTML) da aplicação

├── services/ # Diretório com os serviços que realizam integrações e lógica externa

│ ├── api.js # Serviço de integração com APIs externas

│ └── index.js # Arquivo principal de configuração dos serviços

├── .env # Arquivo de variáveis de ambiente (como configuração de banco de dados)

├── .gitignore # Arquivo que lista os arquivos/diretórios a serem ignorados pelo Git

├── main.js # Arquivo principal do backend, configurando a execução do servidor

├── package-lock.json # Arquivo que bloqueia as versões das dependências instaladas

├── package.json # Arquivo que define as dependências e scripts do projeto

├── preload.js # Arquivo utilizado para pré-processamento ou configuração inicial

├── server.js # Arquivo principal do servidor, configurando rotas e ouvindo requisições

└── SpotEasy\_Documentação.docx # Documento de documentação do sistema

**Considerações**

* A estrutura do sistema é organizada de forma modular, separando as responsabilidades de acordo com as boas práticas de desenvolvimento.
* A pasta public é responsável por fornecer os arquivos que são acessados diretamente pelo navegador, como HTML, CSS e JavaScript.
* O sistema utiliza o **Prisma** para interação com o banco de dados MongoDB, e todos os detalhes do esquema de banco de dados estão no arquivo schema.prisma.
* O código está preparado para integrar com APIs externas e organizar a lógica de controle através de controllers e serviços.

# Estrutura do Documento HTML

Esta documentação descreve a estrutura HTML do sistema "SpotEasy". A aplicação é estruturada de maneira modular, com componentes como a barra lateral para navegação, cabeçalho para exibição de informações importantes, formulários para check-in de veículos, e uma tabela para visualizar e exportar registros. O sistema é projetado para ser dinâmico e responsivo, com suporte a interações do usuário por meio de JavaScript.

Cabeçalho da Página (Head)  
  
<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>SpotEasy</title>

<!-- Links para folhas de estilo -->

<link rel="stylesheet" href="...">

<link rel="stylesheet" href="...">

<link rel="stylesheet" href="...">

<!-- Scripts JavaScript -->

<script defer src="..."></script>

<script defer src="..."></script>

</head>

* **Metadados**: Define o conjunto de caracteres e configura a página para ser responsiva.
  + **<meta charset="UTF-8">**: Define a codificação de caracteres.
  + **<meta name="viewport" content="width=device-width, initial-scale=1.0">**: Torna a página responsiva.
* **<title>SpotEasy</title>**: Define o título da página que será exibido na aba do navegador.
* **Links de estilo**: Carrega os arquivos CSS necessários para o layout e o design do sistema.
* **Scripts**: Carrega arquivos JavaScript, incluindo bibliotecas como axios, moment.js e scripts específicos da aplicação.

## Corpo da Página (Body)

<body>

<div class="window">

<div class="window-content">

<!-- Conteúdo principal do sistema -->

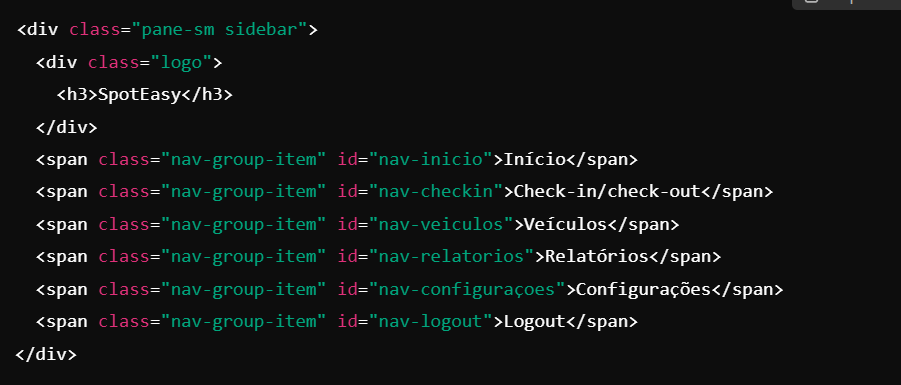
</div>

</div>

</body>

* **<div class="window">**: Contém toda a interface gráfica da aplicação.
* **<div class="window-content">**: Área onde o conteúdo da aplicação será carregado dinamicamente.

## Barra Lateral (Sidebar)



**Barra lateral**: Contém os links de navegação para diferentes seções do sistema.

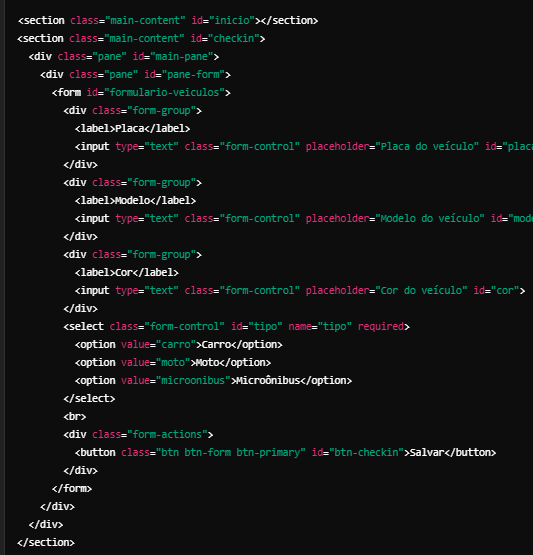
* **<h3>SpotEasy</h3>**: Logo ou título principal da aplicação.
* **Itens de navegação**: Cada item de navegação (<span class="nav-group-item">) leva o usuário a diferentes seções como "Início", "Check-in", "Veículos", "Relatórios", "Configurações" e "Logout".
* **Ícones**: Usados para representar visualmente as seções, como ícones de casa, livro, pasta, gráfico e configurações.

## Cabeçalho (Header)



* **Cabeçalho principal**: Exibe o título "Dashboard" e áreas de informações como a "Taxa de ocupação".
  + **<h1 class="title">Dashboard</h1>**: Título da página exibido no cabeçalho.
  + **Quadros de informação**: Área para exibir dados como a taxa de ocupação e a quantidade de veículos estacionados, com ícones representando carros, motos e micro-ônibus.

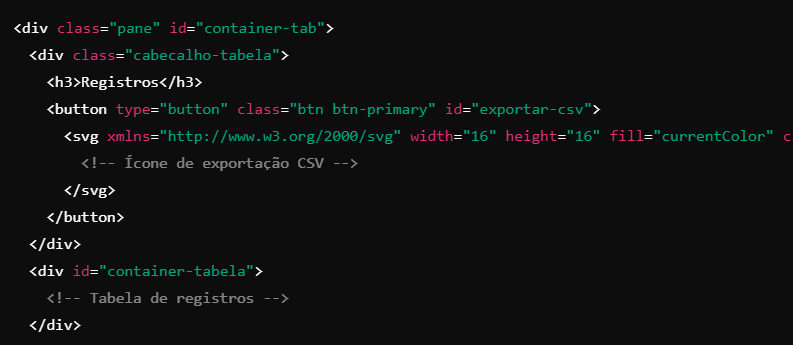
## **Conteúdo Principal**



**Seções de conteúdo**:

* **#inicio**: Seção de início, geralmente usada para carregar informações gerais.
* **#checkin**: Seção que contém o formulário de check-in, onde os usuários podem registrar informações sobre veículos que estão entrando no estacionamento.
* **Formulário de check-in**: Contém campos de entrada para "Placa", "Modelo", "Cor" e "Tipo" do veículo. O botão "Salvar" envia os dados do formulário para o sistema.

### **Tabela de Registros**



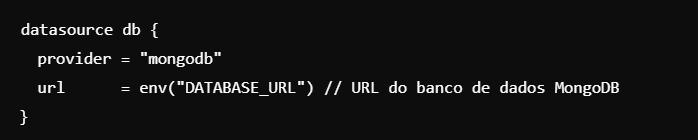
**Exportação de Dados**: Permite exportar os registros para um arquivo CSV.

* **Botão de exportação CSV**: Usa um ícone representando um arquivo CSV. Ao clicar, o sistema exporta os dados da tabela para um arquivo CSV.

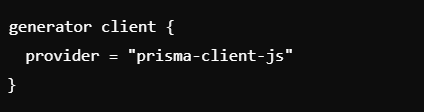
Documentação do Arquivo schema.prisma

O arquivo schema.prisma define o esquema do banco de dados utilizando o Prisma para um sistema de gerenciamento de estacionamento. A base de dados utilizada é o **MongoDB**, e o esquema contém as entidades principais: **Estacionamento**, **Vaga**, **Veículo** e **Check-in**.

**Configuração do Banco de Dados**



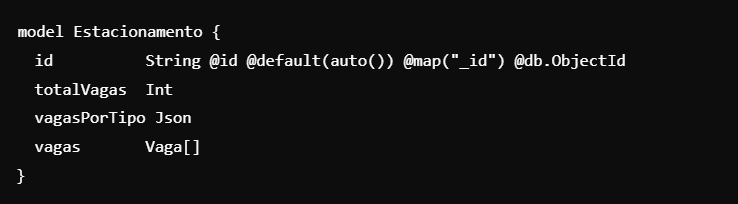
* **provider**: Define o tipo do banco de dados, que é o **MongoDB** neste caso.
* **url**: A URL de conexão com o banco de dados, definida a partir da variável de ambiente DATABASE\_URL.



* **generator client**: Define o gerador do Prisma Client, que será utilizado para interagir com o banco de dados. Neste caso, a configuração indica que o Prisma Client JavaScript será utilizado.

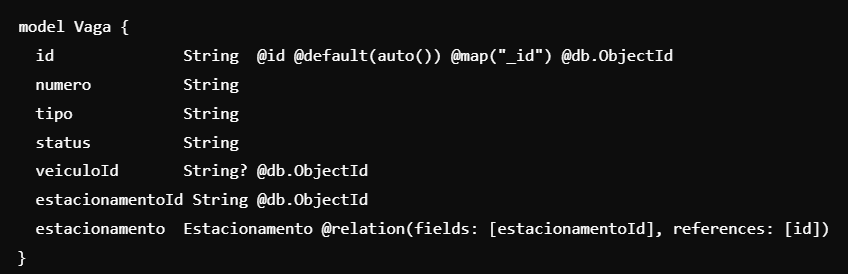
# Modelos de Dados

**1. Modelo Estacionamento**



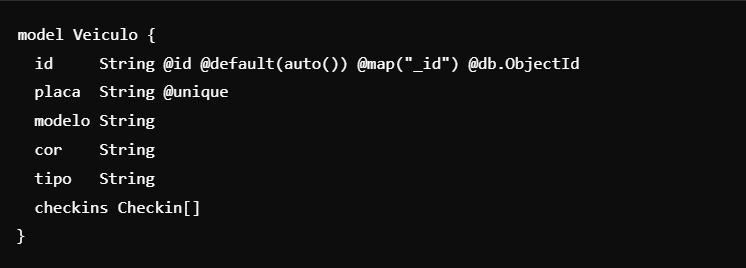
* **id**: Identificador único do estacionamento, com tipo String e mapeado para o campo \_id no MongoDB.
* **totalVagas**: Número total de vagas no estacionamento (tipo Int).
* **vagasPorTipo**: Objeto JSON contendo a quantidade de vagas por tipo de veículo.
* **vagas**: Relacionamento com o modelo Vaga, representando as vagas associadas a este estacionamento.

**2. Modelo Vaga**



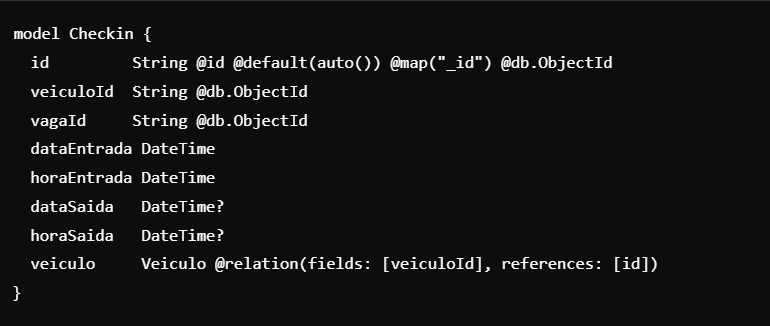
* **id**: Identificador único da vaga, com tipo String e mapeado para o campo \_id no MongoDB.
* **numero**: Número da vaga (tipo String).
* **tipo**: Tipo da vaga, que pode ser para diferentes tipos de veículos (ex. "carro", "moto", etc.).
* **status**: Status da vaga, como "livre" ou "ocupada" (tipo String).
* **veiculoId**: Referência opcional ao id do veículo que está estacionado nesta vaga (tipo String e mapeado para ObjectId).
* **estacionamentoId**: Referência ao estacionamento ao qual a vaga pertence (tipo String e mapeado para ObjectId).
* **estacionamento**: Relacionamento com o modelo Estacionamento, representando o estacionamento associado a esta vaga.

**3. Modelo Veiculo**



* **id**: Identificador único do veículo, com tipo String e mapeado para o campo \_id no MongoDB.
* **placa**: Placa do veículo, que deve ser única (tipo String).
* **modelo**: Modelo do veículo (tipo String).
* **cor**: Cor do veículo (tipo String).
* **tipo**: Tipo do veículo, como "carro", "moto", etc. (tipo String).
* **checkins**: Relacionamento com o modelo Checkin, representando os registros de entrada e saída do veículo no estacionamento.

**4. Modelo Checkin**



* **id**: Identificador único do check-in, com tipo String e mapeado para o campo \_id no MongoDB.
* **veiculoId**: Referência ao id do veículo que realizou o check-in (tipo String e mapeado para ObjectId).
* **vagaId**: Referência ao id da vaga ocupada pelo veículo durante o check-in (tipo String e mapeado para ObjectId).
* **dataEntrada**: Data de entrada do veículo no estacionamento (tipo DateTime).
* **horaEntrada**: Hora de entrada do veículo no estacionamento (tipo DateTime).
* **dataSaida**: Data de saída do veículo do estacionamento (tipo DateTime?, opcional).
* **horaSaida**: Hora de saída do veículo do estacionamento (tipo DateTime?, opcional).
* **veiculo**: Relacionamento com o modelo Veiculo, representando o veículo que realizou o check-in.

**Relacionamentos**

* **Estacionamento ↔ Vaga**: Um estacionamento pode ter muitas vagas (vagas), e cada vaga pertence a um estacionamento específico (estacionamentoId).
* **Vaga ↔ Veiculo**: Uma vaga pode ser ocupada por um veículo, que é referenciado por veiculoId na vaga. No entanto, o campo veiculoId é opcional, pois uma vaga pode estar livre.
* **Veiculo ↔ Checkin**: Um veículo pode ter múltiplos check-ins, representando as entradas e saídas do veículo do estacionamento.

**Considerações**

* O banco de dados utilizado é o MongoDB, o que significa que o sistema se beneficia de um banco de dados NoSQL, facilitando a flexibilidade nos esquemas de dados.
* **Campos Opcionais**: Em alguns modelos, como Checkin, os campos de dataSaida e horaSaida são opcionais, uma vez que a saída do veículo pode ainda não ter ocorrido.
* **Relacional no MongoDB**: Embora o MongoDB seja um banco de dados NoSQL, o Prisma permite estabelecer relacionamentos entre modelos, como mostrado nos modelos Estacionamento, Vaga, Veiculo e Checkin.

# Documentação do arquivo main.js

**Visão Geral**

Este código é um arquivo principal para uma aplicação Electron que inicia um servidor Node.js e cria uma janela do navegador para a interface do usuário. Ele combina funcionalidades do Electron para gerenciamento da interface gráfica com o Node.js para a execução de um servidor backend.

**Dependências**

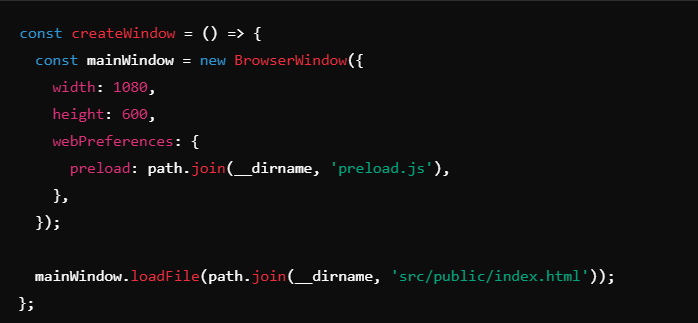
* **Electron**: Para criar a janela da aplicação e gerenciar o ciclo de vida do aplicativo.
* **Path**: Para manipulação de caminhos de arquivos no sistema.
* **URL**: Para obter informações sobre a URL do módulo.
* **Child Process (exec)**: Para executar comandos do sistema, neste caso, iniciar um servidor Node.js.

## Descrição das Funções

**1. createWindow**

A função createWindow é responsável por criar a janela principal da aplicação Electron.

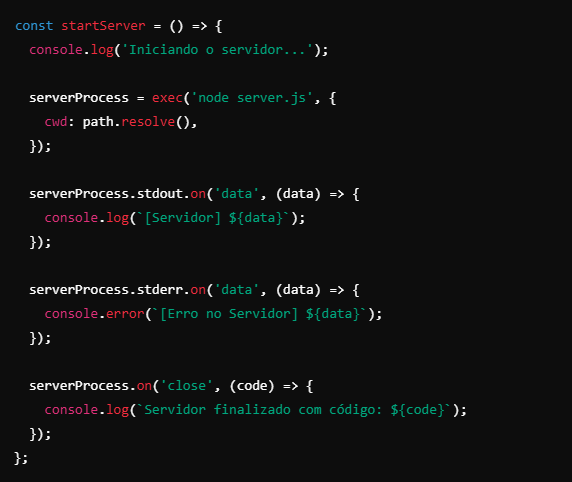
* **Objetivo**: Inicializa a janela do navegador com as configurações definidas, como dimensões e pré-carregamento de scripts.
* **Comportamento**:
  + Cria uma janela de 1080x600 pixels.
  + Carrega o arquivo index.html da aplicação localmente.
  + (Opcional) Abre as ferramentas de desenvolvimento do navegador (comentado).



**2. startServer**

A função startServer é responsável por iniciar um servidor Node.js, especificamente executando o script server.js.

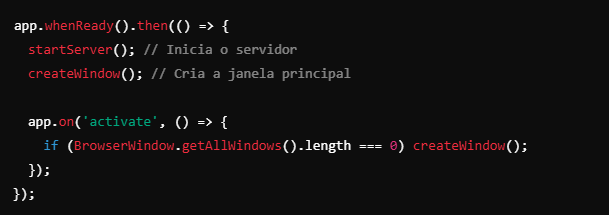
* **Objetivo**: Inicia o servidor e monitora a saída para capturar logs ou erros.
* **Comportamento**:
  + Usa exec para executar o comando node server.js.
  + Define o diretório de trabalho para o servidor como o diretório raiz do projeto.
  + Monitora e imprime no console qualquer saída do servidor.
  + Monitora erros e também imprime mensagens de erro no console.
  + Informa quando o processo do servidor for finalizado.



**3. app.whenReady()**

Quando o Electron finalizar a inicialização e estiver pronto, a função app.whenReady() é executada, disparando a criação da janela e o início do servidor.

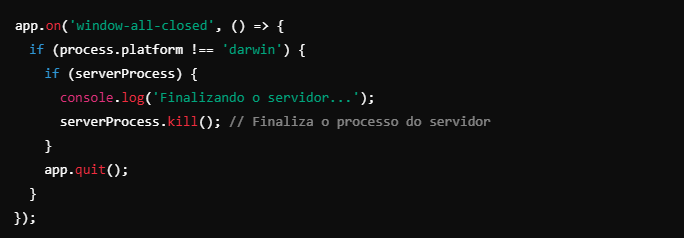
* **Objetivo**: Garante que a janela seja criada e o servidor seja iniciado assim que o Electron estiver pronto.
* **Comportamento**:
  + Chama a função startServer() para iniciar o servidor.
  + Chama a função createWindow() para criar a janela principal.
  + Define um ouvinte para o evento activate que recria a janela no macOS se não houver janelas abertas.



**4. app.on('window-all-closed')**

A função app.on('window-all-closed') é responsável por finalizar o servidor e fechar a aplicação quando todas as janelas forem fechadas, exceto no macOS.

* **Objetivo**: Garante que o servidor seja finalizado quando o aplicativo for fechado.
* **Comportamento**:
  + Verifica se o sistema operacional não é o macOS (process.platform !== 'darwin').
  + Se o servidor estiver em execução, finaliza o processo do servidor.
  + Encerra a aplicação chamando app.quit().



# Documentação do arquivo server.js

**Descrição Geral**

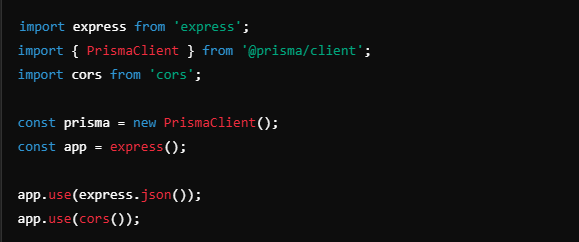
Este arquivo define um servidor Express para gerenciar as operações de estacionamento, incluindo veículos, vagas e check-ins, utilizando o Prisma para interagir com o banco de dados. O servidor oferece rotas para criar, listar, atualizar e deletar veículos, vagas e check-ins, além de manipular as configurações do estacionamento.

**Dependências**

* express: Framework para a criação do servidor web e definição das rotas.
* @prisma/client: Cliente Prisma para interagir com o banco de dados.
* cors: Middleware para habilitar o CORS (Cross-Origin Resource Sharing), permitindo que o servidor aceite requisições de diferentes origens.

**Configuração Inicial**

O servidor utiliza o Prisma Client para interagir com o banco de dados e a biblioteca express para definir as rotas HTTP. As respostas JSON são enviadas para o cliente conforme as operações no banco de dados.



## Endpoints

**1. Veículos**

* **POST /veiculos**: Cria um novo veículo no banco de dados.
  + **Corpo da Requisição**:
    - placa: String
    - modelo: String
    - cor: String
    - tipo: String
  + **Resposta**: Retorna o veículo criado com status 201.
* **GET /veiculos**: Lista todos os veículos registrados.
  + **Resposta**: Retorna a lista de veículos com status 200.
* **PUT /veiculos/:id**: Atualiza as informações de um veículo pelo ID.
  + **Parâmetros**:
    - id: ID do veículo a ser atualizado.
  + **Corpo da Requisição**:
    - placa: String (opcional)
    - modelo: String (opcional)
    - cor: String (opcional)
    - tipo: String (opcional)
  + **Resposta**: Retorna o veículo atualizado com status 200.
* **DELETE /veiculos/:id**: Deleta um veículo pelo ID.
  + **Parâmetros**:
    - id: ID do veículo a ser deletado.
  + **Resposta**: Retorna uma mensagem de sucesso com status 200.

**2. Estacionamento**

* **POST /estacionamento/configuracao**: Cria uma nova configuração de estacionamento.
  + **Corpo da Requisição**:
    - totalVagas: Número total de vagas.
    - vagasPorTipo: Objeto contendo a quantidade de vagas por tipo de veículo.
  + **Resposta**: Retorna a configuração do estacionamento criada com status 201.
* **GET /estacionamento/configuracao**: Lista todas as configurações de estacionamento.
  + **Resposta**: Retorna a lista de configurações com status 200.
* **PUT /estacionamento/configuracao/:id**: Atualiza a configuração de um estacionamento.
  + **Parâmetros**:
    - id: ID da configuração a ser atualizada.
  + **Corpo da Requisição**:
    - totalVagas: Número total de vagas.
    - vagasPorTipo: Objeto contendo a quantidade de vagas por tipo de veículo.
  + **Resposta**: Retorna a configuração atualizada com status 200.
* **DELETE /estacionamento/configuracao/:id**: Deleta uma configuração de estacionamento.
  + **Parâmetros**:
    - id: ID da configuração a ser deletada.
  + **Resposta**: Retorna uma mensagem de sucesso com status 200.
* **GET /vagas/status**: Retorna informações sobre o status das vagas, incluindo o total de vagas, vagas livres, vagas ocupadas e taxa de ocupação.
  + **Resposta**: Retorna os dados com status 200.

**3. Vagas**

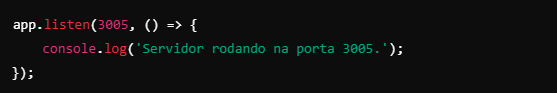
* **POST /criarVagas**: Cria as vagas com base na configuração do estacionamento.
  + **Corpo da Requisição**:
    - estacionamentoId: ID do estacionamento para o qual as vagas devem ser criadas.
  + **Resposta**: Retorna as vagas criadas com status 201.
* **GET /vagas**: Lista todas as vagas disponíveis.
  + **Resposta**: Retorna a lista de vagas com status 200.
* **GET /vagas/status/:status**: Lista as vagas com o status especificado (ex: "livre" ou "ocupada").
  + **Parâmetros**:
    - status: Status da vaga ("livre" ou "ocupada").
  + **Resposta**: Retorna as vagas com o status correspondente com status 200.

**4. Check-in**

* **POST /checkins**: Cria um novo check-in para um veículo, associando-o a uma vaga livre.
  + **Corpo da Requisição**:
    - tipoVeiculo: Tipo do veículo (ex: "carro", "moto").
    - veiculoId: ID do veículo.
    - dataEntrada: Data de entrada no estacionamento.
    - horaEntrada: Hora de entrada no estacionamento.
  + **Resposta**: Retorna o check-in criado com status 201.
* **GET /checkins**: Lista todos os check-ins realizados.
  + **Resposta**: Retorna a lista de check-ins com status 200.
* **PUT /checkins/:id**: Atualiza um check-in (registro de check-out), liberando a vaga.
  + **Parâmetros**:
    - id: ID do check-in a ser atualizado.
  + **Corpo da Requisição**:
    - dataSaida: Data de saída do veículo.
    - horaSaida: Hora de saída do veículo.
  + **Resposta**: Retorna o check-in atualizado com status 200.
* **DELETE /checkins/:id**: Deleta um check-in.
  + **Parâmetros**:
    - id: ID do check-in a ser deletado.
  + **Resposta**: Retorna uma mensagem de sucesso com status 200.

**Inicialização do Servidor**

O servidor é iniciado na porta 3005 com a seguinte linha de código:



# Documentação do Arquivo controllers.js

**Descrição Geral**

O arquivo controllers.js é responsável pela implementação de diversas funcionalidades de interação com a interface do usuário (UI) de um sistema de gerenciamento de estacionamento. Ele se conecta com uma API backend utilizando o Axios para realizar operações de consulta e manipulação de dados sobre veículos, vagas de estacionamento e configurações do estacionamento. O código é executado quando o conteúdo da página é carregado, utilizando eventos DOM, e manipula dados como entrada e saída de veículos (check-in), exibição de informações sobre vagas e veículos, e a exclusão de configurações do estacionamento.

**Funcionalidades**

1. **Carregamento de Configurações do Estacionamento**
   * A função loadConfigurations busca as configurações do estacionamento, como número de vagas totais, vagas livres, vagas ocupadas e taxa de ocupação, através de uma requisição GET para a URL /estacionamento/configuracao.
   * Exibe essas informações na interface, preenchendo as labels correspondentes na página.
   * Caso não haja dados ou ocorra um erro, uma mensagem de alerta é exibida.
2. **Cadastro de Veículo e Check-in**
   * Ao clicar no botão de **Check-in**, o código valida os campos do formulário (placa, modelo, cor, tipo) e verifica se há vagas livres no estacionamento.
   * Caso haja vagas livres, o status da vaga é alterado para "ocupada", e as informações do veículo são cadastradas na API.
   * Uma entrada de check-in também é registrada, vinculando o veículo à vaga, com a data e hora de entrada.
   * Em caso de sucesso, uma mensagem de confirmação é exibida e as configurações do estacionamento são recarregadas.
3. **Exclusão de Configurações**
   * A função vinculada ao botão **Excluir Configurações** permite ao usuário excluir as configurações do estacionamento, fazendo uma requisição DELETE para a URL /estacionamento/configuracao.
   * Após a exclusão, as configurações são recarregadas ou uma mensagem de erro é exibida.
4. **Carregamento de Dados de Vagas**
   * A função carregarDadosVagas solicita informações sobre o status das vagas de estacionamento, como o total de vagas, vagas livres, vagas ocupadas e taxa de ocupação.
   * Esses dados são exibidos na interface, permitindo que o usuário visualize as condições atuais do estacionamento.
5. **Exibição de Veículos**
   * A função carregarVeiculos busca todos os veículos cadastrados na API e exibe suas informações em uma tabela HTML, incluindo ID, placa, modelo, cor, tipo e um botão de ação para deletar o veículo.
   * Os dados dos veículos são carregados automaticamente assim que a página é carregada.
6. **Deletar Veículo**
   * Cada veículo listado na tabela possui um botão de **Deletar**, que ao ser clicado, envia uma requisição DELETE para a API para remover o veículo correspondente.
   * Após a exclusão, a tabela de veículos é recarregada automaticamente para refletir as mudanças.

**Detalhamento das Funções e Eventos**

1. **Função loadConfigurations**
   * Carrega as configurações do estacionamento da API.
   * Atualiza a interface com o número total de vagas, vagas livres, vagas ocupadas e taxa de ocupação.
2. **Evento de Click no Botão btn-checkin**
   * Valida os campos do formulário de cadastro de veículo.
   * Realiza uma requisição para buscar uma vaga livre.
   * Realiza o cadastro do veículo e o check-in com a vaga alocada.
3. **Função carregarDadosVagas**
   * Solicita e exibe os dados de vagas, como o número total de vagas, vagas livres, vagas ocupadas e taxa de ocupação.
4. **Evento de Click no Botão btn-excluirConfiguracoes**
   * Exclui as configurações do estacionamento da API e recarrega os dados na interface.
5. **Função carregarVeiculos**
   * Busca todos os veículos cadastrados na API e exibe-os na tabela.
6. **Função adicionarEventosBotoes**
   * Adiciona eventos de click nos botões de deletar de cada linha de veículo na tabela.

**URLs e Endpoints Utilizados**

* **/veiculos**: Endpoint utilizado para registrar e consultar veículos.
* **/estacionamento/configuracao**: Endpoint para obter e excluir configurações do estacionamento.
* **/vagas**: Endpoint para buscar vagas livres e atualizar o status das vagas.
* **/checkins**: Endpoint para registrar entradas de veículos (check-ins).
* **/vagas/status**: Endpoint para consultar o status das vagas (total, livres, ocupadas, etc.).

**Dependências Externas**

* **Axios**: Biblioteca para realizar requisições HTTP para a API backend.
* **DOM Manipulation**: Utiliza os eventos e manipulação do DOM para interagir com a interface do usuário (ex. addEventListener, getElementById, etc.).

**Como Funciona**

1. Quando a página é carregada, as configurações e dados de vagas e veículos são solicitados automaticamente.
2. O usuário pode realizar o cadastro de novos veículos, registrar a entrada (check-in) e visualizar as informações do estacionamento em tempo real.
3. É possível excluir configurações e deletar veículos da tabela de forma simples e rápida.

**Possíveis Melhorias**

* **Validação de Entrada**: Melhorar a validação de entrada para garantir que os dados inseridos sejam válidos antes de enviá-los para a API.
* **Feedback de Sucesso/Erro**: Fornecer mensagens de sucesso ou erro mais detalhadas para o usuário após ações como cadastro ou exclusão.
* **Gerenciamento de Vagas**: Adicionar funcionalidades para gerenciar o número de vagas, como permitir a alteração do número de vagas ou a adição de novas vagas.

Este arquivo é essencial para o gerenciamento de veículos e vagas em um sistema de estacionamento, fornecendo uma interface interativa para os usuários interagirem com os dados e atualizações em tempo real.

Documentação do arquivo .env  
  
O arquivo .env contém variáveis de ambiente usadas para configurar a aplicação, incluindo a conexão com bancos de dados e outras credenciais sensíveis. No exemplo que você forneceu, a variável DATABASE\_URL está configurando a URL de conexão do banco de dados MongoDB, que será utilizada pelo Prisma para se conectar ao banco de dados SpotEasyDb.

As variáveis de ambiente declaradas nele são automaticamente disponíveis para o Prisma. O Prisma é uma ferramenta de ORM (Object-Relational Mapping) para interagir com o banco de dados.

**DATABASE\_URL**: A variável DATABASE\_URL é usada para armazenar a URL de conexão com o banco de dados MongoDB. Ela contém todas as informações necessárias para estabelecer a conexão com o banco, como:

* **mongodb+srv**: Indica que é uma conexão com MongoDB usando o padrão de conexão SRV.
* **matheusfer200**: Nome do usuário no banco de dados MongoDB.
* **matheusfer200**: Senha do usuário.
* **spoteasy.jsijb.mongodb.net**: O domínio do cluster MongoDB.
* **SpotEasyDb**: O nome do banco de dados.
* **retryWrites=true&w=majority**: Parâmetros adicionais de configuração, indicando que o MongoDB deve tentar reescrever operações e garantir que elas sejam aplicadas à maioria dos nós.
* **appName=SpotEasy**: Nome do aplicativo usado para identificar a conexão com o MongoDB.

# Documentação da Função exportarParaCSV

A função exportarParaCSV() é responsável por exportar os dados contidos em uma tabela HTML (identificada como tabelaVeiculos) para um arquivo CSV, permitindo o download do arquivo gerado. O processo inclui a coleta de dados da tabela, formatação no padrão CSV e criação de um link para o download.

**Fluxo de Execução:**

1. **Seleção da Tabela**:
   * A função começa selecionando a tabela HTML com o ID tabelaVeiculos:
2. **Coleta das Linhas da Tabela**:

* Em seguida, a função seleciona todas as linhas (tr) da tabela:

1. **Construção do Conteúdo CSV**:

**Cabeçalho**: A primeira linha da tabela (presumivelmente, o cabeçalho) é lida. Os títulos das colunas (th) são extraídos e separados por vírgulas, formando a primeira linha do arquivo CSV:

**Dados**: Para cada linha subsequente, os dados (td) são extraídos. Os valores são escapados para garantir que as aspas duplas dentro dos dados não corrompam o formato CSV. Cada valor é separado por vírgulas:

1. **Criação do Arquivo CSV**:

* O conteúdo CSV é armazenado em uma string, com quebras de linha (\n) para separar as linhas de dados. A função então cria um arquivo CSV formatado com base nos dados extraídos:

1. **Criação do Link para Download**:

Um link de download é gerado dinamicamente utilizando a API Blob e URL.createObjectURL(). Este link é configurado para fazer o download do arquivo gerado com o nome dados\_veiculos.csv:

1. **Simulação de Clique**:

* O clique do usuário é simulado chamando link.click(), o que inicia o download do arquivo CSV gerado:

1. **Liberação de Recursos**:

Após o clique, o URL temporário criado para o arquivo é revogado, liberando os recursos utilizados:

javascript

Copiar código

**Interação com o DOM:**

* **Botão de Exportação**: A função é acionada por um evento de clique no botão com o ID exportar-csv. O código adiciona um ouvinte de evento ao botão:

# Documentação do Arquivo scripts.js

O arquivo scripts.js contém funcionalidades relacionadas à navegação entre seções de uma página e à troca de abas dentro de containers. Ele manipula a exibição de seções de conteúdo com base nas interações do usuário, especificamente para um sistema com abas e menus de navegação.

**Estrutura do Código**

**1. Navegação entre seções com base nos itens de menu**

* **Objetivo**: Permitir que o usuário navegue entre diferentes seções da página ao clicar nos itens do menu de navegação.
* **Seleção de Elementos**: O código seleciona todos os itens do menu e as seções de conteúdo, utilizando querySelectorAll:

**Função showSection(sectionId)**: Esta função é responsável por ocultar todas as seções e exibir a seção correspondente ao sectionId passado:

**Adição de Event Listeners nos Itens de Menu**: A função adiciona ouvintes de evento aos itens de menu, de modo que, ao clicar, a seção correspondente seja exibida:

**2. Troca de abas de conteúdo**

* **Objetivo**: Gerenciar a exibição de conteúdo dentro de containers com abas, permitindo alternar entre a exibição de diferentes seções de conteúdo ao clicar nas abas.
* **Seleção de Elementos**: O código seleciona todos os itens de aba e os containers de conteúdo, com a ajuda de querySelectorAll:

**Função toggleTabs(tabIndex)**: Esta função alterna entre a exibição das abas e containers. Quando uma aba é clicada, ela recebe a classe active e o container correspondente é exibido, enquanto os outros containers são ocultados:



**Adição de Event Listeners nas Abas**: A função adiciona ouvintes de evento aos itens de aba para alternar entre os containers de conteúdo: