Backend Software Manual

324827328, 213713522

Contents

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 Technologies

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

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 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.

2.3 Uvicorn

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

2.4 MySQL Connector

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

2.5 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.

2.6 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.

3 Core Infrastructure

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 1: CORS Configuration

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: Database execution wrapper

4 How It Works: API Modules

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 3: Aggregation in memory

4.1.1 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 4: Transactional update

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 5: Analytics query example