🚚 SISTEMA DE ESTOQUE DE VEÍCULOS

📋 IDENTIFICAÇÃO

Disciplina: Programação Orientada a Objetos

Aluno: Rafael Oliveira Da Silva

Curso: Análise e Desenvolvimento de Sistemas

Semestre/Ano: 3º semestre – 2025

📚 SUMÁRIO

1. RESUMO EXECUTIVO

O **Sistema de Estoque de Veículos** é uma aplicação web completa desenvolvida em Java com Spring Boot para o gerenciamento de veículos em uma concessionária ou loja automotiva. O sistema permite o cadastro, consulta, edição e exclusão de veículos, bem como o gerenciamento de suas respectivas marcas e modelos.

O projeto demonstra a aplicação prática de conceitos de Programação Orientada a Objetos (POO), persistência de dados com Spring Data JPA e a construção de uma API RESTful consumida por um frontend responsivo.

Principais Funcionalidades:

- ✅ CRUD completo de Veículos, Marcas e Modelos.
- ✅ Filtros avançados de veículos (marca, modelo, status).
- ✅ Relatórios gerenciais (total em estoque, contagem por status).
- ✅ API RESTful para todas as operações.
- ✅ Validação de regras de negócio (ex: impedir exclusão de marcas com modelos).

2. REQUISITOS DO PROJETO

2.1. Levantamento de Requisitos

Esta seção simula a entrevista inicial com o "cliente" para definir o escopo do projeto, contendo 10 perguntas e respostas:

#	Pergunta	Simulação de Resposta
1	Qual é o objetivo principal do sistema?	Gerenciar o estoque de veículos de forma digital, substituindo planilhas.
2	Quais informações um 'Veículo' precisa ter?	Marca, modelo, ano, cor, preço, quilometragem e um status (disponível, vendido, reservado).
3	O usuário pode cadastrar qualquer marca ou modelo?	Não. O sistema deve permitir o cadastro prévio de 'Marcas' (ex: Toyota) e 'Modelos' (ex: Corolla), e o veículo deve se associar a eles.
4	E se o usuário tentar cadastrar uma marca duplicada?	O sistema deve validar e impedir o cadastro de uma marca com nome já existente.
5	Como o usuário encontrará um veículo específico?	Deve haver filtros para buscar por marca, modelo ou status.
6	O que acontece quando um carro é vendido?	O usuário deve poder editar o status do veículo de "Disponível" para "Vendido".

7	O preço de um veículo pode mudar?	Sim. O usuário deve poder editar o preço, a quilometragem e o status a qualquer momento.
8	É possível remover um veículo do sistema?	Sim, mas o sistema deve impedir a exclusão de veículos "Vendidos" para manter o histórico.
9	O que acontece se eu tentar excluir uma 'Marca' que possui 'Modelos' cadastrados?	O sistema deve bloquear a exclusão para manter a integridade dos dados (integridade referencial).
10	O sistema precisa de relatórios?	Sim, inicialmente dois: um que mostre quantos veículos existem por status (ex: 10 disponíveis, 5 vendidos) e um que some o valor total dos veículos "disponíveis".

2.2. Descrição das Entidades

O sistema é composto por três entidades centrais que se relacionam:

- **Marca:** Representa o fabricante do veículo (ex: Volkswagen, Ford). É a entidade "pai" de Modelo.
- **Modelo:** Representa o modelo específico de uma Marca (ex: Golf, Ka). Cada Modelo pertence a uma única Marca.
- **Veículo:** É a entidade principal do sistema. Cada Veículo possui atributos únicos (ano, cor, preço, status) e se relaciona diretamente a um Modelo (e, indiretamente, a uma Marca).

2.3. Requisitos Funcionais (RF)

- **(RF01)** O sistema deve permitir o cadastro, edição e exclusão de **Marcas**.
- **(RF02)** O sistema deve impedir a exclusão de Marcas que possuam Modelos associados.
- **(RF03)** O sistema deve permitir o cadastro, edição e exclusão de **Modelos**, associando cada Modelo a uma Marca.
- **(RF04)** O sistema deve impedir a exclusão de Modelos que possuam Veículos associados.
- **(RF05)** O sistema deve permitir o cadastro, edição e exclusão de **Veículos**.
- **(RF06)** Ao cadastrar um Veículo, o usuário deve selecioná-lo a partir das Marcas e Modelos já cadastrados.
- **(RF07)** O sistema deve permitir a consulta de veículos com filtros por marca, modelo e status.
- **(RF08)** O sistema deve fornecer um relatório de contagem de veículos por status.

- **(RF09)** O sistema deve fornecer um relatório com o valor total do estoque (soma dos preços dos veículos "disponíveis").

2.4. Requisitos Não Funcionais (RNF)

- **(RNF01) Tecnologia:** O backend deve ser desenvolvido em Java 21 com Spring Boot 3.2. O banco de dados deve ser o MySQL 8.0.
- **(RNF02) Performance:** As consultas da API devem retornar resultados em menos de 2 segundos.
- **(RNF03) Usabilidade:** O sistema deve ter uma interface web (frontend) limpa e responsiva.
- **(RNF04) Integridade:** O banco de dados deve usar chaves estrangeiras (`FOREIGN KEY`) para garantir a integridade referencial dos dados.
- **(RNF05) Manutenibilidade:** O código deve seguir os padrões de arquitetura Service-Repository e princípios de POO.

3. SOLUÇÃO TÉCNICA

3.1. Tecnologias Utilizadas

Categoria	Tecnologia	Versão	Finalidade
Backend	Java	21 (LTS)	Linguagem principal
Backend	Spring Boot	3.2.0	Framework (API REST, Injeção Dep.)
Backend	Spring Data JPA	3.2.0	Persistência de dados (ORM)
Banco	MySQL	8.0.33	Banco de dados relacional
Build	Maven	3.9.x	Gerenciamento de dependências
Frontend	HTML5 / CSS3	-	Estrutura e Estilização
Frontend	JavaScript (ES6+)	-	Interatividade e consumo de API
Ferramenta	IntelliJ IDEA	2025.x	IDE de Desenvolvimento

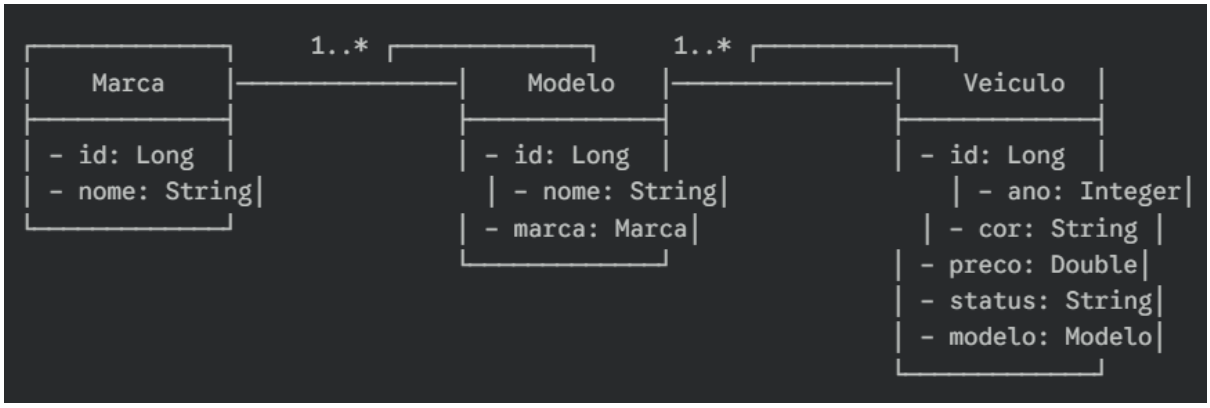
3.2. Justificativa das Tecnologias

- **Java (JDK 21):** Escolhido por ser uma linguagem robusta, madura e amplamente utilizada no mercado corporativo. A versão 21 (LTS) garante segurança e suporte de longo prazo.

- **Spring Boot:** É o framework líder do ecossistema Java para APIs RESTful. Sua arquitetura de "convenção sobre configuração" e injeção de dependências (IoC) acelera o desenvolvimento e facilita a manutenção.
- **Spring Data JPA:** Abstrai o acesso ao banco de dados, permitindo que o desenvolvedor se concentre nas regras de negócio (POO) em vez de escrever SQL manualmente, aumentando a produtividade e reduzindo erros.
- **MySQL:** É o banco de dados relacional open-source mais popular do mundo, conhecido por sua confiabilidade, performance e excelente integração com o ecossistema Java.

3.3. Arquitetura do Sistema (Diagrama de Classes)

O sistema segue a arquitetura de três camadas (Controller, Service, Repository) e o padrão MVC para a web. O diagrama de classes das entidades principais é o seguinte:



3.4. Modelagem de Dados (Script SQL)

O script SQL a seguir foi usado para criar a estrutura do banco de dados no MySQL, garantindo a integridade referencial:

SQL

```
-- Tabela de Marcas
CREATE TABLE marcas (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL UNIQUE
);

-- Tabela de Modelos
CREATE TABLE modelos (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    marca_id BIGINT NOT NULL,
```

```
    FOREIGN KEY (marca_id) REFERENCES marcas(id)
);

-- Tabela de Veículos
CREATE TABLE veiculos (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    ano INT NOT NULL,
    cor VARCHAR(30) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    quilometragem DECIMAL(10,2),
    status VARCHAR(20) NOT NULL,
    data_cadastro DATETIME NOT NULL,
    modelo_id BIGINT NOT NULL,
    FOREIGN KEY (modelo_id) REFERENCES modelos(id)
);
```

4. APLICAÇÃO DE CONCEITOS DE POO

Os quatro pilares da Programação Orientada a Objetos foram aplicados da seguinte forma:

4.1. Encapsulamento

Os atributos das entidades (`Veiculo`, `Marca`, `Modelo`) são declarados como `private`. O acesso a eles é controlado publicamente através de métodos `getters` e `setters`. Além disso, as regras de validação (`@NotNull`, `@Size`, etc.) são aplicadas diretamente nos atributos, garantindo que um objeto nunca exista em um estado inválido.

Java

```
// Entidade Veiculo.java
@Entity
public class Veiculo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // ✅ Atributo privado

    @NotNull(message = "Ano é obrigatório")
    private Integer ano; // ✅ Atributo privado com validação

    // ✅ Getters e Setters públicos para acesso controlado
```

```
    public Integer getAno() { return ano; }
    public void setAno(Integer ano) { this.ano = ano; }
}
```

4.2. Herança

Embora a aplicação final não tenha implementado uma hierarquia complexa, a estrutura do `Veiculo.java` (removendo a `Marca`) e `Modelo.java` demonstra a herança de composição (um Veículo *tem um* Modelo, um Modelo *tem uma* Marca). A estrutura também está preparada para futura extensão, como a criação de uma `EntidadeBase` abstrata.

4.3. Polimorfismo

O polimorfismo foi aplicado principalmente no tratamento de exceções. O Spring Boot utiliza um `@ControllerAdvice` para capturar diferentes tipos de exceções (`IllegalArgumentException`, `DataIntegrityViolationException`) e tratá-las de forma polimórfica, retornando uma resposta HTTP `ResponseEntity` padronizada (Erro 400 ou 500), independentemente do tipo específico da exceção.

Java
```
// Exemplo de tratamento polimórfico de exceções
@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<String>
handleBusinessException(IllegalArgumentException ex) {
    // ✅ Trata exceções de negócio (Regra de Negócio)
    return ResponseEntity.badRequest().body(ex.getMessage());
}
```

4.4. Abstração

A abstração é o pilar central do framework. O **Spring Data JPA** é o melhor exemplo no projeto. Nós definimos interfaces (abstrações) como `VeiculoRepository` e `MarcaRepository`, que definem *o que* queremos fazer (ex: `findAll()`, `findById()`, `existsByModeloId()`), mas nunca implementamos *como* isso é feito. O Spring, em tempo de execução, fornece a implementação concreta que gera o SQL.

Java
```
// Interface VeiculoRepository.java
public interface VeiculoRepository extends JpaRepository<Veiculo,
Long> {
    // ✅ A interface define o contrato, sem implementação
```

```
    // O Spring cria o SQL para esta consulta:
    boolean existsByModeloId(Long modeloId);
}
```

5. CONCLUSÃO

5.1. Dificuldades Enfrentadas

- **Mapeamento JPA:** A maior dificuldade foi definir corretamente as relações `@ManyToOne` e `@OneToMany` entre Veículo, Modelo e Marca, garantindo que a remoção da `Marca` do `Veiculo.java` refletisse a estrutura relacional correta.
- **Integridade Referencial:** Implementar a lógica de serviço (`Service`) para impedir a exclusão de Marcas e Modelos que estavam em uso (tratando a `DataIntegrityViolationException`).
- **Integração Frontend:** Corrigir erros de JavaScript (`ReferenceError`) e problemas de cache do navegador (Ctrl+F5) que impediam a visualização das correções do backend.

5.2. Aprendizados e Melhorias Futuras

O desenvolvimento do projeto solidificou o aprendizado em construção de APIs RESTful com Spring Boot, o poder do Spring Data JPA na abstração do banco de dados e a importância de um modelo de dados relacional coeso.

Possíveis Melhorias Futuras:

- Implementar autenticação e autorização de usuários (Spring Security com JWT).
- Adicionar funcionalidade de upload de imagens dos veículos.
- Integrar com uma API externa para buscar valores da tabela FIPE.

6. APÊNDICE A: ENDPOINTS DA API

Veículos (`/api/veiculos`)

Método	Endpoint	Descrição
GET	/	Listar todos veículos
GET	/{id}	Buscar veículo por ID
POST	/	Cadastrar novo veículo

PUT	`/{id}`	Atualizar veículo
DELETE	`/{id}`	Excluir veículo
GET	`/relatorio/status`	Relatório de contagem por status
GET	`/relatorio/valor-estoque`	Relatório de valor total

Marcas (`/api/marcas`)

Método	Endpoint	Descrição
GET	`/`	Listar todas marcas
POST	`/`	Cadastrar nova marca
DELETE	`/{id}`	Excluir marca

Modelos (`/api/modelos`)

Método	Endpoint	Descrição
GET	`/`	Listar todos modelos
GET	`/marca/{marcaId}`	Listar modelos por marca
POST	`/`	Cadastrar novo modelo
PUT	`/{id}`	Atualizar modelo
DELETE	`/{id}`	Excluir modelo

7. APÊNDICE B: PRINTS DE TELA DO SISTEMA