DOCUMENTAÇÃO SOBRE O BANCO DE DADOS DE FILME

Relatório Técnico

Case DTI Digital

Equipe:

Gabriel Campos Prudente - gabrielcamposprudente19@gmail.com

5 de agosto de 2025

Conteúdo

1 Resumo Executivo

Este relatório detalha a arquitetura e a implementação de um sistema de banco de dados voltado ao gerenciamento de filmes desenvolvido em C++. O software oferece funcionalidades de Cadastro, Leitura, Atualização e Exclusão (CRUD) para uma coleção de filmes, utilizando um banco de dados SQLite para a persistência dos dados. A interação com o usuário é realizada através de uma interface de linha de comando (CLI). O sistema foi projetado com foco na separação de responsabilidades, robustez no tratamento de erros, validação de entradas e uma cobertura de testes abrangente, garantindo alta confiabilidade e manutenibilidade.

2 Introdução

A necessidade de gerenciar coleções de dados de forma eficiente é um desafio comum em desenvolvimento de software. Este projeto aborda essa necessidade no contexto de um bando de dados de filmes, fornecendo uma solução de desktop para catalogar e manter informações sobre as produções cinematográficas. O sistema foi construído utilizando C++ e SQLite, com ênfase em tecnicas de engenharia de software, como o encapsulamento de lógica em classes, o tratamento explícito de exceções; a implementação de uma validações de robustez, como tratamento de erros, validações e entradas e uma suíte de testes; e a conteinerização via Docker, para permitir a distribuiçaõ mais factível do software.

3 Objetivo

3.1 Objetivo Geral

O principal objetivo desse software é desenvolver um sistema de software robusto, interativo e bem testado para o gerenciamento completo de um banco de dados de filmes, permitindo ao usuário final realizar as operações fundamentais de CRUD de forma intuitiva e segura. O sistema destaca-se na versalitade e na facilidade do uso, sendo ideal para servir de "back-end"para aplicações com usuários. Além disso, o código também é fortemente estruturado via técninas de engenharia de softwate, permitindo assim, uma manutenção facil, bem como a expansão do código.

3.2 Objetivos Específicos

- **Persistência de Dados:** Utilizar o SQLite para armazenar as informações dos filmes.

- **Interface de Usuário:** Implementar uma interface de linha de comando clara e objetiva.
- **Operações CRUD:** Fornecer funcionalidades completas para cadastrar, listar, buscar, atualizar e deletar filmes.
- **Validação de Entradas:** Assegurar a integridade dos dados através de validações rigorosas das entradas do usuário.
- **Tratamento de Erros:** Implementar um sistema de tratamento de exceções para lidar com falhas operacionais de forma controlada.
- **Geração de Relatórios:** Oferecer a funcionalidade de gerar relatórios de atividades (curtos ou completos).
- **Garantia de Qualidade:** Validar o comportamento do sistema através de testes unitários e de integração.
- **Conteineirização:** Permitir que o sistema seja executado em diversos ambientes, por meio de uma padronização via Docker.

4 Ferramentas utilizadas

- **Linguagem de Programação:** C++ (padrão C++17).
- **Banco de Dados:** SQLite 3.
- **Framework de Testes:** Doctest, uma biblioteca de testes "single-header"para C++.
- **Compilador:** g++ / Clang.
- **Sistema de Arquivos:** Biblioteca `<filesystem>` do C++17. `Sistema de Containers:` Docker

5 Arquitetura e Modelagem

5.1 Modelagem do Banco de Dados

O banco de dados é criado utilizando uma query SQL de criação de tabela. Com isso, é criado uma tabela que servirá de componente principal do banco de dados.

Essa tabela, denominada `filmes`, é projetada para armazenar todas as informações essenciais de cada obra catalogada. Cada filme é unicamente identificado por uma coluna `id` numérica, que é uma chave primária de autoincremento, garantindo que cada registro

tenha um identificador exclusivo gerado automaticamente pelo banco de dados. Campos fundamentais como `titulo`, `diretor` e `data_lancamento` são definidos como obrigatórios (`NOT NULL`), assegurando a integridade e a completude dos dados essenciais para cada entrada. Para maior flexibilidade, a modelagem inclui campos opcionais: o `genero` pode ser deixado em branco, e a `duracao_minutos` assume um valor padrão de `0` caso não seja especificada, evitando a ausência de valor para um campo numérico.

- `id`: Identificador único para o filme. O tipo de dado é `INTEGER` e atua como chave primária com autoincremento automático (`PRIMARY KEY AUTOINCREMENT`).
- `titulo`: Armazena o título do filme. É um campo do tipo `TEXT` e de preenchimento obrigatório (`NOT NULL`).
- `diretor`: Guarda o nome do diretor. O tipo de dado é `TEXT` e também é um campo obrigatório (`NOT NULL`).
- `data_lancamento`: Armazena a data de lançamento do filme. Seu tipo é `DATE` e seu preenchimento é obrigatório (`NOT NULL`).
- `genero`: Campo para o gênero do filme. É do tipo `TEXT` e seu preenchimento é opcional, podendo ser nulo.
- `duracao_minutos`: Guarda a duração do filme em minutos. É um campo numérico do tipo `INT`. Caso não seja fornecido, assume o valor padrão de 0 (`DEFAULT 0`).

5.2 Arquitetura do Sistema

O sistema foi modelado seguindo o princípio da separação de responsabilidades.

1. **Filme:** Classe de entidade que modela um filme.
2. **GerenciadorDeFilmes:** Camada de acesso a dados (DAL) estática, responsável pela comunicação com o banco de dados.
3. **Interface:** Camada de apresentação, responsável pela interação com o usuário (CLI).
4. **GeradorRelatorioAtividade:** Módulo para registrar as operações da sessão.
5. **ArquivosUteis:** Namespace com utilitários, como o `GerenciadorDeEntradas` para validação de input e a função `lerQueriesDoArquivo`.

6 Compreensão das classes

A seguir, será detalhada cada classe principal utilizada no software. A definição macro do projeto foi feita utilizando o princípio de Responsabilidade Única, no qual cada classe possui apenas uma responsabilidade. Essa abordagem foi feita visando separar as funcionalidades do projeto, para que as classes sejam coesas, mas não dependam exclusivamente de outras classes. Isso torna, também, a testagem muito mais fácil, bem como a verificação de erros e os tratamentos deles.

6.1 Classe `Filme`

Arquivos: `Filme.hpp`, `Filme.cpp`

Esta classe é a representação de dados (entidade) do sistema. Ela é a principal representação do artefato de estudo, logo encapsula todos os atributos de um filme, servindo como uma estrutura de dados utilizada em toda a aplicação para transportar informações sobre filmes.

- **Responsabilidade:** Manter o estado de um único filme, incluindo seu ID, título, diretor, data de lançamento, gênero e duração.
- **Construtores:** Possui múltiplos construtores para flexibilidade na criação de objetos: um construtor padrão, um para criar filmes novos (sem ID) e outro para recriar objetos a partir de dados do banco de dados (com ID).
- **Métodos Principais:**
  - `getters` e `setters`: Fornecem acesso controlado aos atributos privados da classe (e.g., `getTitulo()`, `setTitulo()`).
  - `toString()`: Converte os dados do filme em uma única string formatada, útil para registro em logs e relatórios.
  - `exibir()`: Imprime no console os detalhes do filme de forma legível e organizada, sendo o principal método para apresentação ao usuário.
- **Interações:** É criada na `Interface` com dados do usuário, manipulada pelo `GerenciadorDeFilmes` para persistência e utilizada pelo `GeradorRelatorioAtividade` para detalhar operações.

6.2 Classe `GerenciadorDeFilmes`

Arquivos: `GerenciadorDeFilmes.hpp`, `GerenicadorDeFilmes.cpp`

Atua como a camada de acesso a dados (Data Access Layer - DAL). Esta classe é inteiramente estática, pois suas funções são operações sem estado que atuam diretamente

sobre a conexão com o banco de dados - BD. É nela que ocorre todas as interações com o banco de dados, e de onde são lançadas as exceções de falhas no BD. Ela também é responsavel por ler os arquivos, que contém as querys, da pasta SQL

- **Responsabilidade:** Abstrair toda a complexidade da comunicação com o banco de dados SQLite. É a única classe que executa queries SQL.
- **Métodos Principais:** Implementa as operações CRUD.
  - `inserir(db, filme)`: Recebe um objeto `Filme`, lê a query de `inserção` associa os dados do filme aos parâmetros da query e a executa.
  - `buscarPorId(db, id)`: Lê a query de `seleção`, executa a busca e, se encontrar, popula e retorna um objeto `Filme` com os dados. Caso não encontre, retorna um Filme vázio, que é tratado na camada de interface
  - `listarTodos(db)`: Executa a query de `listagem` e retorna um `std::vector<Filme>` com todos os registros do banco. Caso não haja filme, retorna um vetor vazio.
  - `atualizar(db, filme)`: Lê `atualização` e atualiza o registro no banco correspondente ao ID do objeto `Filme` fornecido. Se o ID não existir no BD, retorna uma filme vazio, que é tratado na Interface.
  - `deletar(db, id)`: Lê `delete_filme.sql` para remover o filme com o ID especificado. Se o ID não existir no BD, retorna uma filme vazio, que é tratado na Interface.
- **Tratamento de Erros:** Lança exceções específicas para cada tipo de falha no banco de dados (e.g., `Erro_ao_inserir_no_banco`), permitindo que a camada de interface trate os erros de forma granular. Caso ids fornecidos não possuam correspondência no BD, a classe interpreta isso como uma query válida que retorna nulo. Por isso, é retornado um filme vazio. Porém, caso um id seja fornencido e possua correspondência no BD, mas não houve alteração no banco, no caso de atualizações e deleções, o gerenciador retorna um erro.

6.3 Classe `Interface`

Arquivos: `Interface.hpp`, `Interface.cpp`

É a camada de apresentação e o ponto de entrada da lógica da aplicação para o usuário final. É por ela que se da a maior parte dos tratamentos de erros lançados do banco de dados. Além disso, ela se conecta com todas as outras classes e faz uso da classe filme, para exibição das informações no console.

- **Responsabilidade:** Gerenciar o fluxo de interação com o usuário, exibindo o menu principal, coletando as entradas e orquestrando as chamadas para as outras camadas do sistema.

- **Estado Interno:** Mantém uma instância do `GeradorRelatorioAtividade` para registrar as ações e um ponteiro para a conexão do banco de dados (`sqlite3* _db`).
- **Métodos Principais:**
  - `ExibirMenu()`: É o coração da classe. Contém o laço principal do programa, que exibe as opções e direciona o fluxo com base na escolha do usuário através de um `switch-case`.
  - Métodos privados como `cadastrarNovoFilme()`, `atualizarFilme()`, etc., encapsulam a lógica de cada funcionalidade. Eles coletam os dados do usuário (usando `ArquivosUteis`), invocam os métodos correspondentes do `GerenciadorDeFilmes` e, em caso de sucesso, registram a operação no `geradorRelatorio`.
- **Interações:** Utiliza `ArquivosUteis` para ler e validar todas as entradas do usuário. Delega as operações de persistência ao `GerenciadorDeFilmes`. Registra todas as ações bem-sucedidas (ou tentativas) no `GeradorRelatorioAtividade`.

6.4 Classe `GeradorRelatorioAtividade`

Arquivos: `GeradorRelatorioAtividade.hpp`, `GeradorRelatorioAtividade.cpp`

Um componente utilitário focado em auditoria e logging das atividades da sessão. A toda operação realizada pelo usuário, um log é salvo em um arquivo `.txt`. Se ao final da interação, o usuário quiser manter uma relatório de atividades, ele tem a opção de salvar esse arquivo `.txt` na pasta reports. Caso o usuário não queira salvar seus logs, o arquivo é descartado. Além disso, é possível gerar um relatório curto, contendo apenas a operação realizada, juntamente com a data e hora, e um relatório completo, contendo a operação com data e hora, bem como o resultado dessas operação - geralmente os r`returns` das funções da classe de gerenciador de filmes.

- **Responsabilidade:** Registrar e, posteriormente, salvar um relatório das operações realizadas pelo usuário durante uma única execução do programa.
- **Métodos Principais:**
  - `registrarOperacao(operacao, resultado)`: Adiciona um novo `RegistroAtividade` (uma struct contendo a operação, o resultado e a hora) a um vetor interno. O resultado é opcional.
  - `salvarRelatorio(tipo)`: Com base no tipo ("curto"ou "completo"), formata o conteúdo do vetor de atividades e o salva em um arquivo de texto. O nome do arquivo inclui um timestamp para evitar sobreposições. A função também cria o diretório `reports` se ele não existir.
- **Interações:** É instanciado e utilizado exclusivamente pela classe `Interface`.

6.5 Namespace `ArquivosUteis`

Arquivos: `ArquivosUteis.hpp`, `ArquivosUteis.cpp`

Agrupa um conjunto de funcionalidades de suporte, focadas em validação de entrada e acesso a arquivos, que são usadas por todo o sistema.

- **Responsabilidade:** Fornecer ferramentas reutilizáveis para garantir a robustez e a segurança das entradas e para abstrair operações de I/O de arquivos.
- **Componentes Principais:**
  - **Classe `GerenciadorDeEntradas`:** Sub-componente mais crítico. Seus métodos estáticos (`lerInt`, `lerString`, `lerData`) encapsulam a lógica de leitura do `std::cin`, validação e tratamento de erros de entrada em um laço `while`, forçando o usuário a fornecer dados válidos antes de prosseguir.
  - **Função `checaData(data)`:** Realiza uma validação rigorosa de strings de data, verificando formato, limites de calendário (incluindo anos bissextos) e conformidade com os limites históricos do cinema.
  - **Função `lerQueriesDoArquivo(nomeArquivo)`:** Abstrai a leitura de arquivos de texto, usada especificamente para carregar as queries SQL para o `GerenciadorDeFilmes`. Lança `Erro_ao_ler_query` se o arquivo não puder ser aberto.

6.6 Classe `GerenciadorDeEntradas` (dentro de `ArquivosUteis`)

Arquivos: `ArquivosUteis.hpp`, `ArquivosUteis.cpp`, `test_GerenciadorDeEntradas.cpp`, `test_Data.cpp`

Este componente, aninhado dentro do namespace `ArquivosUteis`, é uma classe de utilitários estática que serve como um portal seguro e robusto para toda a entrada de dados proveniente do usuário via linha de comando. Sua principal função é abstrair a complexidade de ler, converter (parse) e, crucialmente, validar os dados, garantindo que a lógica de negócios da aplicação receba apenas informações em conformidade com os tipos e formatos esperados.

- **Responsabilidade:** Proteger a aplicação de dados inválidos que poderiam causar erros de execução ou corrupção de dados. Ela força o usuário a corrigir suas entradas em tempo real até que sejam válidas.
- **Métodos Principais e Lógica de Validação:**
  - `lerInt(opicional)`: Lê uma linha inteira do console. Utiliza um bloco `try-catch` para tentar converter a entrada para um inteiro usando `std::stoi`. Se a conversão falhar (e.g., o usuário digita "abc") ou se houver caracteres extras após o

número, uma exceção é capturada, uma mensagem de erro é exibida através de `imprimeFalhaDeTipoNumerico`, e o laço `while` solicita a entrada novamente. O parâmetro booleano `opicional` permite que uma entrada vazia seja aceita e retorne 0, útil para campos não obrigatórios.

- `lerFloat()` e `lerDouble()`: Seguem o mesmo padrão do `lerInt`, mas utilizam `std::stof` e `std::stod` para a conversão, respectivamente, garantindo a leitura de valores de ponto flutuante. A lógica de repetição em caso de erro é idêntica.
- `lerString(atualizar, opicional)`: Utiliza `std::getline` para capturar a linha inteira, permitindo entradas com espaços. A validação principal é feita pela função auxiliar `stringVazia`, que verifica se a entrada contém algum caractere que não seja espaço em branco. Se a string for considerada vazia e não for um campo opcional (controlado pelos parâmetros), a função `imprimeFalhaString` é chamada e o usuário deve digitar novamente. Os testes em `test_GerenciadorDeEntradas.cpp` validam esse comportamento, incluindo a aceitação de strings vazias quando o parâmetro opcional é verdadeiro.
- `lerData(atualizar, opicional)`: Orquestra a validação de datas. Assim como `lerString`, lê a linha de entrada, mas delega a validação para a função `checaData`. Se `checaData` determinar que a data é inválida (por formato, valor ou estar fora dos limites históricos), ela lança uma exceção `Data_invalida`. O método `lerData` captura essa exceção, informa o usuário através de `imprimeFalhaData`, e exige uma nova entrada.
- `lerEntradaRelatorio()`: Uma função de leitura especializada que força a entrada do usuário a ser estritamente "curto"ou "completo", rejeitando qualquer outra string em um laço de repetição.

- **Tratamento de Erros e Robustez:** A estratégia central de robustez da classe é o padrão "Laço de Repetição com Tratamento de Exceção"(`while(!leitura_com_sucesso) { try-catch }`). Este design garante que a execução do programa não prossiga com dados malformados. Ele isola completamente a lógica de negócios da complexidade e imprevisibilidade da entrada do usuário. Os testes unitários, como o que simula uma entrada de "string"seguida por "42", confirmam que este mecanismo de recuperação funciona como esperado.

- **Interações:** Por ser uma classe de utilitários estática, ela não mantém estado nem depende de outras partes da aplicação. É utilizada exclusivamente pela classe `Interface` no momento de coletar dados para as operações de CRUD. Essa baixa acoplagem a torna um componente altamente modular e reutilizável.

7 Estratégias de Robustez

7.1 Tratamento de Exceções

O sistema utiliza exceções customizadas para sinalizar erros específicos da camada de dados e de utilitários, como `Erro_ao_atualizar_no_banco` e `Data_invalida`. Para isso, são criadas diversas classes de exceção vazias, que permitem representar diferentes tipos de falhas de forma clara e eficiente, sem a necessidade de carregar mensagens ou dados adicionais. Esse tipo de abordagem facilita o uso de blocos `try-catch` específicos, onde a camada de interface pode capturar erros de forma granular e informar o usuário adequadamente. Além disso, essa estratégia mantém o código leve e organizado, permitindo futuras expansões caso se deseje adicionar mais informações às exceções.

7.2 Validação de Entradas

A classe `ArquivosUteis::GerenciadorDeEntradas` é fundamental para a integridade dos dados. Os testes em `test_GerenciadorDeEntradas.cpp` e `test_Data.cpp` confirmam a eficácia dessas validações.

- **Entradas Numéricas:** O método `lerInt` rejeita entradas não numéricas e solicita uma nova entrada até que um valor válido seja fornecido.
- **Entradas de Texto:** O método `lerString` impede a submissão de campos obrigatórios vazios ou contendo apenas espaços em branco.
- **Validação de Data:** A função `checaData` é rigorosa e rejeita múltiplos formatos inválidos, como demonstrado nos testes:
  - Formato incorreto: `"2023-8-3"` ou `"2023_08_03"`.
  - Dias/Meses inválidos: `"2023-13-10"` ou `"2023-04-31"`.
  - Anos fora do limite: Anos anteriores a 1895 (marco do cinema) ou datas futuras são rejeitados.
  - Anos não bissextos: Uma data como `"2023-02-29"` lança uma exceção.

  As datas válidas, como `"03-08-2023"`, são padronizadas para o formato `"03/08/2023"`.

Essa classe faz o usuário ficar "preso"dentro do console, caso não digite uma entrada válida. Logo, ela mostra-se como uma ferramenta robusta para lidar com erros de entradas do usuário.

7.3 Testes e Validação

A qualidade e a corretude do software são garantidas por uma suíte de testes automatizados usando o framework Doctest. Os testes cobrem diferentes camadas da aplicação e visam cobrir todos as classes e interfaces.

7.3.1 Testes Unitários

- **Classe Filme (`test_Filme.cpp`):** Testa os construtores, getters e setters para garantir que o estado do objeto `Filme` é gerenciado corretamente. Também valida a saída formatada do método `exibir()`.
- **Utilitários (`test_LerQuerys.cpp` e `test_Data.cpp`):**
  - A leitura de arquivos de query é testada para cenários de sucesso, arquivo inexistente e arquivo vazio.
  - A validação de data é extensivamente testada contra formatos válidos e uma vasta gama de formatos inválidos, garantindo a robustez da função `checaData`.
- **Gerenciador de Entradas (`test_GerenciadorDeEntradas.cpp`):** Testa a leitura de diferentes tipos de dados (int, float, double, string), incluindo cenários de falha onde o usuário fornece uma entrada inválida antes de uma válida.

7.3.2 Testes de Integração

Os testes em `test_GerenciadorDeFilmes.cpp` validam a interação entre a lógica da aplicação e o banco de dados SQLite. Um banco de dados de teste temporário (`testes_filme.bd`) é criado e destruído para cada conjunto de testes, garantindo um ambiente limpo e isolado.

- **Operações CRUD:** São testadas as operações de inserir, buscar por ID, listar todos, atualizar e deletar filmes.
- **Casos de Borda:** São verificados cenários como tentar atualizar ou deletar um filme com um ID que não existe no banco de dados. O teste de deleção de ID inexistente confirma que a operação falha de forma controlada, retornando `false`. O teste de atualização de ID inexistente confirma o lançamento da exceção `Erro_ao_atualizar_no_banco`.

8 Detalhes Técnicos Adicionais

8.1 Separação de Queries SQL

Uma decisão de design importante foi externalizar as queries SQL para arquivos `.sql`. Isso melhora a manutenibilidade e a legibilidade, permitindo que o código SQL seja modificado sem a necessidade de recompilar o código C++. Abaixo estão as queries utilizadas:

```sql
INSERT INTO filmes (titulo, diretor, data_lancamento, genero, duracao_minutos)
VALUES (?, ?, ?, ?, ?);
```

Listing 1: Inserir filme (insert_filme.sql)

```sql
SELECT id, titulo, diretor, data_lancamento, genero, duracao_minutos FROM
    filmes;
```

Listing 2: Selecionar todos os filmes (select_all_filmes.sql)

```sql
UPDATE filmes SET titulo = ?, diretor = ?, data_lancamento = ?, genero = ?,
    duracao_minutos = ? WHERE id = ?;
```

Listing 3: Atualizar filme (update_filme.sql)

```sql
DELETE FROM filmes WHERE id = ?;
```

Listing 4: Deletar filme (delete_filme.sql)

8.2 Geração de Relatórios

A classe `GeradorRelatorioAtividade` registra cada operação da `Interface` em um vetor. Ao final da sessão, o usuário pode salvar um relatório (curto ou completo) que é gerado e salvo na pasta `reports/` com um timestamp no nome do arquivo para garantir sua unicidade.

9 Requisitos para a execução do código

Os seguintes requisitos foram utilizados no Linux. Para a instalação no Windows, consulte as documentações correspondentes, ou utilize o docker no Windows.

9.1 Localmente

Para a execução do código localmente, é necessário ter os seguintes requisitos instalados no PC. Além disso, antes de executar qualquer comando de execução do sistema, é necessário criar o banco de dados, por meio da execução do seguinte comando:

```
sqlite3 filmes.db < sql/create_table.sql
```

- **Compilador C++**: `g++` com suporte ao padrão `C++17`.
  - Verifique com:

```
g++ --version
```

  - Instale no Ubuntu/Debian:

```
sudo apt update
sudo apt install g++
```

- **Make**: Ferramenta de automação de compilação.
  - Verifique com:

```
make --version
```

  - Instale no Ubuntu/Debian:

```
sudo apt install make
```

- **Biblioteca SQLite3 (com headers)**:
  - Necessária para compilar com o parâmetro `-lsqlite3`.
  - Instale no Ubuntu/Debian:

```
sudo apt install libsqlite3-dev
```

- **gcov** (opcional): Utilizado para gerar relatório de cobertura de código.
  - Já incluído com o `g++` na maioria das distribuições.
  - Para garantir:

```
sudo apt install gcov
```

- **Framework de Testes (opcional)**: Sugere-se o uso de `doctest.h`, incluído na pasta `third_party/`.
  - Para obter o arquivo:

```
wget https://raw.githubusercontent.com/doctest/doctest/master/doctest
   /doctest.h -P third_party/
```

Para compilar e executar o programa principal:

```
make run
```

Para compilar e executar todos os testes:

```
make tests
```

Para compilar e executar um teste específico:

```
make run-test TEST_FILE=nome_do_teste.cpp VERBOSITY=2
```

Para gerar o relatório de cobertura de código:

```
make cov
```

Para limpar os arquivos gerados:

```
make clean
```

9.2 Requisitos com Docker (Alternativa)

- **Docker instalado e funcionando**.
  - Verifique com:

    ```
    docker --version
    ```

  - Instale no Ubuntu:

    ```
    sudo apt update
    sudo apt install docker.io
    sudo systemctl start docker
    sudo systemctl enable docker
    ```

- **Imagem com ambiente de desenvolvimento C++**:
  - Recomenda-se utilizar a imagem oficial `gcc:latest`.
  - Baixe a imagem:

    ```
    docker pull gcc:latest
    ```

- **Volumes montados**: Código-fonte precisa ser montado no container.
  - Estrutura do projeto deve estar acessível com `$(pwd)` ou caminho relativo.
- **Comando típico para executar os testes via Docker**:

```
docker run --rm -v $(pwd):/app -w /app gcc:latest make tests
```

Observação: Caso o projeto utilize `doctest.h`, certifique-se de que o arquivo esteja presente na pasta `third_party/` dentro do volume montado.

10 Utilizando o sistema com o Docker

Para a execução com o Docker, é necessário já ter o docker instalado e funcionando no dispositivo a ser utilizado. Após isso, rode no terminal que contém o repositório, os seguintes comandos:

- **Construir a imagem Docker**:

```
docker build -t filmes-app .
```

- **Subir o container interativo com volume montado**:
  - O comando abaixo monta o diretório atual do seu computador dentro do container na pasta `/app`, para que o container acesse seu código:

```
docker run -it --rm -v $(pwd):/app -w /app filmes-app
```

  - Dentro do container, o diretório de trabalho é `/app`, onde você pode compilar e rodar o projeto normalmente.

- **Comandos para uso dentro do container**:
  - Compilar e executar o programa principal:

```
make run
```

  - Executar todos os testes:

```
make tests
```

  - Executar um teste específico:

```
make run-test TEST_FILE=meuteste.cpp VERBOSITY=2
```

10.1 Execução Direta com `docker run` (Sem Entrar no Container)

Executar comandos diretamente no container, sem abrir o terminal interativo:

- Rodar todos os testes:

```
docker run -it --rm -v $(pwd):/app -w /app filmes-app make tests
```

- Rodar o programa principal:

```
docker run -it --rm -v $(pwd):/app -w /app filmes-app make run
```

- Rodar teste específico:

```
docker run -it --rm -v $(pwd):/app -w /app filmes-app \
  make run-test TEST_FILE=meuteste.cpp VERBOSITY=2
```

- Rodar o arquivo de entrada:

```
docker run -i --rm \
  -v $(pwd):/app \
  -w /app \
  filmes-app \
  ./bin/main < entradas/simulacao_uso.txt
```

11 Conclusão

O desenvolvimento do sistema de gerenciamento de filmes culminou em uma aplicação funcional e robusta, que atende com êxito a todos os objetivos propostos. As funcionalidades essenciais de CRUD foram implementadas de forma segura, apoiadas por uma arquitetura bem definida que preza pela separação de responsabilidades entre as camadas de apresentação, acesso a dados e lógica de negócios.

A ênfase na robustez, evidenciada pelo tratamento rigoroso de exceções, pela validação detalhada das entradas do usuário e pela abrangente suíte de testes automatizados, garante a confiabilidade e a estabilidade do software. A incorporação de ferramentas como `Makefile` e `Docker` não apenas otimizou o ciclo de desenvolvimento, mas também assegurou a portabilidade e a facilidade de distribuição da aplicação, demonstrando a aplicação de práticas de DevOps em um contexto de desenvolvimento C++.

Como trabalho futuro, o projeto estabelece uma base sólida para diversas expansões. A implementação de uma interface gráfica (GUI), a exposição das funcionalidades através de uma API REST para consumo por outras aplicações, ou a adição de consultas mais complexas ao banco de dados são evoluções naturais e desejáveis. Em suma, o projeto não apenas entrega uma solução prática para o problema proposto, mas também serve como um portfólio concreto da aplicação de conceitos fundamentais e modernos da engenharia de software.