Building a multi layer .NET Core 3.0 API from zero

Let's build together a multi layer .NET Core API with the latest framework version and most used patterns

Andre Lopes Follow

Nov 6, 2019 · 14 min read ★

This is my first ever article written. I decided to write because I actually couldn't find many articles explaining on how to separate your application in multiple layers. I hope you guys enjoy it. :)

Here we are building a nice API with basic CRUD (Create, Read, Update, Delete) operations. It will be an API for an app that stores which musics you like with its artists. We will call it **MyMusic**.

I will show you how to:

- Create an application in separated projects to make it decoupled from each module.
- Implement **Repository** and **Unit of Work** pattern.
- Use **Entity Framework Core** for persistence.
- Add **AutoMapper** for mapping models into API resources.
- Add **Swagger** to have a friendly API interface.

Requirements

For building our application we'll need a few tools:

.NET Core 3.0

Microsoft SQL Server Express

If you are on Linux, here is how to run MSSQL Server Express with Docker, but if you choose to use other database provider, here is a list of the current supported database providers

Visual Studio Code or Visual Studio Community

For this article I'll be using Visual Studio Code and I use a few extensions to help coding C# in it: C# and C# Extensions

Ready? Go!

For starting we need to create our app structure. Let's start creating our solution folder which will be named **MyMusic**. Then we navigate inside with a command line tool and run the following command to create a solution:

```
dotnet new solution
```

After we need to create our application projects for API access, Core, Services and Data access. We can do that by running the following commands:

```
dotnet new webapi -o MyMusic.Api

dotnet new classlib -o MyMusic.Core

dotnet new classlib -o MyMusic.Services

dotnet new classlib -o MyMusic.Data
```

The first command generates our `API`, which is the point of access for our application.

The second generates our `Core` library. This is our application's foundation, it will hold our contracts (interfaces, …), our models and everything else that is essential.

The third is for our `Services` layer. This is where we will implement business logic.

And the last is our `Data` access layer where we will connect with data providers (SQL Serve Express).

Add the projects to the solution:

```
dotnet sln add MyMusic.Api/MyMusic.Api.csproj
MyMusic.Core/MyMusic.Core.csproj MyMusic.Data/MyMusic.Data.csproj
MyMusic.Services/MyMusic.Services.csproj
```

And for last link each depending project:

```
dotnet add MyMusic.Api/MyMusic.Api.csproj reference
MyMusic.Core/MyMusic.Core.csproj
MyMusic.Services/MyMusic.Services.csproj

dotnet add MyMusic.Data/MyMusic.Data.csproj reference
MyMusic.Core/MyMusic.Core.csproj

dotnet add MyMusic.Services/MyMusic.Services.csproj reference
MyMusic.Core/MyMusic.Core.csproj

dotnet add MyMusic.Api/MyMusic.Api.csproj reference
MyMusic.Services/MyMusic.Services.csproj
```

```
MyMusic.Core/MyMusic.Core.csproj MyMusic.Data/MyMusic.Data.csproj
```

By the end we should have this structure.

Folder structure

Core layer — The heart of everything

Let's begin building our Core layer. As said before, this will be the home of our Models, our interfaces and so on.

First we create a folder called Models to hold our models, which will be 2— Music and Artist.

We can begin by define our `Music.cs` model. It has a `1:n` relationship with Artist model.

And finally our `Artist.cs` model

Now that we defined our models we can implement our repositories with the repository pattern

Repository Pattern and Unit of Work

The **Repository pattern**, in simple words, is encapsulating your database operations (ORM or other methods of database access) in one place. Basically abstracting your data access layer. This has many advantages, such as:

- **Reusability** — You don't need to rewrite code for using accessing your database
- **Testability** — You can test you database actions without data access logic
- **Separation of concerns** — Your data access is responsibility only of the respoitory
- **Decoupled code** — If you want to change persistence framework, it requires less effort

A **Unit of work** is responsible for keeping track of the list of changes during a transaction and committing it.

Even though Entity Framework internally implements the repository pattern and unit of work pattern, it is a good practice to implement ourselves the repository to decouple our project from Entity Framework. This way we are not strongly bound to it and if we want to switch from this Entity Framework version to another or to another ORM framework, we require much less work to do so. Also it helps when mocking stuff for testing.

First we create a `Repositories` folder and add a `IRepository.cs` interface, which will be our base repository with basic database operations.

Now the other two repositories:

For last our Unit of Work interface:

Services

Services will be the center of our business logic, the link between our API and our Data.

First we create a folder to house our services interfaces, the Services folder.

Then create two services interfaces, `IMusicService.cs`, to handle music logic, and `IArtistService.cs`, to handle artist logics.

Summary

The Core project will hold all our application business logic structure, everything that should tell how it should work. So if we need a big change of technology or even logic, we can just unplug the old module and plug a new one that follows our Core project contracts and models.

By the end you should have this structure in this project:

MyMusic.Core folder structure

Data Access — Persisting our data

Persistence layer is very important because it is how we communicate with our database. And for this .NET Core provides an strong framework for that, Entity Framework Core .

Entity Framework is a Object Relation Model (ORM) which basically maps all database tables and columns to C# objects, making it super easy to manage data and run queries.

Entity Framework command command line tool doesn't come in `dotnet cli` anymore, so we need to install it with the following command:

```
dotnet tool install --global dotnet-ef
```

Now we can execute entity framework actions like creating migrations through the cli with `dotnet ef` command.

Our `MyMusic.Data` will be the project handling all this data layer connections

Dependencies

First we need to add the remaining necessary dependencies to our Data project. Just run the following commands:

```
dotnet add MyMusic.Data/MyMusic.Data.csproj package
Microsoft.EntityFrameworkCore

dotnet add MyMusic.Data/MyMusic.Data.csproj package
Microsoft.EntityFrameworkCore.Design

dotnet add MyMusic.Data/MyMusic.Data.csproj package
Microsoft.EntityFrameworkCore.SqlServer
```

Configuration

Now we need to define how our models behavior, like constraints and relations.

For that we'll create a folder called Configurations, then add one configuration file for each model — `MusicConfiguration.cs` and `ArtistConfiguration.cs` .

We could define all configurations directly in the DbContext (explained later), but for organization purpose, it is better to have it separated in their own files.

DbContext

We are ready to build our DbContext. Let's create a new file in the project level called `MyMusicDbContext.cs` and add our `DbSets` which gives us access to its respective table.

Note that if you do not plan to manage or fetch data individually from a table, you do not necessarily need to add a `DbSet` for it, Entity Framework will create that table as long as the model has any relation with the other models.

Repositories and Unit of work

Remember those interfaces we created in the Core project? Now it is time to implement them so we can have an interface with our `DbContext`.

Create a folder called `Repositories`.

First we implement the base repository (`Repository.cs`):

Note that operations can be added, changed or removed depending on your needs

After having defining all basic operations, we are ready to implement the remaining two repositories.

And for last, our Unit of Work which will wrap all repositories in one place. Create a `UnitOfWork.cs` file in the project level.

And for last we just need to add **dependency injection** so our application knows that when we use the repository interfaces it should inject these repository implementions. And now we just need to add dependency injection for our Unit Of Work in the `Startup.cs` of our API project by adding the next code line to the `ConfigureServices` method:

```
services.AddScoped<IUnitOfWork, UnitOfWork>();
```

Transient vs Scoped vs Singleton

You noticed that we added the dependency as Scoped. We have 3 types of dependency injection:

- **Transient** — Objects are different. One new instance is provided to every controller and every service
- **Scoped** — Objects are same through the request
- **Singleton** — Objects are the same for every request during the application lifetime

Migrations

We are almost over configuring Entity Framework.

Now we need to add the connection strings and tell our API to use it.

Go to `appsettings.Development.json` and add the ConnectionStrings.

Using appsettings.Development.json because it is a dev environment. What happens is when .NET Core is building, it checks every property in

appsettings.json that matches with the current environment one (this case Development) and overrides these properties.

- **server** is your database server
- **database** is your database name
- **user** is an user with admin rights
- **password** is this user password

MSSQL Server requires the password to follow some minimum requirements. If you are having trouble, just use the same as I am using

And now just add the last piece of configuration to `Startup.cs` in the API project. Add this piece of code under the `ConfigureServices` method:

```
services.AddDbContext<MyMusicDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("Default"), x
=> x.MigrationsAssembly("MyMusic.Data")));
```

Here we add our `MyMusicDbContext`, tell to use `SqlServer` using the `Default` connection strings in `appsettings.json` and that our migrations should be run in `MyMusic.Data`.

And now we can create our migrations with the following command:

```
dotnet ef --startup-project MyMusic.Api/MyMusic.Api.csproj migrations
add InitialModel -p MyMusic.Data/MyMusic.Data.csproj
```

`--startup-project` switch tells that MyMusic.Api is the entry project for our app and switch `-p` tells that the target project of our migrations is MyMusic.Data. `InitialModel` is the name of this migration.

You can check in the Data project that a new folder called `Migrations` was automatically created and it contains our new migration.

Migrations folder

For last we just need to reflect our migrations in our database with the following command:

```
dotnet ef --startup-project MyMusic.Api/MyMusic.Api.csproj database
update
```

Seed some data

For last we can add some dummy data to our database.

Just create an empty migration with

```
dotnet ef --startup-project MyMusic.Api/MyMusic.Api.csproj migrations
add SeedMusicsAndArtistsTable -p MyMusic.Data/MyMusic.Data.csproj
```

And then just add the following code to the file `*_SeedMusicsAndArtistsTable.cs` where * represents a number (mine is `20190802142149_SeedMusicsAndArtistsTable.cs`)

Note that there's the `HasData` method in the builder object in the Configuration files, but I found this way of seeding data more useful for this case because we can insert musics and artists together without needing to hard code ids.

Summary

Now we are prepared to start making requests to our database and we have some data seeded in it.

By now you should have the Data project structure similar to this:

MyMusic.Data project folder structure

Services — Business Logic

We arrived at our Services layer, which is responsible for our business logic and interfacing with the Data Access layer. The key here is that we are gonna use the Unit Of Work to handle that interface so we don't have to go adding our `DbContext` directly.

As our API is has just simple CRUD operations, this section is not gonna be too long.

Here we will be needing just the implementation of the two services interfaces defined in Core, which are `MusicService.cs` and `ArtistService.cs` .

Doing so, we abstract our business logic from our presentation layer, which is our API.

And is just left to add dependency injection for our services as we did for the Unit of work by adding the next lines to `Startup.cs` :

```
services.AddTransient<IMusicService, MusicService>();
services.AddTransient<IArtistService, ArtistService>();
```

Summary

Your folder structure should be like this:

MyMusic.Services folder structure

API — Presenting our app

Finally it is time to build our presentation layer. Here we will handle API Restful calls, resource mappings and resources validation.

Swagger

First let's add Swagger to our project to help us visualize our progress (finally). To do so just add Swashbuckle package to our API project with the net commands:

```
dotnet add MyMusic.Api/MyMusic.Api.csproj package
Swashbuckle.AspNetCore --version 5.0.0-rc3
```

Note that we are adding a Release Candidate because the last stable Swashbuckle version does not support .NET Core 3

And for configuring we need to add the following commands to two methods in startup.cs

ConfigureServices method add:

```
services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo { Title = "My Music",
Version = "v1" });
});
```

And in **Configure** method add:

```
app.UseSwagger();

app.UseSwaggerUI(c =>
{
   c.RoutePrefix = "";
   c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Music V1");
});
```

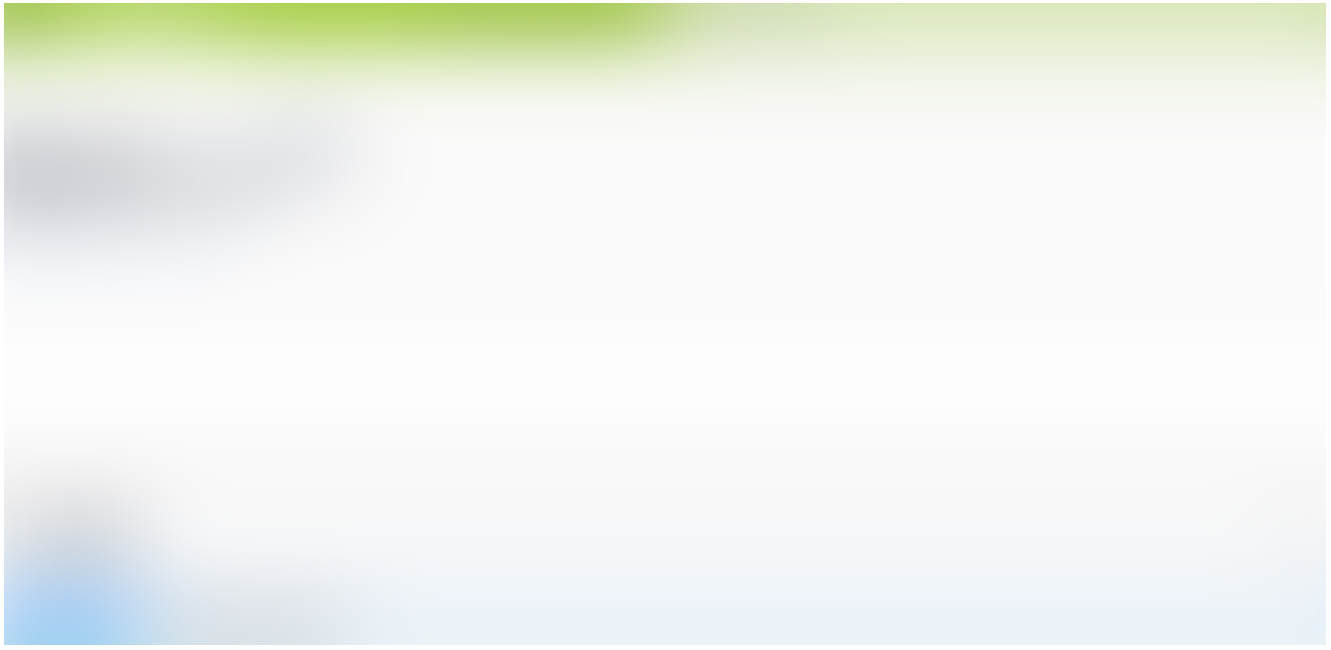
And add Swagger's assembly to the top with, above the **namespace**:

```
using Swashbuckle.AspNetCore.Swagger;
```

Now, to test it, just run our application for the solutions folder with:

```
dotnet run -p MyMusic.Api/MyMusic.Api.csproj
```

And by visiting `https://localhost:5001` you should see the following page:



See that **Values** correspond to the name of the only controller we have under the `Controllers` folder and every HTTP method corresponds to one method in `ValuesController`

Now that we have Swagger running, we can go to the next part which is building our Musics endpoint.

MusicsController

Let start by deleting the `ValuesController.cs` file and creating a new `MusicsController.cs` inside the Controllers folder.

Controllers folder

Now let's work on it. First we need to extend the ControllerBase, this will tell that our class is a controller. Your class should be like this.

```
public class MusicsController : ControllerBase
```

Now let's add the ApiController attribute and define our controllers route by add the next attributes on top of our class:

```
[Route("api/[controller]")]
[ApiController]
```

You class should be like this now:

```
[Route("api/[controller]")]
[ApiController]
public class MusicsController : ControllerBase
```

Now let's create our first endpoint.

Create a method called **GetAllMusics** that is **public**, **asynchronous** and returns a **Task<ActionResult<IEnumerable<MusicResource>>>**.

```
public async Task<ActionResult<IEnumerable<Music>>> GetAllMusics()
{
}
```

As the names describes, it will be a method to return all our musics stored in the database. Now we need to mark it as a HTTP GET method and assign a route to it by adding the next attribute on top of it:

```
[HttpGet("")]
```

This tells that our method is a get and that its route is the base controllers route, which in the end will be like this:

```
api/musics
```

Now we need to fetch and return our musics, for that we need to inject our MusicService in our controller. To do so we just need to add and initialize it

in the **constructor**, like this:

```
private readonly IMusicService _musicService;

public MusicsController(IMusicService musicService, IMapper mapper)
{
  this._musicService = musicService;
}
```

We need a private variable to access the service from within our class, and as we already defined dependency injection before, it will be automatically initialized when needed.

Now to finish our endpoint, just fetch the musics from our service and returning it as a result:

```
[HttpGet("")]
public async Task<ActionResult<IEnumerable<Music>>> GetAllMusics()
{
  var musics = await _musicService.GetAllWithArtist();
  return Ok(musics);
}
```

Now we can test it making a call through using swagger:

But you will probably get an error with this call, that's because we have a self reference that ends up in a loop. To avoid that, and as a best practice, we should always avoid exposing our domain models to our API responses by having resources in our API layer that will be the API contracts for responses.

Let's first create a folder called **Resources** in our project directory and add a `MusicResource.cs` and a `ArtistResource.cs` file to it. This files will have almost the same properties as the domain models, but our **ArtistResources** will not have a reference to **MusicResource**.

Resources folder

And now for the mapping we are gonna use AutoMapper.

For that we need to add to packages to our API project with:

```
dotnet add MyMusic.Api/MyMusic.Api.csproj package AutoMapper
dotnet add MyMusic.Api/MyMusic.Api.csproj package
AutoMapper.Extensions.Microsoft.DependencyInjection
```

Enable AutoMapper by adding the next line to **ConfigureServices** in `Startup.cs` :

```
services.AddAutoMapper(typeof(Startup));
```

And now we need to tell AutoMapper how to map our models. For that create a folder called **Mapping** in our project directory and add a file called `MappingProfile.cs`

Mapping folder

This class should extend **Profile** and in the **constructor** we handle our mappings, in the end you should have this:

Now with that we can inject and use **AutoMapper** in our controller the same way we did with the **MusicService**

MusicsController constructor should be like this now:

And now modify our **GetAllMusics** method to this and run the application:

Now if you test our application through swagger you will see the following result:

GetAllMusics result

We can create a method to get a single music by its id:

See how we put `{id}` in the route? It indicates that `id` corresponds to the `id` in the parameters and it comes from the url.

The following code will get the data from the music with id 1.

```
api/musics/1
```

Creating a music and using FluentValidation

For that we don't want to request unnecessary data, so we need to create a resource for that, we will call it `SaveMusicResource.cs`, and as we will probably be needing this for artists later, we will create a `SaveArtistResource.cs` file too.

Resources folder completed

Notice that these examples are simple, but probably in a real application you would have a lot more properties

Also we need to add mapping from our save resources to our models by adding the following lines in our `MappingProfile.cs` right after `CreateMap<ArtistResource, Artist>();` :

```
CreateMap<SaveMusicResource, Music>();
CreateMap<SaveArtistResource, Artist>();
```

We need make sure that our parameters have the needed values to proceed, for that we will use FluentValidation by adding its package to our project with

```
dotnet add MyMusic.Api/MyMusic.Api.csproj package FluentValidation
```

Now we create a folder called **Validators** and add two files `SaveArtistResourceValidator.cs` and `SaveMusicResourceValidator.cs` .

Validators folder

And now for our **CreateMusic** method:

Notice the `[FromBody]` attribute before the parameter, it indicates that this object is coming from the requests body

Updating a music

Now for updating a music we will create a **PUT** method:

See how we have the id and the body? `Id` is coming from the request url and `saveMusicResource` is coming from the request body. The request url will be something like this:

```
api/musics/1
```

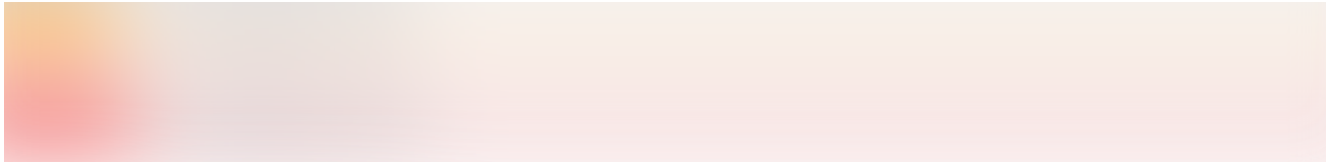
And as it is a **PUT** method, it won't fall on the get by id.

Deleting a music

And for last we need a **DELETE** method to delete a music by its id:

End result

Your endpoints should be this:



Musics endpoints

And your `MusicsController.cs` should be this.

ArtistsController

For Artists we have basically every dependency prepared when we ran through MusicsController (**ArtistResource**, **SaveArtistResource**, **SaveArtistResourceValidator**, **ArtistService**), the code will be almost the same, check the code below:

Artists endpoint

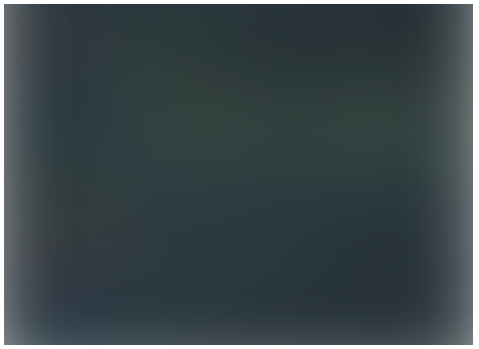
Startup.cs

By the end, your Startup.cs should be looking like this:

Summary

Your API layer will be the one responsible to present and receive requests for your application. It will be the one to first validate your resources specifications that will prevent your application from receiving undesired data. The one to map resources to domain models.

In the end, you API layer folder structure should be this:



MyMusic.Api folder structure

Conclusion

This was a basic tutorial on how to leverage .NET Core and its powerful tools to create a multi layer and maintainable API.

You learned how to separate your API structure in multiple module projects so we don't have a high dependency in one technology.

You learned how to design a Core of an application, how to apply Repository and Unit Of Work pattern and how this helps you abstract the use of ORM.

Speaking of ORM, you learned how to use code first database design with Entity Framewok and to build it in a very good structure way.

You learned the value of a Business Layer that is not coupled to an specific ORM.

In API layer you learned how to not expose your models by using Resources, how to map them using AutoMapper, how to validate your inputs using FluentValidation and how to use Swagger to simplify your API display.

If you want the code for this application, it is available **here**.

Thanks to Andrew Hillier.

API Net Core 3 Web Development Entity Framework Net Core