**Basic api design:**

-e-commerce app:

Create (post)

/api/products

read (get)

//api/products

Read a product (get)

/api/products / {id}

update (put)

//api/products

delete (delete)

/api/products

**Frameworks and middleware:**

Every middleware is put in the pipeline(framework) and every middle ware has a function in the pipeline, the framework has to be flexible to configure different middle ware

**App is an instance that refers to the framework:**

App.useMiddleware1 ();

App.useMiddleware3 ();

App.useMiddleware12 ();

// to allow devs to use defrint md

**Before using the middleware we have to inject the dependencies of the middleware:**

App.AddDependncies<IClass1 , class1>(); //for class1 use the class 1 instances

App.AddDependncies<IClass2 , class2>();

**We can also configure the behavior of the middleware for example:**

App.UseAuthorize();

**We can also configure the Authorization middleware:**

App.ConfigureAutherization (… a collection of parameters to tell how the auth should work …)

**In conclusion there are 3 things a framework should have:**

1. **Add dependencies**

App.AddDependncies<IClass1 , class1>(); //for class1 use the class 1 instances

App.AddDependncies<IClass2 , class2>();

1. **Add middleware**

App.UseAuthorize();

App.useMiddleware1 ();

App.useMiddleware3 ();

App.useMiddleware12 ();

1. **Configure how the middleware behave**

App.ConfigureAutherization (… a collection of parameters to tell how the auth should work …)

This should be called before the middleware

**Creating new web app:**

1. Open vs
2. Create new project
3. Choose asp .net core web app
4. Give the project a name and a solution name then create

**Web app content:**

Program.cs:

* Contains a console app the runs in a loop in main, it creates a hosts that runs constantly, only after the middleware config is done it will run and listens to the ports

* Contains a method (IHostBuilder CreateHostBiulder ) that confiuers the middleware using the startup.cs content

Startup.cs:

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

Configures the dependencies and configures how each middleware work

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

Has all the middleware to add in the pipeline, it has an excpetion page.

UseRouting() to determine which function to call , its mandetroy for the configure method to have a routing function

UseEndponts(…) configures the end points which gets the id from the request

**Web Api solution navigation:**

* **Connected services:**

Connects the api project to external services example azure or sql databases by adding a service dependency and choosing the service

* **Analyzers:**

Is the configured codes for the api example: styles

* **Frameworks:**

Contains all deferent Libraries of .net core

* **Packages:**
* Contains all nugets packages that can be downloaded and customizes (custom libraries)
* **Properties**:

Defining deferent profiles for the vsc to run the project

* **Controllers:**

Contains all default and user defined controllers

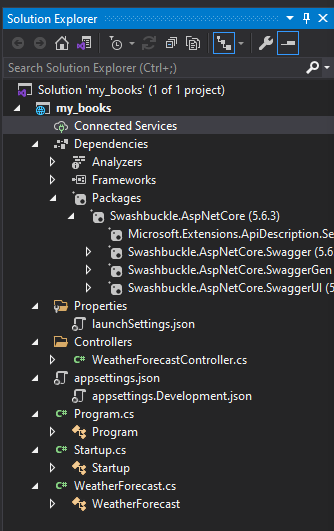
* **Appsetting.json:**

App config files to store config setting to store the db connections

* **Program.cs:**

entry for the api

* **Startup.cs:**
* Config services and the app requests



**Models:**

Models are just c# classes that represents the EDM database. Each model represents a table and each property in the class represents a column of the table

**Creating a model:**

1. Create a new class in the models folder with the name of the table as the name of the class
2. Add properties representing the rows
3. Adding an ? before the name of the property to make it an optional prop
4. Next, the model need to be connected to the EF database shownin the next steps..

**Entity frame work database context;**

A bridge between entity classes (c#) and the database tables (sql)

**Connecting a model to the EF database:**

1. **Install the package nugget (**Microsoft.entityfarmeworkcore) from tools>>nugget package manager>> console or packages from solution
2. Create a database context file by creating a new class in the data folder with name of AppDbContext .cs
3. The AppDbContext class needs to be inherted from DbContext class
4. Add using Microsoft.EntityFrameworkCore; to be used by DbContext class.
5. Add a constructer that takes the DbContextOptions <> that takes asa a parameter the AppDbContext then pass these value to the base class aka DbContext class 🡪 public AppDbContext(DbContextOptions<AppDbContext> options):base(options) {}
6. Define the table names inside the constructer for the c# model that takes the c# class “book” as an entity model and define the Sql table “Books” with a get and set to take and send data to the database **🡪**  public DbSet <Book> Books { get; set; } //<book> is the c# model and variable Book is the name of the table
7. use the libraries of the defined model example: using my\_books.Data.Models;
8. go to AppSettings.json that stores the app rlated settings to add the database connection string to config the bd to work with the api 🡪 "AllowedHosts": "\*"

"ConnectionStrings": {

"DefaultConnectionString": " fake-db-connection"}

1. in startup.cs ,inside the startup class create a property connectionString, and in the startup contructor creaete 🡪 ConnectionString = Configuration.GetConnectionString("DefaultConnectionString");
2. in ConfigrationServices configure the DbContext with the sql db using 🡪

services.AddControllers();

services.AddDbContext<AppDbContext>(options => options.UseSqlServer(ConnectionString)); and include the books library

1. in nugget packet console type: install-package Microsoft.EntityFrameworkCore.SqlServer and use : using Microsoft.EntityFrameworkCore;

**Entity Frame work Migration:**

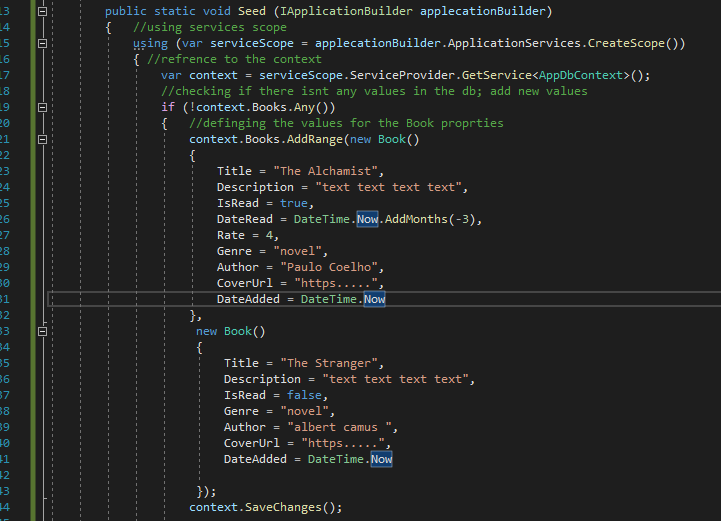
Provides a way to incrementally update the database schema to keep in sync with the application data model while preserving existing data in the data base

**Creating the database and the book table using the EF frame work:**

1. to work with the EF migration first download the Microsoft.EntityFrameworkCore.Tools, in the nugget packeg manager console write: Install-Package Microsoft.EntityFrameworkCore.Tools
2. create the database using server explorer>> databse connections >>right click>>create new database >>server name >>name the database as desirable >>
3. to add tables to the new database server : right click>>properties>>copy connection string>> go to appsetting.json >>paste value in default connection string
4. Add the migrations: in package manager console>> Add-Migration migrationName .. This will ccreate a c# class with the name of the migration which inherets from base class migration, defined a new table that has the properties from the model as colomns , it sets the ID for the table based on the name of the property if it has id in it and of type int , this gets added to the migration foler and gets updated with future migrations
5. The file AppDbContextSnapshot.cs is generated with the migration, it gets updated with other migrations it enables the EF core to calculate the changes to bring the db up to date , to update the changes on the database in manager console type: Update-Database
6. Go to the server explorer and refresh the db now in tables folder will be the new created table form the c# models, right click on the table to show table data

**Seeding the database**

1. Create a new class in the data folder named AppDbInitializer that has a function to seed the database if the databse is empty with a parameter of type IApplecationBuilder that has an if statment to check if the table is empty then adds new values to the table :



1. In start up, in configuration method at the end add: AppDbInitializer.Seed(app);
2. Run the program
3. Go to server explorer , refresh the table the view table data thst contains the new added values.

**Controllers:**

Receives HTTP requests and sends HTTP response

**Adding new controller:**

1. Right click on controllers folder>> add>>controller>>API >>select API controller –empty>> name the controller and add.
2. Next add the API end points..

**API end points:**

* **Sending new data [HttpPost]:**

Adds new values to the database

* **Add a service:**

1. create a new folder in the data folder called services >> add a new class (exp: booksService)
2. Add a reference for the AppDbContext file by defining a new variable with the type of AppDbContext and pass the value as a parameter to the class’s constructer

public class BooksService

{

private AppDbContext \_context;

public BooksService(AppDbContext context)

{

\_context = context;

}

}

1. Create a folder in the data folder with the name ViewModel then create a model view class in the folder to specify which properties the user can provide values for, the properties are taken from the model
2. Create a method that inputs new values to the database with the parameters of type view model that has an object of the model:

public void AddBook(BookMV book)

{

var \_book = new Book()

{

Title = book.Title,

Description = book.Description,

IsRead = book.IsRead,

DateRead = book.IsRead ? book.DateRead.Value : null,

Rate = book.IsRead ? book.Rate.Value : null,

Genre = book.Genre,

Author = book.Author,

CoverUrl = book.CoverUrl,

DateAdded = DateTime.Now,

};

\_context.Books.Add(\_book);

\_context.SaveChanges();

* **Configure the service**

1. In startup, in the configServices method after AddDbContedct servies add : services.AddTransient<BooksService>();

* **Add api endpoint**

1. In the controller inject the service by adding a variable with the type of the service
2. Create a constructor and pass the parameter as the type of the service
3. Add the http post

public BooksService \_booksService;

public BooksController(BooksService booksService)

{

\_booksService = booksService;

}

[HttpPost("add-book")]

public IActionResult AddBook ([FromBody]BookMV book)

{

\_booksService.AddBook(book);

return Ok();

}

}

* **Test the http post**

Save changes>> debug program>> in swaqqer browser window click try>> add value the click execute>> when the post succeeds stop the program and check the database table for the new added values

**Retrieving data [HTTPGet]:**

* **Adding the service method:**

1. In service class after the previous post method, add a new method of type list <model> methodName (), it doesn’t take any parameters only retrieves a list of all books in the table

public List<Book> GetAllBooks()

{

var allBooks = \_context.Books.ToList();

return allBooks;

}

Or

public List<Book> GetAllBooks() => \_context.Books.ToList();

1. Create another method that only retrieves a single book with the id of the books model as a parameter

public Book GetBookById(int bookId) => \_context.Books.FirstOrDefault( n => n.Id == bookId);

* **Add API endpoints to controller to get data:**

Create 2 get methods:

1. to retrieve all books

[HttpGet("get-all-books")]

public IActionResult GetAllBooks()

{

var allBooks = \_booksService.GetAllBooks();

return Ok(allBooks);

}

1. for retrieving a single book:

[HttpGet("get-book-id/{id}")]//the {id} must match the name of the parameter

public IActionResult GetBookById(int id)

{

var book = \_booksService.GetBookById(id);

return Ok(book);

}

* **Testing the get api :**

1. Run the app>> in swagger browser window >> in get methods >> click try it out the execute , if 200 then it’s a success

**Updating data [HttpPut]:**

* **Adding service method:**

1. In services class add a method that retreaves a book by its id then check if the book exist to apply changes

public Book UpdateBookById(int bookId, BookMV book)// bookMV type to update only the specific proprties in the bookMV

{

//check if book exist

var \_book = \_context.Books.FirstOrDefault(n => n.Id == bookId);

if (\_book!=null)

{

\_book.Title = book.Title;

\_book.Description = book.Description;

\_book.IsRead = book.IsRead;

\_book.DateRead = book.IsRead ? book.DateRead.Value : null;

\_book.Rate = book.IsRead ? book.Rate.Value : null;

\_book.Genre = book.Genre;

\_book.Author = book.Author;

\_book.CoverUrl = book.CoverUrl;

\_context.SaveChanges();

}

return \_book;

* **Add endpoint in controller:**

[HttpPut("update-book-by-id/{id}")]

public IActionResult UpdateBookById (int id , [FromBody] BookMV book)

{

var updatedBook = \_booksService.UpdateBookById(id, book);

return Ok(updatedBook);

}

* **Test API**