Task 2

Extend Ticket management application that was created in Task #1 with ability to

- Manage Events;
- Buy Tickets;
- See purchase history;

via Web User Interface based on ASP.NET CORE MVC template. User can have multiple roles (1 – 4). All user roles can act as anonymous user; (see Task #1 description).

Requirements:

1. Add ASP.NET CORE MVC project to solution.
2. Create purchase flow.
    a. User should have ability to navigate to event home page where he can see description of event and all related information (including tickets availability). Authenticated user can see layout map so he can click on area and select seats that he wants to buy.
    b. Authenticated user should have ability to see his purchase history.

    Create Event Management flow

    c. Manager can create Event with following options:
        i. Layout.
        ii. Show time.
        iii. Description.
        iv. Title.
        v. Image (Store URLs in DB).
    d. Manager can Update and Delete Events;
3. Add authentication and authorization using Identity;
    a. Anonymous user can create account and log in using credentials;
4. Application should support 3 languages (Russian, Belarusian and English), all inputs (DateTime pickers, inputs for price [We are using only USD for prices but delimiter can be different regarding culture], inputs for address etc.) should reflect culture specific formats;
5. Use Entity Framework Core:
    a. Use Store procedures for CUD operations on Event Entity; Fetch Data for Event using EF Core. Call Store Procedures using EF Core.
    b. Use EF Core for other entities;
    c. **Don't use navigation properties**;
    d. **Use .sqlproj as a source of the database schema (do not use EF migrations for creating database)**
6. Each authenticated user has his profile where he can modify:
    a. Time Zone;
    b. Language;
    c. First Name;
    d. Surname;
    e. Email;
    f. Password;
7. Authenticated User in his profile should see
    a. Purchase history;
    b. Balance (Amount of USD in account);

        i. User can add amount to his balance from profile;

8. Event Validation:
    a. Event published only when all areas in selected layout have price;
    b. Event can be deleted only when all seats are available to purchase (no one has purchased ticket yet);
    c. When layout on Event has changed old EventArea and EventSeat should be deleted and new created;
9. All new business logic should be covered with Unit Tests and Integration Tests (if required). Controllers should be covered with integration tests using test server: Integration tests in ASP.NET Core | Microsoft Docs
10. DI container should be used for creation of instances and dependency injection in your application. Chose any DI that you like.
11. Your database operations should be implemented in async way. Add Task type to public contracts where it is necessary.

Target framework requirement: NET 5.0

FAQ

Q. Should I remove the old repository implementation and DAL?
A. I would recommend not to remove that. If you've designed your system properly you just have to make an additional implementation of your interfaces based on EF DB context

Q. Old interfaces return IEnumerable or List for read methods. What should I do about it?
A. For the BL layer – keep that contract as it is. My recommendation is to expose the only Lists in public contracts in this case. For DAL layer: change return type of collection method to IQueryable. Inside your implementation, you can cast your collection via AsQueryable() method to remove compile error.

Q. Shall I implement a unit of work pattern or transactions?
A. In the scope of this task, it is not required. EF Core provides built-in UoW capabilities, also it will be harder to combine generic repository pattern and transactions without errors and caveats.

Q. Should I change already existing ADO.NET repositories into async operations?
A. I'd recommend that you do. This is pretty straightforward actually. You should do the following:
1) Change methods like connection.Open() or ExecuteCommands to their async counterparts (OpenAsync, ExecuteCommandAsync, etc.) and await all places where there are called.
2) Change public contracts for that methods and include Task in them (for example Event will be Task<Event>>)
3) await in all places where this code was called. Tests are also included.
The only one exclusion from this migration process will be GetAll-like methods. You should return IQueryable for the EF-core implementation thus you do not have anything to await. So in order to preserve the correct public contract, you have to keep synchronous implementation in ADO.NET implementation as well.
Asynchronous Programming - ADO.NET | Microsoft Docs