Movie Browsing API
Architecture Documentation

Software Architecture Team

December 2025 - Version 1.0

Contents

1 Introduction and Goals

The Movie Browsing API is a RESTful web service providing comprehensive access to movie information, including films, actors, and ratings. This MVP serves as a backend for applications displaying and managing movie-related data.

1.1 Requirements Overview

Core Features: CRUD operations for movies, actors, and ratings; many-to-many movie-actor relationships; one-to-many movie-rating relationships; Richardson Maturity Model Level 2 compliance; embedded responses with full related data; SQLite persistence.

Key Endpoints: /movies, /actors, /ratings with GET, POST, PUT, DELETE methods.

1.2 Quality Goals

1. **Maintainability**: Layered architecture enables independent layer modification
2. **RESTfulness**: Full RMM Level 2 compliance with proper HTTP verbs and status codes
3. **Data Integrity**: Embedded responses, cascade deletes, Pydantic validation
4. **Simplicity**: MVP approach, minimal complexity, straightforward patterns
5. **Deployability**: Docker containerization, dev container support, reproducible builds

1.3 Stakeholders

Frontend Developers expect clean RESTful API with embedded data. Backend Developers need clear architecture and maintainable code. System Administrators require simple containerized deployment. Project Stakeholders want MVP delivery with extension potential.

2 Architecture Constraints

Technical: Python 3.12, FastAPI framework, SQLite database, Poetry dependency management, Docker containerization.

Organizational: MVP approach, mandatory layered architecture (Database, Persistence, Business, API), no authentication in v1.0.

Conventions: PEP 8 coding style, RESTful endpoints, Pydantic validation, SQLAlchemy ORM, camelCase JSON (firstName), snake_case Python (first_name).

3 Context and Scope

3.1 Business Context

The API serves as a centralized movie data service. **Inputs**: HTTP requests for movie/actor data, rating submissions. **Outputs**: JSON responses with embedded relationships, resource IDs. **Partners**: Web/mobile apps, admin tools, data migration scripts.

3.2 Technical Context

Interfaces: REST over HTTP/1.1, JSON request/response, SQLite file protocol, OpenAPI 3.0 documentation.

Deployment: Docker container on port 8000, Uvicorn ASGI server, SQLite (movies.db) in app root, VS Code dev container support.

4 Solution Strategy

4.1 Technology Decisions

Technology	Rationale	Quality Impact
FastAPI	Auto docs, validation, async	Performance, Maintainability
SQLAlchemy	DB abstraction, relationships	Maintainability, Portability
Pydantic	Request/response validation	Data Integrity, Security
SQLite	Zero config, file-based	Deployability, Simplicity
Poetry	Modern dependency mgmt	Maintainability, Reliability

4.2 Architectural Patterns

Four-Layer Architecture:

1. **API Layer**: FastAPI routes, Pydantic schemas, HTTP handling
2. **Business Layer**: Service classes, business logic, data transformation
3. **Persistence Layer**: Repository pattern, data access abstraction
4. **Database Layer**: SQLAlchemy models, connection management

Benefits: Single responsibility per layer, downward-only dependencies, isolated business logic, testable components.

Figure 1 illustrates the complete four-layer architecture with all components and their interactions.

5 Building Block View

5.1 Level 1: System Overview

Component	Responsibility
API Layer	HTTP requests/responses, validation, formatting
Business Layer	Business logic, orchestration, calculations
Persistence Layer	Repository pattern, query abstraction
Database Layer	Data models, connections, ORM

5.2 Level 2: Key Components

API Layer (app/api/):

- routes/movies.py - Movie endpoints, embedded responses, average calculation
- routes/actors.py - Actor endpoints, fullName generation

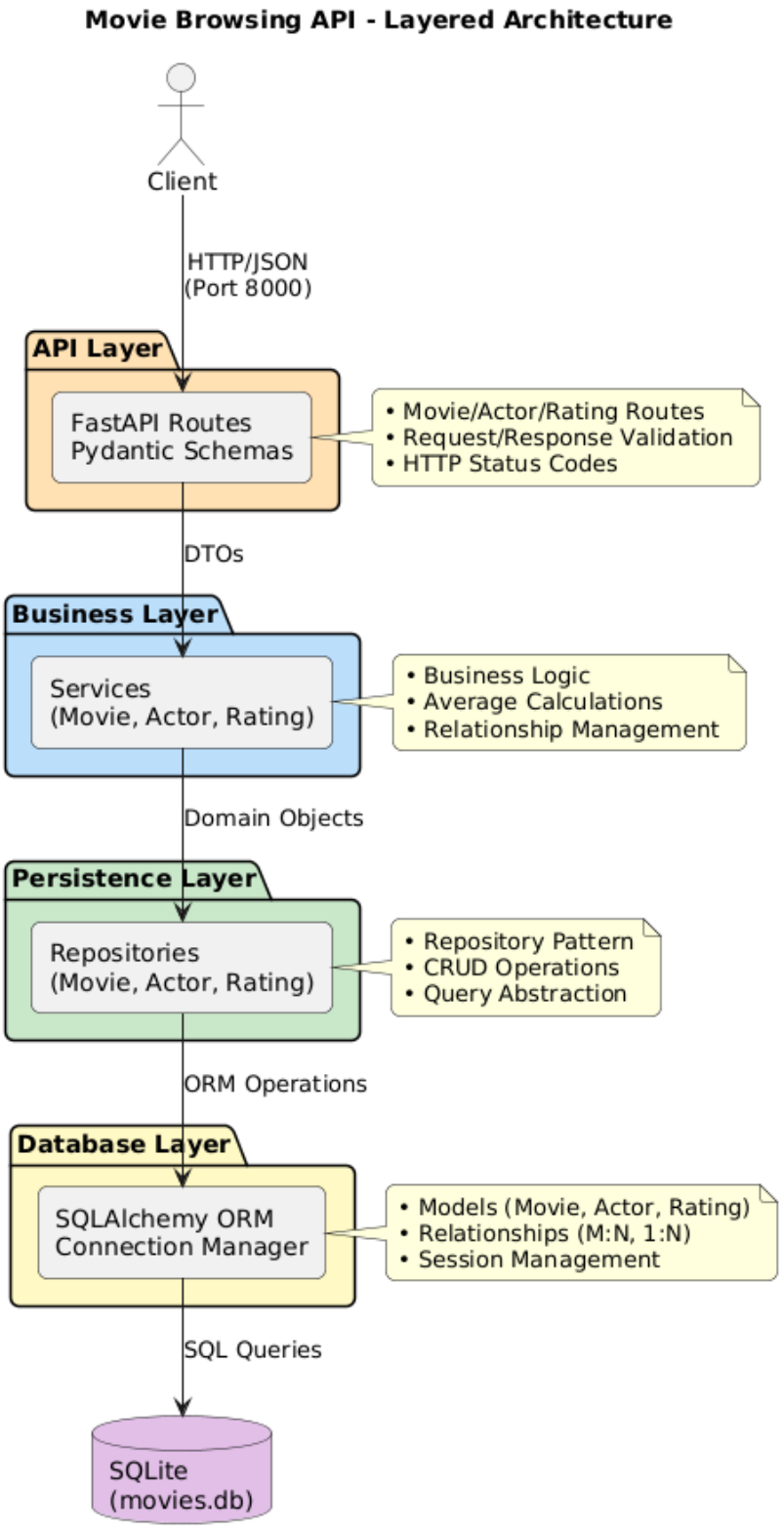

Figure 1: Four-Layer Architecture with Component Interactions

- routes/ratings.py - Rating endpoints, score validation (0-10)
- schemas.py - Pydantic models with aliasing

Business Layer (app/business/services.py):

- MovieService - CRUD, average rating calculation, actor relationship management
- ActorService - Actor lifecycle management
- RatingService - Rating CRUD with movie validation

Persistence Layer (app/persistence/repositories.py):

- MovieRepository, ActorRepository, RatingRepository
- Common: get_all(), get_by_id(), create(), update(), delete()

Database Layer (app/database/):

- models.py - Movie, Actor, Rating models; movie_actors association table
- connection.py - Engine, session factory, database initialization

6 Runtime View

6.1 Create Movie Scenario

1. POST /actors ($\times 2$) $\rightarrow$ Validate $\rightarrow$ ActorService $\rightarrow$ ActorRepository $\rightarrow$ 201 Created
2. POST /movies with actorIds $\rightarrow$ Validate $\rightarrow$ MovieService retrieves actors $\rightarrow$ MovieRepository creates movie + relationships $\rightarrow$ 201 Created with embedded actors
3. POST /ratings with movieId $\rightarrow$ Validate movie exists $\rightarrow$ RatingRepository creates $\rightarrow$ 201 Created
4. GET /movies/{id} $\rightarrow$ MovieService loads with relationships $\rightarrow$ Calculate average $\rightarrow$ 200 OK with embedded actors and ratings

Key Aspects: Layer separation maintained, dependency injection for sessions, embedded responses prevent N+1 queries, proper HTTP status codes throughout.

7 Deployment View

Container Structure:

```
Docker Container (python:3.12-slim)
 Uvicorn ASGI Server :8000
    FastAPI Application (4 layers)
 SQLite Database (movies.db)
```

Deployment Options:

1. **Dev Container**: VS Code, auto-populated DB, hot-reload, port forwarding
2. **Docker**: Build from Dockerfile, expose port 8000, run populate script

3. **Local**: Poetry virtualenv, manual DB init, uvicorn –reload

Artifacts: Dockerfile, pyproject.toml, app/ directory, scripts/populate_data.py, .devcontainer/devcontainer.json

8 Cross-cutting Concepts

8.1 Domain Model

Movie (Aggregate Root): Has many Ratings (cascade delete), has many Actors (independent), contains averageRating (calculated).

Actor (Independent): Associated with multiple Movies, independent lifecycle.

Rating (Value Object): Belongs to one Movie, deleted with parent.

8.2 Error Handling

200 OK (GET/PUT success), 201 Created (POST success), 204 No Content (DELETE success), 404 Not Found (resource missing), 422 Unprocessable Entity (validation failure), 500 Internal Server Error (unexpected errors).

8.3 Data Validation

Three layers: (1) Pydantic schema validation (types, formats, ranges), (2) Business validation (movie exists, actor IDs valid), (3) Database constraints (PK, FK, NOT NULL).

8.4 Data Transformation

API ↔ Business: camelCase ↔ snake_case via Pydantic aliases. Business ↔ Persistence: genre list ↔ CSV string. Database → API: eager-loaded relationships, calculated fields.

9 Architecture Decisions

9.1 ADR-001: Layered Architecture

Decision: Four-layer architecture (API, Business, Persistence, Database).

Rationale: Single responsibility, downward dependencies, testability, framework independence.

Consequences: (+) High maintainability, clear organization. (-) More boilerplate, transformations between layers.

9.2 ADR-002: SQLite Database

Decision: Use SQLite over PostgreSQL/MySQL.

Rationale: Zero config, file-based, suitable for MVP, easy reset/populate.

Consequences: (+) Simple deployment, fast dev. (-) Limited concurrency. Migration: SQLAlchemy enables easy PostgreSQL switch.

9.3 ADR-003: Embedded Responses

Decision: Return full embedded actors/ratings, not just IDs.

Rationale: Fewer client requests, complete data, better UX.

Consequences: (+) Better performance, simpler clients. (-) Larger payloads, not RMM Level 3.

9.4 ADR-004: Repository Pattern

Decision: Repository pattern for all data access.

Rationale: Encapsulates queries, abstraction over ORM, easy mocking.

Consequences: (+) Testability, maintainability. (-) Additional layer.

10 Quality Scenarios

PS-1 Performance: GET /movies (50 items) responds within 200ms with single DB query via eager loading.

MS-1 Maintainability: Switch SQLite to PostgreSQL by modifying only connection.py, <1 hour effort.

US-1 Usability: New developer accesses /docs, understands API, makes first successful call within 15 minutes.

RS-1 Reliability: POST /ratings with score=15 returns 422 with clear error, no DB write.

SS-1 Security: Malicious SQL in title field properly escaped by ORM, movie created safely.

11 Risks and Technical Debts

11.1 Risks

Priority	Risk	Mitigation
High	SQLite concurrency limits	Document as dev DB, PostgreSQL migration path
Medium	No authentication	Acceptable for MVP, JWT planned for v2
Medium	Large embedded responses	Monitor sizes, add pagination if needed

11.2 Technical Debts

Debt	Solution
No unit/integration tests	Implement pytest suite, httpx TestClient
No pagination	Add limit/offset parameters
No filtering/search	Query params for genre/year/actor
No CORS config	Add FastAPI CORS middleware
Hardcoded paths	Environment variables
No logging	Structured logging with levels
No health check	/health endpoint with DB check

12 Glossary

API Layer: HTTP handling via FastAPI routes and Pydantic schemas. **Business Layer**: Service classes with business logic. **Cascade Delete**: Auto-delete child records when parent deleted. **DTO**: Data Transfer Object (Pydantic models). **Eager Loading**: Load relationships in single query. **Embedded Response**: Full objects vs IDs in API responses. **MVP**: Minimum Viable Product. **N+1 Problem**: Separate query per related object. **ORM**: Object-Relational Mapping (SQLAlchemy). **Repository Pattern**: Data access encapsulation. **RMM Level 2**: Richardson Maturity Model - proper HTTP methods/status codes. **Session**: SQLAlchemy DB connection manager.