Backend Software Manual

324827328, 213713522

Contents

1 Introduction

When we designed the backend for the Movie Database, our philosophy was simple: build an engine that is invisible. It should be fast, reliable, and get out of the way. We chose a layered architecture not because it's "standard", but because it gave us the separation we needed. The Routers handle the "traffic", the Repository handles the messy database work, and the application logic sits comfortably in between.

2 Technologies

We didn't just pick random technologies; we chose a stack that allowed us to move fast without breaking things.

2.1 Python

Python is the backbone of our development. We chose it because it lets us express complex ideas in very few lines of code. Whether it was writing the data migration scripts or defining the API endpoints, Python's readability allowed us to spot bugs before we even ran the code.

2.2 FastAPI

Finding FastAPI was a game-changer. In the past, building APIs felt like boilerplate heaven. FastAPI is different.

1. **It's incredibly fast**: By using modern Python features like `async/await`, it handles traffic almost as well as Go or Node.js.
2. **It documents itself**: The best part for us was the automatic Swagger UI. Being able to test our endpoints in the browser without writing a single line of documentation code saved us days of work.
3. **It validates everything**: Using Pydantic models meant we never had to write "if x is not None" checks manually. The framework rejects bad data at the door.

2.3 Uvicorn

FastAPI is the brain, but Uvicorn is the heart. It's the server that actually runs our application. We picked it because it implements the ASGI standard, meaning it can handle many concurrent connections without blocking.

2.4 MySQL Connector

We needed a solid driver to talk to our database. `mysql-connector` gave us robust connection pooling out of the box. We wrapped this in our own `Repository` class so that we never had to worry about opening or closing connections manually—the code just handles it.

2.5 Pandas

Before we could serve any data, we had to import it. The raw datasets were massive CSV files. Processing these row-by-row would have taken forever. We used Pandas to load, clean, and transform the data in memory before dumping it into our SQL database. It turned a painful migration task into a quick script.

2.6 Python-Dotenv

We take security seriously. Hardcoding passwords is a novice mistake we wanted to avoid. `python-dotenv` allowed us to keep all our sensitive credentials in a local `.env` file that never gets committed to Git.

3 Core Infrastructure

3.1 The Entry Point (`main.py`)

This file is where the application boots up. We configured the CORS middleware right at the start. This was crucial because our Angular frontend runs on a different port (4200) than our backend (8000). Without this explicit permission, the browser would block every request.

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

Listing 1: Opening the gates for the Frontend

3.2 The Data Access Layer

We didn't want SQL queries scattered all over our codebase. To solve this, we built a generic 'Repository' class in 'app/repository.py'. Think of it as a safe wrapper around the database. You give it a query, and it handles the connection pool, executes the query safely (preventing SQL injection), and cleans up afterwards.

```
@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: Our safe execution wrapper

4 How It Works: API Modules

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

This is the workhorse of the API. The most interesting challenge here was the "N+1 Problem".

When you ask for a list of 20 movies, you also want their genres, their top cast, and their crew. A naive approach would query the 'movies' table once, and then run 3 more queries *for each movie*. That's 61 queries for a single page load!

We solved this by doing "Bulk Fetching". We query the movies, grab all their IDs, and then run just 3 extra queries to get ALL the genres, ALL the cast, and ALL the crew for those specific IDs in one go. We then stitch them together in Python.

```
# 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 3: Stitching data together in memory

4.1.1 The Rating Transaction

Rating a movie isn't just inserting a row. We also update the 'vote_average' on the movie itself. We do this transactionally so the global average is always accurate.

```
# 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 4: Two steps, one consistent result

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

This router is where we ask the difficult questions. Instead of simple lookups, these endpoints run complex aggregation queries. For example, to find "Flops", we ask the database to do the math: comparing budget versus revenue and sorting by the loss.

```
@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 5: Finding the biggest flops