1. ***Brief snapshot of the features that will be implementing in this app.***

**Web App** with **.Net 8, Entity Framework and Angular 17 and 18.** This is an example of a Developers Connecting app, a social kind of app which has features common to many different types of apps, such as registration and login, we will use **ASP.Net identity** to do that.

Also got **live presence**, so users will be able to see who else is online, **at the same time.**

We can like other users as well, and there's an area where **users can view the list of members that they like and who like them.** We'll have the ability **to edit a profile** and persist those changes to our **back-end database**, which is going to **be our API and using Entity Framework.**

There is also **a system for uploading a photo**, so users can upload their own photos. They can **switch the photo** **and it updates** everywhere in our app.

Also, a **messaging system** so users can message other users and **this is live**. So, if a user is connected somewhere else in the world, then they'll be able to see this message coming in and **there'll be live chats** between two different users.

Also, implementing **pagination** with **caching**, so, we can instantly go to other user’s profiles and it remembers what we've done and if they go to a different page and reset the filters, then we see all of this in place.

1. ***Technical features***

Using **dotnet** (was dotnet core). That is going to form the **back-end of the application** the **API**. Also adding **angular** **framework** to provide the **client-side,** and build a single page application. Display the user interface to the client. Also going to use **Entity Framework** that provide us with the **ability to query our database**. To build this app, we'll be using **HTML bootstrap to provide the styling framework** and by adding a touch of CSS.

*Hint\** it is useful to have the NVM (node version manager), for choosing between deferent node.js versions and not need to uninstall and reinstall.

1. ***Simple walking skeleton***

We're going to **have some data in a database,** then **create an API project** that's going to **fetch the data out of that database,** also **create an angular project** that's going to **query our API and receive the data from our API that comes from our database**, and then we're going to **display it in the client's browser**. It will be really simple initially and then it will add complexity.

* First, implement the **basic API functionality** and have an introductory understanding of using the **dotnet command line interface (CLI)**. A command line interface that we can use to do various things with dotnet.
* **API controllers and endpoints**, and understand the structure of an API controller.
* **Entity Framework**
* We're going to take a look at **the API project structure**. What involves in a .Net project, how our application starts up.
* The **configuration and environment variables** that it uses inside our application.
* Then **commit the code into source control**, which allows us to take effectively a backup of our code and it also contains a version history of the code as well.

In the cmd id we run *dotnet –info* then we will see all the different SDKs we have installed on our computer. When we create the project from the CLI it automatically creates it by using the latest SDK. But if I want to use an older one I need to create this *global.json* as we see below:



Also, we have the list of all the runtimes installed.

Useful CLI commands:

* dotnet -h: for help, where it lists all available commands
* add, new, sln are the most common commands.

For our case the steps are:

Type *mkdir ProfessionalsConnectApp* -> *cd ProfessionalsConnectApp -> dotnet new sln -> dotnet new webapi -h* (for help to see what it includes) ->

**

We are going to use this option.

*-> dotnet new webapi -controllers -n API*, so this will create a folder with the name API and inside

-> *dotnet sln list,* we see there are no projects found so we need to add it.

*-> dotnet sln add API,* and the result



When we open the project and run the server `***dotnet run***`, it says that **listens to a port**, and how the CLI command knows to listen to this port? In the properties folder the ***launchSettings.json***file where we got a number of **different settings and profiles** of what's going to happen when the application starts. The first one in the list is typically the default one.



That's the reason my application started on that particular URL. And it started in **development mode** because that's the setting contained in **this environment variables section.**  In this project we will use postman instead of swagger.

When we **installed the .Net SDK is that dot net has added a self-signed certificate** so that it's **trusted by our browser when we use this URL**, and let’s restart the server. We can see it's running on these two URLs that I've specified there.

**An important file is the Program.cs class**. This is our main entry point for our application. When we run the .net run command this is where it looks into and here is **the App.run method. Then this executes and effectively runs our application.** Inside this program class the first area is the **services area,** where we **register any service** that our application needs inside this part of the file, **then the application is getting built**, and then is the section to **configure the HTTP request pipeline**. The stuff that's contained in the **HTTP request pipeline** is considered middleware and we can put things inside here that allow us to do something with the HTTP request on its way into our API service or on its way out of our API service.

So, if we wanted to check if a user is authorized to access an API endpoint, then we've got middleware such as use authorization or if we wanted to redirect them, then we could use the Https redirection middleware contained.

If we click on API (API.csproj), then this our project file contains the target framework that I'm using.

The API.http file is just another way of testing an API endpoint in Visual Studio 2022.

Some configuration files are the appsettings.Development.json for developer mode and Appsettings.json, which is used whatever mode we're running our application in.

This is a professionals connect app, so it focuses on users. We will create something that's referred **as an entity (model).** When we will use **Entity Framework** into our application, each **entity is going to represent a table** in that database, this folder will be called as entities. So, we create a class in the folder with the name ‘AppUser’.

*Hint \* At each one of our classes goes inside a namespace. A namespace is at logical naming structure and if I wanted to call a class app user somewhere else in my app, I can do so, but only if it's inside a different namespace, otherwise we'll have a conflict. When creating a class, typically you would create the namespace as the physical name of the folder where you've created the class. So, typically this would be API entities as the namespace. And then if I decided for whatever reason, I wanted to create an app user class inside the controller’s folder, I could do so as long as it used a different namespace.*

First part here is its access modifier. How visible should this class be to the rest of my application? When it comes to Entity Framework, we have to make these public properties.

Each one of these properties that we create is going to represent a column in the EF. We're just going to have an ID and a name property effectively.

As of C-sharp 11, we have another modifier that we can use in our properties, and that's to specify required. It means that we cannot create an app user without also specifying the username and this nullable thing is actually part of the nullable flag that we have inside our .csproj.



1. **Entity Framework**

**EF is an object relational mapper (ORM) and its job is to translate our code into SQL commands that update our tables in the database**. In the dotnet 3.5 Framework we used ADO.Net code.

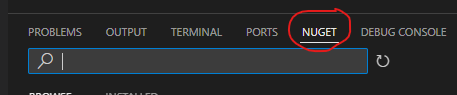
When we add **Entity Framework**, we need to **create an important class that derives from the DB context class that we get with Entity Framework**. This **class acts as a bridge** between our domain or entity classes and the database, and this DBcontext class is the primary class we use for interacting with our database.

AppUser ------ EF (DbContext)---------DB

With **SQLite** we don’t need to install anything, and it simply adds a file into our projects folder for the database. EF use **querying**, **change tracking** it will keep track of changes occurring in our entities which need to be submitted to the database, it also **saves** the database and Entity Framework will execute, insert, update and delete commands to the database, Dbcontext class gives us a save changes method that we can use, it also gives us **concurrency**, it uses optimistic concurrency by default to protect overwriting changes made by another user, it also deals with **transactions** and it provides automatic transaction management whilst querying Or saving data. It includes first level **caching** out of the box, so repeated querying will return data from the cache instead of hitting the database. It offers **built in conventions**, and when I was talking about why the ID property needs to be named, ID is because Entity Framework follows conventions and includes a set of default rules which automatically configure the Entity Framework schema, or the model that we use to create our database. We can also **configure** our entities, and there's ways to configure these so that we can override the conventions if we wish to do so, offers us **migrations**. So, we don't need to create our database manually Entity framework can do this for us.

1. **Use of EF**

From the Nuget package we will install the two Entity Framework packages that we need and when we do install any packages then they get listed inside our .csproj file.



Microsoft.EntityFrameworkCore.SQLite version 8.0.8 (matches .Net runtime version)

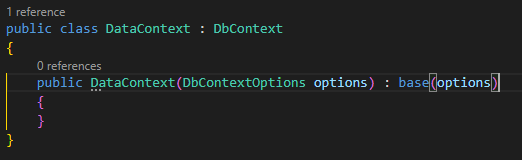
Microsoft.EntityFramework.Design version 8.0.8 (matches .Net runtime version)

1. **Adding DbContext in our class**

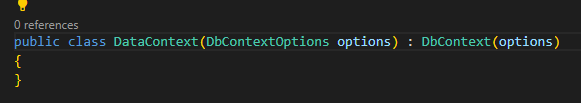
Inside the API create a new folder and call it data, inside the data folder create a new file call this DataContext. **We need to derive from an Entity framework class called Dbcontext**. We need to tell it where to find our database and how to get to our database. **A Dbcontext instance represents a session between the database and can be used to query and save instances of our entities.** We have an entity called the app user and the Dbcontext is a combination of the unit of work and repository patterns.

**We need to generate a constructor that gives our data context some options.** When a new instance of this data context is created then the constructor is executed. In this case it allows us to pass some options to the parent class the DB context and those options will be our database connection string, for example.

Usual way of creating a class with a contractor



And the option to use a primary constructor:



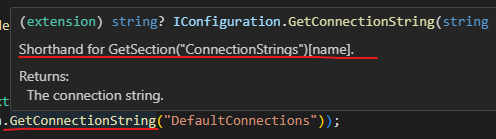
.Net eight comes with C sharp 12 And one of the features of C sharp 12 is this concept of a primary constructor.

Now another thing that we're going to add inside here is we need to **tell our DB context about our entities.** We use a DbSet<> a DbSet can be used to query and save instances of a type of entity. We use Linq queries against a DB set, which are then translated into queries against the database. If we call this prop Users then this is going to be the name of our table in our database.

**It will use the ID as the primary key for the database** if I was to call it something else anything but ID then this would not be valid or Entity Framework would not know which one of these to choose to use as a primary key. **Then we need to tell our application about this DB context class.** Similarly, when we add a new service to our application, we need to **register it in the program class**, then we pass the options, because when we add a service or register a service it is going to use this is via dependency injection and through dependency injection, then dot net is going to create a new instance of our data context class. Then via the constructor, it's going to pass the options that we specify when we register this service. The connection string we're going to get this from configuration.

1. **Create the connection string**

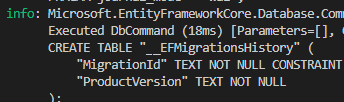
We need to provide this connection string. The **configuration files are the appsettings.json** which **is read from no matter** which **mode** we're using and currently from our **launchSettings.json, we're running in development mode.** Also have an **appsettings.development.json** and when we're running in development mode then this one is read from. The Appsettings.json is read from if we're running in production, in development, in staging and testing. So the **appsettings.Development.json is good place to add our development connection string**. After the logging section use ConnectionStrings which matches the data context registration when we added it to our program.cs class and use the GetConnectionString method.



Then we need to **create a new migration**, which effectively Entity Framework is going to take a look at our code inside the data context. We need to see what you currently have installed by running *dotnet tool list -g* and this will tell you what global net tools you have available. Navigate to nuget.org go to versions for the *dotnet ef* command then click on the one that matches the runtime that we are using, and this is the command that we'll go ahead and use to install it. So, copy this into my clipboard and run it to vs code. Then in vs code we can run *dotnet ef -h* to see the commands and finally **add a migration by typing *dotnet ef migrations add InitialCreate -o Data/Migrations.***

1. **Create database**

Let’s create our database based on the code in this migration class, make sure the app is not running, and run ***dotnet ef database update***. Once that has happened, then we'll see some outputs in our log files. In the terminal we can see that it created a table called *\_\_EFMigrationsHistory*, this keeps track of the migrations that have already been applied to the database.



Then creating the users table, and then inserting a new entry into the *\_\_EFMigrationsHistory* table. Inside our solution explorer we now have a file called *connecting.db* and if I click on this then I can't open my database using VS code in this way because it's either binary or uses an unsupported text encoding, and it's an SQLite file.

1. **API Controller**

Let's create a **new API controller** so we can create an API endpoint and **receive HTTP requests that** we can then use to respond with the data from our database. We will ensure that we've got a way to query and return data from our database to whatever client is requesting that data. Inside our controller’s folder at the top, we have this **ApiController attribute**, **this indicates that a type and all derived types are used to serve HTTP API responses**. We also give our API controller a route. So now we will create our controller for the Users, and also create a method GetUsers() and return a list of users, so we need access to the DB. **Dependency injection means that the framework is responsible for creating a new instance of our data context class, and then disposing of that class that we inject after we finished using it.** So, when we inject something like the data context into our class, we don't need to worry about its disposal. To use DataContext in our class we need to add it as a parameter to the contractor and then assign it with the readonly field in out class. Then use it in our method to retrieve data from the DB.