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” and “Questionable Questing”.

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:

* administrators 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

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.

However, I chose to make the C# “MVC” and the database (DB) migration for the *content* first.

# Content Entity Context, Views & DB migration

In the solution root, next to the “Migrations” made earlier, add a new folder, **“*Entities*”** to hold the abstraction entity models for the data storage we will 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 it ends wrong.









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*”:



**Configuration** for the “*PostEntity*”:



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, by right clicking on the “Controller” folder in the solution root, and select “**Add Controller**”. From the offered list we need **“MVC Controller with views, using Entity Framework”**.

A controller scaffolding parameter window opens asking for:

* Model class the model we want to scaffold, eg.: **Board.cs**
* Data context class the class that tells how to scaffold what: **ApplicationDbContext.cs**
* Views
  + Generate views [x] auto-generated views are a great help.
  + Reference script libraries [ ] unless we use such, don’t.
  + Use a layout page [x] default: **~/Views/Shared/\_Layout.cshtml**
* Controller name give a fitting name, eg.: **BoardsController.cs**

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

IF it still fails, then one or more prerequisite files /code is missing or is incorrect: model, their matching configuration and the context that pairs them up. Check them, make sure that all model files have a configuration file and that they are paired in the context, otherwise at best it won’t work at worst the process ends with incorrect results.

**IF** that happens just delete the controller, correct what needs to be and redo the process.

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** there is a mistake like mentioned a step above, refer to the start on [how to undo migrations](#UndoMigration).

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.

# Continuing with Users – Roles & Authorization

The first step I performed at the very beginning, and ***now*** perform “Identity Scaffolding” by right clicking the project in the **Solution Explorer** then selecting **Add / Scaffold item** where we pick **Identity**.

After processing we are offered a lot of options to choose from, what we want the scaffolding to cover. For starters check just three: “**\Login**”, “**\Register**”, “**\Register Confirmation**”. If more is needed later we can perform this again to add more.

For the Data context class we select out only one: “**ApplicationDbContext.cs**”.

New items appear in the “Areas” root folder under “Identity / Pages / Account”.

## **User Role restrictions**

**Fallback Policy** – global restriction on all areas: access only if logged in (with certain Roles). So we restrict everyone from everywhere, so users can’t even access the *Home Page* without logging in. *Then* we allow some access for certain Roles to certain areas AND certain actions (eg. thread deletion only for Admins & Mods). Thus we can ensure that we do not accidentally miss and leave critical things open to all.

**First**, In “**Startup.cs**” inside “**ConfigureServices**” method; also add minimal “Cookie” handling.



**Second**, in the “Controllers” root folder’s “HomeController.cs” file we establish what roles or under what policies will certain areas or elements be created/ prepared for user access. In this case I have added an example for “Roles” and one for “Policy”. *The special pages* for site administration for *Admins* & *Mods*:



**Policies** are more versatile than Roles. In order for it to work we need to register the option in the root “**Startup.cs**” file inside the newest “**AddAuthorization**” service, we just added.



Now I create these pages by adding a templated “Razor View” file in the “View / Home” folder. We don’t make anything fancy so just enter the name, eg.: “AdminUI.cshtml” and “ModUI.cshtml”.

## **User Role migrations & first Seeding of special roles**

Now that we made some changes to how user authentication is handled we have to add **Roles** and then can enter the first users with special roles. It can be done directly in the DB or by C# code that is migrated and then with a DB update reflected in the database. Following the course, I chose the second, **“Code First”** approach.

So, first add a new “seeding” migration: Add-Migration RolesSeeding -verbose

I don’t have a model-context pair and have not made any modifications to the DB either, so the created migration file’s “Up” and “Down” (grading) methods are empty, allowing us to specify what exactly we want to update in the DB.

Before we update the DB, first we have to define what to do and how to do it during the update, such as adding the user roles as well as a couple new users which already possess some of those roles. In my case I added one Admin User and two Mod Users.

The newly generated migration file’s “Up” method, add the new entries /tuples for roles & users:



Filling out the “Down” method is unnecessary (at least for this application, there will never be less special roles than Admin and Moderator) even if the same person handles both “jobs”. For the removal of these specially introduced users the Admin can do it either directly in the DB (manager) tool, or (this is how it should be done) in-app on the appropriate “AdminUI” page dedicated for granting and revoking roles, including passing along the admin role to others.

For the coming section I tried following along the course instructions but because I was /am stuck with SQLite (for now) each MS SQL script that was used to add roles and some users to the DB then connect roles to those users … I had to transcribe all that to SQLite script … because SQLite is designed simpler but by comparison is dumber, for example there is no script function for generating new Id-s like “NEWID()”, in addition to everything being of type TEXT. I succeeded for some but not all. In the end I had to request Instructor assistance, and also separated the large code block into several smaller methods.

**This** is the base **“MigrationBuilderExtension.cs”** file, we put in the root “Data” folder.



**Inside** we put the various migration configuration, context & controller methods.

To start with, this is the definition on how to ass the new Roles.



**NOTE:** that there is a “TODO” notification marking that the “ActiveProvider” will have to be set & configured if this application is to be deployed. It is a module that verifies what DB type (MS SQL Server, SQLite, MySQL, PostgreS, etc.) is used and selects the appropriate, predefined SQL syntax to utilize, and if it does not exist then throw appropriate errors.

*((*

*At this point in time there is no need for this. It is not even meant for actual deployment but as a working, useable example of what I can do. An exercise and portfolio item.*

*Of course, it can be turned into a deployable application, but at inception it was not meant as such.*

*))*

**NOTE:** that formatting here has been a little “compressed” to fit less page space.

We use this to define how to add the new Users from code.





Finally, we define how to match the Users to the Roles, then the “master” method calls then as needed to create the new special users and also match them to their intended roles.



**NOTE:** as we see in the “AddUser” method it is so during login we ask for a registered E-mail address instead of a “User Name”. For two reasons: first, because many systems do it so it isn’t unusual, and second (and primarily) because the course showed this and there are more important aspects to handle.

***Finally*** we can update the DB: Update-Database -verbose

If we did everything correctly, we can check the DB manager to see that we have roles and some more users with those roles inside. We can also try out the application, just log in as anyone and type “/AdminUI” or “/ModUI” in the address bar after the base address. We can see that authorized users can access those pages but no one else can.

Now we add some access buttons on the top **“Nav-bar”** that are generated and thus visible / accessible only for the appropriate users. Open the root **“Views / Shared / \_Layout.cshtml”** file, and inside the **header** find the **div** of class **“navbar-collapse”**.

Inside the **“ ul ”** surround each **“ li ”** in *Razor code* syntax conditions based on *User Identity* or *Role* that will ensure that unauthorized users won’t even get the element generated on the page they get.

Here is an example for the “Home” button.



Now let’s add a new button each for the two new pages: *AdminUI* and *ModUI*.



**NOTE:** the code omission marked with three dots; don’t forget to fill them out normally!

## **Exceptions from the Global authentication requirement**

As I understood the documentation, there are several methods of excluding specific files, folders or even just actions & events. **First** I have tried the “service setting” method in “startup.cs”: in “ConfigurationServices”, at the end of the “AddAuthorization” service by adding a new option to exclude the desired View file. It did not work as I expected (I may have misunderstood the instructions).

**The other method** I tried (worked as expected), was to add the “ ***[AllowAnonymous]*** ”attribute to the appropriate actions /events (functions) that I want even guest users to have access to for Reading.

For html elements (eg.: buttons) to generate for such Users, I used a case-by-case exclusion. In this example, it is in the “\_Layout.cshtml” file on the globally visible top navigation bar, the “Home” access:

