Movie Browser

CIT12

Timothy Stoltzner • Christopher Bouet

https://github.com/TimoSR/ruc-12-portfolio-project

Word count	**Student ID / Email**
x / 380 = x pages (text)	**stud-timothyr@ruc.dk** **cmhb@ruc.dk**

MSc in Computer Science

18. December 2025

Contents

Chapter 1

Application Design

1.1 Core Features

The application is designed to support three primary user needs:

1. **Exploration of movie data**: searching titles, browsing episodes, inspecting ratings and alternative names.
2. **Personal engagement**: rating movies, bookmarking items, tracking search history.
3. **Knowledge discovery**: analyzing relationships between actors, co-players, and genres.

To support these needs, the database is split into three cooperating schemas:

- `movie_db` $\rightarrow$ authoritative metadata about titles, people, ratings, episodes, and genres.
- `profile` $\rightarrow$ account-centered data such as search history, rating history, and bookmarks.
- `api` $\rightarrow$ a functional abstraction layer providing controlled access (functions, procedures, materialized views).

This design enforces a separation between user-generated data and static metadata. It allows parallel growth of both without coupling, and supports later database separation.

1.2 User Interface

User interaction centers on searching, browsing, and contributing ratings or bookmarks.

- **Dashboard**: Centralized search bar and navigation drawer, emphasizing bookmarks and search as most-used features.
- **Search Results**: Compact, scannable presentation (poster, title, year, short plot).
- **Detail Views**: Rich metadata with linked navigation between titles and people; actions to bookmark or rate directly.
- **User Profile**: Access to bookmarks and ratings, reflecting the account-centered design.
- **Bookmark and Rating History Views**: Dedicated displays with potential for future sorting/filtering.

These workflows mirror expectations from comparable platforms and directly map to database features.

1.3 Implications for the Data Model

Supporting account-centered features requires more than movie and person metadata. Users must persist their own state — bookmarks, searches, and ratings — which requires additional tables and clear separation from canonical film data.
Key principles:

- **API contracts**: Clients never access tables directly; they call API functions or views. This allows the underlying schema to evolve without breaking clients.
- **No cross-boundary mixing**: Each schema owns its data. Updates that affect both `profile` and `movie_db` are two API calls, orchestrated by the backend.
- **No joins across boundaries**: The backend combines data; the database does not.
- **Function evolution**: New versions (e.g., `api.add_rating_v2`) can be added while old ones remain for compatibility.

This adds some orchestration complexity in the backend but ensures clear ownership, maintainability, and resilience. The model avoids tight coupling and remains adaptable to future growth.

Chapter 2

Database

2.1 Database Rework

2.1.1 Observed Issues

Looking at the legacy schema, several recurring problems stand out:

- **Denormalized fields**: Lists of genres, directors, writers, professions, and "known for" titles are stored as comma-separated strings, which are hard to query and validate.
- **Weak typing**: IDs stored as fixed-length strings (`char(10)`), numerics embedded in text, and no constraints on common fields (years, seasons, episodes).
- **Inconsistent and opaque naming**: Identifiers such as `tconst`, `nconst`, and table names like `wi` rely on short aliases rather than descriptive terms. Column naming mixes camelCase and snake_case. These practices reduce readability, increase onboarding time, and make the schema harder to maintain.
- **Coupled enrichment**: `omdb_data` mixes attributes like awards, runtime, and ratings, some overlapping with existing tables, leading to redundancy.

2.1.2 Name Changes

A major issue in the legacy schema is the **naming convention**. Identifiers often use short aliases (`tconst`, `nconst`, `wi`) that are not self-explanatory. This reduces readability and requires external documentation to understand.
Problems observed:

- **Opaque identifiers** — `tconst` and `nconst` do not indicate titles or people.
- **Abbreviations** — short names like `wi` and fields such as `isAdult` mix casing and lack clarity.
- **Inconsistent conventions** — camelCase and snake_case appear in the same schema; column capitalization is inconsistent.

Redesign principles:

- **Primary keys:** always named `id`.
- **Foreign keys:** named after the referenced table (`title_id`, `person_id`).
- **Tables:** descriptive, singular, snake_case (`title`, `person`, `actor`, `crew`, `word_index`).
- **Booleans:** prefixed with `is_` (e.g., `is_adult`).
- **Timestamps:** suffixed with `_at` (e.g., `created_at`).

This consistent convention improves readability, makes queries self-explanatory, and reduces the risk of misinterpretation.

2.1.3 Table Redesign

The legacy schema stored relationships such as genres, directors, writers, professions, and "known for" titles as comma-separated strings. This introduced several problems:

- **Unqueryable values** — filtering required string parsing instead of joins.
- **No referential integrity** — no guarantee that referenced titles or people existed.
- **Redundant storage** — values repeated across rows, causing inconsistency.
- **Difficult maintenance** — updating a single attribute required rewriting multiple rows.

Normalization

The redesign applies **3rd Normal Form (3NF)** to eliminate repeating groups and ensure each fact is stored once. Complexity moves from string parsing to relational joins, which databases handle efficiently.

- **Join tables** replace string lists:
  - `movie_db.title_genre` links titles to genres.
  - `movie_db.title_director`, `movie_db.title_writer`, `movie_db.actor`, and `movie_db.crew` capture people-to-title relationships.
  - `movie_db.person_profession` and `movie_db.person_known_for` normalize person attributes.
- **Controlled vocabularies** define common categories (genres, professions) via reference tables or enums.
- **Consistent datatypes** ensure clarity and safety: UUIDs for identifiers, booleans prefixed with `is_`, timestamps suffixed with `_at`.

Key Analysis

In the legacy schema, `tconst` and `nconst` served as primary keys stored as `VARCHAR(20)` to align with IMDb/OMDb datasets and the custom word index (`wi`). While this simplified ingestion, it caused inefficiency in joins, lacked uniqueness guarantees, and tightly coupled the schema to external formats.

Redesign

- All entities use **UUIDs** as primary keys.
- **Foreign keys** enforce referential integrity across join tables.
- External identifiers (e.g., IMDb `legacy_id`) are retained only as metadata, not as keys.

UUIDs provide globally unique and system-controlled identifiers, independent of external systems. Foreign keys guarantee valid relationships, while legacy IDs remain available for interoperability. This improves **integrity, performance, and maintainability** without losing compatibility with external datasets.

2.2 Design

2.2.1 Movie Data Model

The Movie Data Model provides the structural foundation for representing titles, people who worked on those titles, and their relationships.

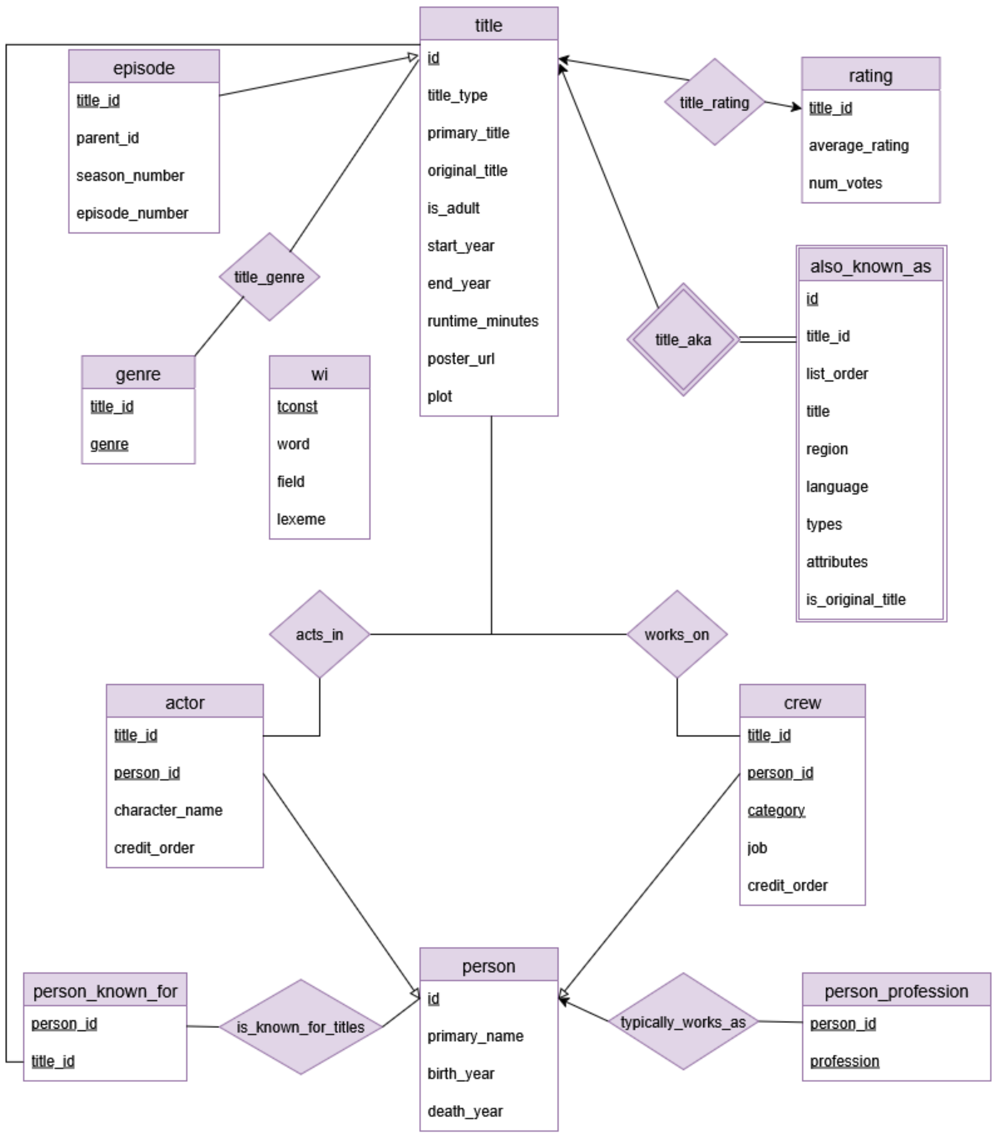


Figure 2.1: Movie Data Model — ER Diagram

Titles and Episodes

Titles are central to the model. Since each title has a unique plot and poster, these attributes are stored directly in the `title` table to simplify queries. Specialized cases such as episodes are treated as **partial specializations of titles**, with additional attributes (e.g., `parent_id`) stored in a dedicated table. This approach avoids overcomplicating the schema while retaining essential hierarchical information.

Genres

Genres are normalized into a separate table because titles can belong to multiple genres. The many-to-many relationship between titles and genres prevents redundancy and allows efficient retrieval of genre-based queries, which are frequent in user-driven searches.

Ratings

A dedicated rating table maintains the current average rating and number of votes per title. This ensures that aggregation is efficient and that global ratings remain synchronized with user contributions.

2.2.2 Alternative Titles (Also Known As)

The *also_known_as* entity is modeled as a weak entity dependent on titles, since it cannot exist on its own. Each alias must reference a title, ensuring total participation in the relationship. Constraints ensure consistency between the alias and the original title, preserving data integrity.

People and Professions

The person entity is normalized by splitting "known for" titles and professions into separate many-to-many relationships. This prevents storage of comma-separated lists and allows efficient querying.
A person may be associated with multiple titles and professions, while each title or profession may also relate to multiple people.

Cast and Crew

Actors and crew are modeled as **specializations of person**, since they require role-specific attributes. Actors are separated from crew to reflect their distinct relevance to users and to eliminate unnecessary null values (e.g., character fields that do not apply to crew).
Relationships to titles are represented as many-to-many (e.g., "acts_in" and "works_on"). Ordering information is preserved for actors to reflect their prominence in credits.
The original *title_crew* table from IMDB was excluded because its information is redundant with other sources (e.g., *title_principals*). This decision reduces duplication while retaining all necessary information.
We found the *category* and *job* attributes in the original *title_principals* to be too inconsistent in their naming to be mapped in a separate table. Ideally, a function could have been written to streamline this person metadata but we decided against it not to overcomplicate our design. Perhaps this is something that could be improved upon later in the project.

2.2.3 Framework Model

The Framework Model provides the structures necessary to support user-facing features and complements the Movie Data Model.

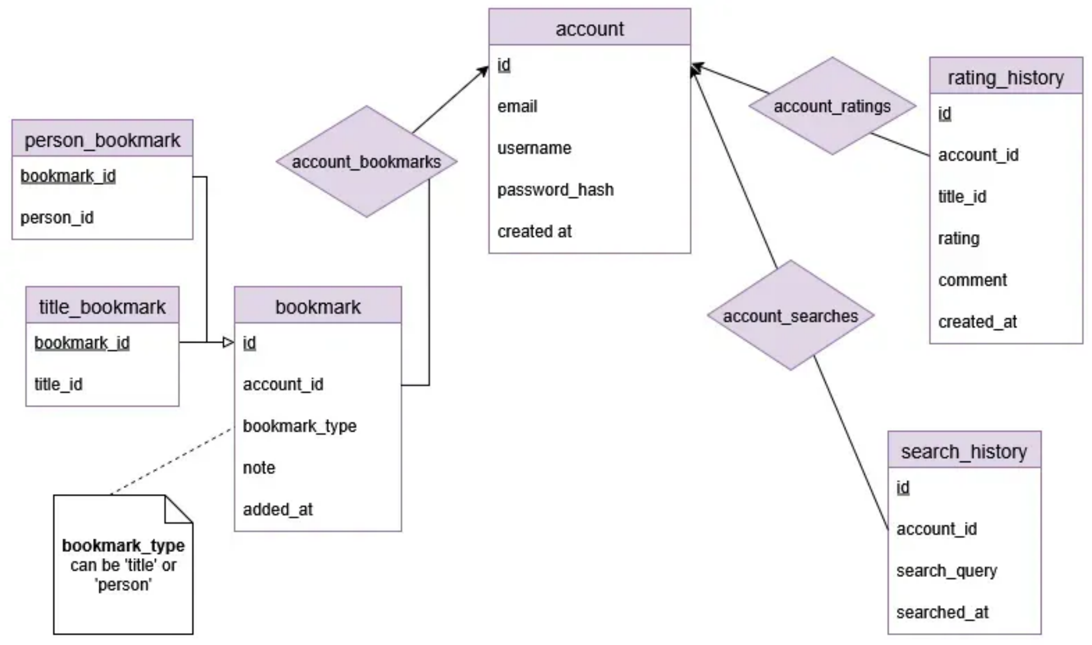


Figure 2.2: Framework Model — ER Diagram

Account-Centered Design

At the core of the Framework Model is the account entity. Most framework tables—`search_history`, `rating_history`, and `bookmark`—are linked to accounts in a many-to-one relationship, ensuring that user activity is consistently tracked.

Search History

The search history table logs all user queries with timestamps. This enables transparency for the user and establishes a reliable dataset for potential analytics or personalization features.

Rating History

Each rating action is stored in the rating history table with timestamps. Even if a user updates a rating, prior ratings should remain in the database. This approach preserves historical accuracy while ensuring that only the most recent rating contributes to the title's global average.

Bookmarks

The bookmark entity is generalized to accommodate both people and titles. Subtypes (`person_bookmark` and `title_bookmark`) store specific references. A type indicator ensures clarity while avoiding schema duplication.

2.2.4 Integration with the Movie Data Model

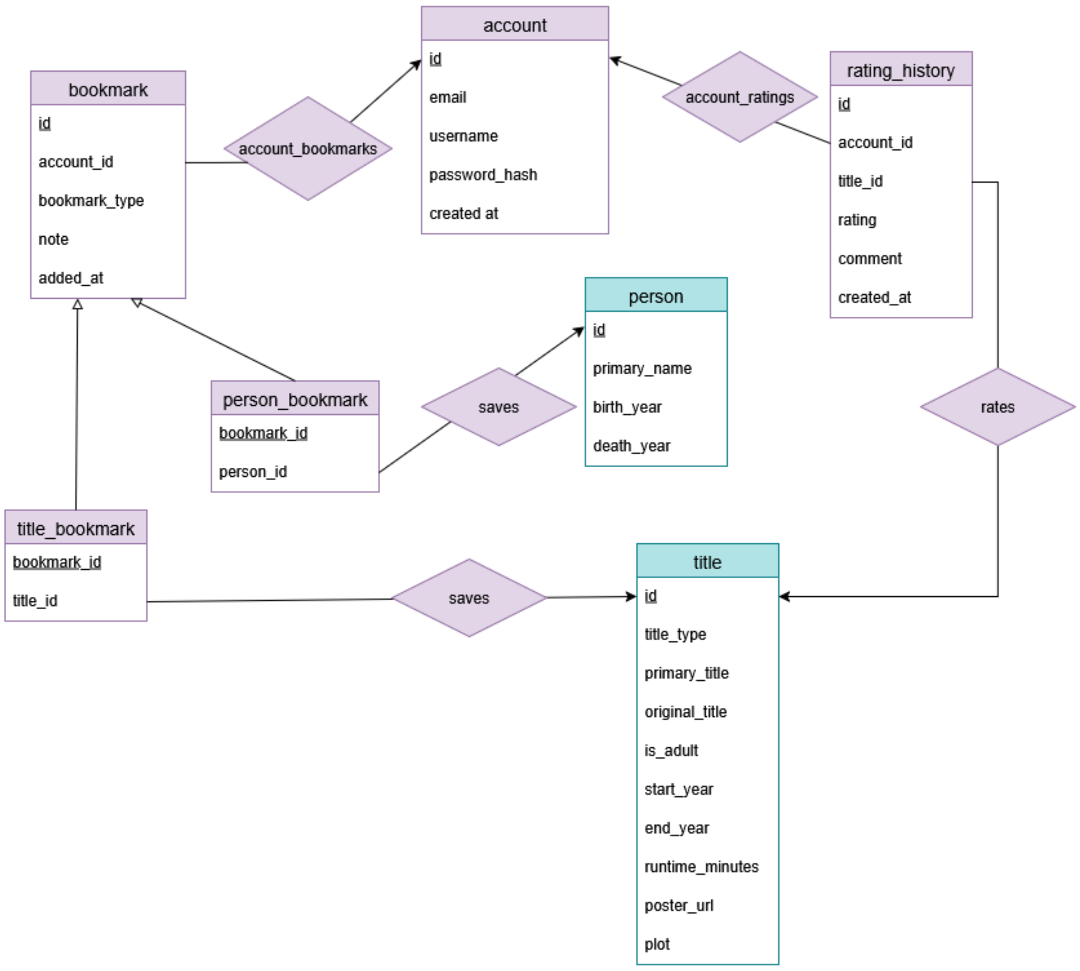


Figure 2.3: Framework Model integrated with the Movie Data Model — ER Diagram

Framework entities reference Movie Data Model entities through identifiers (`title_id`, `person_id`). These relationships are conceptual rather than enforced by strict foreign keys, preserving separation between the two models.

2.2.5 Reverse-engineered ERD

Movie_DB_Schema

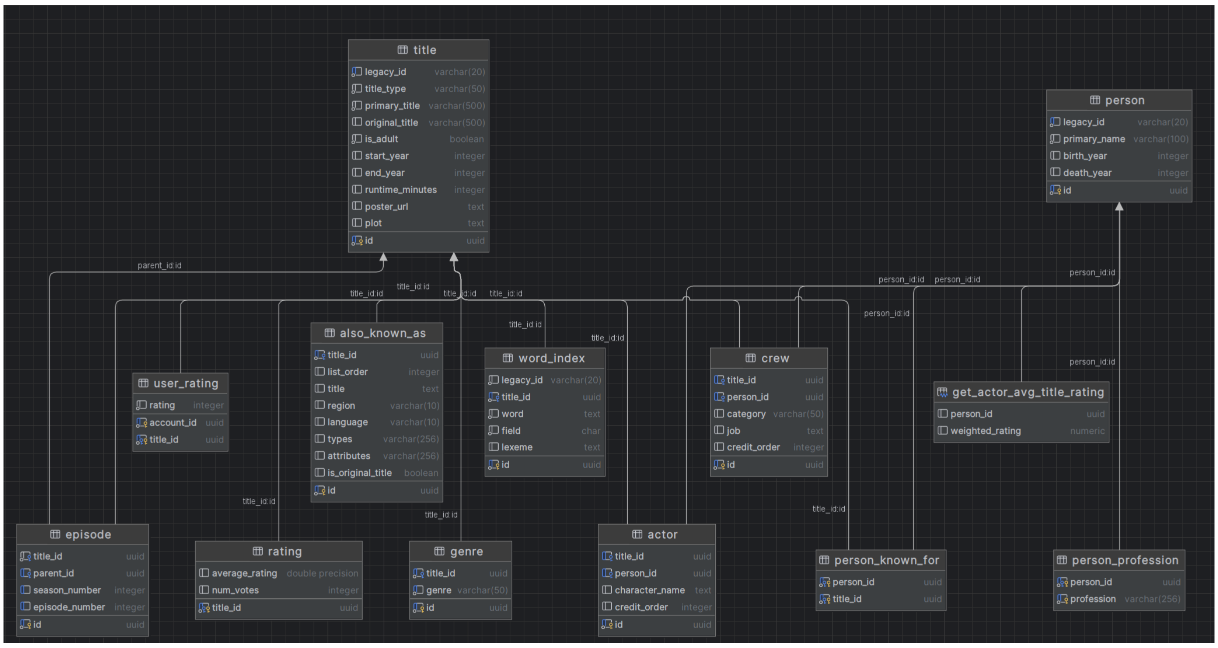


Figure 2.4: Reverse-engineered ERD — Movie_DB_Schema

Profile_Schema

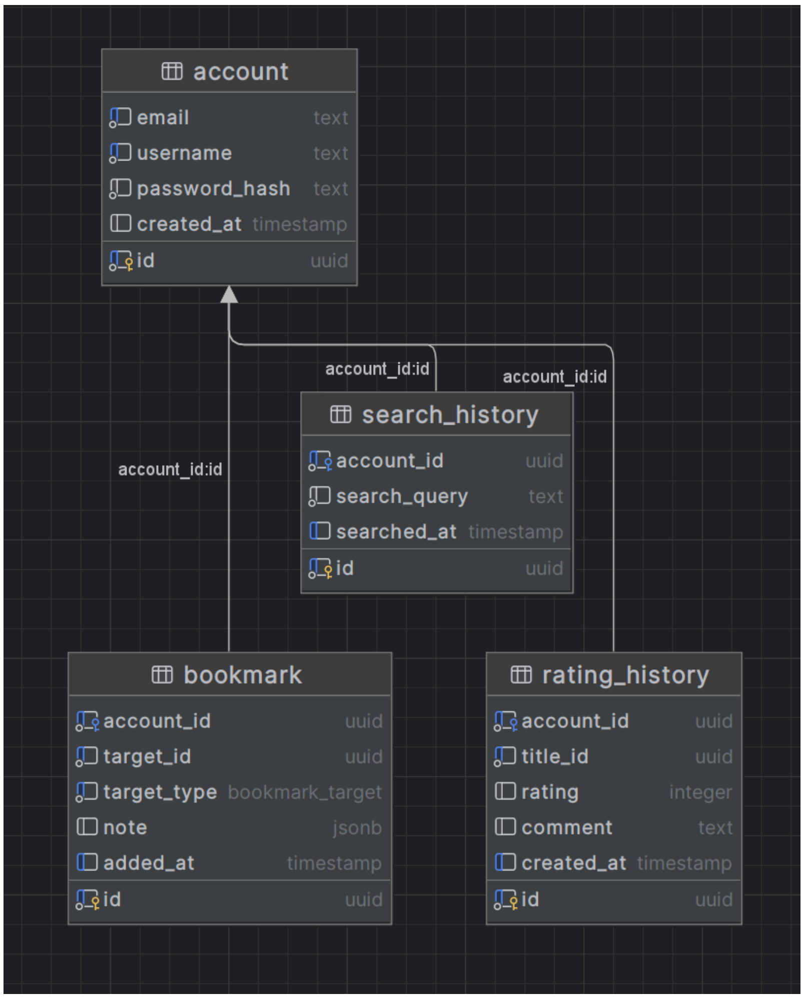


Figure 2.5: Reverse-engineered ERD — Profile_Schema

2.3 Implementation

2.3.1 Tables

The database is organized into two main schemas:

- `movie_db` — normalized metadata about titles, people, and their relationships.
- `profile` — user accounts, interactions, and personal activity history.

All entities use **UUID primary keys** for global uniqueness. IMDb identifiers are preserved as `legacy_id` for interoperability but are no longer primary keys.

2.3.2 `movie_db` schema (content metadata)

- `title` — Core table for all films, shows, and episodes. Stores identifiers, titles, adult flag, release years, runtime, poster, and plot.
- `user_rating` — One rating per `(account, title)`. Ensures each user has at most one active rating for a title. Used to calculate aggregates.
- `rating` — Maintains aggregated statistics: average rating and number of votes per title. Derived from `user_rating` for fast lookups.
- `genre` — Join table linking titles to one or more genres. Replaces comma-separated strings with one row per `(title, genre)`.
- `episode` — Represents episodic structure. Links episodes to parent series (`parent_id`) and records season/episode numbers.
- `also_known_as` — Alternative titles (regional, language-specific, or descriptive variations). Normalized from denormalized aliases.
- `person` — Stores people (actors, directors, writers, crew). Contains legacy IMDb IDs, names, and life dates.
- `person_known_for` — Many-to-many join between people and their "known for" titles.
- `person_profession` — Join table linking people to professions (e.g., "actor", "director"). Normalizes multi-valued legacy field.
- `crew` — Assigns people to titles with roles, categories, and credit order. Covers directors, writers, and crew not in the actor table.
- `actor` — Specialized table for cast members. Links titles to persons with character names and credit order.
- `word_index` — Inverted index for search and analytics. Stores tokenized words from titles/plots with optional lexemes. Used for keyword queries and co-occurrence analysis.

2.3.3 `profile` schema (user data)

- `account` — User account data with email, username, password hash, and creation timestamp.
- `bookmark` — Unified bookmark system. Each row links a user to either a title or a person, enforced with the `bookmark_target` enum. Optional JSON notes can be attached.
- `search_history` — Tracks user search queries with timestamps. Used for personalization and analytics.
- `rating_history` — Stores a historical record of user ratings with optional comments. Unlike `movie_db.user_rating`, this table preserves history for audit and UI purposes.

2.3.4 The API Schema

The `api` **schema** provides the interface layer between clients and the underlying database. It exposes functionality through a mix of stored procedures, functions, views, and materialized views.

- **Stored procedures** $\rightarrow$ safe, transactional operations for writes and deletes.
- **Functions** $\rightarrow$ parameterized queries and small building blocks that return sets or values; used where logic is read-focused but requires input arguments.
- **Views** $\rightarrow$ read-only abstractions that simplify access and hide schema joins.
- **Materialized views** $\rightarrow$ precomputed aggregations optimized for heavy queries.

This design separates concerns: **procedures** handle state changes, **functions** provide composable logic with parameters, **views** expose stable abstractions, and **materialized views** deliver precomputed performance.

Framework functionality

Provide account, bookmark, note, and history management as the foundation for later features.

- **Accounts.** `create_account`, `delete_account`, `get_account_info`, and `get_accounts` encapsulate lifecycle operations. Interfaces are deterministic: `create_account` returns the new UUID, updates/deletes return `void`.
- **Bookmarks.** A single table supports either `title` or `person` targets, enforced by an enum (`bookmark_target`). Functions add, remove, update notes, and list bookmarks with pagination.
- **Notes.** User-authored notes can be attached to titles or persons. CRUD functions upsert text and fetch notes with `updated_at`.
- **History.** Search and rating history functions return time-ordered, paginated results, joined with titles for readability.

Using one bookmark table avoids duplication and simplifies uniqueness constraints across `(account, target)`. Notes extend functionality without altering the core schema.

Search

- **Simple search.** `string_search_title(query)` performs full-text search using `to_tsvector` and `plainto_tsquery`, ranked with `ts_rank`. Trigram indexes provide substring fallback.
- **Structured search.** `structured_string_search(title_q, plot_q, char_q, person_q)` allows multi-field filtering with `ILIKE` conditions.

Ratings

- **Per-user ratings.** `add_user_title_rating(account_id, title_id, rate)` inserts or updates `movie_db.user_rating` and recalculates aggregates in `movie_db.rating`.
- **History.** `add_rating` writes to `profile.rating_history` for auditing and UI purposes.

Policy: one rating per `(account, title)`. Aggregates derive only from `user_rating`.

People and actors

- **Find persons.** `get_person_by_name(search_term)` returns `(id, name)` for autosuggest.
- **Co-players.** `get_coplayers(actor_name)` lists actors appearing together with frequency counts.
- **Name ratings.** `get_actor_avg_title_rating` (materialized view) stores weighted averages across an actor's filmography.
- **Popular co-actors.** `get_popular_co_actors(actor_name)` ranks collaborators by weighted rating, then name.

Similarity and tags

- **Similar movies.** `get_similar_movies(title_id, limit)` computes Jaccard similarity over genres.
- **Person words.** `person_words(name, max_words)` extracts the most frequent indexed words from a person's associated titles.
- **Word queries.** `exact_match_query`, `best_match_query`, and `word_to_words_query` provide keyword-based lookups and co-occurrence analysis using `movie_db.word_index`.

2.3.5 Stored Procedures

Stored procedures encapsulate transactional operations where multiple changes must succeed or fail together.
Examples:

- `api.create_account` inserts a new account and raises a notice with the generated UUID.
- `api.delete_account` deletes an account and validates existence.

Procedures are more suitable for side-effect-driven operations (insert, update, delete) and allow exception handling and notices.

2.3.6 Views

Views expose read-only abstractions that simplify client access.
Examples:

- `api.get_all_accounts` provides a unified account list.

Views behave like tables, can be queried flexibly, and avoid parameter boilerplate.

2.3.7 Materialized Views

Materialized views precompute expensive aggregations that are repeatedly queried.
Examples:

- `movie_db.get_actor_avg_title_rating` stores weighted averages of actor ratings.

Functions would recalculate results every call. Materialized views persist results on disk, can be indexed, and refreshed periodically, making them ideal for heavy analytics.

2.4 Data Migration

The migration script transfers data from the legacy IMDb-style tables into the redesigned `movie_db` schema while applying normalization, type corrections, and referential integrity.
Process:

1. **Word index** — Legacy `wi` entries are migrated into `movie_db.word_index`, mapping legacy `tconst` values to new UUID-based `title.id`. `TRIM` is applied to remove fixed-length `CHAR` padding.
2. **Titles** — Records from `title_basics` and extended attributes from `omdb_data` are merged into `movie_db.title`. Fields such as `start_year`, `end_year`, and `runtime_minutes` are cast to integers with `NULLIF` guards for missing values.
3. **Episodes** — Hierarchical relationships from `title_episode` are migrated into `movie_db.episode`, linking each episode to both its own title UUID and its parent series UUID.
4. **Genres** — The comma-separated `genres` field is split into multiple rows in `movie_db.genre` using `string_to_array`, ensuring each `(title, genre)` pair is normalized.
5. **Alternate titles** — Data from `title_akas` is moved into `movie_db.also_known_as`, preserving ordering, region, language, and flags.
6. **Ratings** — Ratings are transferred from `title_ratings` into `movie_db.rating` with integrity checks applied.
7. **Persons** — Records from `name_basics` are migrated into `movie_db.person`, normalizing `birth_year` and `death_year`.
8. **Known-for titles** — Comma-separated `knownForTitles` values are expanded into rows in `movie_db.person_known_for`, with each entry validated against existing `title.id`.
9. **Professions** — Multi-valued `primaryProfession` entries are normalized into `movie_db.person_profession`.
10. **Crew and actors** — `title_principals` is split into two tables:
    - Non-actors $\rightarrow$ `movie_db.crew`.
    - Actors, actresses, self-roles $\rightarrow$ `movie_db.actor`.

Post-migration steps:

- Legacy source tables (`title_basics`, `name_basics`, `title_ratings`, etc.) are dropped once data is confirmed migrated.
- Verification queries confirm row counts for titles, episodes, genres, ratings, persons, actors, and crew.

The migration process ensures:

- Legacy identifiers (`tconst`, `nconst`) are preserved as `legacy_id` attributes for interoperability.
- Relationships are converted from comma-separated lists to normalized join tables.
- Datatypes are corrected (e.g., years as `INT`, UUIDs as PKs).
- Referential integrity is enforced through foreign keys.

This step transforms the dataset from a bulk-ingestion format into a fully relational structure aligned with the new design, while preserving compatibility with legacy IDs where required.

2.5 Indexing

Efficient query performance requires careful use of indexes. In this design, indexes are added selectively to support common access patterns: search, joins, and time-ordered lookups. Both B-tree and GIN indexes are used, depending on the query type.

2.5.1 Title Search

- **Trigram indexes:**
  - `idx_title_primary_title_trgm` and `idx_title_plot_trgm` accelerate substring and `ILIKE` searches on titles and plots.
- **Full-text index:**
  - `idx_title_fulltext` supports language-aware searching via `to_tsvector`.
- **Use case:** enables fast ranked title lookups for both substring and semantic search.

2.5.2 User Ratings

- `idx_user_rating_account` optimizes queries by `account_id`, e.g. listing a user's ratings.

2.5.3 Genre

- `idx_genre_title` speeds joins from titles to their genres.
- `idx_genre_genre` accelerates filtering by genre.

2.5.4 Episodes

- `idx_episode_parent` optimizes lookups of episodes by series (`parent_id`).
- `idx_episode_season` supports navigation by season and episode numbers.

2.5.5 Alternate Titles

- `idx_aka_titleid` supports joins from titles to alternate names.
- `idx_aka_title_trgm` enables fast substring search on alternate titles.

2.5.6 People

- `idx_person_name_trgm` accelerates substring search on person names.
- `idx_person_known_for_title` and `idx_person_profession` support queries by known-for titles and profession categories.

2.5.7 Actor and Crew

- `idx_actor_title`, `idx_actor_person`, `idx_crew_title`, `idx_crew_person` speed joins between people and titles in both directions.

2.5.8 Word Index

- `idx_word_index_word` and `idx_word_index_title` support keyword lookups and joins with titles.
- `idx_word_index_word_trgm` (optional) provides fuzzy search on words.

2.5.9 Profile Schema

- **Bookmarks:** `idx_bookmark_account_type_added` enables fast lookups of a user's bookmarks, ordered by recency.
- **Search history:**
  - `idx_search_history_account_time` supports chronological history queries.
  - `idx_search_history_query_trgm` allows fast substring search on past queries.
- **Rating history:**
  - `idx_rating_history_title` links histories to titles.
  - `idx_rating_history_account_time` enables time-ordered user rating history retrieval.

2.6 Testing

Testing is automated using SQL scripts executed inside transactions. By enabling `ON_ERROR_STOP`, any failing statement aborts the test immediately. Wrapping each test script in a `BEGIN ...ROLLBACK` block ensures that test data does not persist in the database.
Two categories of tests are included:

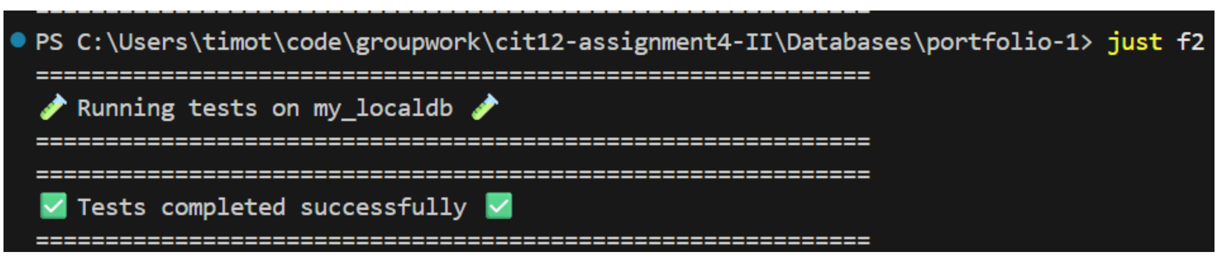
1. **Profile and movie database tests**

   These scripts validate core functionality of the `profile` and `movie_db` schemas, including constraints, inserts, and relationships.

2. **Migration tests**

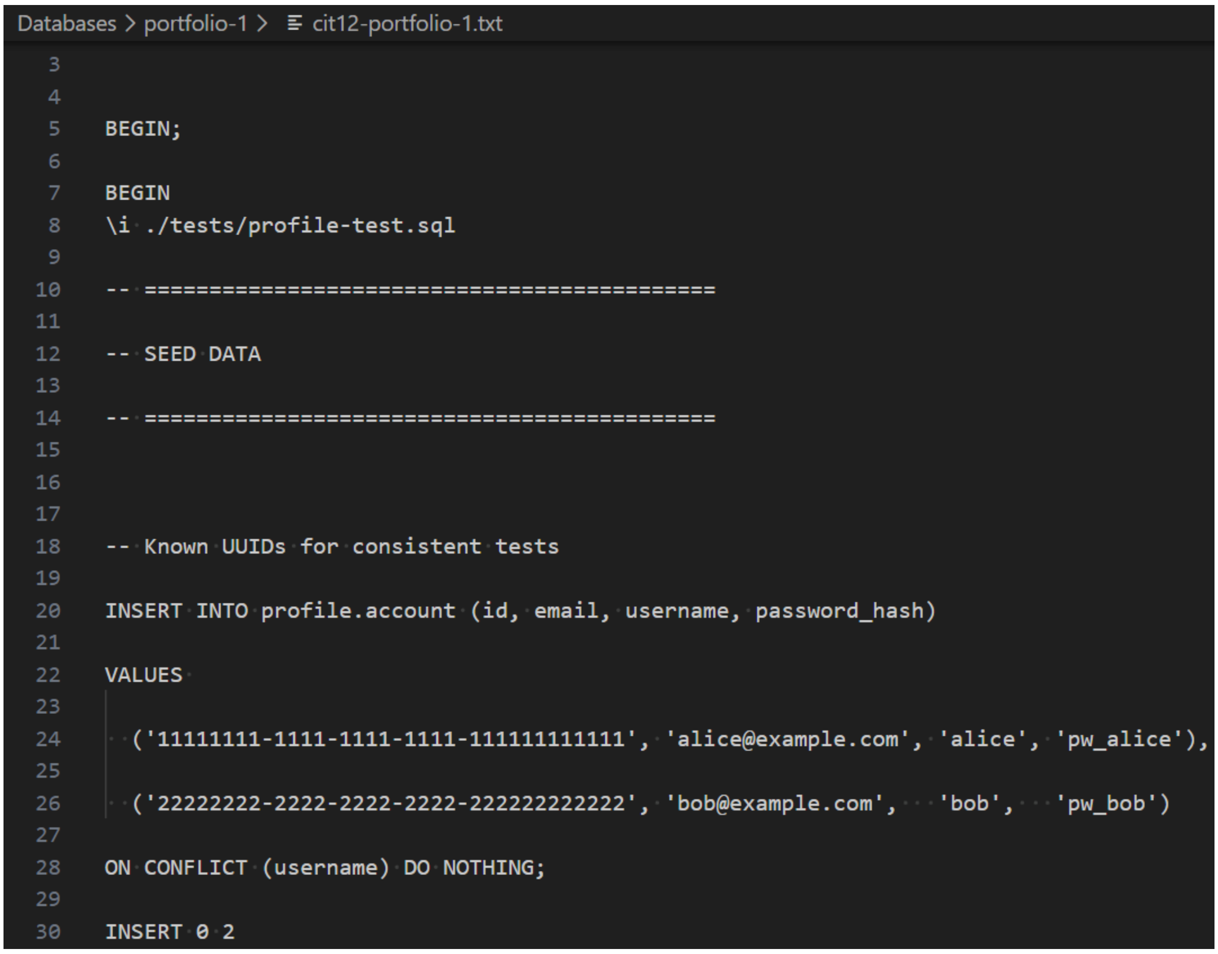
   This verifies that data migration scripts (from legacy schemas to the redesigned schema) preserve integrity and correctness.

The approach guarantees that the schema can be validated repeatedly without side effects, supporting regression testing after each migration.



```
● PS C:\Users\timot\code\groupwork\cit12-assignment4-II\Databases\portfolio-1> just f2
  ============================================================
  🧪 Running tests on my_localdb 🧪
  ============================================================
  ============================================================
  ✅ Tests completed successfully ✅
  ============================================================
```

Figure 2.6: db testing script with rollback, that gives a .txt test result


Databases > portfolio-1 > cit12-portfolio-1.txt

```


BEGIN;

BEGIN
\i ./tests/profile-test.sql

-- ============================================

-- SEED DATA

-- ============================================




-- Known UUIDs for consistent tests

INSERT INTO profile.account (id, email, username, password_hash)

VALUES

  ('11111111-1111-1111-1111-111111111111', 'alice@example.com', 'alice', 'pw_alice'),

  ('22222222-2222-2222-2222-222222222222', 'bob@example.com',   'bob',   'pw_bob')

ON CONFLICT (username) DO NOTHING;

INSERT 0 2
```

Figure 2.7: a small part of the bd test result output

2.7 Database Deployment

Deployment scripts focus on reproducibility and automation. All tasks are defined in a `justfile`, which provides a portable command runner for database operations. This ensures the same steps are executed consistently across developer machines and CI/CD environments.
The environment configuration is specified in `.env`:

```
DB_NAME=my_localdb
DB_USER=postgres
DB_HOST=localhost
DB_PORT=5432
PASSWORD=1234
GROUP_NUMBER=12
PROJECT_NAME=portfolio-1

DB_URL_ADMIN=
    postgresql://${DB_USER}:${PASSWORD}@${DB_HOST}:${DB_PORT}/postgres
DB_URL_TARGET=
    postgresql://${DB_USER}:${PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

Listing 2.1: ENV example

This setup allows the deployment scripts to use the same commands with different projects, groups, or hosts by adjusting only the environment variables.
The 'justfile' task `f5-restore-from-backup-no-drop` restores the database into a clean state using the latest backup file.

1. **Log start of restore** — prints clear markers to the console.
2. **Restore from backup file** — loads the most recent `backup_${DB_NAME}_latest.sql`.
3. **Exit on error** — each command checks `$LASTEXITCODE`, stopping immediately if a step fails.
4. **Log completion** — prints confirmation that the database has been restored.

The deployment process is **deterministic**: running the same task on the same backup file always produces the same result, ensuring reliable recovery, migration testing, and local developer setup.

Chapter 3

Backend

3.1 Application Documentation

3.1.1 Architecture Overview

The system requires a REST backend that exposes CRUD features with our existing PostgreSQL database. A clear separation of concerns is needed to keep read-heavy IMDB data isolated from user-owned data and to ensure predictable paging and discoverability across endpoints.
We Adopted a 4 layered architecture, API (Presentation), Application (Service), Domain, Infrastructure (persistence) with Domain driven design [Roba].

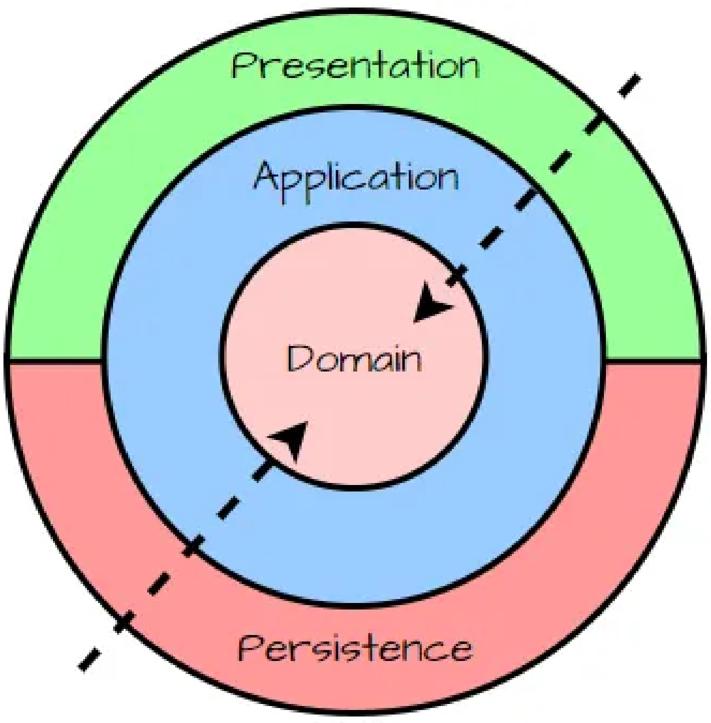


Figure 3.1: Architecture Overview [Robb]

The API layer provides versioned controllers with HATEOAS and pagination; the Application layer encapsulates use-cases and returns `Result<T>`; the Domain layer defines aggregates and contracts; the Infrastructure layer maps EF Core to the existing schema and provides repositories through Unit of Work.

- **API:** Versioned routes `/api/v{version}/...`, controllers for titles/persons, Swagger, and shared HATEOAS/paging utilities (`LinkDto`, `PagedResult<T>`, `UH.GeneratePaginatedUrl`, `ToPagedResult`) extentions for a cross system solution.
- **Application:** `TitleService`, `PersonService` expose queries/commands and return `Result<T>` of type `DTO` for explicit success/failure.
- **Domain:** Aggregates `Title`, `Person`, `Rating` with domain events; repositories defined via interfaces (e.g., `ITitleRepository`).

- **Infrastructure:** `MovieDbContext` maps to `movie_db` and `profile` without changing the schema; repositories and `UnitOfWork` implement persistence

The architecture keeps IMDB data read-only and profile data authoritative, enabling future expansion (e.g., additional controllers) without revisiting foundational choices. If a multi-database split becomes desirable, the repository/Unit of Work boundary and explicit mappings allow the context to be separated with minimal surface-area changes.

3.1.2 Key Design Decisions

1) Identifier Strategy (UUID + Legacy IMDB IDs)

- String IMDB IDs complicate type-safe APIs and cross-schema joins.
- Use UUIDs as primary keys; persist IMDB identifiers in `legacy_id` on `title` and `person`.
- All routes and services resolve by UUID; legacy lookups remain available for compatibility.
- Stronger referential integrity and EF compatibility, with preserved external linkage.
- The dual-identifier model supports analytics imports and external referencing without constraining internal typing.

2) HATEOAS + Pagination as Shared Infrastructure

- Inconsistent collection responses hinder client navigation and caching.
- Standardize on `PagedResult<T>` with {self, first, prev, next, last} links; default page size 20, maximum 100.
- `ToPagedResult` composes items, totals, and links from the current request context.
- Endpoints expose uniform pagination and discoverability, satisfying portfolio requirements 2-C.5/2-C.6.
- The same wrapper can be extended to remaining list endpoints (e.g., persons' auxiliary lists) for full coverage.

3) Repository + Unit of Work

- Direct EF usage in services couples business logic to persistence and weakens transaction control.
- Define repository interfaces per aggregate and coordinate them with a Unit of Work boundary.
- `IUnitOfWork` aggregates repositories and manages transactions for multi-entity workflows (e.g., ratings).
- Improves testability and ensures atomicity across tables.
- The boundary supports future cross-database orchestration without distributed transactions.

4) Result Pattern for Application Flow

- Exception-driven flow obscures expected outcomes and complicates HTTP mapping.
- Services return `Result<T>` with typed errors (e.g., `NotFound`, `Duplicate`).
- Controllers translate results to ProblemDetails with appropriate status codes.
- Error semantics are explicit and controller logic remains minimal.
- The pattern scales to additional endpoints without altering domain contracts.

5) EF Core with PostgreSQL Query Shaping

- Search-heavy endpoints require efficient projections and database-backed filtering.
- Use `AsNoTracking` and `Select` for read paths; employ PostgreSQL features (ILIKE, trigram/FTS) backed by existing indexes.
- Repositories return `(items, totalCount)` with stable ordering before `Skip/Take`.
- Queries remain performant and deterministic; tests acknowledge in-memory limitations and validate SQL behavior against Postgres.
- Additional read-models can be introduced without changing API contracts.

6) API Versioning

- The frontend requires a stable contract during iterative backend changes.
- Apply URL-segment versioning with `Microsoft.AspNetCore.Mvc.Versioning` and ApiExplorer.
- Routes follow `/api/v1/...`; Swagger groups by version; new breaking changes ship as new versions.
- Backward compatibility is maintained while enabling controlled evolution.
- The policy keeps migration costs predictable for the SPA as scope grows.

7) DTO/Entity Separation

- Exposing domain entities risks leaking invariants and internal evolution to clients.
- Publish dedicated DTOs (optional `Url` for self-links); keep entities internal to the domain.
- Controllers map domain results to DTOs; item self-links are set via route helpers.
- API contracts remain stable and HATEOAS-ready.
- DTOs allow additive fields (e.g., links, summaries) without affecting domain models.

8) Transactional Integrity for Multi-Entity Operations

- Account/title/rating flows span multiple aggregates and must be atomic.
- Enforce explicit transactions at the Unit of Work level.
- Services open a transaction boundary around coordinated writes, then commit or roll back.
- Operations preserve consistency without exposing transaction details to controllers.
- The approach is compatible with future outbox-based integration if multiple databases are introduced.

9) Database Reuse (No Migrations in This Subproject)

- Subproject 1 already provides a vetted schema and indexes.
- Map EF Core to existing schema objects; avoid schema mutations here.
- Schema-qualified mappings for `movie_db` and `profile`; materialized views and functions remain available for analytics.
- Responsibilities stay clean between database and backend; delivery risk is reduced.
- If domain needs change, migrations can be introduced in a future scope without disrupting current contracts.

3.2 Domain Boundaries

In the **domain boundaries**— Each bounded context defines its own aggregates (e.g., `Account`, `Title`, `Person`) and enforces local invariants. Interactions between contexts occur through the application layer, ensuring that each model remains clean, explicit, and purpose-driven.

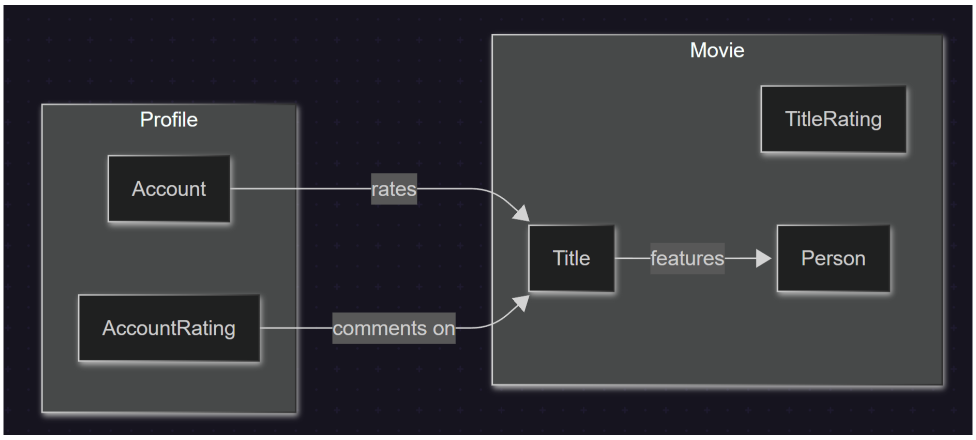


Figure 3.2: Domain Boundaries Overview

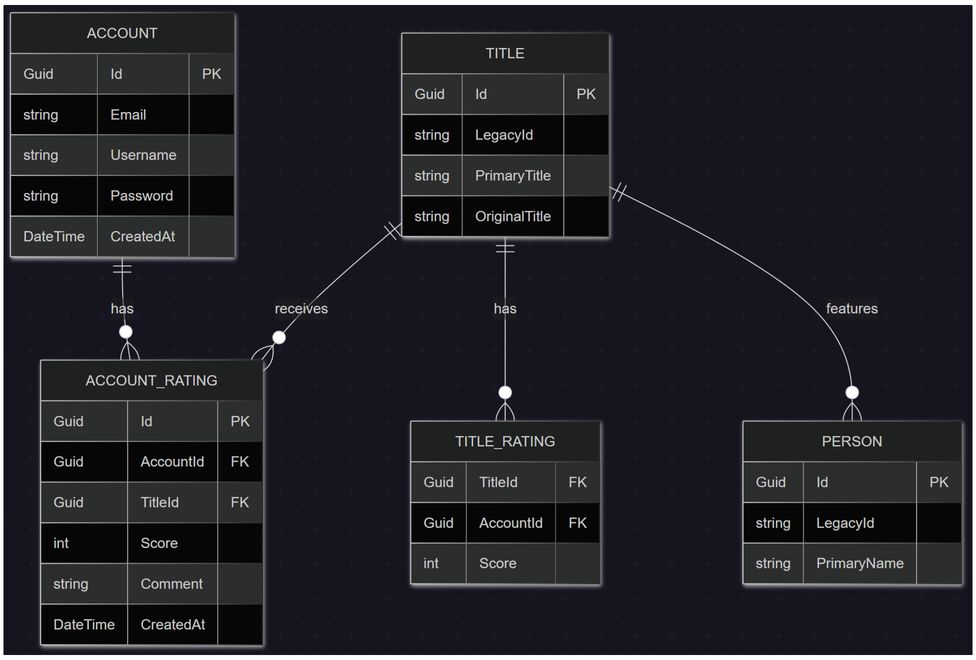


Figure 3.3: Context Map

3.2.1 Profile

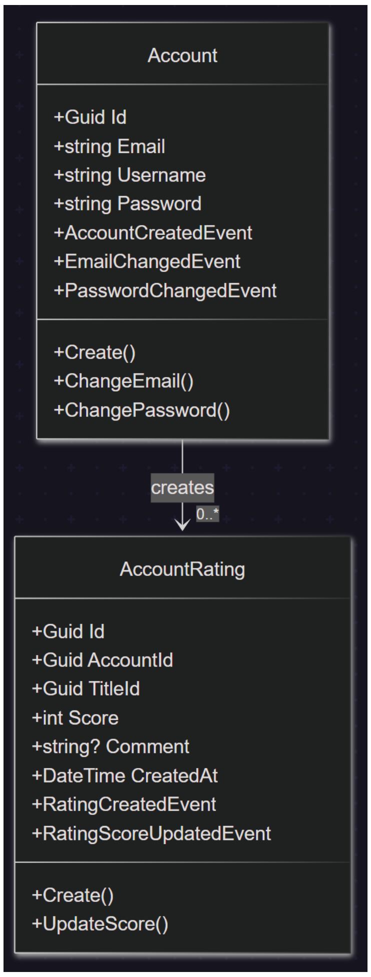


Figure 3.4: Profile Domain

Account

The `Account` aggregate encapsulates user identity, acting as the entry point for all profile-related interactions. It manages essential credentials including email, username, and password.
The entity is designed to enforce invariants through strict validation at creation and mutation points. It uses domain events (`AccountCreatedEvent`, `EmailChangedEvent`, etc.) to notify other bounded contexts or subsystems of critical state changes, adhering to eventual consistency principles. Interfaces (`IAccount`) support decoupling for dependency inversion.

- Constructor logic is internal or private to enforce controlled instantiation via the static `Create(...)` factory method.
- Email, username, and password are trimmed and validated to avoid state corruption.

- Mutations (email/password changes) check for identity before updating to prevent redundant events.

The class adheres to aggregate rules: changes are atomic, domain events are explicit, and side effects are not coupled to command handlers. Validation exceptions prevent illegal state. The entity avoids exposure of internal behavior and enforces encapsulation.
The domain could be extended with richer policies (e.g., email verification status, password complexity checks). Password hashing is assumed to occur elsewhere (e.g., in an application or infrastructure service), which should be documented.

Account Ratings

The `AccountRating` aggregate models an account's qualitative feedback on a specific title, with optional commentary.
Designed for expressive user interaction, it captures both quantitative (score) and qualitative (comment) input. Domain events (`RatingCreatedEvent`, `RatingScoreUpdatedEvent`) represent the lifecycle of the rating. It enforces business rules: score must be within [1–10], and comments (if present) must be concise ($\leq$ 500 chars).

- Ratings are instantiated via the `Create(...)` method to centralize validation.
- Entity ensures immutability of keys (AccountId, TitleId) post-construction.
- Comment and score updates raise events and use UTC timestamps for consistency.

The design allows the system to respond reactively to rating changes (e.g., updating aggregate movie score). The model keeps temporal context via `CreatedAt`.
Future enhancements may include versioning for edits, audit trails, or sentiment analysis integration.

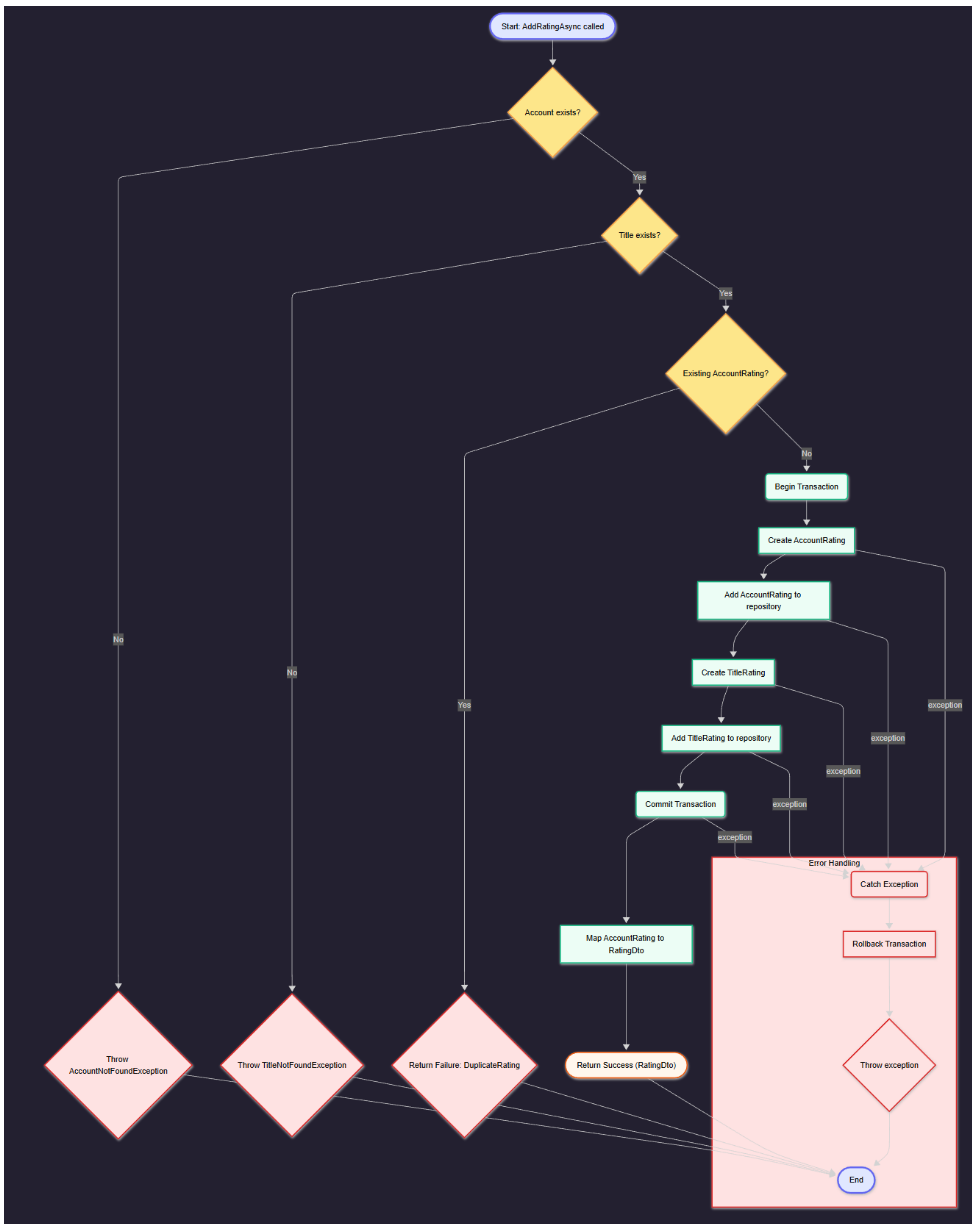


Figure 3.5: Account Rating Aggregate

3.2.2 Movie

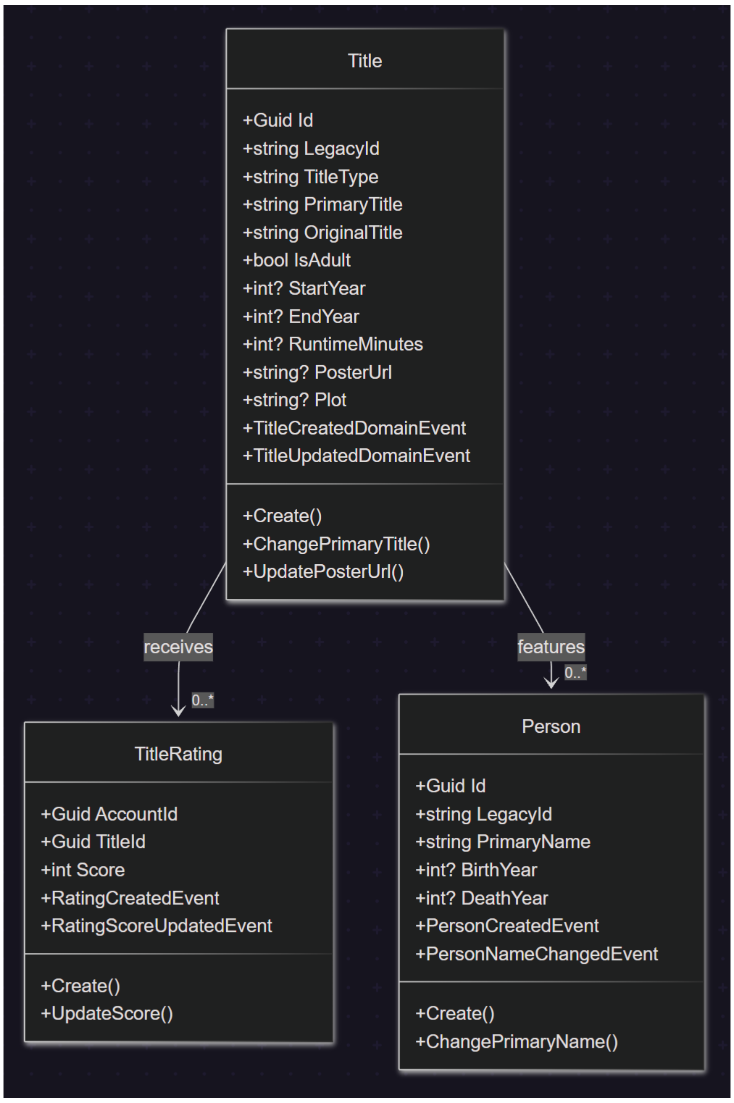


Figure 3.6: Movie Domain

Title

The `Title` aggregate models a movie, series, or other media entry. It aggregates metadata such as identifiers, titles, release years, adult flag, runtime, poster, and plot.
Defaults are handled defensively: original title falls back to primary title, `IsAdult` defaults to `false`, and missing years are defaulted safely. Domain events (`TitleCreatedDomainEvent`, `TitleUpdatedDomai`
track lifecycle changes. A static `GenerateLegacyId()` method ensures legacy compatibility.

- Factory method enforces the creation contract.

- Multiple update methods (for runtime, titles, poster, plot, etc.) provide fine-grained mutability.
- Redundant changes are skipped via value comparison.

The entity adheres to DDD principles: internal state is private, updates are intentional, and events reflect significant state changes.
Dynamic title updates may warrant event versioning. Consideration for localization or multi-language support could be factored in future domain expansions.

Title Ratings

The `TitleRating` aggregate captures a user's numerical evaluation of a title, typically rendered as a score between 1 and 10.
This model intentionally avoids commentary, distinguishing itself from `AccountRating`. It tightly binds `AccountId` and `TitleId` to a specific score, with all updates triggering domain events.

- The `Create(...)` method encapsulates input validation and event emission.
- Mutating the score uses an `UpdateScore(...)` method with event dispatch.
- Duplicate updates are avoided by early exits.

The model is purpose-built for performance and simplicity in scoring aggregation use cases. It is lightweight and sufficient for statistical computation.
Could be extended to support weighted ratings or timestamped rating history. Event consumers may aggregate or normalize data asynchronously.

Person

The `Person` aggregate represents a real-world individual associated with a title (e.g., actor, director). Key attributes include legacy identifiers, names, and birth/death years.
Creation and mutation logic follow invariants: identifiers and names must be valid and trimmed. Domain events communicate identity establishment and name updates.

- Static `Create(...)` method enforces required fields and raises a `PersonCreatedEvent`.
- The `ChangePrimaryName(...)` method ensures idempotency and avoids unnecessary updates.
- Domain-specific exceptions (`InvalidLegacyIdException`, `InvalidPrimaryNameException`) clarify intent.

The class preserves historical data while allowing non-destructive updates. Validation at boundaries prevents invalid records from entering the domain.
Additional roles or aliases may be introduced via value objects or child entities to enrich the representation.

3.3 Data Access Layer Implementation

3.3.1 Data Access Overview

The backend reads IMDB-sourced data and manages user-owned records with low latency and strict ownership boundaries. Search endpoints are read-heavy and optimized around fast projections, while write paths guarantee transactional integrity across user-focused tables.
A single EF Core `MovieDbContext` is used, mapped to PostgreSQL schemas `movie_db` and `profile`. Schema-qualified mappings and existing `legacy_id` columns are preserved. Reads are optimized using lightweight projections and `AsNoTracking`, and search queries are delegated to PostgreSQL features—functions, views, and indexes—whenever they yield better performance or translation than LINQ.
Entities are mapped 1:1 to existing tables. No migrations are introduced. All paginated queries enforce deterministic ordering before applying `Skip/Take` and return `(items, totalCount)` for API pagination. Search operations rely on `EF.Functions.ILike`, trigram or full-text indexes, and `api` schema functions where appropriate.
This setup delivers predictable paging, minimal tracking overhead, and efficient, index-driven filtering. Backwards compatibility is maintained via `legacy_id` lookups without sacrificing UUID primary keys.
If query complexity grows, dedicated read models, materialized views, or compiled EF Core queries can be introduced without changing API contracts or the underlying schema.

3.3.2 Repository Pattern

Using Entity Framework directly inside application services can easily lead to leaky abstractions, unclear transaction boundaries, and reduced testability. To avoid this, keep EF concerns isolated within the infrastructure layer and let application services depend only on well-defined repository and query interfaces.
Each aggregate should have a narrow repository interface, while read-specific needs should be handled by separate query interfaces. For example:

- `ITitleRepository` – `GetByIdAsync`, `GetByLegacyIdAsync`, and `SearchAsync(query, page, pageSize)` returning `(IEnumerable<Title> items, int totalCount)`.
- `IPersonRepository` – standard CRUD operations plus `ExistsByLegacyIdAsync`.
- `IPersonQueriesRepository` – read-only access with `AsNoTracking` projections, `ILike` filtering, ordered and paged results, and delegation to SQL functions for scenarios like co-actors, "known for" lists, or popularity rankings.
- `IRatingRepository` – per-account reads and concurrency-safe upserts with reliable lookup strategies.

With this separation, application services remain EF-agnostic and express use cases through commands and queries that return DTOs or domain entities where appropriate. This design makes unit testing easier using mocks or fakes, enforces cleaner boundaries, and keeps services focused solely on business logic and orchestration.

When new endpoints or use cases emerge, additional repositories or query handlers can be added without modifying existing contracts, keeping the system more maintainable and open to extension.

3.3.3 Unit of Work

Multi-entity operations—such as rating flows that modify account, title, and rating data—must execute atomically to maintain consistency. This requires explicit transaction boundaries and lifecycle management.
To achieve this, transaction control is centralized in an `IUnitOfWork`, which groups repositories and exposes methods like `BeginTransactionAsync`, `CommitTransactionAsync`, `RollbackTransactionAsync`, and `SaveChangesAsync`. Each application service defines a single transactional scope per use case.
Application services access repositories through the `IUnitOfWork`, perform coordinated changes across entities, and commit once at the end of the workflow. If an exception occurs, the transaction is rolled back. Read operations typically skip transactions unless snapshot-level consistency is required.
This approach ensures clear commit points, unified error handling, and predictable resource usage while preventing partial writes across aggregates.
Additionally, this transaction boundary becomes a foundation for future needs such as cross-context coordination, and it naturally supports patterns like outbox/inbox when integrating with external systems.

3.3.4 Portability to Multi-Database Deployment

Future separation of `profile` and `movie_db` into independent databases should not impact application logic or API contracts. To ensure this, all persistence concerns remain hidden behind repository interfaces and EF mappings that use schema-qualified configurations.
The current single `MovieDbContext` can be replaced with multiple database contexts—for example, `ProfileDbContext` and `MovieDbContext`—without touching domain services or controllers. Only connection strings and dependency injection registrations need to change. Each repository is bound to its specific context, and any workflow that spans multiple databases is orchestrated at the application service layer, without relying on distributed transactions.
Where eventual consistency is acceptable, an outbox pattern can be introduced to coordinate cross-database operations through reliable messaging.
This design keeps refactoring contained to composition and wiring rather than domain logic, minimizing migration risk and avoiding downtime.
If strict consistency across databases becomes a future requirement, the existing Unit of Work boundary can evolve into a saga-style coordinator. This preserves application service interfaces while swapping only the orchestration strategy.

3.4 Web Service Layer

3.4.1 Implementation Overview

The service layer must expose stable, discoverable endpoints for titles and persons, while shielding use-case logic from transport and persistence layers. Error behavior must be explicit and consistently translated to HTTP responses.
This is achieved through thin controllers (`PersonsController`, `TitlesController`) that delegate all behavior to application services. Controllers handle only request/response translation. All IO uses DTOs; domain entities stay internal to the application. Application services return `Result<T>` objects with strongly typed errors, which controllers translate into HTTP responses using `ProblemDetails`.
Cross-cutting conventions are applied uniformly:

- **API Versioning** via URL segments (`/api/v{version}/[controller]`)
- **HATEOAS** links for discoverability
- **Consistent pagination structures**
- **OpenAPI/Swagger** for contract visibility and documentation

DTOs can optionally include a `Url` field to expose their own self-link. Shared abstractions like `LinkDto`, `PagedResult<T>`, and response helpers ensure consistency across controllers.
This design keeps controllers small, testable, and free from business or persistence logic. Error handling remains uniform across endpoints, and the resulting API is self-describing and easy to navigate.
New resources can be introduced simply by adding new services and DTOs—without modifying existing endpoints, controller logic, or error policies—allowing the API surface to scale cleanly over time.

3.4.2 API Contract

Clients need predictable identifiers, media types, and response formats to efficiently support search, retrieval, and navigation at scale.
To achieve this, the API uses UUIDs as primary identifiers in resource URLs, while still exposing IMDB identifiers as `legacyId` for alternate lookup scenarios. JSON is the sole media type for all requests and responses, and collections follow a consistent wrapper shape that includes data, pagination metadata, and navigation links.
Routing conventions:

- `/api/v1/titles/{id}` – primary lookup by UUID
- `/api/v1/titles/{legacyId}` – alternate lookup by IMDB ID
- `/api/v1/titles?query=...` – searching or filtering

(Same pattern applies for persons.)
Media type:

- `application/json` for all endpoints.

Collection format:

- `PagedResult<T>` containing:
- `items`, `page`, `pageSize`, `totalItems`, `totalPages`, and `links`.

HTTP status codes:

- `200` – successful reads
- `201` – resource created, with `Location` header
- `204` – successful idempotent write with no body
- `400`, `404`, `409`, `500` – errors serialized as RFC 7807 `ProblemDetails`

This contract is explicit, cache-friendly, and versionable. Clients can traverse the API using links rather than relying on out-of-band knowledge.
Future changes are additive—new fields or new versions—without breaking existing clients. IMDB-based lookups remain supported without compromising type safety or identifier consistency.

3.4.3 HATEOAS

Clients benefit from first-class navigation links to avoid manually assembling URLs and to enable cursorless pagination. This reduces coupling to route structures and improves interoperability across clients.
Each item DTO includes an optional `Url` (self-link), populated using route helpers in controllers. Collections implement canonical pagination links—`self`, `first`, `prev`, `next`, and `last`—generated centrally to avoid duplication.
These links are emitted through a `PagedResult<T>`, which is built using a `ToPagedResult(...)` helper that derives link values from the current request path and query parameters.
As a result, responses become self-descriptive: clients can follow links to related resources or additional pages without reconstructing URLs or relying on external documentation.
The same linking strategy extends cleanly to secondary collections such as “known-for” titles or co-actors, enforcing consistent navigation semantics across the entire API surface.

3.4.4 Pagination

Search endpoints are high-volume and must guarantee consistent paging and stable ordering to ensure both correctness and a good user experience.

Paging Rules

- `page` must be ≥ 1
- Default `pageSize = 20`
- Maximum `pageSize = 100`
- Repository methods return `(items, totalCount)`
- Controllers wrap results in `PagedResult<T>` and add HATEOAS navigation links

Query Behavior

- Apply deterministic ordering before `Skip`/`Take`
  - e.g., sort by normalized title or person name
- Clients supply paging via `page` and `pageSize` query parameters
- Response shape always includes:
  - `items`
  - `page`, `pageSize`
  - `totalItems`, `totalPages`
  - HATEOAS pagination links (`self`, `first`, `prev`, `next`, `last`)

This guarantees that clients receive stable, predictable slices with accurate item counts. The approach is efficient, cache-friendly, and easy to reason about.
If demand grows or keyset-based performance is needed, cursor-based pagination can be introduced alongside page/size parameters without breaking existing consumers.

3.4.5 API Versioning

The frontend requires backward compatibility as endpoints evolve. To support this, the API uses URL-segment versioning with `Microsoft.AspNetCore.Mvc.Versioning`, integrated with `ApiExplorer` so each version is exposed in OpenAPI.
Routes follow the pattern `/api/v1/...`, and controllers or individual actions are annotated with their supported versions. When a breaking change is required, a new version such as `v2` is introduced, while older versions remain accessible during a deprecation period.
This makes version boundaries explicit and allows the SPA to upgrade specific endpoints independently rather than rewriting the entire API surface at once.
The approach enables parallel development of new capabilities while maintaining stability for existing consumers.

3.5 Testing

The system spans domain logic, Unit Of Work persistence, and HTTP endpoints, so testing must isolate responsibilities while still verifying that layers interact correctly.
To achieve this, the test strategy mirrors the architecture:

- **Domain and application tests** focus purely on logic, without any I/O or EF Core dependencies.
- **Infrastructure tests** run against a real PostgreSQL instance to validate repositories, EF Core mappings, and transactional behavior.
- **API and controller tests** exercise HTTP endpoints directly, including scripted flows, and assert status codes, DTO shapes, pagination metadata, and RFC 7807 `ProblemDetails`.

Test projects follow the structure of the production code. Shared fixtures supply deterministic data and object builders, enabling consistent setup across tests. Assertions emphasize both behavior and contract accuracy.
Because each test targets a clearly defined boundary, failures are easy to trace to a specific layer, reducing diagnosis time and avoiding false positives caused by excessive mocking or incorrect fakes.
The test suite can be extended over time with SPA contract tests and performance baselines for high-traffic endpoints such as search.

3.6 Reflections and Discussion

Several architectural and design considerations emerged that were not explicitly addressed in previous sections. One of them concerns future support for multiple databases. Although the current solution operates on a single `DbContext`, the use of repositories, dependency injection, and schema-qualified EF Core mappings ensures that the application layer remains unaffected if the user-owned `profile` schema and IMDB-derived `movie_db` schema are later separated into distinct databases. Should workflows span across both, the outbox pattern would be a suitable mechanism for ensuring reliable messaging and eventual consistency.
HATEOAS has only been partially implemented. Search endpoints already expose navigation links and pagination metadata, whereas several list endpoints, particularly those related to persons, accounts, and ratings, still return raw collections. This creates an inconsistent client experience. Extending the existing link and pagination components to all list endpoints would provide a uniform contract and improve overall discoverability.
Another important design decision is the treatment of IMDB-derived data as strictly read-only. The system deliberately avoids exposing modification routes for these entities. All user-generated content, including rating history and account information, is instead stored exclusively in the `profile` schema. If additional metadata or annotations become necessary in the future, they should be modelled as auxiliary tables rather than altering the imported IMDB structures.
Related to this is the distinction between transactional and analytical data. Individual user interactions, such as ratings, are persisted in `profile.rating_history`, while aggregated rating data in `movie_db.rating` serves statistical and reporting purposes. This separation preserves data provenance and supports auditability, while still enabling analytical queries. If real-time aggregates become necessary, asynchronous projections could be introduced without reworking the core write model.
From a collaboration perspective, the use of feature branches, small and reviewable pull requests, and clearly owned segments of the domain helped reduce coupling and refactoring conflicts. DTO conventions and explicit route versioning acted as the shared interface across teams. One improvement would be the introduction of a lightweight API changelog to complement Swagger and make version transitions more predictable.
Several lessons emerged during development. Test instability often stemmed from database-specific assertions being used in general unit tests; these were moved to isolated integration tests. Pagination initially produced inconsistent results due to missing ordering prior to `Skip` and `Take` operations; enforcing explicit ordering and centralizing URL generation resolved these issues.
Looking ahead, three areas require focus: completing HATEOAS and pagination across all endpoints, implementing authentication and authorization using JWT with a secure-by-default model, and improving operational performance for search-intensive interactions through compiled queries, caching static resources, and ongoing monitoring of query behaviour. These steps would strengthen the system’s robustness, security, and consistency as it evolves.

3.7 Conclusion

Subproject 2 successfully delivers a maintainable, versioned REST backend that cleanly separates responsibilities, aligns with domain-driven design, and reuses Subproject 1's PostgreSQL schemas without schema mutations.
The backend now provides stable, evolvable API contracts with HATEOAS navigation, consistent pagination, EF Core mapping to existing schemas, clear transaction boundaries through repositories and Unit of Work, and a strong Result/DTO-based application flow.
This architecture enables the backend to scale functionally without introducing hidden coupling between layers or leaking persistence logic into business workflows. It also ensures that future teams can extend the system—either by adding new endpoints, bounded contexts, or database sources—without refactoring foundational components.
The system is in a mature architectural state and ready for future extensions like authentication, full HATEOAS coverage, and multi-database separation.
The implementation meets the core portfolio requirements by providing:

- **HATEOAS-compliant endpoints (2-C.5)** using self-links and paginated navigational links.
- **Consistent, reusable pagination infrastructure (2-C.6)** shared across controllers.
- **Stable versioned contracts (`/api/v1/...`)** that isolate frontend-breaking changes.
- **Explicit application flows using** `Result<T>` for predictable HTTP translation and error handling.
- **Efficient search and projections** using EF Core with PostgreSQL features like ILIKE and trigram/FTS indexes.
- **Comprehensive testing at all relevant boundaries** (unit, integration, controller, and `.http`/Swagger validation).

3.7.1 Ready for Future Enhancements

- Completing HATEOAS + pagination coverage across all remaining list endpoints.
- Introducing authentication/authorization (JWT, roles, secure defaults).
- Scaling toward a multi-database setup by splitting `movie_db` and `profile` without impacting application logic.
- Optimizing performance further using compiled queries, caching, and read models.

Chapter 4

Frontend

4.1 User Interface Design

4.1.1 Site Structure

The application is structured as a Single-Page Application (SPA) with two primary layout branches: the Authentication Layout (for login/registration) and the Main Application Layout (which includes the global navigation bar).
The user navigates through the application primarily via the top navigation bar, which provides access to the Home, Movies, and Persons sections. Detailed views for movies and persons are accessed by clicking on items within the list views or search results.

Site Structure Diagram

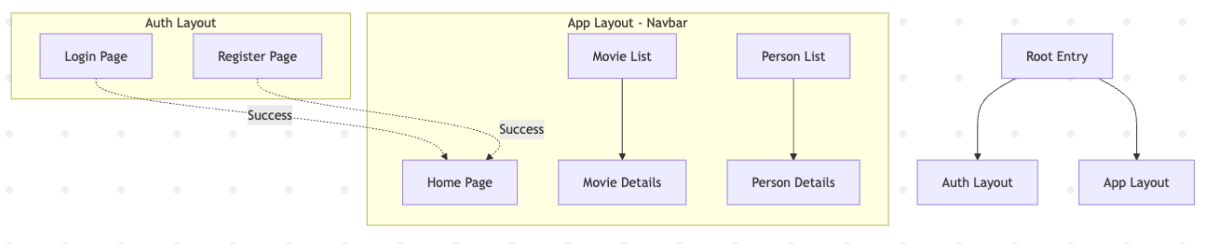


Figure 4.1: Site Structure Diagram

4.1.2 Wireframes

The user interface is designed with a focus on content and following Jakob Nielsen principles [Jak]. The core layout consists of:

1. **Global Navigation Bar:** Fixed at the top, containing links to major sections (Home, Movies, People) and a global search bar.

2. **Content Area:** The main view area below the navigation bar where pages are rendered.

3. **Cards & Grids:** Lists of movies and people are presented in responsive grids using card components to display key information (images, titles, ratings) efficiently.

Navbar

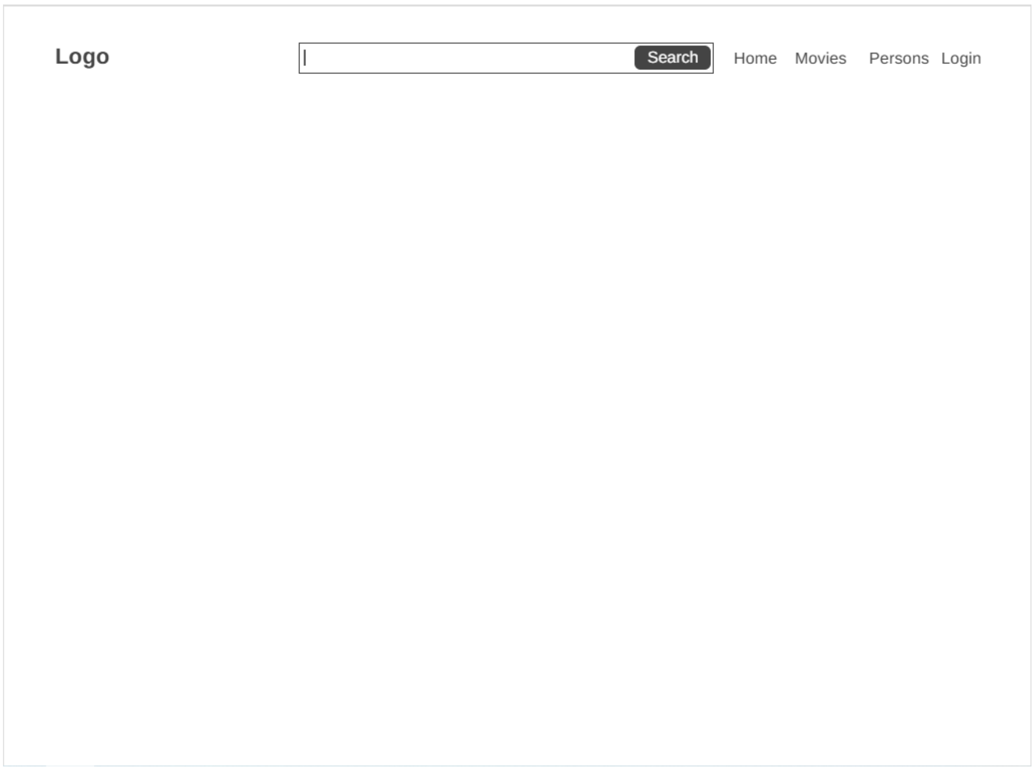


Figure 4.2: Navbar Wireframe

Content Area

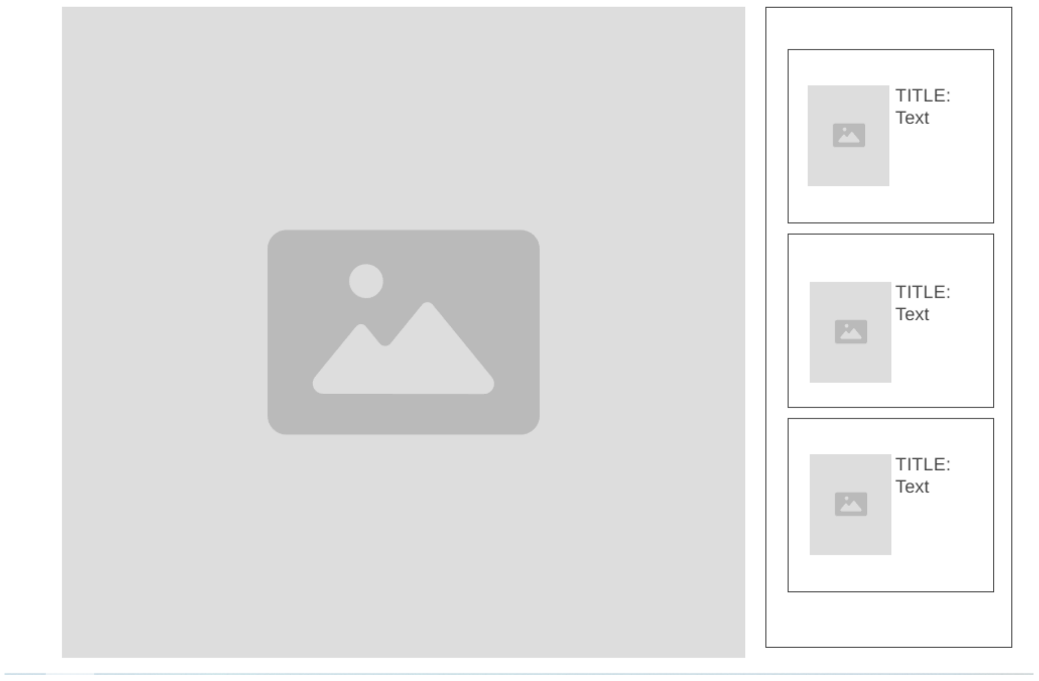


Figure 4.3: Content Area Wireframe

Cards/Grids

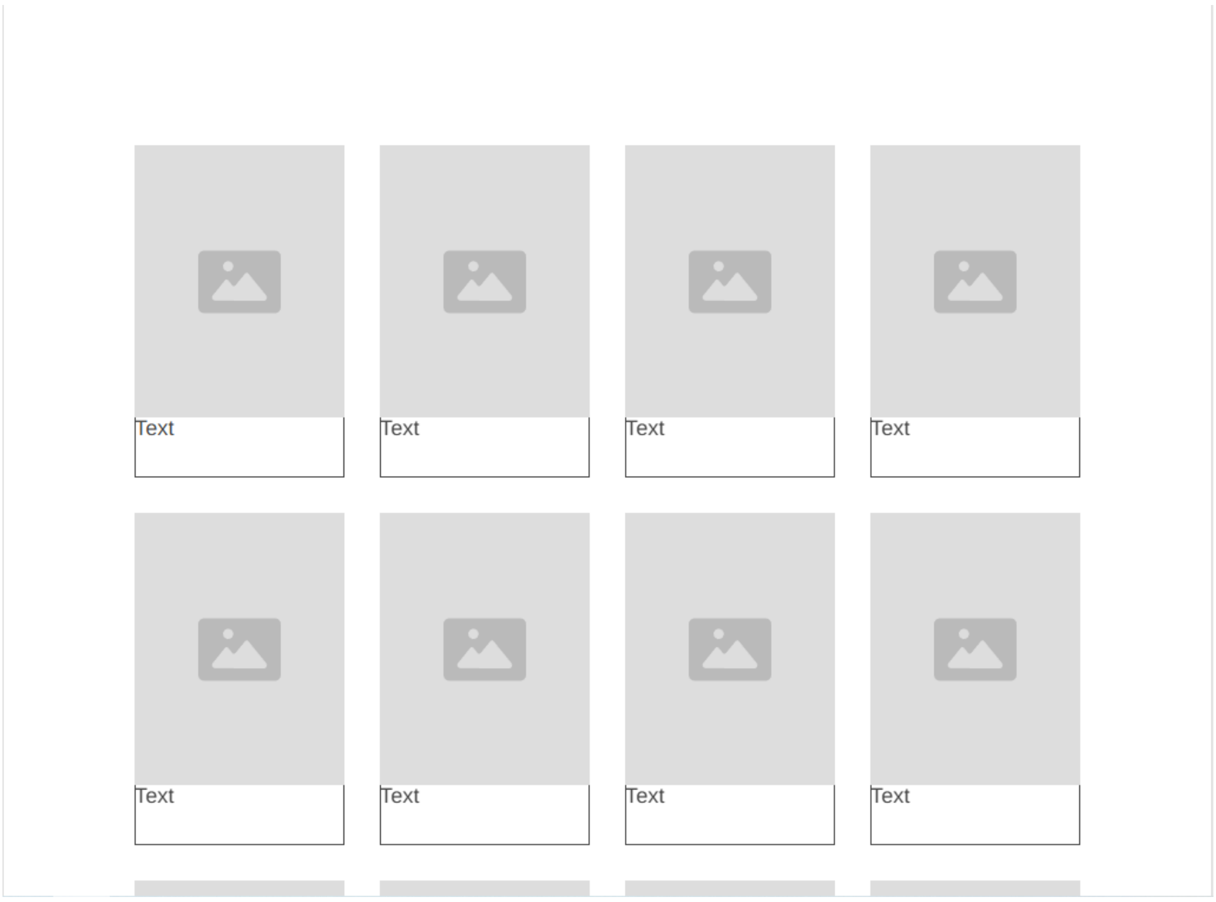


Figure 4.4: Cards and Grids Wireframe

Actor Details

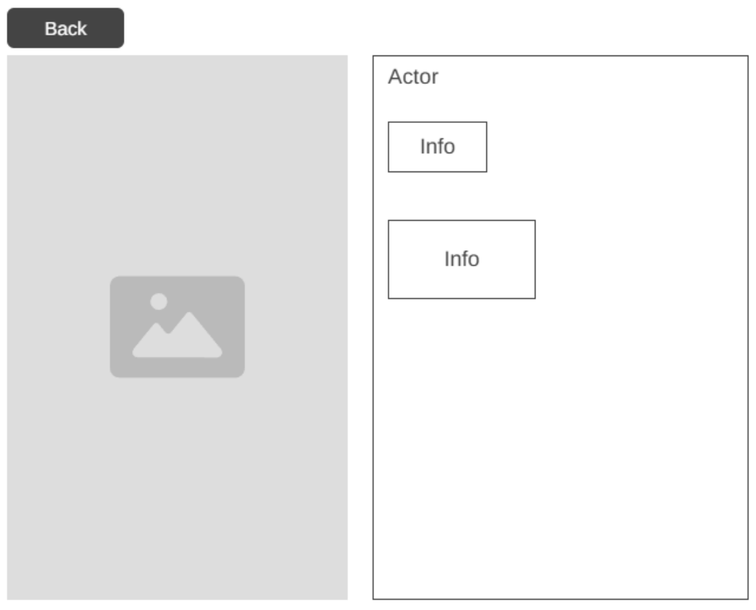


Figure 4.5: Actor Details Wireframe

4.1.3 Routes

The application uses TanStack Router for client-side routing. The route configuration is defined in `src/routeTree.tsx` and supports the following paths:

- `/` : `HomeView` - The landing page of the application.
- `/login` : `LoginView` - User authentication page.
- `/register` : `RegisterView` - User registration page.
- `/movies` : `MovieListView` - Displays a paginated grid of movies.
- `/movies/$movieId` : `MovieDetailsView` - Detailed view for a specific movie (dynamic route).
- `/persons` : `PersonListView` - Displays a paginated grid of actors/persons.
- `/persons/$personId` : `PersonDetailsView` - Detailed view for a specific person (dynamic route).

4.1.4 Usability Heuristics

We designed the user interface to comply with **Jakob Nielsen's 10 Usability Heuristics**. Key examples include:

1. **Visibility of System Status:** The application provides immediate feedback for all asynchronous actions. For example, `MovieListView` and `PersonDetailsView` display loading spinners (controlled by `store.isLoading`) while data is being fetched, ensuring the user knows the system is working.
2. **Match Between System and the Real World:** We use familiar terminology such as "Movies", "Actors", and "Search" throughout the interface, rather than technical database terms like "tconst" or "nconst". This ensures the application speaks the user's language.
3. **User Control and Freedom:** Users can easily navigate back from detailed views. The `PersonDetailsView` includes a dedicated "Back to Actors" button, allowing users to reverse their navigation without relying solely on the browser's back button.
4. **Error Prevention:** The `AuthStore` implements client-side validation (e.g., checking for empty email/password) before sending requests to the backend. This prevents "bad" requests and provides immediate, constructive feedback to the user.

4.2 Presentation Layer

4.2.1 React Components

The presentation layer is built using React with a component-based architecture. The application is organized into feature-based modules (e.g., movies, persons, navigation), ensuring separation of concerns and maintainability.

App Component (App.tsx)

The root component serves as the entry point for the application. It must initialize global providers (routing, data fetching) to ensure all child components have access to necessary context.
We utilized the Provider Pattern to wrap the application with `QueryClientProvider` (for TanStack Query) and `RouterProvider` (for TanStack Router). This ensures that routing and caching logic are available globally.
The `App` component initializes a `QueryClient` instance and a router instance with a type-safe

4.3 Business Logic Layer

4.3.1 Data Representation (Internal)

The Business Logic Layer is implemented using MobX Stores. These stores act as the "single source of truth" for the frontend application, managing state, handling business rules, and orchestrating data fetching. This architecture decouples the React components (Presentation Layer) from the raw API responses and backend logic.

Movie Store (MovieStore.ts)

The application needs to manage a list of movies, including pagination state (current page, total pages) and search filters. The state must be reactive so that the UI updates automatically when data changes.
We implemented a Store Pattern using a class-based approach with MobX. The store encapsulates the state (movies, isLoading, error) and exposes actions (loadMovies, nextPage) to modify that state.
The `MovieStore` class uses `makeAutoObservable` to automatically make all properties observable and methods actions.

- **State:** Holds movies array, currentPage, and searchQuery.
- **Actions:** `loadMovies()` fetches data from the API and updates the state inside a `runInAction` block to ensure transactional updates.

The store successfully manages the movie list state. The separation of `loadMovies` logic allows it to be reused by pagination methods (`nextPage`, `previousPage`), reducing code duplication.
By keeping the business logic in the store, the `MovieListView` component remains thin and focused solely on rendering. This makes the logic easier to unit test independent of the UI.

Person Store (PersonStore.ts)

Similar to movies, the application requires managing a list of actors/persons with pagination.
The design mirrors the `MovieStore` for consistency. It implements an interface `IPersonStore` to define the contract, which aids in dependency injection and testing.
The store manages actors data. It currently uses a mock implementation (`fetchActors`) to simulate API calls, which allows frontend development to proceed in parallel with backend API development.
The store handles the asynchronous nature of data fetching correctly, setting `isLoading` flags to control UI loading states.
The use of an interface (`IPersonStore`) is a strong design choice. It allows us to easily swap the mock implementation with a real API service later without changing the components that consume the store.

Search Store (SearchStore.ts)

The application requires a global search feature that provides instant feedback (live search) but avoids overwhelming the server with requests for every keystroke.

We implemented a Debounce Pattern within the store. The store manages the raw query input and the debounced search execution.

- `searchDebounced(delayMs)`: Sets a timer (e.g., 300ms) that clears any pending search before scheduling a new one.
- `searchNow()`: The actual async action that performs the API request.
- **State:** Manages `isSearching` to show a spinner in the search bar while the request is in flight.

The debouncing logic effectively reduces API calls. The `cancelPendingSearch` method ensures that race conditions (where an old request finishes after a new one) are minimized.
Encapsulating the debounce logic inside the store is superior to handling it in the React component (e.g., `useEffect`). It keeps the component code clean and makes the search logic reusable across different components (e.g., Navbar search vs. a dedicated Search page).

4.3.2 Helper Functions

The Business Logic Layer also includes helper functions and utilities to support data manipulation.

- **runInAction (MobX):** Used extensively within stores to group state updates. This ensures that multiple state changes (e.g., setting movies, currentPage, and isLoading = false) trigger only a single re-render of the UI, improving performance.
- **URLSearchParams:** Used within the stores to construct query strings for API requests. This ensures that parameters are correctly encoded and handles edge cases like empty queries gracefully.

4.4 Data Access Layer

4.4.1 Data Access Implementation

The Data Access Layer (DAL) is responsible for all communication with the backend API. Ideally, the Presentation Layer should not know about the specifics of HTTP requests (URLs, headers, parsing). To achieve this, we implemented a Service Layer pattern using a combination of a base API client and TanStack Query options.

API Client (client.ts)

The application needs a consistent way to make HTTP requests to the backend, handling base URLs and common error states (e.g., non-200 responses) centrally.
We implemented a Facade Pattern with a simple `apiClient` function. This function abstracts the underlying fetch API, providing a clean interface for the rest of the application. It automatically prepends the `BASE_URL` (configured via environment variables) and throws standardized errors.
The client uses the native fetch API. This lightweight implementation avoids the overhead of larger libraries like Axios while meeting our requirements.
The client successfully centralizes the request logic. Changing the backend URL or adding global headers (e.g., for auth tokens in the future) can be done in this single file.
The current implementation is simple and effective. However, it currently only supports GET requests implicitly (via fetch defaults). Future extensions would need to support methods for full POST/PUT/DELETE CRUD functionality.

Query Options (queries/*.ts)

We need to integrate the API client with TanStack Query to handle caching, loading states, and deduplication.
We adopted the Query Options Factory pattern recommended by TanStack Query. Instead of writing `useQuery` hooks directly in components with inline fetch functions, we define reusable `queryOptions` objects.
Files like `movieQueries.ts` and `personQueries.ts` export functions that return the configuration object. This decouples the definition of the data dependency from the consumption of it.
This pattern allows us to use the same query logic in multiple places:

1. Inside components via `useSuspenseQuery`.
2. Inside the router for preloading via `queryClient.ensureQueryData`.

This is a highly scalable approach. It ensures that cache keys are consistent across the application, preventing bugs where different components might cache the same data under different keys.

4.4.2 TMDB Integration (Images)

The project requirements specify that actor images must be fetched from **The Movie Database (TMDB)** using the `nconst` (IMDb ID) stored in our local database. This is because our local database does not store image binaries or URLs.
The plan is to implement a **Proxy Service** or a direct client-side fetch to TMDB. Given the frontend-heavy architecture, a client-side fetch is the most straightforward approach for this subproject. The flow is:

1. Frontend receives `nconst` (e.g., "nm0000158") from our backend.
2. Frontend calls TMDB `/find/{nconst}` endpoint to get the TMDB-specific ID.
3. Frontend uses the TMDB ID to construct the image URL

Current Status: This feature is currently **planned but not fully implemented** in the provided codebase. The `PersonDetailsView` currently uses a placeholder icon. To implement this, we would create a `tmdbClient.ts` similar to our backend `client.ts`, but configured with the TMDB base URL and API key.
The absence of this feature means the UI lacks visual richness for actor profiles. However, the architecture (separate `PersonDetailsView` and `PersonCard`) is ready to accept image URLs as soon as the data source is connected.
Implementing this will require handling an external API key securely (likely via environment variables `VITE_TMDB_API_KEY`). We must also be mindful of TMDB's rate limits.

4.5 Functional Requirements

4.5.1 Single-Page Application (SPA)

The application must be a Single-Page Application where navigation does not trigger a full page reload.
We utilized **TanStack Router** to handle client-side routing. The `RouterProvider` in `App.tsx` intercepts URL changes and dynamically renders the appropriate view component (e.g., swapping `MovieListView` for `MovieDetailsView`) without refreshing the browser. This provides a seamless, app-like user experience.

4.5.2 Navigation

A global navigation bar must be present on all pages.
The `NavigationWithSearch` component is part of the main layout route (`_layout.tsx`). It remains persistent at the top of the viewport while the content below changes. It includes links to "Home", "Movies", and "Persons", and integrates the global search bar.

4.5.3 Pagination

Lists of movies and actors must be paginated to handle large datasets.
We created a reusable `Pagination` component.

- **Visuals:** It displays "Previous" and "Next" buttons along with the current page number (e.g., "Page 1 of 42").
- **Logic:** It receives `onNext` and `onPrevious` callbacks from the parent view. These callbacks trigger methods in the MobX stores (e.g., `movieStore.nextPage()`), which fetch the new data and update the list.

4.5.4 Framework Features (Auth & Bookmarks)

Users must be able to register, login, and bookmark movies/actors.

- **Authentication:** The frontend logic for Login and Register is implemented in `AuthStore.ts` and corresponding views (`LoginView`, `RegisterView`). The store handles form validation (email/password) and manages the loading state. *Note: The actual API endpoints in the current code are placeholders (`/api of login`) and need to be connected to the real backend.*
- **Bookmarks: This feature is currently missing.** The frontend does not yet have a "Bookmark" button or a "My Bookmarks" page. This is a planned feature for the next iteration.

4.5.5 Search & Rating

Users must be able to search for content and rate movies/actors.

- **Search:** Implemented via `SearchStore.ts` and `NavigationWithSearch.tsx`. It supports live search with debouncing to minimize API load.
- **Rating: User rating functionality is currently missing.** While the application displays *average ratings* (read-only) fetched from the backend, there is no UI component allowing users to submit their own ratings. This is a planned feature.

4.6 Non-functional Requirements

4.6.1 Tech Stack Justification

We selected a modern technology stack to ensure performance, maintainability, and developer productivity.

- **React (v18):** Chosen for its component-based architecture and vast ecosystem. It allows us to build reusable UI elements and manage complex state efficiently using the Virtual DOM.
- **TypeScript:** Adopted to enforce type safety across the application. This significantly reduces runtime errors by catching type mismatches during development (e.g., ensuring API responses match our interfaces) and improves developer experience with better IDE autocompletion.
- **Vite:** Selected as the build tool and dev server. Compared to Create React App (Webpack), Vite offers significantly faster cold start times and Hot Module Replacement (HMR), which accelerates the development feedback loop.
- **Styled Components:** Used for styling to enable "CSS-in-JS". This provides scoped styling (preventing global namespace pollution) and allows us to adapt styles dynamically based on component props (e.g., changing a button's color based on its `disabled` state).

4.6.2 Code Quality & Architecture

To ensure the codebase remains maintainable and scalable, we adhered to several architectural principles:

- **Feature-Based Folder Structure:** Instead of grouping files by type (e.g., all controllers in one folder), we grouped them by feature (e.g., `features/movies`, `features/persons`). This "Colocation" principle keeps related code close together, making it easier to understand and modify specific functionality without traversing the entire directory tree.
- **Separation of Concerns (SoC):** We strictly separated the different layers of the application:
  - **Views:** Only handle rendering and user interaction.
  - **Stores (MobX):** Handle business logic and state management.
  - **API Client:** Handles raw HTTP communication.

  This separation ensures that changes in one layer (e.g., switching from Fetch to Axios) do not ripple through to others (e.g., the UI components).
- **Linter & Formatter:** We utilized **ESLint** and **Prettier** to enforce consistent coding standards and formatting rules automatically. This reduces code review friction and ensures a uniform code style across different team members.

Chapter 5

Conclusion

Bibliography

[Jak] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. https://www.nngroup.com/articles/ten-usability-heuristics/. Accessed: 25 November 2025.

[Roba] Robert C. Martin (Uncle Bob). *Domain Driven Design*. https://martinfowler.com/bliki/DomainDrivenDesign.html. Accessed: 17 Oktober 2025.

[Robb] Robert C. Martin (Uncle Bob). *The Clean Architecture*. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Accessed: 11 Oktober 2025.