**BUNTS SANGHA'S**

**S.M. SHETTY COLLEGE OF SCIENCE, COMMERCE &**

**MANAGEMENT STUDIES, POWAI MUMBAI – 400076**


**A PRACTICAL JOURNAL**

**CLASS: M.Sc.I.T Part-1(2022-23)**

**Semester:** Sem-2

**Subject: Micro-service Architecture**

**Submitted by: VIKAS OMPRAKASH YADAV**

**Roll no: _____ Seat Number: _____**

## SUBMITTED TO

**DEPARTMENT OF INFORMATION TECHNOLOGY**

_____ _____ *Dr. Tushar Sambare*

**Subject In Charge** **External Examiner** **Coordinator**

FOR PARTIAL FULFILMENT OF DEGREE

**MASTER OF SCIENCE (INFORMATION TECHNOLOGY)**

(2022 - 2023)

# INDEX

| Pr. No. | Title | Date | Sign |
|---|---|---|---|
| 1 | Write the steps for installing .NET Core, Docker, Postman with appropriate explanation. | 04/02/23 | |
| 2 | Build and run a simple .NET Core console application. | 11/02/23 | |
| 3 | Build and run a simple .NET Core console application with Docker. | 18/02/23 | |
| 4 | Build and run a Team Service Testing example with integration testing with Docker. | 25/02/23 | |
| 5 | Build and run a Location Service (Data Service) with Docker to demonstrate Backing Service without Database. | 04/03/23 | |
| 6 | Building an ASP.NET Simple Core Web Application – Stock Quote example (Use console or Visual Studio). | 25/03/23 | |
| 7 | Build and run CRUD Microservice with Visual Studio to demonstrate Data Service using Database running on IIS Server. | 01/04/23 | |
| 8 | Build and run CRUD Microservice with Visual Studio to demonstrate Data Service using Database running on Docker. | 08/04/23 | |
| 9 | Build and run an Event-Driven ASP.NET Core Microservice Architecture | 15/04/23 | |
| 10 | Build and run application for Upload and Download Multiple Files Using Web API | 22/04/23 | |
| 11 | Demonstrate the use of ASP.NET Core 2.0 Session State in an Application | 29/04/23 | |
| 12 | Demonstrate the use of ASP.NET Core 2.0 MVC Layout Pages | 06/05/23 | |
| 13 | Demonstrate Simple Insert and Select (CRUD) Operation Using .NET Core MVC With ADO.NET And Entity Framework Core | 13/05/23 | |
| 14 | Demonstrate the use of Entity Framework (2), With .Net MVC, Database-First | 22/05/23 | |

# Practical N0. 1

**Aim:** Write the steps for installing .NET Core, Docker, Postman with appropriate explanation.

**Software requirement:**

1) **Dotnet Core**

- **Download and install**

  To start building .NET apps you just need to download and install the .NET SDK

  (Software Development Kit version 3.0 above). Download the dotnet from the link given below,

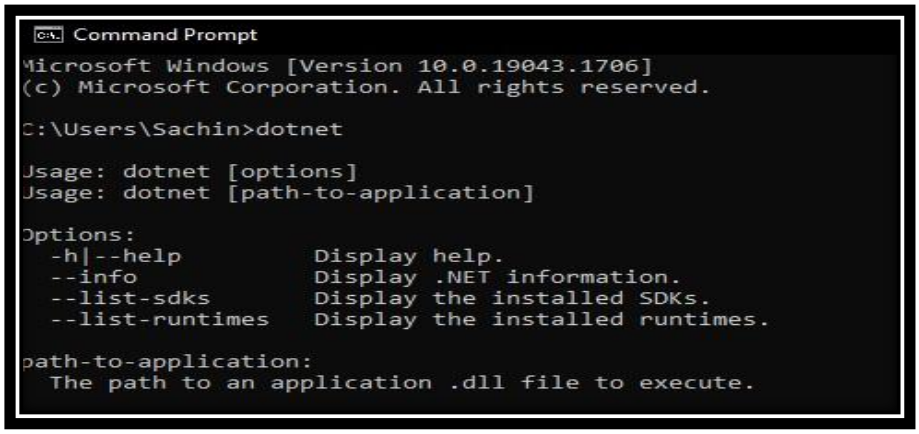
  **Link**: https://dotnet.microsoft.com/learn/dotnet/hello-world-tutorial/install

- **Check everything installed correctly**

  Once you've installed, open a new command prompt and run the following command:

  Command prompt > dotnet

  To check version of dotnet just type >Dotnet --version



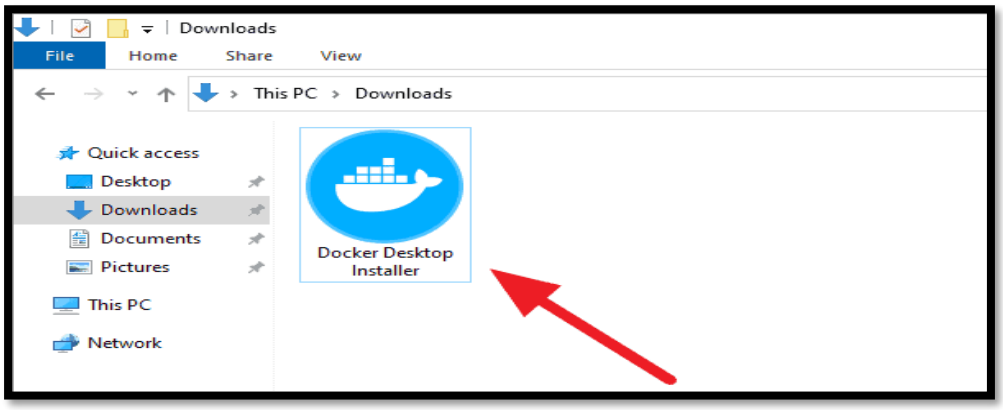

## 2) Docker

### ➢ Download and install

Download docker from the given link: *https://www.docker.com/products/docker-desktop/?ref=allthings.how*

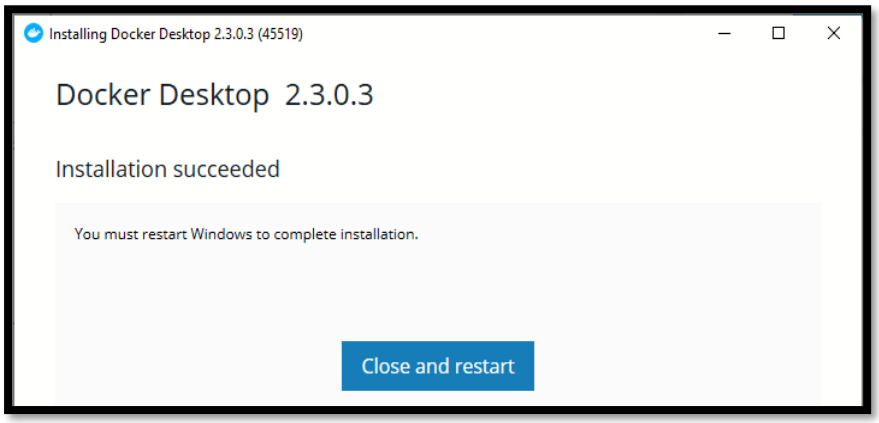
Click on 'Download for Windows (Stable)' button to download the installer file.


Then, go to downloads folder and double click on 'Docker Desktop Installer' setup file to initiate the installation process.




You will be presented with a configuration window in the setup process. Tick ‘Enable WSL 2 Windows Features’ if you are on Windows 10 Home edition or want to use Docker’s WSL 2 backend and Tick ‘Add shortcut to desktop’ if you want Docker Desktop shortcut, then press ‘Ok’ to begin the installation.


Once the Docker Desktop setup finished unpacking and installing the files, click on the ‘Close and restart’ button to complete the docker installation process.




**Enable Hyper-V or WSL?**

Now all that is left is to enable the Hyper-V or WSL depending upon the Windows 10 edition and version you have

- Windows 10 Pro, Enterprise & Education edition with **1703** update or later: If you are not on **2004** update or later, then only Hyper-V backend can be used.
- Windows 10 Home edition with **2004** update or later: Only WSL can be enabled as Hyper-V feature is not available on Home edition.
- Windows 10 Pro, Enterprise & Education edition with **2004** update or later: Both the Hyper-V & WSL can be enabled and used with docker.

**Enable Hyper-V**

Hyper-V is a native hypervisor for Windows 10 which can be used to create and run virtual machines. Hyper-V is on the route to becoming the legacy option to

run containers on Windows 10, as docker is planning to use WSL as its main backend to run containers.

But you still need Hyper-V if you want to run Docker native Windows containers. Thus, to enable Hyper-V, open the PowerShell as administrator and run the following command:

**Enable-WindowsOptionalFeature -Online -FeatureName $("Microsoft-Hyper-V", "Containers") -All**


PowerShell will prompt you to restart the computer to complete Hyper-V installation, type Y and hit enter to do the same. After rebooting the computer, you can run Docker Desktop and use containers.

**Enable WSL**

Windows Subsystem for Linux (WSL) is a compatibility layer which allows users to run Linux application natively on Windows 10. The Docker WSL backend allows users to run native Linux Docker containers on Windows without Hyper-V emulation.

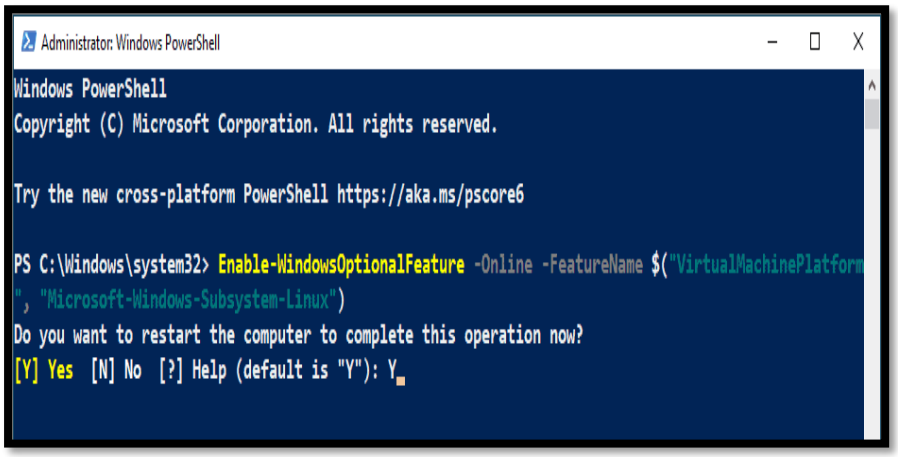
If you have the latest Windows 10 2004 update, then it is recommended to use the WSL as Docker backend as it performs better than Hyper-V backend. The Windows 10 Home edition users have no other option than using the WSL backend for Docker as Home edition don't have Hyper-V feature.

**Note**: If you have ticked 'Enable WSL 2 Windows Feature' in the setup, this command can be skipped as Docker Setup enables WSL automatically. Go to the 'Update WSL' section below to continue with the process.

Open PowerShell as administrator then run the following commands to enable WSL and 'Virtual Machine Platform' WSL component for Windows 10.

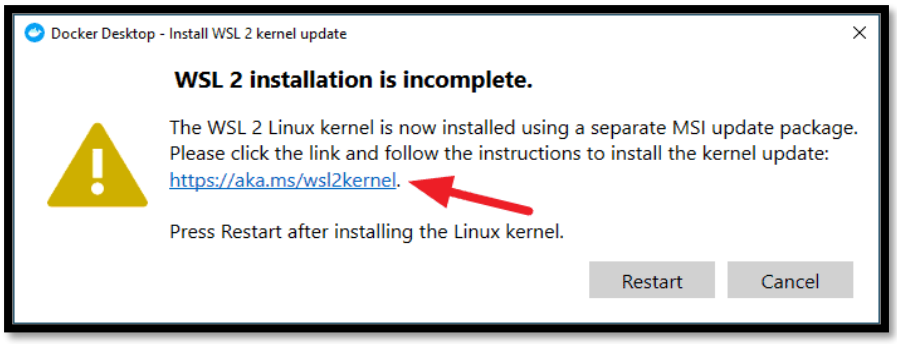
**Enable-WindowsOptionalFeature -Online -FeatureName $("VirtualMachinePlatform", "Microsoft-Windows-Subsystem-Linux")**

Press 'Y' and hit enter to restart the computer and complete the process.



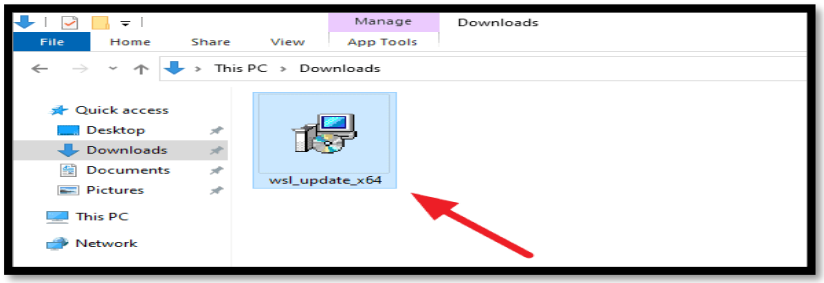

## Update WSL

After you have Completed the Docker Installation and restarted the system, when you run the Docker Desktop you will see an error as shown below.

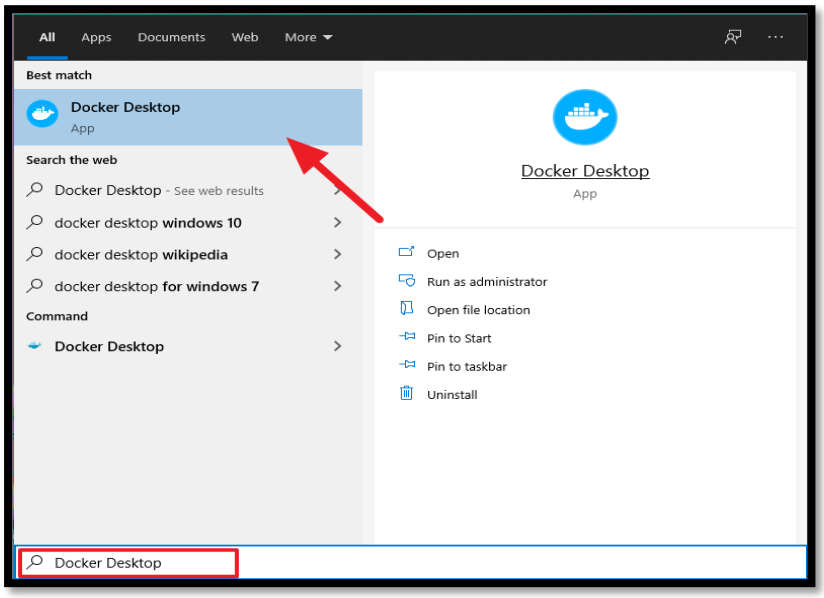



Click on this link or link in the error to go to Microsoft Docs page with the latest WSL2 kernel update. Then click on ‘download the latest WSL2 Linux kernel’ link on the page as shown below to download ‘wsl_update_x64’ setup file.

**Double click on the downloaded file and press 'yes' when prompted for permission.**



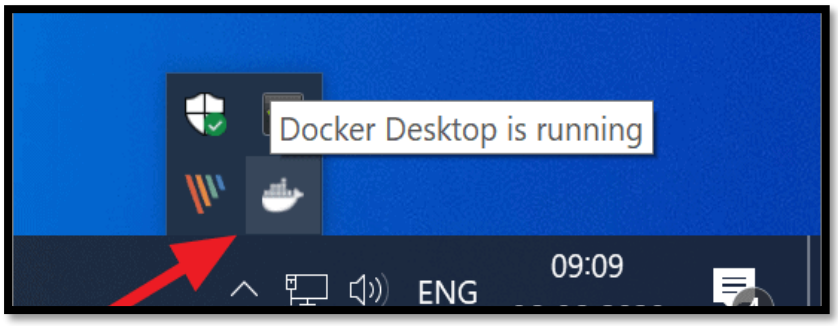

Once you've enabled and updated the WSL for Windows 10, you can run Docker searching for it in the Start menu.




- **Verify Docker Installation**

  The white whale in the system tray indicates that Docker is running. But you can also test your docker installation by opening the CMD and typing docker --version

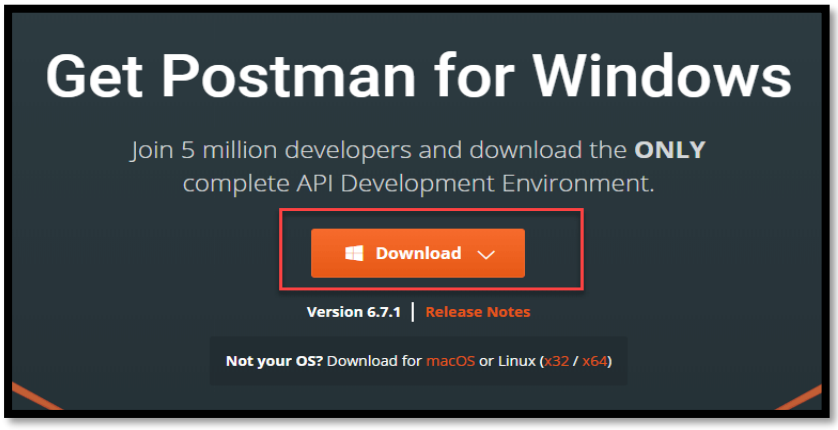
**C:\Users\ATH> docker --version**




### 3) Postman

- **Download and install**

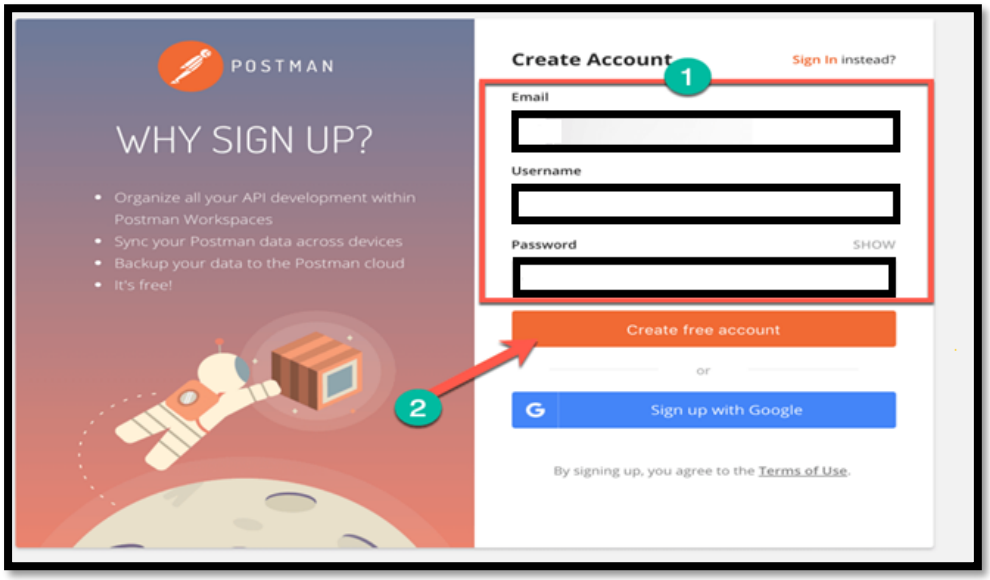
Go to https://www.postman.com/downloads/ and choose your desired platform then Click Download.




**Step 2) Click on Run**


**Step 3) Postman Installation Start**

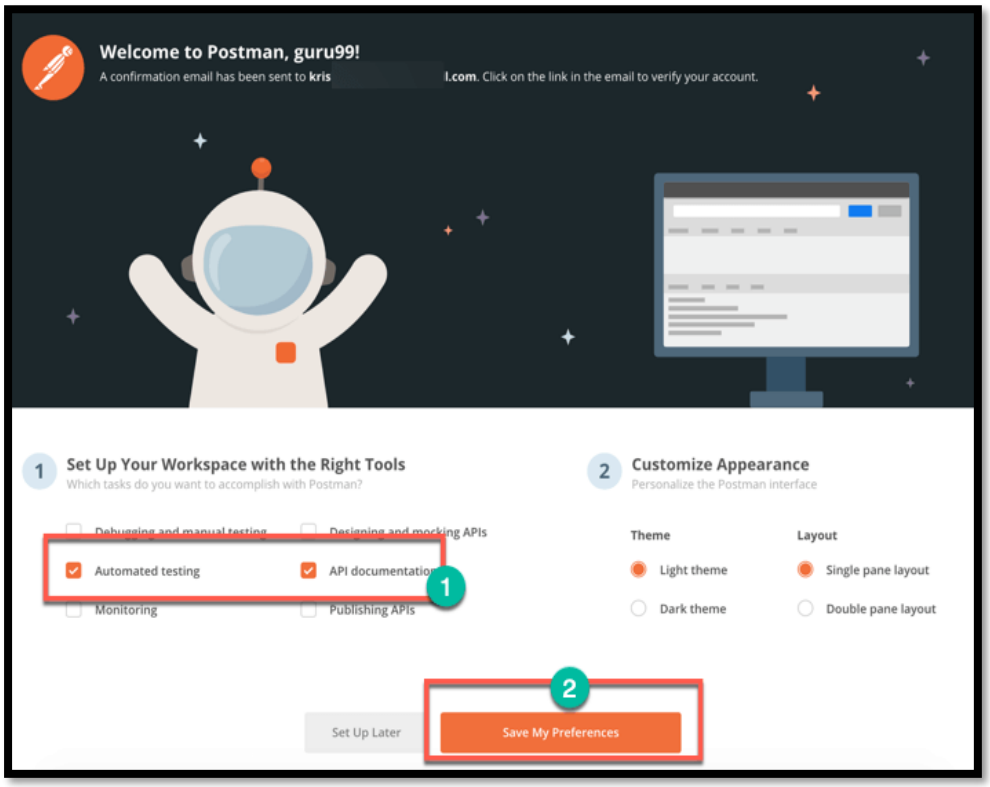

**Step 4) Signup for Postman Account**




**NOTE:** Though Postman allows users to use the tool without logging in, signing up ensures that your collection is saved and can be accessed for later use.

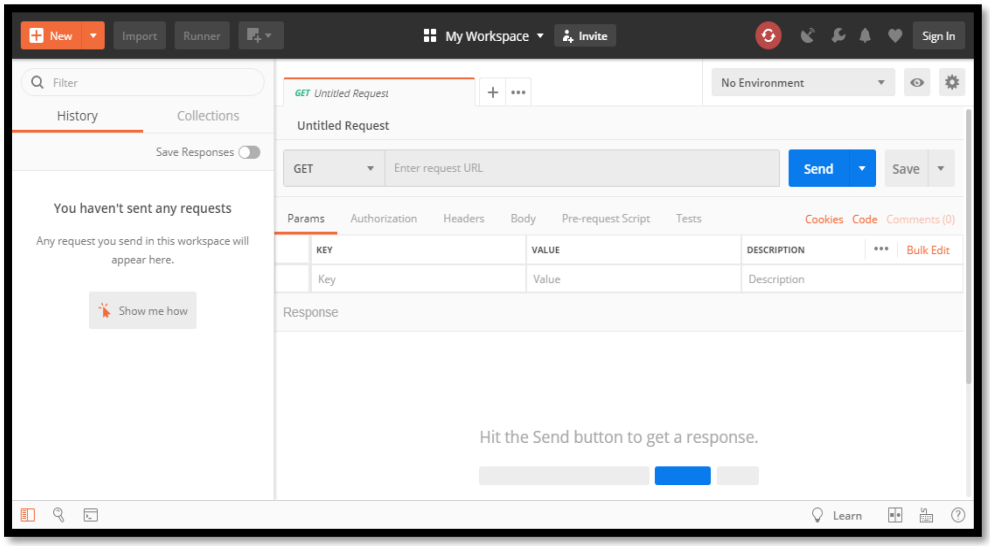
**Step 5) Click on Save My Preferences**

Select the workspace tools you need and click Save My Preferences




## Step 6) Congratulation!

You will see the Startup Screen



# Practical N0. 2

## Aim: Build and run a simple .NET Core console application.

**Theory:** A .NET Core console application is a type of application that runs on the .NET Core runtime and is designed to

be run from the command line. It is a lightweight application that typically performs a specific task or set of tasks, such as processing data, performing calculations, or interacting with a database.

- A .NET Core console application consists of a set of code files, including a Program.cs file that contains the entry point for the application. The entry point is typically a Main method that is responsible for starting the application and handling any command-line arguments that are passed to it.
- .NET Core console applications can be created using a variety of programming languages, including C#, F#, and Visual Basic. They can also make use of third-party libraries and frameworks, which can be added to the application using the NuGet package manager.
- When a .NET Core console application is run, it typically executes a set of instructions and then terminates, returning any output to the console. This makes

it a useful tool for performing batch processing tasks, automating repetitive tasks, or building command-line utilities.

This practical shows how to create and run a .NET console application in Visual Studio 2019.

**Steps: 1) Build .NET Core console application.**

open command prompt, run the following command to create your app:

- **dotnet new webapi -o MyMicroservice --no-https -f net7.0**

Then, navigate to the new directory created by the previous command:

- **cd MyMicroservice**

The following code shows the contents of the WeatherForecastController.cs file located in the Controllers directory:

```
WeatherForecastController.cs

using Microsoft.AspNetCore.Mvc;

namespace MyMicroservice.Controllers;

[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;
```

```
    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

## 2) Run a simple .NET Core console application.

In command prompt, run the following command:

- **dotnet run**

You should see an output similar to the following:

```
Command prompt

Building...
info: Microsoft.Hosting.Lifetime[14]
Now listening on: http://localhost:5020
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
```

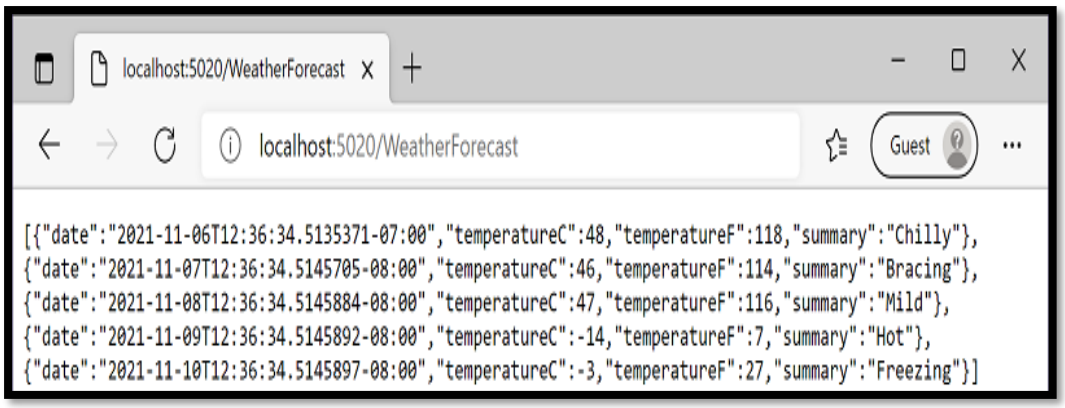
Wait for the app to display that it's listening on

**http://localhost:<port number>**,

and then open a browser and navigate to

**http://localhost:<port number>/WeatherForecast**.

In this example, it showed that it was listening on port 5020,




Congratulations, you've got a simple service running!

Press **ctrl+c** on your cmd to end the dotnet run command that is running the service locally.

# Practical N0. 3

**Aim:** Build and run a simple .NET Core console application with Docker.

**Theory:** Docker is a platform that enables you to combine an app plus its configuration and dependencies into a single,

independently deployable unit called a container.

To run with Docker image, we need **Dockerfile → a text file that contains instructions for how to build your app as a Docker image**. A Docker image contains everything needed to run your app as a Docker container.

To perform this practical we need to setup Docker first, perform prerequisite under heading Docker setup.

After successful installation of docker, perform following steps.

**Steps: 1)** Since we opened a new cmd during installation, we'll need to return to the directory we created our service in.

In command prompt, run the following command:

- **cd MyMicroservice**

**2)** Create a file called Dockerfile with this command:

- **fsutil file createnew Dockerfile 0**

**4)** You can then open it in your favorite text editor manually or with this command:

- **start Dockerfile**

5) Replace the content of the **Dockerfile** to the following in the text editor:

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build

WORKDIR /src

COPY MyMicroservice.csproj .

RUN dotnet restore

COPY . .

RUN dotnet publish -c release -o /app

FROM mcr.microsoft.com/dotnet/aspnet:7.0

WORKDIR /app

COPY –from=build /app .

ENTRYPOINT [“dotnet”, “MyMicroservice.dll”]
```

Note: Make sure to name the file as Dockerfile and not Dockerfile.txt or some other name.

**Optional: Add a .dockerignore file**

.dockerignore file reduces the set of files that are used as part of ‘docker build’

Fewer files will result in faster builds.

Create a file called .dockerignore file with this command:

- **fsutil file createnew .dockerignore 0**

You can then open it in your favorite text editor manually or with this command:

- **start .dockerignore**

Replace the content of the .dockerignore to the following in the text editor:

**Dockerfile**

**[b|B]in**

**[O|o]bj**

**Create Docker image**

Run the following command:

- **docker build -t mymicroservice .**

we can run the following command to see a list of all images available on your machine,

including the one we just created.

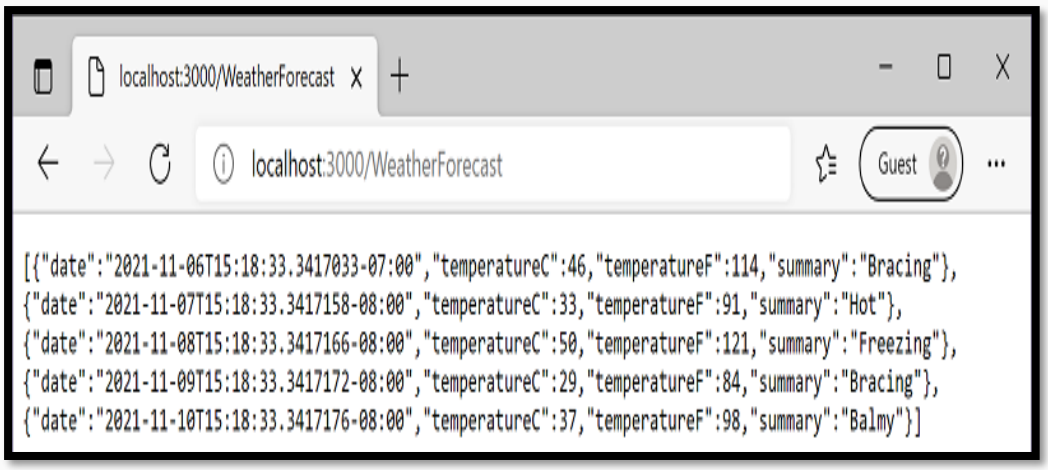
- **docker images**

**Run Docker image**

we can run your app in a container using the following command:

- **docker run -it –rm -p 3000:80 –name mymicroservicecontainer mymicroservice**

We can browse to the following URL to access your application running in a container http://localhost:3000/WeatherForecast




Optionally, we can view our container running in a separate command prompt using the following command:

- docker ps


Press **ctrl+c** on your cmd to end the docker run command that is running the service in a container.

Congratulations! You've successfully created a small, independent service that can be deployed and scaled using

Docker containers.

# Practical N0. 4

**Aim: Build and run a Team Service Testing example with integration testing with Docker.**

**Theory:**

## Team Service Testing:

Team Services is a cloud-based service that provides a suite of tools for managing software development projects. One of the features of Team Services is the ability to perform automated testing as part of the build process. This can include unit testing, functional testing, and integration testing.

**Integration testing:**

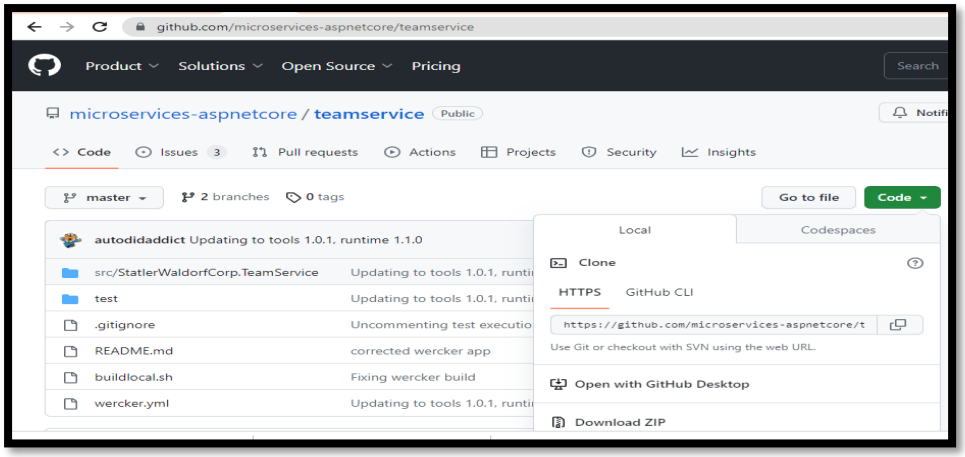
Integration testing is a type of testing that verifies that different components of an application work together as expected. One way to perform integration testing in a .NET Core application is to use Docker containers. Docker is a platform for building, shipping, and running distributed applications.

Hope you understand the concept Team Service Testing and Integration testing,

Now let's implement it practically. Follow bellow steps to perform Team Service Testing and Integration testing,

**Step 1: Download the teamservice folder from the github link:**

https://github.com/microservices-aspnetcore/teamservice




**Step 2: Integration Testing.**

Navigate the downloaded folder in cmd and type dotnet restore.

D:\roughmsa\teamservice-

master\test\statlerwaldorfcorp.TeamService.Tests.Integration> dotnet restore

```
D:\roughmsa\teamservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration>dotnet restore
  Determining projects to restore...
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\src\StatlerWaldorfCorp.TeamService\StatlerWaldorfCorp.TeamService.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
  All projects are up-to-date for restore.
```

**Step 3: after restoring type dotnet build command.**

```
D:\roughmsa\teamservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration>dotnet build
Microsoft (R) Build Engine version 17.2.0+41abc5629 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\src\StatlerWaldorfCorp.TeamService\StatlerWaldorfCorp.TeamService.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
  All projects are up-to-date for restore.
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\src\StatlerWaldorfCorp.TeamService\StatlerWaldorfCorp.TeamService.csproj]
  StatlerWaldorfCorp.TeamService -> D:\roughmsa\teamservice-master\src\StatlerWaldorfCorp.TeamService\bin\Debug\netcore
  app1.1\StatlerWaldorfCorp.TeamService.dll
  StatlerWaldorfCorp.TeamService.Tests.Integration -> D:\roughmsa\teamservice-master\test\StatlerWaldorfCorp.TeamServic
  e.Tests.Integration\bin\Debug\netcoreapp1.1\StatlerWaldorfCorp.TeamService.Tests.Integration.dll

Build succeeded.
```

**Step 4: Now type dotnet test command.**

```
D:\roughmsa\teamservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration>dotnet test
  Determining projects to restore...
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\src\StatlerWaldorfCorp.TeamService\StatlerWaldorfCorp.TeamService.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
C:\Program Files\dotnet\sdk\6.0.300\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.EolTargetFrameworks.targets(28,5): war
ning NETSDK1138: The target framework 'netcoreapp1.1' is out of support and will not receive security updates in the fu
ture. Please refer to https://aka.ms/dotnet-core-support for more information about the support policy. [D:\roughmsa\te
amservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration\StatlerWaldorfCorp.TeamService.Tests.Integration
.csproj]
  All projects are up-to-date for restore.
```

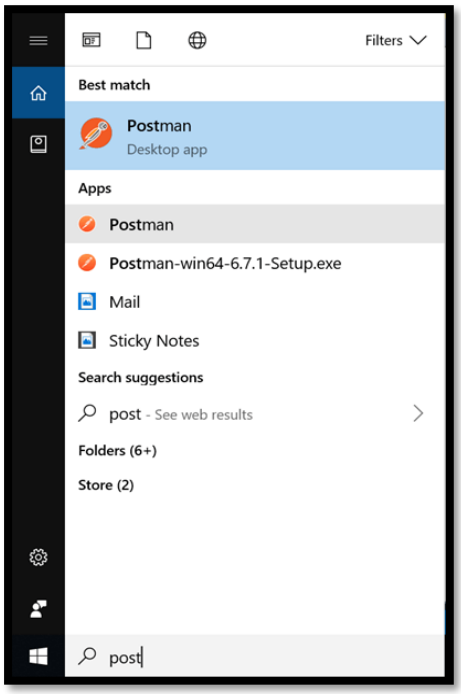
**Step 5:- We have tested and its working now we'll run example of team service.**

For that we require a docker image of the project. So turn on the docker and type docker run -p 8080:8080 dotnetcoreservices/teamservice command on cmd.

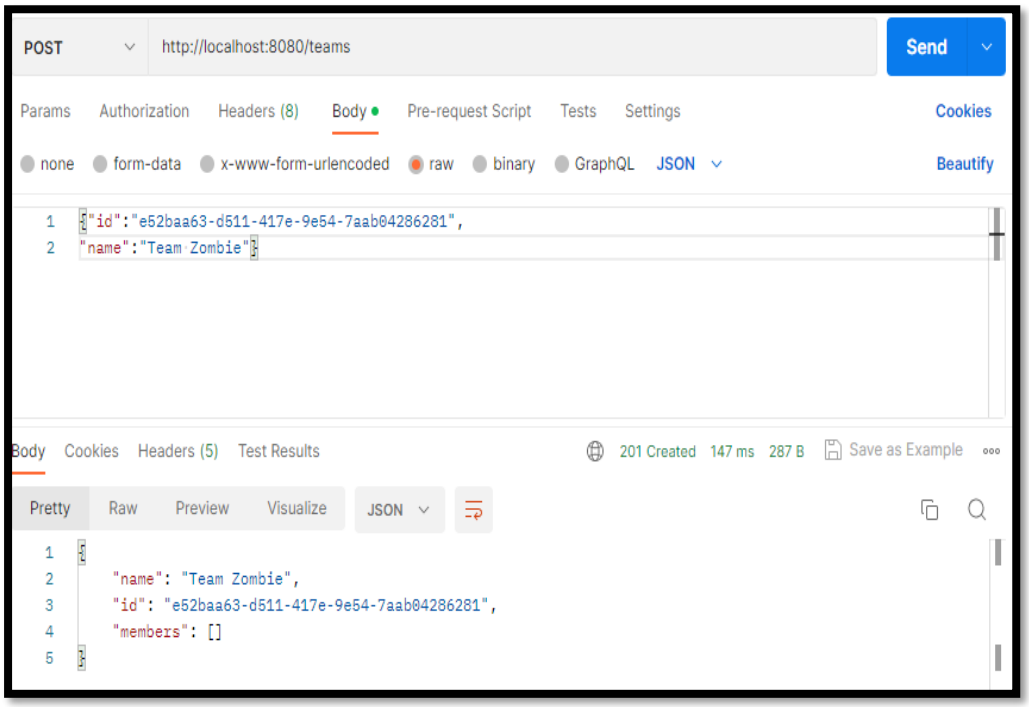
Copy the ip address, which says listening on: http://0.0.0.8080

```
D:\roughmsa\teamservice-master\test\StatlerWaldorfCorp.TeamService.Tests.Integration>docker run -p 8080:8080 dotnetcoreservices/teamservice
Unable to find image 'dotnetcoreservices/teamservice:latest' locally
latest: Pulling from dotnetcoreservices/teamservice
693502eb7dfb: Pull complete
081cd4bfd521: Pull complete
5d2dc01312f3: Pull complete
585880aea240: Pull complete
3905d0c644ea: Pull complete
c59037c90022: Pull complete
b14e24545f9c: Pull complete
Digest: sha256:fc48247a529f307e599ed19eac61f5637181905a0c46b9e9b10eb5cf62e8b855
Status: Downloaded newer image for dotnetcoreservices/teamservice:latest
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:8080
Application started. Press Ctrl+C to shut down.
```

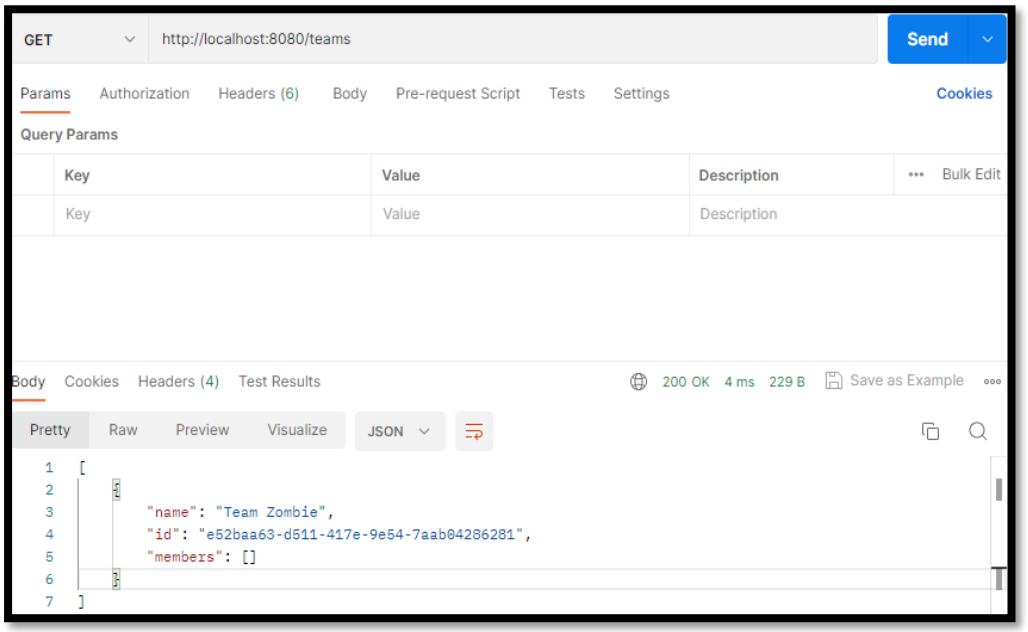
## Step 6: Now open postman




## Step 7: Now post the team id & team name as shown in the output below




**Step 8: To check whether it has been added or not we use GET method, the below output shows Get method.**



# Practical N0. 5

## Aim: Build and run a Location Service with Docker to demonstrate Backing Service without Database.

**Theory:** Location service is a type of microservice that provides location-based data to other applications. It can be used

for a variety of purposes, such as providing geolocation data for mapping applications, or providing location-based recommendations for e-commerce platforms.

To demonstrate a location service as a backing service without a database using Docker, follow these steps:

- Create a Dockerfile that specifies the environment for your location service, including the operating system, runtime, and any dependencies.
- Use the Dockerfile to build a Docker image of your location service.
- Set up a Docker Compose file that specifies the different containers that make up your application, including the container for your location service and any other containers that it depends on.

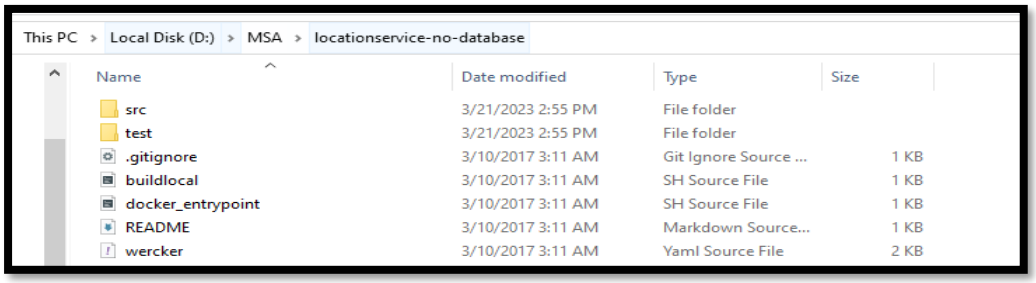
- ❖ Use Team Services to build the Docker image and run the Docker Compose file.
- ❖ Use a testing framework like xUnit to write unit tests that verify that your location service works as expected.
- ❖ Run the unit tests as part of the build process in Team Services.

By using Docker containers to provide a location service as a backing service without a database, you can ensure that your application is portable and can be run in any environment that supports Docker. This can help to reduce the risk of issues arising when your application is deployed to different environments.

**Step 1:** Download the location service folder from the github from the link:-

https://github.com/microservices-aspnetcore/locationservice/tree/no-database

Step 2: Extract the folder.




Step 3: We had previously downloaded the file Teamservice in previous practical number 4. So now we have to open

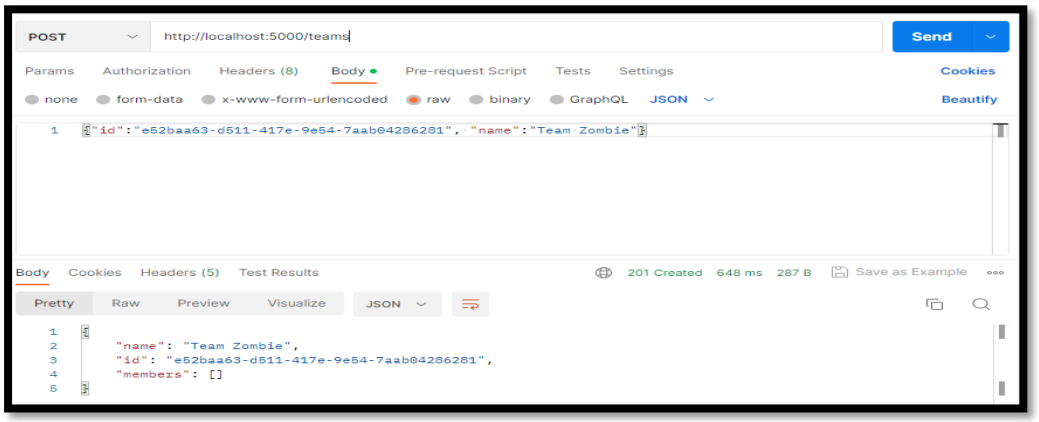
command prompt in our PC & run team service in docker with the command as follows:

**docker run -p 5000:5000 -e PORT=5000 -e LOCATION__URL=http://localhost:5001 dotnetcoreservices/teamservice:location**


Step 4: Now we have to open command prompt in our PC & run location service in docker with the command as follows:

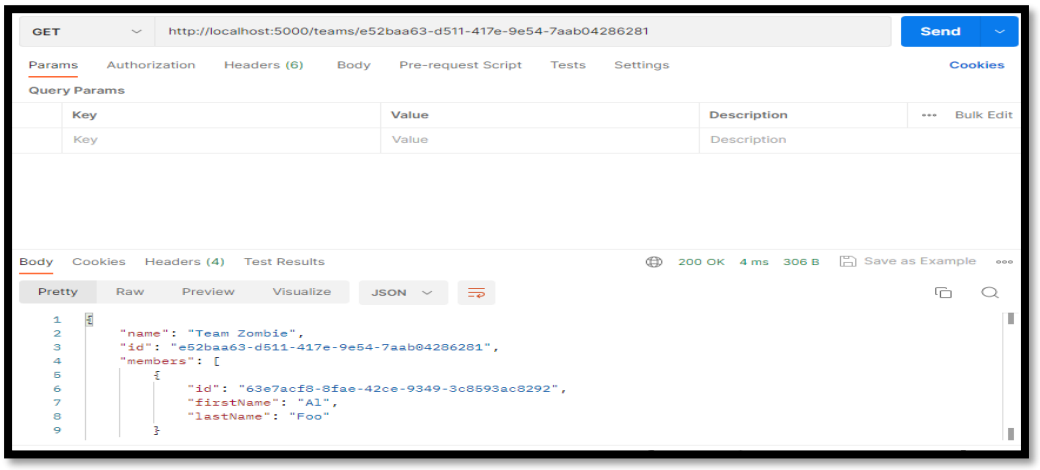
docker run -p 5001:5001 -e PORT=5001 dotnetcoreservices/locationservice:nodb


Step 5: Create a new team using JSON format using REST API.



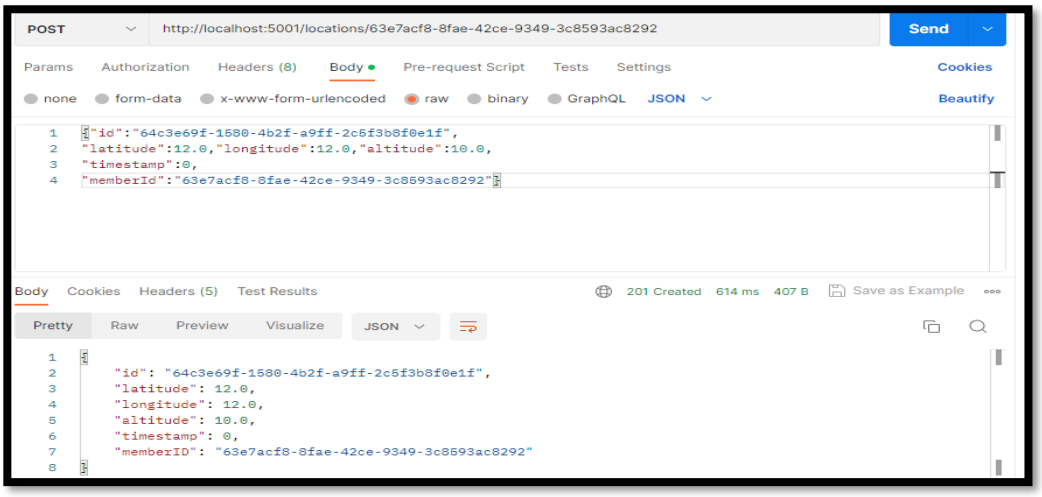
Step 6: Add a new member by posting to the /teams/{id}/members resource:.


Step 7: To confirm that everything has worked so far, query the team details resource:



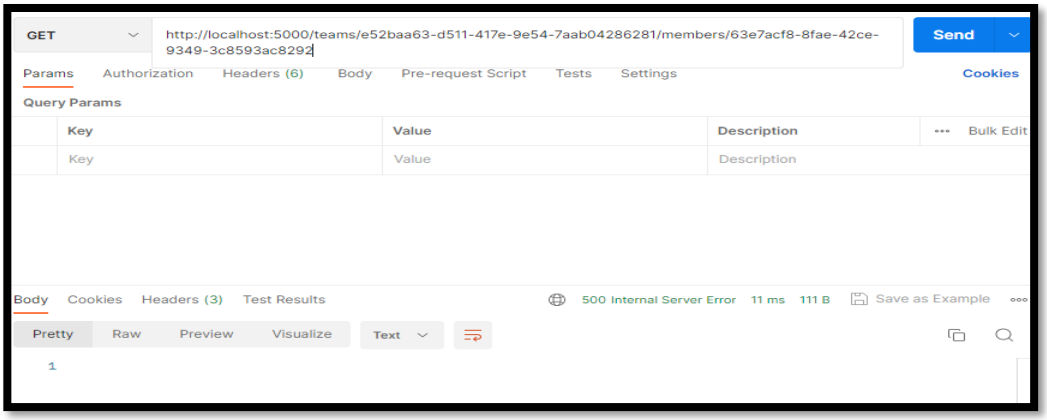

Step 8: Now that the team service has been properly primed with a new team and a new member, we can add a location

to the location service. Note that we could have just added an arbitrary location, but the team service wouldn't be able to find it without at least one team with one member with a location record:




Step 9: Finally, everything is set up to truly test the integration of both the team and the location service.

To do this, we'll query for the member details from the teams/{id}/members/{id} resource:



# Practical N0. 6

## Aim: Building an ASP.NET Simple Core Web Application – Stock Quote example (Use Visual Studio).

**Theory:**

An ASP.NET Core application is a web application that is built using the ASP.NET Core framework.

It allows developers to build web applications that run on Windows, Linux, and macOS.

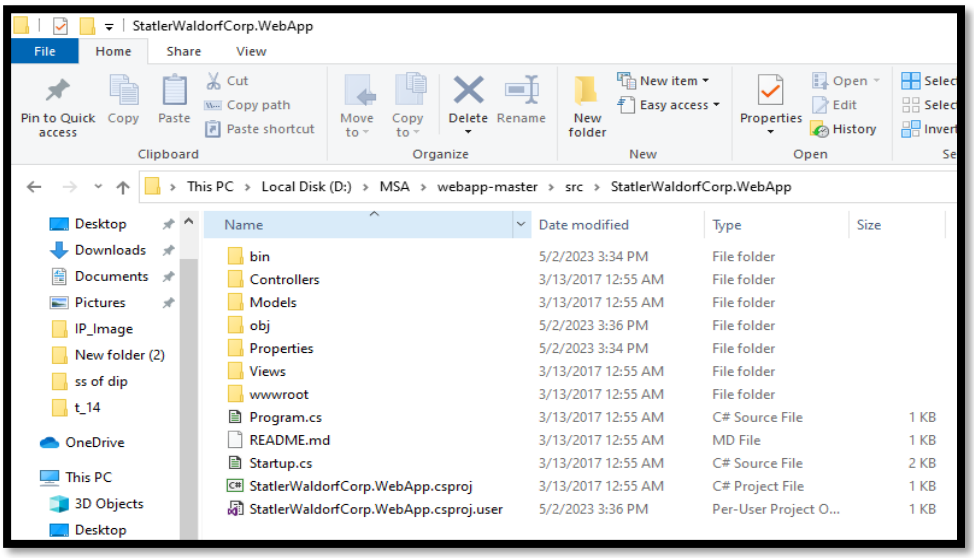
To create a simple ASP.NET Core application that demonstrates a stock quote example, you can follow below steps.

This will help you to learn how to use ASP.NET Core to build web applications that provide data to users.

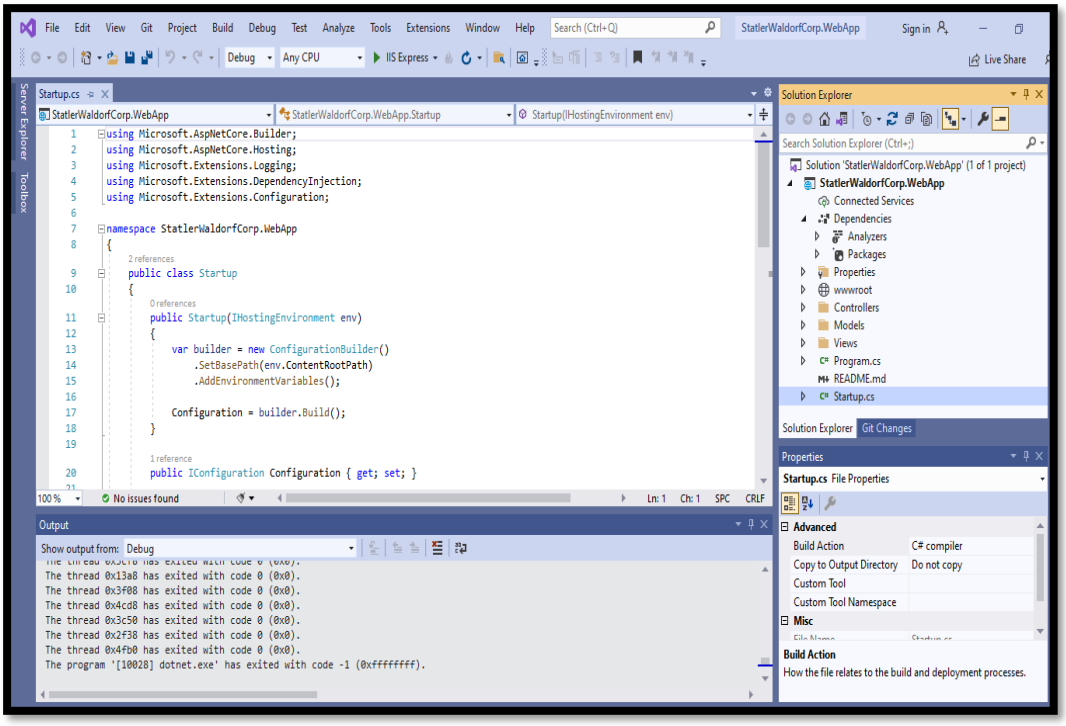
**Steps:** Download the "webapp: Sample ASP.NET Core MVC Application" folder from the github

link: GitHub - microservices-aspnetcore/webapp: Sample ASP.NET Core MVC Application

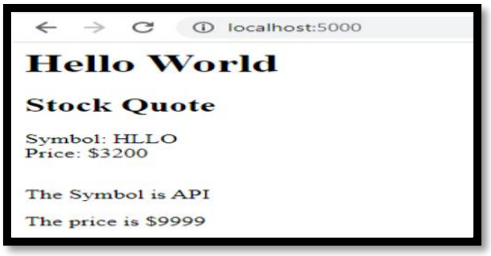
- Extract the "webapp: Sample ASP.NET Core MVC Application" folder in D drive. open the staatlerWaldorfCorp.WebApp.csproj in visual studio.




- we add a line that invokes the UseDeveloperExceptTonPage method to our Startup class, in the Configure method. Here is our new and complete Startup class:



- Run (IIS Express) to start our application. Hitting the home page should combineour controller, our view, and our model to produce a rendered HTML page for the browser,



# Practical N0. 7

**Aim: Build and run CRUD Microservice on Visual Studio to demonstrate Data Service using Database running on IIS Server.**

**Theory:**

A CRUD microservice is a microservice that provides Create, Read, Update, and Delete operations for a particular type of data. It is a common pattern used in web development to manage data in a database.

To create a CRUD microservice on Visual Studio that demonstrates data service using a database running on an IIS server, you can follow these steps:

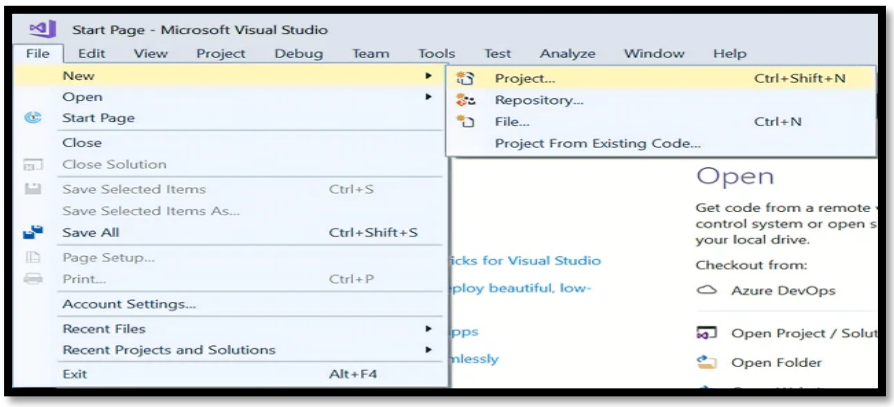
- Open Visual Studio and create a new ASP.NET Core web application.
- Add a new database to your project. For example, you might use SQL Server Express.
- Create a new model that represents the data that you want to store in the database. For example, you might create a Customer model.
- Create a new controller that will handle requests for data. For example, you might create a CustomerController.

- In the CustomerController, create action methods that will handle requests for Create, Read, Update, and Delete operations. For example, you might create a CreateCustomer, GetCustomer, UpdateCustomer, and DeleteCustomer method.
- In the Startup.cs file, configure the services that your application will use. For example, you might configure a service that provides access to your database.
- Test the application to make sure that it works as expected.

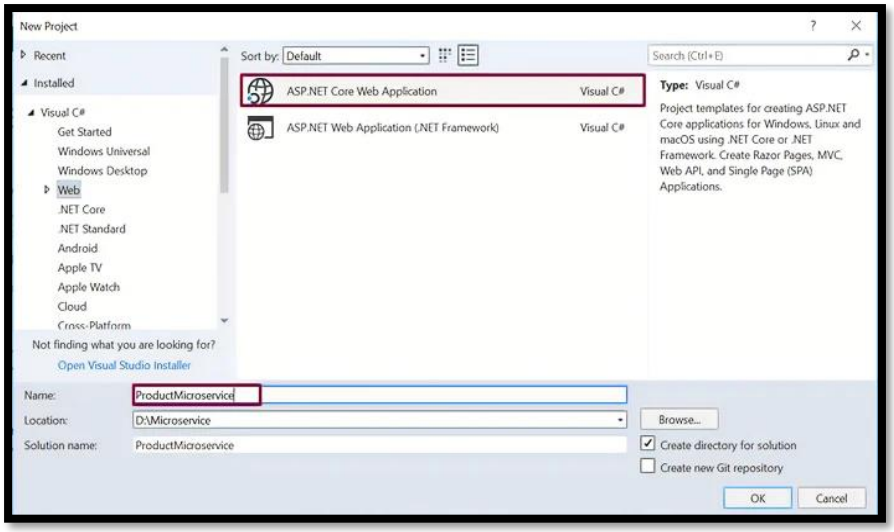
By following these steps, you can create a CRUD microservice that provides data service using a database running on an IIS server. This can help you to learn how to use ASP.NET Core to build web applications that manage data in a database.

**Steps: 1) Creating an ASP.NET Core Application Solution**

Open the Visual Studio and add a new project.

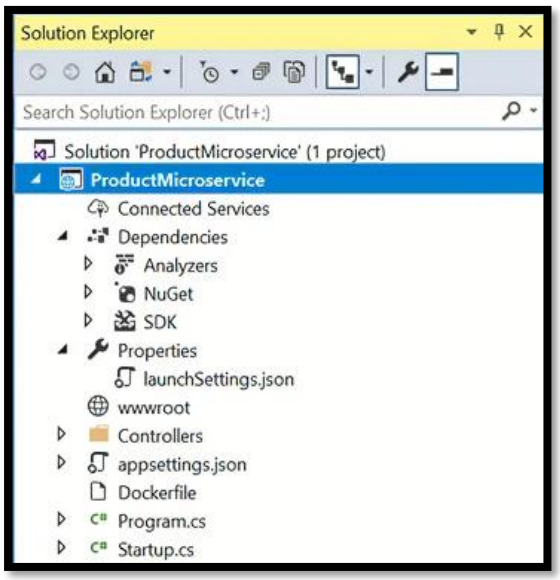


Choose the application as ASP.NET Core Web Application and give it a meaningful name.



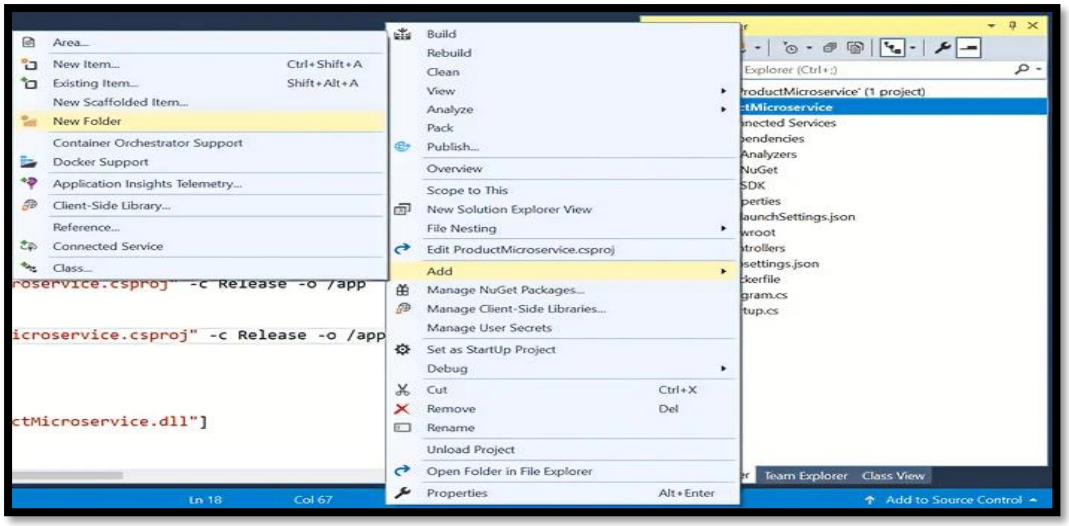

Next, choose API as the type of the project and make sure that “Enable Docker Support” option is selected with OS type as Linux.

The solution will look as shown below.

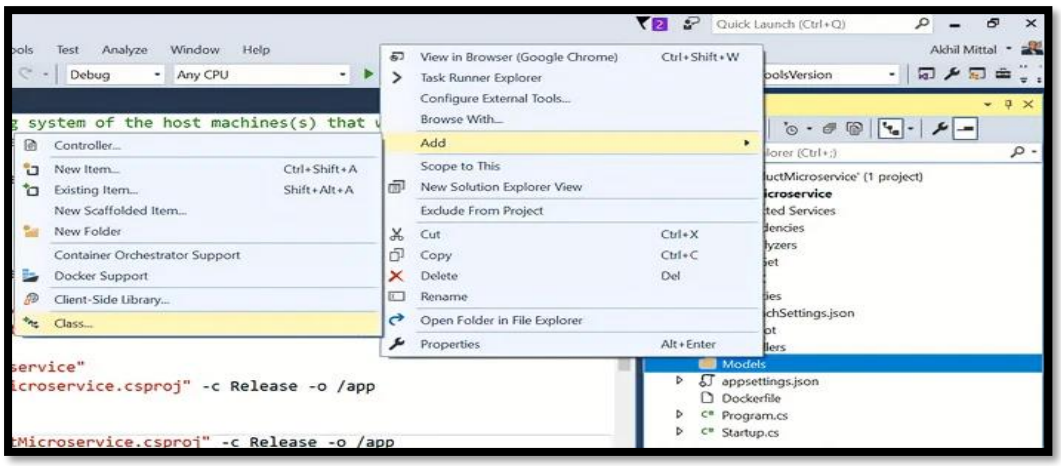


## Adding Models

Add a new folder named “Model” to the project.

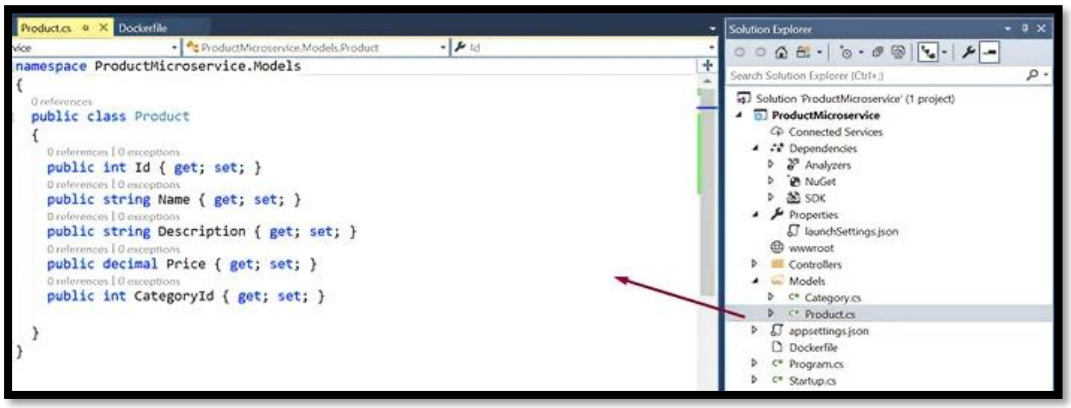



In the Models folder, add a class named Product.

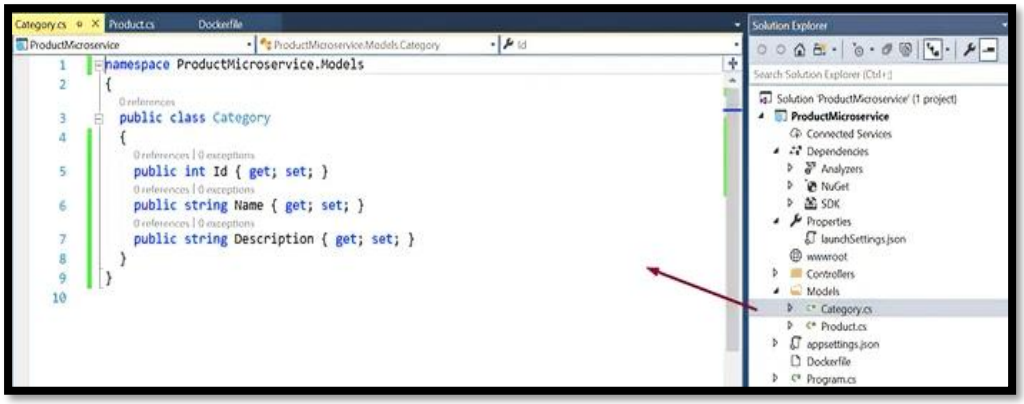



Add a few properties like Id, Name, Description, Price to the product class. The product should also be of some kind and for that, a category model is defined

and a CategoryId property is added to the product model.

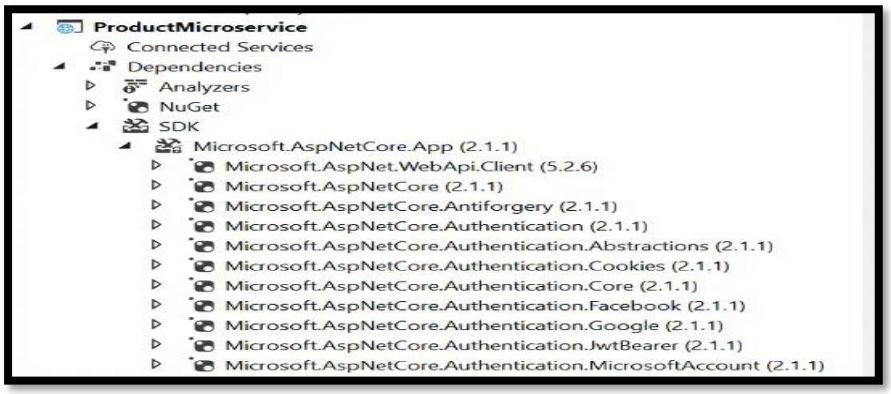



Similarly, add Category model.




## Enabling EF Core

Though .NET Core API project has inbuilt support for EF Core and all the related dependencies are downloaded at the time of project creation and compilation that could be found under SDK section in the project as shown below.

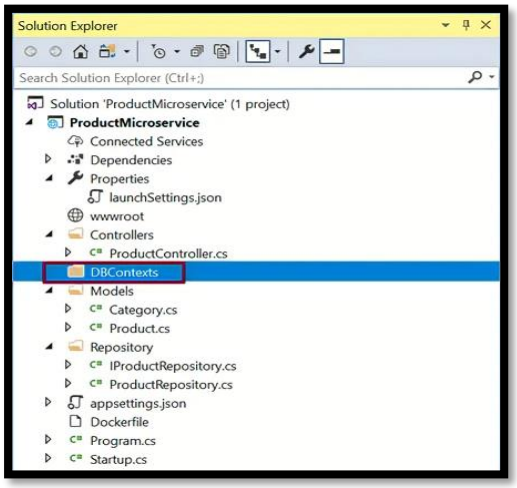



Microsoft.EntityFrameworkCore.SqlServer (2.1.1) should be the package inside the downloaded SDK's. If it is not present, it could be explicitly added to the project via Nuget Packages.

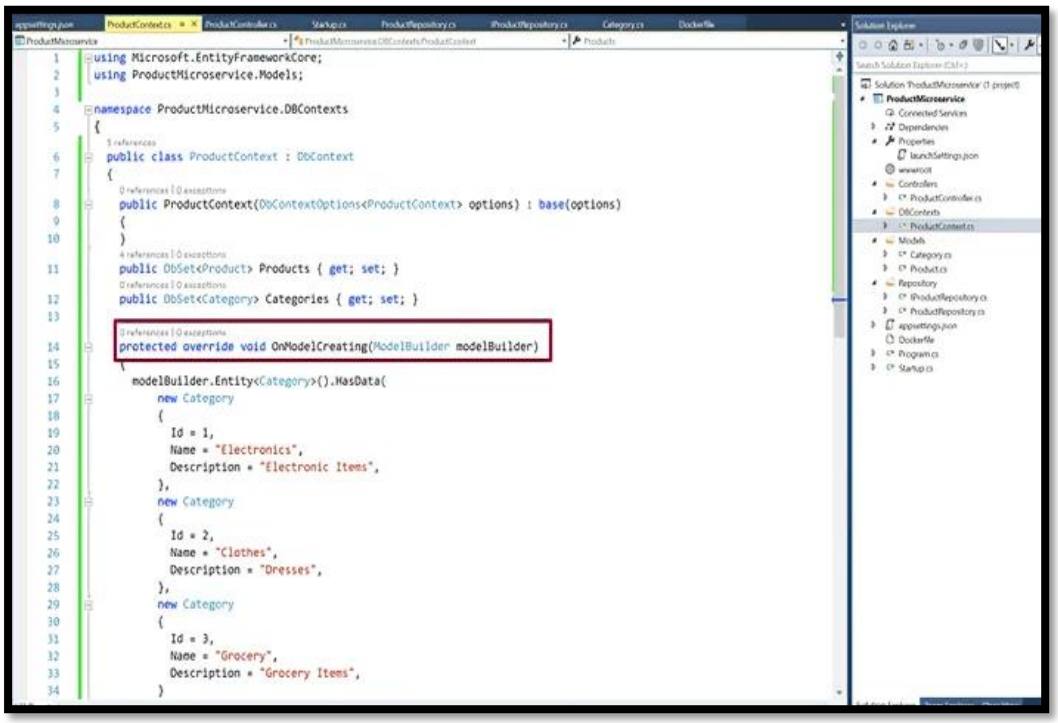
**Adding EF Core DbContext**

A database context is needed so that the models could interact with the database.

Add a new folder named DBContexts to the project.



Add a new class named ProductContext which includes the DbSet properties for Products and Categories. OnModelCreating is a method via which the master data could be seeded to the database. So, add the OnModelCreating method and add some sample categories that will be added to the database initially into the category table when the database is created.



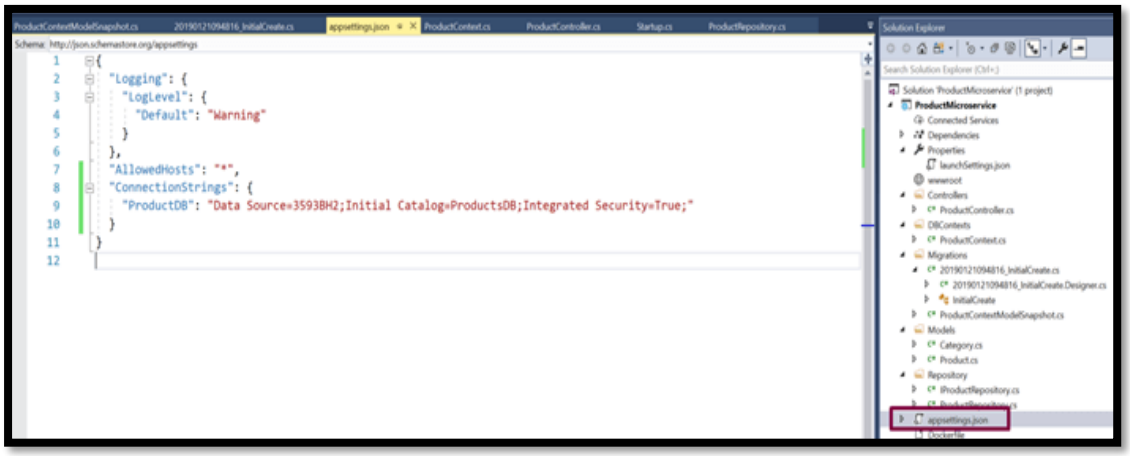
ProductContext code:

```
using Microsoft.EntityFrameworkCore;
using ProductMicroservice.Models;

namespace ProductMicroservice.DBContexts
{
```

```
public class ProductContext : DbContext
{
  public
ProductContext(DbContextOptions<ProductContext> options)
: base(options)
  {
  }
  public DbSet<Product> Products { get; set; }
  public DbSet<Category> Categories { get; set; }
  protected override void OnModelCreating(ModelBuilder
modelBuilder)
  { modelBuilder.Entity<Category>().HasData(
      new Category
      {Id = 1,
       Name = "Electronics",
       Description = "Electronic Items",
      },
      new Category
      {Id = 2,
```

```
            Name = "Clothes",
            Description = "Dresses",
          },
          new Category
          { Id = 3,
            Name = "Grocery",
            Description = "Grocery Items",
          }
        );
      }
    }
  }
```

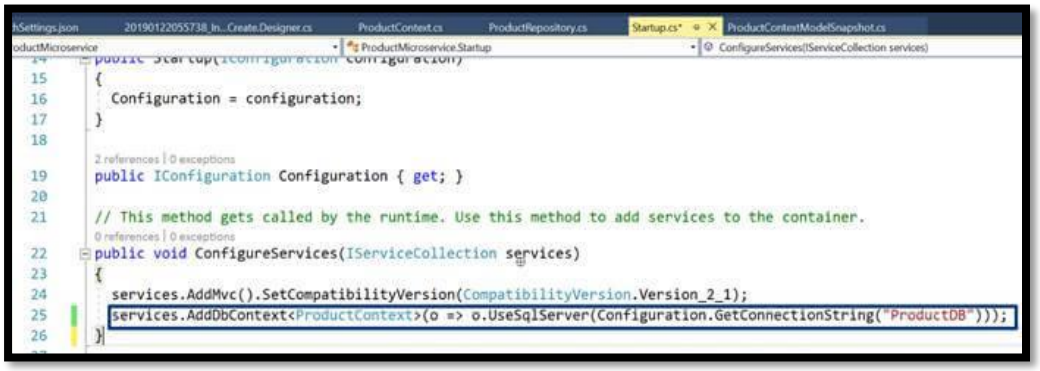
Add a connection string in the appsettings.json file.



Open the Startup.cs file to add the SQL server db provider for EF Core. Add the code

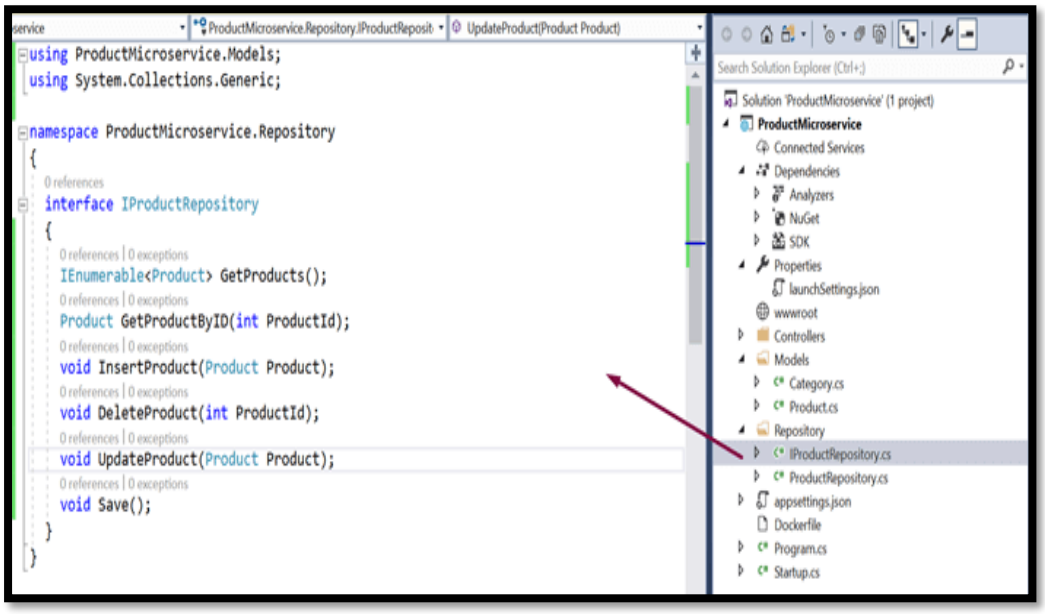
**services.AddDbContext<ProductContext>(o => o.UseSqlServer(Configuration.GetConnectionString("ProductDB")));**

under ConfigureServices method. Note that in the GetConnectionString method the name of the key of the connection string is passed that was added in appsettings file.

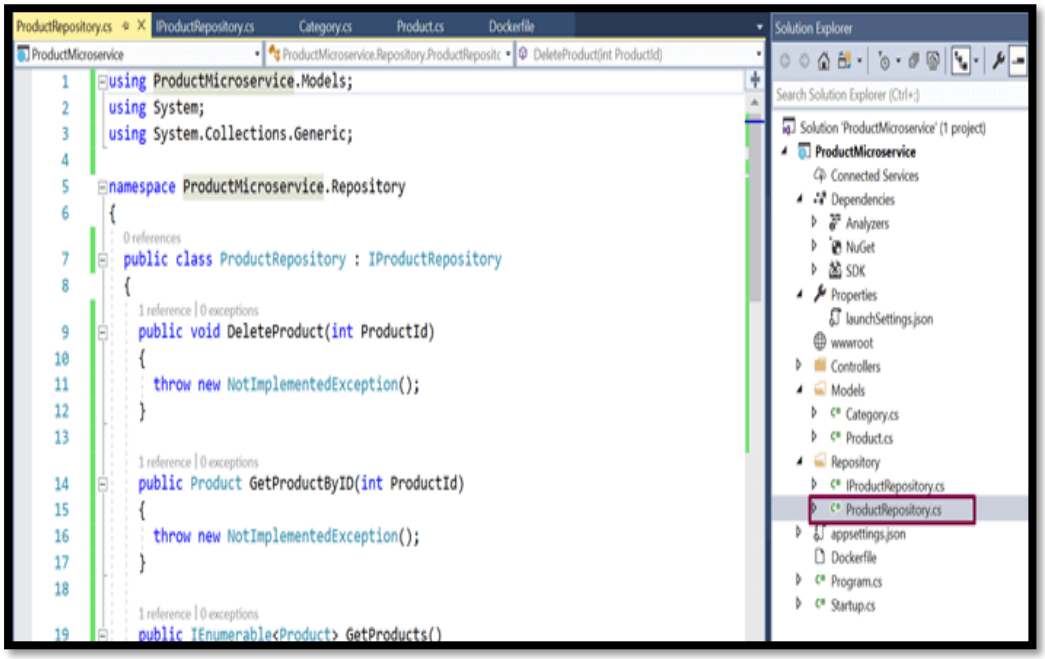



**Adding Repository**

Add a new folder named Repository in the project and add an Interface name IProductRepository in that folder. Add the methods in the interface that performs CRUD operations for Product microservice.




Add a new concrete class named ProductRepository in the same Repository folder that implements IProductRepository. All these methods need:



Add the implementation for the methods via accessing context methods.

**ProductRepository.cs**

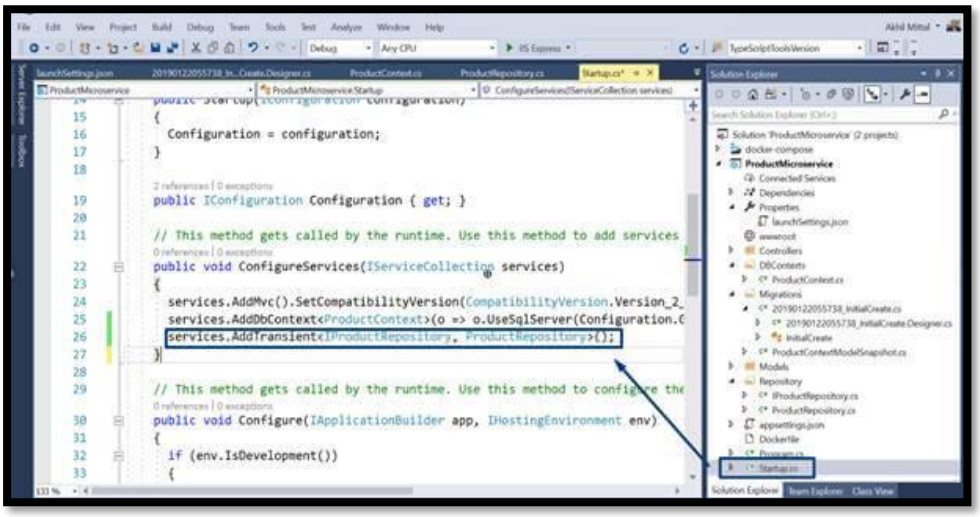
```
using Microsoft.EntityFrameworkCore;
using ProductMicroservice.DBContexts;
using ProductMicroservice.Models;
using System;
using System.Collections.Generic;
using System.Linq;
namespace ProductMicroservice.Repository
{ public class ProductRepository: IProductRepository
 { private readonly ProductContext _dbContext;
        public ProductRepository(ProductContext dbContext)
  {_dbContext = dbContext; }
  public void DeleteProduct(int productId)
  {var product = _dbContext.Products.Find(productId);
   _dbContext.Products.Remove(product);
   Save();  }
  public Product GetProductByID(int productId)
  { return _dbContext.Products.Find(productId); }
```

```
public IEnumerable<Product> GetProducts()
{ return _dbContext.Products.ToList();}
public void InsertProduct(Product product)
{ dbContext.Add(product);
  Save();    }
public void Save()
{_dbContext.SaveChanges();
}  public void UpdateProduct(Product product)
{_dbContext.Entry(product).State = EntityState.Modified;
  Save();
}
}
}
```

Open the Startup class in the project and add the code as

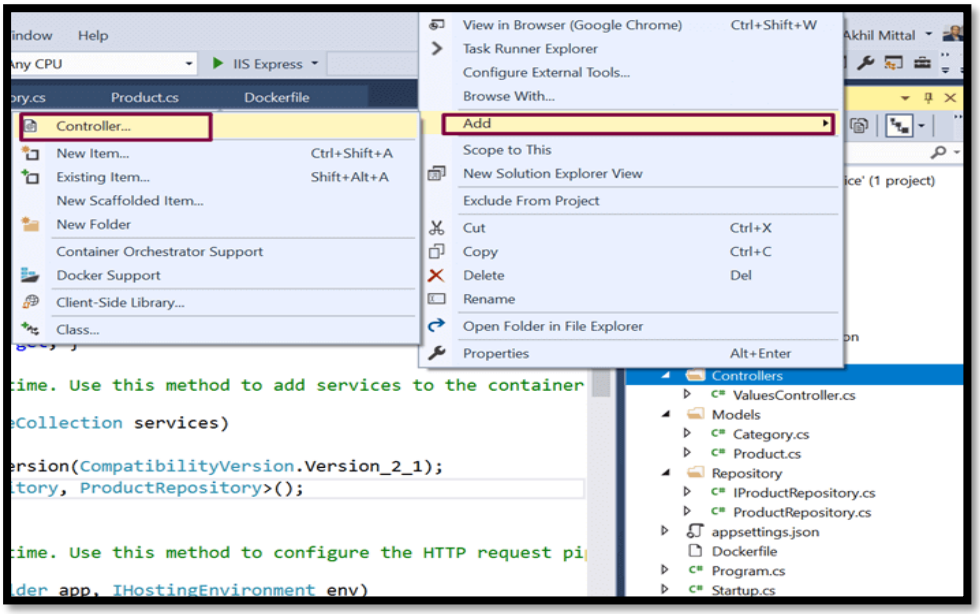
**services.AddTransient<IProductRepository, ProductRepository>();**

inside ConfigureServices method so that the repository's dependency is resolved at a run time when needed.

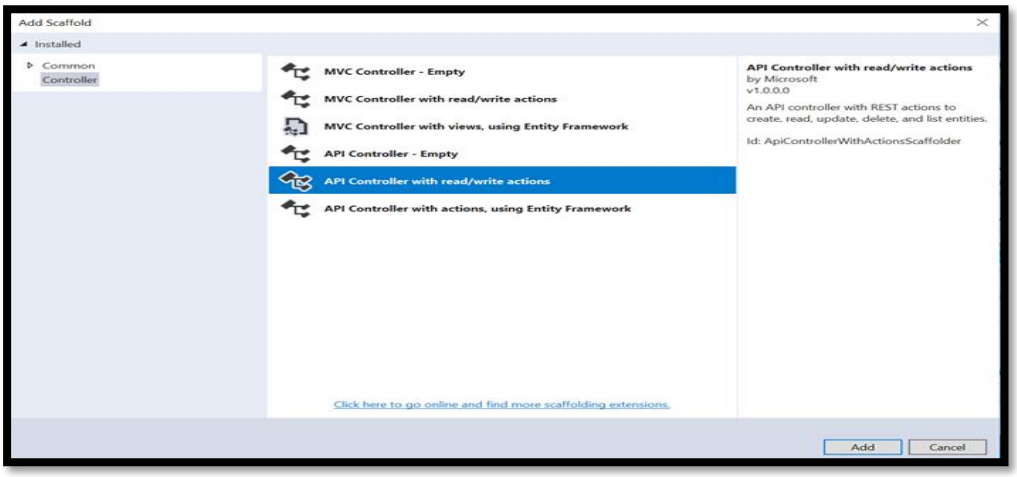


## Adding Controller

Right click on the Controllers folder and add a new Controller as shown below.

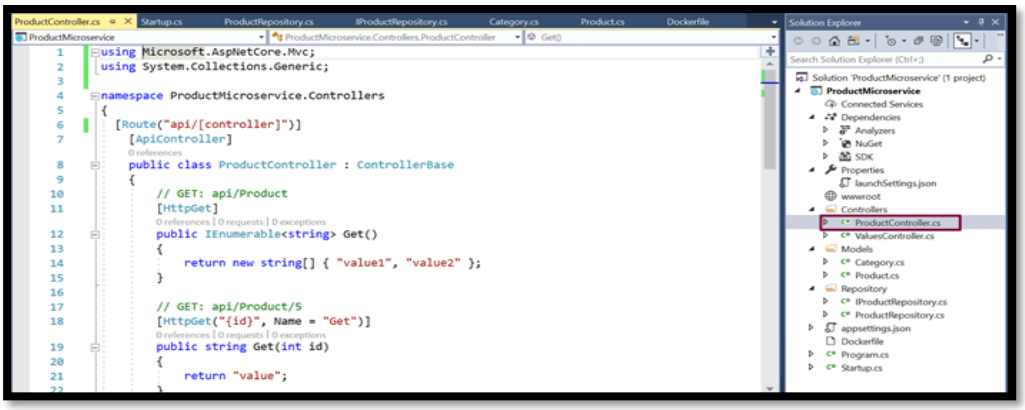



Select the option “API Controller with read/write actions” to add the controller.

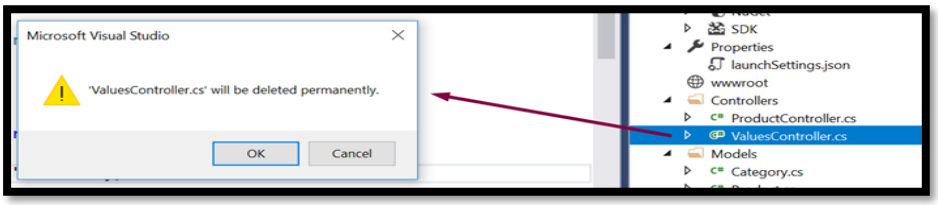



Give the name of the controller as ProductController.


A ProductController class will be added in the Controllers folder with default read/write actions that will be replaced later with product read/write actions and HTTP methods are created acting as an endpoint of the service.




ValuesController can be deleted as it is not needed.




Add implementation to the methods by calling the repository methods as shown below. The basic implementation is shown here for the sake of understanding the concept. The methods could be attribute routed and could be decorated with more annotations as per need.

**ProductController.cs**

```
using Microsoft.AspNetCore.Mvc;
using ProductMicroservice.Models;
using ProductMicroservice.Repository;
using System;
using System.Collections.Generic;
using System.Transactions;
namespace ProductMicroservice.Controllers
{ [Route("api/[controller]")]
 [ApiController]
 public class ProductController : ControllerBase
 { private readonly IProductRepository _productRepository;
```

```
    public ProductController(IProductRepository
productRepository)
    {  _productRepository = productRepository;
    }
    [HttpGet]
    public IActionResult Get()
    { var products = _productRepository.GetProducts();
      return new OkObjectResult(products);
    }
    [HttpGet("{id}", Name = "Get")]
    public IActionResult Get(int id)
    { var product = _productRepository.GetProductByID(id);
      return new OkObjectResult(product);
    }
    [HttpPost]
    public IActionResult Post([FromBody] Product product)
    { using (var scope = new TransactionScope())
      { _productRepository.InsertProduct(product);
        scope.Complete();
```

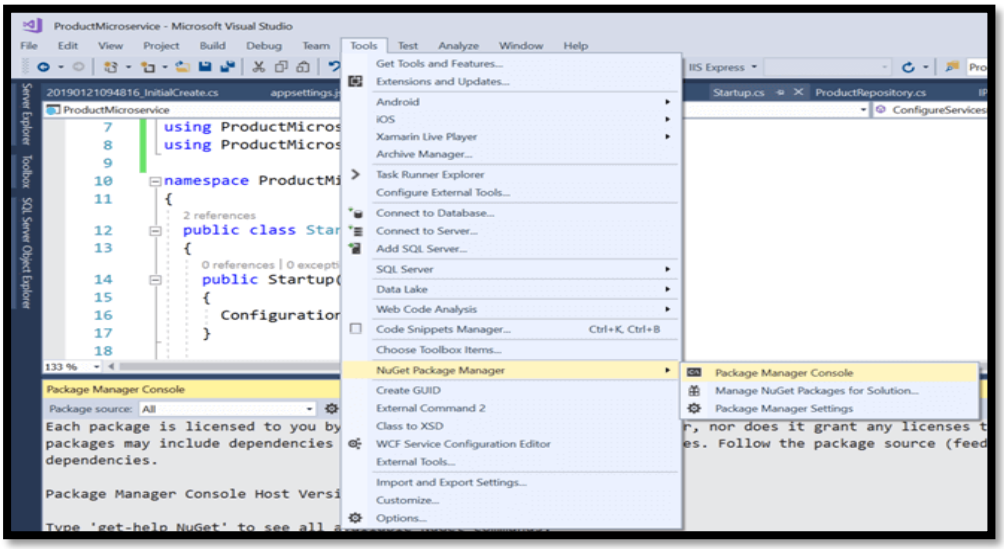
```
        return CreatedAtAction(nameof(Get), new { id =
product.Id }, product);
    }   }
   [HttpPut]
   public IActionResult Put([FromBody] Product product)
   {  if (product != null)
    {  using (var scope = new TransactionScope())
     {_productRepository.UpdateProduct(product);
       scope.Complete();
       return new OkResult();
     }     }
    return new NoContentResult();
   }
   [HttpDelete("{id}")]
   public IActionResult Delete(int id)
   {
    _productRepository.DeleteProduct(id);
    return new OkResult();
   }
```

```
        }
    }
```

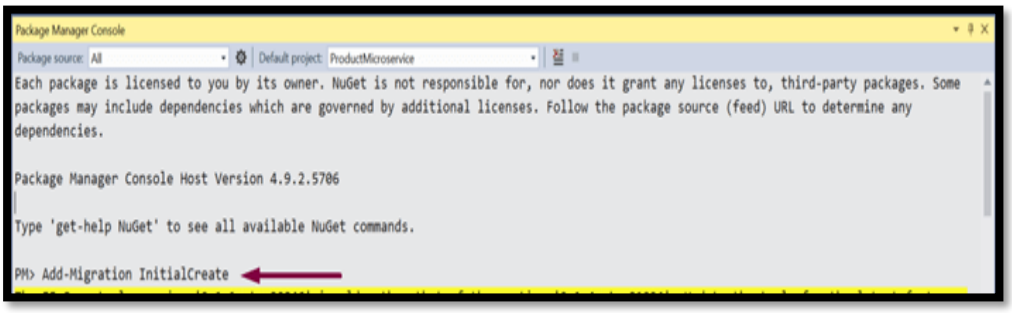
**Entity Framework Core Migrations**

Migrations allow us to provide code to change the database from one version to another.

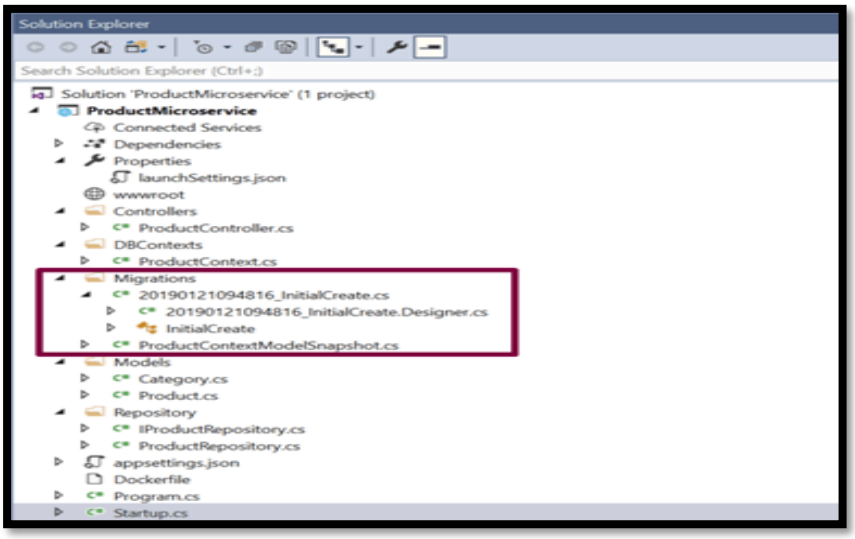
Open Package Manager Console.




To enable the migration, type the command, Add-Migration and give that a meaningful name for e.g. InitialCreate and press enter.



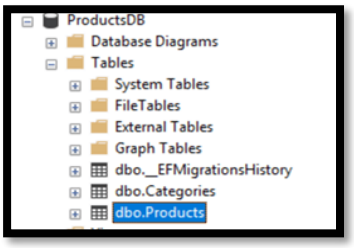
Once the command is executed, if we look at our solution now, we see there's a new Migrations folder. And it contains two files. One, a snapshot of our current context model. Feel free to check the files. The files are very much self-explanatory.



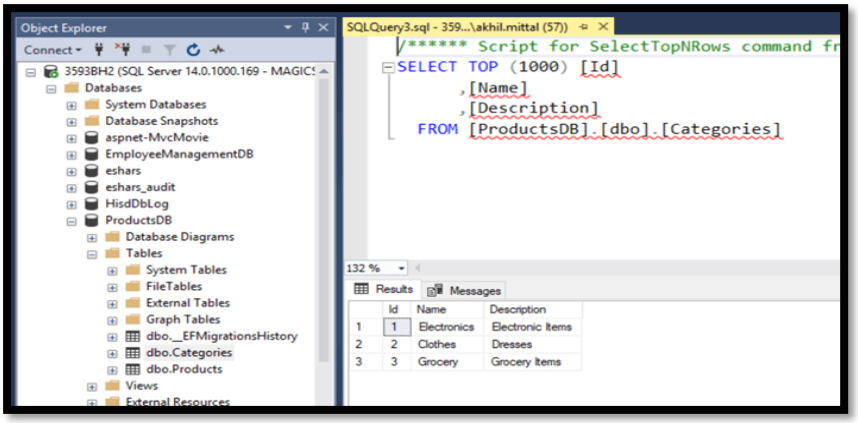

To ensure that migrations are applied to the database there's another command for that. It's called the

**update-database** If executed, the migrations will be applied to the current database.


Check the SQL Server Management Studio to verify if the database got created.

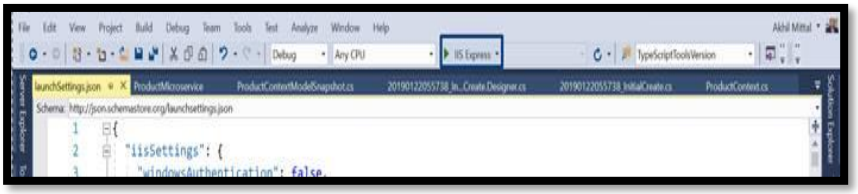



When data of the Categories table is viewed the default master data of three categories is shown.



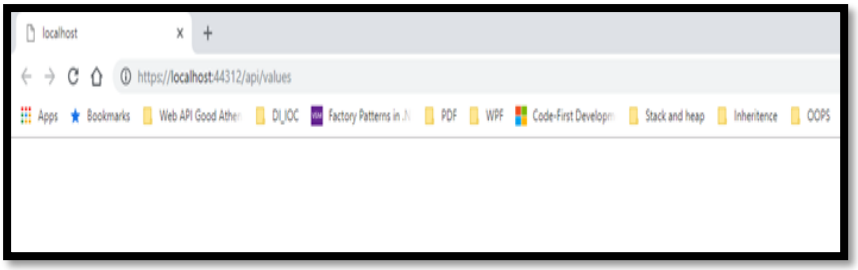

**Run the Product Microservice**

Choose IIS Express in the Visual Studio as shown below and press F5 or click that IIS Express button itself.

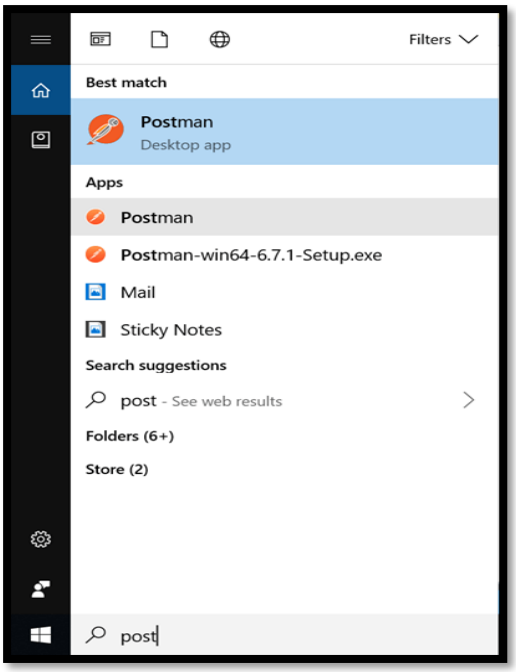



The application will be up once the browser page is launched. Since it has nothing to show, it will be blank, but the service could be tested via any API

testing client. Here Postman is used to testing the service endpoints. Keep it opened and application running.




Install Postman if it is not on the machine and launch it.



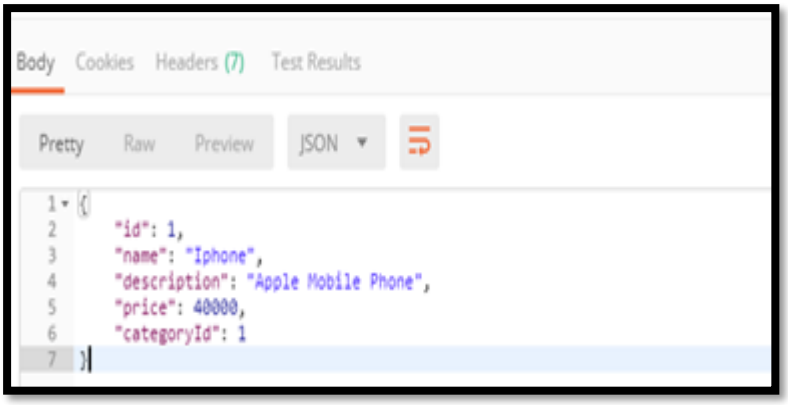

**POST**

To test the POST method; i.e. create a new resource, select the method as POST in postman and provide the endpoint,

i.e. https://localhost:44312/api/product and in the Body section, add a JSON

similar to having properties of Product model as shown below and click on Send.

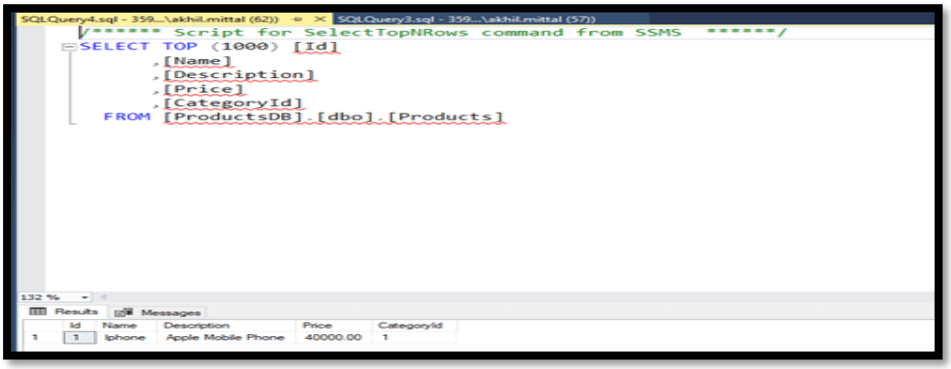

The response is returned with the Id of the product as well.



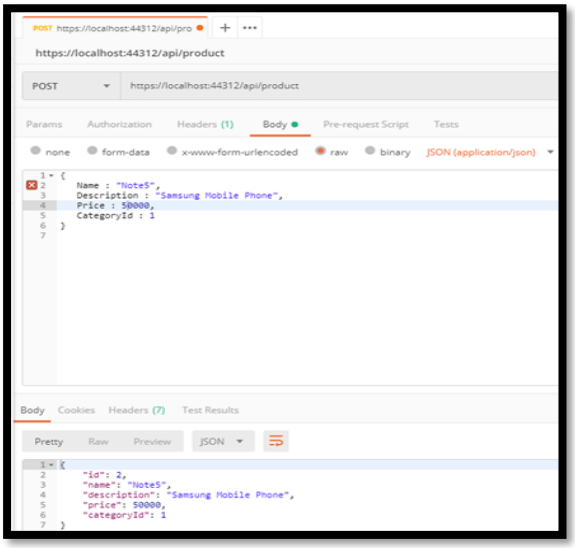

The “Post” method of the controller is responsible to create a resource in the database and send the response.

The line return CreatedAtAction(nameof(Get), new { id=product.Id }, product); returns the location of the created resource that could be checked in Location attribute in the response under Headers tab.


Perform a select query on the product table and an added row is shown for the newly created product.

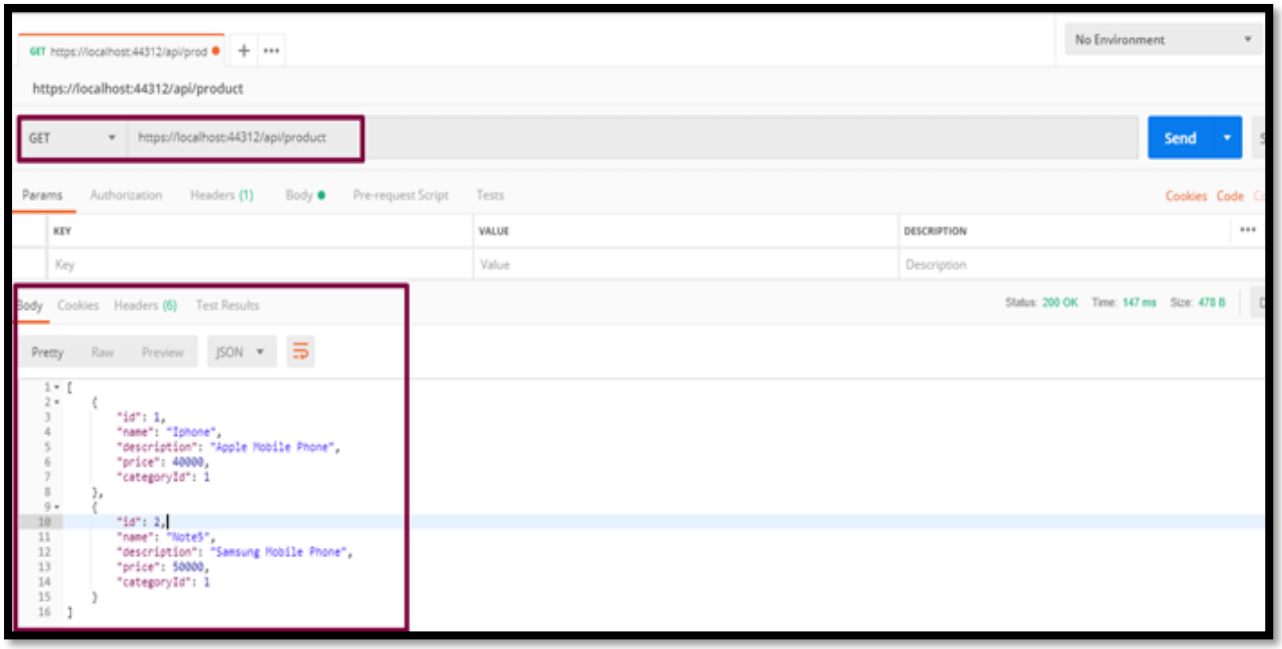



Create one more product in a similar way.



## GET

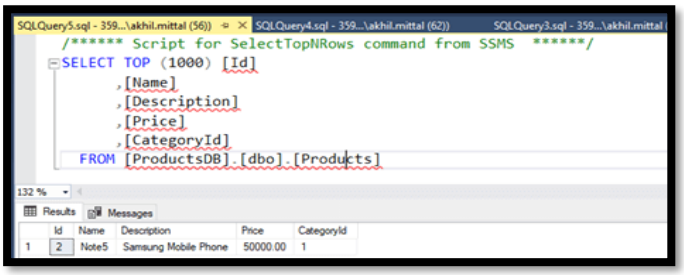
Perform a GET request now with the same address and two records are shown as a JSON result response.




## DELETE

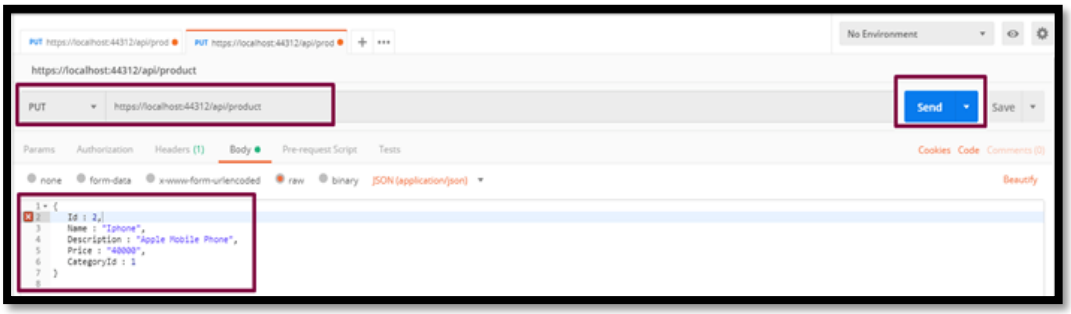
Perform the delete request by selecting DELETE as the verb and appending id as 1 (if the product with id 1 needs to be deleted) and press Send.


In the database, one record with Id 1 gets deleted.

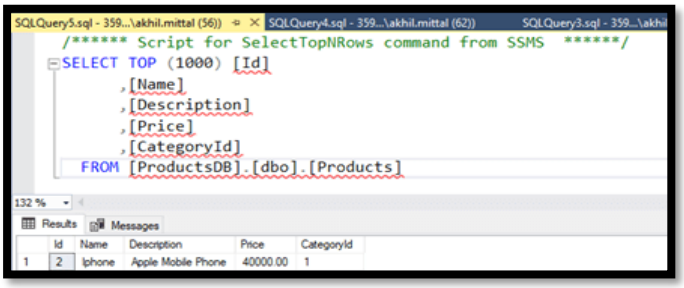



**PUT**

PUT verb is responsible for updating the resource. Select PUT verb, provide the API address and in the Body section, provide details of which product needs to be updated in JSON format. For example, update the product with Id 2 and update its name, description, and price from Samsung to iPhone specific. Press Send.




Check the database to see the updated product.



# Practical N0. 8

## Aim: Build and Run CRUD ASP.NET Core Using Entity Framework Core with Swagger.

**Theory:**

CRUD (Create, Read, Update, Delete) is a common pattern used in web development to manage data in a database.

Entity Framework Core is an object-relational mapper (ORM) that enables .NET developers to work with a database using .NET objects.

Swagger is an open-source tool that helps developers design, build, document, and consume RESTful web services.

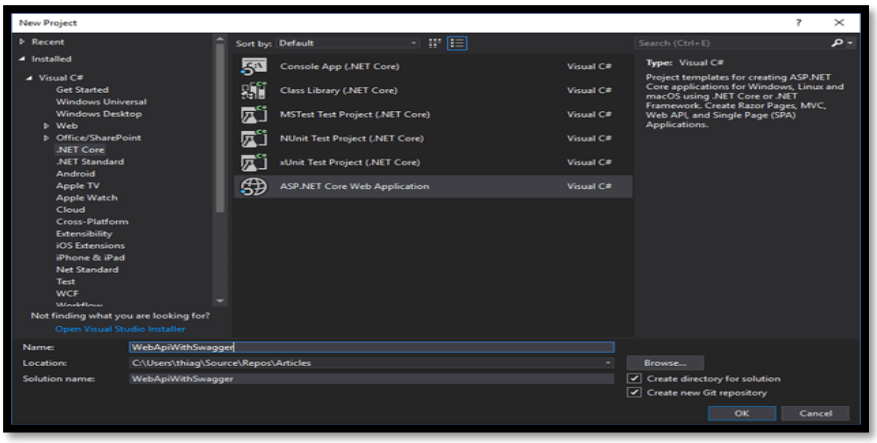
By following below steps, you can create a CRUD ASP.NET Core application using Entity Framework Core with Swagger.

This will help you to learn how to use ASP.NET Core to build web applications that manage data in a database and document your RESTful web services using Swagger.

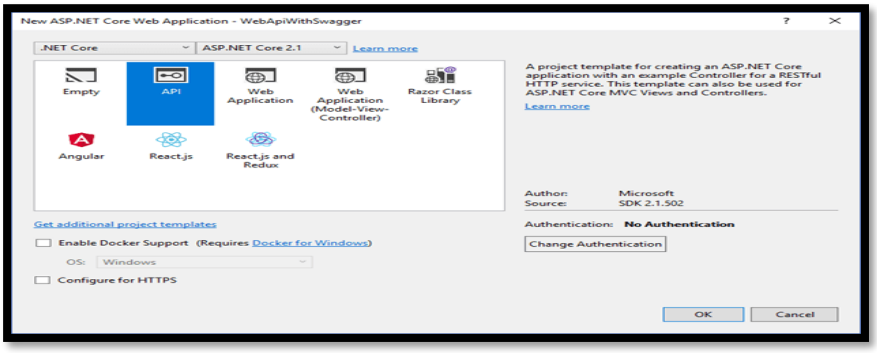
I hope you have understood about the Swagger , Entity Framework Core and CRUD (Create, Read, Update, Delete),

Now let’s implement it practically.

**Steps: 1)** Create a new project of type ASP.NET Core Web Application.

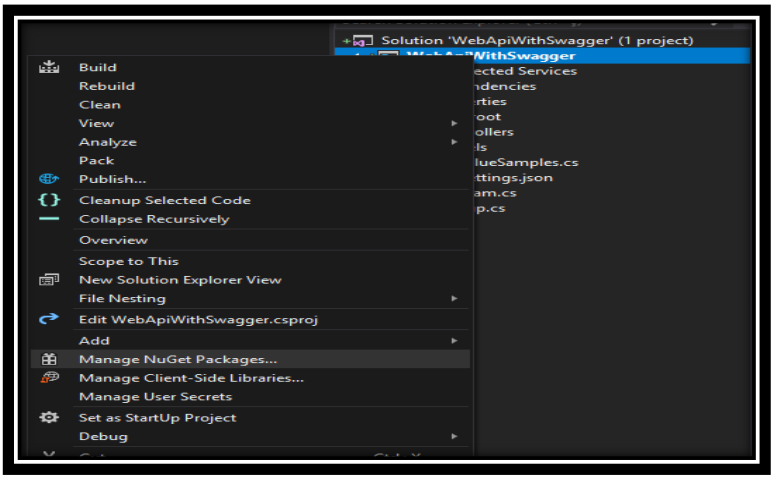



Select the project subcategory as API.

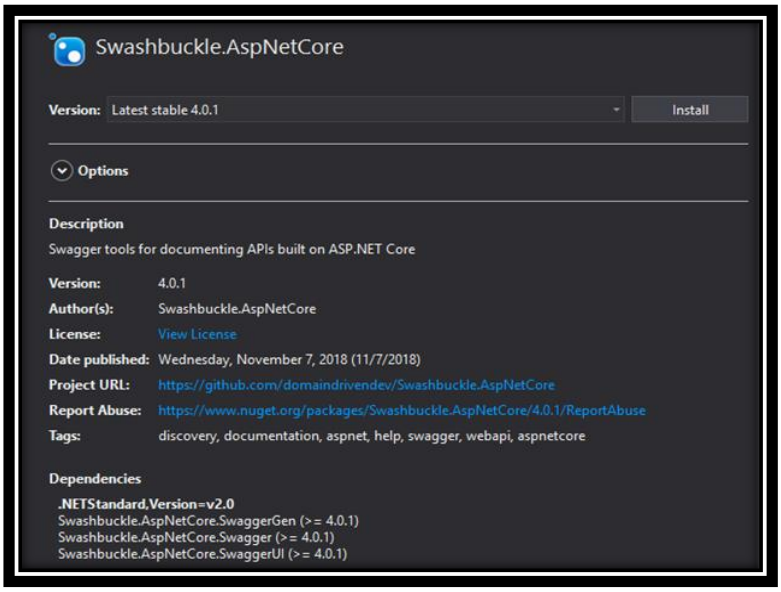



This is the result of your project creation.


Double-click on your project and click on "Manage NuGet Packages...".




Install the Swashbucle.AspNetCore NuGet,




Update your StartUp class in order for your project to recognize Swagger.

```
public class Startup
  {  public Startup( IConfiguration configuration )
    {  Configuration = configuration;
    }
```

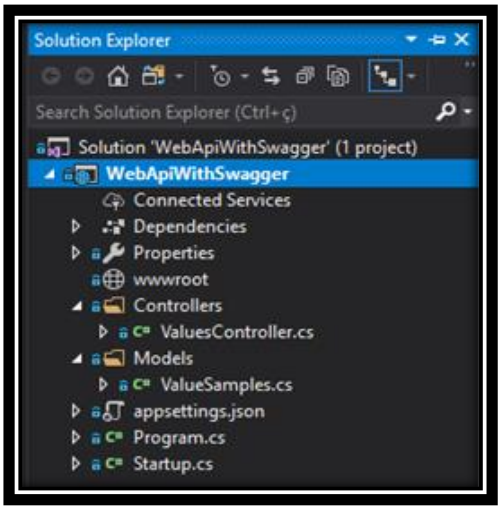
```
        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
      public void ConfigureServices( IServiceCollection services )
      { services.AddMvc().SetCompatibilityVersion( CompatibilityVersion.Version_2_1 );
            // Register Swagger
          services.AddSwaggerGen( c =>
          {
              c.SwaggerDoc( "v1", new Info { Title = "Sample API", Version = "version 1" } );
          } );
      }
      // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
      public void Configure( IApplicationBuilder app, IHostingEnvironment env )
      {
          if ( env.IsDevelopment() )
```

```
            {
                app.UseDeveloperExceptionPage();
                 // Enable middleware to serve generated Swagger as a JSO
N endpoint.
                app.UseSwagger();
                 // Enable middleware to serve swagger-
ui (HTML, JS, CSS, etc.),
                // specifying the Swagger JSON endpoint.
                app.UseSwaggerUI( c =>
                {   c.SwaggerEndpoint( "/swagger/v1/swagger.json", "My A
PI V1" );
                } );
            }
              app.UseMvc();
        }    }
```

**Project customization:**

In order to use the Swagger API, let's create some scenarios that could take advantage of the Swagger usage.

The project with the customization will be like below.




**ValueSamples class:**

```
public static class ValueSamples
 {  public static Dictionary<int, string> MyValue;
    public static void Initialize()
    {  MyValue = new Dictionary<int, string>();
       MyValue.Add( 0, "Value 0" );
       MyValue.Add( 1, "Value 1" );
       MyValue.Add( 2, "Value 2" );
    }
 }
```

**ValuesController controller:**

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

```
public class ValuesController : ControllerBase
{
    public ValuesController()
    {
ValueSamples.Initialize();
    }
    // GET api/values
    [HttpGet]
    public ActionResult<Dictionary<int, string>> Get()
    {
return ValueSamples.MyValue;
    }
    // GET api/values/5
    [HttpGet( "{id}" )]
    public ActionResult<string> Get( int id )
    {
return ValueSamples.MyValue.GetValueOrDefault( id );
    }
    // POST api/values
```

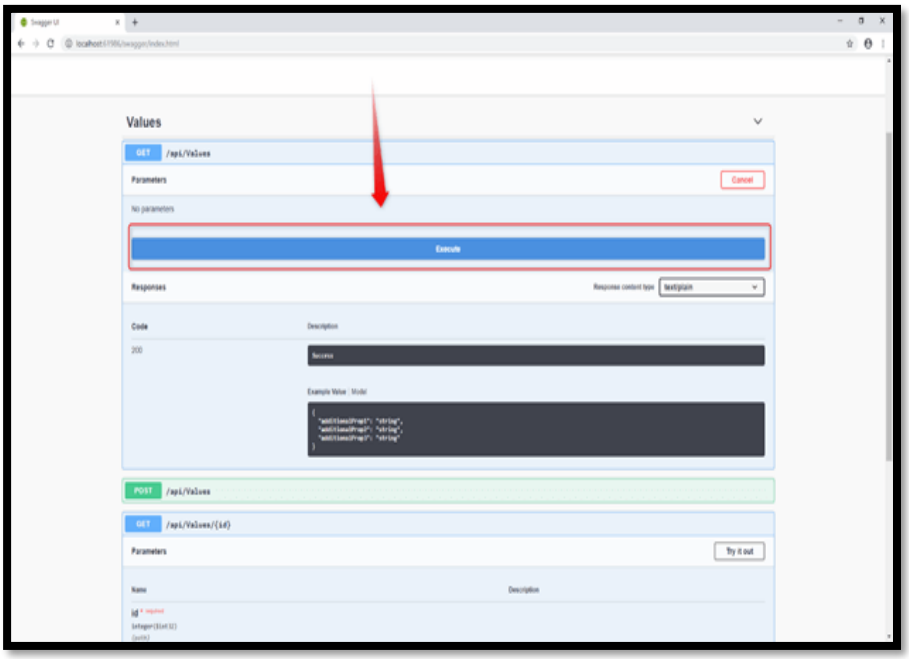
```
        [HttpPost]
        public void Post( [FromBody] string value )
        {
    var maxKey = ValueSamples.MyValue.Max( x => x.Key );


           ValueSamples.MyValue.Add( maxKey + 1, value );
        }
        // PUT api/values/5
        [HttpPut( "{id}" )]
        public void Put( int id, [FromBody] string value )
        {
   ValueSamples.MyValue.Add( id, value );
        }
        // DELETE api/values/5
        [HttpDelete( "{id}" )]
        public void Delete( int id )
        { ValueSamples.MyValue.Remove( id );
        }
     }
```

**Using the Swagger:**

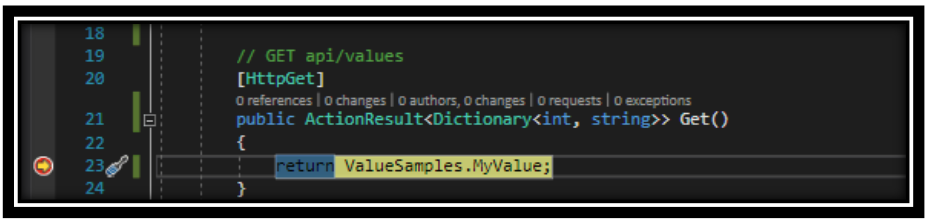
Now, push F5 and complete your URL with "/swagger". It must look like this.http://localhost:61986/swagger


Let's start testing our Web APIs. **Testing the "Get all" method**

Calling the method from Swagger,

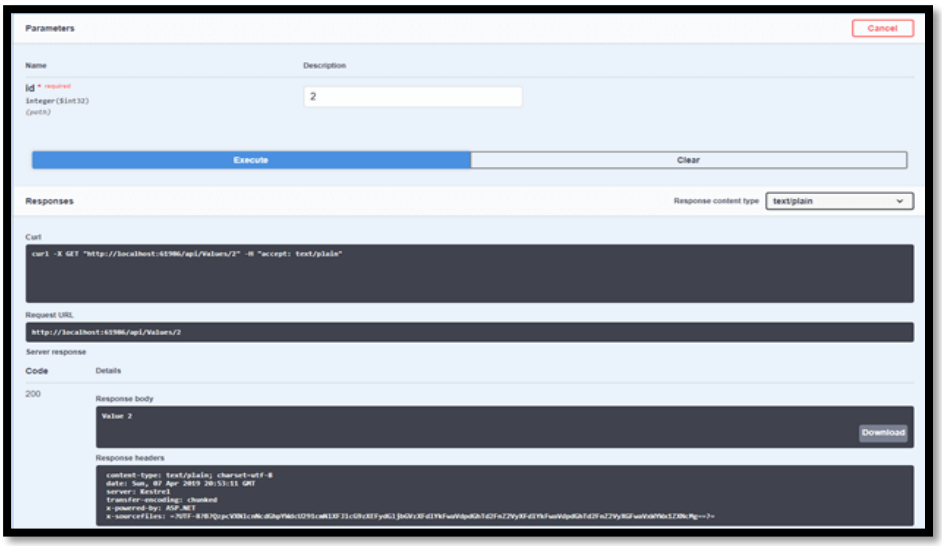


Validating the method called from the controller.

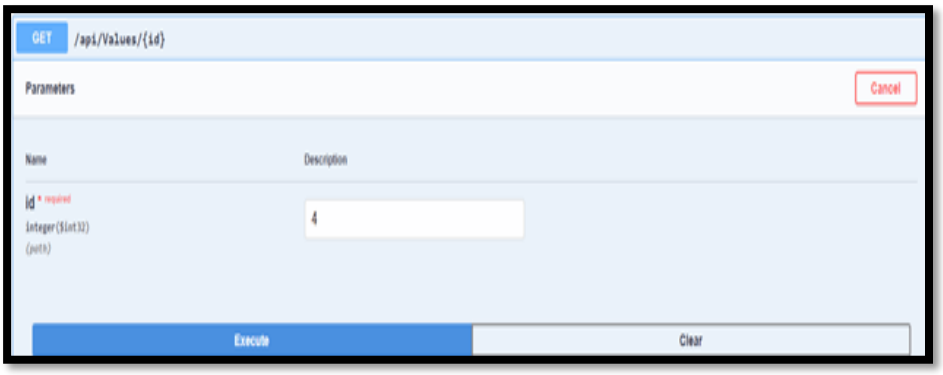



Checking the API response.


**Testing getting a single result method:**Inputting data in Swagger,



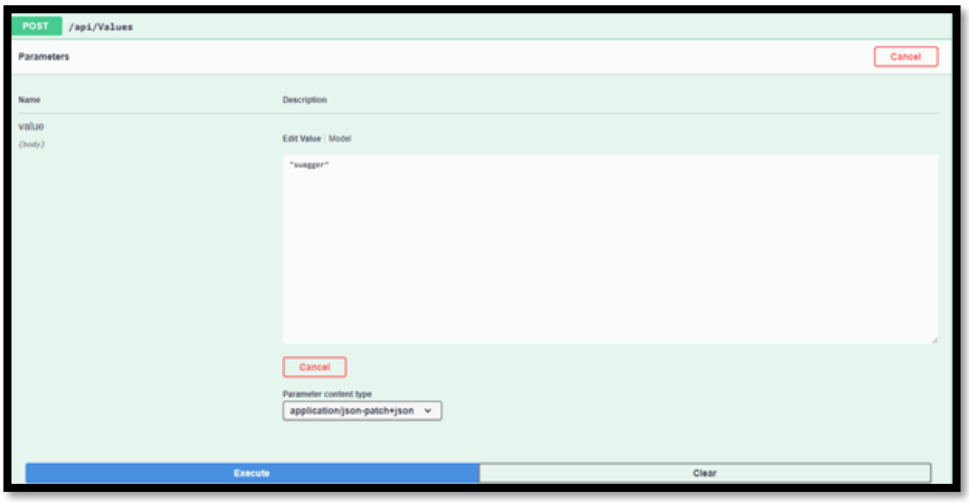
**Testing to get a single non-existent record:** Calling the method from,




Validating the call from the controller,


**Swagger Response**


**Testing the post method**: Input the data in the Post Method,



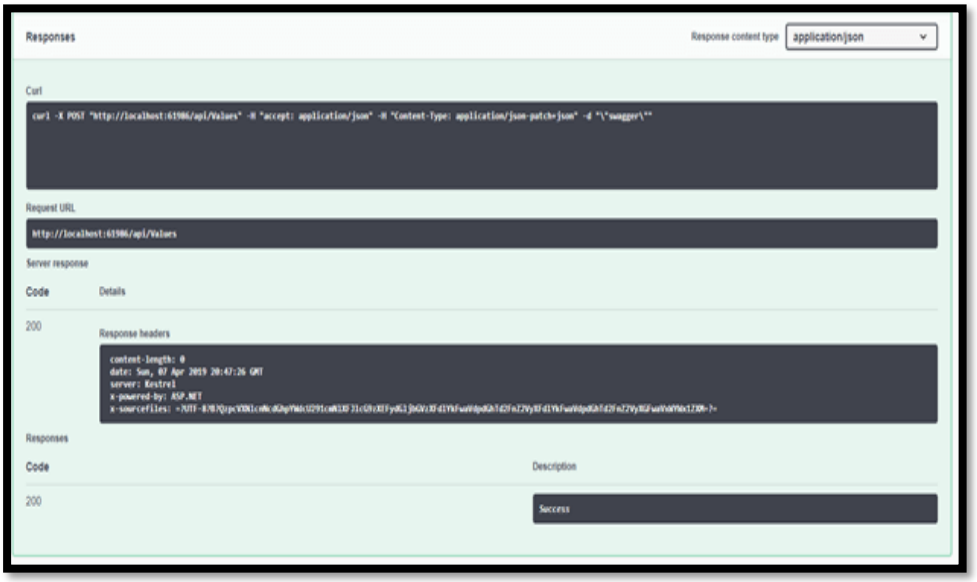

Validating the received data in the controller

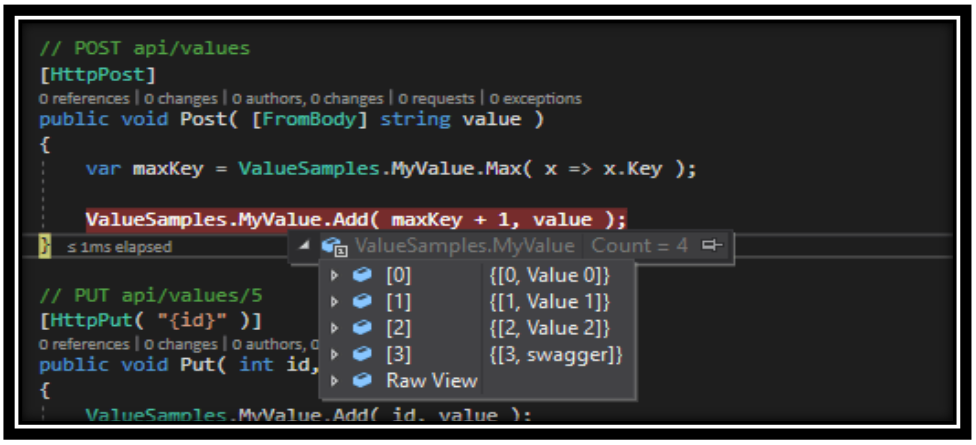
```
// POST api/values
[HttpPost]
0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
public void Post( [FromBody] string value )
{
    var maxKey = ValueSamples.MyValue.Max( x => x.Key );
    ValueSamples.MyValue.Add( maxKey + 1, value );
}
```

value "swagger"

≤ 143,451ms elapsed

## Swagger Response






Congratulations, you have successfully integrated Swagger with your Rest API,

**CRUD Implementation:**

Adjust the model in order to make it able to be stored into a database,

```
public class ValueSamples
{  [Key]
   public int Id { get; set; }
   public string Name { get; set; }
}
```

**Create your DB context file:**

This is the class that is going to connect to your database, each object declared inside DbSet type is going

to be a table.

```
public class CrudSampleContext : DbContext
  { public CrudSampleContext(DbContextOptions<CrudSampleCon
text> options) : base(options)
      {
      }
        public DbSet<ValueSamples> ValueSamples { get; set; }
  }
```

**Adjust your StartUp class:**

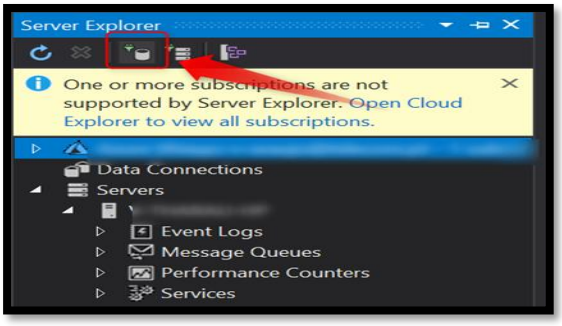
You must register and initialize your DB context using dependency injection.

```
public void ConfigureServices( IServiceCollection services )
      {  services.AddMvc().SetCompatibilityVersion( CompatibilityVe
rsion.Version_2_1 );
services.AddDbContext<CrudSampleContext>(options => options.Us
eSqlServer("your database connection string"));
          // Register Swagger
         services.AddSwaggerGen( c =>
         {  c.SwaggerDoc( "v1", new Info { Title = "Sample API", Vers
ion = "version 1" } );
         } );
```

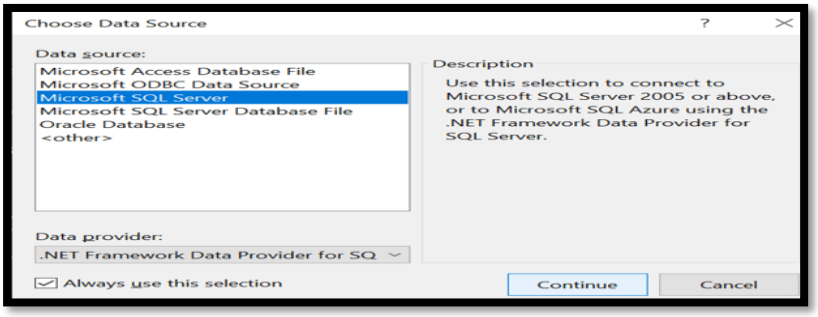
}

**Getting your database connection string:**

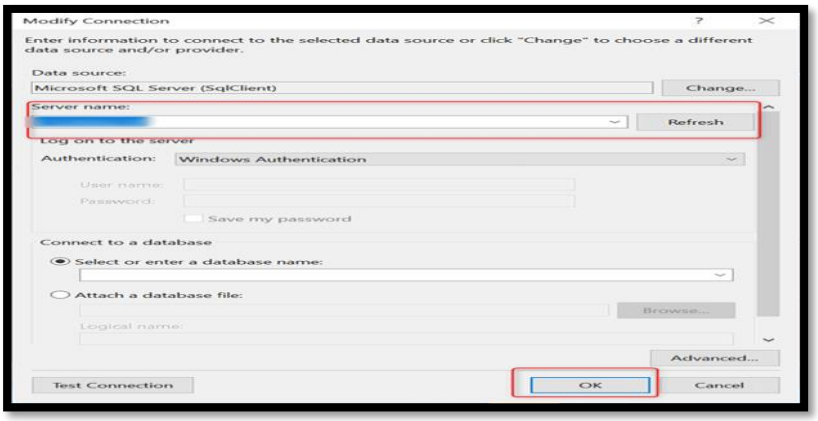
Open Server Explorer and click on "Connect to Database".



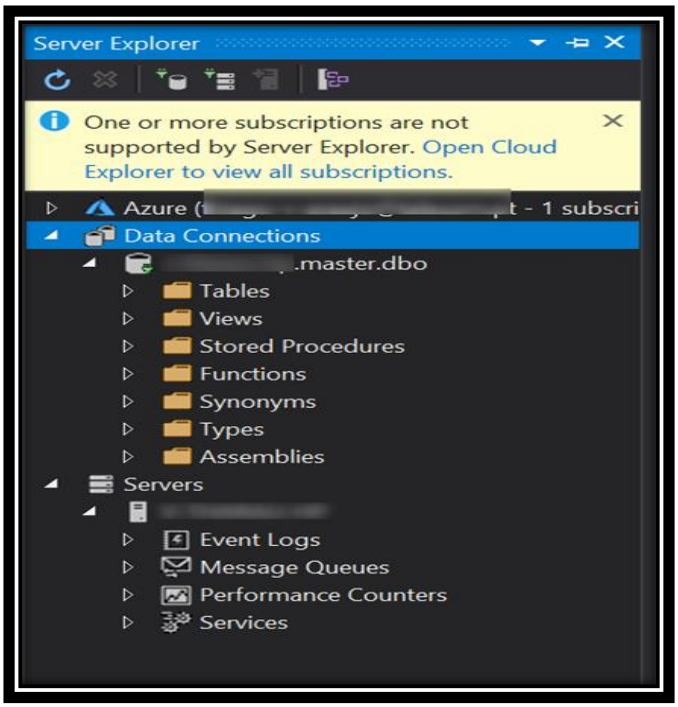

Select Microsoft SQL Server.



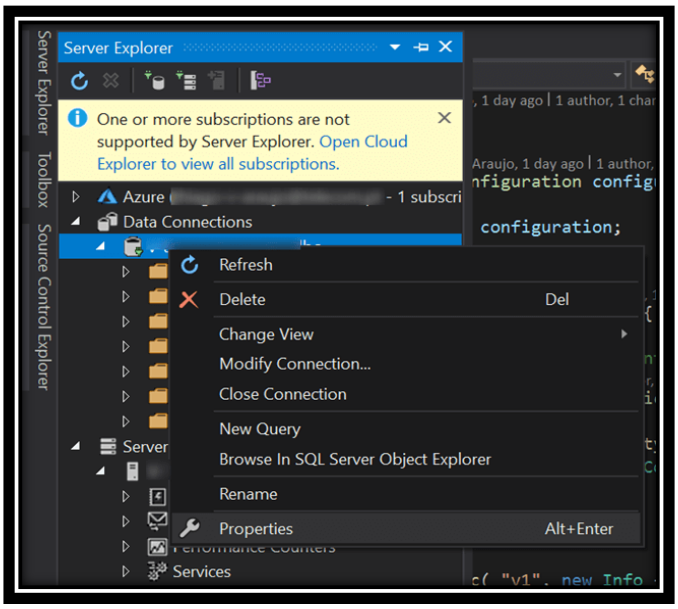

Pick up your server name and push OK.



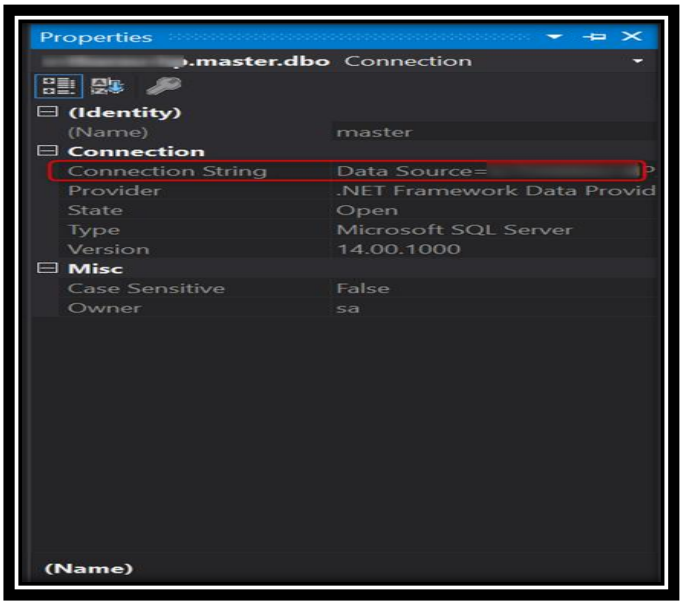
Check the result.




Right-click on your newly added Data Connection and go to properties.




Get your connection string.




*Your final Start Up class*

With your connection string, your StartUp should look like this.

The items in yellow were added from the connection string above.

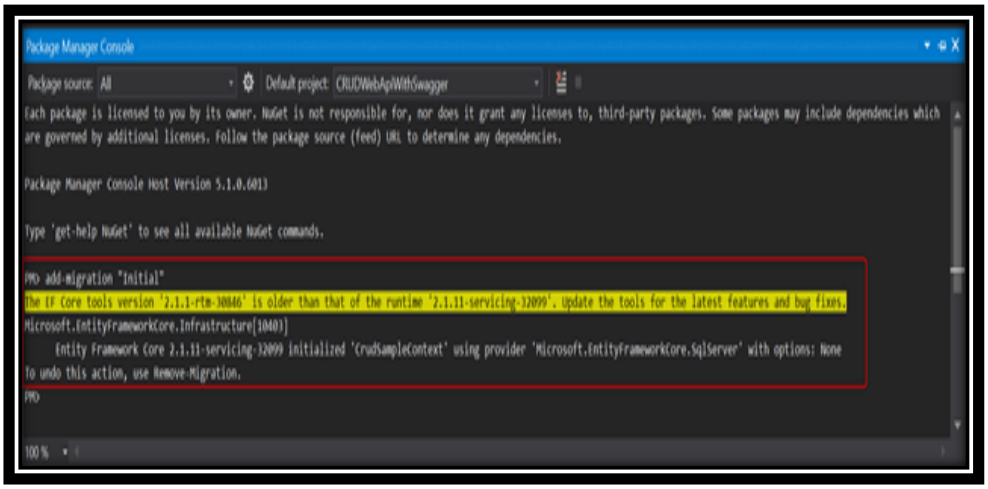
```
public class Startup
  {  public Startup( IConfiguration configuration )
     { Configuration = configuration;
     }
       public IConfiguration Configuration { get; }
       // This method gets called by the runtime. Use this metho
d to add services to the container.
```

```
        public void ConfigureServices( IServiceCollection services
    )
        { services.AddMvc().SetCompatibilityVersion( Compatibil
    ityVersion.Version_2_1 );
            services.AddDbContext<CrudSampleContext>(options
    => options.UseSqlServer(@"Data
    Source=yourServer;Integrated Security=True;Database=CRU
    DSample"));
             // Register Swagger
            services.AddSwaggerGen( c =>
            {  c.SwaggerDoc( "v1", new Info { Title = "Sample API"
    , Version = "version 1" } );
            } );
        }
 // This method gets called by the runtime. Use this method to configu
re the HTTP request pipeline.
        public void Configure( IApplicationBuilder app, IHosting
    Environment env )
        {  if ( env.IsDevelopment() )
```

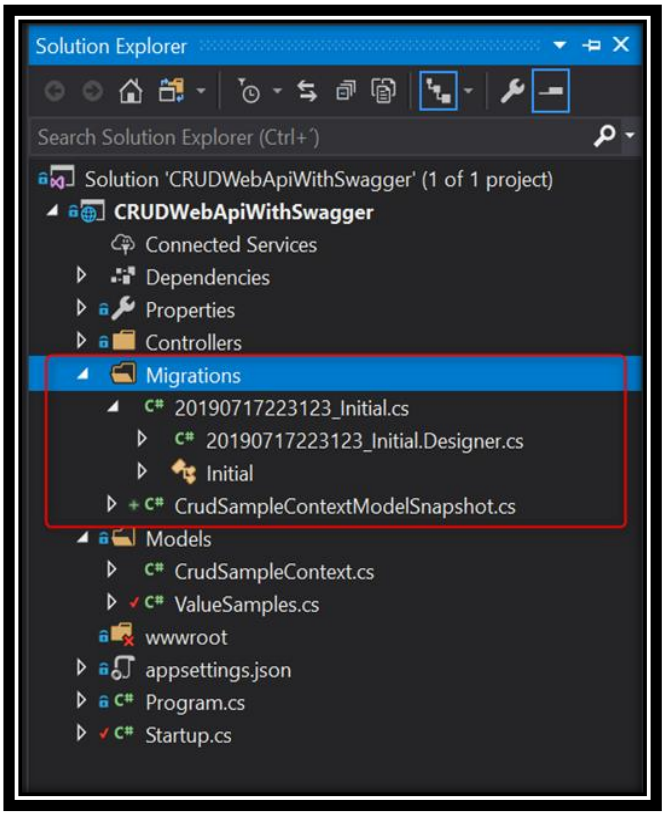
```
            {  app.UseDeveloperExceptionPage();
             // Enable middleware to serve generated Swagger as a J
SON endpoint.
                app.UseSwagger();
                // Enable middleware to serve swagger-
ui (HTML, JS, CSS, etc.),
                // specifying the Swagger JSON endpoint.
                app.UseSwaggerUI( c =>
                {c.SwaggerEndpoint( "/swagger/v1/swagger.json", "
My API V1" );
                } );
              }
               app.UseMvc();
          }
       }
```

**Create your database:**

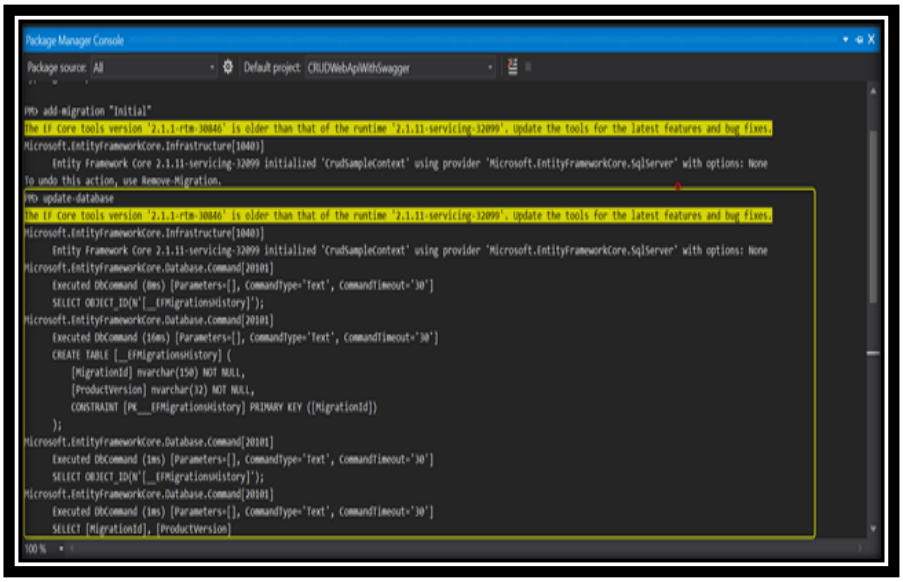
*Add initial migration*, Open your Package Manager Console (PMC) and create your initial migration



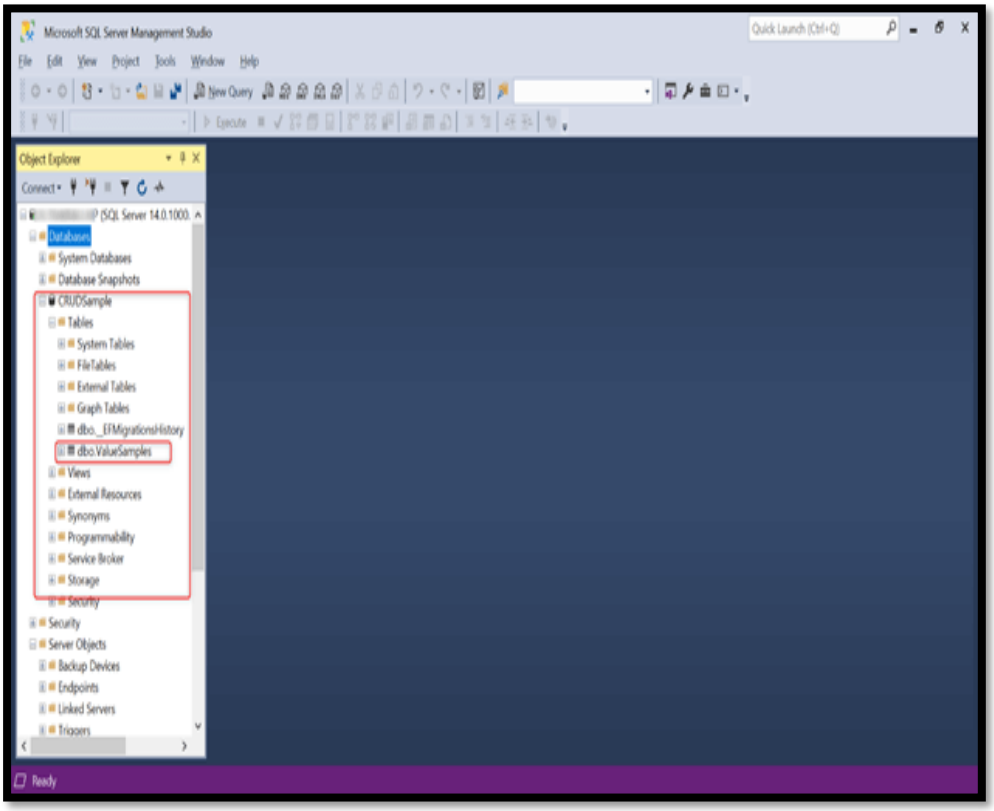

Check the new folder in your project solution




*Create the database*, Run the command update-database




Check your new Database in your SQL Server Management Studio



**CRUD Samples**

Remember, to access your Api through swagger you must go to swagger endpoint. In my case, it is

http://localhost:61986/swagger/index.html

**Update your controller actions**

- **Get:**

**Code:[HttpGet]**

```
public ActionResult<List<ValueSamples>> Get()
{   var itemLst = _crudSampleContext.ValueSamples.ToList();
    return new List<ValueSamples>(itemLst);
}
```

**Result from swagger:**

**Database**


- **Create:**

**Code:** [HttpPost]

public void Post([FromBody] ValueSamples createSample)

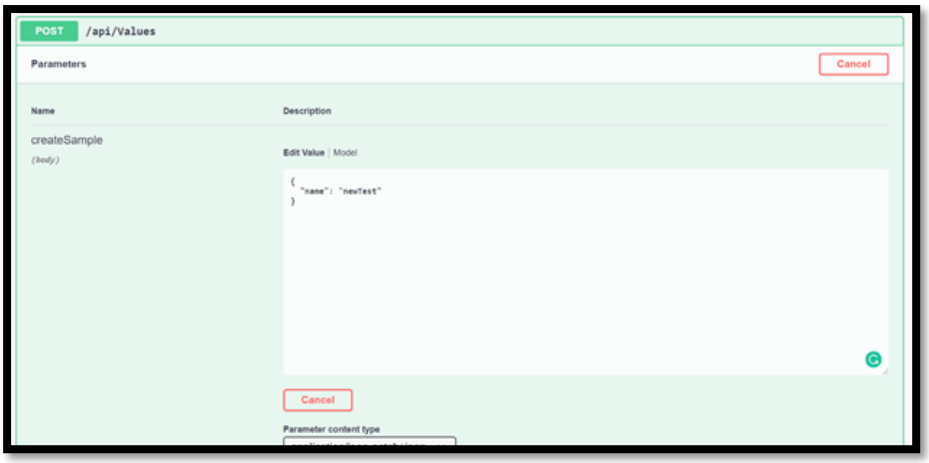
{ _

crudSampleContext.ValueSamples.Add(createSample);

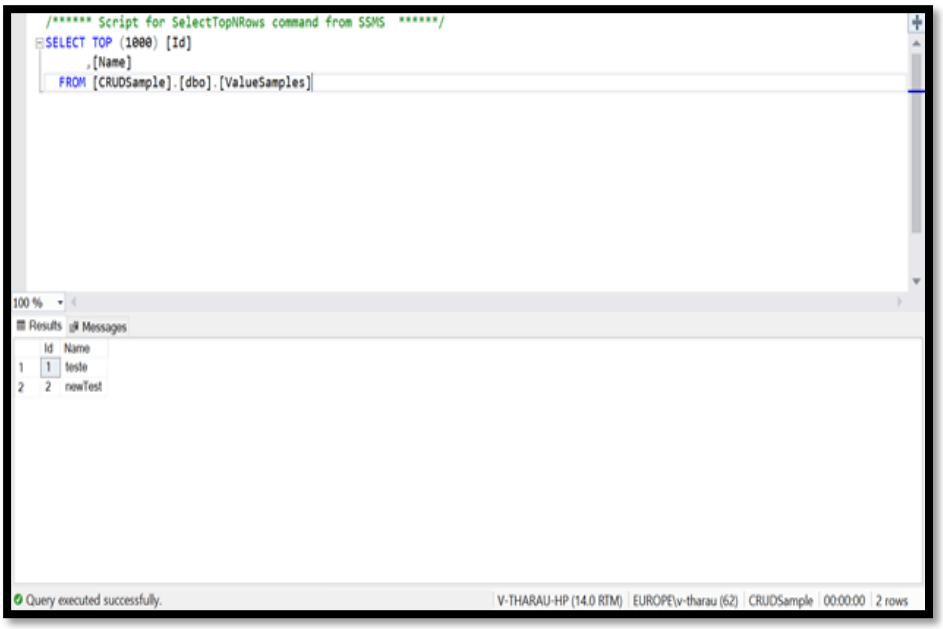
_crudSampleContext.SaveChanges();

}

**Result from swagger:**



**Database**

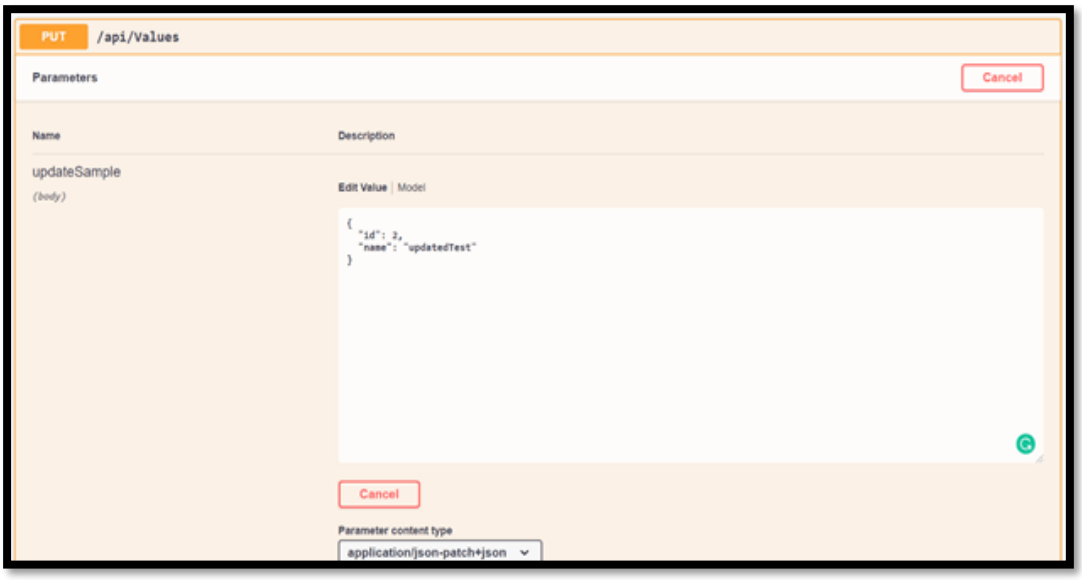



- **Update**

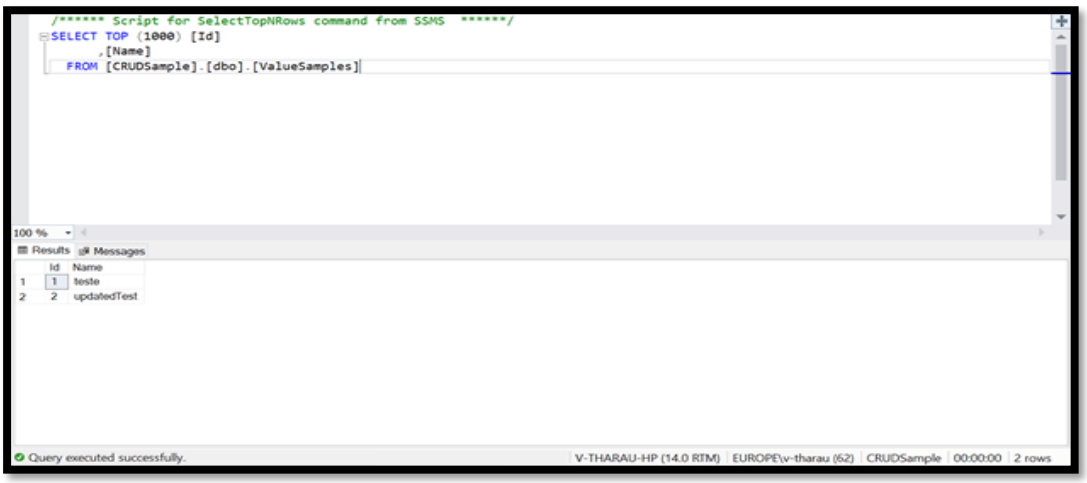
**Code:**

```
[HttpPut]
public void Put([FromBody] ValueSamples updateSample)
        {
          _crudSampleContext.ValueSamples.Update(updateSample);
          _crudSampleContext.SaveChanges();
        }
```

**Result from swagger:**




**Database**



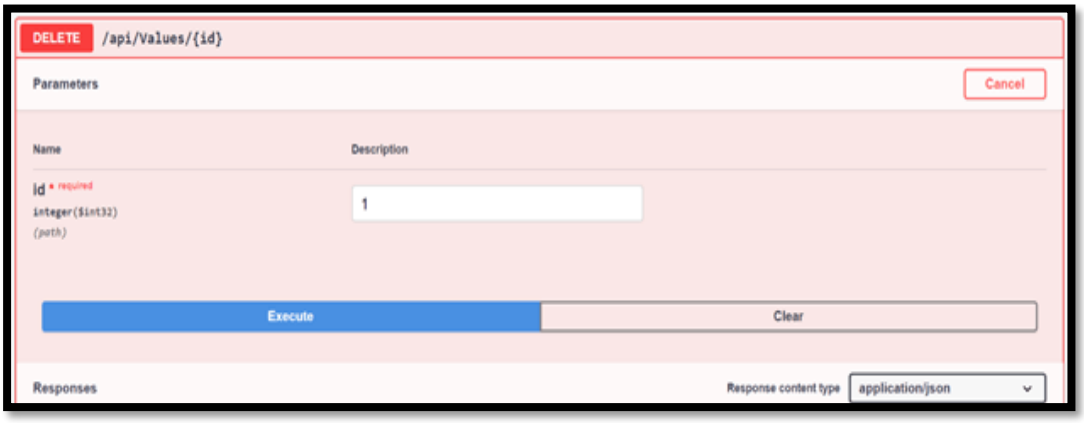

- **Delete**

**Code: [HttpDelete("{id}")]**

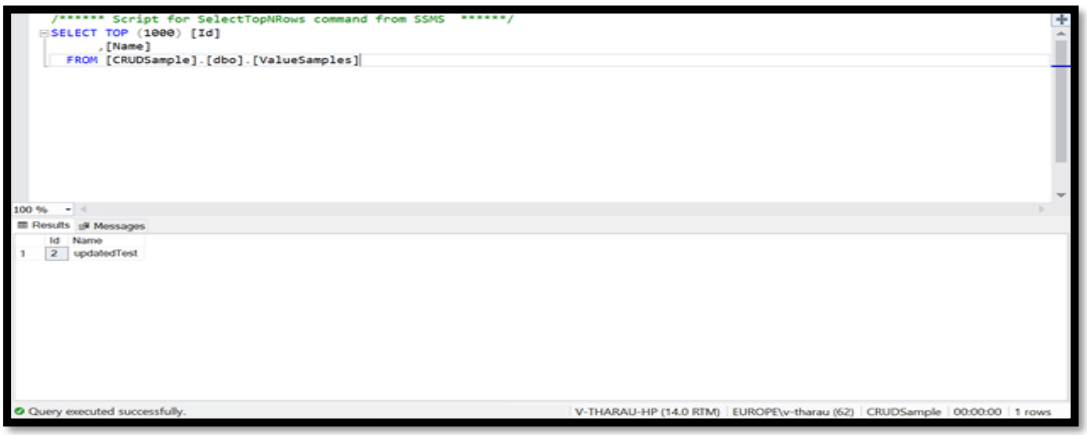
```
public void Delete(int id)
```

```
{
    var itemToDelete = _crudSampleContext.ValueSamples.Where(x =
> x.Id == id).FirstOrDefault();
    _crudSampleContext.ValueSamples.Remove(itemToDelete);
    _crudSampleContext.SaveChanges();
}
```

**Result from swagger:**




**Database**



- **Single Get:**

**Code:** **[HttpGet("{id}")]**

```
public ActionResult<string> Get(int id)
{  var itemToReturn = _crudSampleContext.ValueSamples.Where(x
=> x.Id == id).FirstOrDefault();
   return itemToReturn.Name;
}
```

**Result from swagger:**


Your controller must look like this,

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

```
{  private readonly CrudSampleContext _crudSampleContext;
    public ValuesController(CrudSampleContext crudSampleConte
xt)
    {
       this._crudSampleContext = crudSampleContext;
    }
     // GET api/values
    [HttpGet]
    public ActionResult<List<ValueSamples>> Get()
    { var itemLst = _crudSampleContext.ValueSamples.ToList();
       return new List<ValueSamples>(itemLst);
    }
     // GET api/values/5
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    { var itemToReturn = _crudSampleContext.ValueSamples.Wher
e(x => x.Id == id).FirstOrDefault();
       return itemToReturn.Name;
    }
```

```
        // POST api/values
        [HttpPost]
        public void Post([FromBody] ValueSamples createSample)
        {  _crudSampleContext.ValueSamples.Add(createSample);
           _crudSampleContext.SaveChanges();
        }
        // PUT api/values/5
        [HttpPut]
        public void Put([FromBody] ValueSamples updateSample)
        {
           _crudSampleContext.ValueSamples.Update(updateSample);
           _crudSampleContext.SaveChanges();
        }
        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
           var itemToDelete = _crudSampleContext.ValueSamples.Wher
e(x => x.Id == id).FirstOrDefault();
```

```
            _crudSampleContext.ValueSamples.Remove(itemToDelete);
            _crudSampleContext.SaveChanges();
        }
    }
```

Congratulations, you have successfully implemented your CRUD using Entity Framework Core.

# Practical N0. 9

## Aim: Build and Run an Event-Driven ASP.NET Core Microservice Architecture.

**Theory:** Event-driven architecture is a software design pattern that focuses on the production, detection, consumption,

and reaction to events that occur in a system. Microservice architecture is a software design pattern that structures an application as a collection of small, independent services that communicate with each other over a network.

When combined, event-driven architecture and microservice architecture can provide a scalable, flexible, and resilient system that can quickly respond to changes in the environment.

By following below steps, you can create an event-driven ASP.NET Core microservice architecture that can quickly respond to changes in the environment, scale horizontally, and be resilient to failures.

**Steps: 1: Install Visual Studio Community** (it's free) with the ASP.NET and web development workload.

**Create a solution** and add the two ASP.NET Core 5 Web API projects "UserService" and "PostService". Disable HTTPS and activate OpenAPI Support.

For both projects **install the following NuGet packages**:

- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore.Sqlite
- Newtonsoft.Json
- RabbitMQ.Client

**Implement the UserService**

Create the User Entity: [ **User.cs**]

```
namespace UserService.Entities
{    public class User
   {  public int ID { get; set; }
      public string Name { get; set; }
      public string Mail { get; set; }
      public string OtherData { get; set; }
   }
}
```

Create the UserServiceContext: [**UserServiceContext.cs**]

```
using Microsoft.EntityFrameworkCore;
namespace UserService.Data
{   public class UserServiceContext : DbContext
   {   public UserServiceContext
(DbContextOptions<UserServiceContext> options)
          : base(options)
      {
      }
      public DbSet<UserService.Entities.User> User { get; set; }
   }
}
```

**Edit Startup.cs** to configure the UserServiceContext to use Sqlite and call Database.EnsureCreated() to make sure the database contains the entity schema:

```
public void ConfigureServices(IServiceCollection services)
{ services.AddControllers();
   services.AddSwaggerGen(c =>
   { c.SwaggerDoc("v1", new OpenApiInfo { Title =
"UserService", Version = "v1" });
   });
```

```
        services.AddDbContext<UserServiceContext>(options =>
            options.UseSqlite(@"Data Source=user.db"));
    }
    public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env, UserServiceContext dbContext)
    {
        if (env.IsDevelopment())
        {       dbContext.Database.EnsureCreated();
```

**Create the UserController** (It implements only the methods necessary for this demo):[**UserController.cs**]

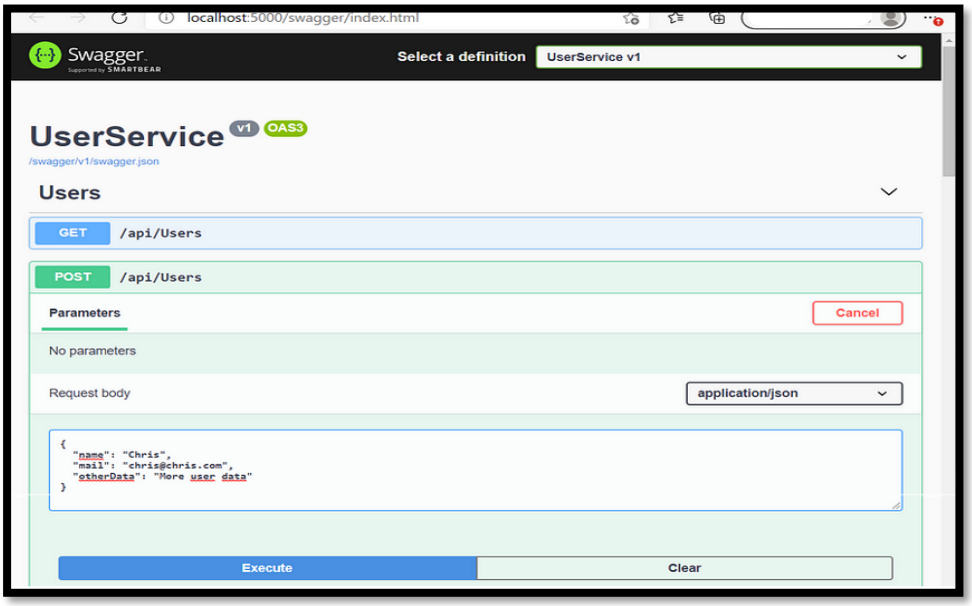
```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using UserService.Data;
using UserService.Entities;
namespace UserService.Controllers
{  [Route("api/[controller]")]
   [ApiController]
```

```
public class UsersController : ControllerBase
{  private readonly UserServiceContext _context;
   public UsersController(UserServiceContext context)
   {   context = context;
   }
[HttpGet]
   public async Task<ActionResult<IEnumerable<User>>> GetUser()
   { return await _context.User.ToListAsync();
   }
[HttpPut("{id}")]
   public async Task<IActionResult> PutUser(int id, User user)
   {_context.Entry(user).State = EntityState.Modified;
      await _context.SaveChangesAsync();
      return NoContent();
   }
[HttpPost]
   public async Task<ActionResult<User>> PostUser(User user)
   {_context.User.Add(user);
      await _context.SaveChangesAsync();
```

```
            return CreatedAtAction("GetUser", new { id = user.ID }, user);
        }
    }
}
```

Debug the UserService project and it will start your browser. You can use the swagger UI to

test if creating and reading users is working:




## Implement the PostService

Create the User and Post entities:

**[User.cs]**

```
namespace PostService.Entities
```

```
{  public class User
   {   public int ID { get; set; }
       public string Name { get; set; }
   }
}
```

**[Post.cs]**

```
namespace PostService.Entities
{ public class Post
   {  public int PostId { get; set; }
       public string Title { get; set; }
       public string Content { get; set; }
       public int UserId { get; set; }
       public User User { get; set; }
   }
}
```

Create the PostServiceContext: **[PostServiceContext.cs ]**

```
using Microsoft.EntityFrameworkCore;
namespace PostService.Data
{ public class PostServiceContext : DbContext
```

```
    {  public PostServiceContext
(DbContextOptions<PostServiceContext> options)
            : base(options)
        {
        }
        public DbSet<PostService.Entities.Post> Post { get; set; }
        public DbSet<PostService.Entities.User> User { get; set; }
    }
}
```

**Edit startup.cs** to configure the UserServiceContext to use Sqlite and call Database.EnsureCreated() to make sure the database contains the entity schema:

```
public void ConfigureServices(IServiceCollection services)
{  services.AddControllers();
    services.AddSwaggerGen(c =>
    {    c.SwaggerDoc("v1", new OpenApiInfo { Title = "PostService",
Version = "v1" });
    });
    services.AddDbContext<PostServiceContext>(options
=>options.UseSqlite(@"Data Source=post.db"));
```

```
}
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env, PostServiceContext dbContext)
{    if (env.IsDevelopment())
   {       dbContext.Database.EnsureCreated
```

Create the PostController: **[ PostController.cs]**

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using PostService.Data;
using PostService.Entities;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace PostService.Controllers
{  [Route("api/[controller]")]
[ApiController]
   public class PostController : ControllerBase
   {  private readonly PostServiceContext _context;
      public PostController(PostServiceContext context)
```

```
        { _context = context;
        }
    [HttpGet]
        public async Task<ActionResult<IEnumerable<Post>>> GetPost()
        { return await _context.Post.Include(x => x.User).ToListAsync();
        }
    [HttpPost]
        public async Task<ActionResult<Post>> PostPost(Post post)
        {  _context.Post.Add(post);
           await _context.SaveChangesAsync();
           return CreatedAtAction("GetPost", new { id = post.PostId },
post);
        }
    }
}
```

Currently, we can't insert posts, because there are no users in the PostService database.

**Step2: Use RabbitMQ and Configure Exchanges and Pipelines**

In the second part we will **get RabbitMQ running**. Then we will **use the RabbitMQ admin web UI to configure**

**the exchanges and pipelines** for the application. Optionally we can use the admin UI to send messages to RabbitMQ.

The easiest way to get RabbitMQ running is to **install Docker Desktop**. Then **issue the following command** (in one line in a console window) to start a RabbitMQ container with admin UI :

```
C:\dev>docker run -d  -p 15672:15672 -p 5672:5672 --hostname my-rabbit --name some-rabbit rabbitmq:3-management
```

**Open your browser** on port 15672 and log in with the username “guest” and the password “guest”.Use the web UI

to **create an Exchange** with the name “user” of type “Fanout” and **two queues** “user.postservice” and

“user.otherservice”. You can also use the web UI to publish messages to the exchange and see how they get queued:


**Step3: Publish and Consume Integration Events in the Microservices**

In this part we will **bring the .NET microservices and RabbitMQ together**.

The **UserService publishes** events. The **PostService consumes** the events and **adds/updates the users in its database**.

**Modify UserService.UserController** to publish the integration events for user creation and update to RabbitMQ:

```
using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Newtonsoft.Json;

using RabbitMQ.Client;

using System.Collections.Generic;

using System.Text;

using System.Threading.Tasks;
```

```
using UserService.Data;
using UserService.Entities;
namespace UserService.Controllers
{  [Route("api/[controller]")]
   [ApiController]
   public class UsersController : ControllerBase
   { private readonly UserServiceContext _context;
      public UsersController(UserServiceContext context)
      { _context = context;
      }
[HttpGet]
      public async Task<ActionResult<IEnumerable<User>>> GetUser()
      {   return await _context.User.ToListAsync();
      }
      private void PublishToMessageQueue(string integrationEvent, string
eventData)
      { // TOOO: Reuse and close connections and channel, etc,
         var factory = new ConnectionFactory();
         var connection = factory.CreateConnection();
```

```
        var channel = connection.CreateModel();
        var body = Encoding.UTF8.GetBytes(eventData);
        channel.BasicPublish(exchange: "user",
                             routingKey: integrationEvent,
                             basicProperties: null,
                             body: body);
    }
[HttpPut("{id}")]
    public async Task<IActionResult> PutUser(int id, User user)
    {  _context.Entry(user).State = EntityState.Modified;
        await _context.SaveChangesAsync();

        var integrationEventData = JsonConvert.SerializeObject(new
        {  id = user.ID,
           newname = user.Name
        });
        PublishToMessageQueue("user.update", integrationEventData);
        return NoContent();
    }
```

```
[HttpPost]
    public async Task<ActionResult<User>> PostUser(User user)
    { _context.User.Add(user);
      await _context.SaveChangesAsync();
      var integrationEventData = JsonConvert.SerializeObject(new
      {  id = user.ID,
         name = user.Name
      });
      PublishToMessageQueue("user.add", integrationEventData);
      return CreatedAtAction("GetUser", new { id = user.ID }, user);
    }
  }
}
```

The connection and other RabbitMQ objects are not correctly closed in these examples. They should also be reused. See the official RabbitMQ .NET tutorial and my follow-up article.

**Modify (**and misuse**) PostService.Program** to subscribe to the integration events and apply the changes to the PostService database:

```
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Hosting;
using Newtonsoft.Json.Linq;
using PostService.Data;
using PostService.Entities;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Linq;
using System.Text;
namespace PostService
{ public class Program
   { public static void Main(string[] args)
      {  ListenForIntegrationEvents();
         CreateHostBuilder(args).Build().Run();
      }
      private static void ListenForIntegrationEvents()
      {  var factory = new ConnectionFactory();
         var connection = factory.CreateConnection();
```

```
var channel = connection.CreateModel();
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{     var contextOptions = new
DbContextOptionsBuilder<PostServiceContext>()
    .UseSqlite(@"Data Source=post.db")
    .Options;
   var dbContext = new PostServiceContext(contextOptions);
   var body = ea.Body.ToArray();
   var message = Encoding.UTF8.GetString(body);
   Console.WriteLine(" [x] Received {0}", message);

   var data = JObject.Parse(message);
   var type = ea.RoutingKey;
   if (type == "user.add")
   {   dbContext.User.Add(new User()
      {  ID = data["id"].Value<int>(),
         Name = data["name"].Value<string>()
      });
```

```
                dbContext.SaveChanges();
            }
            else if (type == "user.update")
            {  var user = dbContext.User.First(a => a.ID ==
data["id"].Value<int>());
                user.Name = data["newname"].Value<string>();
                dbContext.SaveChanges();
            }
        };
        channel.BasicConsume(queue: "user.postservice",
                             autoAck: true,
                             consumer: consumer);
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
```

```
        }
    }
```

**Step 4: Test the Workflow**

In the final part we will test the whole workflow:

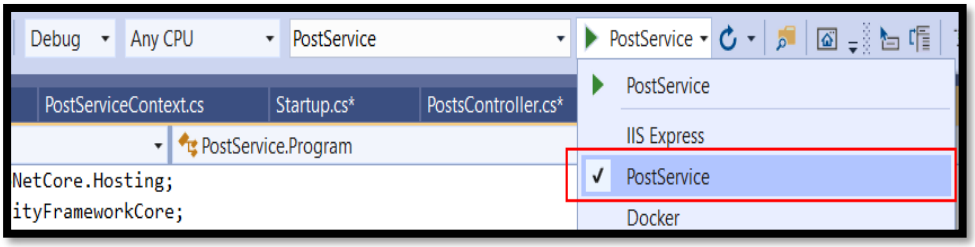
**Summary of the steps in the last part** (you can access the services with the Swagger UI):

- Call the UserService REST API and add a user to the user DB
- The UserService will create an event that the PostService consumes and adds the user to the post DB
- Access the PostService REST API and add a post for the user.
- Call the PostService REST API and load the post and user from the post DB
- Call the UserService REST API and rename the user
- The UserService will create an event that the PostService consumes and updates the user's name in the post DB
- Call the PostService REST API and load the post and renamed user from the post DB

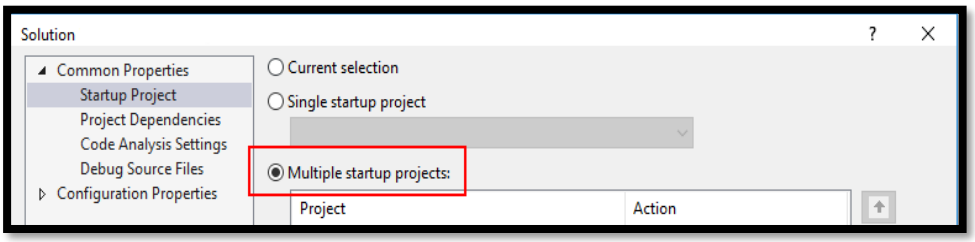
The user DB must be empty. You can delete the user.db (in the Visual Studio explorer) if you

created users in previous steps of this guide. The calls to Database.EnsureCreated() will recreate the DBs on startup.

**Configure both projects to run as service**:




**Change the App-URL of the PostService to another port** (e.g. http://localhost:5001) so that both projects can be run in parallel. Configure the solution to start both projects and start debugging:




Use the Swagger UI to create a user in the UserService:

```
{
 "name": "Chris",
 "mail": "chris@chris.com",
 "otherData": "Some other data"
}
```

The generated userId might be different in your environment:


The integration event replicates the user to the PostService:


Now you can create a post in the PostServive Swagger UI (use your userId):

{

"title": "MyFirst Post",

"content": "Some interesting text",

"userId": 1

}

Read all posts. The username is included in the result:

Change the username in the UserService Swagger UI:

```
{
 "id": 1,
 "name": "My new name"
}
```

Then read the posts again and see the changed username:

# Practical N0. 10

## Aim: Build and Run application for Upload and Download Multiple Files Using Web API.

**Theory:**

Begin with creating an empty web API project in visual studio & for target framework choose

**.Net 5.0.**

Create a Services folder and inside that create one FileService class and IFileService Interface in it.

We have used three methods in this FileService.cs

- UploadFile
- DownloadFile

Since we need a folder to store these uploading files, here we have added one more parameter to pass the folder name as a string where it will store all these files.

**[FileService.cs]**

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using System.Linq;
using System.Threading.Tasks;
namespace UploadandDownloadFiles.Services
{  public class FileService :IFileService
   {   #region Property
       private IHostingEnvironment _hostingEnvironment;
       #endregion
        #region Constructor
       public FileService(IHostingEnvironment hostingEnvironment)
       { _hostingEnvironment = hostingEnvironment;
       }
       #endregion
        #region Upload File
       public void UploadFile(List<IFormFile> files, string subDirectory)
       {  subDirectory = subDirectory ?? string.Empty;
```

```
            var target = Path.Combine(_hostingEnvironment.ContentRootPat
h, subDirectory);
             Directory.CreateDirectory(target);
             files.ForEach(async file =>
            {
               if (file.Length <= 0) return;
               var filePath = Path.Combine(target, file.FileName);
               using (var stream = new FileStream(filePath, FileMode.Create)
)
               {  await file.CopyToAsync(stream);
               }
            });
        }
        #endregion
         #region Download File
        public (string fileType, byte[] archiveData, string archiveName) Do
wnloadFiles(string subDirectory)
  {   var zipName = $"archive-{DateTime.Now.ToString("yyyy_MM_dd-
HH_mm_ss")}.zip";
```

```
var files = Directory.GetFiles(Path.Combine(_hostingEnvironment.C
ontentRootPath, subDirectory)).ToList();              using (var memoryStr
eam = new MemoryStream())
        {  using (var archive = new ZipArchive(memoryStream, ZipArchi
veMode.Create, true))
            {  files.ForEach(file =>
               {  var theFile = archive.CreateEntry(file);
                  using (var streamWriter = new StreamWriter(theFile.Open
()))
                  {  streamWriter.Write(File.ReadAllText(file));
                  }
               });
            }
             return ("application/zip", memoryStream.ToArray(), zipName
);
         }
       }
      #endregion
       #region Size Converter
```

```
public string SizeConverter(long bytes)
{  var fileSize = new decimal(bytes);
   var kilobyte = new decimal(1024);
   var megabyte = new decimal(1024 * 1024);
   var gigabyte = new decimal(1024 * 1024 * 1024);
    switch (fileSize)
   {  case var _ when fileSize < kilobyte:
         return $"Less then 1KB";
      case var _ when fileSize < megabyte:
         return $"{Math.Round(fileSize / kilobyte, 0, MidpointRoundi
ng.AwayFromZero):##,###.##}KB";
      case var _ when fileSize < gigabyte:
         return $"{Math.Round(fileSize / megabyte, 2, MidpointRoun
ding.AwayFromZero):##,###.##}MB";
      case var _ when fileSize >= gigabyte:
         return $"{Math.Round(fileSize / gigabyte, 2, MidpointRound
ing.AwayFromZero):##,###.##}GB";
      default:
         return "n/a";
```

```
            }
        }
        #endregion
    }
}
```

*SizeConverter function is used to get the actual size of our uploading files to the server.*

**[IFileService.cs]**

```
using Microsoft.AspNetCore.Http;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace UploadandDownloadFiles.Services
{public interface IFileService
   {void UploadFile(List<IFormFile> files, string subDirectory);
      (string fileType, byte[] archiveData, string archiveName) Download
Files(string subDirectory);
       string SizeConverter(long bytes);
```

```
    }
}
```

*Let's add this service dependency in a startup.cs file*

**[Startup.cs]**

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.OpenApi.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using UploadandDownloadFiles.Services;
namespace UploadandDownloadFiles
```

```
{  public class Startup
   {  public Startup(IConfiguration configuration)
      { Configuration = configuration;
      }
        public IConfiguration Configuration { get; }
        // This method gets called by the runtime. Use this method to add s
ervices to the container.
      public void ConfigureServices(IServiceCollection services)
      {  services.AddControllers();
services.AddSwaggerGen(c =>{c.SwaggerDoc("v1", new OpenApiInfo { Title = "Upl
oadandDownloadFiles", Version = "v1" });
    });
          services.AddTransient<IFileService, FileService>();
      }
        // This method gets called by the runtime. Use this method to confi
gure the HTTP request pipeline.
      public void Configure(IApplicationBuilder app, IWebHostEnvironment e
nv)
      {
```

```
                if (env.IsDevelopment())
                {  app.UseDeveloperExceptionPage();
                   app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Upl
oadandDownloadFiles v1"));
                }
                 app.UseHttpsRedirection();
                 app.UseRouting();
                 app.UseAuthorization();
                 app.UseEndpoints(endpoints =>
                {endpoints.MapControllers();
                });
              }
            }
          }
```

Create a FileController & now inject this IFileService using Constructor injection inside this FileController.

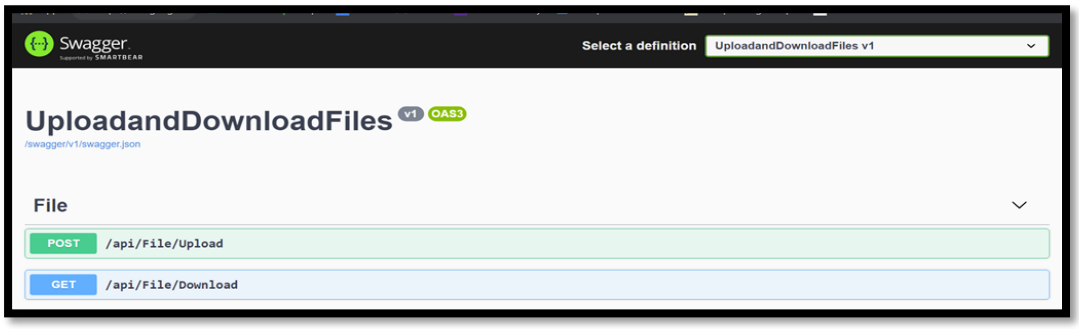
**[FileController.cs]**

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using UploadandDownloadFiles.Services;
namespace UploadandDownloadFiles.Controllers
{  [Route("api/[controller]")]
   [ApiController]
   public class FileController : ControllerBase
   {  #region Property
      private readonly IFileService _fileService;
      #endregion
       #region Constructor
      public FileController(IFileService fileService)
```

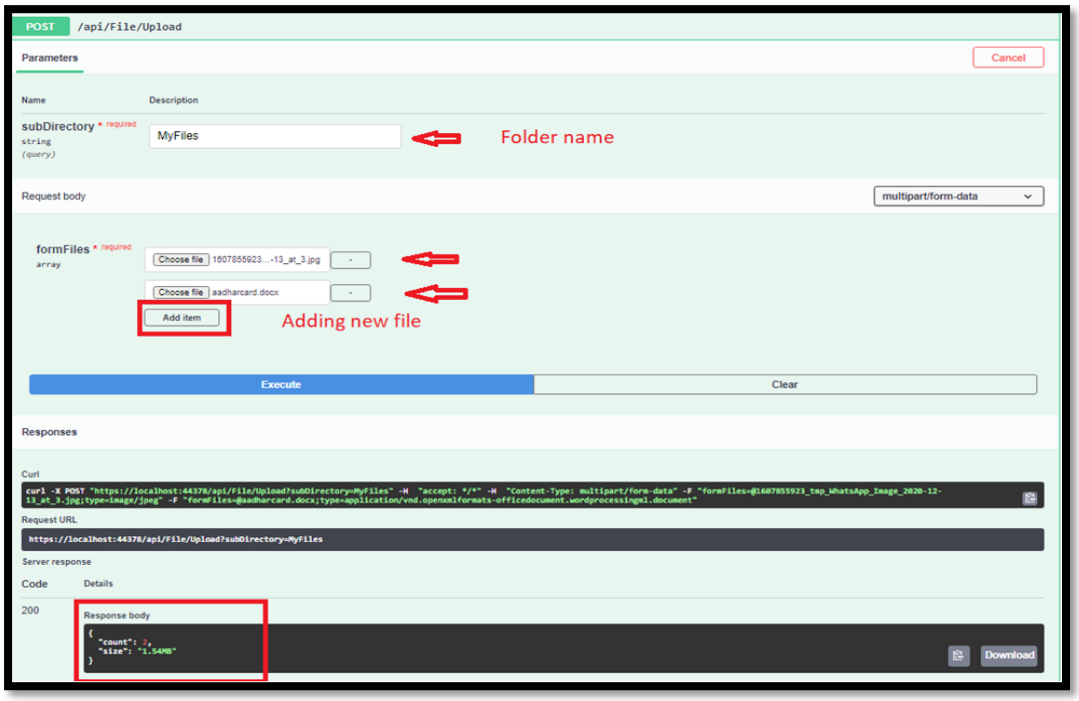
```
        {
            _fileService = fileService;
        }
        #endregion
        #region Upload
        [HttpPost(nameof(Upload))]
      public IActionResult Upload([Required] List<IFormFile> formFiles, [Requi
red] string subDirectory)
            {
            try
            {   _fileService.UploadFile(formFiles, subDirectory);
  return Ok(new { formFiles.Count, Size = _fileService.SizeConverter(formFiles
.Sum(f => f.Length)) });
           }
            catch (Exception ex)
            { return BadRequest(ex.Message);
            }
        }
        #endregion
```

```
        #region Download File
        [HttpGet(nameof(Download))]
        public IActionResult Download([Required]string subDirectory)
        { try
          {   var (fileType, archiveData, archiveName) = _fileService.DownloadFi
    les(subDirectory);
              return File(archiveData, fileType, archiveName);
          }
          catch (Exception ex)
          {  return BadRequest(ex.Message);
          }
         }
        #endregion
    }
}
```

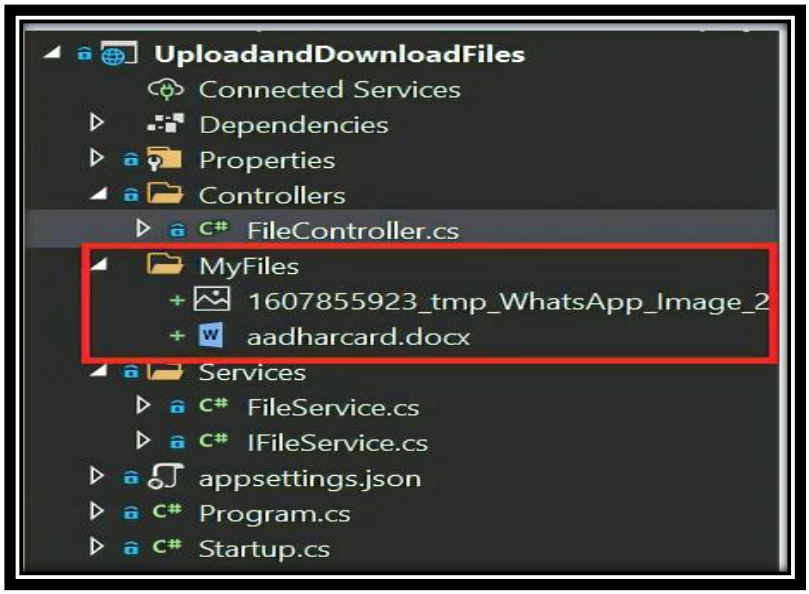
We can test our API's in both swagger and postman.



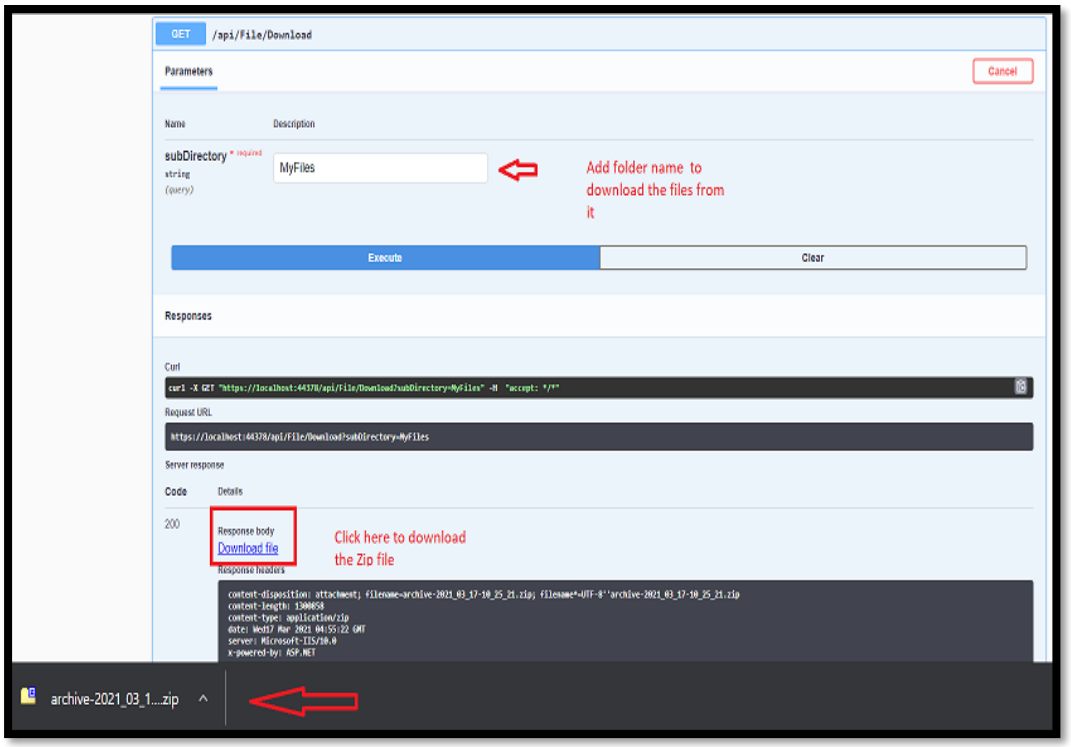

Here we see our two API's which we have created to upload and download, so let's test each of these individually.




Pass the folder name inside the subDirectory and add files below to save inside the server and under the folder name. In response we see the total count of our files and the actual size of our entire files.




Now will check with Download API. Since we have multiple files inside of our folder it will download as a **Zip file** where we need to extract that to check the files.



# Practical N0. 11

## Aim: Demonstrate the use of ASP.NET Core 2.0 Session State in an Application.

**Theory:** In this practical, we will create an empty ASP.NET Core project that doesn't include default features, i.e., an empty shell.

Session state is a feature in ASP.NET Core that you can use to save and store user data while the user browses your web app. Consisting of a dictionary or hash table on the server, session state persists data across requests from a browser. The session data is backed by a cache.

ASP.NET Core maintains session state by giving the client a cookie that contains the session ID, which is sent to the server with each request. The server uses the session ID to fetch the session data. Because the session cookie is specific to the browser, you cannot share sessions across browsers. Session cookies are deleted only when the browser session ends. If a cookie is received for an expired session, a new session that uses the same session cookie is created.

**Steps:** First, create a new empty project using Visual Studio.

File → New →Project

Under ".NET Core", select "ASP.NET Core Web Application". Enter Name and Location. Click OK.

Select "Empty". Click OK.

Next, remove the code from Program.cs and Startup.cs so that they look like below

(Keeping the necessary "using" statements).

```
public class Program {
    public static void Main(string[] args) {
        BuildWebHost(args).Run();
    }
  public static IWebHost BuildWebHost(string[] args) => WebHost.Creat
eDefaultBuilder(args).UseStartup().Build();
}
public class Startup {
public Startup(IHostingEnvironment env,ILoggerFactory loggerFactory,I
Configuration config) {}
    public void ConfigureServices(IServiceCollection services) {
        // setup dependency injection in service container
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment
env) {
        // setup request pipeline using middleware
    }
}
```

***Empty project template in Visual Studio creates a project with Program.cs and Startup.cs classes.***

**[Program.cs]**

Just like a Console application, "public static void Main()" is the starting point of ASP.NET Core applications. We're setting up a host (WebHost) that references the server handling requests (Kestrel). This is achieved using CreateDefaultBuilder() method that configures -

- Kestrel- cross-platform web server
- Root- content root will use the web project's root folder
- IIS- as the reverse proxy server
- Startup- points to a class that sets up configuration, services, and pipeline.

- Configuration- adds JSON and environment variables to IConfiguration available via Dependency Injection (see the next section).
- Logging- adds a Console and Debugs the logging providers.

Once configured, we Build() and Run() the host, at which point, the main thread is blocked and the host starts listening to the request from the Server.

When setting up WebHostBuilder, we could set values for various settings via UseSetting()method, wi8'hich takes in a key/value pair for a property. These properties include applicationName (string), contentRoot (string), detailedErrors (bool), environment (string), URLs (semicolon separated list) and Webroot (string). Some of these properties can also be set via extension methods on WebHostBuilder.

**[Startup.cs]**

This class sets up the application dependency injection services and requests pipeline for our application, using the following methods.

- ConfigureServices(): add services to the service container
- Configure(): setup request pipeline using Middleware.

The parameters for these methods can be seen in the code attachment above. These methods are called in the same sequence as listed above. The important point to remember about the sequence in which they run is that the service container is available after ConfigureServices, i.e., in Configure() method, and not before that.

Using an empty project created above,

amend Startup class ConfigureServicee() method, add services

for session and its backing store,

```
public void ConfigureServices (  IServiceCollection services)
  {  services.AddDistributedMemoryCache();
     services.AddSession();
  }
```

Add the session middleware in Configure() method,

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
   {  app.UseSession();
      app.Use(async (context, next) =>
     {  context.Session.SetString("GreetingMessage", "Hello Session State");
        await next();
     });
```

```
        app.Run(async (context) =>
      { var message = context.Session.GetString("GreetingMessage");
        await context.Response.WriteAsync($"{message}");
      });
    }
```

We can use session in order to share information between different HTTP requests coming from a single browser. The data is stored in a cache (IDistributedCache implementation to be specific) and accessed via HttpContext.Session property.

A cookie is stored in browser to correlated the HTTP requests. The default name of this cookie is AspNet.Session.

During the configuration of session services we can set various properties,

- HttpOnly - sets whether cookie is accessible through JavaScript. Default is true, which means it can't be accessed via scripts on the client-side.
- Name - used to override the default cookie name.
- SecurePolicy - determines if session cookie is only transmitted via HTTPS requests.
- IdleTimeout - sets the time for session expiry, each request resets the timeout. Default is 20 mintues.

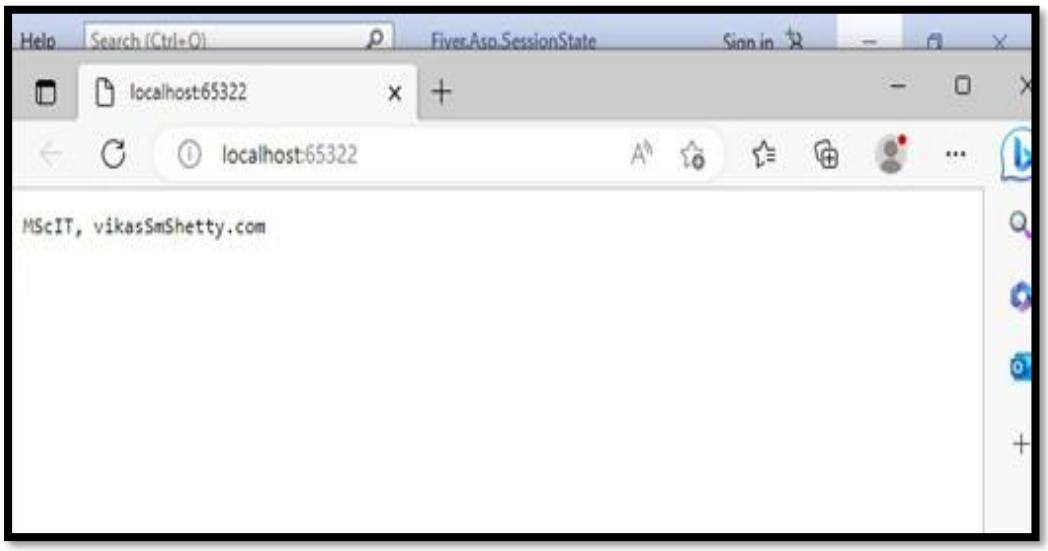
**Storing Objects:**

HttpContext.Session (or ISession that it implements) does not provide a built-in way to store complex objects, however, we can serialise objects into JSON strings to achieve this,

Now we can use these extension methods like below,

```
public void Configure(
    IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
 {  app.UseSession();
     app.Use(async (context, next) =>
    {  context.Session.SetObject("CurrentUser",
          new UserInfo { Username = "MScIT", Email = "vikasSmShetty.com"  });
       await next();
    });
    app.Run(async (context) =>
    {  var user = context.Session.GetObject<UserInfo>("CurrentUser");
```

```
            await context.Response.WriteAsync($"{user.Username}, {user.Em
ail}");
        });
    }
```



# Practical N0. 12

## Aim: Demonstrate the use of ASP.NET Core 2.0 MVC Layout Pages.

**Theory:** MVC layout pages in asp.net core 2.0 help us to create reusable views for our web application.

- we can use them to define the structure of your web pages and include common elements such as headers, footers, and navigation menus.
- This makes it easier to maintain your web application and ensure consistency across all your pages.
- In asp.net core 2.0, layout pages are written in Razor syntax and typically have a .cshtml file extension.
- They can be defined at the application level or at the page level, and can be nested to create more complex layouts.
- Layout pages can also include sections that are defined in content pages, allowing you to customize the content of each page while still maintaining a consistent layout.

I hope you have understood about the layout page from the preceding brief summary.

Now let's implement it practically.

**Steps:** Using an empty project from previous practical i.e practical number 15 amend Startup class to add

services and middleware for MVC.

```
public void ConfigureServices(IServiceCollection services)
  {  services.AddScoped<IGreetingService, GreetingService>();
     services.AddMvc();
  }
   public void Configure(  IApplicationBuilder app, IHostingEnvir
onment env)
  {  app.UseMvc(routes =>
     {    routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
     });
  }
```

**Add a service and model.**

```
public interface IGreetingService
 {  string Greet(string firstname, string surname);
 }
```

```
public class GreetingService : IGreetingService
{  public string Greet(string firstname, string surname)
   { return $"Hello {firstname} {surname}";
   }
}
public class UserViewModel
{  public string Firstname { get; set; }
   public string Surname { get; set; }
}
```

**Add a controller with action method returning ViewResult**.

```
public IActionResult Index()
{ var model = new UserViewModel
   {  Firstname = "Tahir",
      Surname = "Naushad"
   };
   return View(model);
}
```

**Add _Layout.cshtml.**

```
<!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        <strong>I'm in Layout page</strong>
        @RenderBody()
        @RenderSection("footer", required: false)
        @if (IsSectionDefined("links"))
        {  @RenderSection("links", required: false)
        }
        else
        {   <em>No social media links supplied</em>
        }
    </div>
</body>
</html>
```

**Add Index.cshtml.**

```
@model UserViewModel
@{
ViewBag.Title = "ASP.NET Core MVC";
   }
<strong>I'm in View page</strong>
<p>@Greeter.Greet(@Model.Firstname, @Model.Surname)</p>
@section footer{
   <strong>I'm in footer section</strong>
}
```

**Add _ViewImports.cshtml.**

```
@using Fiver.Mvc.Layout.Models.Home
@inject IGreetingService Greeter
```

**Add _ViewStart.cshtml,**

```
@{
   Layout = "_Layout";
}
```

ASP.NET Core MVC provides ways to reuse and share visual elements and common code between different Views. These are,

Layout Page

Start Page

Imports Page.

**Layout Page**

These are used to share common visual elements in your pages and provide a consistent look and feel throughout your application. A layout page is added to the Views/Shared folder and is named (as a convention) _Layout.cshtml. There can be more than one layout pages in your application too.

Views have a Layout property through which they set the layout to use. The layout page is searched in Controller-specific folder and then in the shared folder. Layout page calls @RenderBodymethod to render the contents of a View.

Layout page can also use @RenderSection to decide where sections defined in Views will be placed. These sections can be required or optional. Views define the contents of a section using @sectionsyntax. Layout page can check if a section is defined in the View and acts accordingly

using IsSectionDefined method on the View.

```
@if (IsSectionDefined("links"))
{
```

```
    @RenderSection("links", required: false)
}
else
{
    <em>No social media links supplied</em>
}
```

**Import Page**

As we discussed in Razor post, Views can use directives for a number of things, e.g. importing namespaces (@using), injecting dependencies (@inject), and declaring model type (@model). MVC provides an import page to declare directives common to one or more Views.

Import page is added usually in Views folder and is named _ViewImports.cshtml. It can also be added to other folders (e.g., Controller specific Views folder) however, in this case it will apply to views inside this folder (and its subfolders).

In case of multiple import pages, either the directives close to the Views are used (e.g. @model,@inject) or all are combined (@using, @addTagHelper).

**Start Page**

MVC provides a mechanism to run code common to all views using a start page. Start page run before every view, except for layout page and partial views. Start page is added usually in Views folder and is named _ViewStart.cshtml. There can be multiple start pages, in which case, they will run in hierarchical order from root to subfolders.

Start page is often used to set the Layout page for all the Views in a folder.

# Practical N0. 13

## Aim: Demonstrate Simple Insert and Select (CRUD) Operation Using .NET Core MVC With

### ADO.NET And Entity Framework Core

**Theory:** In this practical, we are going to learn about how to use ADO.NET and Entity framework core in

.NET Core MVC. This way we can use either both or on way to implement your application as per our requirements.

- In .NET Core MVC, CRUD operations refer to the four basic functions of persistent storage: create, read, update, and delete.
- ADO.NET is a data access technology that provides a set of classes for working with relational databases. To perform CRUD operations using ADO.NET, you can use the SqlCommand object to execute SQL queries against the database. For example, to insert a new record into a table, you could create a SqlCommand object with an INSERT statement and execute it using a SqlConnection object.
- Entity Framework Core is an object-relational mapping (ORM) framework that provides a higher-level abstraction for working with

databases. To perform CRUD operations using Entity Framework Core, you can define a model that maps to your database schema, and then use a DbContext object to interact with the database. For example, to insert a new record into a table, you could create a new instance of your model class and add it to the DbContext object using the Add method.

- Both ADO.NET and Entity Framework Core can be used to perform CRUD operations in .NET Core MVC. The choice between them depends on your specific requirements and preferences. ADO.NET provides more control over the SQL queries that are executed, while Entity Framework Core provides a higher-level abstraction and can simplify your code.

I hope you have understood, Now let's implement it practically.

**Step 1:** I have used MS SQLServer for the database.

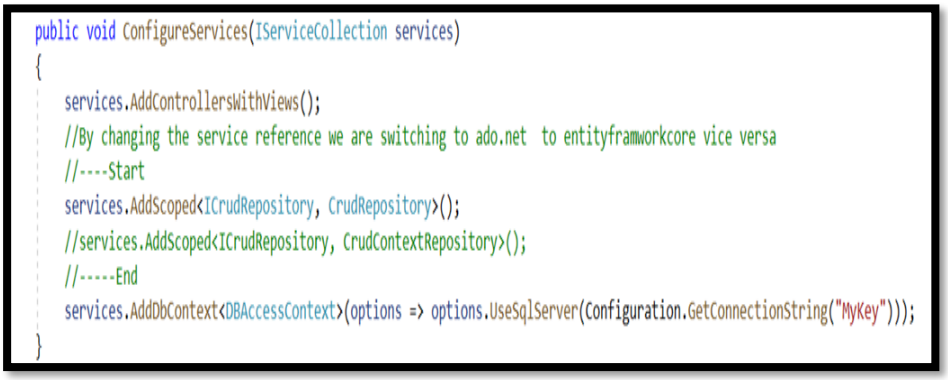
```
USE [CrudDB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Crud_Data](
```

```
[id] [int] IDENTITY(1,1) NOT NULL,
[Name] [varchar](50) NULL,
[City] [varchar](50) NULL,
[InsertDate] [datetime] NULL,
[FatherName] [varchar](50) NULL,
CONSTRAINT [PK_Crud_Data] PRIMARY KEY CLUSTERED
(
[id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_
PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF)
ON [PRIMARY]
) ON [PRIMARY]
GO
```

**Step-2**: Change in the startup.cs file in the project root

```
public void ConfigureServices(IServiceCollection services)
    { services.AddControllersWithViews();
        //By changing the service reference we are switching to ado.net  to e
ntity framwork core vice versa
```

```
        //----Start
        services.AddScoped<ICrudRepository, CrudRepository>();
        //services.AddScoped<ICrudRepository, CrudContextRepository>()
;
        //-----End
        services.AddDbContext<DBAccessContext>(options => options.Us
eSqlServer(Configuration.GetConnectionString("MyKey")));
    }
```




**Step 3:** Connection string in appsettings.json:

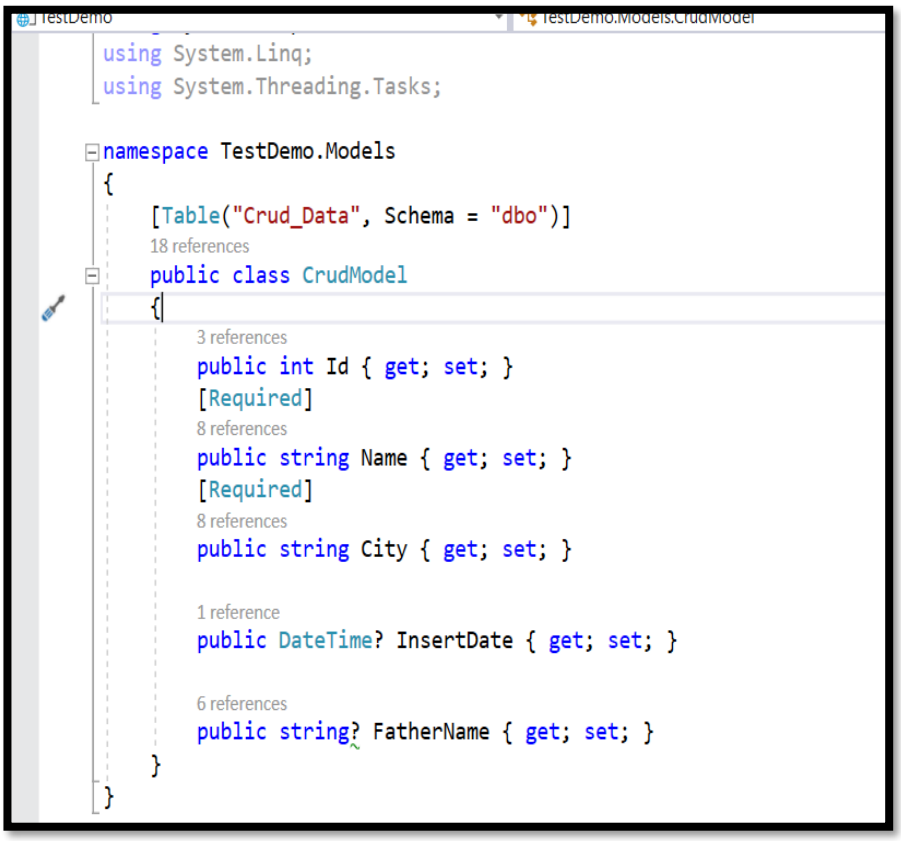
```
{ "Logging": {
   "LogLevel": {
     "Default": "Information",
     "Microsoft": "Warning",
     "Microsoft.Hosting.Lifetime": "Information"
```

```
  } },
 "AllowedHosts": "*",
 "ConnectionStrings": {
   "MyKey": "Data Source=PRO-
ACQER8AM;Initial Catalog=CrudDB;Integrated Security=True;"
 } }
```


**Step 4:** Made model to communicate with View and DB Layer.

```
namespace TestDemo.Models
{ [Table("Crud_Data", Schema = "dbo")]
   public class CrudModel
   { public int Id { get; set; }
      [Required]
      public string Name { get; set; }
      [Required]
```

```
        public string City { get; set; }
        public DateTime? InsertDate { get; set; }
        public string? FatherName { get; set; }
    }
}
```




**Step 5:**Controller class CrudController.cs with dependency injection use with helps to lose coupling while switching from ADO.NET to Entity framework and vice-versa.

```
public class CrudController : Controller
```

```
{  private readonly ICrudRepository crudRepository;
   public CrudController(ICrudRepository crudRepository)
   { this.crudRepository = crudRepository;
   }
   public IActionResult Index()
   { ViewBag.ModelList = crudRepository.GetData();
      return View();
   }
   [HttpPost]
   public IActionResult Index(CrudModel crudModel)
   {  try
      {
         if (ModelState.IsValid)
         {  // ICrudRepository crudRepository = new CrudRepository();
            var result = crudRepository.insert(new string[] { crudModel.
Name, crudModel.City, System.DateTime.Now.ToString(), crudModel.F
atherName });
            ViewBag.ModelList = crudRepository.GetData();
            if (result)
```

```
                {  ViewBag.Msg = "Succeed";
                   ViewBag.alertType = "alert alert-success";
                }
                else
                { ViewBag.Msg = "Insertion failed";
                   ViewBag.alertType = "alert alert-danger";
                }

            }
          }
          catch (Exception ex)
          {  ViewBag.Msg = "Insertion failed";
             ViewBag.alertType = "alert alert-danger";
             ModelState.AddModelError("Message", ex.Message);
          }
            return View();
        }
    }
```

TestDemo | TestDemo.Controllers.CrudController | Index(CrudModel crudModel)

```
public class CrudController : Controller
{
    private readonly ICrudRepository crudRepository;
    0 references
    public CrudController(ICrudRepository crudRepository)
    {
        this.crudRepository = crudRepository;
    }
    0 references
    public IActionResult Index()
    {
        ViewBag.ModelList = crudRepository.GetData();
        return View();
    }
    [HttpPost]
    0 references
    public IActionResult Index(CrudModel crudModel)
    {
        try
        {
            if (ModelState.IsValid)
            {
                // ICrudRepository crudRepository = new CrudRepository();
                var result = crudRepository.insert(new string[] { crudModel.Name, crudModel.City, System.DateTime.Now.ToString(), crudModel.FatherName });
                ViewBag.ModelList = crudRepository.GetData();
                if (result)
                {
                    ViewBag.Msg = "Succeed";
                    ViewBag.alertType = "alert alert-success";
                }
                else
                {
                    ViewBag.Msg = "Insertion failed";
                    ViewBag.alertType = "alert alert-danger";
                }

            }
        }
        catch (Exception ex)
        {
            ViewBag.Msg = "Insertion failed";
            ViewBag.alertType = "alert alert-danger";
            ModelState.AddModelError("Message", ex.Message);
        }

        return View();

    }
}
}
```

**Step 6:** The project consists of class files for the DB layer and Repository folder and one partial view for showing the detail view.

We have created an interface in the repository folder to make a service and use both the classes as a service.

We have created the DBLayer folder and created 2 classes one consists of ado.net insert and select the method and another one for Entity framework Core DBContext.

In Partial View, I have injected the service directly so that you no need to pass the model to communicate with a partial view. you can check the getdata() method by debugging and uncommenting the specific section over there.

System.Data.sqlclient reference required to use ado.net, you can add this by using NuGet package manager.

**DataAccessDB.cs**

```
public class DataAccessDB : IDataAccessDB
  {  private readonly IConfiguration config;
     string connsString = string.Empty;
     public DataAccessDB(IConfiguration config)
     {  this.config = config;
        connsString = config.GetConnectionString("MyKey");
     }
```

```
public List<CrudModel> GetData()
{  // string connsString = config.GetConnectionString("MyKey");// "Data S
ource=PRO-
ACQER8AM;Initial Catalog=CrudDB;Integrated Security=True;";
    List<TestDemo.Models.CrudModel> ModelList = new List<Models.Cru
dModel>();
    using (SqlConnection conn = new SqlConnection(connsString))
    {  conn.Open();
       using (SqlCommand command = new SqlCommand($"select * from [d
bo].[Crud_Data]", conn))
        {  try
           { using (var result = command.ExecuteReader())
             { while (result.Read())
               { ModelList.Add(
              new Models.CrudModel { Id = (int)result.GetValue("Id"), Name =
(string)result.GetValue("Name"), City = (string)result.GetValue("City"),  Father
Name = (string)result.GetValue("FatherName").ToString() });
               }
             }
```

```
                }
                catch (Exception ex)
                {
                }
                finally
                { conn.Close();
                }
                return ModelList;
            }
        }
    }
  public bool insert(string[] Param)
      {   //   string connsString = config.GetConnectionString("MyKey");// "Data
Source=PRO-
ACQER8AM;Initial Catalog=CrudDB;Integrated Security=True;";
          using (SqlConnection conn = new SqlConnection(connsString))
          {  conn.Open();
```

```
            using (SqlCommand command = new SqlCommand($"INSERT INTO
 [dbo].[Crud_Data] ([Name],[City],[InsertDate],FatherName) VALUES ('{Para
m[0]}','{Param[1]}',getdate(),'{Param[3]}')", conn))
            { try
               {   var result = command.ExecuteNonQuery();
                    if (result > 0)
                  {  return true;
                  }
               }
               catch (Exception)
               {
               }
               finally
               { conn.Close();
               }
               return false;
            }
         }
      }
```

```
    }
```

**DBAccessContext.cs**

```
public class DBAccessContext:DbContext
 {  public DBAccessContext(DbContextOptions<DBAccessContext> options):
base(options)
    {
    }
      public DbSet<CrudModel> Crud_Data { get; set; }
   }
```

**CrudRepository.cs**

```
public class CrudRepository : ICrudRepository
{  private readonly IConfiguration configuration;
   private readonly DataAccessDB DB;
   public CrudRepository(IConfiguration configuration)
   { this.configuration = configuration;
      this.DB = new DataAccessDB(configuration);
   }
   public List<CrudModel> GetData()
   {   return DB.GetData();
```

```
    }
      public bool insert(string[] Param)
    { return DB.insert(Param);
      }
   }
```

**CrudContextRepository.cs**

```
public class CrudContextRepository : ICrudRepository
  {  private readonly DBAccessContext dBAccessContext;
     private readonly IConfiguration configuration;
     public CrudContextRepository(DBAccessContext dBAccessContext,IConfi
guration configuration)
     {  this.dBAccessContext = dBAccessContext;
        this.configuration = configuration;
     }
     public  List<CrudModel> GetData()
     { return dBAccessContext.Crud_Data.ToList();
     }
     public bool insert(string[] Param)
     { var model = new CrudModel()
```

```
            {  Name = Param[0],
               City = Param[1],
               InsertDate = System.DateTime.Now,
               FatherName= Param[3]
             };
        dBAccessContext.Crud_Data.Add(model);
          var result= dBAccessContext.SaveChanges();
           if (result>0)
          { return true;
          }
          return false;
       }
    }
```

**ICrudRepository.cs**

```
    public interface ICrudRepository
     {  bool insert(string[] Param);
        List<TestDemo.Models.CrudModel> GetData();
      }
```

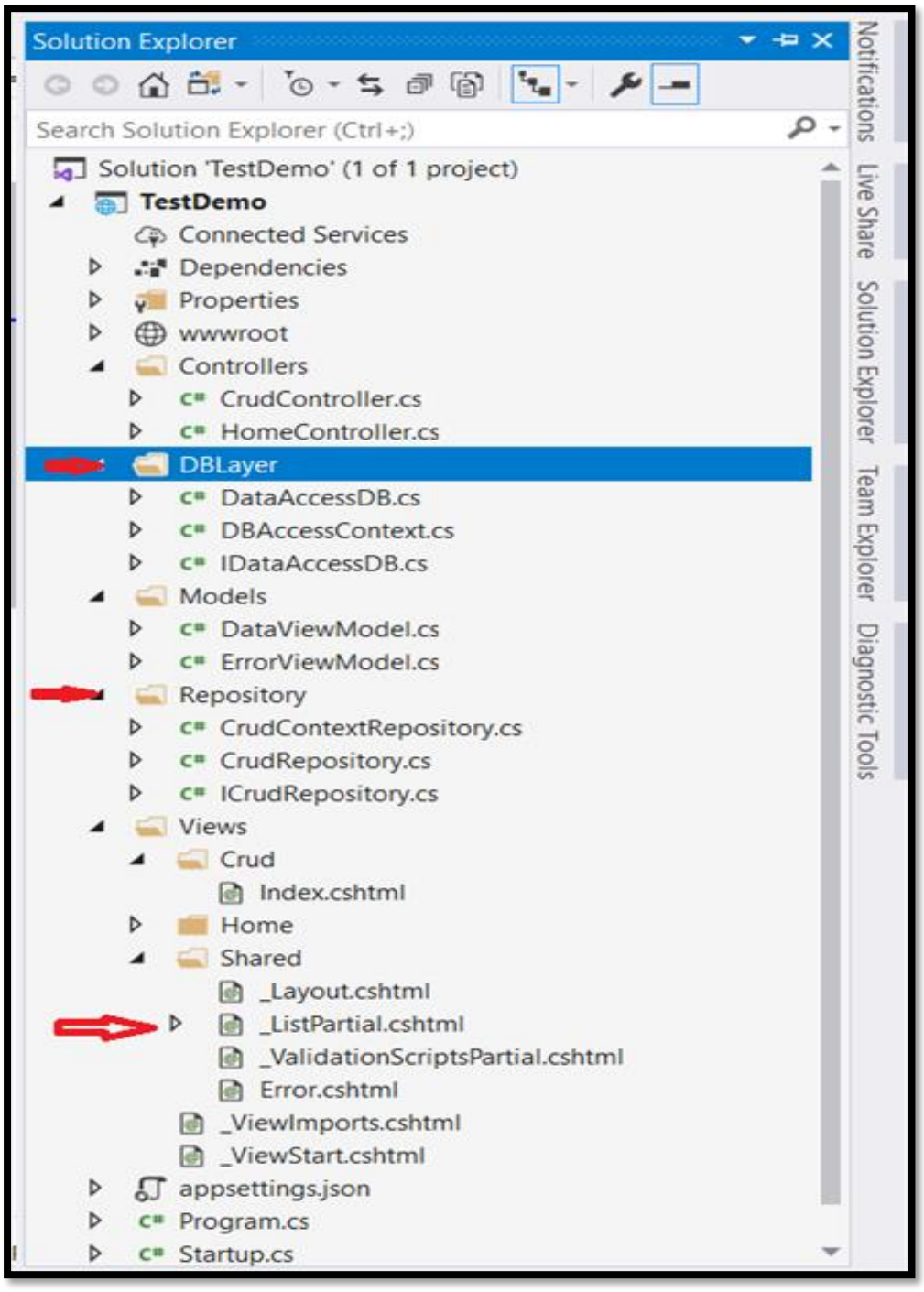
**Index.cshtml under View/Crud/**

```
@model TestDemo.Models.CrudModel
 <partial name="_ValidationScriptsPartial" />
<style>
   .bg-secondary {
      background-color: #f4f5f7 !important;
   }
</style>
<h1>Welcome to Crud</h1>
<form id="Form1" method="post" asp-action="Index" asp-
controller="Crud" >
   @if (ViewBag.Msg != null)
   { <div class="@ViewBag.alertType" role="alert">
         <div asp-validation-summary="All">@ViewBag.Msg</div>
      </div>
      @*<script>
            $(function () {
               $.notify({
                  title: "<strong>Message:</strong> ",
                  message: "@ViewBag.Msg"
```

```
            });
        });
    </script>*@
    ViewBag.Msg = null;
}
<div class="card">
    <div class="card-body">
        <div class="form-group">
            <label asp-for="Name">Name</label>
            <input asp-for="Name" class="form-control">
            <span asp-validation-for="Name" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="City">City</label>
            <input asp-for="City" class="form-control" />
            <span asp-validation-for="City" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="FatherName">Father Name</label>
```

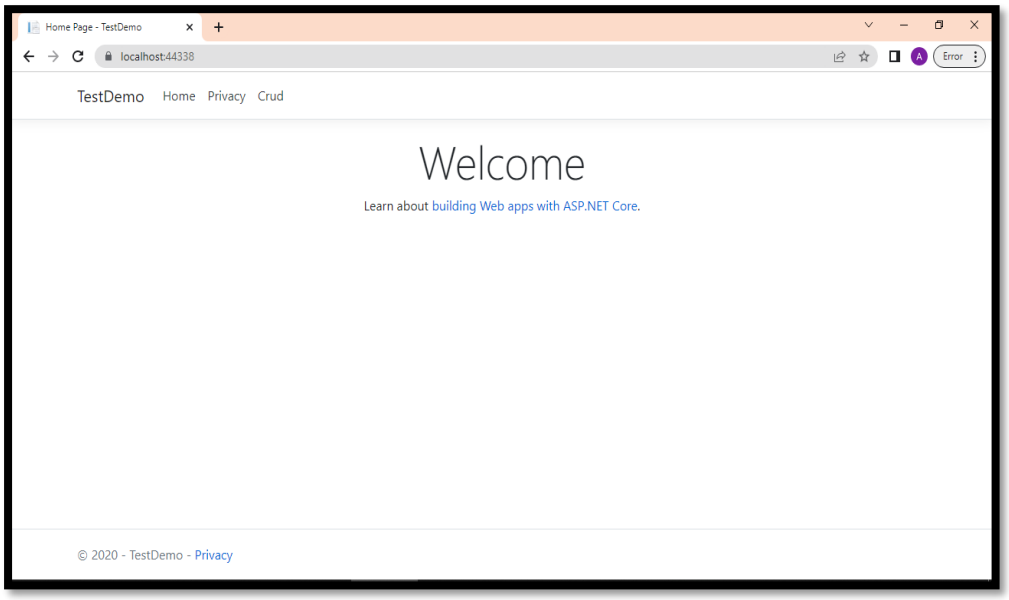
```
                <input asp-for="FatherName" class="form-control" />
                <span asp-validation-for="FatherName" class="text-
danger"></span>
            </div>
        </div>
        <div class="card-footer">
            <button type="submit" class="btn btn-primary">Submit</button>
        </div>
    </div>
     @{ if (ViewBag.ModelList != null)
      { <div class="card">
            <div class="card-body">
                <partial name="_ListPartial" model="@ViewBag.ModelList" />
            </div>
        </div>
      }}
     @*@await Html.PartialAsync("_ListPartial", Model)*@
     @*@(await Html.RenderComponentAsync<TestDemo.Components.Compo
nent>(RenderMode.ServerPrerendered,null))*@
```

```
</form>
```




**Output: click on IIS server and run the project.**

you can see the output on the browser.



# Practical N0. 14

## Aim: Demonstrate the use of Entity Framework (2), With .Net MVC, Database-First

**Theory:** *Entity Framework is an object-relational mapping (ORM) framework that allows us to work with databases*

*using .NET objects. With Database-First approach, we can create a model from an existing database schema. Here's how you can use Entity Framework 2 with .NET MVC using the Database-First approach:*

*We will make a sample app step by step,*

*Step 1: Create a Database and insert Data.*

*Step 2: Create an ASP.NET MVC application*

*Step 3: Reverse Engineer Model*

*Step 4: Create Controller to access data from entity framework*

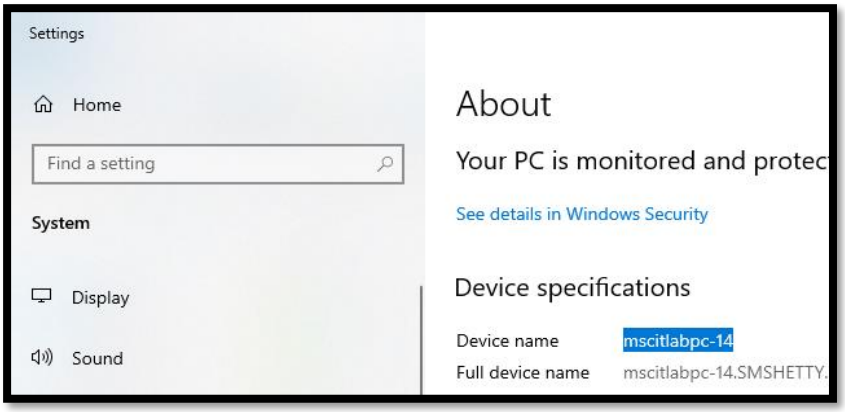
*Step 5: Run the app*

*At the end, we will have an .Net MVC app that can consume a database directly through entity framework.*

**Step 1: Create a Database and insert Data.**

Start SQL Server Management Studio.

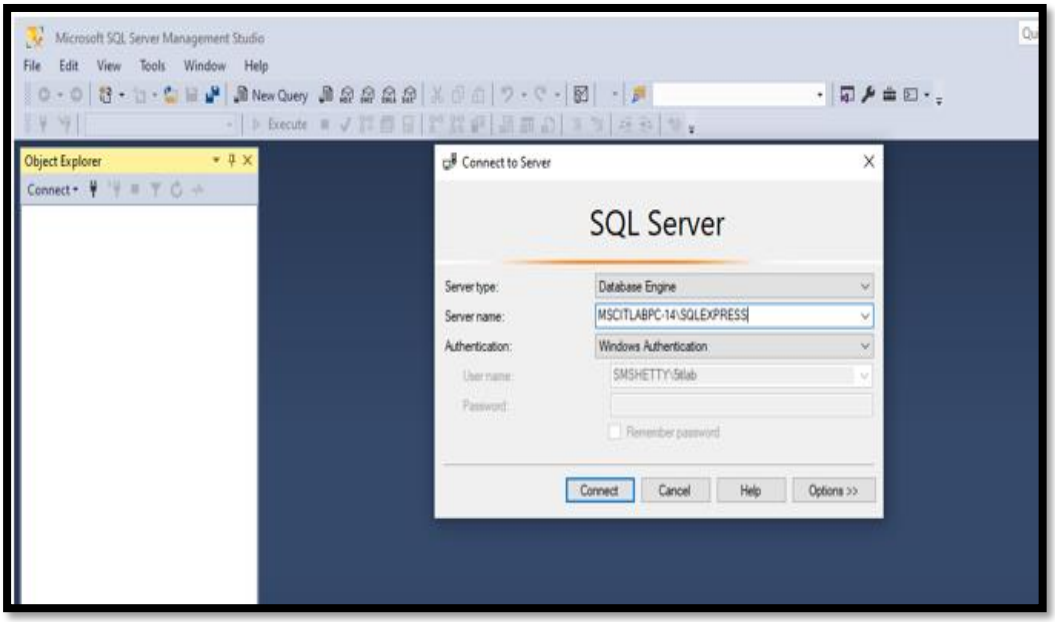

Copy the Copy the "**DEVICE NAME**" from the settings.




Once SQL Server Management Studio start It will show following window,

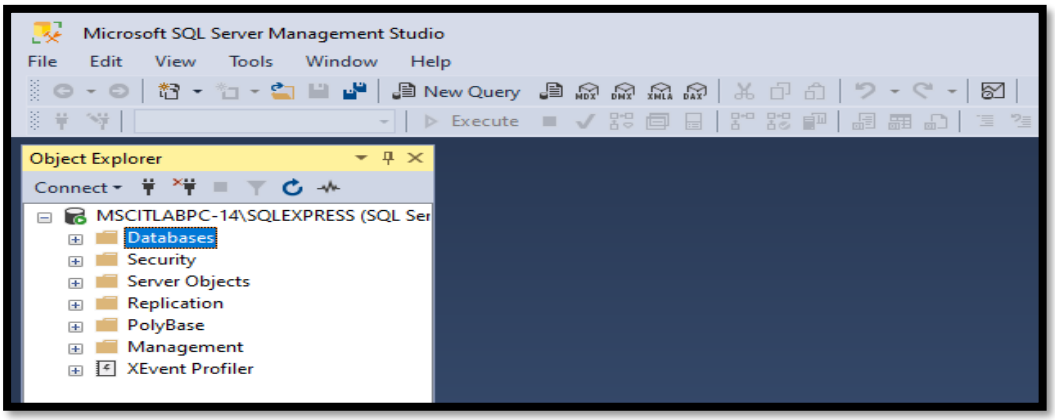
under server name enter copied **device name** along with **\SQLEXPRESS,** in my case its **mscitlabpc-14\SQLEXPRESS**

Then click on connect.




It will show following things, right click on **database** and select **new database** and give database name,

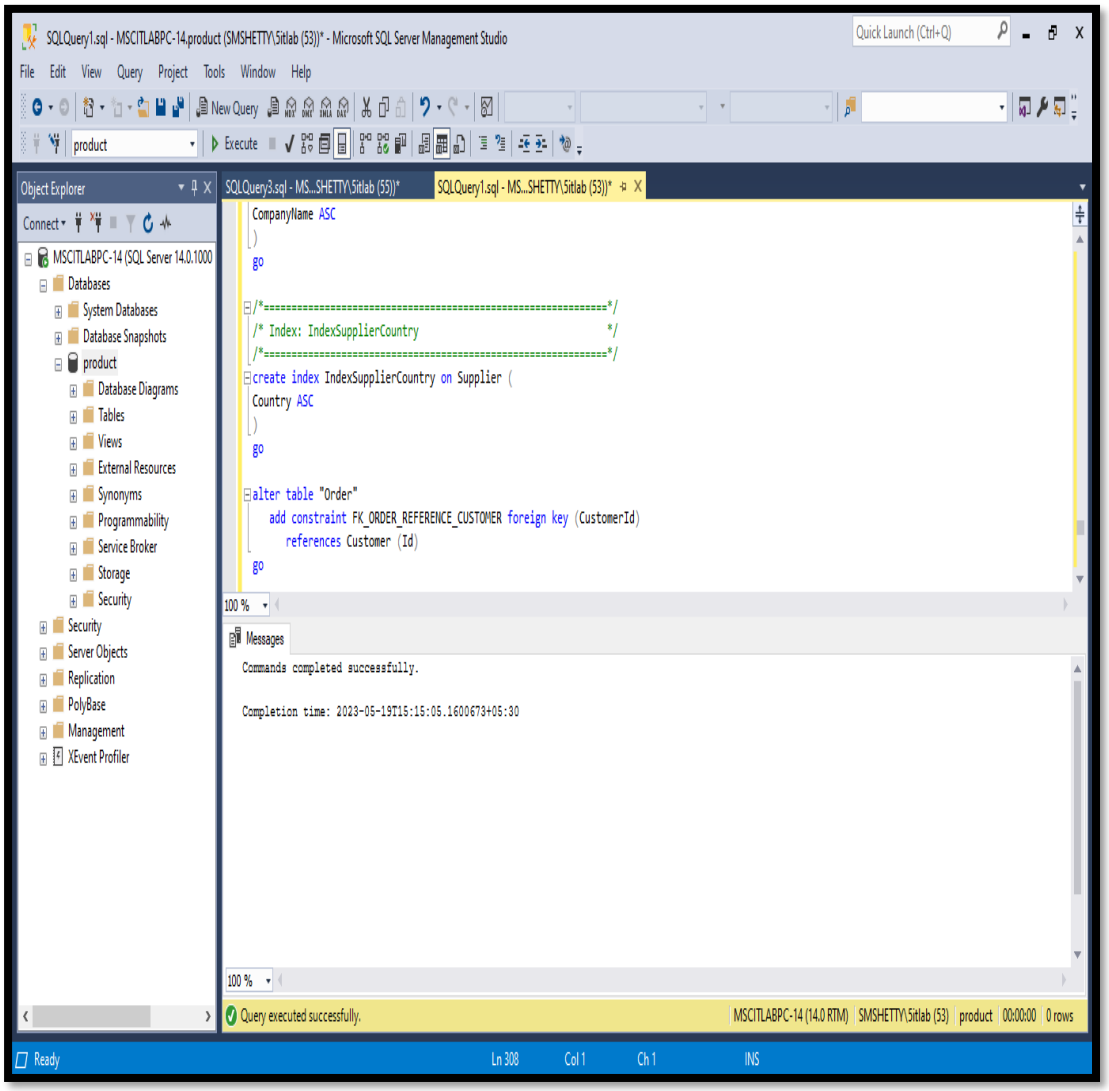
in my case database name is **product**



After creating database product**,** right click on **product** then select **add new query** and copy paste the

**sample-model.sql** file. Then click on **excute**

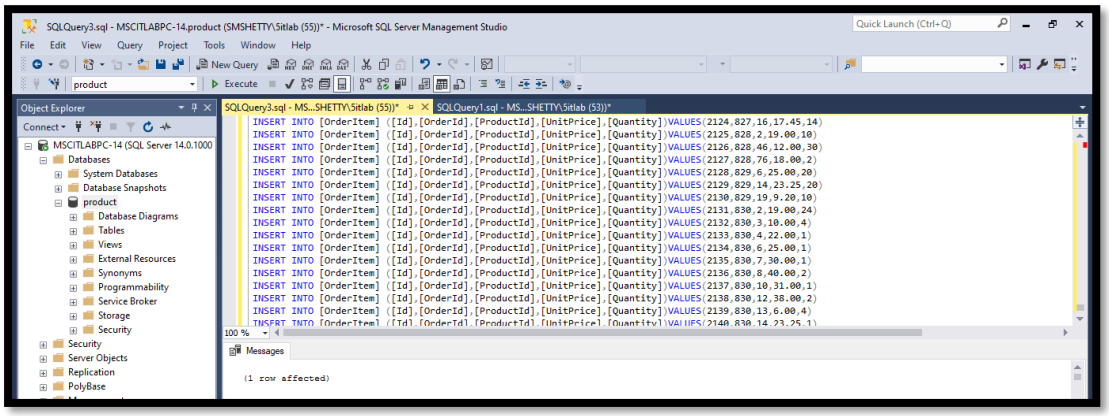
**sample-model.sql → file is given in college pc**




Again, right click on **product** then select **add new query** and copy paste the

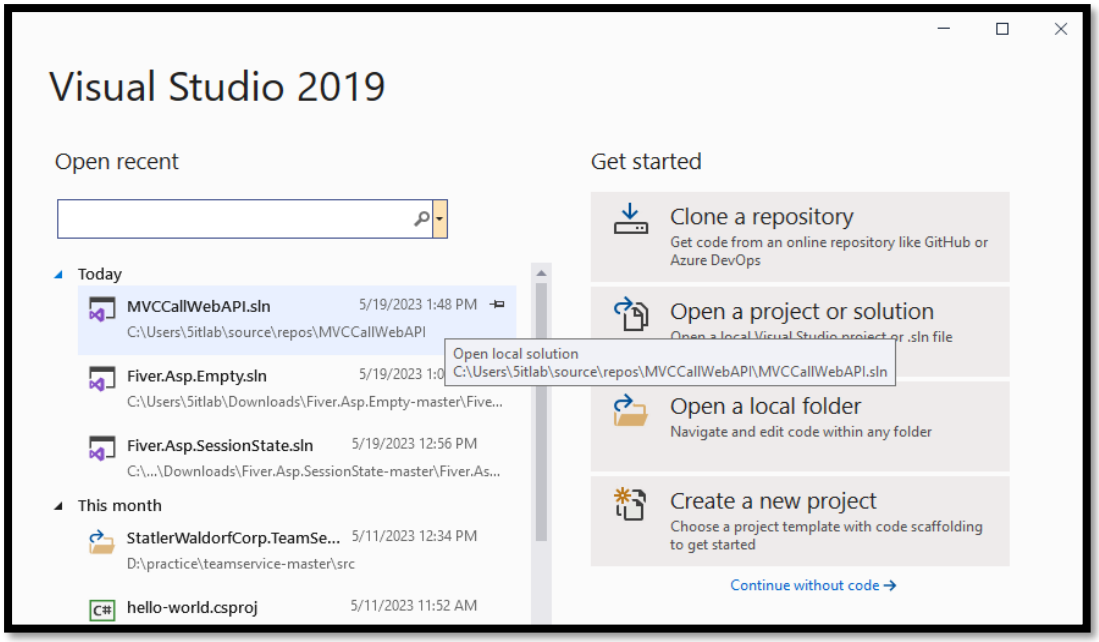
**sample-data.sql** file. Then click on **excute.**

**sample-data.sql → file is given in college pc**



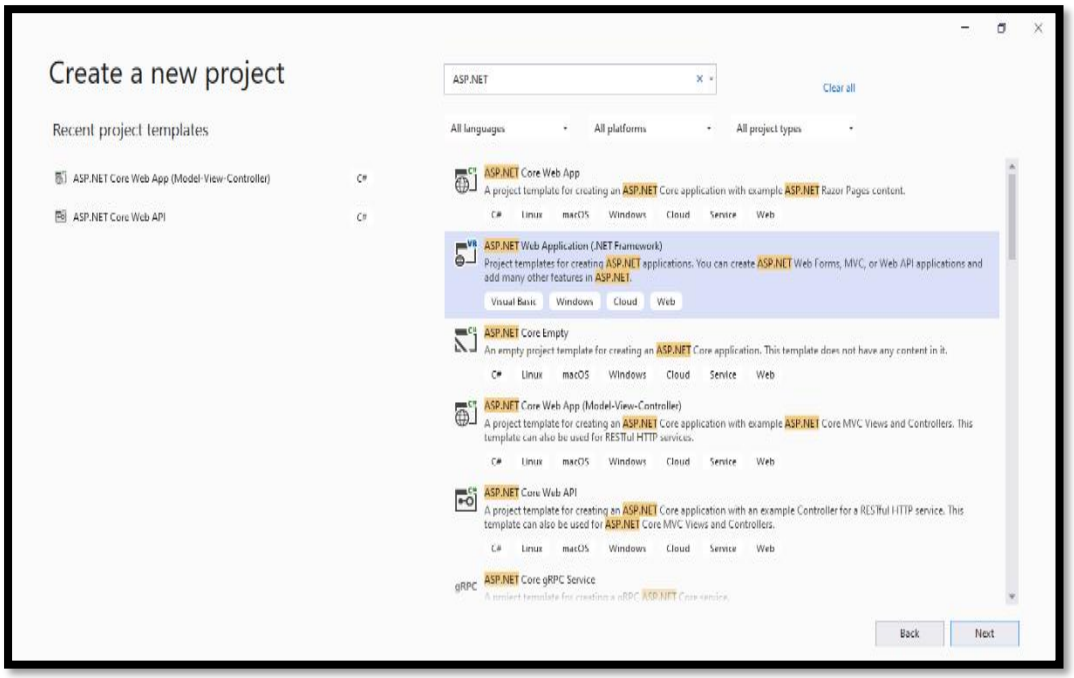

## Step 2: Create an ASP.NET MVC application

Now open Visual Studio and click on create a new project.

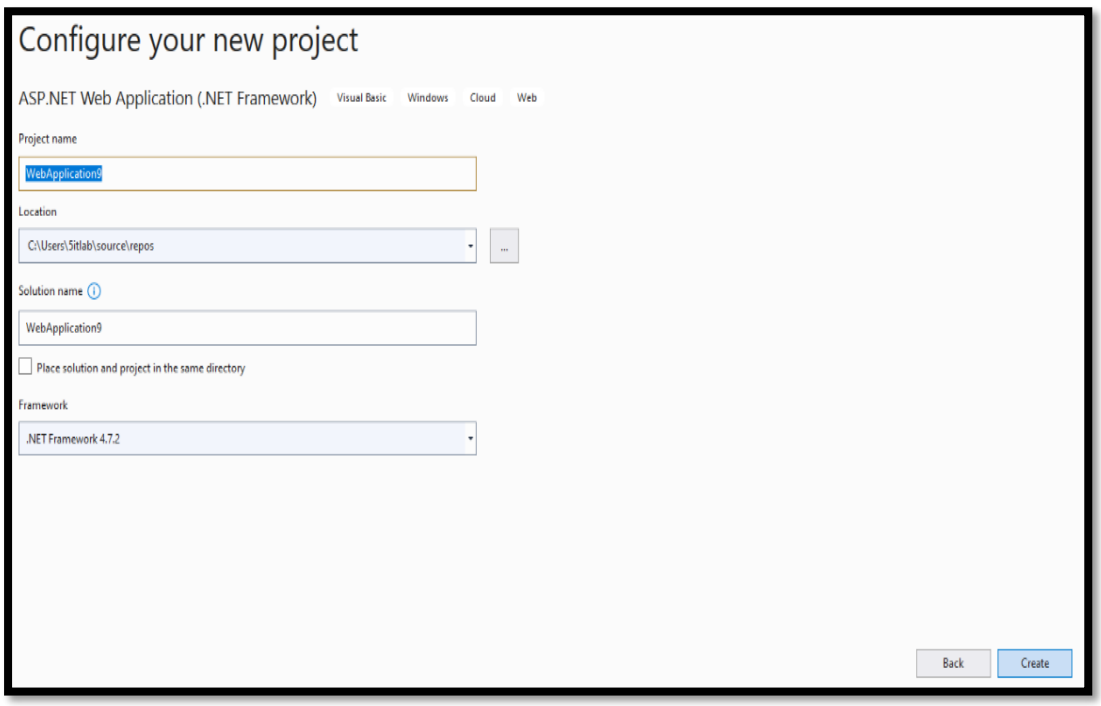



It will show following window, now select ASP.NET Web Application (.NET Framwork) then click on next.

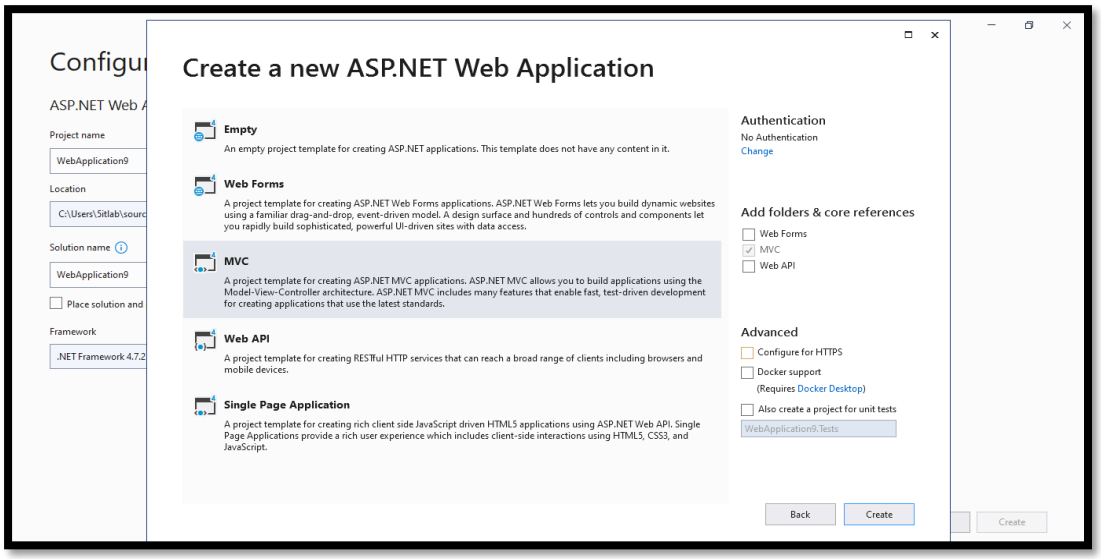
Note → U should not select core application.



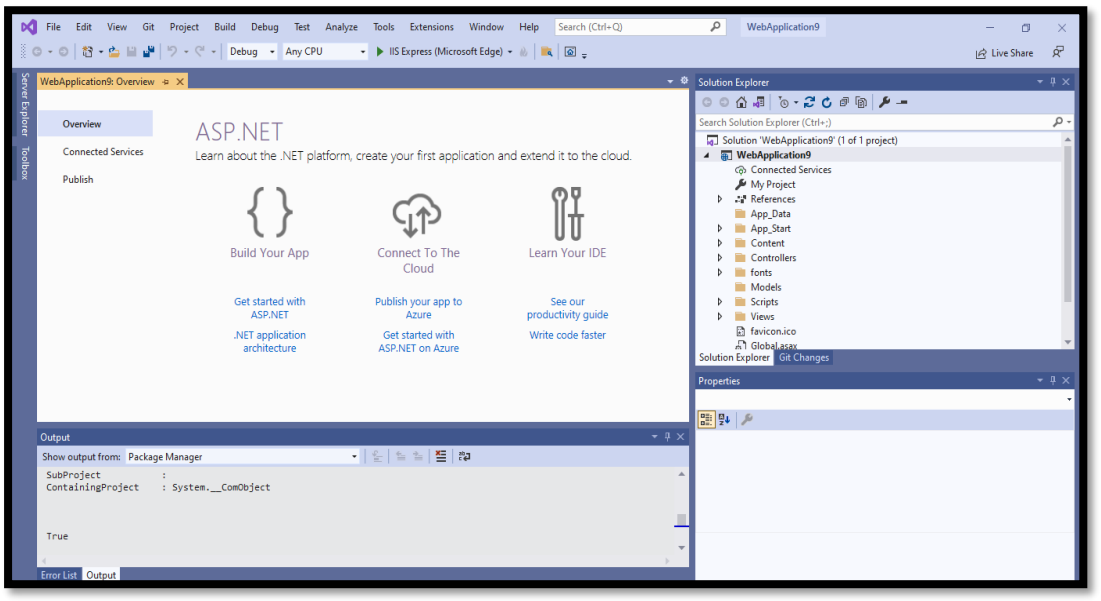

Now give project name and click on create.



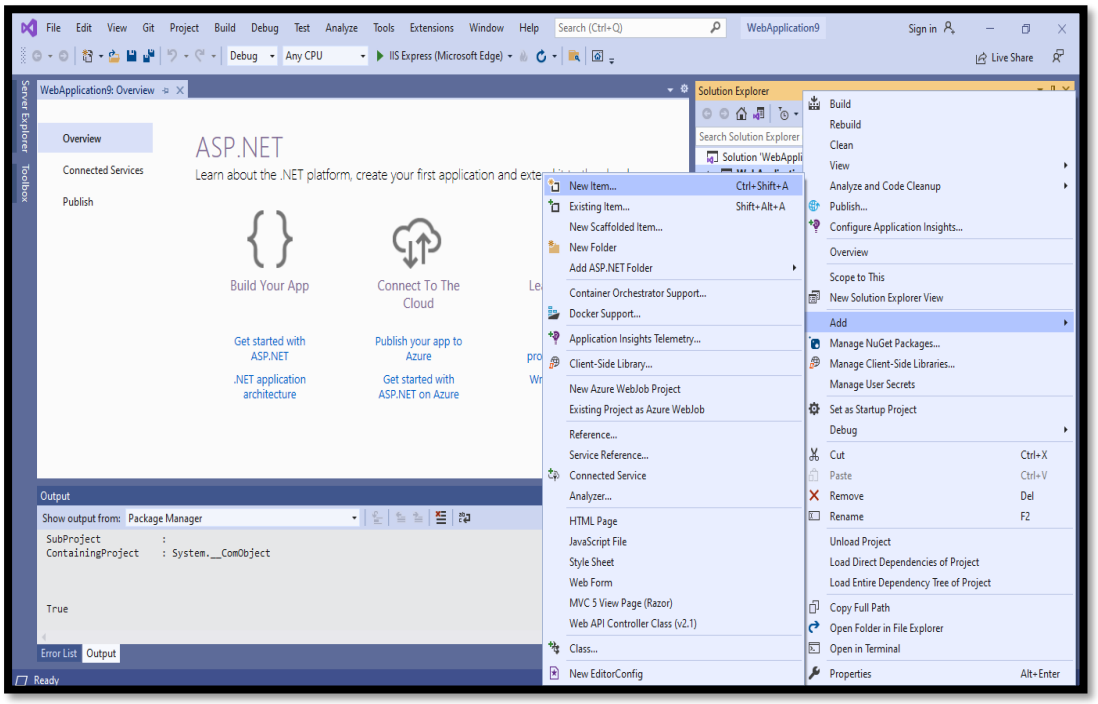
Here select **MVC** and Uncheck configure for https then click on create.



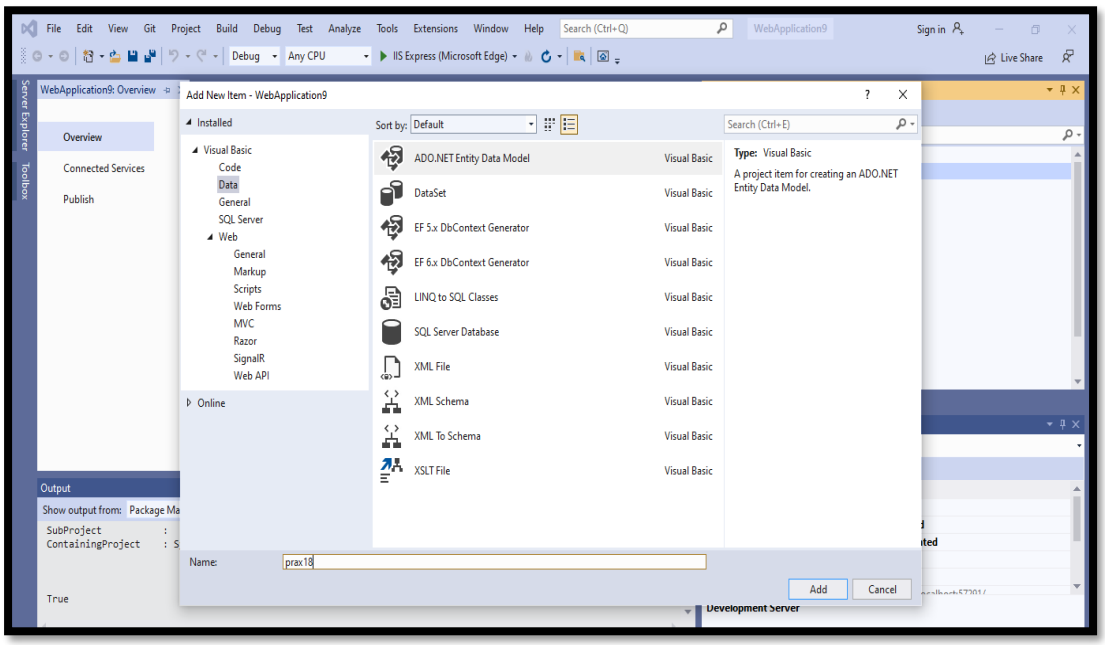

It will show following project in visual studio,



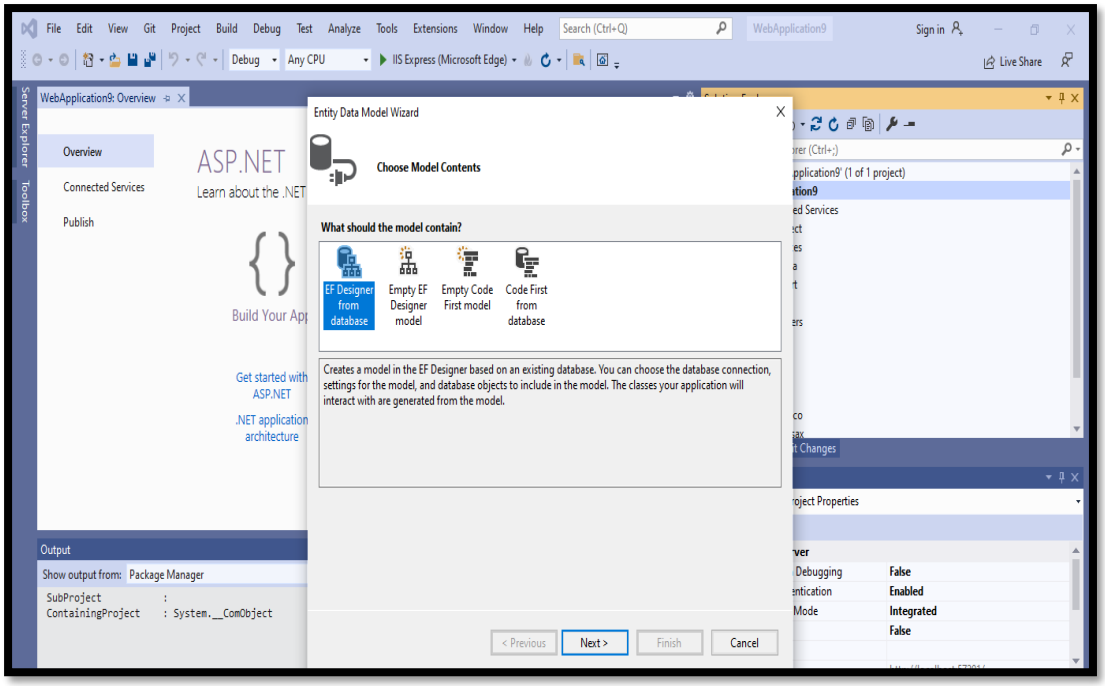

Right click on project **“webapplication9”** select **Add** then select **New item,**



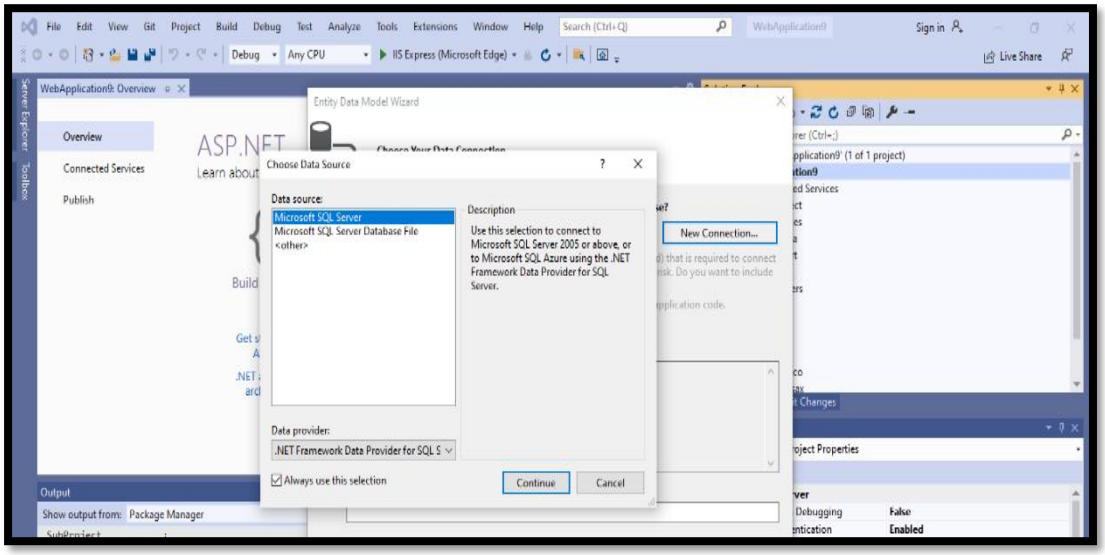

Select **Data** then select **ADO.NET Entity Data Model** then give name and click on Add,



It will show window **"Entity Data Model Wizard"** click on next after selecting first option named EF design for database




Click on new connection select Microsoft Sql Server then click continue



After clicking on continue you will notice following window,

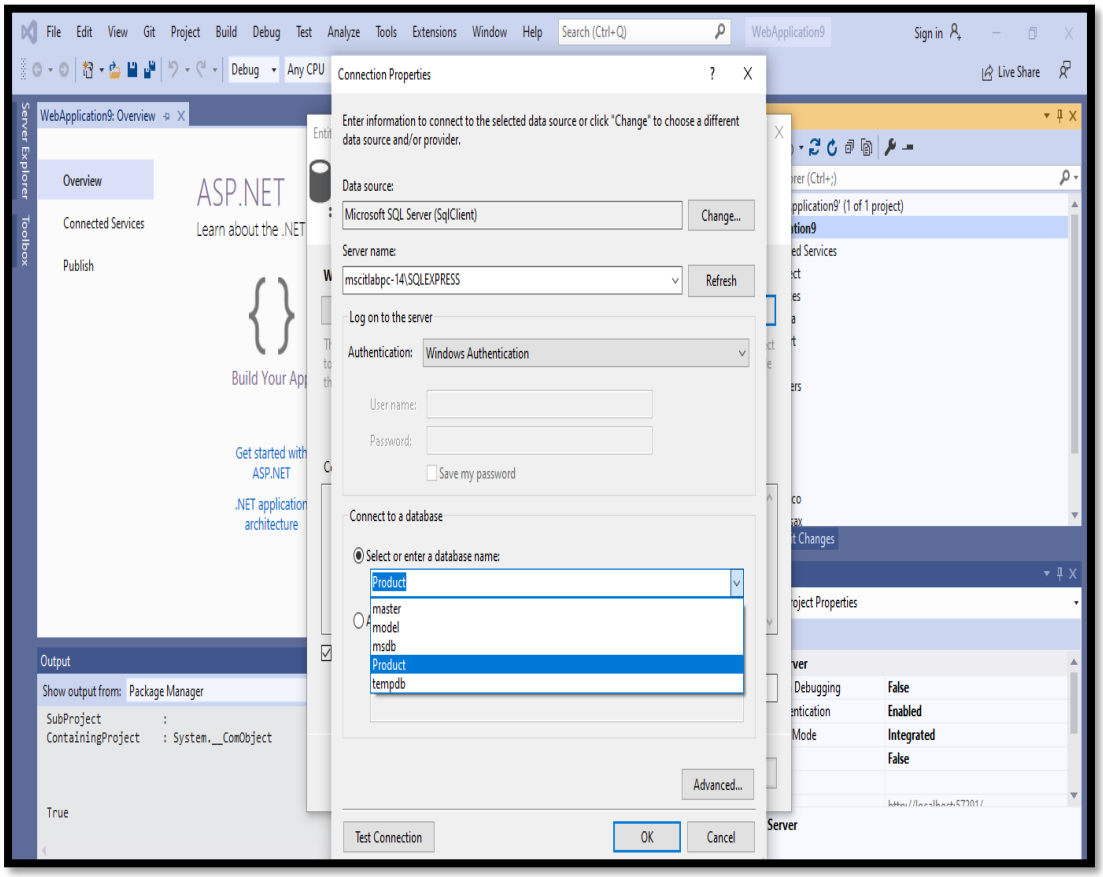
give **server name** of your pc,

in my case I have copied from device name.

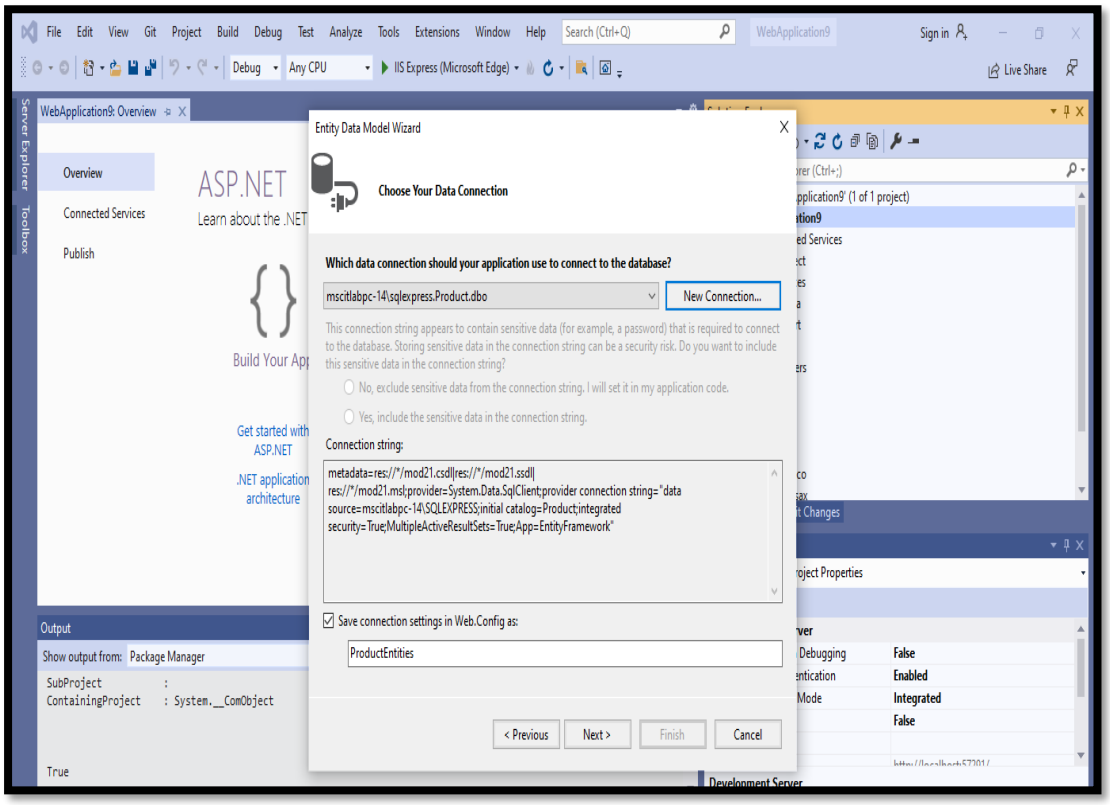
then under **connect to a database** select the database name you have created,

then click on **ok.**

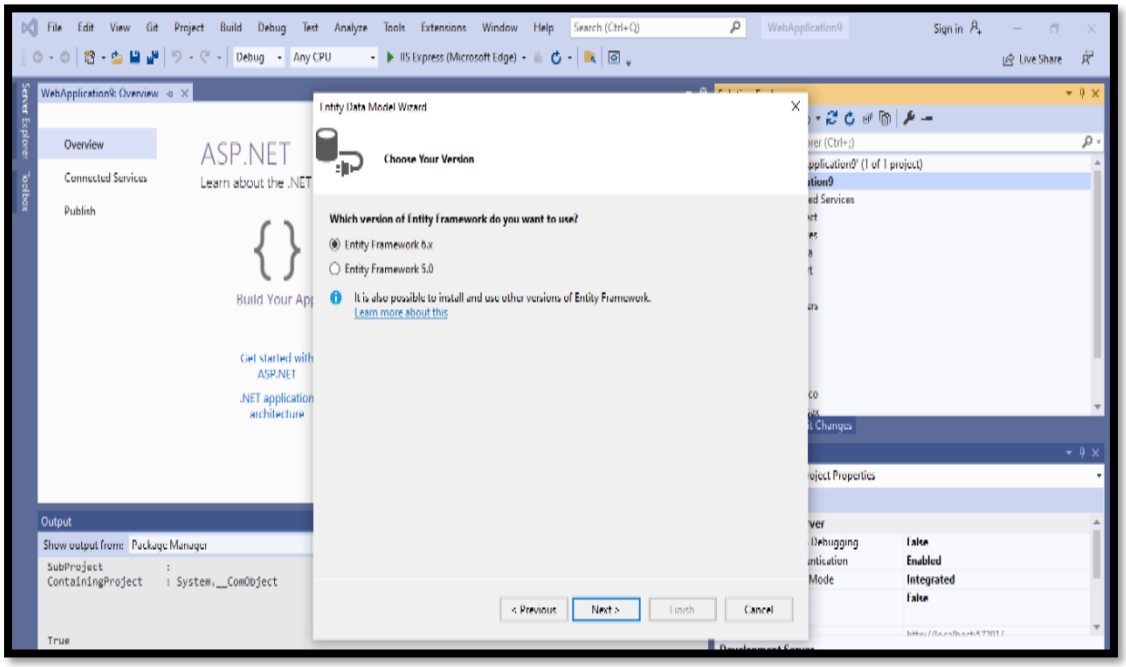
In my case I have created database product.



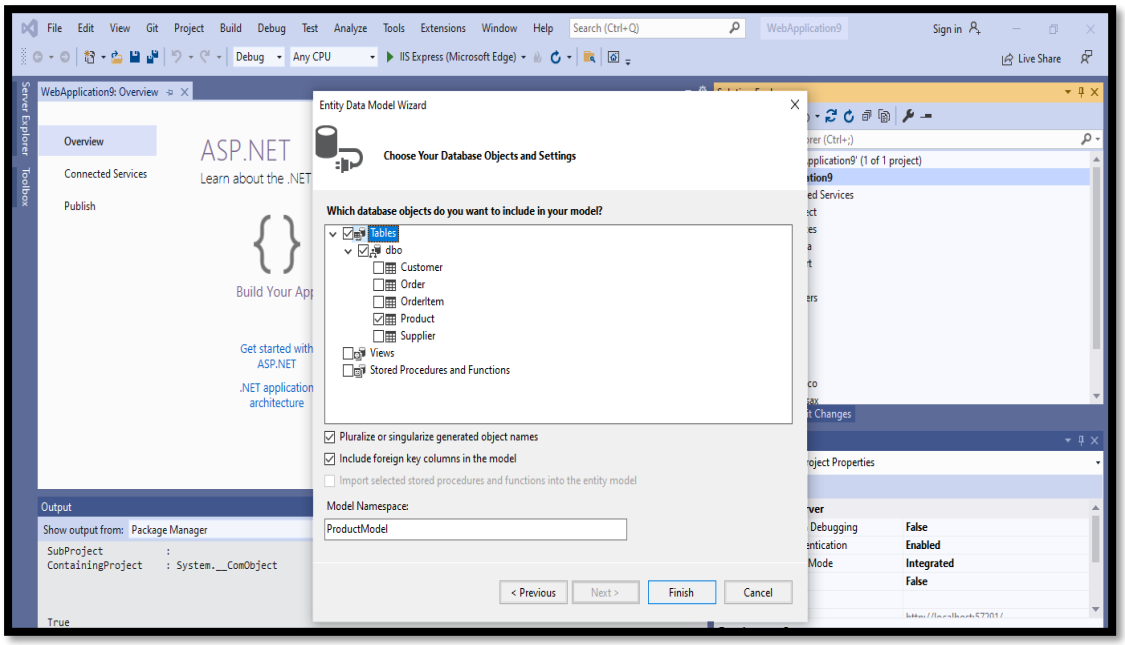

Now click on next.



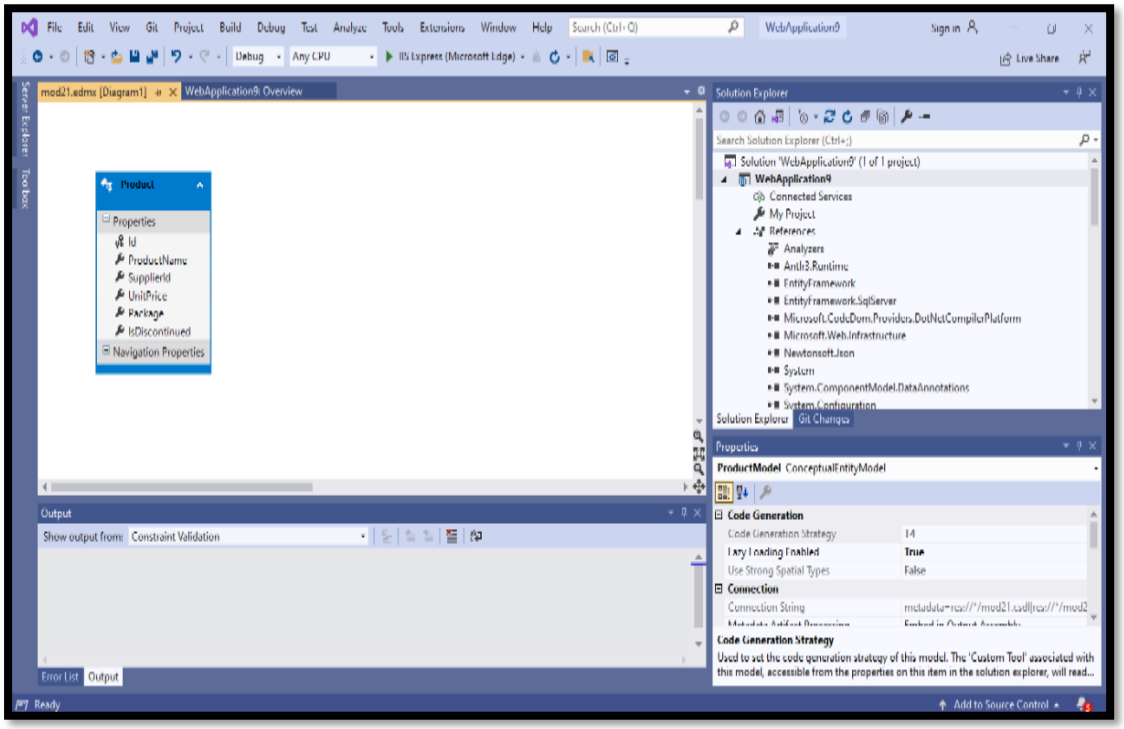

Select Entity Framework 6x then click on next.



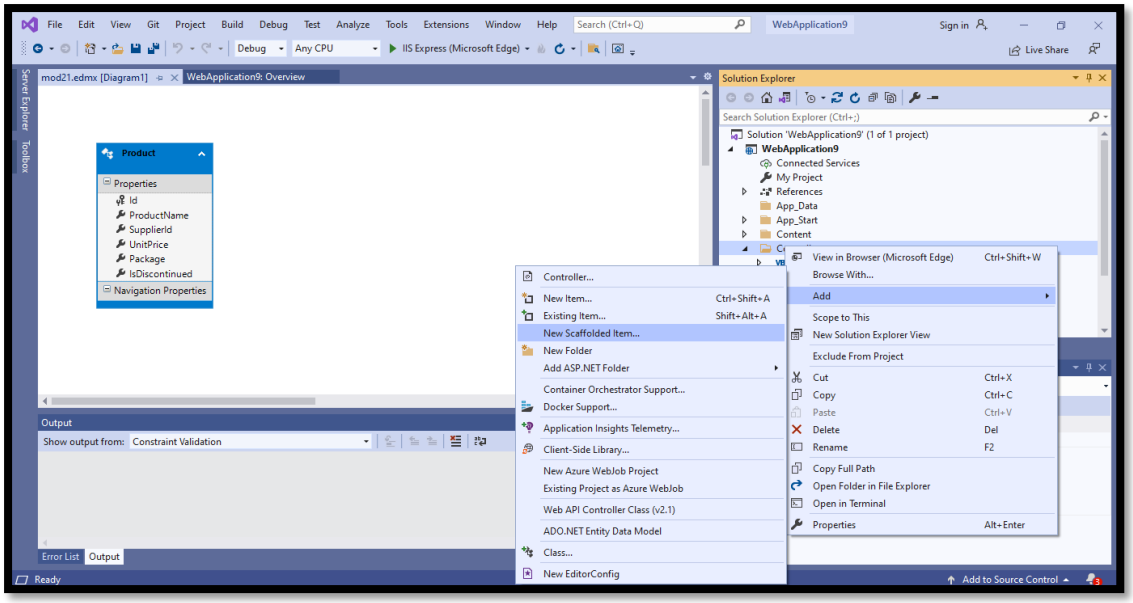
Now select the object that you have to include in your model, I have selected product



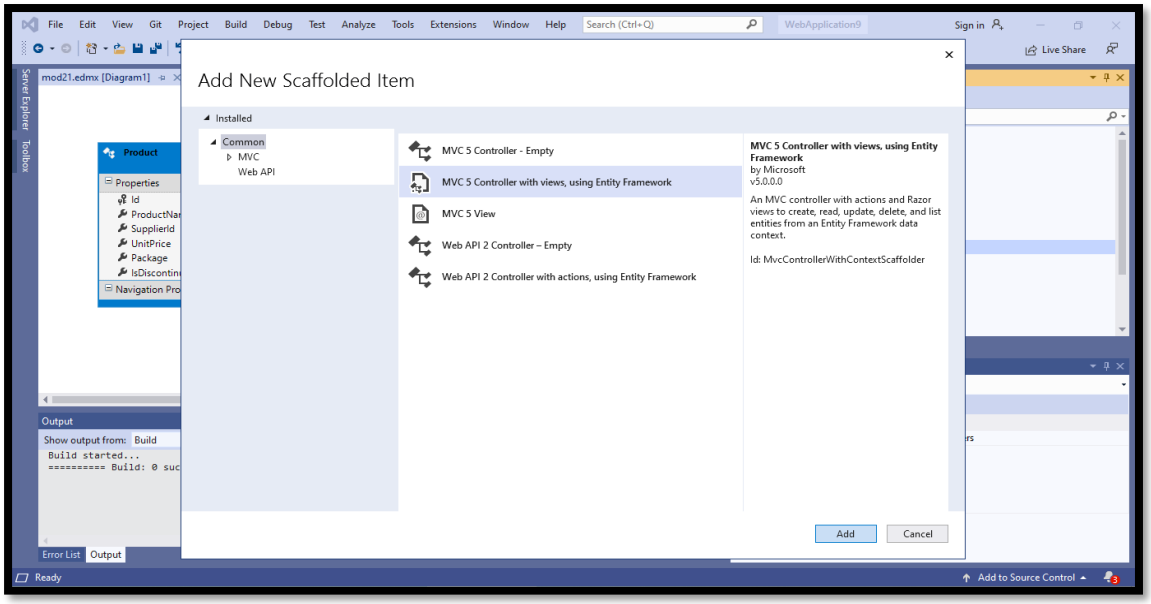

It will show following model,

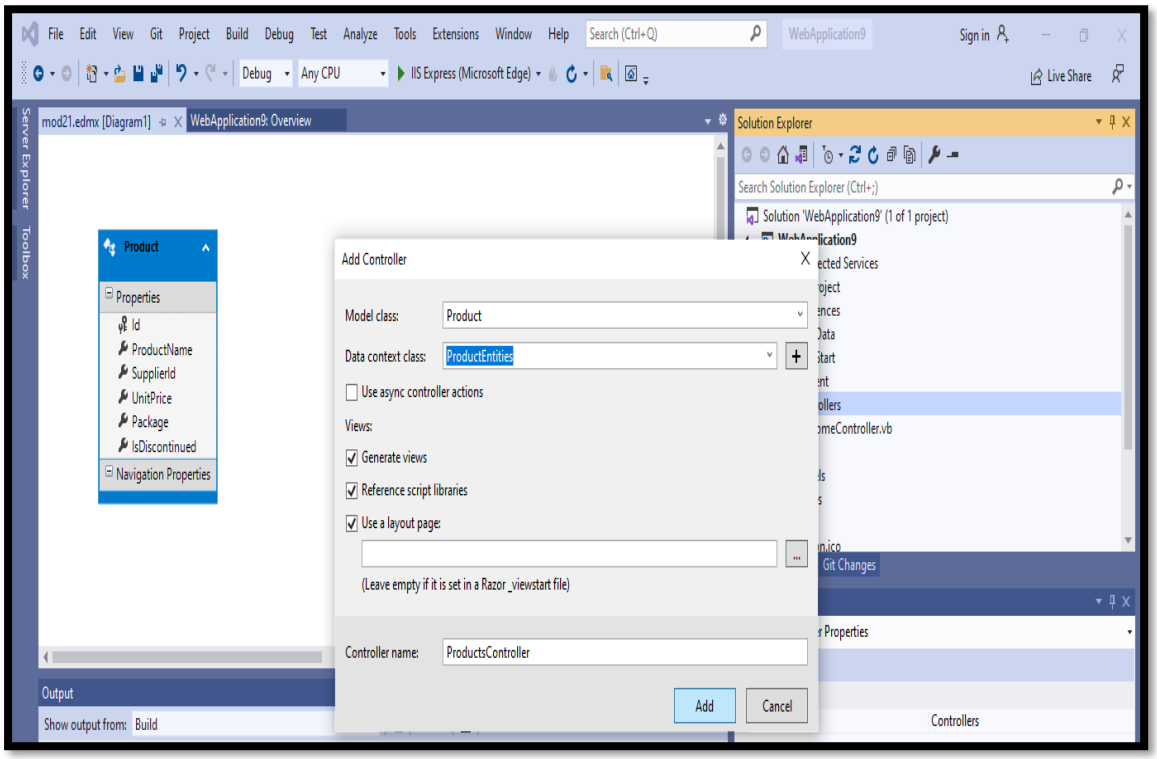


Now right click on **Controller** in solution explorer & select **Add** then **New Scaffolded Item.**

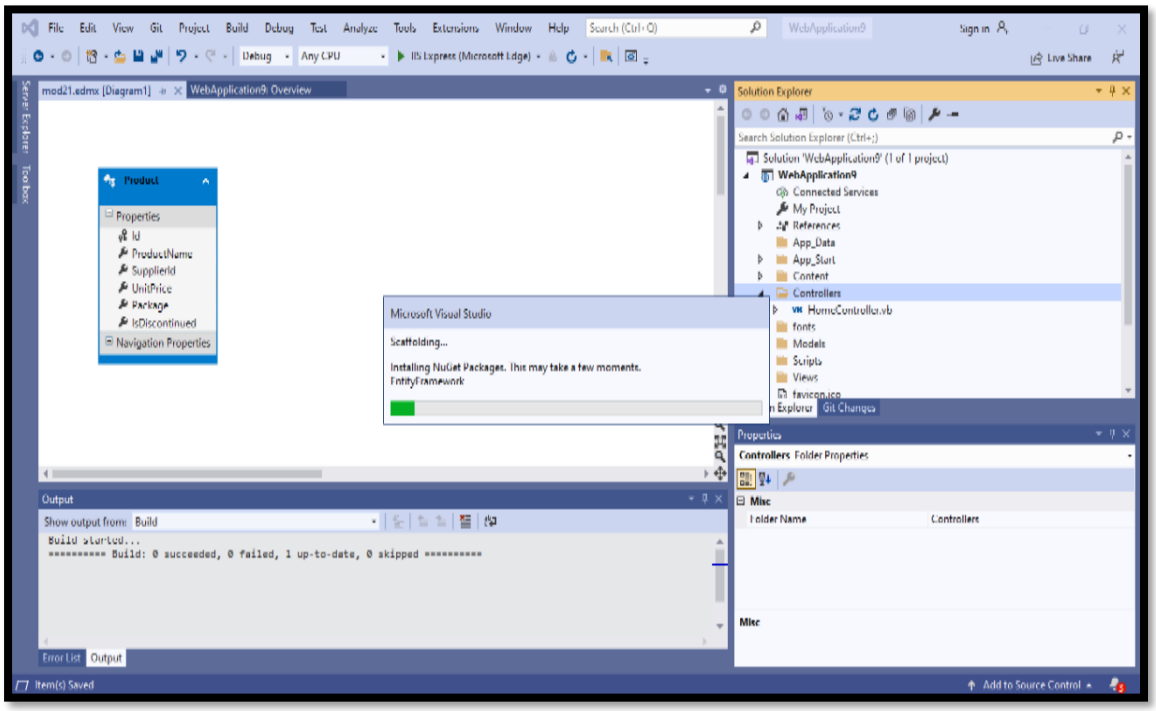



Then in common select MVC 5 Controller With Views,Using Entity Framework & click on Add





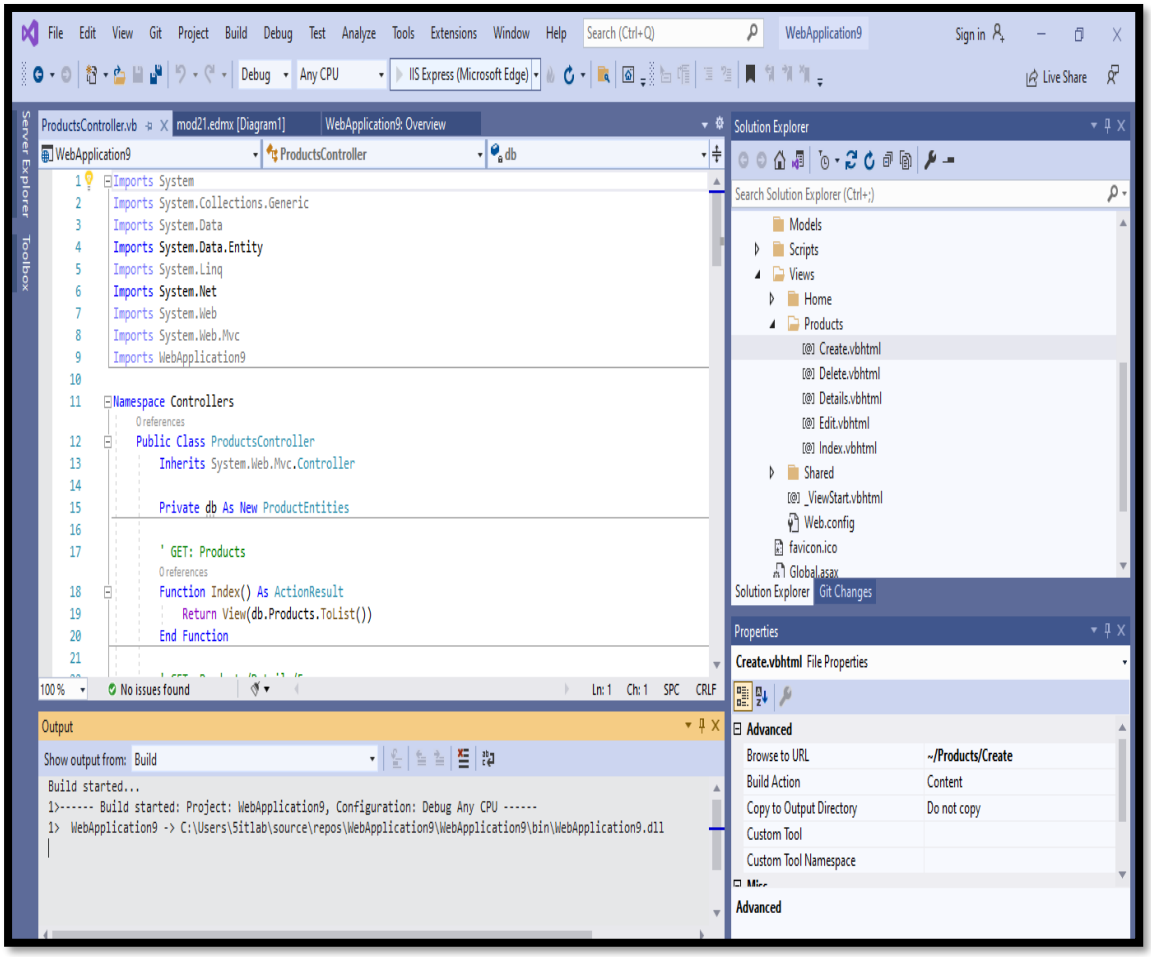

After clicking on Add, it will show following window



Next is build the project by clicking on build solution &

then run the project by clicking on **IIS Express.**

it will open browser for the output.

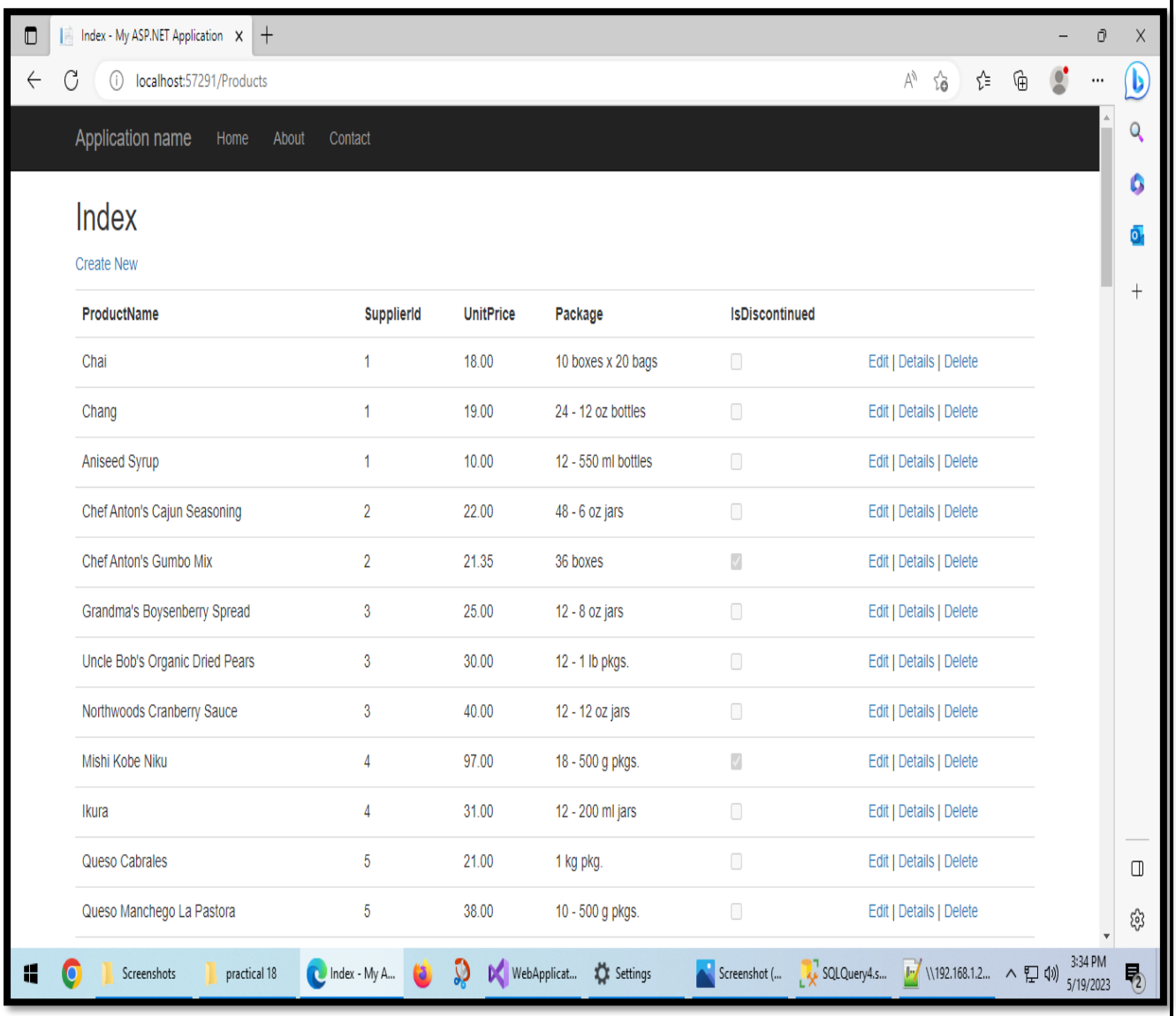



It will show following output of products.

Note: make sure you added /products after localhost : followed by port number 57291

In my case its, localhost : 57291/products

To get following output.


Index - My ASP.NET Application
localhost:57291/Products
Application name
Home
About
Contact
Index
Create New
ProductName
SupplierId
UnitPrice
Package
IsDiscontinued
Chai 1 18.00 10 boxes x 20 bags Edit | Details | Delete
Chang 1 19.00 24 - 12 oz bottles Edit | Details | Delete
Aniseed Syrup 1 10.00 12 - 550 ml bottles Edit | Details | Delete
Chef Anton's Cajun Seasoning 2 22.00 48 - 6 oz jars Edit | Details | Delete
Chef Anton's Gumbo Mix 2 21.35 36 boxes Edit | Details | Delete
Grandma's Boysenberry Spread 3 25.00 12 - 8 oz jars Edit | Details | Delete
Uncle Bob's Organic Dried Pears 3 30.00 12 - 1 lb pkgs. Edit | Details | Delete
Northwoods Cranberry Sauce 3 40.00 12 - 12 oz jars Edit | Details | Delete
Mishi Kobe Niku 4 97.00 18 - 500 g pkgs. Edit | Details | Delete
Ikura 4 31.00 12 - 200 ml jars Edit | Details | Delete
Queso Cabrales 5 21.00 1 kg pkg. Edit | Details | Delete
Queso Manchego La Pastora 5 38.00 10 - 500 g pkgs. Edit | Details | Delete
Screenshots
practical 18
Index - My A...
WebApplicat...
Settings
Screenshot (...
SQLQuery4.s...
\\192.168.1.2...
3:34 PM
5/19/2023