CinePax

**Table of Contents**

[Executive Summary 3](#_Toc199382781)

[Project Overview 3](#_Toc199382782)

[Project Objectives 3](#_Toc199382783)

[Key Features 5](#_Toc199382784)

[Database Design 5](#_Toc199382785)

[Data Model 5](#_Toc199382786)

[Entity Relationships 6](#_Toc199382787)

[Entity Relationship Diagram (ERD): 7](#_Toc199382788)

[Schema Design 8](#_Toc199382789)

[MongoDB Implementation 8](#_Toc199382790)

[Importing Data: 8](#_Toc199382791)

[After Importing Data: 9](#_Toc199382792)

[Schema Definition 9](#_Toc199382793)

[Class Diagram: 10](#_Toc199382794)

[JSON Schema: 10](#_Toc199382795)

[Indexing and Performance Optimization 14](#_Toc199382796)

[Aggregation And Pipelining 21](#_Toc199382797)

[Aggregation Queries 21](#_Toc199382798)

[Aggregation Through Graphical User Interface (GUI) 27](#_Toc199382799)

[Query Patterns 28](#_Toc199382800)

[Basic CRUD Operations: 28](#_Toc199382801)

[Transactions: 31](#_Toc199382802)

[Features and Functionality 33](#_Toc199382803)

[Entity Management 33](#_Toc199382804)

[Relationship Management 34](#_Toc199382805)

[Search 34](#_Toc199382806)

[Advanced Database Concepts 34](#_Toc199382807)

[NoSQL Database Benefits 34](#_Toc199382808)

[Document-Based Data Storage 35](#_Toc199382809)

[Denormalization Strategies 35](#_Toc199382810)

[Embedded vs. Referenced Documents 36](#_Toc199382811)

[Challenges and Solutions 36](#_Toc199382812)

[Data Integrity 36](#_Toc199382813)

[Complex Relationships 36](#_Toc199382814)

[Query Optimization 36](#_Toc199382815)

[Data Consistency 36](#_Toc199382816)

[Future Enhancements 37](#_Toc199382817)

[User Interface 37](#_Toc199382818)

[Admin Interface 42](#_Toc199382819)

[Conclusion 44](#_Toc199382823)

## 

## Executive Summary

Cinepax is a full-stack web application built using the **MERN stack** (MongoDB, Express.js, React.js, Node.js). The system incorporates **JWT-based authentication and authorization** to ensure secure user access. It features **separate dashboards for admins and users**, allowing role-specific access and functionalities.

This project serves as an advanced database-driven application, managing critical operations of a movie theater, including **movies, theaters, showtimes, tickets, customers, seat allocation, payments, and staff**. Leveraging **MongoDB**, a NoSQL document-oriented database, the system demonstrates the practical application of **document-based data modeling**, complex **relationship handling**, and **scalable design**, which are essential for managing dynamic and real-time ticketing and showtime data.

The architecture follows a **modular and layered structure**, implementing the **Repository pattern** to maintain a clear separation of concerns among the **data access layer, business logic, and frontend user interface**. This design improves maintainability, scalability, and future extensibility of the system.

Overall, the Cinepax showcases the integration of modern web technologies and advanced database design to deliver a robust, secure, and user-friendly solution for cinema management and movie bookings.

## Project Overview

### Project Objectives

Cinepax was developed :

* To design and implement an **advanced, scalable database solution** tailored for movie theater management.
* To showcase the effectiveness of **NoSQL (MongoDB)** in handling complex, dynamic data typical of entertainment and booking systems.
* To enable **comprehensive entity management**, including movies, theaters, showtimes, tickets, customers, payments, seats, and staff.
* To model and maintain **complex relationships** among theater-related entities within a document-oriented database.
* To support **advanced querying and reporting capabilities**, such as real-time seat availability, ticket sales analysis, and customer insights.
* To ensure **data integrity and consistency** through the use of **transactions, locking mechanisms**, and proper document references (e.g., linking tickets with corresponding payments and seat reservations).
* To handle **concurrent operations** (e.g., simultaneous seat bookings) efficiently and safely, preventing conflicts and overbooking.
* To implement robust **authentication and authorization mechanisms** using **JWT**, ensuring secure access control for both admin and user roles.

A screen shot of a phone

AI-generated content may be incorrect.

### Key Features

* **Full-Stack MERN Implementation**: Built using MongoDB, Express.js, React.js, and Node.js, providing modern and scalable architecture.
* **Secure Authentication & Authorization**: Role-based access control using **JWT**, with **separate dashboards** for admins and users.
* **Complete CRUD Operations**: Manage all core entities including **Movies, Theaters, Showtimes, Tickets, Customers, Seats, Payments, and Staff**.
* **Complex Relationship Management**: Implementation of embedded documents and references (e.g., **embedding seats in theaters**, **linking payments to tickets**) to maintain data integrity.
* **Concurrency Control & Transaction Handling**: Use of **MongoDB transactions and locking mechanisms** to ensure consistency in **seat booking and payment operations**, especially under concurrent access.
* **Advanced Search & Filtering**: Search and filter across collections (e.g., **available seats**, **showtimes by movie**, **booking status**, etc.) with optimized query performance.
* **Dynamic Showtime Scheduling**: Real-time creation and update of **showtimes**, with automatic **seat allocation** and conflict prevention.
* **Automated Ticket Pricing & Payment Integration**: Handles **dynamic price calculations** based on showtime, seat type, and theater location, integrated with secure payment handling.
* **Customer Booking Management**: Allows users to view, book, cancel, and manage their ticket bookings through an intuitive interface.
* **Staff Management**: Admins can assign and manage staff across different theaters and roles.
* **Modular & Layered Architecture**: Follows the **Repository Pattern**, separating data access, business logic, and UI for maintainability and scalability.
* **Real-Time Feedback**: Ensures **instant feedback on booking status** (success/failure) and prevents double bookings using transaction-safe operations.

.

## Database Design

### Data Model

The system implements a comprehensive data model that represents essential entities in a movie theater environment. The primary entities include:

1. **Movies**
   * Movie details (MovieID, title, duration, genre, release date).
   * Relationships with showtimes.
2. **Theaters**
   * Theater information (TheaterID, name, location, capacity).
   * Embedded seats.
   * Relationships with showtimes and staff.
3. **Showtimes**
   * Showtime details (ShowtimeID, start time, end time).
   * Associations with movies and theaters.
   * Relationships with tickets.
4. **Tickets**
   * Ticket information (TicketID, price, purchase date).
   * Embedded payment details.
   * Relationships with showtimes, customers, and seats.
5. **Customers**
   * Personal information (CustomerID, first name, last name, email, phone).
   * Relationships with tickets.
6. **Seats**
   * Seat details (SeatID, seat number, row, availability).
   * Association with theaters (embedded).
   * Relationship with tickets.
7. **Payments**
   * Payment details (PaymentID, amount, payment method, payment date).
   * Association with tickets (embedded).
8. **Staff**
   * Personal and professional information (StaffID, first name, last name, role).
   * Association with theaters.

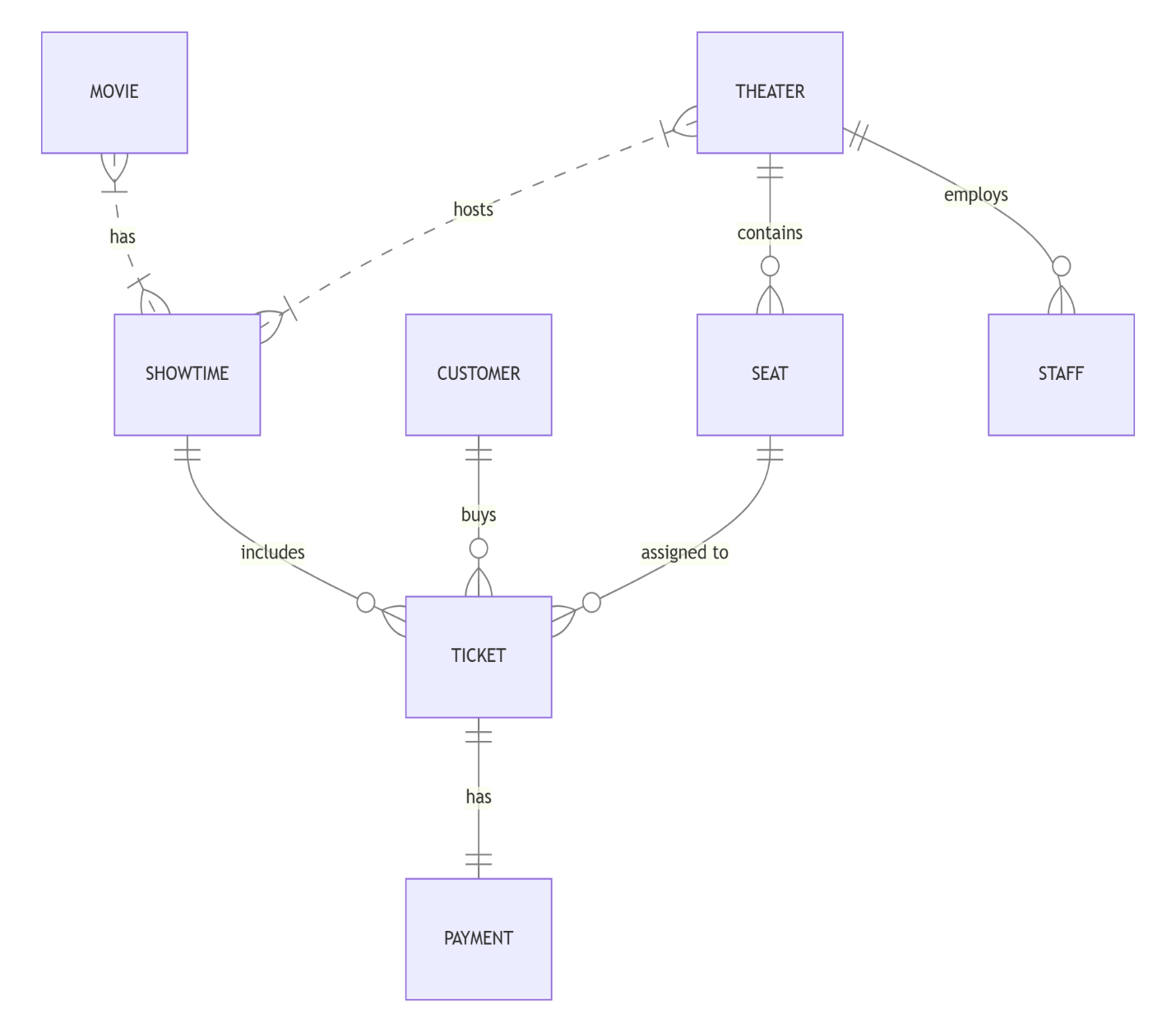
### Entity Relationships

The system incorporates complex relationships between entities, implementing both embedded and referenced document patterns based on access patterns and data cohesion:

1. **One-to-Many Relationships:** 
   * Movies to Showtimes: One movie can have many showtimes (referenced via MovieID in Showtimes).
   * Theaters to Showtimes: One theater hosts many showtimes (referenced via TheaterID in Showtimes).
   * Theaters to Seats: One theater contains many seats (embedded as an array in Theaters).
   * Showtimes to Tickets: One showtime has many tickets (referenced via ShowtimeID in Tickets).
   * Customers to Tickets: One customer can purchase many tickets (referenced via CustomerID in Tickets).
   * Theaters to Staff: One theater employs many staff members (referenced via TheaterID in Staff).
2. **Many-to-Many Relationships:** 
   * None explicitly present in this subset of entities. However, if extended (e.g., Customers to Movies for favorites), a separate collection would be used with references to CustomerID and MovieID.
3. **One-to-One Relationships:** 
   * Tickets to Payments: One ticket has one payment (embedded within Tickets for efficient retrieval).
   * Seats to Tickets: One seat is assigned to one ticket (referenced via SeatID in Tickets, with Seats embedded in Theaters).

The implementation of these relationships in MongoDB required careful consideration of document embedding versus referencing strategies, optimizing for the most common query patterns (e.g., retrieving ticket details with payments, checking seat availability per theater) while maintaining data integrity.

## Entity Relationship Diagram (ERD):



## Schema Design

Unlike traditional relational databases with rigid schemas, MongoDB’s schema flexibility was leveraged to adapt to the varying needs of different entities. However, a consistent schema approach was maintained at the application level to ensure data validation and consistency.

Key schema design decisions include:

1. **Referenced Documents**:
   * Movie references in Showtimes
   * Theater references in Showtimes
   * Customer references in Tickets.
   * Theater references in Staff.
   * Showtime references in Tickets.
2. **Partial Embedding**:
   * Selective embedding of frequently accessed related data
   * Storing minimal reference information for related entities
3. **ID Generation**:
   * Custom ID schemes for user-friendly entity identification
   * ObjectId for internal document references

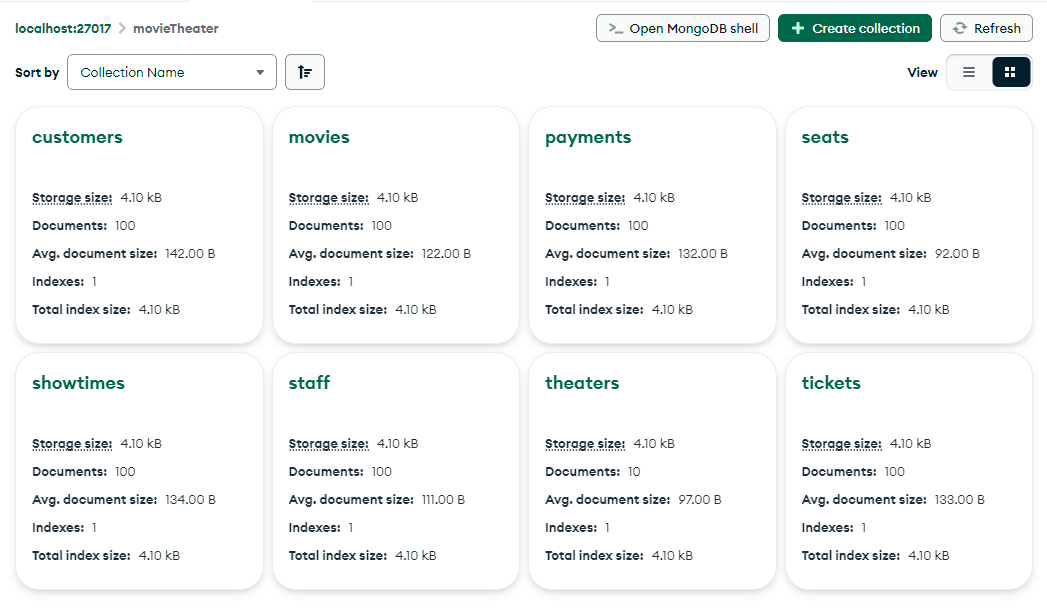
## MongoDB Implementation

### Importing Data:

Data in the mongo-dB collections is imported using these commands.

* mongoimport --db movieTheater --collection movies --file movies.json --jsonArray
* mongoimport --db movieTheater --collection theaters --file theaters.json --jsonArray
* mongoimport --db movieTheater --collection seats --file seats.json --jsonArray
* mongoimport --db movieTheater --collection showtimes --file showtimes.json --jsonArray
* mongoimport --db movieTheater --collection customers --file customers.json --jsonArray
* mongoimport --db movieTheater --collection tickets --file tickets.json --jsonArray
* mongoimport --db movieTheater --collection payments --file payments.json --jsonArray
* mongoimport --db movieTheater --collection staff --file staff.json –jsonArray

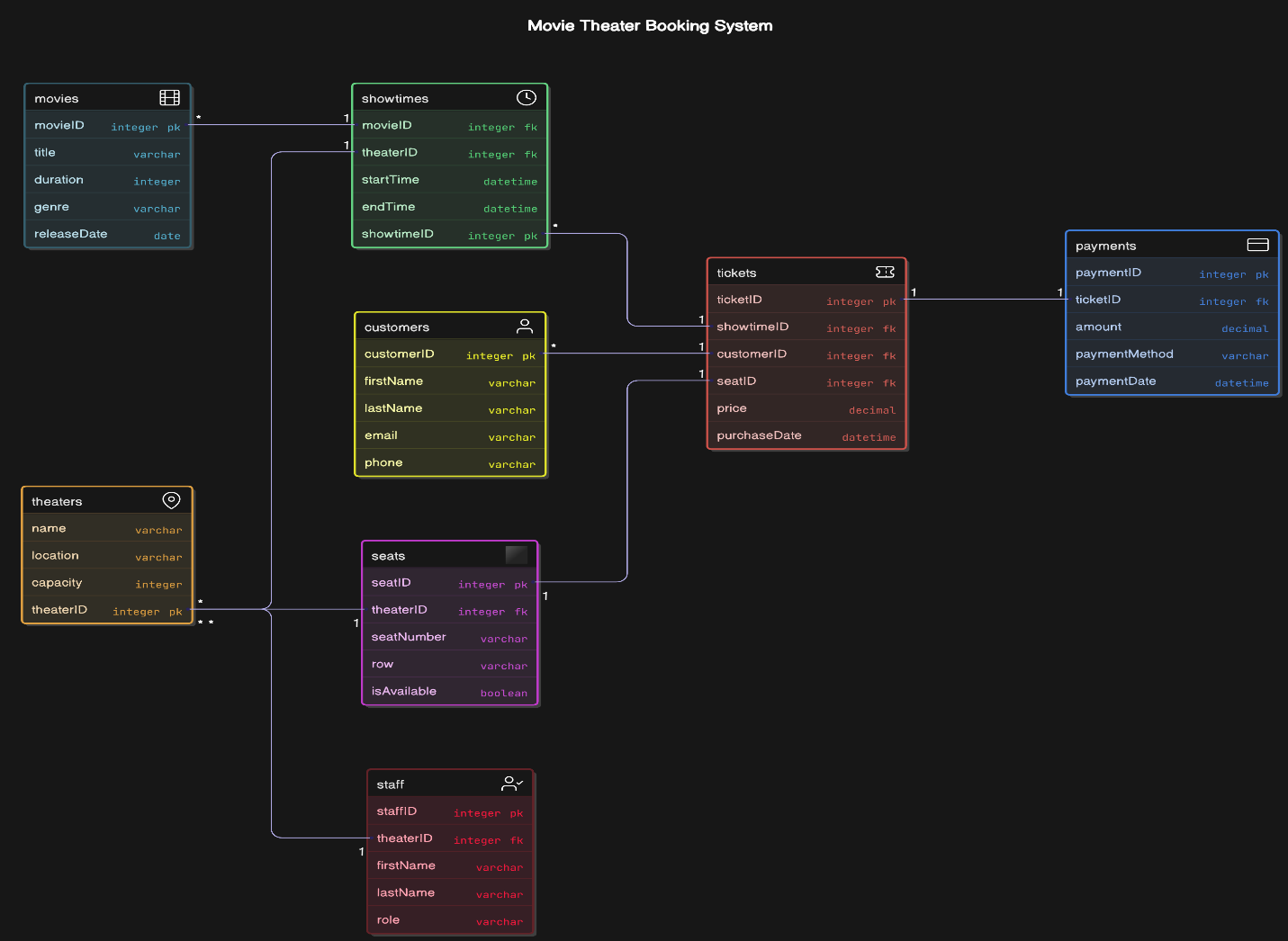
### After Importing Data:



## Schema Definition

While MongoDB is schema-less at the database level, the application implements schema validation through Python models. Each entity class defines the expected properties, types, and relationships:

## Class Diagram:



## JSON Schema:

**1) movies**

{

"$id": "xxxxx",

"type": "object",

"properties": {

"movieID": { "type": "integer" },

"title": { "type": "string" },

"genre": { "type": "string" },

"duration": { "type": "integer", "description": "Duration in minutes" },

"releaseDate": { "type": "string", "format": "date" },

"rating": { "type": "number", "minimum": 0, "maximum": 10 }

},

"required": ["movieID", "title", "genre", "duration", "releaseDate", "rating"]

}

**2) theaters**

{

"$id": "xxxxx",

"type": "object",

"properties": {

"theaterID": { "type": "integer" },

"name": { "type": "string" },

"location": { "type": "string" },

"totalSeats": { "type": "integer", "minimum": 0 }

},

"required": ["theaterID", "name", "location", "totalSeats"]

}

**3) seats**

{

"$id": "xxxx",

"type": "object",

"properties": {

"seatID": { "type": "integer" },

"theaterID": { "type": "integer" },

"seatNumber": { "type": "string" },

"type": { "type": "string", "enum": ["Standard", "VIP", "Premium"] },

"isAvailable": { "type": "boolean" }

},

"required": ["seatID", "theaterID", "seatNumber", "type", "isAvailable"]

}

**4)showtimes**

{

"$id": "xxxx",

"type": "object",

"properties": {

"showtimeID": { "type": "integer" },

"movieID": { "type": "integer" },

"theaterID": { "type": "integer" },

"startTime": { "type": "string", "format": "date-time" },

"language": { "type": "string" },

"format": { "type": "string", "enum": ["2D", "3D", "IMAX"] }

},

"required": ["showtimeID", "movieID", "theaterID", "startTime", "language", "format"]

}

**5) customers**

{

"$id": "xxxxx",

"type": "object",

"properties": {

"customerID": { "type": "integer" },

"name": { "type": "string" },

"email": { "type": "string", "format": "email" },

"phone": { "type": "string", "pattern": "^[0-9]{10}$" },

"memberSince": { "type": "string", "format": "date" }

},

"required": ["customerID", "name", "email", "phone", "memberSince"]

}

**6) tickets**

{

"$id": "xxxx",

"type": "object",

"properties": {

"ticketID": { "type": "integer" },

"customerID": { "type": "integer" },

"showtimeID": { "type": "integer" },

"seatID": { "type": "integer" },

"price": { "type": "number", "minimum": 0 },

"purchaseDate": { "type": "string", "format": "date-time" }

},

"required": ["ticketID", "customerID", "showtimeID", "seatID", "price", "purchaseDate"]

}

**7) payments**

{

"$id": "xxx",

"type": "object",

"properties": {

"paymentID": { "type": "integer" },

"ticketID": { "type": "integer" },

"amount": { "type": "number", "minimum": 0 },

"method": { "type": "string", "enum": ["Credit Card", "Debit Card", "Cash", "Online"] },

"paymentDate": { "type": "string", "format": "date-time" }

},

"required": ["paymentID", "ticketID", "amount", "method", "paymentDate"]

}

**8) staff**

{

"$id": "xxxx",

"type": "object",

"properties": {

"staffID": { "type": "integer" },

"name": { "type": "string" },

"role": { "type": "string", "enum": ["Manager", "Cashier", "Technician", "Cleaner", "Security"] },

"theaterID": { "type": "integer" },

"email": { "type": "string", "format": "email" }

},

"required": ["staffID", "name", "role", "theaterID", "email"]

}

### Indexing and Performance Optimization

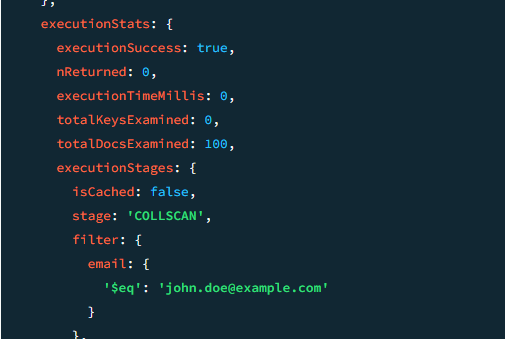
The system implements strategic indexes to optimize query performance:

**1)For Customers**:

**Before Indexing:**



**Result**:



**Creating Index:**

A computer screen with white text

AI-generated content may be incorrect.

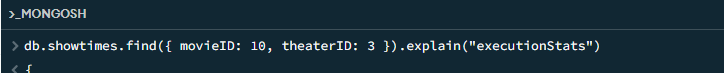
**After Indexing:**

A computer screen shot of a program code

AI-generated content may be incorrect.

**2)For ShowTime:**

**Before Indexing:**

****

**Result:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Creating Index:**

**A screen shot of a computer

AI-generated content may be incorrect.**

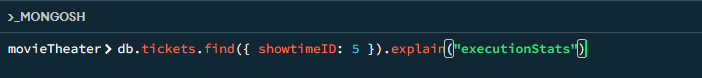
**After Indexing:**

**A computer screen shot of text

AI-generated content may be incorrect.**

**3) For Tickets:**

**Before Indexing:**

****

**Result:**

**A computer screen with text

AI-generated content may be incorrect.**

**Creating Indexing:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

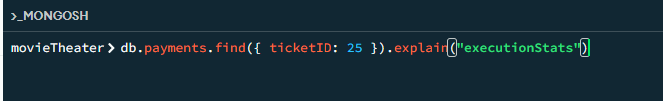
**After Indexing:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**4) For Payments:**

**Before Indexing:**

****

**Result:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Creating Indexing:**

**A screen shot of a computer

AI-generated content may be incorrect.**

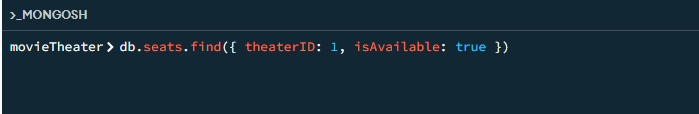
**After Indexing:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**5) For Seats:**

**Before Indexing:**

****

**Result:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Creating Indexing:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**After Indexing:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**6) For Movies:**

**Before Indexing:**

****

**Result:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Creating Indexing:**

**A computer screen shot of a code

AI-generated content may be incorrect.**

**After Indexing:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**7) For Staff:**

**Before Indexing:**

****

**Result:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Creating Indexing:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**After Indexing:**

A screen shot of a computer program

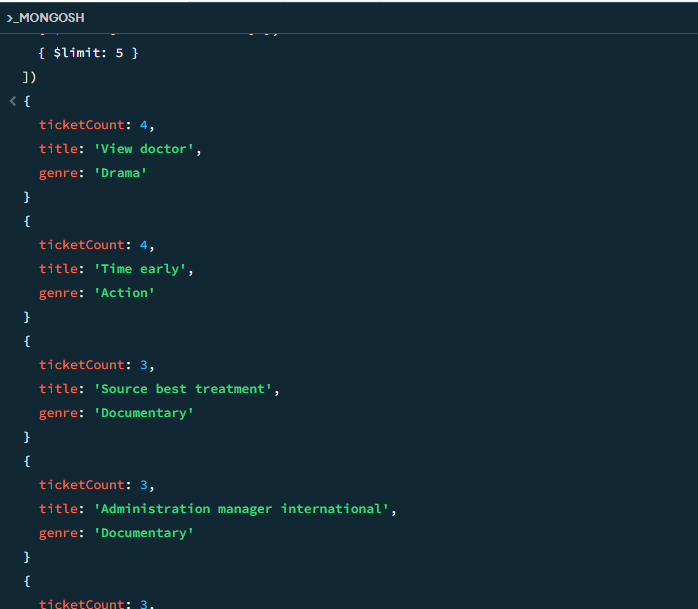
AI-generated content may be incorrect.

## Aggregation And Pipelining

### Aggregation Queries

* 1. **Top 5 most popular movies (based on ticket count)**

**Query:**

**Output:**

* 1. **Available vs Booked seats per theater**

**Query:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

A screen shot of a computer code

AI-generated content may be incorrect.

* 1. **Total Revenue Per Movie**

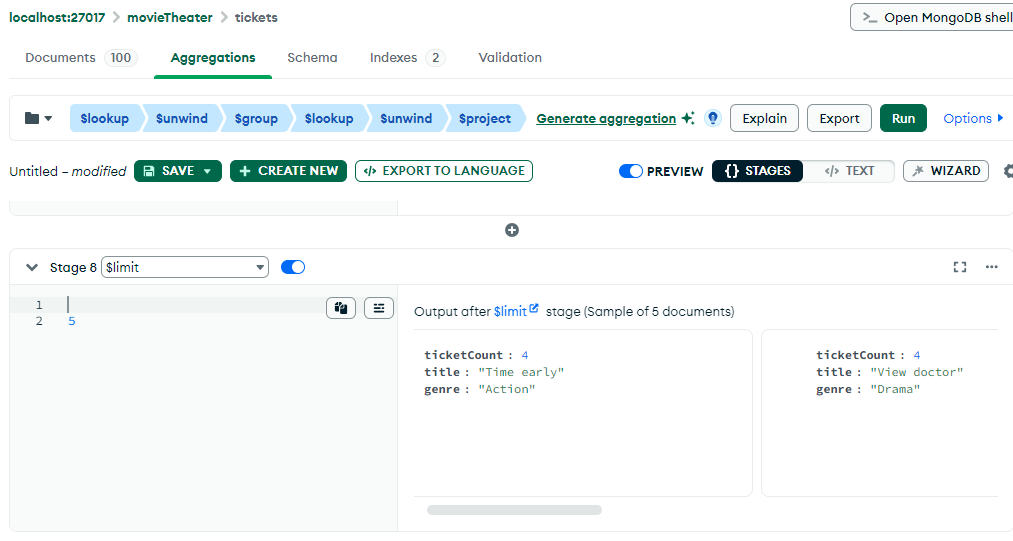
**A screenshot of a computer program

AI-generated content may be incorrect.Query:**

A screenshot of a computer program

AI-generated content may be incorrect.**Output:**

### Aggregation Through Graphical User Interface (GUI)



A screenshot of a computer

AI-generated content may be incorrect.

### Query Patterns

The application leverages MongoDB’s powerful query capabilities for efficient data retrieval:

### Basic CRUD Operations:

Document creation, retrieval, updating, and deletion

* **Create:**

****

* **Read:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

* **Delete:**

**A screen shot of a computer

AI-generated content may be incorrect.**

* **Find One Movie:**

****

* **Sort:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

* **Update:**

A screen shot of a computer program

AI-generated content may be incorrect.

## Transactions:

The entire booking process now runs in a transaction. If any part fails, all changes are rolled back. This ensures atomicity of the booking operation. Optimistic Locking:

Added version field to the Seat model Version is incremented on each save This helps detect concurrent modifications.

**Compound Index:**

Added a unique compound index on {showtimeId, seatNumber, row}

Prevents duplicate seat bookings at the database level

**Session-based Operations:**

All database operations use the same session. Ensures consistency across related documents

Error Handling:

* Proper transaction rollback on errors
* Detailed error logging
* Clean session cleanup
* The flow now works like this:
* Start a transaction session
* Check if the seat is available (with transaction)
* Create seat document (with transaction)
* Create ticket (with transaction)
* Create payment (with transaction)
* Update showtime's available seats (with transaction)
* Commit the transaction if everything succeeds
* Rollback if any error occurs.

This implementation provides:

* **Atomicity**: All operations succeed or fail together
* **Consistency:** Data remains in a valid state
* **Isolation**: Concurrent bookings don't interfere
* **Durability:** Changes are permanent once committed



## Features and Functionality

### Entity Management

#### The system provides comprehensive management capabilities for all movie theater entities:

#### **Movie Management**

#### Movie registration and details management (title, duration, genre, release date).

#### Catalog tracking.

#### Association with showtimes.

#### **Theater Management**

#### Theater information management (name, location, capacity).

#### Seat allocation and tracking (embedded seats).

#### Staff assignment.

#### **Showtime Management**

#### Showtime creation and scheduling (start time, end time).

#### Assignment to movies and theaters.

#### Ticket tracking.

#### **Ticket Management**

#### Ticket creation and booking management (price, purchase date).

#### Customer and showtime association.

#### Embedded payment processing.

#### Seat assignment.

#### **Customer Management**

#### Customer registration and profile management (first name, last name, email, phone).

#### Ticket purchase history.

#### **Seat Management**

#### Seat information (seat number, row, availability).

#### Embedded within theaters for capacity tracking.

#### Assignment to tickets.

#### **Payment Management**

#### Payment recording (amount, payment method, payment date).

#### Embedded within tickets for transaction tracking.

#### **Staff Management**

#### Staff information management (first name, last name, role).

#### Theater affiliation.

### Relationship Management

The system implements sophisticated relationship management between entities:

1. **Ticket Booking**
   * Bidirectional relationship between customers and tickets
   * Automatic updating of seat availability (IsAvailable in Seats) during booking/cancellation.
2. **ShowTime**
   * Association of movies and theaters with showtimes (MovieID and TheaterID in Showtimes).
   * Tracking of showtime schedules and ticket sales.
3. **Theater** 
   * Association of staff and seats with theaters (TheaterID in Staff, Seats embedded in Theaters).
   * Theater-specific seat and staff management.
4. **Payment Processing**
   * Embedded relationship between tickets and payments (Payments within Tickets).
   * Ensures payment completion for ticket issuance.

### Search

Advanced searches include:

1. **Multi-criteria Search**
   * Flexible search across all entity types (e.g., movies by genre, showtimes by theater).
   * Combination of exact (e.g., MovieID)
   * and partial matching (e.g., movie title keywords).

## Advanced Database Concepts

### NoSQL Database Benefits

The project leverages several key advantages of NoSQL databases:

1. **Schema Flexibility**
   * Adaptation to evolving data requirements
   * Varying attributes across similar entities
   * Extension of models without database migrations
2. **Scalability**
   * Horizontal scaling potential
   * Efficient handling of large document collections
   * Distributed database capabilities
3. **Document-Oriented Model**
   * Natural representation of hierarchical data
   * Intuitive mapping between application objects and database documents
   * Reduced need for complex joins

### Document-Based Data Storage

The system demonstrates advanced document-based storage patterns:

1. **Document Structure**
   * Rich document structures with nested fields
   * Arrays for multi-valued attributes
   * Document references for related entities
2. **BSON Data Types**
   * Utilization of MongoDB’s BSON format
   * Support for various data types (ObjectId, etc.)
   * Binary data handling
3. **Document Validation**
   * Application-level schema validation
   * Consistent document structure enforcement

### Denormalization Strategies

Strategic denormalization is employed to optimize query performance:

1. **Selective Field Duplication**
   * Copying frequently accessed fields to related documents
   * Balancing redundancy against query performance
2. **Aggregate Caching**
   * Storing pre-computed values (e.g, availableSeatCount)
   * Updating aggregates during data modifications
3. **Access Pattern Optimization**
   * Document structure tailored to common query patterns
   * Minimizing the need for joins or multiple queries

### Embedded vs. Referenced Documents

The system showcases the thoughtful application of embedded versus referenced document patterns:

1. **Embedding Documents**
   * Used for tightly coupled, frequently accessed together data
   * Implemented for managing contained relationships
   * Optimized for atomic updates and reads
2. **Referencing Documents**
   * Used for loosely coupled relationships
   * Implemented for many-to-many relationships
   * Applied for data that changes frequently or independently
3. **Hybrid Approaches**
   * Combination of embedding and referencing
   * Embedding summary information while referencing complete documents
   * Optimizing for specific query patterns

## Challenges and Solutions

### Data Integrity

**Challenge**: Maintaining referential integrity across collections without traditional foreign key constraints.

**Solution**: - Implemented application-level integrity checks - Created cascade update/delete operations - Developed transaction-like sequences for multi-document operations

### Complex Relationships

**Challenge**: Representing many-to-many relationships efficiently in a document database.

**Solution**: - Bidirectional references between documents - Atomic updates to maintain consistency - Careful management of reference synchronization

### Query Optimization

**Challenge**: Ensuring efficient query performance across complex document structures.

**Solution**: - Strategic indexing on frequently queried fields - Denormalization to reduce the need for joins - Query analysis and pattern optimization

### Data Consistency

**Challenge**: Maintaining consistent data across distributed document updates.

**Solution**: - Atomic document updates where possible - Transaction-like patterns for multi-document operations - Versioning mechanisms for conflict detection

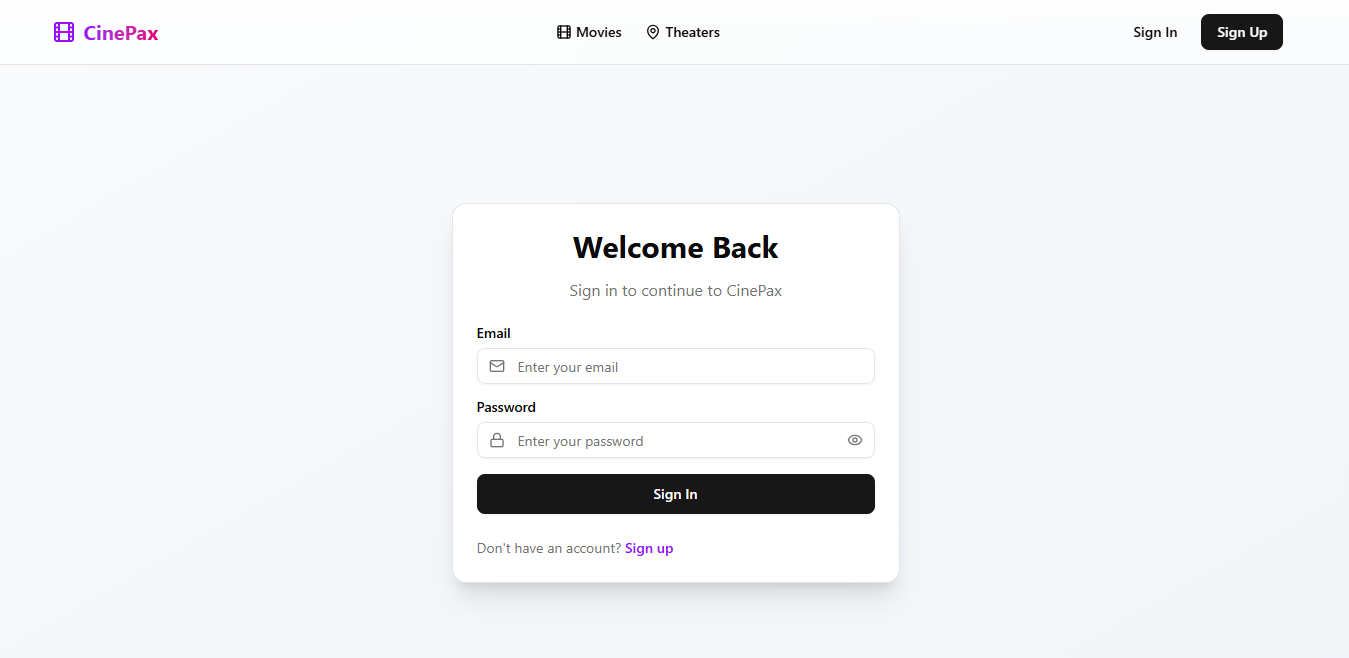
## Future Enhancements

1. **Advanced MongoDB Features**
   * Implementing change streams for real-time updates
   * Using MongoDB transactions for multi-document operations
   * Implementing sharding for horizontal scaling
2. **Extended Functionality**
   * Will introduce loyalty programs, concessions
3. **Performance Optimizations**
   * Implementing caching layers
   * Query optimization for large datasets
   * Aggregation pipeline improvements
4. **User Interface Improvements**
   * Web-based interface
   * Mobile application
   * API development for integration with other systems

## User Interface

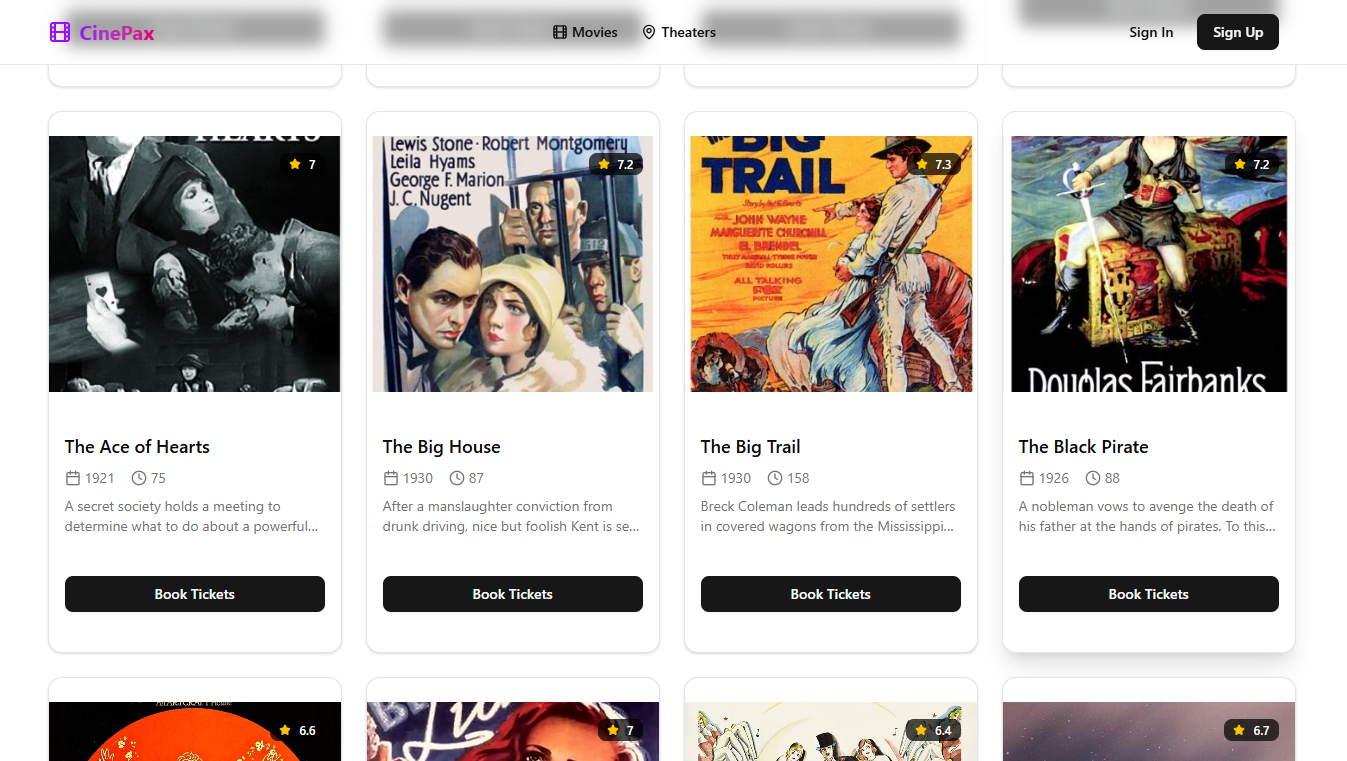
**A screenshot of a computer

AI-generated content may be incorrect.**

****

**A screenshot of a computer

AI-generated content may be incorrect.**

****

A screenshot of a movie theater

AI-generated content may be incorrect.

A screenshot of a computer

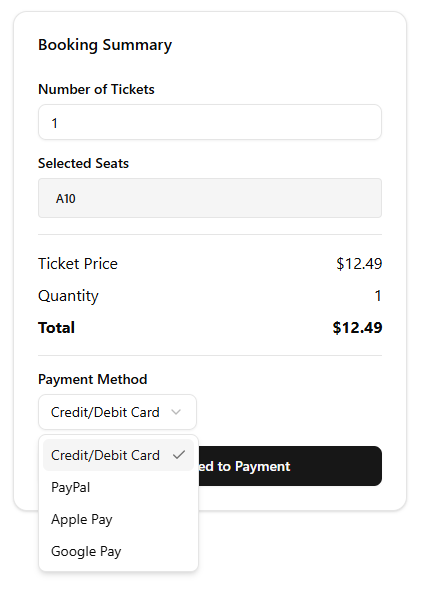
AI-generated content may be incorrect.

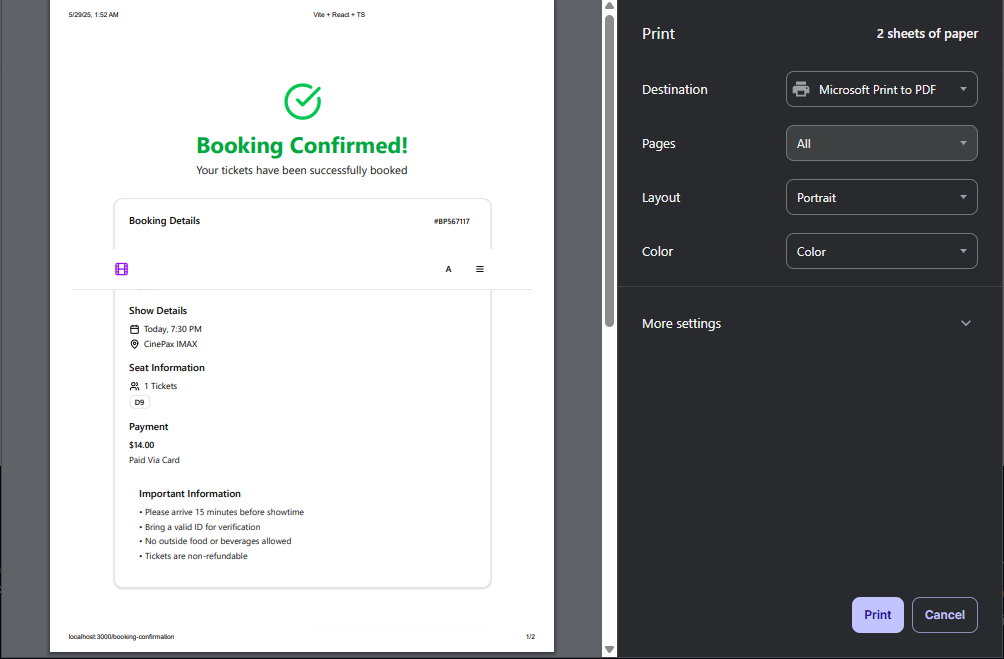
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.





## Admin Interface

****

**A screenshot of a computer

AI-generated content may be incorrect.**

## 

## 

## A screenshot of a computer AI-generated content may be incorrect.

## Conclusion

**Cinepax** demonstrates the practical implementation of an advanced, full-fledged movie theater booking web application, built using the **MERN stack** with **TypeScript** and deployed following **modern industry best practices**. The system effectively leverages **MongoDB** to model and manage complex relationships between theater-related entities, taking full advantage of NoSQL’s flexibility, scalability, and document-based design.

The architecture is clean and modular, adhering to the **Repository-Service-Controller pattern**, ensuring a clear separation of concerns and maintainability. Security and data integrity are central to the system, with robust implementation of **JWT-based authentication and authorization**, as well as support for **concurrency control**, **locking**, and **transaction management** to maintain consistency during critical operations like seat bookings and payments.

**🔑 Key Achievements**

* Developed a **production-grade, full-stack web application** using **TypeScript**, React, Node.js, Express, and MongoDB.
* Followed **clean architecture principles** ensuring modularity, testability, and scalability.
* Properly **deployed and containerized** the application for seamless production rollout.
* Implemented **advanced database features** including **transaction management**, **concurrency handling**, and **data locking** to ensure data consistency.
* Integrated secure **JWT-based authentication and role-based authorization**, with separate dashboards for **Admin** and **Users**.
* Designed and implemented **complex document-based relationships** (e.g., embedding Seats in Theaters, linking Payments to Tickets) in MongoDB.
* Enabled advanced **search, reporting, and booking functionalities**, including real-time seat availability and ticket price calculations.
* Ensured seamless and efficient management of Movies, Theaters, Showtimes, Customers, Staff, Payments, and Tickets.

This project serves as a scalable and extensible platform with a strong foundation for future enhancements like analytics dashboards, third-party payment integrations, and mobile app support. Cinepax successfully demonstrates how modern tech stacks and NoSQL databases can power robust, secure, and user-friendly booking systems.