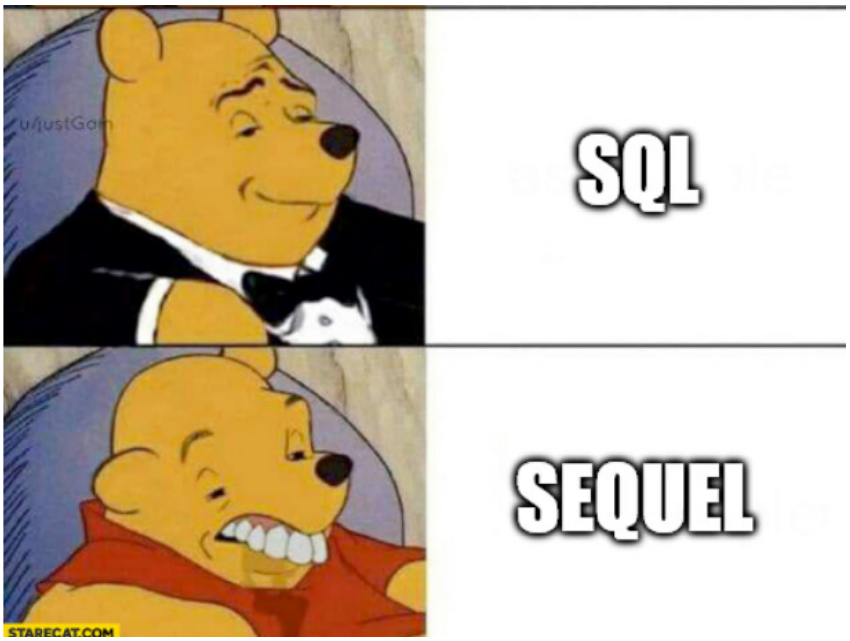
Educational Movie Database Software Manual

"CineLearn" (Cinematic Learning)

324827328, 213713522



Contents

Chapter 1

Client Application

1.1 Introduction

The development of the client-side application using Angular focused on creating a responsive and interactive user interface.

1.2 Technologies

Choosing the right tools was the first major decision we faced. We chose technologies based on two reasons, the first one is how much we can learn from it. For example, we really wanted to try out the new Angular Signals API, which we werent familiar with. The second reason is more technical:

1.2.1 TypeScript

We utilized TypeScript to leverage static typing in our development process. This provided type safety and facilitated refactoring, helping to identify potential errors across the codebase during compilation rather than at runtime.

1.2.2 Angular

We chose Angular because we wanted a framework that included everything out of the box. Unlike other libraries where you have to hunt for a router or a state management solution, Angular gave us a complete toolkit. We were particularly excited to use the new **Signals** feature. It allowed us to manage the complexity of our application state—like keeping the search results in sync with the selected genres—without writing complex boilerplate code.

1.2.3 Chart.js

For the analytics dashboard, we needed a library that could handle heavy lifting. We tried a few options, but Chart.js stood out because of its performance with the canvas element. When you're rendering multiple bar charts showing thousands of data points (like budget vs. revenue for flops), you need something that won't freeze the browser.

1.2.4 RxJS

RxJS was employed to manage asynchronous operations. As the application relies heavily on backend communication, Observables were used to handle complex data flows, such as coordinating concurrent API requests (using `forkJoin`) and managing search input streams (using `switchMap`).

1.3 Components

We broke the application down into three main areas, each addressing a specific user need.

1.3.1 The Home Page (HomeComponent)

The Home component (`client/src/app/components/home/home.ts`) serves as the primary interface for content discovery. A key technical requirement was the implementation of a responsive filtering system that updates results based on both text input and genre selection.

Instead of writing a complex chain of event listeners, we used a reactive approach. We created a "computed" signal that listens to both the search bar and the genre checkboxes. Whenever either changes, it automatically re-runs the filter logic.

```
filteredMovies = computed(() => {
  const query = this.searchQuery();
  const selectedGenres = this.selectedGenres();

  // This re-runs automatically whenever query or selectedGenres changes
  return this.movies().filter(movie => {
    const matchesSearch = movie.title.toLowerCase().startsWith(query.toLowerCase());
    const matchesGenre = selectedGenres.size === 0 ||
      movie.genres.some(genre => selectedGenres.has(genre));
    return matchesSearch && matchesGenre;
  });
});
```

Listing 1.1: Our reactive filtering solution

1.3.2 Movie Details (MovieDetailComponent)

The Detail view (`client/src/app/components/movie-detail/movie-detail.ts`) displays comprehensive movie metadata including cast, crew, and financial statistics.

A user identification mechanism was implemented to allow ratings without requiring account registration.

Shadow Identity

When a user visits for the first time, we check their local storage. If they don't have an ID, we generate a random string and save it. This acts as their "shadow identity," allowing us to track their votes across sessions.

```
private getUserId(): string {
  let userId = localStorage.getItem('cinelearn_user_id');
  if (!userId) {
    // Generate a sufficiently random ID
    userId = 'user_' + Math.random().toString(36).substring(2, 15) +
             Math.random().toString(36).substring(2, 15);
    localStorage.setItem('cinelearn_user_id', userId);
  }
  return userId;
}
```

Listing 1.2: Creating a persistent user identity

Optimistic UI

To enhance perceived performance, we implemented an optimistic UI for the voting system. The interface updates the vote count and user rating immediately upon submission, verifying the operation with the server in the background.

```
submitRating(): void {
  const movie = this.movie();
  const rating = Number(this.userRating());

  if (movie && rating >= 0 && rating <= 10) {
    this.movieService.rateMovie(movie.id, rating, this.userId)
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe({
        next: (response) => {
          this.message.set('Rating submitted successfully!');
          // We save it locally so the user sees their own vote next time
          localStorage.setItem(`rating_${movie.id}_${this.userId}`, rating.toString());
```

```
13
14            // And we update the global stats with the fresh data from the server
15            const updated = {
16               ...movie,
17               vote_average: response.vote_average,
18               vote_count: response.vote_count
19            };
20            this.movie.set(updated);
21          },
22          error: (e) => {
23            this.message.set('Error submitting rating.');
24          }
25        });
26    }
27 }
```

Listing 1.3: Handling the rating submission

1.3.3 Analytics Dashboard (AnalyticsComponent)

The Analytics component (`client/src/app/components/analytics/analytics.ts`) visualizes statistical data derived from the movie database.

Performance Challenges

Initial testing revealed latency issues during data loading, as the application awaited all data sources before rendering.

We solved this by decoupling the data loading. Instead of one giant request, we made each chart independent. We used Angular's 'toSignal' to transform each HTTP request into a signal that updates its specific chart whenever it arrives. This means the "Top Actors" chart might pop in first, followed by "Genres", keeping the user engaged while the heavier data loads.

```
1  // This doesn't block the rest of the application
2  readonly popularGenres = toSignal(
3    this.movieService.getPopularGenres(),
4    { initialValue: [] as any[] }
5  );
6
7  // This runs in parallel
8  readonly topActors = toSignal(
9    this.movieService.getTopActors(),
10   { initialValue: [] as any[] }
11 );
```

Listing 1.4: Decoupled data loading

1.4 Caveats & Usage Notes

1.4.1 The "Soft Session" Trade-off

You might notice there isn't a login screen. We consciously chose a "soft session" approach. The user ID is stored in the browser's Local Storage. This lowers the barrier to entry—anyone can just start rating movies immediately. The trade-off is that if you clear your cache, you become a new user. For this kind of public workshop application, we felt this was the right call.

1.4.2 One Vote Rule

To prevent spam, we enforced a rule: one vote per movie per user. However, we didn't want to lock users in. If you change your mind, you can rate again, and your new vote overwrites the old one. We handle this logic on the backend, but the frontend reflects it by loading your previous rating whenever you revisit a movie page.

Chapter 2

Backend API

2.1 Introduction

The backend for the Movie Database was designed with a focus on performance, reliability, and modularity. We adopted a layered architecture to ensure clear separation of concerns: Routers manage HTTP requests, the Repository layer abstracts database interactions, and the application logic connects these components.

2.2 Technologies

We chose the technilgies for the same reasons as written in the client:

2.2.1 Python

Python serves as the core language for our development. Its concise syntax allows for efficient expression of complex logic. Whether for data migration scripts or defining API endpoints, Python's readability facilitates maintenance and debugging.

2.2.2 FastAPI

FastAPI was chosen for its high performance and modern features, representing a significant improvement over older frameworks.

1. **Performance**: Utilizing modern Python features like `async/await`, it offers high throughput comparable to lower-level languages.
2. **Automatic Documentation**: The automatic generation of Swagger UI documentation facilitates efficient endpoint testing and verification.
3. **Data Validation**: Integration with Pydantic ensures rigorous data validation, rejecting invalid requests automatically.

We also just wanted to learn this technology, as its a standard for modern web development.

2.2.3 Uvicorn

Uvicorn serves as the ASGI (Asynchronous Server Gateway Interface) server for the application, enabling the handling of concurrent connections efficiently without blocking.

2.2.4 MySQL Connector

The `mysql-connector` library provides robust database connectivity. We encapsulated this within a `Repository` class to manage connection pooling and execution automatically.

2.2.5 Pandas

Pandas was utilized to efficienty import and process large CSV datasets. It allowed for in-memory data cleaning and transformation before migration to the SQL database.

2.2.6 Python-Dotenv

`python-dotenv` is used to manage environment variables, ensuring that sensitive credentials are kept secure in a local `.env` file and excluding them from version control.

2.3 Core Infrastructure

2.3.1 The Entry Point (`main.py`)

This file initializes the application. We configured the CORS (Cross-origin resource sharing) middleware to allow requests from the Angular frontend, which operates on a different port (4200).

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:4200"], # Allow our Angular app
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Listing 2.1: CORS Configuration

2.3.2 The Data Access Layer (DAL)

To maintain a clean codebase, we implemented a generic ‘Repository‘ class in ‘app/repository.py‘. This class wraps database interactions, managing connection pools and executing queries safely.

```
@staticmethod
def fetch_all(query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:
    conn = get_db_connection()
    # We use dictionary=True so we get real column names, not just tuples
    cursor = conn.cursor(dictionary=True)
    try:
        cursor.execute(query, params or ())
        return cursor.fetchall()
    finally:
        # Always clean up, even if the query fails
        cursor.close()
        conn.close()
```

Listing 2.2: Database execution wrapper

2.4 How It Works: API Modules

2.4.1 Movies Router (`routers/movies.py`)

A key challenge here was the "N+1 Problem". A naive approach would query the ‘movies‘ table once, and then run additional queries for each movie to fetch related data.

We addressed this by using "Bulk Fetching". We query the movies to obtain IDs, then execute optimized queries to fetch all related genres, cast, and crew data in bulk, mapping them in memory.

```
# 1. Get the movie IDs
movie_ids = [m['id'] for m in movies]

# 2. Get all related data in bulk
genres_data = Repository.fetch_all(f"SELECT ... WHERE movie_id IN ({placeholders})",
    movie_ids)

# 3. Map them in memory
genres_by_movie = defaultdict(list)
for g in genres_data:
```

```
    genres_by_movie[g['movie_id']].append(g['genre_name'])

# 4. Attach to movie objects
for movie in movies:
    movie['genres'] = genres_by_movie[movie['id']]
```

Listing 2.3: Aggregation in memory

The Rating Transaction

We perform ratings transactionally to ensure data consistency. We insert the rating and update the movie's average score within the same operation.

```
# Step 1: Insert the user's rating
Repository.execute("INSERT INTO ratings ... ON DUPLICATE KEY UPDATE...", (...))

# Step 2: Update the movie's cached average immediately
Repository.execute("UPDATE movies SET vote_average = %s ...", (...))
```

Listing 2.4: Transactional update

2.4.2 Analytics Router (`routers/analytics.py`)

These endpoints perform complex aggregation queries. For example, to identify "Flops", we calculate the difference between budget and revenue at the database level.

```
@router.get("/analytics/flops")
def flops():
    # We let the database engine do the subtraction and sorting
    query = """
        SELECT title, budget, revenue, (budget - revenue) as loss
        FROM movies
        WHERE budget > 1000000 AND revenue < budget AND revenue > 0
        ORDER BY loss DESC
        LIMIT 10
    """
    return Repository.fetch_all(query)
```

Listing 2.5: Analytics query example

Chapter 3

Database Schema

3.1 Introduction

The database design was a critical component of the project, establishing a foundation for API performance and application integrity. We aimed for a normalized schema to minimize redundancy while maintaining the flexibility required for analytical queries.

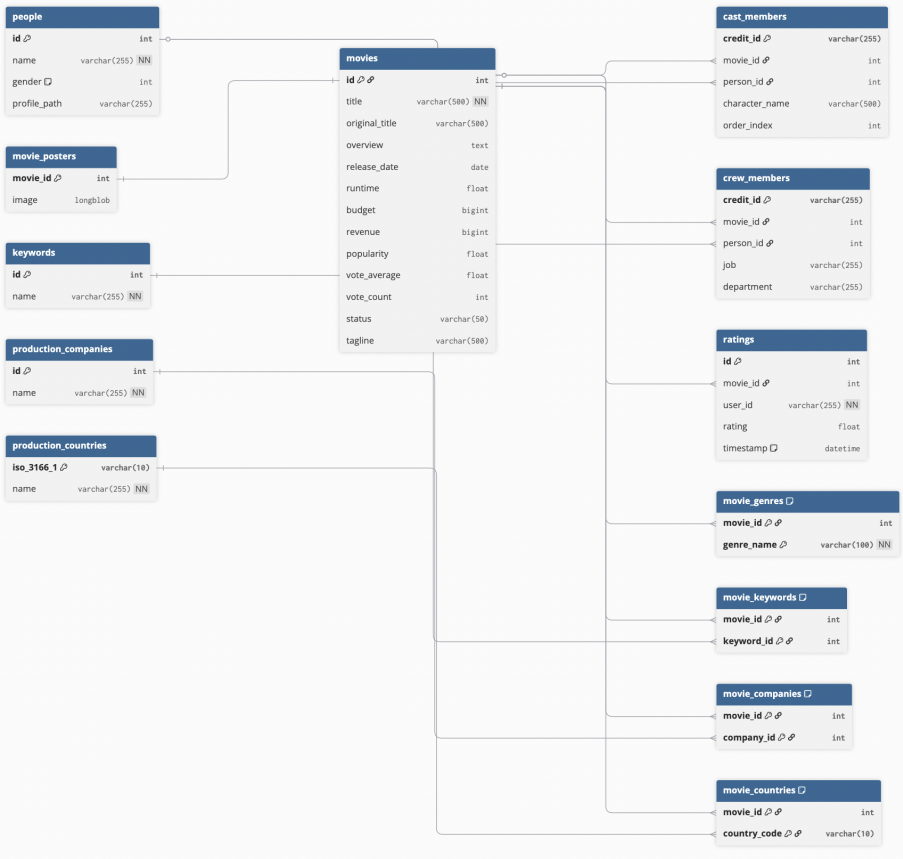


Figure 3.1: Full Database Schema Diagram

3.2 The Core Entities

3.2.1 Movies

The `movies` table is the central entity of the database. We explicitly included `vote_average` and `vote_count` fields in this table.

This denormalization strategy was chosen to optimize read performance. Instead of calculating averages from the `ratings` table for every query, we maintain these aggregated values transactionally.

```
CREATE TABLE IF NOT EXISTS movies (
    id INT PRIMARY KEY,
    title VARCHAR(500) NOT NULL,
    -- We keep the aggregated stats right here for speed
    vote_average FLOAT,
    vote_count INT,
    -- ... other metadata
    budget BIGINT,
    revenue BIGINT,
    status VARCHAR(50)
);
```

Listing 3.1: The heart of the system

3.2.2 People

We unified actors, directors, and writers into a single `people` table. This approach avoids data duplication for individuals who fulfill multiple roles (e.g., both acting and directing).

```
CREATE TABLE IF NOT EXISTS people (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    gender INT,
    profile_path VARCHAR(255)
);
```

Listing 3.2: A unified People table

3.3 Relationships

The database relational structure defines the connections between entities.

3.3.1 The Cast & Crew

Connecting `movies` to `people` required more than a simple link. We needed to know *what role* they played. For the **Cast**, we stored the `character_name` and an `order_index`. The index is crucial because it allows the frontend to validly display the "Top Billed" actors first, rather than a random alphabetical list. For the **Crew**, it's about the `job` and `department`. This granular detail allows us to answer interesting questions like "Which directors also acted in their own movies?"

```
CREATE TABLE IF NOT EXISTS cast_members (
    credit_id VARCHAR(255) PRIMARY KEY,
    movie_id INT,
    person_id INT,
    character_name VARCHAR(500),
    order_index INT,
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (person_id) REFERENCES people(id)
);
```

Listing 3.3: Linking People to Movies

3.3.2 Categorization (Genres, Keywords)

We handled categories like Genres and Keywords as Many-to-Many relationships. This means a movie can be both "Action" and "Comedy". We separated the names into their own lookup tables (`genres`, `keywords`) and used junction tables (`movie_genres`) to link them. This makes searching for "All Action movies" incredibly fast because it's just a simple index lookup.

```
CREATE TABLE IF NOT EXISTS movie_genres (
    movie_id INT,
    genre_name VARCHAR(100) NOT NULL,
    PRIMARY KEY (movie_id, genre_name),
    FOREIGN KEY (movie_id) REFERENCES movies(id)
```

```
);
```

Listing 3.4: Genre Association

```
CREATE TABLE IF NOT EXISTS keywords (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

Listing 3.5: Keywords Definition

```
CREATE TABLE IF NOT EXISTS movie_keywords (
    movie_id INT,
    keyword_id INT,
    PRIMARY KEY (movie_id, keyword_id),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (keyword_id) REFERENCES keywords(id)
);
```

Listing 3.6: Keyword Association

3.4 Production Details

To provide a comprehensive view of the film's origin, we track both the production companies and the countries involved. These are modeled as many-to-many relationships, acknowledging that modern films are often co-productions.

```
CREATE TABLE IF NOT EXISTS production_companies (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS movie_companies (
    movie_id INT,
    company_id INT,
    PRIMARY KEY (movie_id, company_id),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (company_id) REFERENCES production_companies(id)
);
```

Listing 3.7: Production Companies

```
CREATE TABLE IF NOT EXISTS production_countries (
    iso_3166_1 VARCHAR(10) PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS movie_countries (
    movie_id INT,
    country_code VARCHAR(10),
    PRIMARY KEY (movie_id, country_code),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (country_code) REFERENCES production_countries(iso_3166_1)
);
```

Listing 3.8: Production Countries

3.5 User Interaction

3.5.1 Ratings

We built the `ratings` table. We decided to use a string-based `user_id` to align with our "Soft Session" strategy on the client. This table grows the fastest, so we kept it lightweight: just who, what, what score, and when.

```
CREATE TABLE IF NOT EXISTS ratings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    movie_id INT,
```

```
    user_id VARCHAR(255) NOT NULL,
    rating FLOAT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 3.9: Capturing user votes

3.6 Binary Storage

3.6.1 Storing Images in the DB

We opted to store movie posters directly in the database using the `LONGBLOB` data type. While external storage (like S3) is a common pattern, storing images within the database ensures the dataset is self-contained. This simplifies backup and deployment procedures by keeping the entire application state in a single location.

```
CREATE TABLE IF NOT EXISTS movie_posters (
    movie_id INT PRIMARY KEY,
    image LONGBLOB,
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 3.10: Self-contained image storage

3.7 Application Logic Queries

This section documents the SQL queries used within the application's backend logic.

3.7.1 Movie Retrieval

For the main listing, we fetch movies with pagination logic.

```
SELECT * FROM movies LIMIT %s OFFSET %s
```

3.7.2 Movie Details Enrichment

To efficiently load related data for a list of movies (or a single movie), we use `IN` clauses or direct equality checks to fetch Genres, Cast, and Crew.

```
SELECT mg.movie_id, mg.genre_name
FROM movie_genres mg
WHERE mg.movie_id IN ({placeholders})
```

Listing 3.11: Fetching Genres

```
SELECT c.movie_id, p.id, p.name, p.gender, p.profile_path,
       c.character_name, c.order_index
FROM people p
JOIN cast_members c ON p.id = c.person_id
WHERE c.movie_id IN ({placeholders})
ORDER BY c.movie_id, c.order_index
```

Listing 3.12: Fetching Cast

```
SELECT c.movie_id, p.id, p.name, p.gender, p.profile_path,
       c.job, c.department
FROM people p
JOIN crew_members c ON p.id = c.person_id
WHERE c.movie_id IN ({placeholders})
```

Listing 3.13: Fetching Crew

3.7.3 Search

We implement a simple but effective text search using `LIKE` on the movie title.

```
SELECT * FROM movies WHERE title LIKE %s
```

3.7.4 Rating System

The rating system involves a transactional flow to ensure data consistency.

First, we check if the user has already rated the movie:

```
SELECT rating FROM ratings
WHERE movie_id = %s AND user_id = %s
```

Then, we perform an "Upsert" operation. If the record exists, we update the timestamp and rating; otherwise, we insert a new record.

```
INSERT INTO ratings (movie_id, user_id, rating)
VALUES (%s, %s, %s)
ON DUPLICATE KEY UPDATE
    rating = VALUES(rating),
    timestamp = CURRENT_TIMESTAMP
```

Finally, to keep the read-heavy `movies` table fast, we recalculate and update the aggregated statistics on the movie record itself.

```
UPDATE movies
SET vote_average = %s, vote_count = %s
WHERE id = %s
```

3.7.5 Assets

Retrieving the binary image data for a movie poster.

```
SELECT image FROM movie_posters WHERE movie_id = %s
```

3.8 Analytics & Reporting

The system enables complex analytical queries to derive insights from the movie database. We create 10 different queries to analyze the data. This section details the SQL formulation for these reports.

3.8.1 Genre Popularity

To determine which genres are most prevalent in the database, we aggregate the count of movies for each genre and order them descending.

```
SELECT genre_name AS genre, COUNT(movie_id) AS movie_count
FROM movie_genres
GROUP BY genre_name
ORDER BY movie_count DESC
LIMIT 5;
```

3.8.2 Actor Reuse (Prolific Actors)

This query identifies actors who have appeared in more than one movie. It uses a `HAVING` clause to filter groups (actors) after aggregation.

```
SELECT p.name as actor_name, COUNT(c.movie_id) as movie_count
FROM people p
JOIN cast_members c ON p.id = c.person_id
GROUP BY p.id, p.name
HAVING movie_count > 1
ORDER BY movie_count DESC
LIMIT 10
```

3.8.3 Runtime Analysis

We calculate the average runtime for each genre. Since a movie can have multiple genres, it contributes to the average of each genre it belongs to.

```
SELECT mg.genre_name AS genre, AVG(m.runtime) AS avg_runtime
FROM movie_genres mg
JOIN movies m ON mg.movie_id = m.id
WHERE m.runtime IS NOT NULL
GROUP BY mg.genre_name
ORDER BY avg_runtime DESC
LIMIT 10
```

3.8.4 Financial Underperformance ("Flops")

We identify movies with a significant budget (over $1M) that failed to recover their costs. The loss is calculated directly in the projection as `budget - revenue`.

```
SELECT title, budget, revenue, (budget - revenue) as loss
FROM movies
WHERE budget > 1000000 AND revenue < budget AND revenue > 0
ORDER BY loss DESC
LIMIT 10
```

3.8.5 Director-Actors

This complex join finds individuals who have both directed and acted in the *same* movie. It requires joining `people` to `cast_members` and `crew_members`, and ensuring the `movie_id` matches across both roles.

```
SELECT p.name, m.title
FROM people p
JOIN cast_members c ON p.id = c.person_id
JOIN crew_members cr ON p.id = cr.person_id AND c.movie_id = cr.movie_id
JOIN movies m ON c.movie_id = m.id
WHERE cr.job = 'Director'
```

3.8.6 Global Reach

We aggregate production countries to see which nations produce the most content in our database.

```
SELECT pc.name, COUNT(mc.movie_id) as movie_count
FROM production_countries pc
JOIN movie_countries mc ON pc.iso_3166_1 = mc.country_code
GROUP BY pc.name
ORDER BY movie_count DESC
LIMIT 10
```

3.8.7 Content Richness

We rank movies by the number of associated keywords, using this as a proxy for the depth of metadata available.

```
SELECT m.title, COUNT(mk.keyword_id) as keyword_count
FROM movies m
JOIN movie_keywords mk ON m.id = mk.movie_id
GROUP BY m.id, m.title
ORDER BY keyword_count DESC
LIMIT 10
```

3.8.8 Genre Complexity

We identify movies that are categorized into more than 3 genres. This helps find films with complex or cross-cutting themes.

```sql
SELECT m.title, COUNT(mg.genre_name) as genre_count
FROM movies m
JOIN movie_genres mg ON m.id = mg.movie_id
GROUP BY m.id, m.title
HAVING genre_count > 3
ORDER BY genre_count DESC
```

3.8.9 Golden Era

We determine the "Best Year" for cinema based on the average user rating. We filter for years with at least 3 movies to avoid statistical outliers from years with very few releases.

```sql
SELECT YEAR(release_date) as year, AVG(vote_average) as avg_vote, COUNT(*) as
    movie_count
FROM movies
WHERE release_date IS NOT NULL
GROUP BY year
HAVING movie_count >= 3
ORDER BY avg_vote DESC
LIMIT 1
```

3.8.10 Multiskilled Crew

We find crew members who are talented enough to work in multiple different departments (e.g., Sound and Editing). We use `COUNT(DISTINCT cr.department)` to ensure we are counting different functional areas, not just multiple credits in the same department.

```sql
SELECT p.name, COUNT(DISTINCT cr.department) as dept_count
FROM people p
JOIN crew_members cr ON p.id = cr.person_id
GROUP BY p.id, p.name
HAVING dept_count > 1
ORDER BY dept_count DESC
LIMIT 10
```

Chapter 4

Bonus: Exceptional UI Design

4.1 Philosophy

The design objective was to create a modern, user-friendly interface comparable to commercial streaming platforms, moving beyond standard administrative layouts.

4.2 Visual Identity

4.2.1 Glassmorphism

We utilized a modern "Glassmorphism" aesthetic. By using semi-transparent backgrounds with background blur filters, we created a sense of depth and hierarchy. The movie cards are visually distinguished from the background, separating the foreground content from the ambient background to enhance user focus. We specifically chose this style, because Apple recently updated all there devices to this style, and we wanted to learn how to build it ourselves.

4.2.2 Micro-Interactions

Subtle hover effects were implemented on interactive elements. Buttons elevate slightly, and movie posters illuminate upon interaction. These micro-interactions provide tactile feedback, enhancing the responsiveness of the application.

4.2.3 Color Palette

We selected a curated palette of deep midnight blues for the background and vibrant accents for actions, replacing standard primary colors. This dark mode-first approach reduces eye strain and highlights the movie artwork.

4.3 Evolution

We started with a functional but stark layout and iterated towards the polished version we have today.

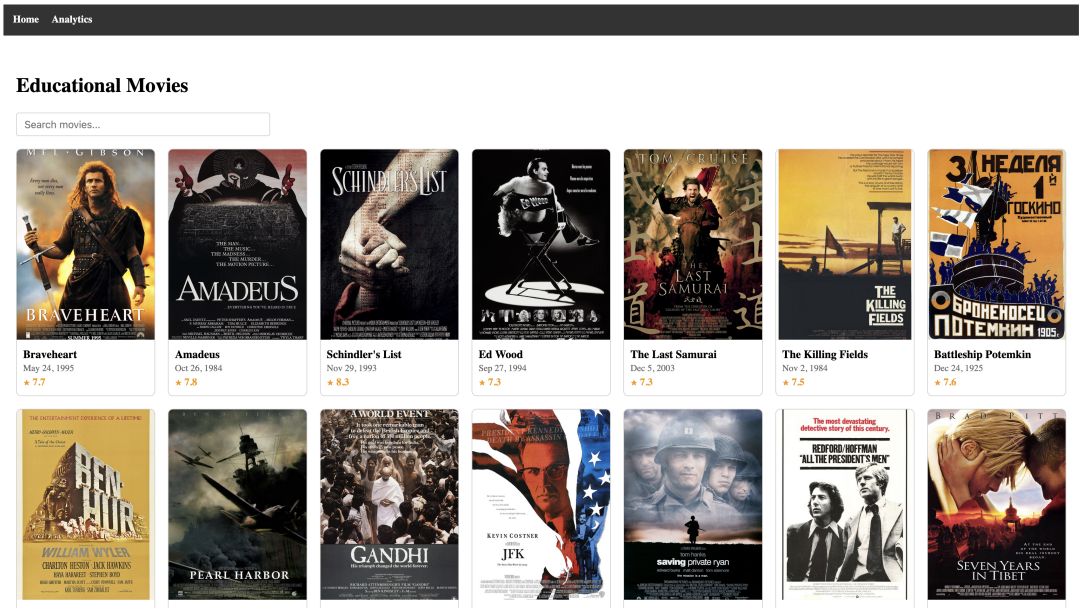


Figure 4.1: The initial functional prototype.

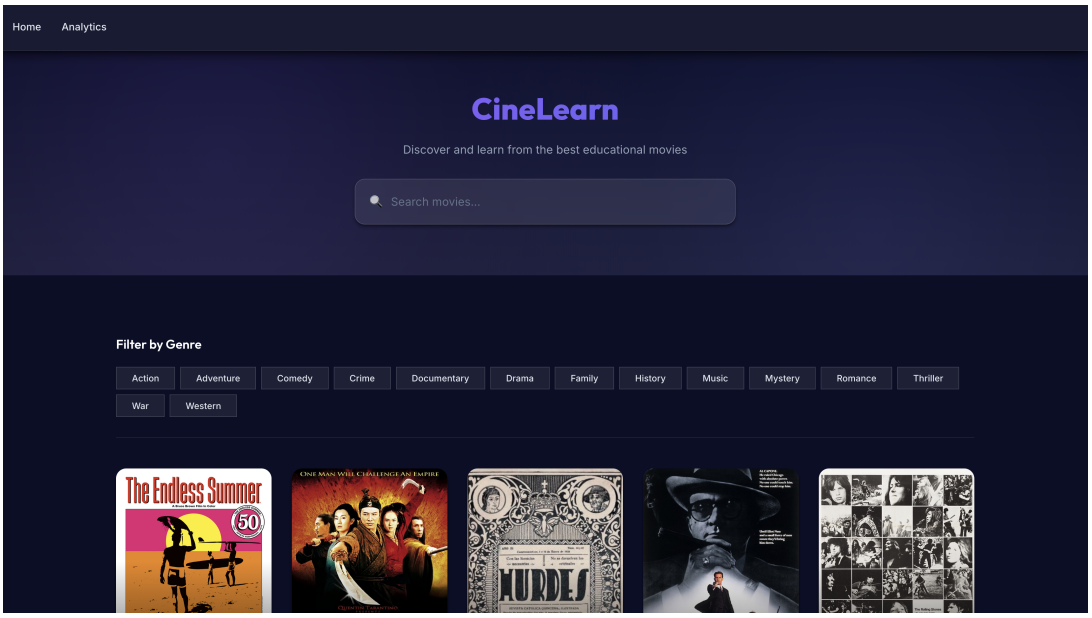


Figure 4.2: The final design with rich imagery and dark mode.