# FleetRent – Documentação Técnica Integrada

## 1. Resumo executivo

FleetRent é uma aplicação de referência para gestão de frotas de motocicletas construída sobre .NET 8. A solução segue arquitetura em camadas com divisão clara entre domínio, aplicação, infraestrutura e API. Os principais casos de uso cobrem cadastro e gestão de motos, motoristas e locações, além de emissão de notificações via RabbitMQ quando uma moto do ano vigente é registrada.

Este documento apresenta a visão arquitetural, os componentes centrais, fluxos de negócio prioritários, contratos de API, integrações externas, requisitos de execução e pontos de extensão recomendados.

## 2. Arquitetura e tecnologias

A solução está organizada em quatro projetos principais (Domain, Application, Infrastructure e API) vinculados pelo arquivo de solução `FleetRent.sln`. Cada camada possui responsabilidades bem definidas e comunica-se através de interfaces que permitem testes e substituição de implementações.

- Domain: entidades, regras de negócio e catálogo de erros reutilizável.
- Application: serviços orquestradores, DTOs e validações de uso.
- Infrastructure: Entity Framework Core com PostgreSQL, repositórios, mensageria RabbitMQ e armazenamento de arquivos.
- API: host ASP.NET Core com controllers REST, cultura padrão pt-BR, Swagger e logging com Serilog.

Tecnologias-chave: .NET 8, ASP.NET Core, Entity Framework Core 8, PostgreSQL, RabbitMQ, Docker/Docker Compose e Serilog.

## 3. Camada Domain

A camada de domínio modela os conceitos centrais do negócio e mantém a consistência das regras. As entidades principais incluem:

- Bike: agrega identificador, ano, modelo, placa e o histórico de locações, impedindo exclusão quando há vínculos ativos.

- Driver: representa motorista com CNPJ, data de nascimento, número e categoria da CNH, além da imagem armazenada em disco.
- Rental: define locação com plano ( `RentalPlan` ) responsável por cálculo de valores, multas por devolução antecipada e cobrança por atraso.
- Notification: armazena mensagens geradas a partir de eventos de mensageria (como `bike.created` ).

O catálogo `DomainErrors` centraliza mensagens e códigos reutilizados em exceções ( `DomainException` ), promovendo consistência nas respostas e na instrumentação.

# 4. Camada Application

Os serviços de aplicação traduzem os casos de uso expostos pela API em operações sobre o domínio. Destaques:

- `BikeService` : garante unicidade de placa, aplica mudanças por meio do agregado e publica o evento `bike.created` no RabbitMQ.
- `DriverService` : valida duplicidade de CNPJ e CNH, persiste motoristas e delega o upload de imagem à abstração `IFileStorageService` .
- `RentalService` : gera identificadores sequenciais ( `RENT-0001` ), verifica habilitação categoria A, previne locações com data passada e bloqueia conflitos de agenda da moto.

Os DTOs localizados em cada subpasta ( `Bikes/Dtos` , `Drivers/Dtos` , `Rentals/Dtos` ) desacoplam a API das entidades persistidas e encapsulam contratos de entrada/saída.

# 5. Camada Infrastructure

A infraestrutura fornece implementações concretas para as interfaces de repositório, mensageria e armazenamento, além do `FleetRentDbContext` .

- Persistência: `FleetRentDbContext` aplica configurações de entidade, migrations e seeds ( `BikeSeed` , `DriverSeed` ).
- Repositórios: implementações EF Core para `Bike` , `Driver` , `Rental` e `Notification` , com filtros específicos (por placa, CNPJ, disponibilidade de moto etc.).
- Mensageria: `RabbitMqPublisher` publica mensagens; `BikeCreatedConsumer` consome `bike.created` e gera notificações persistidas.
- Armazenamento: `FileStorageService` salva imagens de CNH em disco utilizando regras de permissão e tratamento de erros encapsulados em `DomainErrors.Storage` .

# 6. Camada API

O projeto `FleetRent.API` hospeda controllers RESTful (`BikesController`, `DriversController`, `RentalsController`). O `Program.cs` configura cultura pt-BR, Swagger, Serilog, injeção de dependências e execução automática de migrations ao inicializar.

# 7. Fluxos de negócio

As rotas expostas orquestram casos de uso principais. Cada fluxo é sustentado por validações de domínio e integração com infraestrutura.

## 7.1 Cadastro de motocicletas

- A API recebe `CreateBikeDto` e delega ao `BikeService`.
- Validação de unicidade de placa via `IBikeRepository.GetByPlateAsync`.
- Persistência via `FleetRentDbContext` e publicação do evento `bike.created` com dados da moto.

## 7.2 Cadastro de motoristas

- `DriverService` assegura CNPJ e número de CNH únicos.
- Upload de imagem recebe base64, converte para stream e grava arquivo (PNG/BMP) com nome determinístico `<driverId>_license`.
- Caso ocorra erro de IO ou permissão, o serviço retorna erro específico de armazenamento.

## 7.3 Ciclo de locação

- Criação: verifica existência de motorista e moto, categoria A habilitada, data de início futura e disponibilidade da moto.
- Encerramento: método `Rental.CloseAndCalculateTotal` calcula custo base conforme plano (`7, 15, 30, 45 ou 50 dias`).
- Multas: devolução antecipada aplica percentual definido; atraso adiciona R$50 por dia excedente.

## 7.4 Notificações assíncronas

- `BikeCreatedConsumer` escuta a fila RabbitMQ configurada e cria notificações para motos do ano vigente (2024).
- Notificações são persistidas e podem ser expandidas para disparo de e-mails/SMS futuramente.

# 8. Contratos de API

| Recurso | Método | Descrição |
|---|---|---|
| /bikes | POST | Cadastra nova moto e publica `bike.created`. |
| /bikes?plate=XXX1234 | GET | Lista motos filtrando opcionalmente por placa. |
| /bikes/{id} | GET | Recupera moto por identificador. |
| /bikes/{id}/plate | PUT | Atualiza placa garantindo unicidade. |
| /bikes/{id} | DELETE | Remove moto sem locações ativas. |
| /drivers | POST | Cadastra motorista validando CNPJ e CNH. |
| /drivers/{id}/license | POST | Realiza upload de imagem da CNH (PNG/BMP base64). |
| /rentals | POST | Cria locação com data futura e plano definido. |
| /rentals/{id} | GET | Consulta locação incluindo dados do plano. |
| /rentals/{id}/return | PUT | Finaliza locação e calcula totais. |

# 9. Persistência de dados

O banco PostgreSQL é gerenciado via Entity Framework Core. As migrations residem em `FleetRent.Infrastructure/Migrations` e são aplicadas automaticamente na inicialização (`db.Database.Migrate()`). Seeds adicionam motos e motoristas para facilitar testes.

# 10. Mensageria e integração

RabbitMQ é utilizado para comunicação assíncrona. O publisher recebe o evento após cadastro da moto e o consumidor `BikeCreatedConsumer` transforma mensagens em notificações persistidas. As configurações de conexão são definidas em `appsettings.json`.

# 11. Armazenamento de arquivos

A implementação `FileStorageService` garante criação de diretórios, valida permissão de escrita e normaliza o caminho relativo retornado ao domínio. Erros de IO são traduzidos para `DomainErrors.Storage`, permitindo respostas claras na API.

# 12. Observabilidade e concerns transversais

- Logging com Serilog enriquecido com ambiente, máquina, processo e thread.
- `RequestLocalizationOptions` define cultura `pt-BR` para formatos de data/moeda.
- Swagger habilitado em ambientes de desenvolvimento para inspeção interativa dos endpoints.

## 13. Testes automatizados

O projeto `FleetRent.Tests` utiliza xUnit para cobrir regras críticas. Existem suítes para o domínio (`Bike`, `Driver`, `Rental`) e para `RentalService`, validando cálculo de totais, restrições de placa e habilitação, além de conflitos de agenda.

## 14. Guia de execução

Para ambientes locais recomenda-se Docker Compose (`docker compose up -d`) que provisiona API, PostgreSQL e RabbitMQ. Sem Docker, é necessário configurar PostgreSQL (`fleetrent/fleetrent`), aplicar migrations e executar `dotnet run --project FleetRent.API`.

## 15. Próximos passos e extensões sugeridas

- Criar endpoint para consumir notificações e atualizar seu status (lido/processado).
- Adicionar autenticação/autorização (por exemplo, JWT) para separar perfis operacionais.
- Monitorar filas RabbitMQ e implementar retentativas ou DLQ para mensagens não processadas.
- Automatizar pipeline CI/CD com execução de testes e análise estática.

## 16. Estrutura do repositório

| Caminho | Descrição |
|---|---|
| FleetRent.Domain | Modelos de domínio, enums, erros e contratos. |
| FleetRent.Application | Serviços de aplicação, DTOs e casos de uso. |
| FleetRent.Infrastructure | Persistência EF Core, mensageria, storage e migrations. |
| FleetRent.API | Host ASP.NET Core com controllers REST e configuração. |
| FleetRent.Tests | Testes automatizados de domínio e aplicação. |
| docker-compose.yml | Provisiona API, PostgreSQL e RabbitMQ em contêineres. |