**Configurations**

**Project creation through CLI**

1. Create solution file

* dotnet new sln

1. Create a WebApi project

* dotnet new webapi -o <projectName>

1. Add project in Solution file

* dotnet sln add <projectName>

1. List projects in solution

* dotnet sln list

1. Run project

* dotnet run
* dotnet run watch

**Base Controller**

1. Create a base Controller and add the attributes on the base controller level.
2. Add **[ApiController]** attribute to handle the controller level validations for the correct parameter type in the payload
3. Add **[Route(“api/[controller]”)]** attribute to have a same URL structure for all endpoints.

**Base Entity Class**

1. Create a BaseEntity class as a parent class for all the entity classes.
2. This class would have the common properties that we could use in all the other entities. For example, the ID property.

**public int Id {get; set;}**

The primary key for all the entities would be an integer value.

1. The Entity Framework would read an Id field as a Primary Key and would auto increment the value when inserting a new record.

**Data Seeding**

1. Use JSON files for inserting initial values in the tables.
2. Create a **DbContextSeed{}** class with static methods to insert the records.

**public** class StoreContextSeed

**{**

**public** static **async** Task SeedAsync**(**StoreContext context**,** ILoggerFactory loggerFactory**)**

**{**

**try**

**{**

**if(!**context**.**Entity**.**Any**()){**

var entityData **=** File**.**ReadAllText**(**"../seedEntity.json"**);**

var entities **=** JsonSerializer**.**Deserialize**<**List**<**Entity**>>(**entityData**);**

**foreach(**var item **in** entities**)**

**{**

context**.**Entity**.**Add**(**item**);**

**}**

**await** context**.**SaveChangesAsync**();**

**}**

**}**

**catch** **(**Exception ex**)**

**{**

var logger **=** loggerFactory**.**CreateLogger**<**DbContextSeed**>();**

logger**.**LogError**(**ex**.**Message**);**

**}}}**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Update the **Program.cs** file and call the **SeedAsync()** method.

**public** static **async** Task Main**(**string**[]** args**)**

**{**

var host **=** CreateHostBuilder**(**args**).**Build**();**

**using(**var scope **=** host**.**Services**.**CreateScope**()){**

var services **=** scope**.**ServiceProvider**;**

var loggerFactory **=** services**.**GetRequiredService**<**ILoggerFactory**>();**

**try**

**{**

var context **=** services**.**GetRequiredService**<**DbContext**>();**

**await** context**.**Database**.**MigrateAsync**();**

**await** DbContextSeed**.**SeedAsync**(**context**,** loggerFactory**);**

**}**

**catch** **(**Exception ex**)**

**{**

var logger **=** loggerFactory**.**CreateLogger**<**Program**>();**

logger**.**LogError**(**ex**,** "And error occured during migration."**);**

**}**

**}**

host**.**Run**();**

**}**

**Cleanup startup.cs class**

1. Create a new Class for **Dependency Injections**.

**public** static class ApplicationServicesExtensions

**{**

**public** static IServiceCollection AddApplicationServices**(this** IServiceCollection services**)**

**{**

services**.**AddScoped**<**IInterface**,** ImplementationClass**>();**

**return** services**;**

**}**

**}**

The code above has a static method that is accepting and returning an **IServiceCollection**.

We have defined the services dependencies in this method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Create a new class for Application Configuration in **startup.cs**.

**public** static class SwaggerServiceExtensions

**{**

**public** static IServiceCollection AddSwaggerDocumentation**(this** IServiceCollection services**)**

**{**

services**.**AddSwaggerGen**(**c **=>**

**{**

c**.**SwaggerDoc**(**"v1"**,** **new** OpenApiInfo **{** Title **=** "API"**,** Version **=** "v1" **});**

**});**

**return** services**;**

**}**

**public** static IApplicationBuilder UseSwaggerDocumentation**(this** IApplicationBuilder app**)**

**{**

app**.**UseSwagger**();**

app**.**UseSwaggerUI**(**c **=>** c**.**SwaggerEndpoint**(**"/swagger/v1/swagger.json"**,** "API v1"**));**

**return** app**;**

**}**

**}**

The example above is having a static method for configuring the swagger with the application in **startup.cs** class.

The method above returns and accepts **IServiceCollection**.

1. Update the **startup.cs** class and call the above static methods in respective methods.

services**.**AddApplicationServices**();**

app**.**UseSwaggerDocumentation**();**

**AutoMapper Configuration**

1. Create new class **MappingProfile.cs** as a Helper Class.

**public** class MappingProfiles **:** Profile

**{**

**public** MappingProfiles**()**

**{**

CreateMap**<**SourceEntity**,** DestinationEntity**>();**

CreateMap**<**Product**,** ProductToReturnDto**>()**

**.**ForMember**(**d **=>** d**.**ProductBrand**,** o **=>** o**.**MapFrom**(**s **=>** s**.**ProductBrand**.**Name**));**

**}**

**}**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. If we want to update the value in a property of a class, then we would create a new class as a Resolver class for that entity.

**public** class ProductUrlResolver **:** IValueResolver**<**Product**,** ProductToReturnDto**,** string**>**

**{**

**private** **readonly** IConfiguration \_config**;**

**public** ProductUrlResolver**(**IConfiguration config**)**

**{**

\_config **=** config**;**

**}**

**public** string Resolve**(**Product source**,** ProductToReturnDto destination**,** string destMember**,** ResolutionContext context**)**

**{**

**if(!**string**.**IsNullOrEmpty**(**source**.**PictureUrl**))**

**{**

**return** \_config**[**"ApiUrl"**]** **+** source**.**PictureUrl**;**

**}**

**return** **null;**

**}**

**}**

1. To call the MapperProfile methods

**private** **readonly** IMapper \_mapper**;**

**public** ABCController**(**IMapper mapper**)**

**{**

\_mapper **=** mapper**;**

**}**

**public** void MethodOne**()**

**{**

var result **=** \_mapper**.**Map**<**SourceEntity**,** DestinationEntity**>(**input**);**

**}**

1. **This is Very Important.** Update the Startup.cs class and configure the AutoMapper

**services.AddAutoMapper(typeof(MappingProfiles));**

**Exception Handling and Error Response**

1. Create a Class to handle the errors. We want to send a consistent error message for all the error types. Create a new folder, **Errors**.

**public** class ApiResponse

**{**

**public** ApiResponse**(**int statusCode**,** string message **=** **null)**

**{**

StatusCode **=** statusCode**;**

Message **=** message **??** GetDefaultMessageForStatusCode**(**statusCode**);**

**}**

**public** int StatusCode **{** **get;** **set;** **}**

**public** string Message **{** **get;** **set;** **}**

**private** string GetDefaultMessageForStatusCode**(**int statusCode**)**

**{**

**return** statusCode **switch**

**{**

400 **=>** "A bad request, you haev made"**,**

401 **=>** "Authorized, you are not"**,**

404 **=>** "Resource found, it was not"**,**

500 **=>** "Errors are the path to the dark side. Errors lead to anger. Anger leads to hate. Hate leads to career change"**,**

\_ **=>** **null**

**};**

**}**

**}**

To call the above method we would return an **ApiResponse** object.

1. Example for using the Error Messages

**[**HttpGet**]**

**[**ProducesResponseType**(**StatusCodes**.**Status200OK**)]**

**[**ProducesResponseType**(typeof(**ApiResponse**),** StatusCodes**.Status404NotFound)]**

**[**ProducesResponseType**(typeof(ApiException),** StatusCodes**.Status500InternalServerError)]**

**public** **async** Task**<**ActionResult**<**IReadOnlyList**<**Entity**>>>** GetEntities**()**

**{**

**if(**error**)**

**{**

**return** **NotFound(new** **ApiResponse(**404**));**

**}**

**return** **null;**

**}**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. If we want to return more information in case of an Exception, we would create a new class, which would inherit ApiResponse class.

**public** class **ApiException** **:** ApiResponse

**{**

**public** ApiException**(**int statusCode**,**

string message **=** **null,**

string details **=** **null)**

**:** **base(**statusCode**,** message**)**

**{**

Details **=** details**;**

**}**

**public** string Details **{** **get;** **set;** **}**

**}**

In **point 2** where we are defining the Server500Exception we are returning the **ApiException** type of response, because we want more information regarding the error in that case.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **This a good one.** If we want to validate the API level exceptions and want to show the correct error message, we will create a new class **ApiValidationErrorResponse**.

**public** class ApiValidationErrorResponse **:** ApiResponse

**{**

**public** ApiValidationErrorResponse**()** **:** **base(**400**)**

**{**

**}**

**public** IEnumerable**<**string**>** Errors **{** **get;** **set;** **}**

**}**

To use this above class to handle the API specific errors we also need to configure the **startup.cs** class to handle such exceptions and show the desired error message.

**startup.cs** class configuration:

services**.**Configure**<**ApiBehaviorOptions**>(**options **=>**

**{**

options**.**InvalidModelStateResponseFactory **=** actionContext **=>**

**{**

var errors **=** actionContext**.**ModelState

**.**Where**(**e **=>** e**.**Value**.**Errors**.**Count **>** 0**)**

**.**SelectMany**(**x **=>** x**.**Value**.**Errors**)**

**.**Select**(**x **=>** x**.**ErrorMessage**).**ToArray**();**

var errorResposne **=** **new** ApiValidationErrorResponse

**{**

Errors **=** errors

**};**

**return** **new** BadRequestObjectResult**(**errorResposne**);**

**};**

**});**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **This is very important.** If we want to handle the error message based on the environment, we will create a new class as a Middleware **ExceptionMiddelware.cs**. We would create a new **MiddleWare** folder for this class.

**public** class ExceptionMiddelware

**{**

**private** **readonly** RequestDelegate \_next**;**

**private** **readonly** ILogger**<**ExceptionMiddelware**>** \_logger**;**

**private** **readonly** IHostEnvironment \_env**;**

**public** ExceptionMiddelware**(**RequestDelegate next**,**

ILogger**<**ExceptionMiddelware**>** logger**,**

IHostEnvironment env**)**

**{**

\_next **=** next**;**

\_logger **=** logger**;**

\_env **=** env**;**

**}**

**public** **async** Task InvokeAsync**(**HttpContext context**)**

**{**

**try**

**{**

**await** \_next**(**context**);**

**}**

**catch** **(**Exception ex**)**

**{**

\_logger**.**LogError**(**ex**,** ex**.**Message**);**

context**.**Response**.**ContentType **=** "application/json"**;**

context**.**Response**.**StatusCode **=** **(**int**)**HttpStatusCode**.**InternalServerError**;**

var response **=** \_env**.**IsDevelopment**()**

**?** **new** ApiException**((**int**)**HttpStatusCode**.**InternalServerError**,** ex**.**Message**,** ex**.**StackTrace**.**ToString**())**

**:** **new** ApiException**((**int**)**HttpStatusCode**.**InternalServerError**);**

var options **=** **new** JsonSerializerOptions**{**

PropertyNamingPolicy **=** JsonNamingPolicy**.**CamelCase

**};**

var json **=** JsonSerializer**.**Serialize**(**response**,** options**);**

**await** context**.**Response**.**WriteAsync**(**json**);**

**}**

**}**

**}**

Also, update the **Startup.cs** class for the above-mentioned class.

**public** void Configure**(**IApplicationBuilder app**,** IWebHostEnvironment env**)**

**{**

app**.**UseMiddleware**<**ExceptionMiddelware**>();**

**}**

**Use Sqlite Database**

1. Update the **appsettings.Development.json**

"ConnectionStrings": {

"DefaultConnection" : "Data source=skinet.db"

}

1. Update the **Startup.cs** class

**public** void ConfigureServices**(**IServiceCollection services**)**

**{**

services**.**AddControllers**();**

services**.**AddAutoMapper**(typeof(**MappingProfiles**));**

**services.AddDbContext<StoreContext>(x =>**

**x.UseSqlite(\_configuration.GetConnectionString("DefaultConnection"))**

**);**

**}**

**Use Static files in API**

1. Create a **wwwroot** folder and we would add the static files like images in this folder
2. Update the **Startup.cs** class

**app.UseStaticFiles();**