Genoom Simpsons Family Tree

# Preface

This exercise consists on creating:

* Data structure to store the Simpsons family tree
* Web API to access the tree.

Goals:

* Good analysis of the requisites of the project.
* Have a good starting point for the project, that means:
  + Documentation
  + Tests
* Demonstrate the software architecture skills creating a good project structure and allow evolving the project in the future.
* Demonstrate the developer skills using the latest technologies that will be required in the position:
  + .net Core, Web Api, Azure, MongoDb …

# Database design

This is the **key topic** to analyze for this exercise. The schema of the data seems simple at first sight but digging into it, we face the following problems:

* Each family member (node in a graph) has 3 levels for navigation:
  + Up 🡪 Parents
  + Down 🡪 Children
  + Side 🡪
    - Partner
    - Sibling
* Each relation between family members, is **symmetric**, that means that automatically in a standard RDBMs would **duplicate the amount of entries needed**.
* There is the transitive property between the family members. In the exercise, should be taken in account for getting the tree.
  + We should use **recursive calls** to navigate through the parents tree, while keeping the code needed low.

## SQL Server (or any other standard RDBMS)

* Pros:
  + Pretty standard, very easy to work with because is ubiquitous.
  + Well supported in Azure.
  + Good and stable tools and documentation.
  + ACID compliant.
* Cons:
  + We generate a lot of entries in the table pretty quick due to the symmetric of nature of the data.
  + We need a good set of indexes to not income with serious performance problems soon.
  + We need to set data partitions to manage the large amount of data.
  + We should avoid large transactions for updating the data at all (should not be a big problem because the Create/Update operation is adding a new children member to an existing one directly without needing to traverse large trees).

## MongoDb (NoSQL document based Dbs)

* Pros:
  + The cost of accessing a family member and its direct tree levels is 1. As the document should contain this info.
  + The returning data format is JSON (even better BSON), we could use it in combination with **node.js server so no extra serialization is needed and would be very performant**.
* Cons:
  + Not as good support in Azure as the SQL Server.
  + The documents could become big (be aware of 10MB limit per doc).
  + Costly Add/Update operations because the large documents. (More than the SQL Server).

Should not be a big problem because most of the operations are for reading the tree not updating it.

* + Increased complexity, we should be aware of the **CAP theorem (**Consistency, Availability, Partition tolerance.)

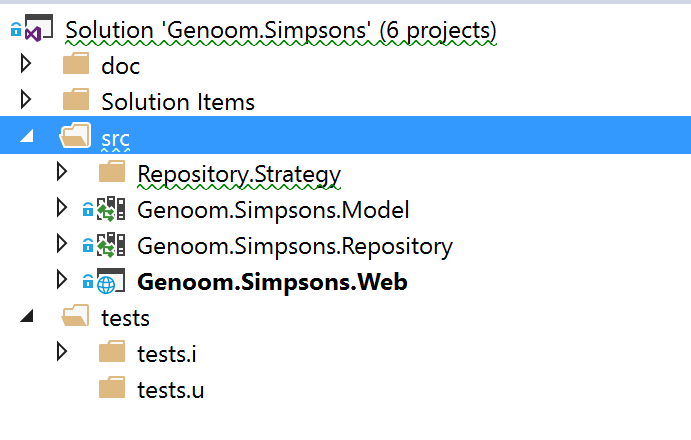
# Project architecture proposal:

## Logic and Code strategy

Before analyzing the architecture let’s set some points:

* There are 4 responsibilities for the project:
  + **Web stuff**: Web Api, Controllers, data serialization to JSON, etc.
  + **Business logic**: like checking if the person has a partner when adding a child.
  + **Data access repository**: to access the data here due to the topics discussed above, we could use a strategy like pattern so we can access the data on SQL Server and MongoDb and we could compare which option is better, also would allow creating other repositories for other technologies.
  + **Model**: simple C# POCOs representing the family members.

## Solution Structure Diagram



For organizing the code I’ve decided the following structure:

* **doc**: contains the documentation for the project, currently this document.
* **src**: contains the source code projects.
* **tests**:
  + **tests.i**: contains the integration tests. Actually, contains the **tests for the repositories** of Sql and MongoDb, because makes sense to test against the real database.
  + **tests.u**: contains the unit tests, they should not have any dependency to a real database so we need to mock those dependencies.

Actually, contains the **unit tests for the web controllers** with mocked in memory data structures.

* + For integration tests of the controllers, due to time constriction, I’ve decided to **rely on swagger** to call the API methods.

## Project Architecture Diagram



## Genom.Simpsons.Web:

* This project contains all the code related to the Web like the Web Api code, and Controllers.
* This will be the entry point for the program.
  + **It also contains all the initialization code for wiring up. Like the DI container**
* **As the business logic is quite simple, for now it’s included here in the controller**.

It should be evaluated if in the future gets more complex to move it to another project: Genoom.Simpsons.Business

* On PeopleController > Post /people/{id}/children, **there is a change in the logic implementation state in the exercise**:

The original was: “The person must already have a partner and the partner id must also be supplied”.

But to make easier for the Api users there is no need to provide any partner id and it is internally managed.

## Genom.Simpsons.Repository:

* Contains only the interface for the People Repository, it’s done this way to avoid adding extra references and avoid circular references to the other projects.

## Genom.Simpsons.Repository.Sql:

Contains the implementation of the People Repository in SQL Server, but it could be used another RBDMS instead.

I’ve decided to use **Dapper** as **light ORM**. Because it’s integrated well with .net core and specially because the **performance is a lot better than Entity Framework**.

The database schema is located at: Datamodel\ GenoomSimpsons.Schema.sql

## Genom.Simpsons.Repository.MongoDb:

Contains the implementation of the People Repository in MongoDb.

**Currently, this is a work in progress and not implemented yet**

## Genom.Simpsons.Model:

Contains the Model for the application as POCOs.

One important note here is:

* Naturally in a real program, we should use a real Key unique for identifying a family member (a guid, a unique integer, etc.)
* For this exercise, due it’s nature and also in sake of simplicity, I’ve decided to use the **Name as Key**, instead.

# Getting Started

## Requisites

* Net core 1.0
* Sql Server or MongoDb servers
* IIS express or IIS to host and launch the website.

## Where to download the code

<https://github.com/albertgimenez/Genoom>

## Application Settings and Configuration

In the **Genoom.Simpsons.Web**, open the file **appSettings.json**, some key points of it are:

* Decide wich strategy we want to use (database):
  + **Sql**: to use SQL Server, please check the **SqlConnection** settings as well.
  + **MongoDb**: to use MongoDb, please check the **MongoDbConnection** settings and you will need also to review the **MongoDbConfig** section as well.

{

  "DbStrategy": "Sql",

  "ConnectionStrings": {

    "SqlConnection": "Server=.;Database=GenoomSimpsons;Trusted\_Connection=True;MultipleActiveResultSets=true",

    "MongoDbConnection": "mongodb://localhost:27017"

  },

  "MongoDbConfig": {

    "Database": "GenoomSimpsons",

    "Collection": "SimpsonsFamilyTree"

  },

  "Logging": {

    "IncludeScopes": false,

    "LogLevel": {

      "Default": "Debug",

      "System": "Information",

      "Microsoft": "Information"

    }

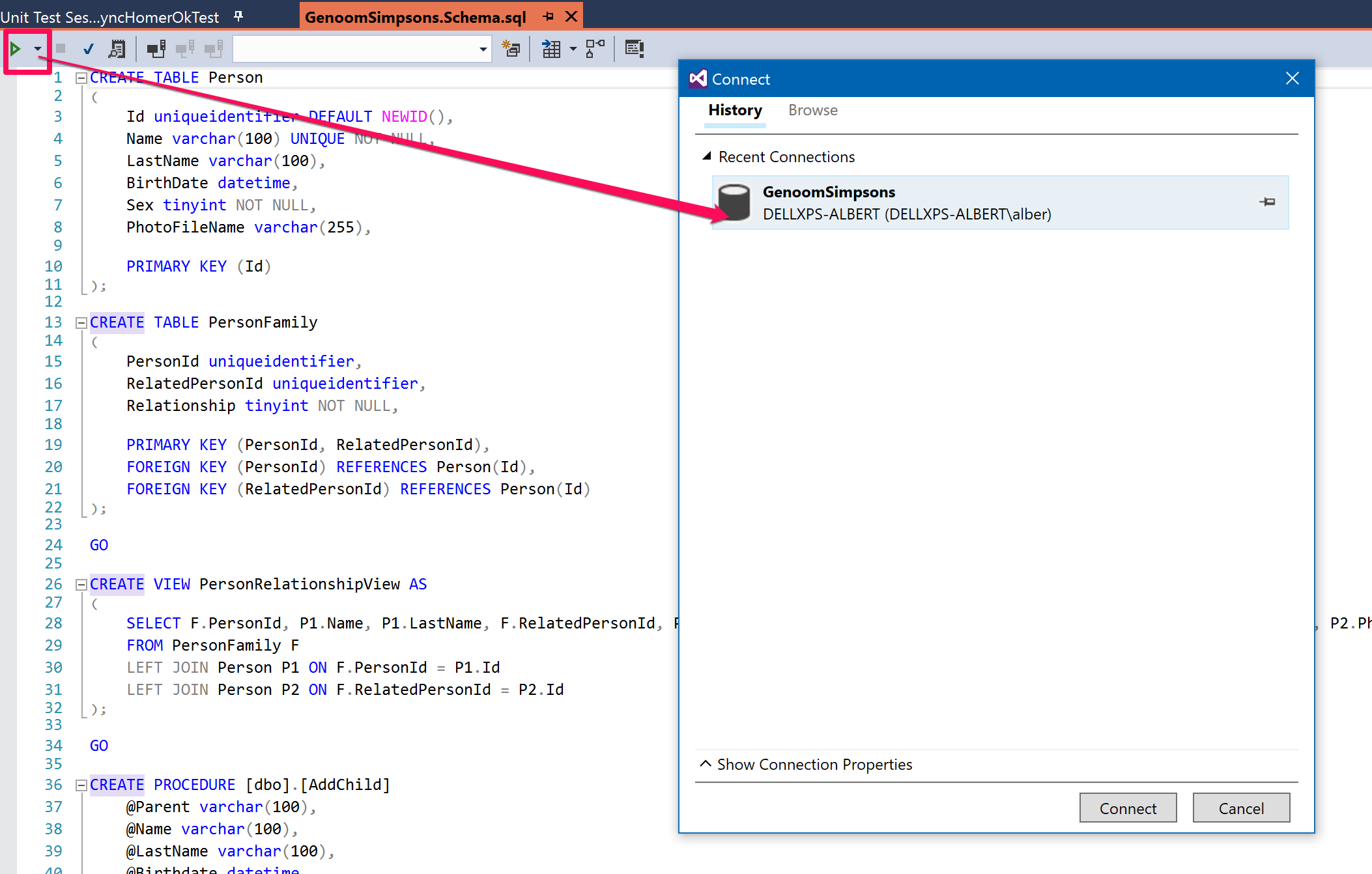
  }

}

## Creating the sample database (SQL Server)

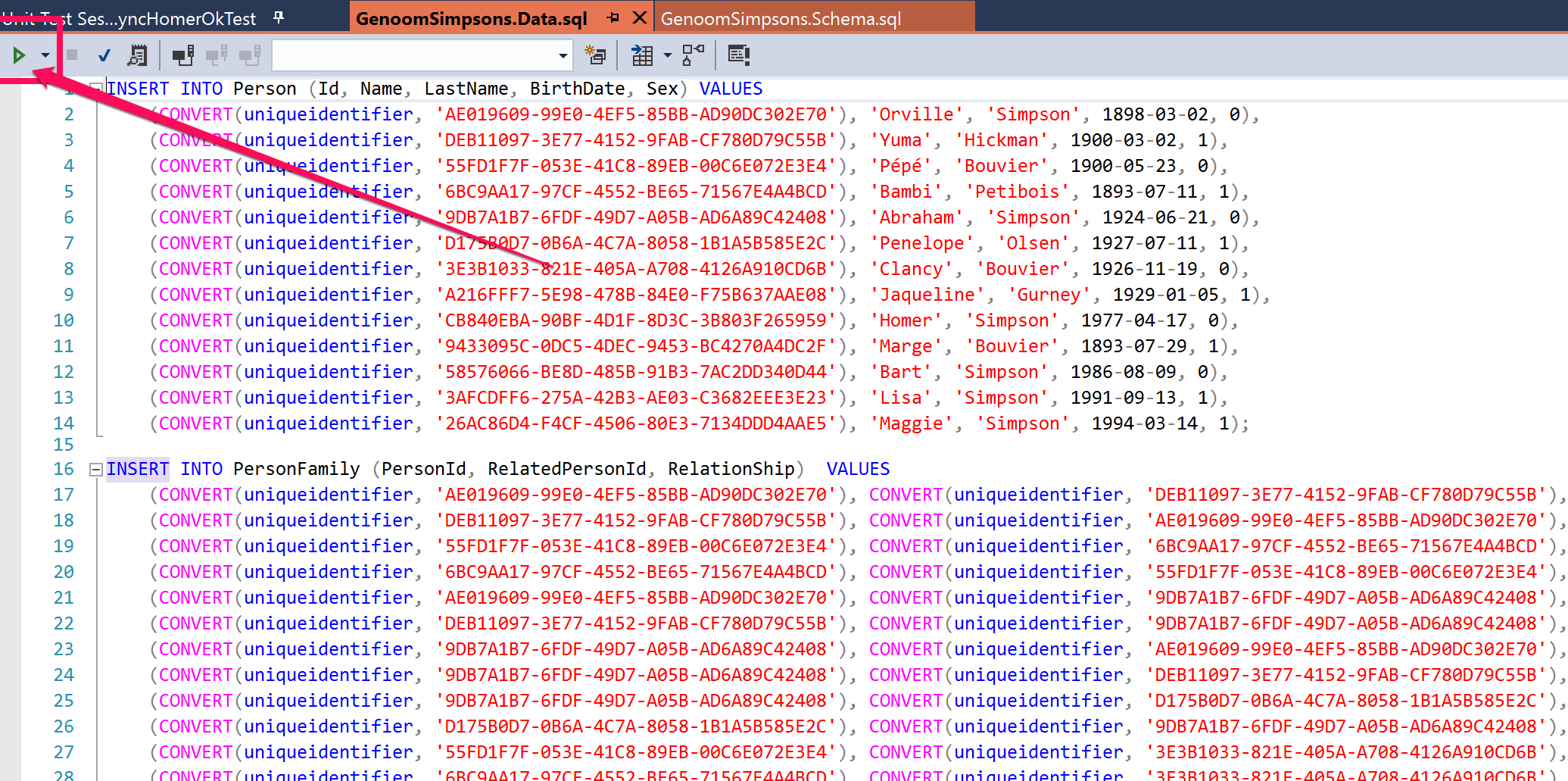
On Genoom.Simpsons.Repository.Sql > Datamodel > Open GenoomSimpsons.Schema.sql

You can execute it from inside Visual Studio:



After that, you can seed the database with the sample data provided:

On Genoom.Simpsons.Repository.Sql > Datamodel > Open GenoomSimpsons.Data.sql



## Startup and DI services registration

To change and configure the DI services injected and the default api routes configuration, check the **Genoom.Simpsons.Web** and open the file **Startup.cs**, some key points of it are:

* **ConfigureServices**: here we add and register the services in the DI, we rely on the new but very simple DI offered by Microsoft.
  + The Swagger Documentation service is added here
  + The repository (sql, mongodb) instance is added here.
* **Configure**: here we configure the services:
  + The Swagger path and endpoint
  + We set a **default error controller** for requests that are nt any of the valid controllers developed (for the 404 errors)

public void ConfigureServices(IServiceCollection services)

{

        …

        // Swagger documentation API

        services.AddSwaggerGen(c =>

        {

            c.SwaggerDoc("v1", new Info { Title = "Genoom Simpsons Tree", Version = "v1" });

        });

        // The database provider (strategy) to use to access the data.

        services.AddSingleton<IPeopleRepository>(Support.PeopleRepositoryFactory.Create(Configuration));

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

        …

        // Swagger API doc

        app.UseSwagger();

        app.UseSwaggerUI(c =>

        {

            c.SwaggerEndpoint("/swagger/v1/swagger.json", "Genoom Simpsons Tree v1");

        });

        // The default routes, by default if does not exist (404) we want to provide a nice response.

        app.UseMvc(routes =>

        {

            routes.MapRoute(

                name: "Error404",

                template: "{\*url}",

                defaults: new { controller = "Error", action = "Handle404" }

            );

        });

    }

}

## Routes

For this exercise is set that the api calls are direct to the controllers like /people instead of /api/v1/people

I’ve followed the route pattern, but if there is intention to continue evolving this project, we should change it to follow the Web Api good practice: **/api/v{xx}/controller**

This will make easier to version the api and handle changes on it while keeping compatibility backwards.

There are two routes levels:

* **Default routes**: set in the Configure Method in Startup.cs
* **Per controller routes**: this allows clear and fine grained control over the routes.

For this exercise because there are few controllers I thought is the best option because it’s clear. You will see that the controller has this decorator in the class declaration:

[Route("[controller]")]