Zachary Renaud, Hangting Zhu, Zhexi Xu
Professor Zadorozhny
INFSCI 1500
December 4, 2025

Movie Rental Project Final Report

1: Overview of the System

This project implements a web based movie rental system that demonstrates how a relational database can support searching, browsing, and managing rental activity. The system is built using Python together with a small set of HTML templates, and it connects directly to a SQLite database that stores movies, customers, inventory copies, rentals, and payments.

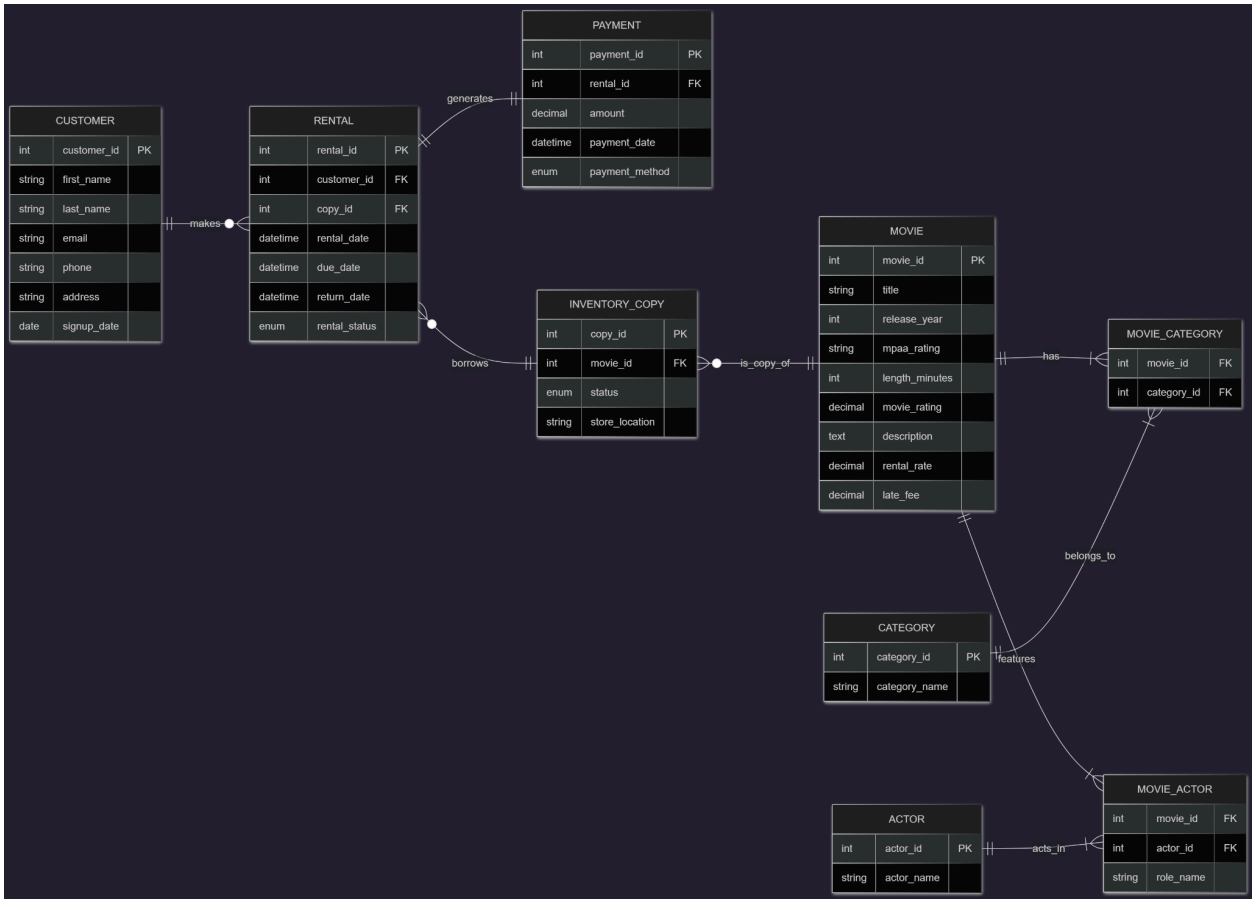
Users can browse all movies, search by title, and view detailed information such as availability and description. Each physical copy is tracked individually in the database, allowing the system to show how many copies are currently available. When a rental or return is submitted through the web interface, Python updates the related tables to keep the inventory and rental records consistent.

The goal of this project is to show how a normalized database schema, SQL queries, and a simple web interface can work together. The final system fulfills the required functionality for browsing, transactions, and data integrity while clearly illustrating the connection between a Python application and its underlying relational structure.

2:System Assumptions

In this project, we had to make some basic assumptions, because not everything was clearly written in the original description. First, we treat the system as only one video store. The store_location field in the inventory table is just where the copy is in the store (for example “Front Shelf” or “Kids Section”), not a different branch. Each physical disc is one row in INVENTORY_COPY, so if the store has three copies of the same movie, we just have three rows with the same movie_id.For customers, we assume that one email belongs to one person, and we set the email to be unique. We also assume that staff type in the correct name, phone, and address, because we did not add extra validation tools. For rentals, we assume one row in the RENTAL table means one customer renting one physical copy. If the customer rents three movies at once, that becomes three separate rental records. In our design we usually think of one payment per rental, and the due date is always five days after the rental_date. Late fees are calculated per day using the late_fee value in the MOVIE table.About who uses the system, we mainly think of store employees using this website, not customers. We assume login and security

are handled outside this project, so we did not build a full login system. Customers do not log in by themselves; staff use the interface to do actions for them. For the data itself, MPAA ratings are just short text like G, PG-13 or R. We also assume staff keep category names and actor names consistent, so we don't end up with the same thing spelled in two different ways.



Entity & Relationship Descriptions:

- Customer: Stores personal information (name, address, customer ID) for individuals who have or are renting movies.
- Rental: A transaction entity recording the event of a Customer borrowing a specific Inventory Copy. It tracks dates (rental, due, return) and the transaction status.
- Payment: Records the financial transaction associated with a specific Rental. It tracks the amount paid and the payment method.
- Inventory_Copy: Represents a physical DVD or Blu-ray disc available in the store. It links a specific physical item to the abstract Movie details and tracks its shelf location and availability status.
- Movie: Stores details about a film (title, rating, release year, pricing) independent of the physical copies.
- Category & Actor: Reference entities storing distinct genres and actor names.

- Movie_Category & Movie_Actor: Associative (Bridge) entities that handle the Many-to-Many relationships. Movie_Actor specifically tracks the role_name for each actor in a specific movie.

CUSTOMER (customer_id, first_name, last_name, email, phone, address, signup_date)
MOVIE (movie_id, title, release_year, mpaa_rating, length_minutes, movie_rating, description, rental_rate, late_fee)
CATEGORY (category_id, category_name)
ACTOR (actor_id, actor_name)
INVENTORY_COPY (copy_id, status, store_location, movie_id*)
- *Foreign Key:* movie_id references MOVIE(movie_id)

RENTAL (rental_id, rental_date, due_date, return_date, rental_status, customer_id*, copy_id*)
- *Foreign Key:* customer_id references CUSTOMER(customer_id)
- *Foreign Key:* copy_id references INVENTORY_COPY(copy_id)

PAYMENT (payment_id, amount, payment_date, payment_method, rental_id*)
- *Foreign Key:* rental_id references RENTAL(rental_id)

MOVIE_CATEGORY (movie_id*, category_id*)
- *Foreign Key:* movie_id references MOVIE(movie_id)
- *Foreign Key:* category_id references CATEGORY(category_id)

MOVIE_ACTOR (movie_id*, actor_id*, role_name)
- *Foreign Key:* movie_id references MOVIE(movie_id)
- *Foreign Key:* actor_id references ACTOR(actor_id)

```sql
DROP DATABASE IF EXISTS movierental;
CREATE DATABASE movierental;
USE movierental;

-- 1. CUSTOMER
CREATE TABLE customer (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    phone VARCHAR(20),
    address VARCHAR(255),
    signup_date DATE NOT NULL
) ENGINE=InnoDB;
```

```sql
-- 2. MOVIE
CREATE TABLE movie (
    movie_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    release_year INT,
    mpaa_rating VARCHAR(10),
    length_minutes INT,
    movie_rating DECIMAL(3,1),
    description TEXT,
    rental_rate DECIMAL(4,2) NOT NULL DEFAULT 4.99,
    late_fee DECIMAL(4,2) NOT NULL DEFAULT 1.00
) ENGINE=InnoDB;

-- 3. CATEGORY
CREATE TABLE category (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    category_name VARCHAR(50) NOT NULL UNIQUE
) ENGINE=InnoDB;

-- 4. ACTOR
CREATE TABLE actor (
    actor_id INT AUTO_INCREMENT PRIMARY KEY,
    actor_name VARCHAR(100) NOT NULL
) ENGINE=InnoDB;

-- 5. INVENTORY_COPY
CREATE TABLE inventory_copy (
    copy_id INT AUTO_INCREMENT PRIMARY KEY,
    movie_id INT NOT NULL,
    status ENUM('AVAILABLE', 'RENTED', 'LOST') NOT NULL DEFAULT
'AVAILABLE',
    store_location VARCHAR(100),
    FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
        ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB;

-- 6. RENTAL
CREATE TABLE rental (
```

```
    rental_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    copy_id INT NOT NULL,
    rental_date DATETIME NOT NULL,
    due_date DATETIME NOT NULL,
    return_date DATETIME,
    rental_status ENUM('OPEN', 'RETURNED', 'LATE') NOT NULL DEFAULT
'OPEN',
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (copy_id) REFERENCES inventory_copy(copy_id)
        ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB;

-- 7. PAYMENT
CREATE TABLE payment (
    payment_id INT AUTO_INCREMENT PRIMARY KEY,
    rental_id INT NOT NULL,
    amount DECIMAL(8,2) NOT NULL,
    payment_date DATETIME NOT NULL,
    payment_method ENUM('CASH', 'CARD') NOT NULL,
    FOREIGN KEY (rental_id) REFERENCES rental(rental_id)
        ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB;

-- 8. MOVIE_CATEGORY (Bridge Table)
CREATE TABLE movie_category (
    movie_id INT NOT NULL,
    category_id INT NOT NULL,
    PRIMARY KEY (movie_id, category_id),
    FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
        ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB;

-- 9. MOVIE_ACTOR (Bridge Table)
CREATE TABLE movie_actor (
    movie_id INT NOT NULL,
```

```
    actor_id INT NOT NULL,
    role_name VARCHAR(100),
    PRIMARY KEY (movie_id, actor_id),
    FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (actor_id) REFERENCES actor(actor_id)
        ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB;
```

This movie rental database schema was designed to ensure data integrity and minimize redundancy, satisfying Third Normal Form (3NF).

First Normal Form (1NF) is satisfied in our schema; we do not store lists of values in a single cell. Instead, our schema handles multiple genres and actors by creating separate rows in the associative entities MOVIE_CATEGORY and MOVIE_ACTOR. Every column holds a single value.

The Second Normal Form (2NF) is satisfied in our schema, which is relevant for the bridge tables. In the MOVIE_ACTOR table, the role_name column depends on a composite primary key formed by movie_id and actor_id. The schema does not entirely depend on the actor or the movie. All other tables use a single-column auto-incrementing primary key, which prevents partial dependencies.

The Third Normal Form (3NF) is satisfied because attributes describe only the entity defined by the primary key. For example, we did not store the category_name text inside the MOVIE table. If we did, updating a genre's name would require updating thousands of movie records. By isolating genres in the CATEGORY table and referencing them with ID, category_name depends only on category_id.

6: Front End Design and Connection to the Back End

The front end of the system is implemented using a small set of HTML templates that provide simple pages for browsing movies, viewing details, creating rentals, and processing returns. All pages share a common layout defined in *base.html*, which includes the navigation bar, consistent styling, and a placeholder where individual pages insert their content. This keeps the interface uniform and makes each page easy to read and navigate.

The main pages include:

- browse_movies.html: displays the list of movies and supports keyword search by title.

- movie_detail.html: shows detailed information for a selected movie, including how many copies exist and how many are currently available.
- rent.html: provides a form for selecting a customer and a movie in order to create a rental.
- return.html: allows submitting a rental ID to process a return and updates the status of the item.
- popular_movies.html: presents an aggregated report of the most frequently rented movies.

Each page sends information to the Python back-end through standard GET or POST requests. When a user submits a form, for example, creating a rental or returning one, the form data is sent to a corresponding Python function, which performs the required SQL queries on the SQLite database. The results are then passed back to the HTML templates along with messages indicating success or errors. This design keeps the interface simple while making it clear how user actions affect the underlying data.

Through this structure, the front end serves mainly as a clean input and display layer, while the Python code handles database operations such as selecting available copies, inserting rental records, updating inventory status, and generating summary reports. The connection between the two sides is lightweight but effective, demonstrating how a basic web interface can interact with a relational database system.

7: Implementation Overview

The system is implemented as a small Python web application that connects to a SQLite database and provides a set of pages for browsing movies, viewing details, and managing rental activity. Each page corresponds to a specific function in the system and interacts with the database through SQL queries executed in the Python code. The following screenshots illustrate the main components of the implementation.

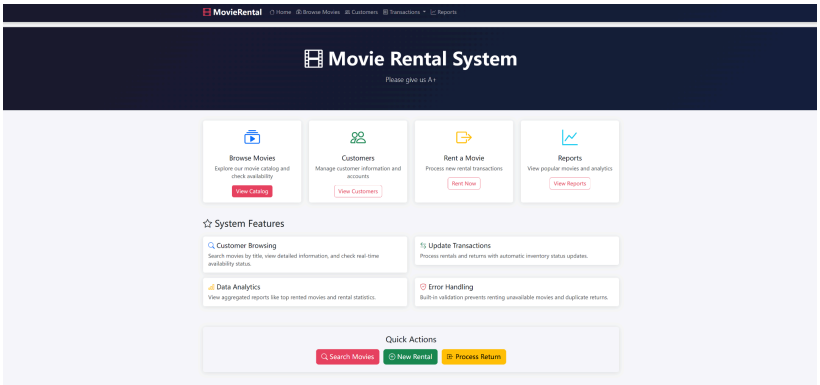


Figure 1 shows the home page, which provides a simple starting point and navigation links to all major functions of the system. The layout is defined in *base.html*, giving all pages a consistent structure.

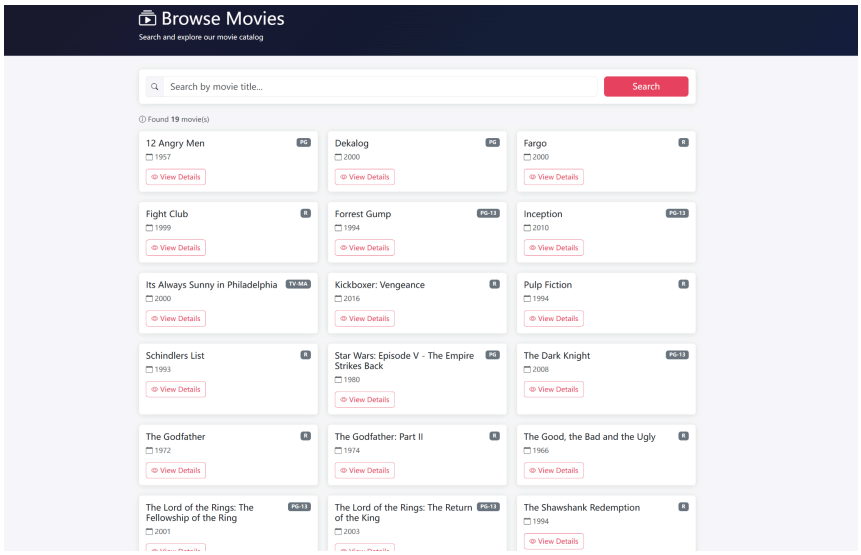


Figure 2 displays the movie browsing page. Users can scroll through the catalog or search by entering a keyword. The page sends a GET request to the Python backend, which retrieves matching movies from the database and returns them to the template for rendering.

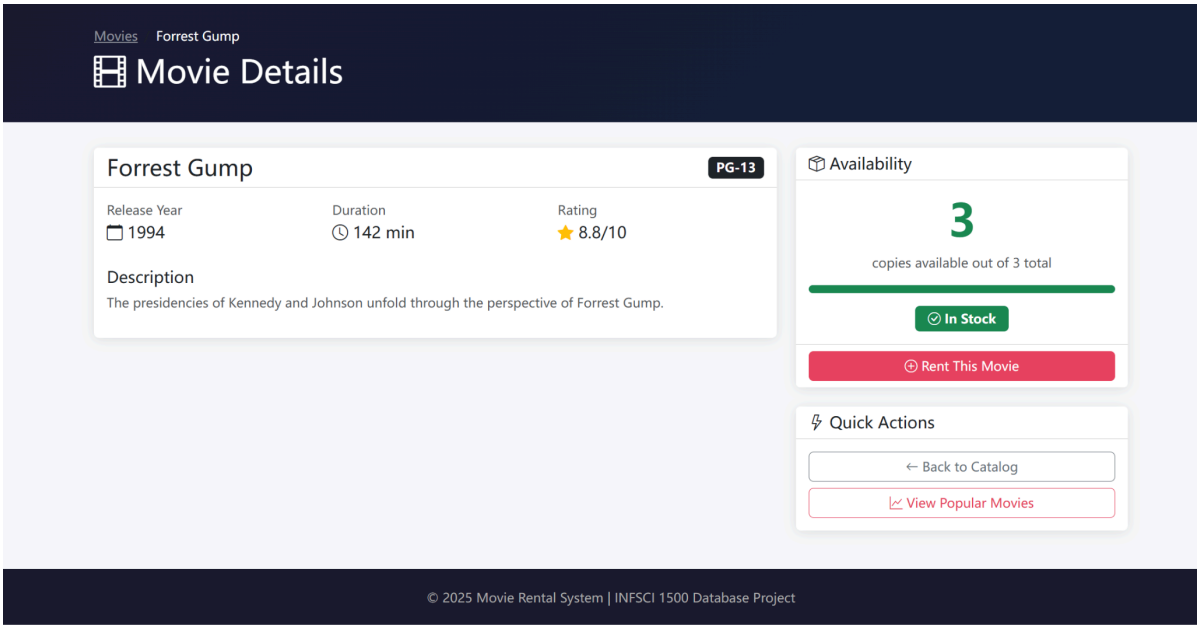


Figure 3 shows the movie detail page. For each movie, the system displays release information, a description, and real-time availability based on the number of physical copies with status "AVAILABLE." This information is calculated using a SQL query that counts copies in the inventory_copy table.

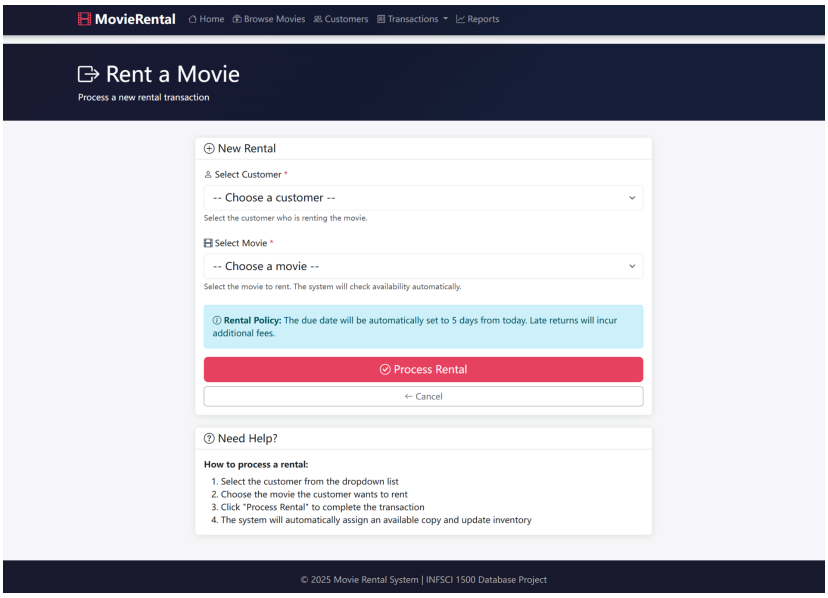


Figure 4 illustrates the rental creation page. When a rental is submitted, Python verifies that an available copy exists, inserts a new row into the *rental* table, updates the status of the selected inventory copy, and then returns a success or error message to the page.

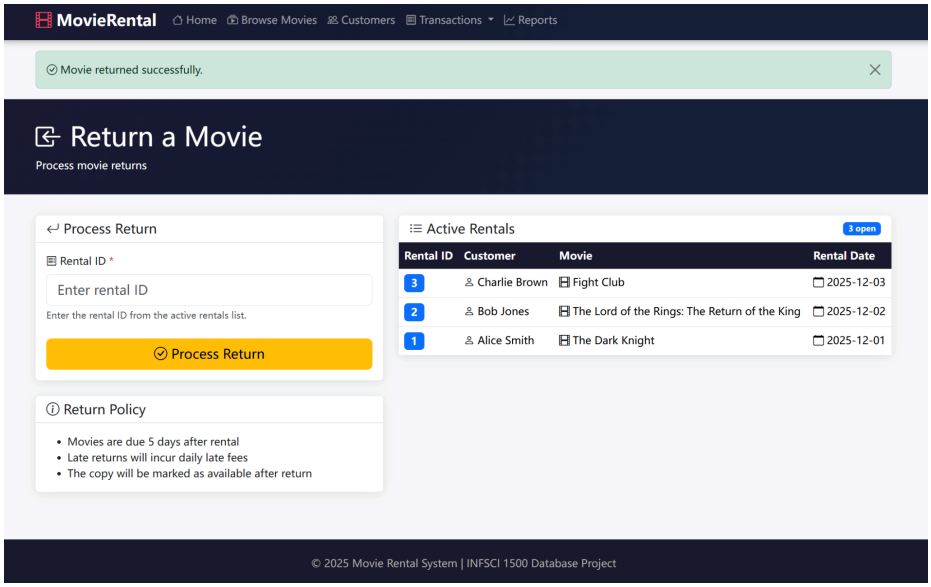


Figure 5 shows the return page, where users can enter the ID of an active rental. The backend checks whether the rental is valid and open, updates the return date, resets the copy's status to "AVAILABLE," and prevents incorrect operations such as returning a movie twice.

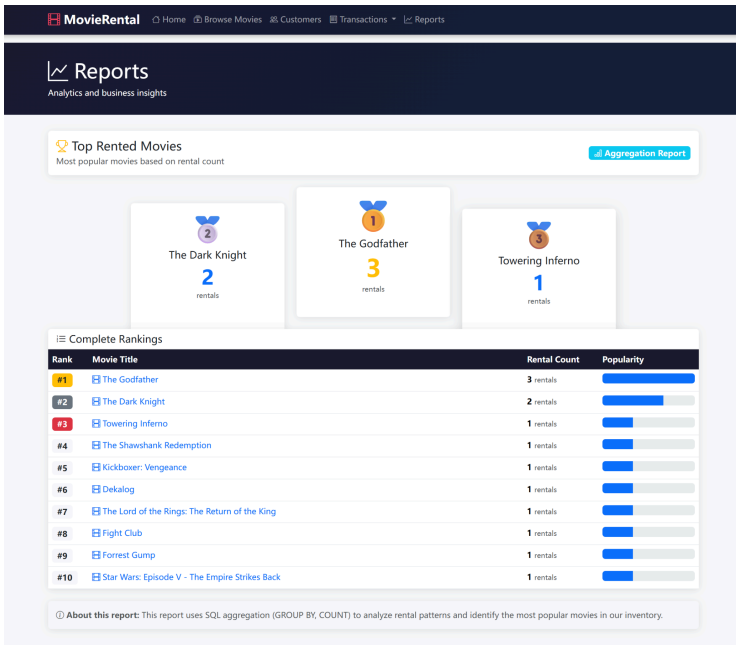


Figure 6 presents the popular-movies report. This page demonstrates the use of aggregation queries by listing the most frequently rented films and displaying their rental counts in descending order.

Together, these components show how the project integrates Python code, HTML templates, and SQL queries to implement browsing, transaction processing, and reporting in a clear and functional way.

8: Testing and Error Handling

For testing, we mainly did two things: we tested the database with SQL, and we tested the Flask web app in the browser.
On the database side, we wrote a set of SQL queries that follow the main tasks in the description. For example, we tested searching by actor by joining the movie, movie_actor and actor tables to find all movies with a specific actor (like Keanu Reeves) and show the character name. We also tested searching by category by joining movie, movie_category and category to list all Sci-Fi movies and their descriptions. To check inventory, we grouped the inventory_copy table by movie and counted how many copies have status “AVAILABLE”, just to make sure our copy status is correct.We also simulated renting and returning. First, we inserted a rental record with a due date five days after the rental_date and changed the copy’s status to “RENTED”. Then we simulated a return by setting rental_status to “RETURNED”, filling in the return_date and changing the copy back to “AVAILABLE”. After each step we ran SELECT queries to see if the rental and inventory tables stayed in sync. We also inserted a payment row to see if the amount, payment date and method are stored correctly. Finally, we ran our “report” queries, such as top

rented movies, top spending customers, overdue rentals, a master rental log, and outstanding late fees, and checked if the results made sense based on our sample data.On the web app side, we manually tested the main Flask routes. On the home page and the movies page, we checked that all movies show up, that keyword search works, and that the movie detail page correctly shows total copies and how many are available. On the customers page, we checked that the list matches what we have in the customer table. On the rent page, when at least one copy is available, the app creates a new rental, sets the due date, changes the copy status to “RENTED”, and shows a success message. If there is no available copy, it shows an error message and does not insert anything. On the return page, when we enter a valid open rental, the app sets the return_date, changes the status to “RETURNED”, switches the copy back to “AVAILABLE”, and shows a success message. If we enter an invalid rental id or one that is already returned, it shows an error and does nothing. We also compared the popular-movies page with our SQL query for top rented movies to make sure the counts are the same.From these tests we can see some basic error handling as well. The system stops us from renting a movie when there is no available copy, and stops us from “returning” a rental that is already closed or does not exist. At the database level, foreign keys prevent us from inserting rentals with customers or copies that do not exist, and the unique email on customers stops duplicate accounts. Overall, this is not perfect testing, but it gives us some confidence that the main logic is working.

9: System Limitations and Possible Improvements

Our system basically finishes what the project asks for, but it is still pretty simple. The biggest limitation is that we do not have real user accounts or roles. Everyone using the site is basically the same “user”, and customers cannot log in to see their own rentals or payments. In a more complete version, we would add login and different roles, so staff and customers see different pages. Another problem is that many things are still done only with SQL, not in the web app. For example, adding new movies, actors, categories, or payments is not supported in the Flask interface. Right now the web app mainly does browsing, renting, returning, and showing a popular movies report. A better system would have full pages to manage movies, customers, inventory, and payments directly in the browser.Also, our payment and late-fee logic is very basic. We can calculate late fees with queries, but we do not automatically update a customer’s balance or block new rentals for people who owe money. The UI and validation are also quite simple: searching is only by title, the styling is plain, and we do not strictly check email, phone format, or release year. For a real store, we would improve the front-end design, add more filters, choose one main database system, and add stronger validation to keep the data clean.