Colosseo Ticketing Exercise

General information

This exercise aims to assess the ability to design and implement REST APIs. You're given a simplified database schema used for ticket sales. Your task is primarily to design and implement an API that allows the listing of events, displaying seats with their status (whether they're available or already sold), and creating new tickets with respect to concurrency – so that it won't be possible to have multiple tickets sold to the same seat at the same event.

Database schema

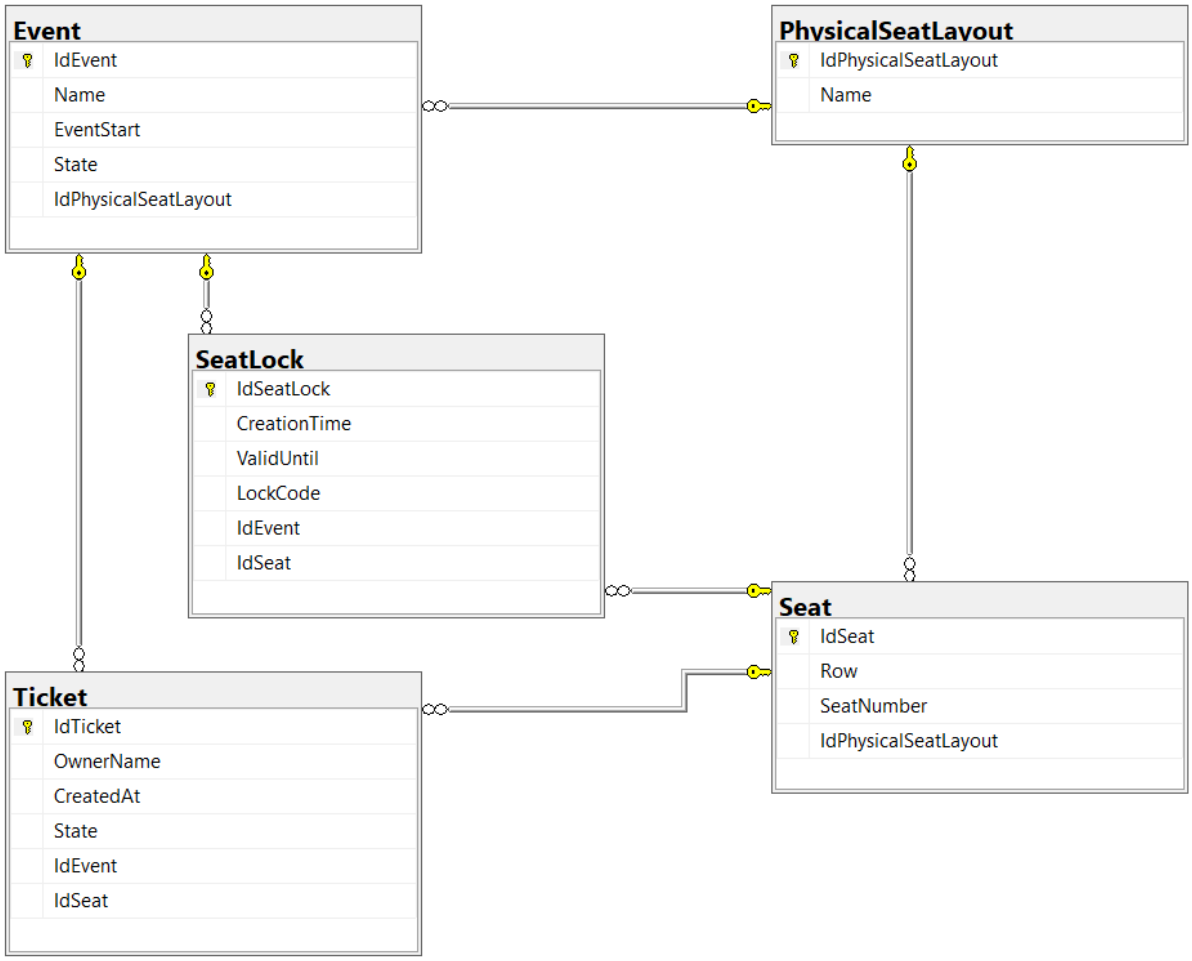

Table Event

This table represents planned events. Each event has a name, time of its planned start, and state. Available event states are defined in C# enum EventState (Available, Finished, Cancelled). Each event also has a seat layout, which determines which seats are available for the event. Multiple events can have the same seat layout.

Table PhysicalSeatLayout

Each row in this table represents a seat layout that can be used by events. Each seat layout has a name and can have many seats.

Table Seat

This table contains the list of defined seats. Each seat belongs to one seat layout and is identified uniquely by integer IdSeat. A more user-friendly identifier of seat is combination of string value Row and integer value SeatNumber, representing seat row and column of the seat.

Since each seat layout can be used by multiple events, also each seat can be used by multiple events.

Table Ticket

This table contains the list of sold tickets. Each ticket belongs to just one event and reserves just one seat for the event. Furthermore, a ticket has State property defined in C# enum TicketState (Valid, Cancelled). For every combination of seat and event, there can be only one ticket in valid state.

Additionally, each ticket should have its string property OwnerName filled in with the name of the user that bought the ticket and datetimeoffset property CreatedAt filled with the time at which the ticket was created.

Table SeatLock

This table is used to store temporary seat reservations for users. Seat lock is acquired from a time when a user selects a seat until the ticket is created, therefore allowing the system to guarantee that no two tickets are ever sold to the same event and seat simultaneously, even in concurrent usage of the service. The table has a unique index on columns IdEvent and IdSeat.

The ticket creation process should consist of these steps:

1. Delete old seat locks.
2. Try to create a seat lock to specified event and seat.
    a. If a seat lock could not be created, that means that another ticket sale process for the event and seat is already in process and the seat for this event is not available.
    The ticket creation process must stop now and result in failure.
3. Check whether no valid ticket exists for the event and seat specified.
    a. If a valid ticket already exists, that means another ticket sale process already finished for the event and seat.
    The ticket creation process must stop now, remove the created seat lock and result in failure.
4. Create a ticket with the specified event, seat, and other properties.
5. Remove the created seat lock.

Example of this process is implemented in method DatabaseTester.TicketInsertExample() in supplied PopulateDatabase tool.

Stored procedure DeleteOldSeatLocks

This stored procedure does not have any input parameters. It deletes all seat locks that have timed out (their ValidUntil property contains time in the past).

Table-valued function GetFreeSeatsForEvent

This function has integer input parameter @idEvent, and returns information about seats that are still available for the event with specified IdEvent.

Development environment

Database server

The simplest way to run a database server with this schema is to run it inside a docker container with MS SQL server 2019, using supplied docker-compose.yml file.

For this to work, you need to have Docker installed.

Next, you need to run the command “docker-compose up” in directory containing the docker-compose.yml file and an empty database server should start and be listening on host’s port 1450. Default user and password (specified in docker-compose.yml file) is “sa” and “SomeStrongP@ssword33”.

You can connect to the server with SQL Server Management Studio with server name “127.0.0.1, 1450”, SQL Server Authentication, and the aforementioned login credentials. There should be no databases present at this time.

Create a database with test data

The exercise package contains source code for project Colosseo.Exercises.Ticketing.Data that contains EntityFrameworkCore database context and all migrations to create the schema including stored procedures and functions.

There is also a project Colosseo.Exercises.Ticketing.PopulateDatabase, which is a console application that tries to connect to the database server using default credentials and create or update database to the specified schema, inserts test data to the tables and runs some simple tests to make sure that the database works correctly.

Database connection from the API

You may use the existing EntityFrameworkCore project Colosseo.Exercises.Ticketing.Data to connect to the database from your API project. The project is built with .NET 5, but you may upgrade it to a newer version of .NET if you wish.

Feel free to use another ORM instead of EntityFrameworkCore, just make sure that in the ticket creation process, deletion of old seat locks is done by calling DeleteOldSeatLocks stored procedure and that you get a list of free seats by calling stored function GetFreeSeatsForEvent.

Assignment specification

Primary task

Your task is to design and implement a REST API that will allow its clients to:

1. List upcoming events
2. Get all seats for the specified event
3. Get free seats for the specified event

4. Lock a seat for future ticket creation
5. Create a ticket
6. Cancel a ticket
7. Get all tickets for an event

The primary task is designing the API, therefore there are no design requirements specified, for example, functionality from points 2. and 3. can be achieved by a single request if you chose so.

The only non-functional requirements are that:

- The API uses supplied database schema as is.
- The API is implemented in C# language and using .NET 5 or higher

You may, but don't need to implement authentication or authorization logic.

You may run the API as a docker container, but it is not required.

Secondary task

The secondary task is to implement a sample client using the whole API. This may be a web application, a C# console application, or a C# WPF/WinForms application. It must demonstrate usage of the API as it is designed.

You don't need to allocate much time to designing the user interface. The main task is the API and this client application should just present how the API is meant to be used.