Documentação Técnica: Sistema de Gerenciamento de Biblioteca

Versão: 2.0

Autores: Daired Almeida Cruz, Hugo Alves dos Santos

Data: 30 de julho de 2025

Sumário

---

1. Introdução

1.1. Visão Geral do Projeto

Este projeto consiste no desenvolvimento de um ecossistema de software coeso e moderno para o gerenciamento de uma biblioteca. A solução é composta por três componentes principais que operam de forma integrada:

1. **Uma Aplicação Desktop robusta**, construída com JavaFX, destinada à administração interna da biblioteca. Permite que os funcionários realizem todas as operações de CRUD (Create, Read, Update, Delete) para gerenciar o acervo, usuários e empréstimos.
2. **Uma API RESTful performática**, desenvolvida com Spring Boot, que serve como uma interface pública para consulta de informações, como a disponibilidade de livros e suas resenhas.
3. **Um Módulo de Integração (Middleware)**, utilizando Apache Camel e JMS (Java Message Service), que atua como a espinha dorsal da comunicação, garantindo que os dados permaneçam sincronizados entre o banco de dados relacional da aplicação desktop e o banco de dados NoSQL da API.

1.2. Descrição do Problema

A gestão de bibliotecas enfrenta desafios significativos que sistemas legados ou processos manuais não conseguem resolver adequadamente. Este projeto aborda diretamente os seguintes problemas:

- **Ineficiência Operacional:** A falta de um sistema centralizado para o gerenciamento do acervo resulta em um gasto excessivo de tempo em tarefas administrativas e aumenta a probabilidade de erros humanos no registro de dados.
- **Falta de Acessibilidade:** Em um cenário tradicional, os usuários não têm meios eficientes para consultar o acervo remotamente, verificar a disponibilidade de um livro ou gerenciar seus empréstimos, o que gera uma experiência de usuário deficiente.
- **Inconsistência de Dados:** Um problema técnico comum é a existência de múltiplos bancos de dados que não se comunicam (silos de dados). Por exemplo, um banco relacional para gestão interna e um NoSQL para consultas públicas. Manter a consistência entre esses dois sistemas é uma tarefa complexa que, se mal executada, leva a informações conflitantes e desatualizadas.

1.3. Objetivos do Sistema

O objetivo geral do projeto é desenvolver um sistema de gerenciamento de biblioteca multiplataforma, eficiente e integrado. Para isso, foram definidos os seguintes objetivos específicos:

1. **Desenvolver a Aplicação Desktop:** Implementar uma interface rica e intuitiva com JavaFX para todas as operações de CRUD (Livro, Autor, Usuário, etc.), utilizando SQLite como banco de dados local e publicando todas as alterações de dados em uma fila de mensagens.
2. **Desenvolver a API RESTful:** Construir endpoints para consulta pública do acervo, utilizando Spring Boot e MongoDB para garantir consultas rápidas e flexíveis. A API

deve ser capaz de consumir mensagens da fila para manter seu próprio banco de dados sincronizado.
3. **Implementar o Integrador de Dados:** Garantir a sincronização de dados de forma assíncrona e desacoplada entre a aplicação desktop e a API, utilizando uma solução de mensageria para mediar a comunicação e aumentar a resiliência do sistema.

---

2. Arquitetura Geral do Sistema

2.1. Visão Arquitetural

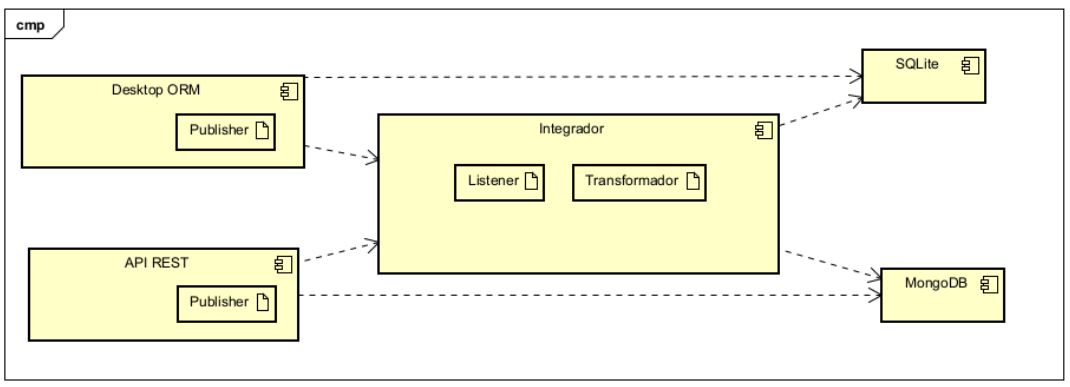
A arquitetura do sistema foi projetada para ser modular e desacoplada, permitindo que cada componente (Desktop, API, Integrador) evolua de forma independente.

- **Desktop ORM (Aplicação Desktop):** Atua como o sistema de gerenciamento principal. Todas as alterações de dados (inserções, atualizações, exclusões) são primeiramente realizadas aqui. Ele utiliza um banco de dados **SQLite** para persistência local e, após cada operação, publica uma mensagem em uma fila do ActiveMQ.
- **API REST:** Serve como a fachada pública para consultas. Ela não acessa diretamente o banco de dados do Desktop. Em vez disso, mantém sua própria base de dados otimizada para leitura em **MongoDB**.
- **Integrador:** É o middleware que conecta os dois sistemas. Ele contém a lógica de integração, composta por:
    - **Listeners:** "Ouvem" as mensagens publicadas tanto pelo Desktop quanto pela API.
    - **Transformadores:** Convertem os modelos de dados (schemas) de um sistema para o outro (ex: de um objeto relacional para um documento NoSQL).
    - **Publishers:** Publicam as mensagens transformadas na fila de destino apropriada.

Essa arquitetura de "publish-subscribe" garante que os sistemas não precisem ter conhecimento direto um do outro, promovendo baixo acoplamento e alta escalabilidade.

2.2. Diagrama de Componentes

O diagrama abaixo ilustra a relação entre os principais componentes do sistema:



2.3. Tecnologias Utilizadas

A tabela a seguir detalha as tecnologias e ferramentas empregadas em cada camada do projeto:

Camada/Componente	Tecnologia/Ferramenta	Finalidade
Aplicação Desktop	JavaFX	Construção da interface gráfica (GUI).
	ORMLite	Mapeamento Objeto-Relacional (ORM) para persistência de dados.
	SQLite	Banco de dados relacional embarcado.
API RESTful	Spring Boot	Framework para a criação da API.
	Spring Data MongoDB	Mapeamento Objeto-Documento (ODM) para o MongoDB.
	MongoDB	Banco de dados NoSQL para consultas públicas.
Integrador	Apache Camel	Framework de integração para orquestrar o fluxo de mensagens.

	Apache ActiveMQ	*Message Broker* para comunicação assíncrona via JMS.
	JMS (Java Message Service)	API padrão Java para envio e recebimento de mensagens.
Geral	Maven	Gerenciamento de dependências e build do projeto.
	JUnit	Framework para testes unitários.
	Git & GitHub	Controle de versão e colaboração.

---

3. Aplicação Desktop (Sistema de Administração)

3.1. Visão Geral e Propósito

A aplicação desktop é a ferramenta central para a administração da biblioteca. Foi desenvolvida em Java 17 com a estrutura de interface gráfica JavaFX. Seu principal objetivo é fornecer uma interface intuitiva e completa para que os funcionários possam realizar todas as operações de CRUD sobre as entidades do sistema. A persistência de dados é gerenciada localmente por um banco de dados SQLite, com o acesso facilitado pelo framework ORMLite.

3.2. Estrutura de Pacotes (MVC)

O projeto foi organizado seguindo uma variação do padrão arquitetural MVC (Model-View-Controller) para garantir uma clara separação de responsabilidades:

- `client`: Contém a classe `MainApp.java`, que é o ponto de entrada da aplicação JavaFX, responsável por inicializar a interface e os dados primários.
- `model`: Contém as classes de entidade (POJOs) que representam as tabelas do banco de dados (ex: `Livro.java`, `Autor.java`). Também inclui a camada de acesso a dados, com o `Repositorio.java` genérico e o `Database.java` para gerenciamento da conexão.

- view: Contém os arquivos FXML que definem a estrutura visual das telas e as classes "ViewModel" (ex: view.Livro.java) que adaptam as entidades do modelo para serem facilmente consumidas pelos componentes do JavaFX (data binding).
- controller: Contém as classes que fazem a ponte entre a visão e o modelo, abrigando toda a lógica de negócio e de manipulação de eventos da interface.

3.3. Camada de Modelo (Model)

A camada de modelo é o coração dos dados da aplicação.

- **Entidades ORMLite:** Classes como model.Livro.java e model.Autor.java são anotadas com as diretivas do ORMLite (ex: @DatabaseTable, @DatabaseField) para mapear seus atributos diretamente para as colunas das tabelas no banco de dados SQLite.
- **Repositório Genérico:** A classe Repositorio<T, ID> é uma implementação genérica do padrão Repository. Ela encapsula as operações CRUD básicas do ORMLite (create, update, delete, queryForId), permitindo uma grande reutilização de código.
- **Fábrica de Repositórios:** A classe Repositorios.java centraliza a inicialização de todos os repositórios da aplicação, fornecendo um ponto de acesso estático e único para cada um (ex: Repositorios.LIVRO, Repositorios.AUTOR).

3.4. Camada de Visão (View)

A camada de visão é responsável pela apresentação dos dados.

- **FXML:** Todos os layouts de tela são definidos em arquivos .fxml (ex: livro.fxml, autor.fxml), que são arquivos XML que descrevem a hierarquia dos componentes da interface. Isso separa o design da interface da lógica de programação.
- **ViewModels:** Para cada entidade principal, existe uma classe correspondente no pacote view (ex: view.Livro.java). Essas classes não são entidades de banco de dados, mas sim representações das entidades otimizadas para o JavaFX, utilizando propriedades como SimpleStringProperty e SimpleIntegerProperty, que facilitam a vinculação de dados com componentes como TableView.

3.5. Camada de Controle (Controller)

Esta camada contém a lógica da aplicação.

- **AbstractCrudController:** Esta classe abstrata é uma das peças centrais da arquitetura do desktop. Ela implementa toda a lógica comum para as telas de CRUD, como o gerenciamento de estados (visualização, criação, edição), o tratamento de cliques nos botões (Novo, Atualizar, Deletar, Confirmar, Cancelar), a validação de campos obrigatórios e a atualização automática da TableView.
- **Controladores Específicos:** Controladores como LivroController.java e AutorController.java herdam de AbstractCrudController e implementam apenas os métodos abstratos necessários para lidar com as particularidades de suas respectivas entidades (ex: qual repositório usar, como converter o modelo para a visão e como preencher os campos específicos da tela).

3.6. Persistência de Dados com ORMLite e SQLite

- **SQLite:** Foi escolhido por ser um banco de dados leve, embarcado e sem a necessidade de um servidor, ideal para aplicações desktop. O banco de dados é um único arquivo (`biblioteca_desktop.sqlite`) na raiz do projeto.
- **ORMLite:** Atua como a camada de abstração entre os objetos Java e o banco de dados relacional. Ele simplifica drasticamente as operações de banco de dados, convertendo chamadas de método (ex: `dao.create(livro)`) em comandos SQL.

---

4. API RESTful (Sistema de Consulta)

4.1. Visão Geral e Propósito

A API RESTful, desenvolvida com Java 17 e Spring Boot, é a interface pública do sistema. Seu principal objetivo é fornecer endpoints rápidos e flexíveis para que sistemas externos ou futuras aplicações (como um aplicativo móvel ou um site) possam consultar informações sobre o acervo da biblioteca, incluindo detalhes de livros e suas resenhas. Ela utiliza o MongoDB como banco de dados, uma escolha que favorece a agilidade em esquemas de dados que podem evoluir e a performance em operações de leitura intensiva.

4.2. Estrutura de Pacotes

O projeto da API segue a estrutura padrão de aplicações Spring Boot:

- `controller`: Contém as classes que definem os endpoints REST da API, como `LivroController.java`.
- `model`: Abriga as classes que representam os documentos no MongoDB, como `Livro.java` e a classe embutida `Resenha.java`.
- `repository`: Contém as interfaces que estendem o `MongoRepository` do Spring Data, automatizando o acesso aos dados.
- `util`: Classes utilitárias, como o `JmsPublisher.java`, responsável por publicar mensagens de alteração na fila.

4.3. Camada de Modelo (Documentos NoSQL)

Diferente do modelo relacional do desktop, a API utiliza um modelo de documentos NoSQL:

- **`Livro.java`:** Esta classe é anotada com `@Document(collection = "livros")`, mapeando-a para a coleção `livros` no MongoDB.
- **Documentos Embutidos:** A classe `Resenha.java` não é uma coleção separada no MongoDB. Em vez disso, uma lista de resenhas (`List<Resenha>`) é embutida diretamente dentro do documento do `Livro` correspondente. Esta abordagem é altamente performática para leitura, pois todas as informações de um livro e suas resenhas podem ser recuperadas em uma única consulta ao banco de dados.

4.4. Camada de Repositório (Spring Data MongoDB)

A interação com o MongoDB é simplificada pelo Spring Data. A interface `LivroRepository.java` estende `MongoRepository<Livro, String>`, e com isso, o Spring automaticamente provê a implementação para todas as operações básicas de CRUD, sem a necessidade de escrever uma única linha de código de acesso a dados.

4.5. Camada de Controle (Endpoints)

A classe LivroController.java utiliza anotações do Spring Web (@RestController, @RequestMapping, @GetMapping, @PostMapping, etc.) para definir os endpoints da API. Ela injeta o LivroRepository para executar as operações de banco de dados e retorna os resultados como JSON.

4.6. Documentação dos Endpoints

A seguir, uma descrição detalhada dos endpoints disponíveis na API, que se encontram em LivrariaAPI/src/main/java/com/livraria/api/controller/LivroController.java.

POST /api/livros

- **Descrição:** Cria um novo livro.
- **Corpo da Requisição (JSON):**
- JSON exemplo

```
{
   "isbn": "978-8576082675",
   "titulo": "teste Script",
   "anoPublicacao": 2008,
   "edicao": "1ª",
   "numPaginas": 464,
   "sinopse": "Um guia para o desenvolvimento de software ágil e de qualidade.",
   "autores": ["Robert C. Martin"],
   "categoria": "Tecnologia",
   "resenhas": [
       {
           "nomeUsuario": "Ana",
           "nota": 5,
           "texto": "Leitura obrigatória para qualquer desenvolvedor!"
       }
   ]
}
```

GET /api/livros

- **Descrição:** Lista todos os livros cadastrados.

GET /api/livros/{id}

- **Descrição:** Busca um livro específico pelo seu ID do MongoDB.

PUT /api/livros/{id}

- **Descrição:** Atualiza um livro existente.

DELETE /api/livros/{id}

- **Descrição:** Deleta um livro pelo seu ID.

POST /api/livros/{livroId}/resenhas

- **Descrição:** Adiciona uma nova resenha a um livro existente.
- **Corpo da Requisição (JSON):**
- JSON

```
{
  "nomeUsuario": "Carlos",
  "nota": 5,
  "texto": "Leitura obrigatória para todo desenvolvedor."
}
```

---

5. Módulo Integrador (Middleware com Apache Camel)

5.1. Visão Geral e Propósito

O Módulo Integrador é o componente central que garante a consistência de dados entre a Aplicação Desktop e a API RESTful. Construído com Apache Camel, ele atua como um middleware de mensagens, orquestrando o fluxo de informações de forma assíncrona. Seu principal papel é consumir mensagens de uma fila, transformar os dados do formato de origem para o formato de destino e, em seguida, interagir com o sistema de destino (seja chamando um endpoint da API ou atualizando diretamente o banco de dados do desktop).

5.2. Fluxo de Sincronização: Desktop → API

Este fluxo é ativado quando uma operação de CRUD é realizada na aplicação desktop. A rota está definida em integrador/src/main/java/com/livraria/integrador/routes/LivrariaRouteBuilder.java.

1. **Consumo da Fila:** A rota rota-desktop-para-api consome mensagens da fila DESKTOP_PARA_API_QUEUE.
2. **Transformação:** A mensagem (um JSON do LivroSyncDto) é passada para o processador DesktopToApiProcessor. Este processador converte o DTO do desktop para o formato LivroApi, que é compatível com o que a API RESTful espera. Ele também consulta o banco de dados canônico para obter o ID do livro na API, caso a operação seja um UPDATE ou DELETE.
3. **Roteamento para a API:** Com base no *header* operacao (CREATE, UPDATE, DELETE), o Apache Camel dinamicamente constrói a URL e o método HTTP corretos e envia a requisição para a API RESTful.
4. **Persistência Canônica:** Após a API responder com sucesso, o PersistenciaCanonicoProcessor é acionado. Se a operação foi um CREATE, ele salva a relação entre o ID do desktop e o novo ID gerado pela API no banco de dados de mapeamento. Se foi DELETE, ele remove o mapeamento.

5.3. Fluxo de Sincronização: API → Desktop

Este fluxo é ativado por operações na API (atualmente via JmsPublisher na API, mas idealmente seria por um mecanismo de captura de eventos do MongoDB).

1. **Consumo da Fila:** A rota rota-api-para-desktop consome mensagens da fila API_PARA_DESKTOP_QUEUE.
2. **Processamento e Atualização:** A mensagem (JSON do Livro do MongoDB) é passada para o ApiToDesktopProcessor. Este processador contém a lógica para:
    - Desserializar o JSON da API.
    - Chamar o DatabaseUpdater, uma classe utilitária que se conecta diretamente ao banco de dados SQLite do desktop.
    - O DatabaseUpdater realiza a sincronização, criando ou atualizando o livro, autores, categorias e, crucialmente, as resenhas no banco de dados do desktop para refletir o estado da API.

5.4. Mapeamento Canônico de Dados

Um dos principais desafios da integração é que os sistemas utilizam chaves primárias de tipos diferentes: Integer no SQLite do desktop e String (ObjectId) no MongoDB da API. Para resolver isso, foi criado um **modelo de dados canônico**.

- **Banco de Dados do Integrador:** O integrador mantém seu próprio banco de dados SQLite (integrador_dados.sqlite).
- **Tabela de Mapeamento:** Este banco contém uma única tabela, definida pela classe MapeamentoID.java, que armazena a relação entre os IDs:
    - idDesktop (Integer)
    - idApi (String)
    - idCanonico (UUID, chave primária do mapeamento)

Quando o integrador processa uma mensagem, ele consulta esta tabela para traduzir o ID de um sistema para o outro antes de realizar a operação no sistema de destino.

5.5. Processadores de Transformação

- **DesktopToApiProcessor:** Converte o LivroSyncDto (com resenhas e autores) do desktop para um objeto LivroApi, que é o formato JSON esperado pela API.
- **ApiToDesktopProcessor:** Contém a lógica principal de sincronização da API para o desktop. Ele invoca o DatabaseUpdater.
- **DatabaseUpdater:** Uma classe crucial que abre uma conexão direta com o arquivo biblioteca_desktop.sqlite e usa DAOs do ORMLite para realizar as operações de CRUD, garantindo que as informações da API (incluindo resenhas) sejam refletidas no banco de dados do desktop.

Com certeza! Preparei a seção de documentação "Instruções de Instalação e Execução" completamente atualizada, refletindo o novo processo simplificado com Docker Compose e o script de inicialização.

Integrei esta seção ao documento técnico completo que estávamos elaborando. Abaixo está a versão final e detalhada da documentação.

---

6. Instruções de Instalação e Execução (Orquestrado com Docker)

Esta seção descreve como compilar, configurar e executar todos os componentes do sistema de forma integrada, utilizando Docker e Docker Compose para simplificar a gestão da infraestrutura de backend.

6.1. Pré-requisitos

Antes de iniciar, garanta que os seguintes softwares estejam instalados e em execução na sua máquina:

- **Java 17 (ou superior):** Necessário para compilar e executar todos os módulos do projeto.
- **Apache Maven:** Utilizado para o gerenciamento de dependências e automação do build.
- **Docker:** A plataforma de contêineres que executará os serviços de backend.
- **Docker Compose:** Ferramenta para definir e gerenciar aplicações Docker multi-container.

6.2. Execução Simplificada (Recomendado)

Para facilitar ao máximo o processo, foi criado um script de inicialização (`start.sh`) na raiz do projeto. Este script automatiza a compilação de todos os módulos, a inicialização dos serviços de backend via Docker Compose e a execução da aplicação desktop.

Para iniciar todo o sistema:

1. **Abra um terminal** na raiz do projeto (`grupo8`).
    - **(Apenas na primeira vez)** Dê permissão de execução ao script:

      chmod +x start.sh
    - **Execute o script:**

      ./start.sh

O script realizará as seguintes ações:

- Compilará os projetos `LivrariaAPI`, `integrador` e `biblioteca-desktop-app` com o Maven.
- Construirá as imagens Docker para a API e o Integrador.
- Iniciará os containers do MongoDB, ActiveMQ, da API e do Integrador em segundo plano (`-d`).

- Por fim, executará a aplicação desktop JavaFX.

Ao fechar a janela da aplicação desktop, o script prosseguirá e executará `docker-compose down` automaticamente, parando e removendo todos os containers do backend.

6.3. Execução Manual (Alternativa)

Caso prefira executar cada passo manualmente:

- **Compile os Módulos:** Execute os seguintes comandos a partir da raiz do projeto:

  mvn clean install -f LivrariaAPI/pom.xml

  mvn clean package -f integrador/pom.xml

  mvn clean install -f biblioteca-desktop-app/pom.xml

- **Inicie a Infraestrutura de Backend:** Na raiz do projeto, execute o Docker Compose. O comando `--build` garantirá que as imagens Docker sejam recriadas com o código mais recente.

  docker-compose up --build -d

- **Inicie a Aplicação Desktop:** Com o backend rodando, execute o seguinte comando para abrir a interface gráfica:

  mvn javafx:run -f biblioteca-desktop-app/pom.xml

- **Para Parar os Serviços:** Quando terminar de usar a aplicação, volte ao terminal na raiz do projeto e execute:

  docker-compose down

6.4. Acessando os Serviços

Com os containers em execução, os serviços estarão disponíveis nos seguintes endereços na sua máquina local (host):

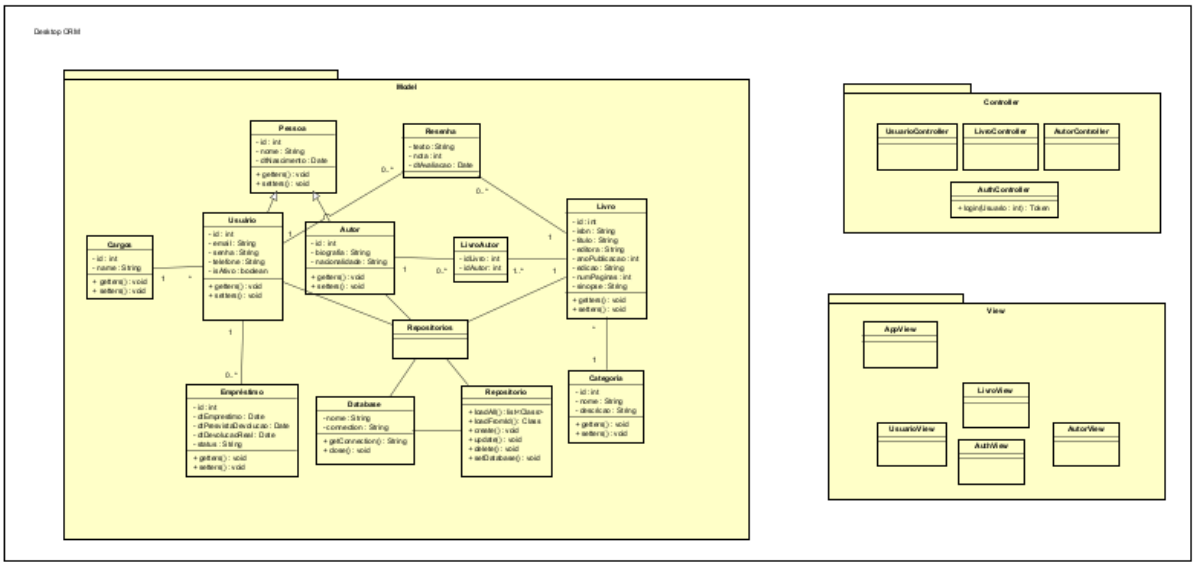
- **API RESTful:**
  - **URL Base:** `http://localhost:8080`
  - **Exemplo (Listar Livros):** `GET http://localhost:8080/api/livros`
- **Console Web do ActiveMQ:**
  - **URL:** `http://localhost:8162`
  - **Credenciais:** `admin` / `admin`

- **Conexão com o MongoDB (via cliente como MongoDB Compass):**
    - **URI de Conexão:** `mongodb://localhost:27018`
- **Aplicação Desktop (Login):**
    - **Email:** `admin@admin`
    - **Senha:** `admin`

---

7. Diagramas UML Adicionais

7.1. Diagrama de Classes ORM



7.2. Diagrama de Sequência