C# ASP.NET Core 5, MVC Web Application  
w/ Identity, DB: SQLite – “Code First Approach”   
(Forum /Message /Bulletin board)

**MVC (Model – View – Controller)** – design pattern: “Model” is an abstraction and or storage of data to be used, “View” is data presentation towards the User, and “Controller” is data manipulation, resource manager, and often the “middle-man” for the first two components’ management.

**Identity** – for registering and or managing User profiles with various roles or access levels in-system.

**SQLite** – a lightweight file based database system often used for prototyping as well as bundling and shipping it with small applications as portable minimalistic relational data storage components.

**Forum, Message /Bulletin board** – digital social system where Users can register then conduct discussions in a wide range of topics, share ideas and even publish their own stories for anyone to read. Users do not need to be registered or logged in to their user account to read anything.

There typically are so called “site rules” that the owners of the system expect all Users to adhere to, such as: don’t be vulgar, don’t outright insult anyone, don’t threaten people, be polite /don’t be rude, etc.

These may also entail “content limitations”, eg.: no explicit or graphic materials (imagery or text).

Finally the system and its users generally have to adhere to geographically local (at least relative to hosting server location) laws about digital media and online social systems & gatherings.

# Concept, source of idea

I often peruse and read discussions and even stories (works of fiction) on forums and message boards like “Space Battles”, “Sufficient Velocity”, “questionable Questing”, and “Archive of Our Own”.

Such as system is basically a relatively simple web application where every visitor can read content. Users can and have to register and log in for them to add new content. Certain Users have higher privileges (roles) than most others, these are:

* prime administration or “admins” who often own the system but sometimes just handle the technical aspects, and although keeping order among the Users is not among their tasks they have the privileges to do so; admins are also the only ones who can change Users’ roles: appoint moderators, demote them to “regular” Users and even give or transfer “administrator” roles;
* moderators or “mods” whose duty is being mediators and when necessary “digital police” to manage and keep Users behaving in an orderly fashion within the bounds of established rules and regulations; their role allows them to temporarily block Users from their accounts or just disallow them to add or modify their content, they also can temporarily or permanently close or lock down discussions, as well as “archive” and “hide-delete” them from regular User view.

# First steps: tools

1. SQLite and a database manager, eg.: DB Browser(SQLite)”
2. A development tool that supports C#, .NET and Microsoft Entity Framework development
   1. Microsoft Visual Studio, Community Edition 2019 – “VS 2019” – preferably)
   2. Microsoft Visual Studio Code – “VSC /VSC 2019” – alternatively

# Next steps (using VS 2019): new project & dependencies

1. Create a new solution /project
   1. Entity Framework Core, MVC Web Application
   2. Select “Individual User Accounts”
2. Install dependencies using “NuGet Package Manager” (or it’s console if preferred)
   1. Microsoft.EntityFrameworkcore.SQLite

# Finally: coding – “code first” approach

## **Database connection**

In the “appsettings.json” file modify the “ConnectionStrings” or add to it after a comma. After the “=” symbol the “**./**” means in the solution’s root folder where files like “Program.cs” & “Startup.cs” are. Next comes the desired name for the future database file with the proper extension, here “**.sqlite**”.



Finally, in the “**Startup.cs**” file (found in the solution’s root folder next to “Program.cs”), change the method “**ConfigureServices**” to contain the right connection functions pointing to the right path string.



## **Basic User registration & login**

Add the minimum necessary function to ensure users will be able to have roles, “**Identities**”.



In the root of the project, here named “*NC5MvcIdentitySqliteWebApp*” create a new folder called “*Migrations*” if there is none. Then perform the first (often repeated) steps to generate the database code. Open “Tools / NuGet Packet Manager / Package Manager Console” and apply these functions for *code-to-database* “**Migration**”. The resulting C# files will appear in the above created folder.

1. Add-Migration *[give a name here, usually “* Initial *” for the first since starting]*
2. Update-Database

It generates SQL-like code in C# to make the database & tables including constraints, then translates that into SQL code for the database type used (here, SQLite) to apply the same there as well, saving it in a file as previously specified in “appsettings.json”.

To undo (in reverse order) a migration (1-st) and apply the change to the DB too (2-nd), use these:

1. Update-Database *[last successful migration’s given name]*
2. Remove-Migration

# Next, there are options

First I made the C# models and the database (DB) for the application content.

However, people advise that when working with **Identity** (users and roles) it may be better to continue working on it until ready to connect “**User Roles**” to actions & pages they could access.

## **Making the content Entities and their Database form**

In the root of the solution, next to the “Migrations” made earlier, add a new folder, “*Entities*”. This will hold the abstraction entity models for the data storage we want to use for the content, as well as mark what kind of information to show Users. Call them “*[content name]*Entity”, like so: *“ BoardEntity* ”.

**However**, unless all tables are made and migrated to the DB at once, it is **highly** advised to add these “Entities” and their matching “Controllers” one at a time between migrations otherwise parts might end up missing or jumbled in undesired ways.









These are just the basic entity models, not yet containing their relation to each other.

Those are defined in “Data” found in the solution root folder, where the other two was just added.

1. **First** create a “**Configuration**” file (eg.: “ *BoardEntityConfiguration*.cs ”) for the SQL-like C# code.



1. **Then** add the necessary code into the “*ApplicationDbContext*.cs” file so the “builder” function knows what entities to use with what configurations in order to create and manage the DB and data to and from the DB.



In the above snippet I already have the code for all four added, but again, unless all is migrated at once add only one at a time paired with the corresponding “Entity” and “Configuration” between migrations.

**Configuration** for the “*ForumEntity*”:



**Configuration** for the “*ThreadEntity*” (this, for now also contains code I am not sure about):



**Configuration** for the “*PostEntity*” (this, for now also contains code I am not sure about):



It is a good idea to check the database with a manager tool such as “DB Browser(SQLite)” after every database update to see if everything was generated and transferred all right.

At this moment if we open the application (“Start with/-out debugging”) only the basic User Registration and Login sections are visible and in rudimentary but working condition.

Therefore, the next step is creating or generating (preferred) “**View**”-files.

## **Getting Views: board, forum, thread, post**

**Views** are stored in the folder of same name found in the solution root folder. Everything inside is a special so called “**Razor**” file with the “**.cshtml**” extension. It basically allows the embedding of C# code into HTML code, AND the substitution of nearly all JavaScript code with C# code.

**First** “view models” have to be made, they are just like “entity models” but instead of having fields that correspond to the database, they correspond to what the User should be presented AND allowed to manipulate – perform “**CRUD**” operations on (Create, Read, Update, Delete).

**Then** “scaffold” the model to generate views, preferably of “Razor” type, by right clicking on the “Controller” folder in the solution root, and select “**Add Controller**”. It will ask for the model among other parameters. WILL LATER ADD WHAT I SELECTED

If it fails once rebuild the solution and try again, sometimes that fixes whatever the issue is.

If all is successful the “Context” and “Views” will be generated.

## **Migrating Views to the DB**

**Make sure** there are controllers for all models before attempting to migrate them.

Then perform the previously shown migration and DB updating commands.

If all goes well the application can be started to see how it looks like. Just type a slash character (“ / ”) followed by the name of the model in the address right after the base URL already there. For example:

* If the base URL is: https://localhost:44375
* The new URL will be: <https://localhost:44375/Boards>

Note: after the localhost the number may differ, it is the port number the server uses, usually depends on the DB system.