Book My Show - Design

Contents

[Overview of the System 2](#_Toc194177305)

[Requirements Gathering & Clarify Requirements 3](#_Toc194177306)

[Identify Classes 5](#_Toc194177307)

[The City class and Theater class 8](#_Toc194177308)

[The Screen class 9](#_Toc194177309)

[The Seat class 10](#_Toc194177310)

[The Show class 12](#_Toc194177311)

[The mapping class - ShowSeat 14](#_Toc194177312)

[The Movie class 15](#_Toc194177313)

[The Actor class 17](#_Toc194177314)

[The Payment class 18](#_Toc194177315)

[The Ticket class 19](#_Toc194177316)

[The User class 21](#_Toc194177317)

[Class Diagram 22](#_Toc194177318)

[Schema Design 23](#_Toc194177319)

[Schema Design Principles: 23](#_Toc194177320)

[Table Schema (**Based on Primitive Attributes**): 23](#_Toc194177321)

[Cardinality and Foreign Keys: 26](#_Toc194177322)

[Model-View-Controller (MVC) + Service Layer + Repository Pattern 47](#_Toc194177323)

[Current Flow: (List all the movies) 48](#_Toc194177324)

[Final Summary 49](#_Toc194177325)

BookMyShow System Design

##### Overview

* Popular System: Frequently used, making it relevant for interviews.
* Concurrency Handling: No two persons should be able to book the same seat.

##### System Design Steps

1. Overview of the System
2. Requirements Gathering
3. Clarify Requirements
4. Class Diagram
5. Schema Design
6. Code Implementation (Spring Boot)

## Overview of the System

* Overview part you will have two cases

1. You know the application.
   * If you know the application you will have to tell the basics that, this is what I understand from the application named BookMyShow.
2. You don't know the application.
   * If you don't know the application you will have to ask that, I don't know the application and I have not used BookMyShow. Can you please elaborate in brief what do you mean by what kind of application we are building.

Assuming that we don't know BookMyShow!

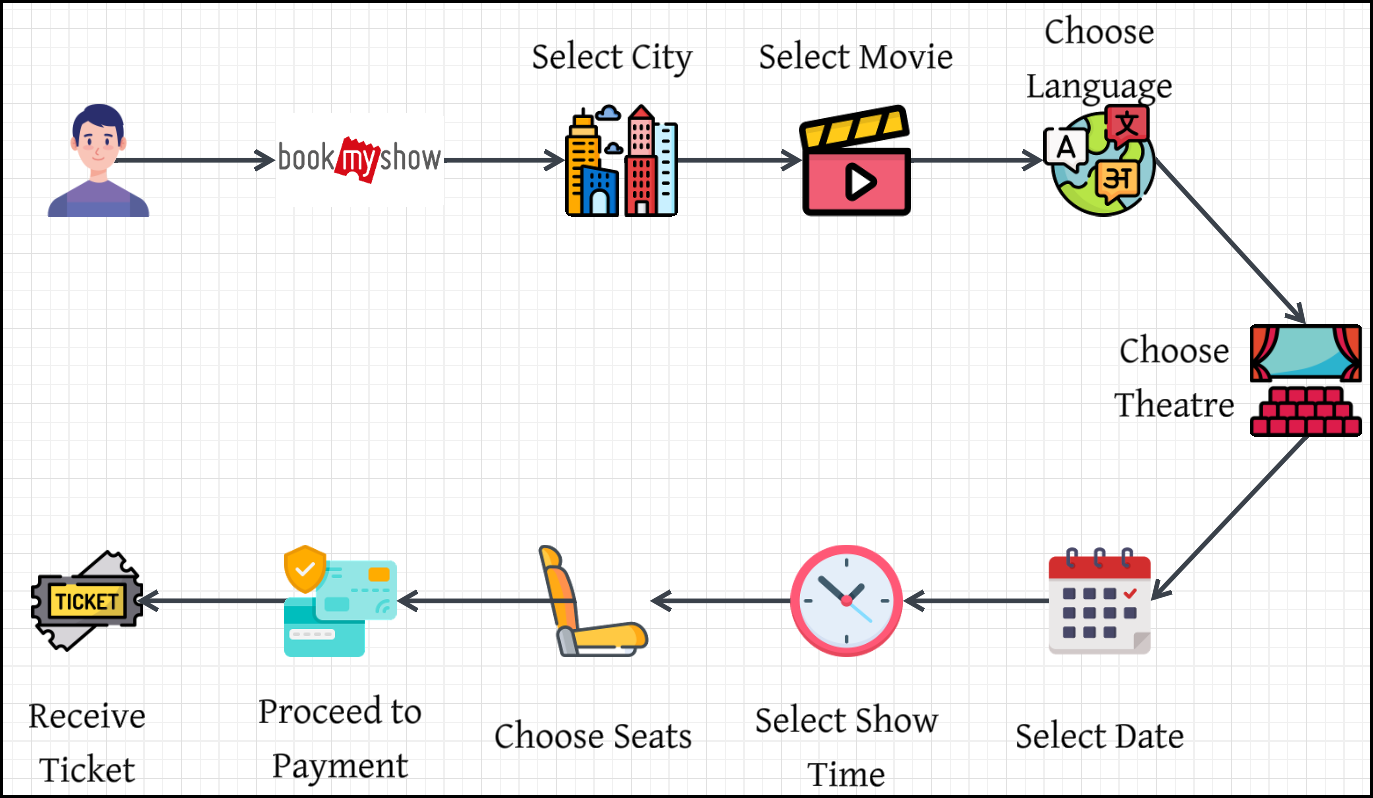
Interviewer will give you the problem statement.

##### Problem Statement: BookMyShow System Design

* Design an online movie ticket booking system (BookMyShow) that allows users to:
  + Browse and search for movies.
  + Select a city and find available theatres.
  + Choose a movie, language, date, showtime, and seats.
  + Make payments online to confirm bookings.
  + Ensure concurrency control so that no two users can book the same seat simultaneously.
* The system should be scalable to handle high traffic and support multiple cities, theatres, and screens with different seat types and pricing. It should also integrate with third-party payment providers and allow refunds and cancellations.

## Requirements Gathering & Clarify Requirements

* We will start with user journey…
* User Journey & Flow

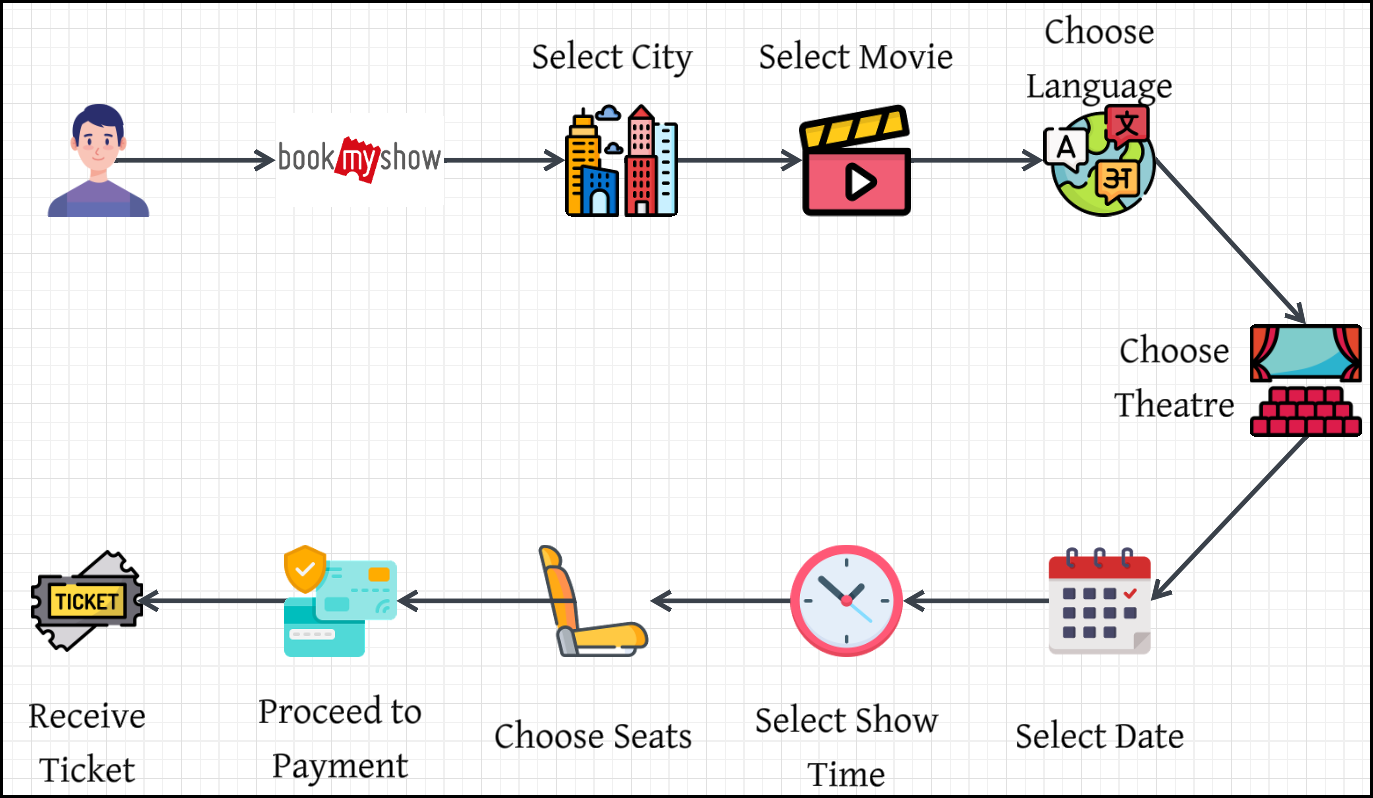


* Once we confirm the user journey, we can ask questions to the interviewer to gather the requirements.
* Start with the 1st spot in the user journey.

1. Cities & Theatres
   * There will be multiple cities.
   * Each city will have multiple theatres.
   * Each theatre can have multiple screens.
2. Screens & Seats
   * Screens will have different features: 2D, 3D, IMAX, Dolby.
   * Each screen will have multiple seats.
   * Seats can be of different types: Sofa, Recliner, Platinum, Gold, Silver, etc…
3. Pricing
   * Pricing is based on:
     + Movie
     + Time of the show
     + Seat type
     + Theatre
     + Day of the week
   * Each type of seat will have a different price per show.
4. Shows & Movies
   * Each theatre will have multiple shows running at the same time.
   * Each show will have one movie running.
   * A movie can have multiple attributes:
     + Languages
     + Cast
     + IMDb rating
     + Genre
     + Features (2D, 3D, IMAX, Dolby, etc.)
5. Booking Restrictions
   * Users cannot book tickets 10 minutes before the show.
   * After seat selection, users proceed to payment.
   * Only online payments are supported.
   * Payment will be outsourced to third-party providers (e.g., Razorpay, PayU).
6. Partial Payments
7. Refunds & Cancellations
8. Concurrency Handling
   * A user can book a maximum of 10 seats per transaction.
   * Once payment is successful, a ticket is generated.
   * Only one person should be able to book a particular seat.
   * Once a user selects a seat, it will be locked for 5 minutes.

## Identify Classes

* The two strategies for creating a class diagram are:
  1. **Visualization** – This includes understanding the system flow, identifying key interactions, and mapping user journeys.
  2. **Identifying Nouns** – Extracting relevant entities (nouns) from requirements or problem statements to determine the potential classes.
* As per the user journey, lets create classes.



* Each **class** in the diagram corresponds to a **table** in the **database**.
* Essential attributes for **database tables**:
  + ID column (Primary Key)
  + Other necessary attributes (e.g., Name, Address, etc.)

##### Identified Core Classes

* City
  + Attributes: ID, Name
  + Stores details of a city where theaters exist.
* Theater
  + Attributes: ID, Name, Address, City ID (Foreign Key linking to City table)
  + Stores details of theaters in a city.
* Screen
  + Attributes: ID, Name, Features
  + Represents individual screens within a theater.
* Movie
  + Attributes: ID, Name, Genre, Language, Duration, Features
  + Represents movies playing in theaters.
* Show
  + Attributes: ID, Movie Name, Status
  + Represents scheduled showtimes for movies.
* Seat
  + Attributes: ID, Type, Price, Status (AVAILABLE, BOOKED)
  + Represents seats in a Screen.
* Payment
  + Attributes: Transaction ID, Amount, Status (CONFIRMED, CANCELLED)
  + Stores payment details.
* Ticket
  + Attributes: ID, Show, Seats, Amount, Status, Payment
  + Represents a booked ticket.

##### City-Theater Relationship and **Access Patterns**

* Two approaches to linking City and Theaters:
  + City stores a list of theaters (Redundant Data)
  + Theater stores City ID (Better Normalization)
* SQL Query for theaters in a city:

SELECT \* FROM THEATERS WHERE CITY\_ID = 'BANGALORE';

* + If city stores a list of theaters, it leads to redundancy.
  + If theater stores City ID, we can fetch theaters dynamically via queries.

##### What is an Access Pattern?

* An **Access Pattern** refers to the **way data is accessed, retrieved, and manipulated** in a system. It determines **how efficiently** data can be fetched based on the frequency and type of queries executed.
* In **database design and system architecture**, access patterns **influence schema design**, indexing, and caching strategies to optimize performance.

##### **Types of Access Patterns**

* 1. **Read-heavy Access Pattern**
  + The system performs frequent reads but fewer writes.
  + Example: **Fetching a list of theaters for a city multiple time**
  + Optimization:
    - Use **denormalization** (store redundant data for faster reads).
    - Apply **caching** (e.g., Redis, Memcached).
    - Use **indexes** to speed up queries.
  1. **Write-heavy Access Pattern**
  + The system frequently writes or updates data (fewer reads).
  + Example: **Real-time stock price updates in a trading system**
  + Optimization:
    - Use **partitioning** to distribute writes across multiple nodes.
    - Use **event-driven architecture** (Kafka, RabbitMQ).
    - Employ **batch processing** to reduce database load.
  1. **Search-based Access Pattern**
  + Data is accessed based on dynamic user input.
  + Example: **Searching for movies by genre, language, or rating**
  + Optimization:
    - Use **inverted indexes** (Elasticsearch, Solr).
    - Implement **full-text search** instead of SQL queries.
  1. **Time-based Access Pattern**
  + Users access recent data more frequently than old data.
  + Example: **Fetching recent transactions in a banking system**
  + Optimization:
    - Use **Time Series Databases** (InfluxDB, TimescaleDB).
    - Implement **data archiving strategies**.

##### **Access Patterns in the BookMyShow Example**

In BMS, the **City-Theater relationship** was discussed in terms of access patterns.

* **Frequent Query**: "List all theaters for a selected city."
* Two possible approaches:

1. **Normalized Schema**: Store only CITY\_ID in the THEATER table and query it dynamically.

SELECT \* FROM THEATERS WHERE CITY\_ID = 'BANGALORE';

* + - **Pros**: Avoids redundancy, ensures data consistency.
    - **Cons**: Requires query execution every time.

1. **Denormalized Schema**: Store a **list of theaters** inside the CITY table for **faster access**.
   * + **Pros**: Quick retrieval without additional queries.
     + **Cons**: Increases storage, may cause **data inconsistency**.

* **Decision depends on how often the query is executed (Access Pattern).** If users frequently request a list of theaters for a city, **denormalization** may be preferable.

### The City class and Theater class

* As per the above, we have class City and class Theater as shown below…

|  |
| --- |
| export module City;  import <string>;  export class City {  int id;  std::string name;  public:  City(int id, std::string name);  int getId() const;  void setId(int id);  const std::string& getName() const;  void setName(const std::string& name);  }; |

|  |
| --- |
| export module Theater;  import Screen;  import City;  import <vector>;  import <string>;  import <memory>;  export class Theater  {  int id;  std::string name;  std::string address;  std::vector<std::unique\_ptr<Screen>> screens;  City city;  public:  Theater(int id, std::string name, std::string address);  void addScreen(std::unique\_ptr<Screen> Screen);  int getId() const;  …  …  }; |

### The Screen class

* The Screen class has following…
  + Id attribute: which is used to map the corresponding table in the database…
  + Name: Name of the screen (Example: Screen 1, Screen 2…)
  + List of Features: Type of the Screen which is 2D, 3D, IMAX, Dolby…

|  |
| --- |
| export module FeatureType;  export enum class FeatureType {  TwoD,  ThreeD,  IMAX,  Dolby  }; |

* + List of seats:

|  |
| --- |
| export module Screen;  import FeatureType;  import Seat;  import <vector>;  import <string>;  import <memory>;  export class Screen  {  int id;  std::string name; // Screen 1, Screen 2  std::vector<FeatureType> features;  std::vector<Seat> seats;  public:  Screen(int id, std::string name);  int getId() const;  …  …  }; |

### The Seat class

* The Seat class has the following…
  + Id attribute: which is used to map the corresponding table in the database…
  + Row: In which row this seat is present.
  + Column: In which column this seat is present.
  + SeatType: Type of the Seat (VIP, Gold, Platinum, etc…)
  + Name: Name of the Seat (A1, B14, etc…)

|  |
| --- |
| export module SeatType;  export enum class SeatType {  VIP,  Gold,  Platinum,  Regular,  Invalid, // For invalid seats or empty spaces  Walkway // For walkways or gaps  }; |

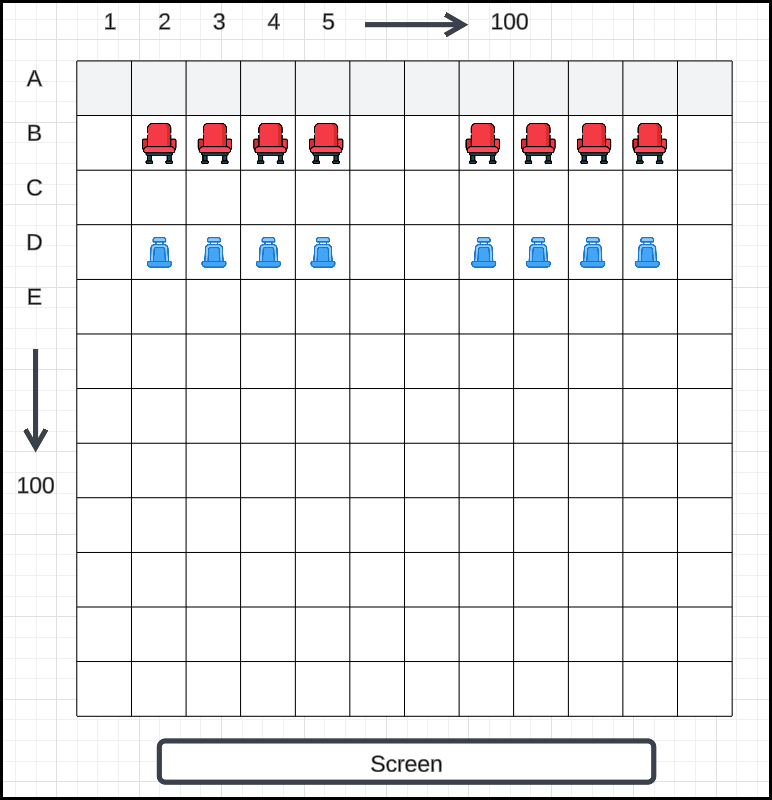
* + We can’t have Price attribute in the Seat class as it depends on the Show and Seat.
  + Similarly, we can’t have Status (Booked, Empty, etc…) attribute as it also depends on Show and Seat.

Understanding the Seating Layout

* A movie theater has a specific seating arrangement with different seat categories (VIP, Gold, Platinum, etc.).
* Each screen in a theater has a different layout.
* The layout includes gaps for walking, different seat sections, and labels for each seat.

Data Structure Used to Store Seats

* BookMyShow uses a predefined large 2D matrix (grid) of size 100x100.
* This grid serves as a universal seating template for all theaters.
* The actual layout is defined by marking certain positions as valid seats, while others remain empty.
* Example:
  + Valid seats are filled with seat identifiers (e.g., A1, B2, C3).
  + Empty spaces are ignored in display.



* Instead of a 2D matrix, for simplicity, we will be using a **list of valid seats**.
* This avoids storing empty spaces explicitly, saving memory.

|  |
| --- |
| export module Seat;  import SeatType;  import <string>;  export class Seat {  int id;  int row;  int column;  SeatType seatType;  std::string name; // A1, B13  public:  Seat(int id, int row, int column, SeatType seatType, std::string name);  // Getters  int getId() const;  …  …  }; |

### The Show class

* There is a mapping between Seat class and Show class as discussed above (Price and Seat Status).
* First, we implement the Show class and then we can discuss the mapping.
* The Show class the following…
  + Id attribute: which is used to map the corresponding table in the database…
  + startTime attribute: When the show starts.
  + endTime attribute: When the show ends.
  + Movie attribute: Which movie is being shown.
  + Screen attribute: On which screen current show is running.
  + Features attribute:
    - Movie **Features**: A movie has its own set of features (e.g., 2D, Dolby).
    - Screen **Features**: A screen has its own capabilities (e.g., supports 2D, 3D, Dolby).
    - Show Features:
      * The intersection of Movie and Screen features.
      * Example: If a movie supports 2D & Dolby, but the screen supports 2D, 3D, and Dolby → The show can only be 2D & Dolby.
  + Language attribute:
    - A movie can support multiple languages (e.g., English, Hindi, Punjabi).
    - A show will be in only one language.
    - Therefore, in the Movie class, we maintain a list of languages, while in the Show class, we specify only one language for a particular show.

|  |
| --- |
| export module Show;  import Movie;  import Screen;  import FeatureType;  import ShowStatus;  #include <chrono>  #include <memory>  export class Show {  private:  int id;  std::chrono::system\_clock::time\_point startTime;  std::chrono::system\_clock::time\_point endTime;  std::shared\_ptr<Movie> movie;  std::shared\_ptr<Screen> screen;  std::vector<FeatureType> features;  std::chrono::system\_clock::time\_point date;  ShowStatus status;  std::string language;  public:  Show(int id, std::chrono::system\_clock::time\_point startTime,  std::chrono::system\_clock::time\_point endTime,  std::shared\_ptr<Movie> movie, std::shared\_ptr<Screen> screen,  std::chrono::system\_clock::time\_point date, ShowStatus status,  const std::string& language);  int getId() const;  …  …  }; |

### The mapping class - ShowSeat

* A seat exists in a theater, but its availability, price, and booking status depend on the specific show.
* A single seat (e.g., Seat A1) can be part of multiple shows (e.g., 12 PM and 3 PM show).
  + The same seat in different shows may have different availability.
  + The price may change based on the time or demand for that show.
* Thus, a direct mapping between Show and Seat is required to track these variations, hence ShowSeat class.
* Attributes of ShowSeat:
  + Id attribute: which is used to map the corresponding table in the database…
  + Attribute show: refers to a show.
  + Attribute seat: refers to a seat.
  + Attribute price: Ticket price can vary depending on the show timing, promotions, or peak hours.
  + Attribute status: A seat’s status (Available, Booked, Locked) depends on the show.
    - The same seat may be booked for the 12 PM show but available for the 3 PM show.

|  |
| --- |
| export module ShowSeat;  import Show;  import Seat;  import SeatStatus;  export class ShowSeat {  private:  int id;  std::shared\_ptr<Show> show;  std::shared\_ptr<Seat> seat;  SeatStatus status;  public:  ShowSeat(int id, std::shared\_ptr<Show> show,  std::shared\_ptr<Seat> seat, SeatStatus status);  // Getters  int getId() const;  …  …  }; |

* Instead of storing the price in ShowSeat for each individual seat, we introduce ShowSeatType mapping.
  + Many seats can have the same price for a particular Show and for a particular SeatType.
  + For example, **all the** GOLD, SeatType for 1 PM Show has the same price.
* Since this is repetitive data in the database, we can store the price in a separate class/table ShowSeatType.

|  |
| --- |
| export module ShowSeatType;  import SeatType;  import Show;  export class ShowSeatType {  private:  int id;  std::shared\_ptr<Show> show;  SeatType seatType;  double price;  public:  ShowSeatType(int id, std::shared\_ptr<Show> show,  SeatType seatType, double price);  // Getters  int getId() const;  …  …  }; |

### The Movie class

* The Movie class should have the following attributes:
  + Id attribute: which is used to map the corresponding table in the database…
  + Name: Title of the movie.
  + List of Actors: Stores actors (both male & female) who acted in the movie.
  + Genre: Type of movie (e.g., Comedy, Action, Drama).
  + List of Languages: The languages in which the movie is available.
  + Duration: Runtime of the movie (can be stored as a double).
  + Rating: Movie rating (e.g., IMDb rating).
  + List of Features: Special features of the movie (e.g., 3D, IMAX).
    - Instead of separate enums for movie features, screen features, and show features, a single shared enum is used across classes.

|  |
| --- |
| export module Movie;  import FeatureType;  import Actor;  import MovieGenre;  #include <string>  #include <vector>  #include <chrono>  #include <memory>  export class Movie {  private:  int id;  std::string name;  std::vector<std::shared\_ptr<Actor>> actors;  std::vector<MovieGenre> genres;  std::vector<std::string> languages;  std::chrono::minutes duration; // Runtime in minutes  double rating; // Movie rating (e.g., IMDb rating)  std::vector<FeatureType> features;  public:  Movie(int id, const std::string& name,  const std::vector<MovieGenre>& genres,  const std::vector<std::string>& languages,  std::chrono::minutes duration, double rating);  void addActor(std::shared\_ptr<Actor> actor);  // Getters  int getId() const;  …  …  }; |

|  |
| --- |
| export module MovieGenre;  export enum class MovieGenre {  Action,  Comedy,  Drama,  SciFi,  Horror,  Thriller,  Romance,  Animation,  Documentary,  Fantasy,  Mystery,  Adventure,  Crime,  Musical,  Western,  Historical,  War,  Sport,  Family,  Other // For genres not explicitly listed  }; |

### The Actor class

|  |
| --- |
| export module Actor;  import <string>;  export class Actor {  private:  int id;  std::string name;  public:  Actor(int id, const std::string& name);  // Getters  int getId() const;  …  …  }; |

##### Finding Movies of an Actor

* To find all movies of an actor, we can query the **movies table**:

SELECT \* FROM MOVIES WHERE ACTOR\_ID = 'TOM CRUISE';

* Some may suggest storing a list of movies inside the Actor class, but this is not required unless frequent queries demand it.
* Decision depends on the access pattern:
  + If movie lists for an actor are frequently needed, caching them in the Actor class might be beneficial.
  + Otherwise, querying the movies table is sufficient.

### The Payment class

* Attributes of Payment Class
  + Id attribute: which is used to map the corresponding table in the database…
  + Amount: Amount paid for the ticket.
  + Status: Payment status, represented as an enum:
    - Successful
    - Failed
    - Refunded
  + Ticket: The ticket associated with this payment.
  + Transaction ID: The transaction ID associated with the 3rd party transaction.
  + Transaction Time: The time when the transaction happened.

|  |
| --- |
| export module Payment;  import PaymentStatus;  import <memory>;  import <chrono>;  export class Ticket; // To avoid Cyclic Dependency  // Instead of using import Ticket we use export statement  // to avoid cyclic dependency.  export class Payment {  private:  int id;  int transactionId;  double amount;  PaymentStatus status;  std::weak\_ptr<Ticket> ticket;  std::chrono::system\_clock::time\_point transactionTime;  public:  Payment(int id, int transactionId, double amount, PaymentStatus status,  std::weak\_ptr<Ticket> ticket,  std::chrono::system\_clock::time\_point transactionTime);  // Getters  int getId() const;  int getTransactionId() const;  …  …  }; |

|  |
| --- |
| export module PaymentStatus;  export enum class PaymentStatus {  SUCCESSFUL,  FAILED,  REFUNDED  }; |

### The Ticket class

* Attributes of Ticket Class
  + Id attribute: which is used to map the corresponding table in the database…
  + Show Details – Information about the show (Movie, Date, Time).
  + Show Seat Objects – The specific seats booked for the show.
  + Amount Paid – Total price paid for the ticket.
  + Status – Ticket status, represented as an enum:
    - Confirmed
    - Cancelled
    - Waiting
  + List of Payments – A ticket can have multiple payment objects (e.g., partial payments).
  + User Details – Information about the user who booked the ticket.

|  |
| --- |
| export module Ticket;  import Show;  import ShowSeat;  import Payment;  import User;  import TicketStatus;  import <vector>;  import <memory>;  export class Ticket {  private:  int id;  std::shared\_ptr<Show> show;  std::vector<std::shared\_ptr<ShowSeat>> showSeats;  double amountPaid;  TicketStatus status;  std::vector<std::shared\_ptr<Payment>> payments;  std::shared\_ptr<User> bookingUser;  public:  Ticket(int id, std::shared\_ptr<Show> show,  const std::vector<std::shared\_ptr<ShowSeat>>& showSeats,  double amountPaid, TicketStatus status,  const std::vector<std::shared\_ptr<Payment>>& payments,  std::shared\_ptr<User> bookingUser);  // Getters  int getId() const;  …  …  }; |

|  |
| --- |
| export module TicketStatus;  export enum class TicketStatus {  CONFIRMED,  CANCELLED,  WAITING  }; |

### The User class

* This class handles information about the user who booked the ticket.
* Attributes of User Class
  + Id attribute: which is used to map the corresponding table in the database…
  + Name: Name of the user who has booked the ticket.
  + Email address: Email address of the user who has booked the ticket.
  + Phone number: Phone number of the user who has booked the ticket.

|  |
| --- |
| export module User;  import <string>;  export class User {  private:  int id;  std::string name;  std::string email;  std::string phoneNumber;  public:  User(int id, const std::string& name, const std::string& email,  const std::string& phoneNumber);  // Getters  int getId() const;  …  …  }; |

## Class Diagram

## Schema Design

### Schema Design Principles:

1. **Class to Table**: Every class in the class diagram corresponds to a table in the database.
2. **Cardinality Matters**: Cardinality (relationships) determines how tables connect.
   * **One-to-One**: Primary key of one table is added as a foreign key in the other table.
   * **One-to-Many/Many-to-One**: Primary key of the "one" side is added as a foreign key in the "many" side.
   * **Many-to-Many**: A mapping (junction) table is created with foreign keys from both original tables.
3. **Primitive vs. Non-Primitive Attributes**:
   * Primitive attributes (int, string, etc.) are directly added as columns.
   * **Non-primitive** attributes (objects of other classes) **require cardinality analysis**.

### Table Schema (**Based on Primitive Attributes**):

#### Cities Table

|  |  |  |  |
| --- | --- | --- | --- |
| city\_id | city\_name |  |  |
|  |  |  |  |

#### Theaters Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| theater\_id | name | address |  |  |
|  |  |  |  |  |

#### Screens Table

|  |  |  |  |
| --- | --- | --- | --- |
| screen\_id | screen\_name |  |  |
|  |  |  |  |

#### Seats Table

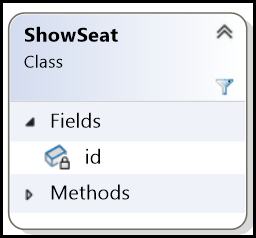
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| seat\_id | row | column | name |  |  |
|  |  |  |  |  |  |

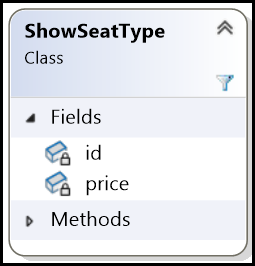
#### Shows Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| show\_id | start\_time | end\_time | date | language |  |  |
|  |  |  |  |  |  |  |

#### Show\_seats Table

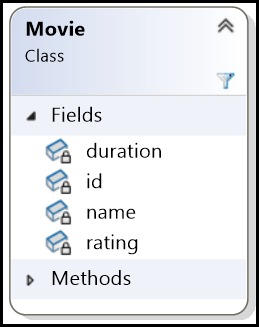
|  |  |  |
| --- | --- | --- |
| show\_seat\_id |  |  |
|  |  |  |





#### Show\_seat\_types Table

|  |  |  |  |
| --- | --- | --- | --- |
| show\_seat\_type\_id | price |  |  |
|  |  |  |  |

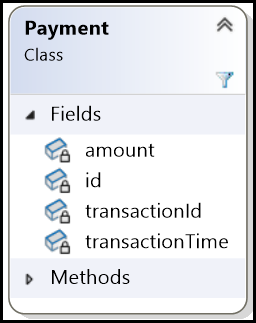


#### Movies Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| movie\_id | name | duration | rating |  |  |
|  |  |  |  |  |  |

#### Actors Table

|  |  |  |  |
| --- | --- | --- | --- |
| actor\_id | name |  |  |
|  |  |  |  |

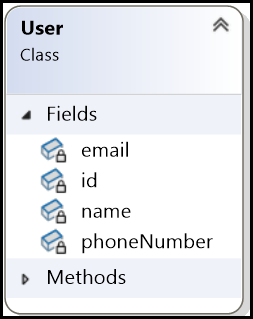


#### Payments Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| payment\_id | transaction\_id | amount | transaction\_time |  |  |
|  |  |  |  |  |  |

#### Tickets Table

|  |  |  |  |
| --- | --- | --- | --- |
| ticket\_id | amount |  |  |
|  |  |  |  |



#### Users Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| user\_id | name | email | phone\_number |  |  |
|  |  |  |  |  |  |

* All enum classes will have the same type of table structure, consisting of an id column (typically an integer primary key) and a value column (typically a VARCHAR to store the enum's string representation).

#### Features Table

|  |  |
| --- | --- |
| feature\_id | feature\_type |
|  |  |

#### Movie\_genres Table

|  |  |
| --- | --- |
| genre\_id | genre\_name |
|  |  |

#### Payment\_status

|  |  |
| --- | --- |
| payment\_status\_id | payment\_status\_name |
|  |  |

#### Seat\_status

|  |  |
| --- | --- |
| seat\_status\_id | seat\_status\_name |
|  |  |

#### Seat\_types

|  |  |
| --- | --- |
| seat\_type\_id | seat\_type\_name |
|  |  |

|  |  |
| --- | --- |
| show\_status\_id | show\_status\_name |
|  |  |

#### Show\_status

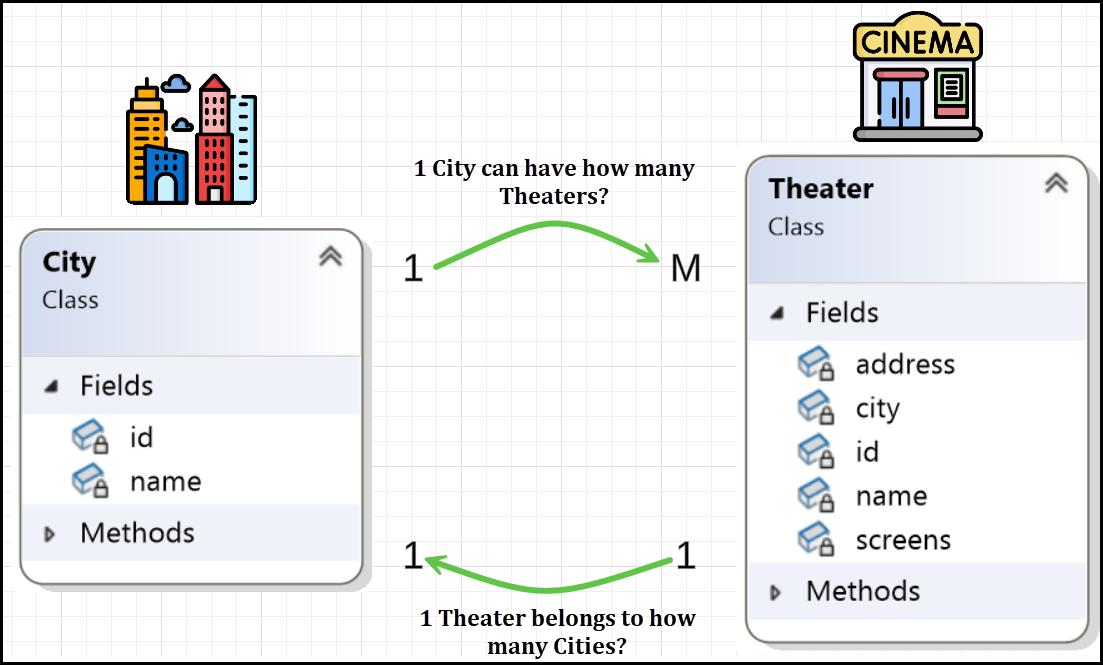
#### Ticket\_statuses

|  |  |
| --- | --- |
| ticket\_status\_id | ticket\_status\_name |
|  |  |

### Cardinality and Foreign Keys:

* Let’s consider all the classes again and find the Cardinality between these classes.

1. City class and Theater class



* ID of ‘1’ side on ‘M’ side.
* So, Theaters table should have Cities table ID column. Let’s add it to Theaters table.

|  |
| --- |
| CREATE TABLE CITIES (      CITY\_ID INT PRIMARY KEY,      CITY\_NAME VARCHAR(255)  ); |
| CREATE TABLE THEATERS (      THEATER\_ID INT PRIMARY KEY,      NAME VARCHAR(255),      ADDRESS VARCHAR(255),      CITY\_ID INT,      FOREIGN KEY (CITY\_ID) REFERENCES CITIES(CITY\_ID)  ); |

* Cities Table

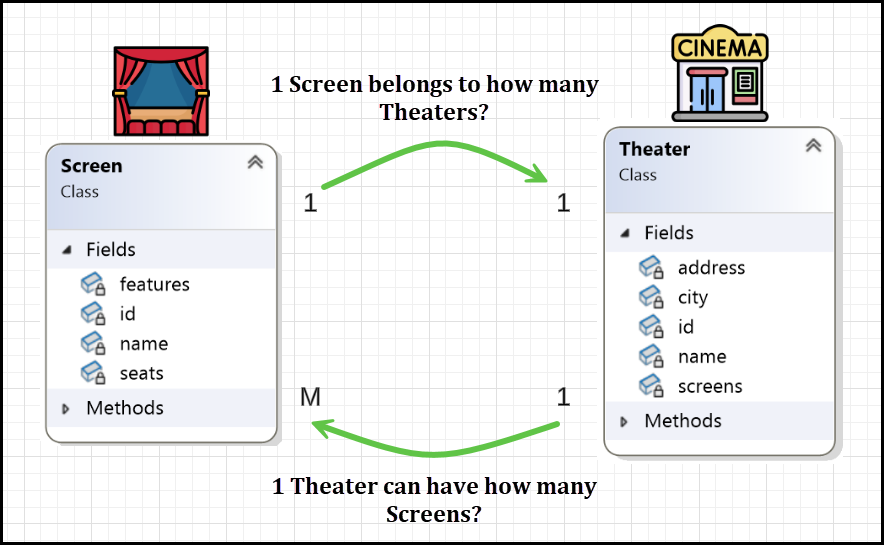
|  |  |
| --- | --- |
| city\_id | city\_name |
|  |  |

* Theaters Table

|  |  |  |  |
| --- | --- | --- | --- |
| theater\_id | name | address | city\_id |
|  |  |  |  |

1. Theater class and Screen class

* Theater also contains non primitive attribute Screens.

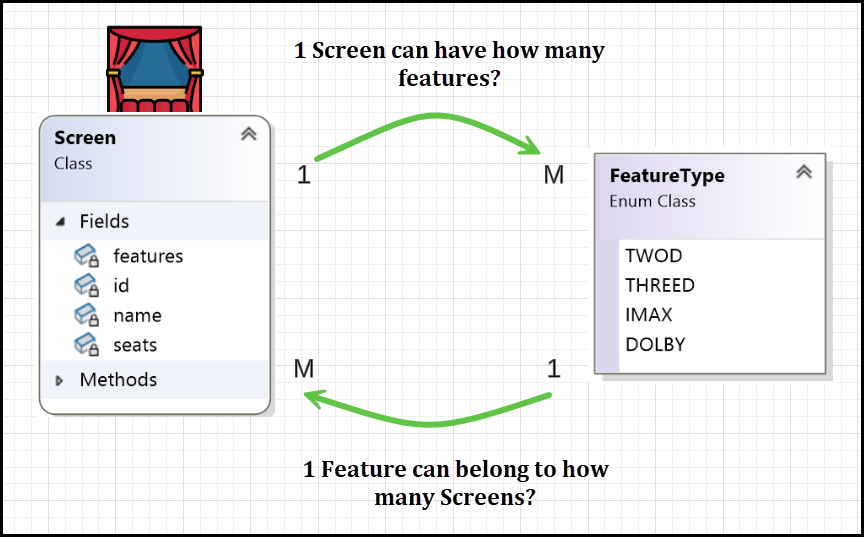


* ID of ‘1’ side on ‘M’ side.
* So, Screens table should have Theaters table ID column. Let’s add it to Screens table.
* Screens Table

|  |  |  |
| --- | --- | --- |
| screen\_id | screen\_name | theater\_id |
|  |  |  |

1. The Screen class and FeatureType enum

* Screen class also has the non-primitive attribute features.



* A Screen can have multiple features (e.g., 2D, 3D, IMAX, Dolby).
* A FeatureType can belong to multiple Screens. It is M:M relation.
* To represent this relationship, we use a mapping table (junction table). Create screen\_features table (Mapping Table)
  + Contains screen\_id and feature\_id as a composite primary key.
  + Establishes many-to-many mapping between screens and features.

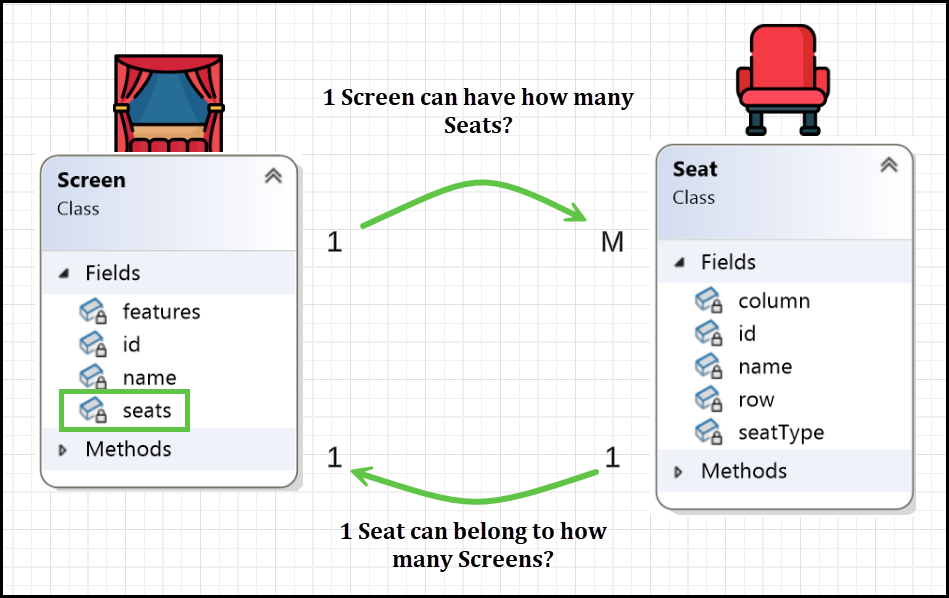
|  |
| --- |
| -- CREATE SCREENS TABLE  CREATE TABLE SCREENS (      SCREEN\_ID INT PRIMARY KEY,      SCREEN\_NAME VARCHAR(255),      THEATER\_ID INT,      FOREIGN KEY (THEATER\_ID) REFERENCES THEATERS(THEATER\_ID)  ); |
| -- CREATE FEATURES TABLE  CREATE TABLE FEATURES (      FEATURE\_ID INT PRIMARY KEY,      FEATURE\_NAME VARCHAR(255) UNIQUE  ); |
| -- CREATE MAPPING TABLE (JUNCTION TABLE)  CREATE TABLE SCREEN\_FEATURES (      SCREEN\_ID INT,      FEATURE\_ID INT,      PRIMARY KEY (SCREEN\_ID, FEATURE\_ID),      FOREIGN KEY (SCREEN\_ID) REFERENCES SCREENS(SCREEN\_ID),      FOREIGN KEY (FEATURE\_ID) REFERENCES FEATURES(FEATURE\_ID)  ); |

* The screen\_features table

|  |  |
| --- | --- |
| screen\_id | feature\_id |
|  |  |

1. The Screen class and Seat class

* The Screen class has non-primitive attribute seats.

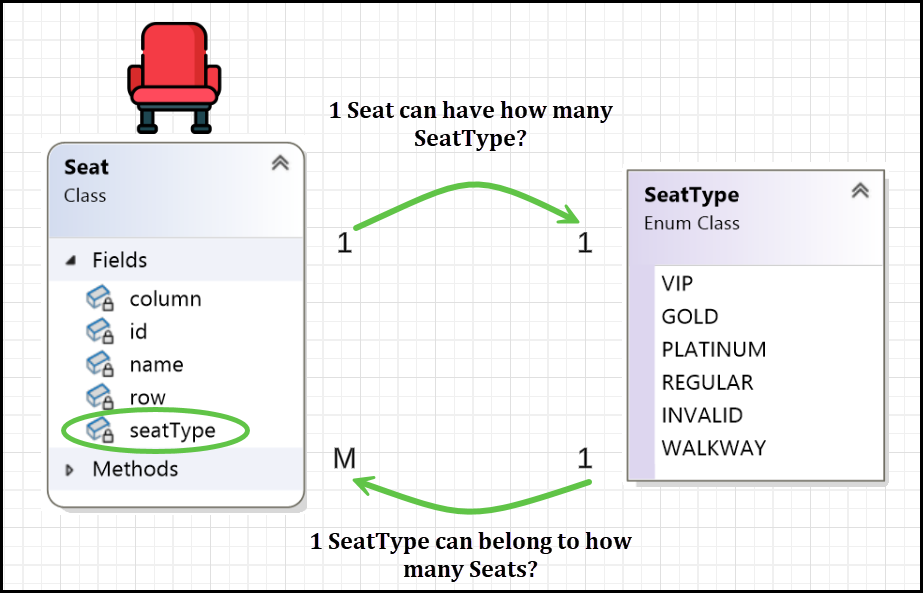


* ID of ‘1’ side on ‘M’ side.
* So, Seats table should have Screens table ID column. Let’s add it to Seats table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| seat\_id | row | column | name | screen\_id |
|  |  |  |  |  |

1. The Seat class and SeatType enum

* Seat class table has non-primitive attribute seatType.



* ID of ‘1’ side on ‘M’ side.
* So, Seats table should have Seat\_Types table ID column. Let’s add it to Seats table.

|  |
| --- |
| CREATE TABLE SEAT\_TYPES (      SEAT\_TYPE\_ID INT PRIMARY KEY,      SEAT\_TYPE\_NAME VARCHAR(50) UNIQUE  ); |
| CREATE TABLE SEATS (      SEAT\_ID INT PRIMARY KEY,      ROW VARCHAR(10),      COLUMN VARCHAR(10),      NAME VARCHAR(255),      SCREEN\_ID INT,      SEAT\_TYPE\_ID INT,      FOREIGN KEY (SCREEN\_ID) REFERENCES SCREENS(SCREEN\_ID),      FOREIGN KEY (SEAT\_TYPE\_ID) REFERENCES SEAT\_TYPES(SEAT\_TYPE\_ID)  ); |

* The SEAT\_TYPES table

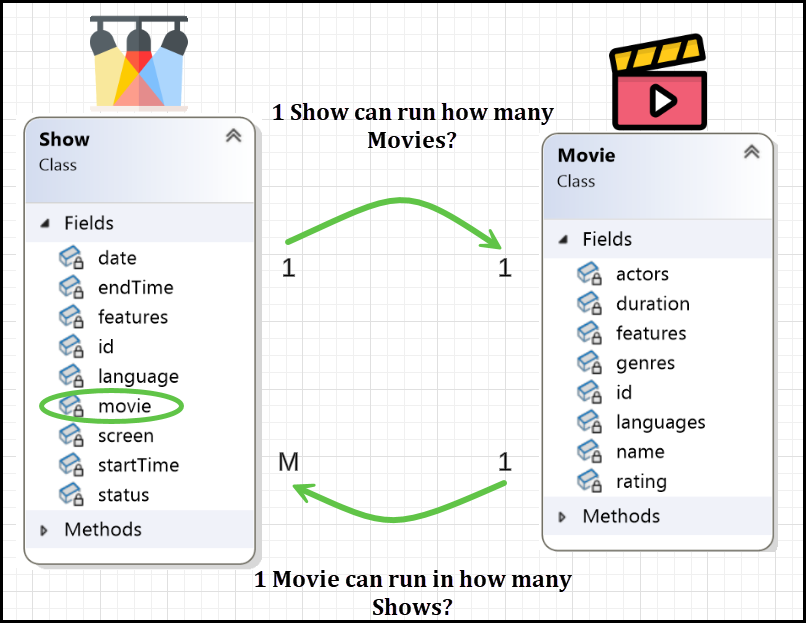
|  |  |
| --- | --- |
| seat\_type\_id | seat\_type\_name |
|  |  |

* The Seats table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| seat\_id | row | column | name | screen\_id | seat\_type\_id |
|  |  |  |  |  |  |

1. The Show class, Movie class, Screen class, ShowStatus enum and FeatureType enum

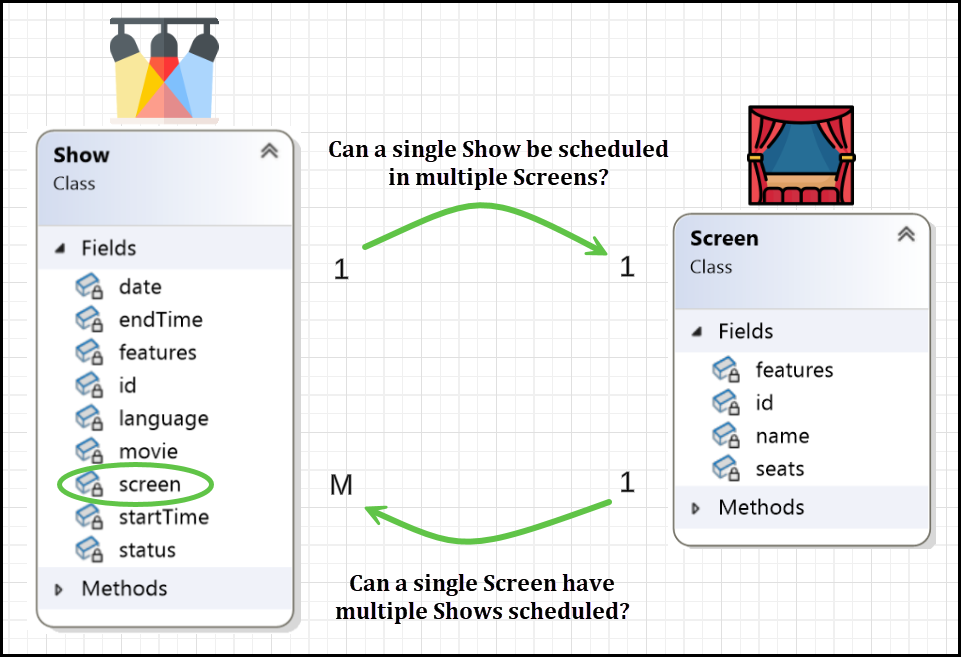
* The Show class has non-primitive attribute movie.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Shows’ table should have Movies table ID column. Let’s add it to Shows table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| show\_id | start\_time | end\_time | date | language | movie\_id |  |
|  |  |  |  |  |  |  |

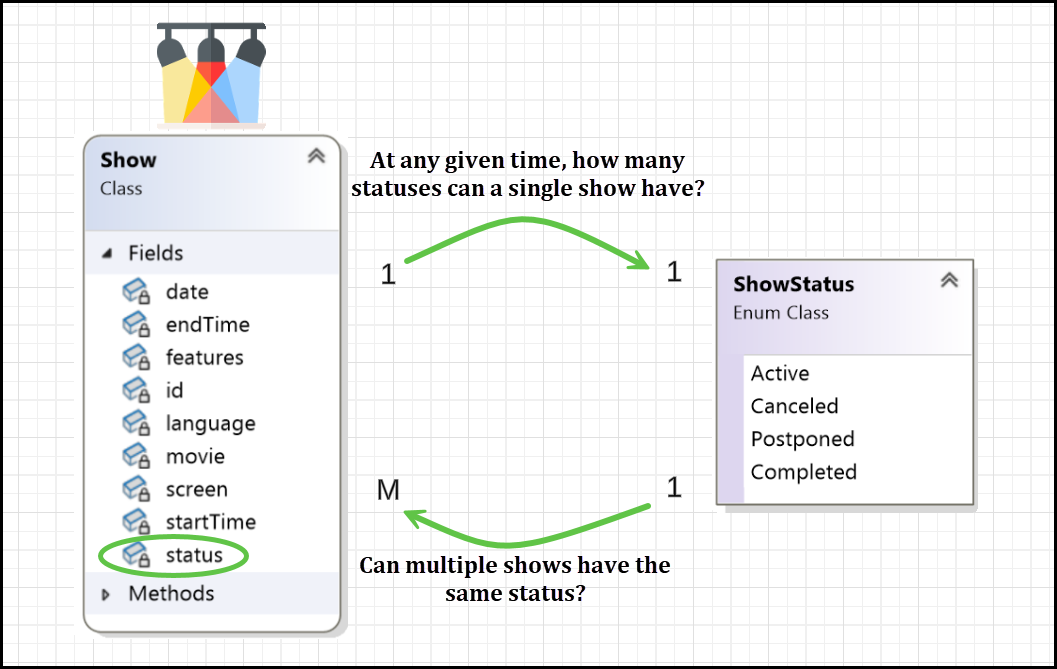
* The Show class also has non-primitive attribute screen.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Shows’ table should have Screens table ID column. Let’s add it to Shows table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| show\_id | start\_time | end\_time | date | language | movie\_id | Screen\_id |
|  |  |  |  |  |  |  |

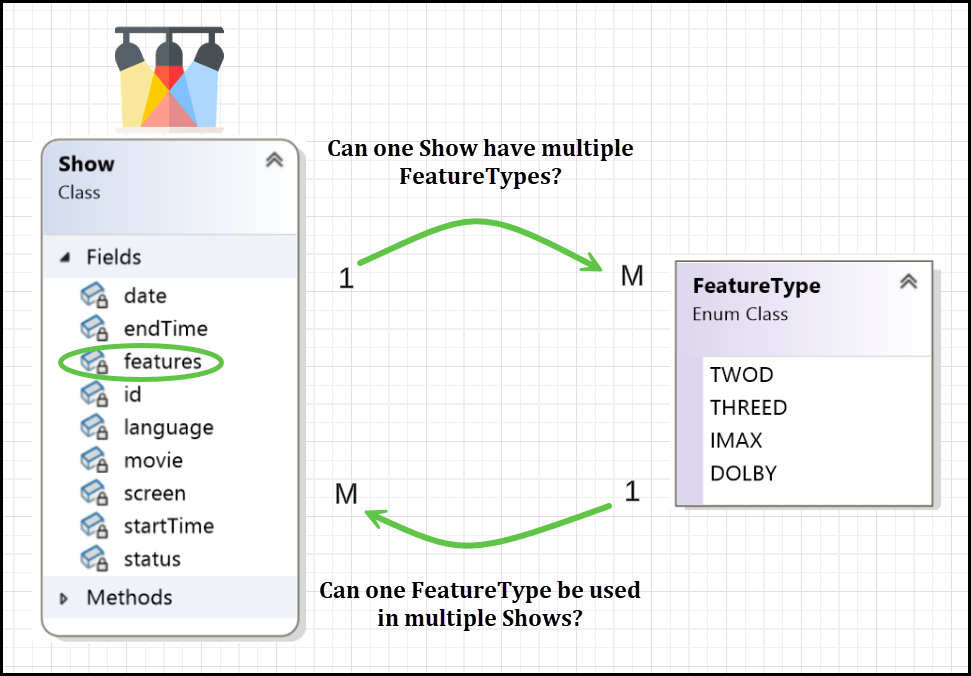
* The Show class also has non-primitive attribute status of type ShowStatus.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Shows’ table should have Show\_status table ID column. Let’s add it to Shows table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| show\_id | start\_time | end\_time | date | language | movie\_id | screen\_id | status\_id |
|  |  |  |  |  |  |  |  |

* The Show class also has non-primitive attribute features of type FeatureType.



* It is M:M relation. To represent this relationship, we use a mapping table (junction table). Create Show\_Features table (Mapping Table).
* We already have Features table. To create Show\_Features table, we need to have ‘Shows’ table first.
* Let’s create show complete show table…

|  |
| --- |
| CREATE TABLE SHOWS (      SHOW\_ID INT PRIMARY KEY,      START\_TIME DATETIME NOT NULL,      END\_TIME DATETIME NOT NULL,      DATE DATE NOT NULL,      LANGUAGE VARCHAR(50) NOT NULL,      MOVIE\_ID INT NOT NULL,      SCREEN\_ID INT NOT NULL,      STATUS\_ID INT NOT NULL,      FOREIGN KEY (MOVIE\_ID) REFERENCES MOVIES(MOVIE\_ID),      FOREIGN KEY (SCREEN\_ID) REFERENCES SCREENS(SCREEN\_ID),      FOREIGN KEY (STATUS\_ID) REFERENCES SHOWSTATUS(STATUS\_ID)  ); |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| show\_id | start\_time | end\_time | date | language | movie\_id | screen\_id | status\_id |
|  |  |  |  |  |  |  |  |

* Let’s create Show\_Features table

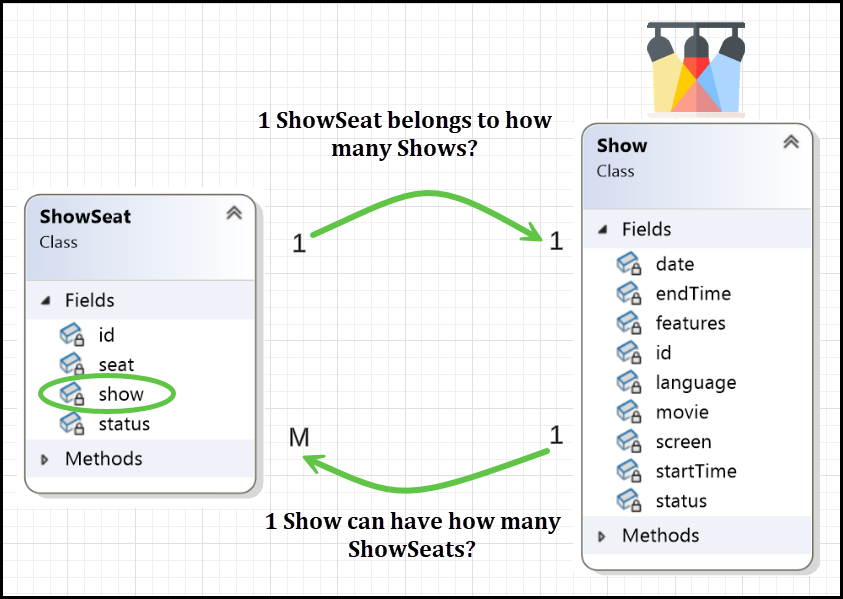
|  |
| --- |
| CREATE TABLE SHOW\_FEATURES (      SHOW\_ID INT,      FEATURE\_ID INT,      PRIMARY KEY (SHOW\_ID, FEATURE\_ID),      FOREIGN KEY (SHOW\_ID) REFERENCES SHOW(ID),      FOREIGN KEY (FEATURE\_ID) REFERENCES FEATURES(FEATURE\_ID)  ); |

* Show\_Features Table

|  |  |
| --- | --- |
| show\_id | feature\_id |
|  |  |

1. The ShowSeat class

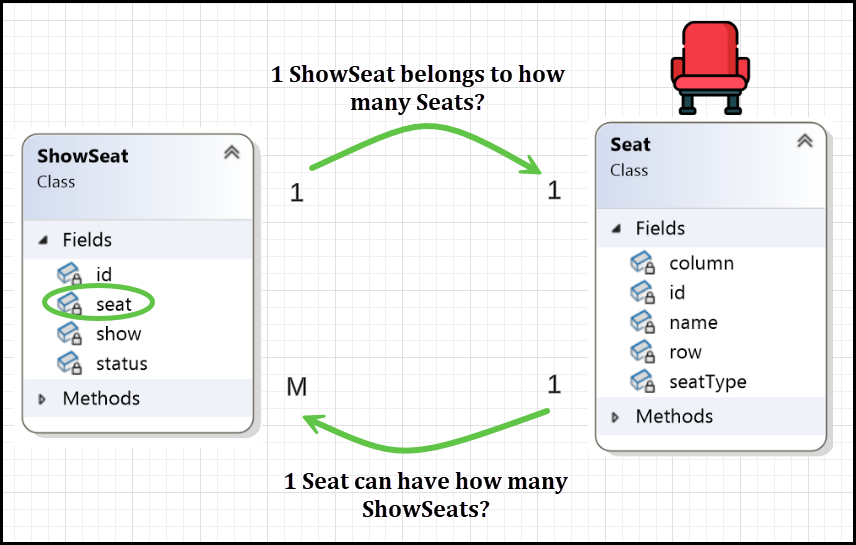
* We created this ShowSeat class, because
  + A seat exists in a theater, but its availability, price, and booking status depend on the specific show.
  + A single seat (e.g., Seat A1) can be part of multiple shows (e.g., 12 PM and 3 PM show).
    - The same seat in different shows may have different availability.
    - The price may change based on the time or demand for that show.
* Thus, a direct mapping between Show and Seat is required to track these variations, hence ShowSeat class.
* Let’s see the cardinality between ShowSeat class, and Show class.



* ID of ‘1’ side on ‘M’ side.

|  |  |  |
| --- | --- | --- |
| show\_seat\_id | show\_id |  |
|  |  |  |

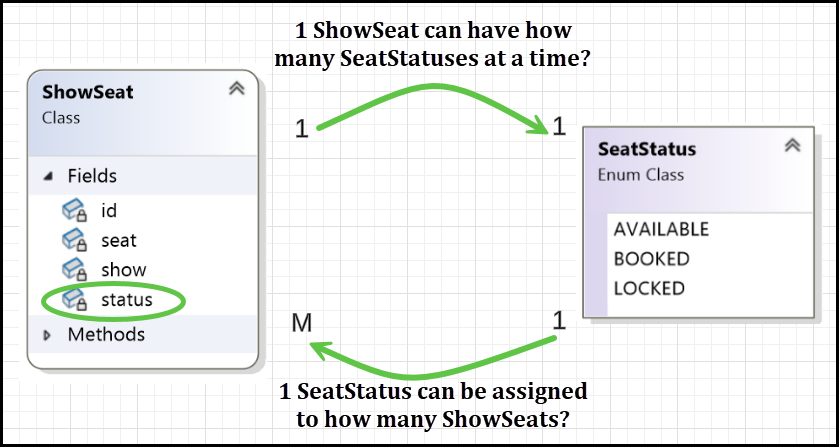
* So, ‘ShowSeats’ table should have Shows table ID column. Let’s add it to ShowSeats table.
* The ShowSeat class also has non-primitive seat attribute of type Seat.



* + One Seat can be part of multiple ShowSeats (because the same seat can be booked for different shows).
  + One ShowSeat is associated with exactly one Seat (each ShowSeat tracks a seat’s availability, price, and status for a specific show).
* ID of ‘1’ side on ‘M’ side.

|  |  |  |
| --- | --- | --- |
| show\_seat\_id | show\_id | seat\_id |
|  |  |  |

* So, ‘ShowSeats’ table should have Seats table ID column. Let’s add it to ShowSeats table.
* The ShowSeat class also has non-primitive status attribute of type SeatStatus.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Show\_Seats’ table should have Seat\_status table ID column. Let’s add it to Show\_Seats table.
* The Show\_Seats table

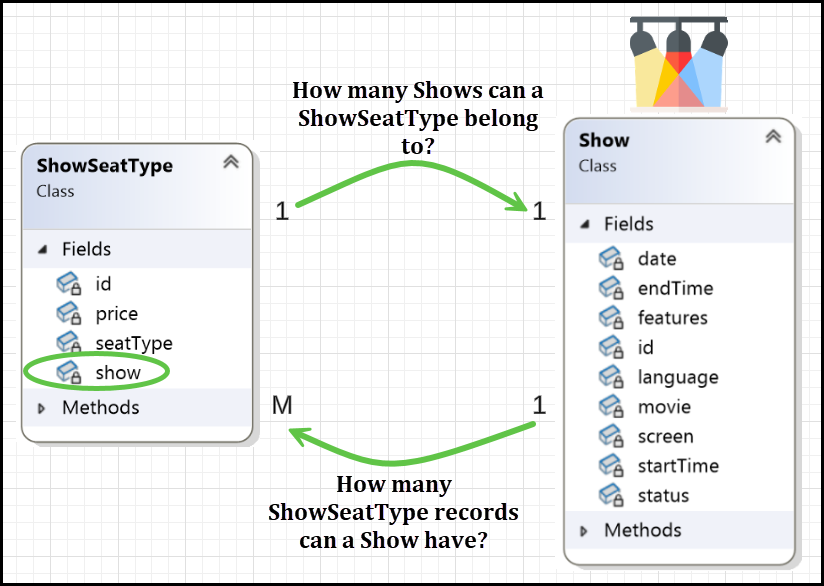
|  |  |  |  |
| --- | --- | --- | --- |
| show\_seat\_id | show\_id | seat\_id | seat\_status\_id |
|  |  |  |  |

|  |
| --- |
| CREATE TABLE SEAT\_STATUSES (      SEAT\_STATUS\_ID INT PRIMARY KEY,      STATUS\_NAME VARCHAR(20) UNIQUE NOT NULL  ); |

|  |
| --- |
| CREATE TABLE SHOW\_SEATS (      SHOW\_SEAT\_ID INT PRIMARY KEY,      SHOW\_ID INT,      SEAT\_ID INT,      SEAT\_STATUS\_ID INT,      FOREIGN KEY (SHOW\_ID) REFERENCES SHOWS(SHOW\_ID),      FOREIGN KEY (SEAT\_ID) REFERENCES SEATS(SEAT\_ID),      FOREIGN KEY (SEAT\_STATUS\_ID) REFERENCES SEAT\_STATUSES(SEAT\_STATUS\_ID)  ); |

1. The ShowSeatType class, Show class and SeatType enum

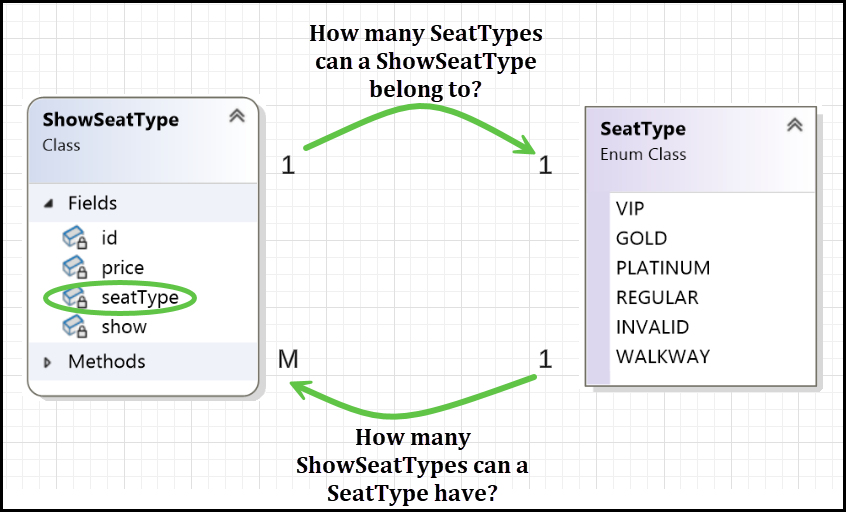
* The ShowSeatType class was introduced to improve data organization and optimize storage in the database. Here's why:
  + **Avoid Redundancy**: Instead of repeatedly storing the same price for multiple seats with the same SeatType (e.g., all "GOLD" seats for a specific show), the price is stored just once in the ShowSeatType table. This reduces data duplication.
  + **Better Maintainability**: If the price for a specific SeatType (e.g., "GOLD") changes for a show, you only need to update the price in one place, making maintenance easier and less error-prone.
  + **Improved Query Performance**: With a separate ShowSeatType class, lookups and updates for seat prices are more efficient since data is centralized in one table rather than scattered across multiple rows in the ShowSeat table.
* The ShowSeatType class has non-primitive attribute show of type Show.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Show\_Seat\_Types’ table should have Shows table ID column. Let’s add it to Show\_Seat\_Types table.
* The Show\_Seat\_Types table

|  |  |  |
| --- | --- | --- |
| show\_seat\_type\_id | price | show\_id |
|  |  |  |

* The ShowSeatType class also has non-primitive attribute seatType of type SeatType.



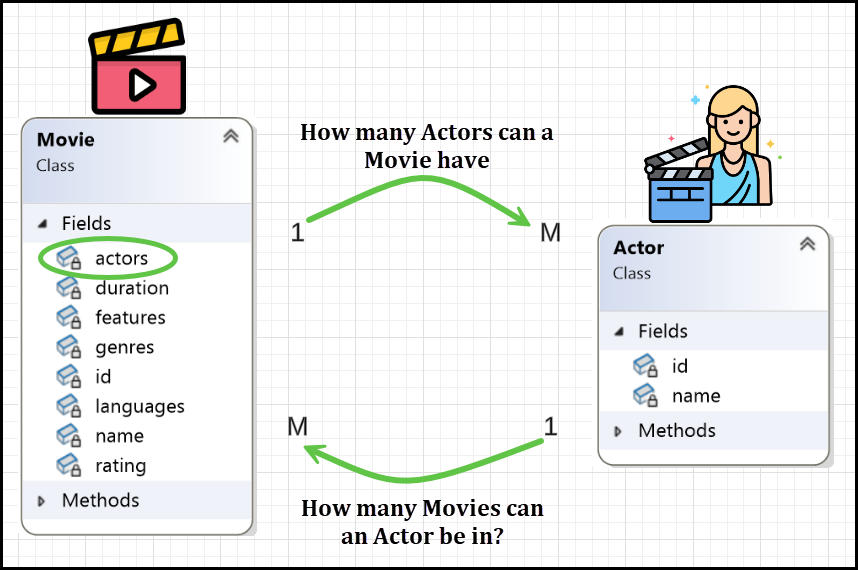
* ID of ‘1’ side on ‘M’ side.
* So, ‘Show\_Seat\_Types’ table should have Seat\_Types table ID column. Let’s add it to Show\_Seat\_Types table.
* The Show\_Seat\_Types table

|  |  |  |  |
| --- | --- | --- | --- |
| show\_seat\_type\_id | price | show\_id | seat\_type\_id |
|  |  |  |  |

|  |
| --- |
| CREATE TABLE SHOW\_SEAT\_TYPES (      SHOW\_SEAT\_TYPE\_ID INT PRIMARY KEY,      PRICE DECIMAL(10, 2),      SHOW\_ID INT,      SEAT\_TYPE\_ID INT,      FOREIGN KEY (SHOW\_ID) REFERENCES SHOWS(SHOW\_ID),      FOREIGN KEY (SEAT\_TYPE\_ID) REFERENCES SEAT\_TYPES(SEAT\_TYPE\_ID)  ); |

1. The Movie class, Actor class and FeatureType enum

* The Movie class also has non-primitive attribute actors of type Actor.

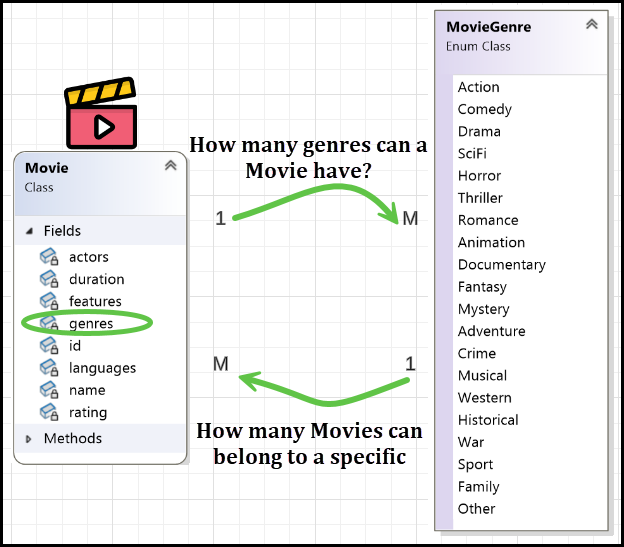


* A **single movie** can have **many actors**, and a **single actor** can act in **many movies**.
* Thus, this is a many-to-many (**M:M**) relationship, requiring a mapping table.
* The Movie\_Actors table

|  |  |
| --- | --- |
| movie\_id | actor\_id |
|  |  |

|  |
| --- |
| CREATE TABLE MOVIE\_ACTORS (      MOVIE\_ID INT NOT NULL,      ACTOR\_ID INT NOT NULL,      PRIMARY KEY (MOVIE\_ID, ACTOR\_ID),      FOREIGN KEY (MOVIE\_ID) REFERENCES MOVIES(ID),      FOREIGN KEY (ACTOR\_ID) REFERENCES ACTORS(ID)  ); |

* The Movie class also has non-primitive attribute genres of type MovieGenre.

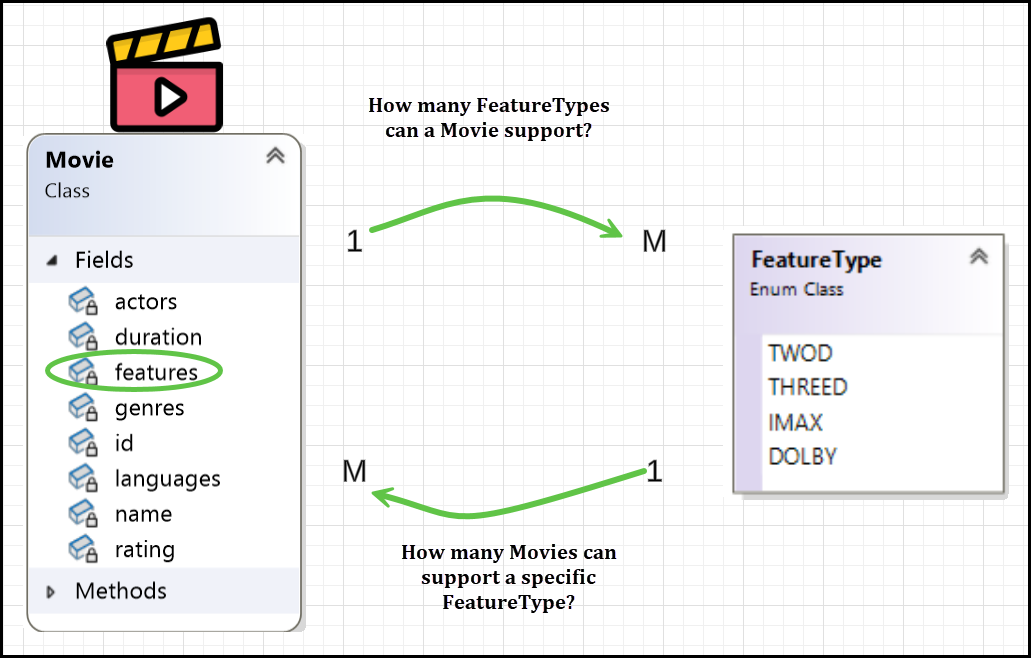


* A **single movie** can belong to **multiple genres** (e.g., Inception is **SciFi, Thriller, and Mystery**). A **single genre** can have **multiple movies** (e.g., SciFi includes Inception, Interstellar, The Matrix).
* Thus, this is a **many-to-many (M:M) relationship**, requiring a **mapping table**.
* The Movie\_Genres table

|  |  |
| --- | --- |
| movie\_id | genre\_id |
|  |  |

|  |
| --- |
| CREATE TABLE MOVIE\_GENRES (      MOVIE\_ID INT NOT NULL,      GENRE\_ID INT NOT NULL,      PRIMARY KEY (MOVIE\_ID, GENRE\_ID),      FOREIGN KEY (MOVIE\_ID) REFERENCES MOVIES(ID),      FOREIGN KEY (GENRE\_ID) REFERENCES MOVIEGENRES(ID)  ); |

* The Movie class also has non-primitive attribute features of type FeatureType.

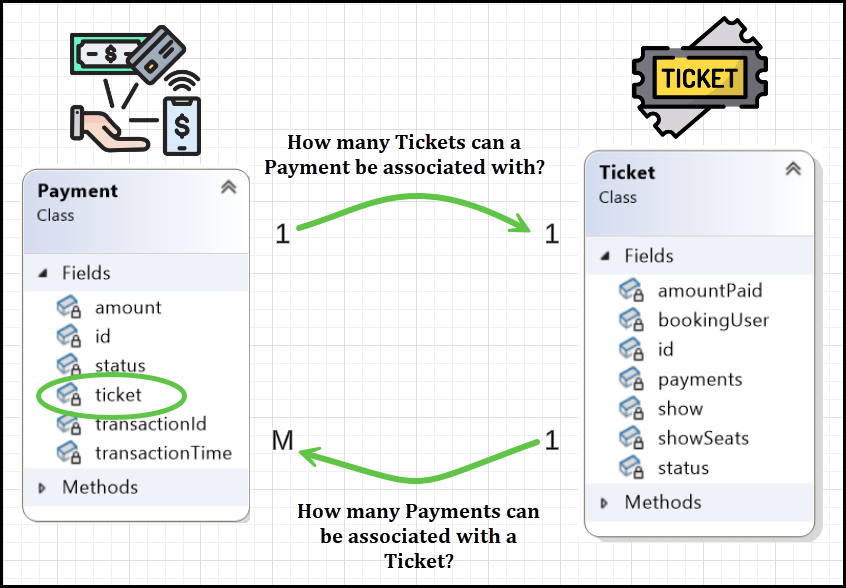


* A **single movie** can belong to **multiple genres** (e.g., Inception is **SciFi, Thriller, and Mystery**), and a **single genre** can have **multiple movies** (e.g., SciFi includes Inception, Interstellar, The Matrix).
* Thus, this is a **many-to-many (M:M) relationship**, requiring a **mapping table**.
* The Movie\_FeatureTypes table.

|  |  |
| --- | --- |
| movie\_id | feature\_type\_id |
|  |  |

1. The Payment class, Ticket class and PaymentStatus enum

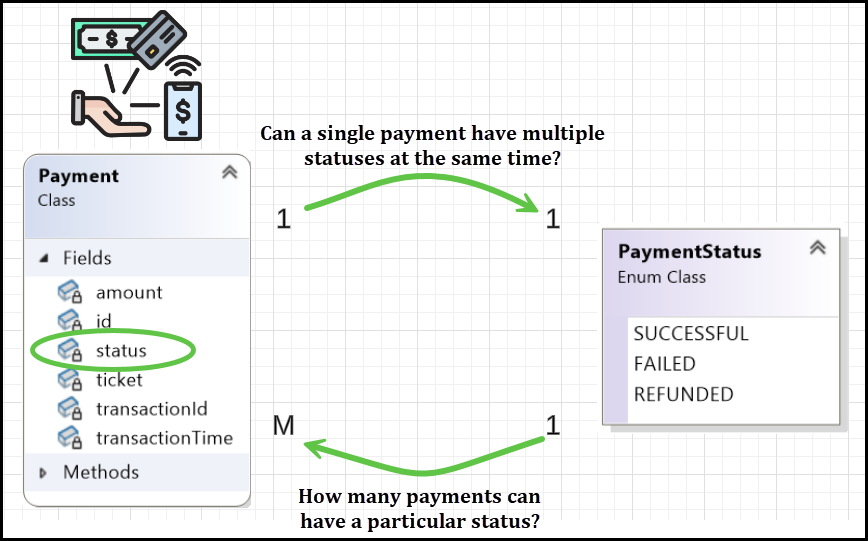
* The Payment class has non-primitive attribute ticket of type Ticket.



* A single ticket may be paid through **multiple partial payments**, **split payments**, or **retry attempts** (e.g., part credit card, part wallet).
* A single payment corresponds to **one ticket** only.
* ID of ‘1’ side on ‘M’ side.
* So, ‘Payments’ table should have Tickets table ID column. Let’s add it to Payments table.
* The Payments table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | transaction\_id | amount | transaction\_time | ticket\_id |
|  |  |  |  |  |

* The Payment class has non-primitive attribute status of type PaymentStatus.
  + One Payment can have one status.
  + One PaymentStatus (e.g., SUCCESSFUL) can apply to many payments.
* ID of ‘1’ side on ‘M’ side.
* So, ‘Payments’ table should have Payment\_statuses table ID column. Let’s add it to Payments table.



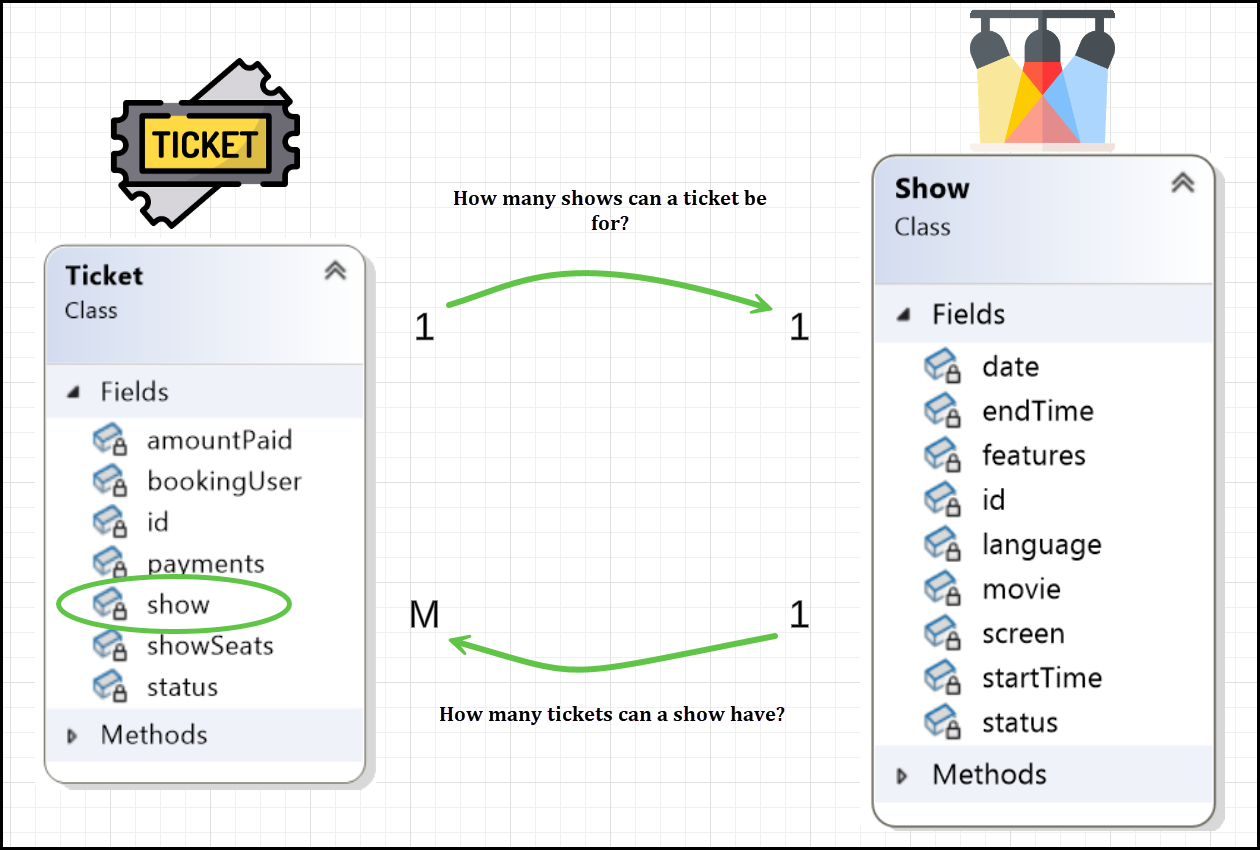
* The Payments table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | transaction\_id | amount | transaction\_time | ticket\_id | payment\_status\_id |
|  |  |  |  |  |  |

|  |
| --- |
| CREATE TABLE PAYMENT\_STATUSES(      PAYMENT\_STATUS\_ID INT PRIMARY KEY,      VALUE VARCHAR(255) UNIQUE  ); |
| CREATE TABLE PAYMENTS (      ID INT PRIMARY KEY,      TRANSACTION\_ID INT NOT NULL,      AMOUNT DECIMAL(10, 2) NOT NULL,      PAYMENT\_STATUS\_ID INT,      TICKET\_ID INT,      TRANSACTION\_TIME DATETIME NOT NULL,      FOREIGN KEY (PAYMENT\_STATUS\_ID) REFERENCES PAYMENT\_STATUSES (PAYMENT\_STATUS\_ID),      FOREIGN KEY (TICKET\_ID) REFERENCES TICKETS(ID)  ); |

1. The Ticket class, Show class, ShowSeat class, TicketStatus enum and User class

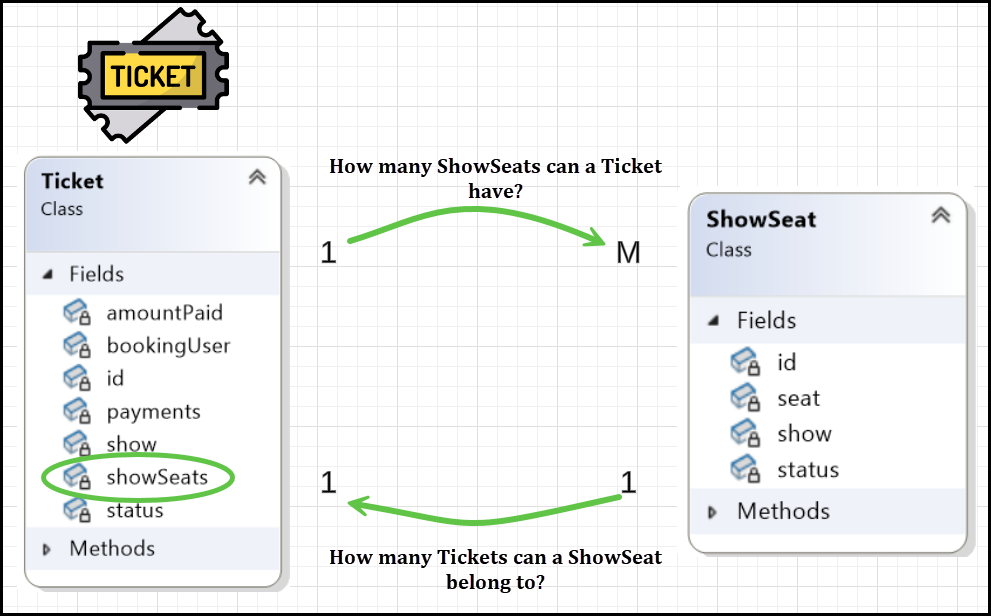
* The Ticket class has non-primitive attribute show of Show.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Tickets’ table should have ‘Shows’ table ID column. Let’s add it to Tickets table.
* The Tickets table

|  |  |  |
| --- | --- | --- |
| id | amount\_paid | show\_id |
|  |  |  |

* The Ticket class has non-primitive attribute showSeats of ShowSeat.
  + Ticket → ShowSeat
    - A Ticket represents a booking.
    - One ticket can include multiple ShowSeats (e.g., A1, A2, A3).
    - So, the cardinality is: One Ticket → Many ShowSeats
  + ShowSeat → Ticket
    - Each ShowSeat can only be booked by one Ticket.
    - So, one ShowSeat belongs to at most one Ticket (or null if it's still available).
    - So, the cardinality is: One ShowSeats -> One Ticket

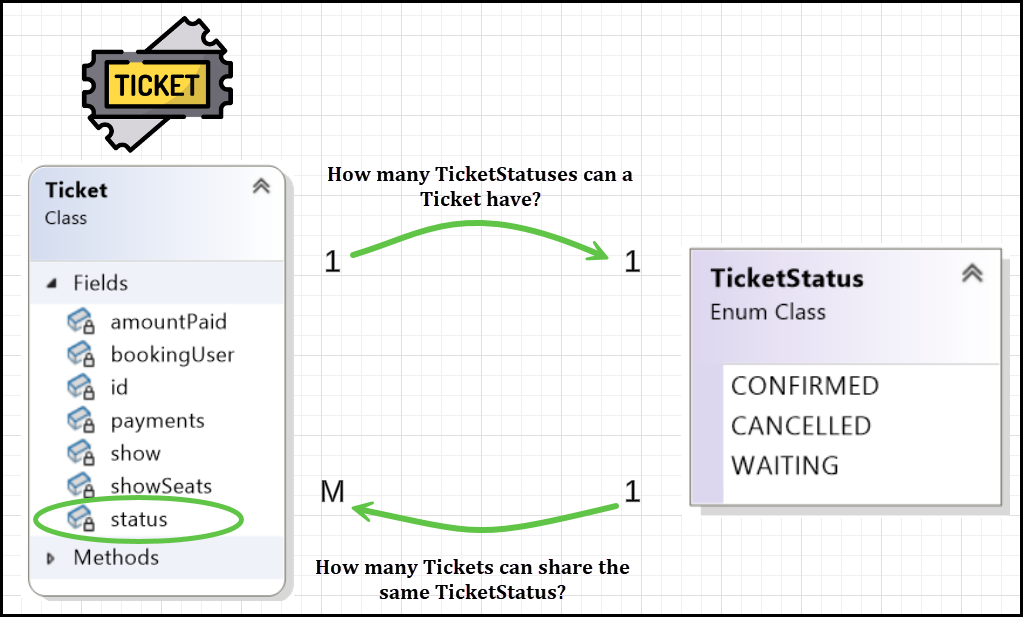


* ID of ‘1’ side on ‘M’ side.
* So, ‘ShowSeats’ table should have ‘Tickets’ table ID column. Let’s add it to ShowSeats table.
* The Show\_Seats Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| show\_seat\_id | show\_id | seat\_id | seat\_status\_id | ticket\_id |
|  |  |  |  |  |

|  |
| --- |
| CREATE TABLE SHOW\_SEATS (      SHOW\_SEAT\_ID INT PRIMARY KEY,      SHOW\_ID INT,      SEAT\_ID INT,      SEAT\_STATUS\_ID INT,      TICKET\_ID INT,  -- New column to reflect 1 Ticket : M ShowSeats      FOREIGN KEY (SHOW\_ID) REFERENCES SHOWS(SHOW\_ID),      FOREIGN KEY (SEAT\_ID) REFERENCES SEATS(SEAT\_ID),      FOREIGN KEY (SEAT\_STATUS\_ID) REFERENCES SEAT\_STATUSES(SEAT\_STATUS\_ID),      FOREIGN KEY (TICKET\_ID) REFERENCES TICKETS(TICKET\_ID)  ); |

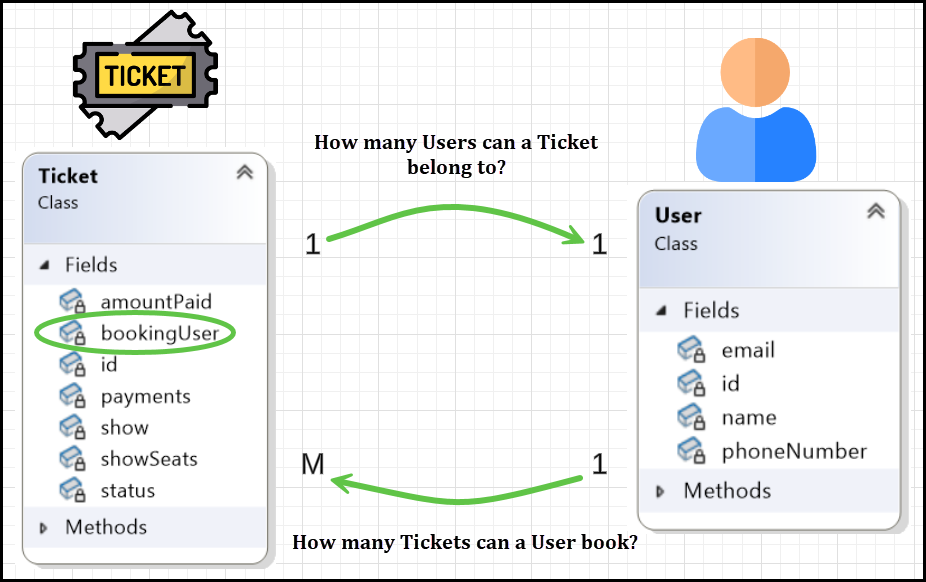
* The Ticket class has non-primitive attribute status of TicketStatus.



* Each Ticket has only one status: CONFIRMED, CANCELLED, or WAITING.
* Each TicketStatus can be associated with many tickets. For example, many tickets can be in the CONFIRMED status.
* ID of ‘1’ side on ‘M’ side.
* So, ‘Tickets’ table should have ‘TicketStatus’ table ID column. Let’s add it to Tickets table.
* The Tickets Table

|  |  |  |  |
| --- | --- | --- | --- |
| id | amount\_paid | show\_id | status\_id |
|  |  |  |  |

* The Ticket class has non-primitive attribute bookingUser of User.



* ID of ‘1’ side on ‘M’ side.
* So, ‘Tickets’ table should have ‘Users’ table ID column. Let’s add it to Tickets table.
* The Tickets Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | amount\_paid | show\_id | status\_id | booking\_user\_id |
|  |  |  |  |  |

|  |
| --- |
| CREATE TABLE TICKETS (      ID INT PRIMARY KEY,      SHOW\_ID INT NOT NULL,      AMOUNT\_PAID DOUBLE,      STATUS\_ID INT NOT NULL,      BOOKING\_USER\_ID INT NOT NULL,      FOREIGN KEY (SHOW\_ID) REFERENCES SHOWS(ID),      FOREIGN KEY (BOOKING\_USER\_ID) REFERENCES USERS(ID),      FOREIGN KEY (STATUS\_ID) REFERENCES TICKETSTATUS(ID)  ); |

## Model-View-Controller (MVC) + Service Layer + Repository Pattern

* Created a visual studio solution with the below mentioned projects and folder structure.

|  |
| --- |
| BookMyShow  ├───BMS.Common  │ ├───include  │ └───src  ├───BMS.Controller  │ ├───include  │ └───src  ├───BMS.DTO  │ ├───include  │ └───src  ├───BMS.Model  │ ├───include  │ └───src  ├───BMS.MySqlRepository  │ ├───include  │ └───src  ├───BMS.Repository  │ ├───include  ├───BMS.Service  │ ├───include  │ └───src  └───BMS.UI |

### Current Flow: (List all the movies)

1. UI Layer (BMSUI- View)

* The UI (Qt-based) contains buttons and user interaction elements.
* The **UI initializes the** MovieController with MovieService, which internally depends on MySqlMovieRepository.
* When a button (‘List All Movies’) is clicked, it calls the MovieController's method.
* **UI initializing the controller with the service** a good approach?
  + Yes, **for small to medium applications**, this is **a practical and perfectly acceptable approach**.
  + Better approach is to use Dependency Inversion.

|  |
| --- |
| int main(int argc, char\* argv[]) {  QApplication app(argc, argv);  auto repo = std::make\_shared<MySqlMovieRepository>();  auto service = std::make\_shared<MovieService>(repo);  auto controller = std::make\_shared<MovieController>(\*service);  BMSUI ui(controller); // Pass controller via constructor injection  ui.show();  return app.exec();  } |

1. Controller Layer (MovieController)

* The MovieController receives the UI request and delegates the call to the Service Layer (MovieService).
* It does not contain business logic—only delegates requests.

1. Service Layer (MovieService)

* The MovieService contains business logic.
* It calls the Repository Layer (IMovieRepository) to get data from the database.
* Converts Movie Model (database entity) to MovieDTO (Data Transfer Object) to avoid exposing raw database entities to the UI.
* All the logic of Price calculation, Applying discounts, Tax/fee computation, Validating booking happens here.

1. Repository Layer (MySqlMovieRepository)

* Responsible for fetching movies from MySQL using mysqlx API.
* Returns a std::vector<std::shared\_ptr<Movie>> to the Service Layer.

1. Data Layer (Database)

* MySQL database holds movie records.
* Repository fetches and maps them to Movie objects.

### **Final Summary**

| **Layer** | **Responsibility** |
| --- | --- |
| **UI (**BMSUI**)** | Handles user interactions, calls MovieController, displays data. |
| **Controller (**MovieController**)** | Delegates UI requests to MovieService. |
| **Service (**MovieService**)** | Business logic, calls IMovieRepository, converts Movie → MovieDTO. |
| **Repository (**MySqlMovieRepository**)** | Fetches data from MySQL and returns std::vector<Movie>. |
| **Database (MySQL)** | Stores movie records. |