**Documentação Técnica para Desenvolvedores – SCV v2.1**

**Projeto:** Sistema de Chamada de Veículos (SCV) **Status:** Build de Produção Estável **Data da Revisão:** 13 de Setembro de 2025

**1. Onboarding: Configurando o Ambiente de Desenvolvimento**

Este guia rápido tem o objetivo de fazer um novo desenvolvedor rodar o projeto localmente em menos de 10 minutos.

**Stack Tecnológica:**

* **Backend:** Node.js, Express.js
* **Comunicação Real-time:** Socket.IO
* **Banco de Dados:** MySQL 8+
* **Frontend:** HTML5, CSS3, JavaScript (Vanilla)

**Passos para a Configuração:**

1. **Configure o Banco de Dados:**
   * Certifique-se de ter um servidor MySQL rodando.
   * Execute o script schema.sql para criar a base e a tabela veiculos.
2. **Configure as Variáveis de Ambiente:**
   * Crie uma cópia do arquivo .env.example (se houver) ou crie um novo arquivo chamado .env na raiz do projeto.
   * Preencha-o com suas credenciais do MySQL:

Snippet de código

DB\_HOST=localhost

DB\_USER=root

DB\_PASSWORD=sua\_senha\_aqui

DB\_NAME=logistica\_db

PORT=3000

1. **Instale as Dependências:**

Bash

npm install

1. **Execute a Aplicação:**

Bash

npm start

* + O terminal deve exibir a mensagem Servidor rodando em produção na porta 3000.
  + Acesse o painel do analista em http://localhost:3000/analista.html.

**2. Arquitetura e Análise do Código Fonte**

**2.1. server.js - O Orquestrador**

Este arquivo é o ponto de entrada e o cérebro da aplicação.

* **Inicialização:** Ele carrega o dotenv para ter acesso às variáveis de ambiente, inicializa o Express para servir os arquivos estáticos da pasta /public e o Socket.IO para a comunicação em tempo real.
* **Lógica de Eventos:** A seção io.on('connection', ...) é o núcleo. Cada evento recebido dos clientes (como motorista:checkin ou analista:chamar) dispara uma função assíncrona que:
  1. Valida os dados (se necessário).
  2. Chama a função apropriada do nosso Data Access Layer (DAL), o db/queries.js, para interagir com o banco de dados.
  3. Após a operação no banco, chama broadcastListUpdate() para notificar todos os clientes sobre a mudança de estado.
* **Ponto Chave:** O server.js é "burro" em relação ao SQL. Ele não sabe como inserir ou atualizar dados; ele apenas sabe *quando* pedir para o db/queries.js fazer isso. Essa separação é fundamental para a manutenção.

**2.2. /db/connection.js - O Gerente de Conexões**

* **Propósito:** Este arquivo tem uma única responsabilidade: criar e exportar um "pool" de conexões com o MySQL.
* **Por que um "Pool"?** Em um ambiente de produção, abrir e fechar uma conexão com o banco para cada consulta é extremamente ineficiente. Um pool mantém um número de conexões prontas para serem reutilizadas, melhorando drasticamente a performance. Ele é configurado com os dados do arquivo .env.

**2.3. /db/queries.js - A Camada de Acesso a Dados (DAL)**

Este é o único arquivo que contém linguagem SQL. Ele abstrai a complexidade do banco em funções JavaScript simples e reutilizáveis.

* **addOrUpdateVehicle():** A query INSERT ... ON DUPLICATE KEY UPDATE é a escolha estratégica aqui. Ela resolve de forma atômica o problema de um motorista tentar fazer check-in mais de uma vez. Em vez de gerar um erro de "chave duplicada", ela atualiza os dados do motorista e reseta seu status e horários, tratando o novo check-in como uma reentrada na fila.
* **updateVehicleCall(placa, status, horarioChamada, tempoEsperaMinutos):** Esta é a função mais crítica para a lógica de negócio. Ela executa uma query UPDATE que modifica três colunas essenciais de uma só vez. Isso garante a atomicidade da operação: ou todos os dados da chamada são registrados com sucesso, ou nada é alterado, evitando inconsistências no banco.

**2.4. public/analista.html - A Interface Reativa**

* **Renderização Dinâmica:** A função renderizarTabela() é o motor da interface. Ela é projetada para ser chamada repetidamente sem causar problemas de performance.
* **Atualização em Tempo Real:** A combinação de socket.on('lista:atualizada', ...) com setInterval(renderizarTabela, 1000) é o que cria a experiência em tempo real.
  1. O socket.on atua como um gatilho: ele só dispara quando o servidor informa que há novos dados (um novo check-in, uma chamada, etc.).
  2. O setInterval então assume, atualizando a contagem de segundos a cada 1000ms. Isso é feito puramente no cliente, sem gerar tráfego de rede, sendo extremamente eficiente.
* **Lógica de KPI:** A verificação if (v.horario\_chamada) dentro do loop de renderização é o que implementa a regra de negócio de "congelar" o tempo de espera. Ela separa claramente os veículos em dois estados: aqueles cujo tempo de espera é dinâmico e aqueles cujo tempo já foi consolidado como uma métrica final.

**2.5. public/motorista.html - A Interface de Ação**

* **Validação e Tratamento de Dados:** O uso de toUpperCase() no momento da submissão do formulário é uma camada de "limpeza" de dados no cliente (client-side sanitization). Garante que todas as placas e nomes sejam padronizados antes mesmo de chegarem ao servidor.
* **Feedback ao Usuário:** A implementação do alarme em loop (notificationSound.loop = true;) e sua interrupção no clique do botão (notificationSound.pause();) é um exemplo de design focado na experiência do usuário, garantindo que a notificação seja impossível de ser ignorada.

**3. Fluxo de Dados Crítico: O Processo de Chamada**

Para solidificar o entendimento, vamos detalhar o fluxo de dados mais importante do sistema:

1. **Ação (Analista):** O analista clica no botão "Chamar" do veículo ABC1234. O evento analista:chamar é emitido para o server.js com { placa: 'ABC1234' }.
2. **Processamento (Backend):** a. server.js recebe o evento. b. Define horarioChamada = new Date(). c. await db.getVehicleByPlaca('ABC1234') -> Banco retorna o objeto do veículo, incluindo sua chegada. d. server.js calcula tempoEsperaMinutos (horarioChamada - chegada). e. await db.updateVehicleCall('ABC1234', 'Chamado', horarioChamada, tempoEsperaMinutos). f. db/queries.js executa o UPDATE no MySQL, salvando o status, o horário da chamada e o tempo de espera calculado na linha da placa ABC1234.
3. **Notificação e Atualização (Broadcast):** a. server.js emite veiculo:chamado para o socket\_id específico do motorista ABC1234. b. server.js emite lista:atualizada para todos os clientes.
4. **Reação (Clientes):** a. O motorista.html do veículo ABC1234 recebe veiculo:chamado e dispara o alarme. b. Todos os analista.html conectados recebem lista:atualizada e renderizam novamente a tabela. Para o veículo ABC1234, a condição if (v.horario\_chamada) agora é verdadeira, e o tempo de espera é exibido como um valor fixo e final.