



TRAINING @ SILVER TOUCH TECHNOLOGIES LTD.

A LEADER IN SYSTEMS INTEGRATION & DIGITAL
TRANSFORMATION

SINCE 1995



API DEVELOPMENT USING ASP.NET CORE WEB API

API Development Using Asp.Net Core Web API

**A practical approach for developing the
APIS in asp.net core**

PREFACE

API Development Using Asp.Net Core Web API in this book we adopted a practical approach for the development of software products. This book is divided into four parts every part has its dimension. This book will explain the fully featured projects and how we can architect the application using the following architectures.

- What is Restful Web Service
- Introduction to API Development
- API Development Using Asp.net Core
- API Development Using Onion Architecture.
- API Testing Using Swagger
- API Testing Using Postman
- API Deployment on Azure Cloud
- API Deployment on Hosting Server

PART 1: API Development Using Asp.net Core

- 1) Introduction
- 2) Why Restful Web Service
- 3) Implementation
- 4) Conclusion
- 5) Complete Project GitHub URL

PART 2: API Development Using Three Tier Architecture.

- 1) Introduction
- 2) Complete Understanding of Three Tier Architecture
- 3) Practical Approach for Architecting the Application with Asp.net Core
- 4) Conclusion
- 5) Complete Project GitHub URL

PART 3: API Development Using Onion Architecture.

- 1) Introduction
- 2) Complete Understanding of Onion Architecture
- 3) Practical Approach for Architecting the Application with Asp.net Core
- 4) Conclusion
- 5) Complete Project GitHub URL

PART 4: API Testing Using Swagger.

- 1) Introduction
- 2) API Testing Using Swagger in Asp.net Core
- 3) Test POST, PUT, DELETE, GET, UPDATE Endpoints Using Swagger
- 4) Conclusion

PART 4: API Testing Using Postman.

- 1) Introduction
- 2) API Testing Using Postman in Asp.net Core
- 3) Test POST, PUT, DELETE, GET, UPDATE Endpoints Using Postman
- 4) Conclusion

PART 5: Application Deployment on Azure Cloud.

- 1) Introduction
- 2) Creation of App on Azure Cloud
- 3) Server Configuration in Visual Studio
- 4) Deployment Steps Using Visual Studio
- 5) Output on Azure Websites
- 6) Conclusion

PART 6: Application Deployment on Server.

- 1) Introduction
- 2) Deployment Procedure
- 3) Server Configuration
- 4) Deployment Steps
- 5) Conclusion

Contents

<u>Chapter 1 – What is REST Architecture</u>	<u>18</u>
<u>What is RESTful Web Service?</u>	<u>19</u>
<u>RESTful Web Service Features</u>	<u>19</u>
<u>Resources</u>	<u>19</u>
<u>Request Verbs</u>	<u>7</u>
<u>Request Header</u>	<u>8</u>
<u>Request Body</u>	<u>8</u>
<u>Response Body.....</u>	<u>8</u>
<u>Response After Positing the Data into Database</u>	<u>9</u>
<u>Response Status Codes</u>	<u>9</u>
<u>RESTful Web Service Methods.....</u>	<u>9</u>
<u>Why RESTful Web Service</u>	<u>12</u>
<u>RESTful Architecture</u>	<u>12</u>
<u>RESTful Principals and Constraints</u>	<u>13</u>
<u>RESTful Client-Server</u>	<u>13</u>
<u>Stateless</u>	<u>13</u>
<u>Cache.....</u>	<u>14</u>
<u>Layered System.....</u>	<u>14</u>
<u>RESTful Web Services Interfaces.....</u>	<u>14</u>
<u>Create your first RESTful Web service in Asp.net Core Web API.....</u>	<u>13</u>
<u>Data Access Layer</u>	<u>15</u>
<u>RESTful Web Service Results.....</u>	<u>35</u>
<u>Debug the project using visual studio Debugger</u>	<u>35</u>
<u>POST Endpoint in Web API Service</u>	<u>36</u>
<u>DELETE Endpoint in Web API Service.....</u>	<u>34</u>
<u>GET All Endpoint in Web API Service</u>	<u>35</u>
<u>GET End Point for Getting the Record by User Email</u>	<u>35</u>
<u>Delete End Point for Deleting the Record by User Email</u>	<u>36</u>
<u>Update End Point for Updating the Record by User Email.....</u>	<u>37</u>
<u>Testing the RESTful Web Service</u>	<u>37</u>
<u>Step 1 Run the Project.....</u>	<u>37</u>

<u>Step 2 Test POST End Point.....</u>	38
<u>Step 3 Test GET End Point.....</u>	39
<u>Step 4 Test <i>DELETE</i> End Point.....</u>	40
<u>Step 5 Test GET by User Email End Point.....</u>	41
<u>Conclusion.....</u>	42
<u>Chapter 2 – API Development In Asp.Net Using Three Tier Architecture</u>	43
<u>Introduction</u>	44
<u>Project Structure.....</u>	44
<u>Presentation Layer (PL)</u>	44
<u>Business Access Layer (BAL)</u>	44
<u>Data Access Layer (DAL)</u>	44
<u>Add the References to the Project.....</u>	47
<u>Data Access Layer</u>	50
<u>Contacts.....</u>	50
<u>Data</u>	52
<u>Migrations</u>	52
<u>Models.....</u>	52
<u>Repository</u>	54
<u>Business Access Layer</u>	57
<u>Services</u>	58
<u>Presentation layer.....</u>	60
<u>Advantages of 3-Tier Architecture.....</u>	61
<u>Dis-Advantages of 3-Tier Architecture.....</u>	61
<u>Conclusion.....</u>	61
<u>Complete Git Hub Project URL.....</u>	62
<u>Chapter 3- API Development In Asp.Net Using Onion Architecture</u>	63
<u>Introduction</u>	64
<u>What is Onion architecture.....</u>	64
<u>Layers in Onion Architecture</u>	64
<u>Domain Layer</u>	65
<u>Repository Layer</u>	65

<u>Service Layer</u>	65
<u>Presentation Layer</u>	65
<u>Advantages of Onion Architecture</u>	65
<u>Implementation of Onion Architecture</u>	66
<u>Project Structure.....</u>	66
<u>Domain Layer.....</u>	66
<u>Models Folder</u>	68
<u>Data Folder.....</u>	71
<u>Repository Layer</u>	73
<u>Service Layer</u>	77
<u>ICustom Service</u>	79
<u>Custom Service</u>	79
<u>Department Service.....</u>	79
<u>Student Service</u>	83
<u>Result Service</u>	86
<u>Subject GPA Service.....</u>	89
<u>Presentation Layer</u>	92
<u>Dependency Injection</u>	93
<u>Modify Program.cs File</u>	93
<u>Controllers.....</u>	94
<u>Output</u>	98
<u>Conclusion.....</u>	99
<u>Complete GitHub project URL.....</u>	99
<u>Chapter 4- API Testing Using Postman</u>	100
<u>Introduction</u>	101
<u>How to Test API Web Service in Asp Net Core 5 Using Postman</u>	101
<u>Test Post Call in Postman.....</u>	105
<u>Test Delete Call in Postman</u>	107
<u>Test Get Call in Postman</u>	108
<u>Test Put Call in Postman</u>	109
<u>Conclusion.....</u>	111

Chapter No 5 API Testing Using Swagger.....	112
Introduction	113
How to Test API Web Service in Asp Net Core 5 Using Swagger	113
Test POST Call in Swagger	114
Test DELETE Call in Swagger	116
Test GET Call in Swagger.....	118
Test PUT Call in Swagger.....	119
Summary	122
Conclusion.....	123
Chapter 6- Application Deployment on Azure Cloud.....	124
Introduction	125
Open Your Azure portal	125
Create Azure Web App	125
Create the Azure Resource Group.....	126
Select the Name of Your Application.....	126
Select the Runtime Stack	127
Select the Azure Region	127
Create App Service Plan.....	128
Select the pricing plan.....	128
Select the Recommended Pricing Tier.....	129
Select the Deployment Option.	129
Select the Net Working Options	130
Select the Azure Monitoring Options	131
Select the Tags Options.....	132
Review and Create	132
Application Status	135
Get the publish profile.....	136
Publish the App Using Visual Studio	137
Select the Publish Profile	139
Deployment Status	141
Application Deployed on Azure Cloud	141
Chapter 7- Application Deployment on Server.	142
Introduction	143

Step 1 Account Setup on Server	143
Step 2 Create the project asp.net Core 5 Web API using VS-2019.....	143
Step 3 Set the Live Server Database Connection string in your application appsetting.json file	144
Step 4 Publish Your Project.....	144
Step 5 Validate the Connection	150
Conclusion.....	154
References	155
Feedback	156

Chapter 1 – What is REST Architecture

- ✓ What is RESTful Web Service?
- ✓ RESTful Web Service Features
- ✓ RESTful Web Service Methods
- ✓ Why RESTful Web Service
- ✓ RESTful Architecture
- ✓ RESTful Principles and Constraints
- ✓ RESTful Client-Server
- ✓ Layered System
- ✓ RESTful Web Services Interfaces
- ✓ Create your first RESTful Web service in Asp.net Core Web API
- ✓ Data Access Layer
- ✓ Business Access Layer
- ✓ Presentation Layer

What is RESTful Web Service?

Restful web services are a lightweight maintainable and scalable service that is built on the REST Architecture. Restful Web service exposes API from your application in a secure uniform stateless manner to the calling client can perform predefined operations using the restful service the protocols for the REST is HTTP and Rest is a Representational State Transfer. Representational state transfer (REST) is a de-facto standard for a software Architecture or Interactive applications that use Web Service a Web Service that follows this standard is called RESTful such web must provide its web resources in a textual representation and allow them to be read and modified with stateless protocol and predefined set of operations this standard allow interoperability between a Machine and the Internet that provide these services. REST Architecture is alternative to SOAP and SOAP Architecture is the way to access a Web Service.

RESTful Web Service Features

Rest Web services have come a long way since their inception. In 2002 the Web Consortium had released the definition of WSDL and SOAP Web services and this formed the standard of how web services are implemented in 2004 the web consortium also released the definition of an additional standard called RESTful in past four to five years this standard has become more popular and being used in many giants like Facebook and Twitter Microsoft. REST is a way to access resources that lie in an articular environment you could have a server that could be hosting important documents or pictures or videos. All these are an example of the resources if a client says a web browser needs any of these resources it must send a request to the server to access the resources.

The key features of the RESTful implementation are as follows:

Resources

The first key feature is the resource itself. Let us assume that a web application on the server has a record of Persons if we want to access the records of the Persons by Id the details are in the given below figure the coding part is also given.

The screenshot shows the Swagger UI interface for a .NET Core Web API. The main title is "PersonDetails". Below it, there are five API endpoints listed under different request verbs:

- POST** /api/PersonDetails/AddPerson
- DELETE** /api/PersonDetails/DeletePerson
- PUT** /api/PersonDetails/UpdatePerson
- GET** /api/PersonDetails/GetAllPersonByName
- GET** /api/PersonDetails/GetAllPersons

Below the endpoints, there is a section titled "Schemas" which contains a single entry: "Person".

Request Verbs

This describes what you want to do with the resources a browser issues a **GET** verb to instruct the endpoints it wants to get the data like **POST** is used to send the data by hitting the endpoints in the postman or swagger and **PUT** is used to update the data and **DELETE** is used to either soft delete the record or on permanent bases.

As shown in the below figure.

The screenshot shows the same Swagger UI interface for the "PersonDetails" API endpoint. The list of request verbs and their corresponding URLs is identical to the one in the first screenshot:

- POST** /api/PersonDetails/AddPerson
- DELETE** /api/PersonDetails/DeletePerson
- PUT** /api/PersonDetails/UpdatePerson
- GET** /api/PersonDetails/GetAllPersonByName
- GET** /api/PersonDetails/GetAllPersons

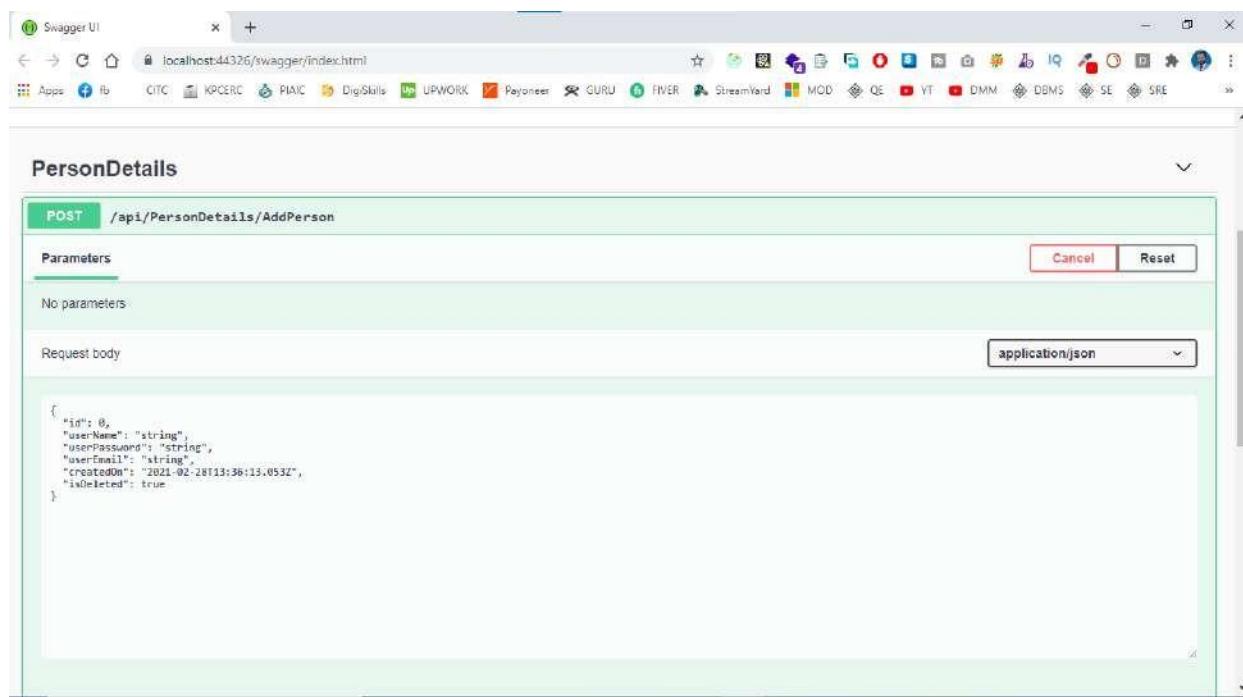
The "Schemas" section also remains the same, showing the "Person" schema.

Request Header

These are the additional instructions sent with the request these might define the types of the response required or authorization details

Request Body

Data is sent with the requested data is normally sent in the request when a POST request is made to the REST web service in a POST call the client tells the rest web Service that it wants to add a resource to the server hence the request body would have the details of the resources which is required to be added to the server. As shown in the figure below.



Response Body

This is the main body of the response so in our RESTful API Example if we were to query the web service via the GET Request, we server returns the response in the JSON format which gives us the details of Persons according to the Person ID.

For Posting the Data

The screenshot shows the Swagger UI interface for a POST request to the endpoint `/api/PersonDetails/AddPerson`. The request body is defined as follows:

```
{
    "id": 0,
    "userName": "AsadAliKhan",
    "userPassword": "Pak2019@",
    "userEmail": "AsadAliKhan@gmail.com",
    "createdOn": "2021-02-28T13:56:22.281Z",
    "isDeleted": false
}
```

The Content-Type is set to `application/json`. Below the request, there are sections for Responses and Requests.

Responses

Curl

```
curl -X POST "https://localhost:44326/api/PersonDetails/AddPerson" -H "accept: */*" -H "Content-Type: application/json" -d "{\"id\":0,\"userName\":\"AsadAliKhan\",\"userPassword\":\"Pak2019@\"}"
```

Request URL

Response After Positing the Data into Database

The screenshot shows the Swagger UI interface displaying the response to the POST request. The status code is 200, and the response body is `true`. The response headers are as follows:

```
access-control-allow-methods: GET,PUT,POST,DELETE,HEAD,OPTIONS
access-control-allow-origin: https://localhost:44326
content-type: application/json; charset=utf-8
date: Sun, 28 Feb 2021 13:56:55 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
```

Responses

Swagger UI

localhost:44326/swagger/index.html

GET /api/PersonDetails/GetAllPersons

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X GET "https://localhost:44326/api/PersonDetails/GetAllPersons" -H "accept: */*
```

Request URL

```
https://localhost:44326/api/PersonDetails/GetAllPersons
```

Server response

Code Details

200

Swagger UI

localhost:44326/swagger/index.html

GET /api/PersonDetails/GetAllPersons

Responses

Curl

```
curl -X GET "https://localhost:44326/api/PersonDetails/GetAllPersons" -H "accept: */*
```

Request URL

```
https://localhost:44326/api/PersonDetails/GetAllPersons
```

Server response

Code Details

200 Response body

```
[{"Id": 1, "UserName": "AsadAliKhan", "UserPassword": "Pak2019@", "UserEmail": "AsadAli10han@gmail.com", "CreationDate": "2021-09-28T13:56:22.283", "IsDeleted": false}]
```

Download

Response headers

```
access-control-allow-methods: GET,PUT,POST,DELETE,HEAD,OPTIONS
access-control-allow-origin: https://localhost:44326
content-encoding: gzip
content-type: application/json; charset=utf-8
date: Sun, 28 Feb 2021 14:00:07 GMT
server: Microsoft-IIS/10.0
vary: Accept-Encoding
x-powered-by: ASP.NET
```

Responses

Response Status Codes

These codes are the general codes that are returned along with the response from the webserver the status code details are the given below table.

	Code	Response
1	100 - 199	Informational Response
2	200 - 299	Successful Response
3	300 - 399	Redirects
4	400 - 499	Client Error
5	500 - 599	Server Error

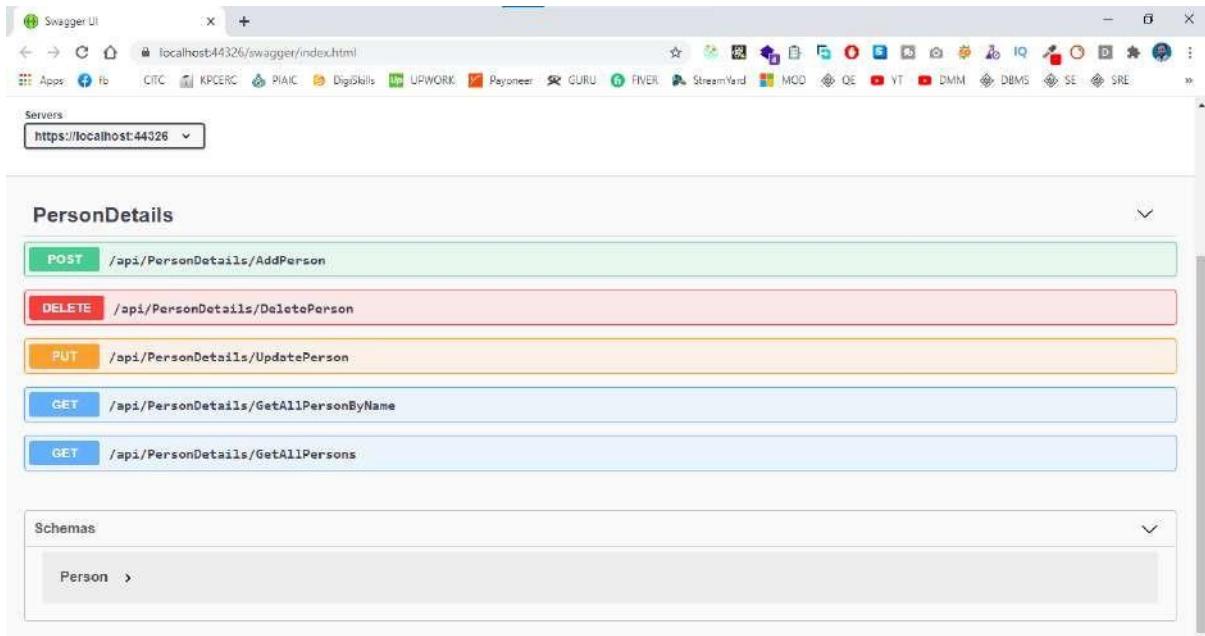
RESTful Web Service Methods

Web Resources were first defined on the world wide web as documents or files identified by their URLs today the definition is much generic and abstract and includes everything entity or action that can be identified Named Addressed handled or performed in any way on the web in a RESTful web service requests made to a resources URI gives us the response with the payload formatted in HTML XML JSON or some other format, for example, the response can be related resources the most common protocol is HTTP provides us following **HTTP Methods** **GET, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS AND TRACE.**

The HTTP methods details are given below.

- **POST** is used to post the data from the webserver to the database using the RESTful Web Service.
- **GET** is used to get the list of all records from the database either all records or based on specific I'd like Person ID or employed.
- **DELETE** is used to delete all the records from the database or based on a specific Id.
- **PUT** is used to Replace the record or update a record based on Specific Id.
- **PATCH** method provides an entity containing the list of changes to be applied to the resources requesting using the HTTP Request-URI.
- **OPTIONS** The options method represents a request for information about the communication options available on the request/response chain identified by Request-URI.

- **TRACE** The trace method performs a message look back test along the path to the target resources providing a useful debugging mechanism.



Why RESTful Web Service

Restful Mostly comes into popularity due to the following features.

- Heterogeneous language and environment this is one of the fundamental reasons which is same as we have seen for SOAP as well
- It enables web applications that are built on various programming languages to communicates with each other.
- RESTful web architecture helps the web applications to reside in different environments some could be on windows and others could be on Linux.
- RESTful web service offers this flexibility to applications built on various programming languages and platforms to talk to each other.

RESTful Architecture

An Application OR Architecture considered RESTful or Rest Style has the following Characteristics

- A-State and functionality are divided into distributed resources which means that every resource should be accessible via the normal HTTP Commands like getting, POST, PUT, DELETE, if someone wanted to get a file from a server, they should be able to issue the GET request and get the file. If we want to put the file on the server, we will use either

POST or PUT request, and finally, if we want to delete the file from the storage, we must use a DELETE request.

- The architecture is client/server stateless layered and supports the caching.
- The client-server is the typical architecture where the server can be web server hosting the application and the client can be as simple as the web browsers.
- Stateless means that the state of the application is not maintained in REST

If we want to delete a resource from a server using the DELETE Command, you cannot pass the deleted information to the next request. To ensure that the resources are deleted you would need to issue the GET request first to get all the information from the server after which one would need to see if the resource were deleted.

RESTful Principles and Constraints

The REST architecture is based on few characteristics which are elaborated below any RESTful web service must comply with the below characteristics for it to be called RESTful these characteristics are known as design principle which need to be filled when working with RESTful based architecture.

RESTful Client-Server

This is the most fundamental requirement of the REST Based architecture it means that the server will have a RESTful web service that would provide the required functionality to the client. The client sends a request to the web service on the server would either reject the request or comply and provide an adequate response to the client,



Stateless

The concept of stateless means that it is up to the client to ensure that all the required information is provided to the server. This is required so that server can process the response appropriately the server should not maintain any sort of the information between requests from the client requests the server and the server give the response to the client appropriately.

Cache

The Cache concept is to help with the problem of stateless which was described in the last point since each client request is independent sometimes the client might ask the server for the same request again even though it had already asked for it in the past. This is even though it had already asked for it in the past. This request will go to the server and the server will give a response. This increase the traffic across the network the cache is the concept implemented on the client to store requests which have already been sent to the server so if the same request is given by the client instead of going to the server this saves the amount of to and fro network traffic from the client to the server.



Layered System

The concept of a layered system is that any additional layer such as a middleware layer can be inserted between the client and the actual server hosting the RESTful web service the middleware layer is where all the business logic is created this can be an extra service created with a client could interact with before it makes a call to the web service. the layered system needs to be transparent so that interaction between client and server should Not Be Affected in any way.

RESTful Web Services Interfaces

This is the underlying technology in the web service how RESTful should work. RESTful works on the HTTP web layer and uses the below key verb to work with resources on the server. Web Resources were first defined on the world wide web as documents or files identified by their URLs today the definition is much generic and abstract and includes everything entity or action that can be identified Named Addressed handled or performed in any way on the web in a RESTful web service requests made to a resources URI gives us the response with the payload formatted in HTML XML JSON or some other format, for example, the response can be related resources the

most common protocol is HTTP provides us following **HTTP Methods** **GET, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS AND TRACE.**

The HTTP methods details are given below.

- **POST** is used to post the data from a web server to the database using the RESTful Web Service.
- **GET** is used to get the list of all records from the database either all records or based on specific I would like Person ID or employed.
- **DELETE** is used to delete all the records from the database or based on a specific Id.
- **PUT** is used to Replace the record or update a record based on a Specific Id.
- **PATCH** method provides an entity containing the list of changes to be applied to the resources requesting using the HTTP Request-URI.
- **OPTIONS** The options method represents a request for information about the communication options available on the request/response chain identified by Request-URI.
- **TRACE** The trace method performs a message look back test along the path to the target resources providing a useful debugging mechanism.

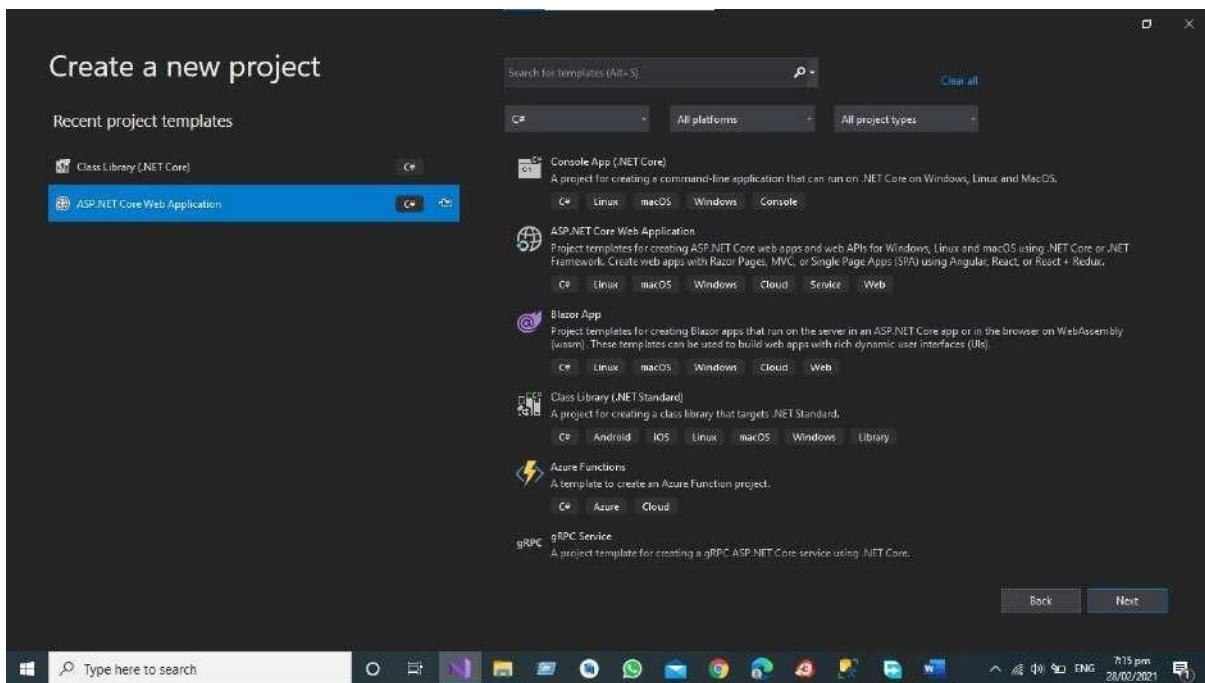
The screenshot shows a web browser window with the title "Swagger UI". The address bar indicates the URL is <https://localhost:44326/swagger/index.html>. The main content area is titled "PersonDetails". It lists several API endpoints:

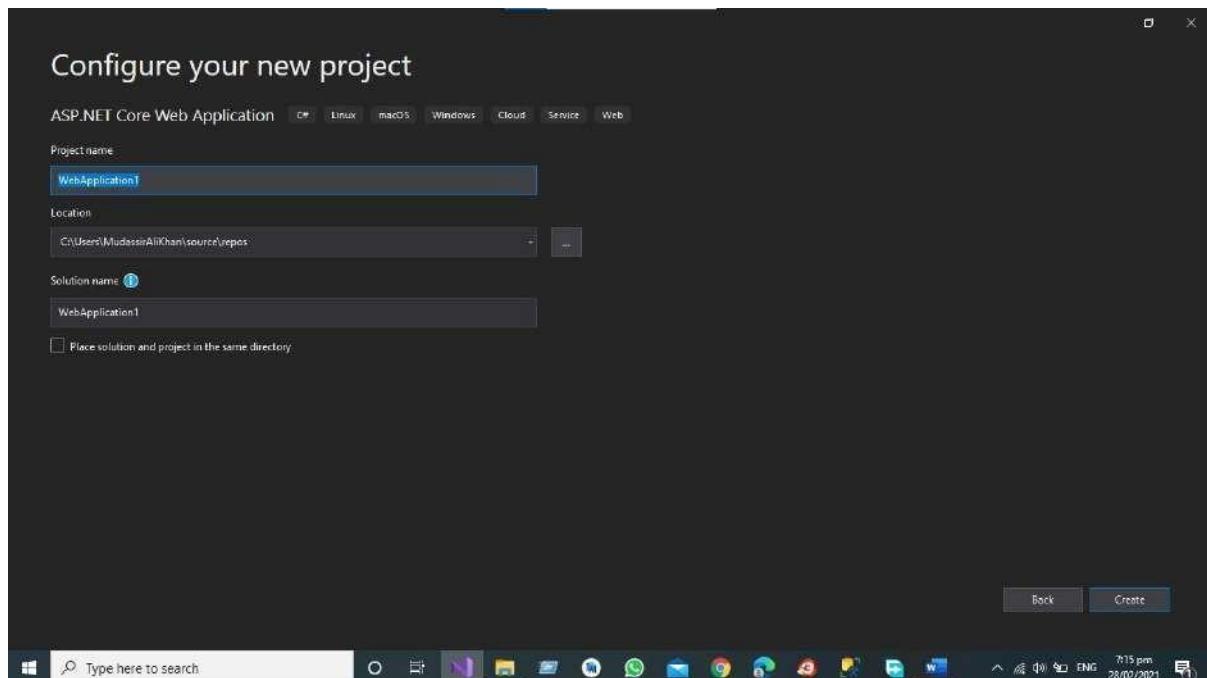
- POST** /api/PersonDetails/AddPerson
- DELETE** /api/PersonDetails/DeletePerson
- PUT** /api/PersonDetails/UpdatePerson
- GET** /api/PersonDetails/GetAllPersonByName
- GET** /api/PersonDetails/GetAllPersons

Below the endpoints, there is a section titled "Schemas" which contains a single entry: "Person".

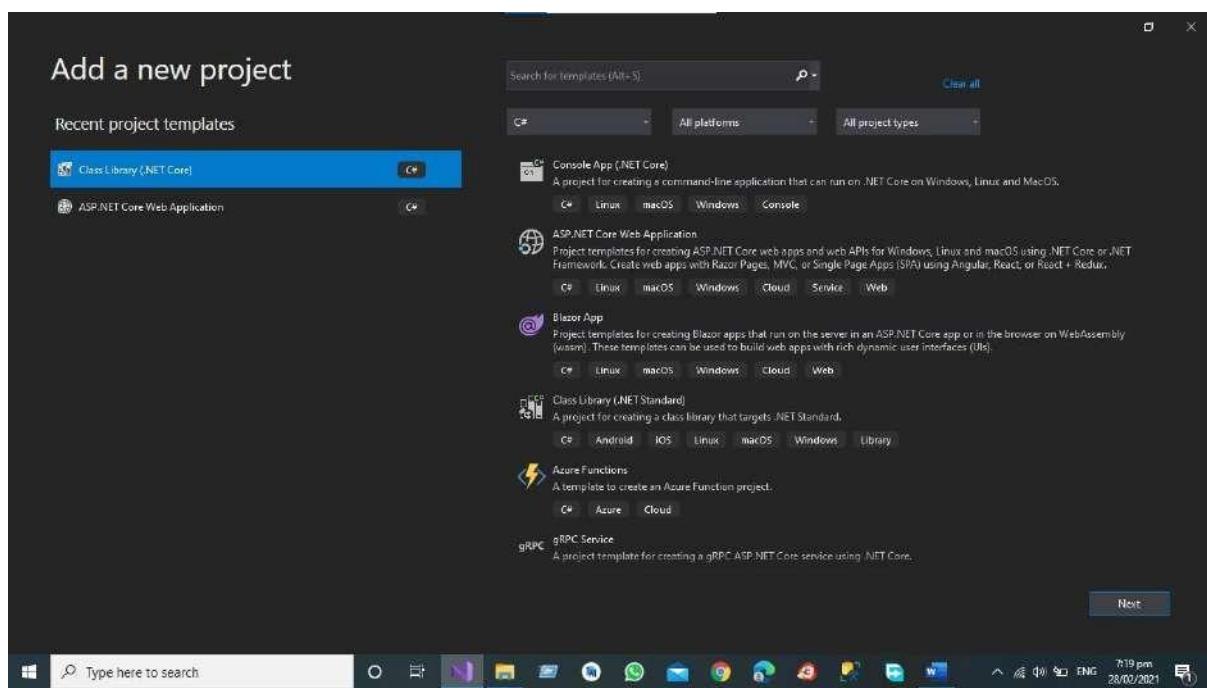
Create your first RESTful Web service in Asp.net Core Web API

Step 1 First, Create the Asp.net Core Web API in Visual Studio.





Step 2 Add the New Project in your Solution like Asp.net Core Class Library for Business Access Layer and Data Access Layer

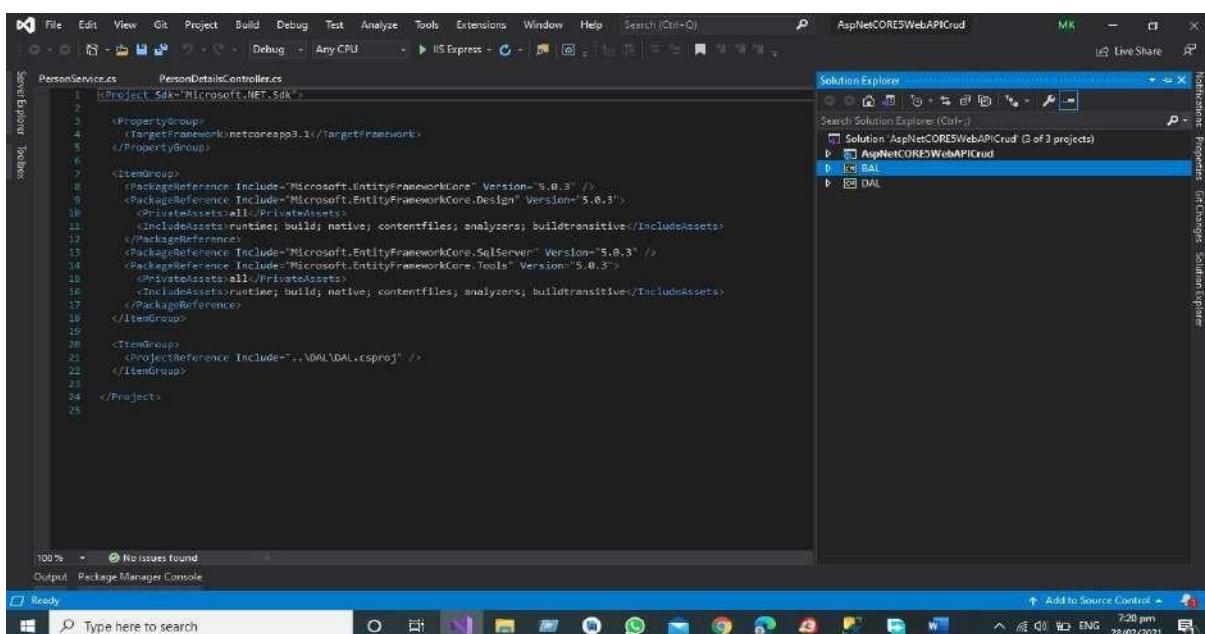


After Adding the BAL (Business Access Layer) And DAL (Data Access Layer)

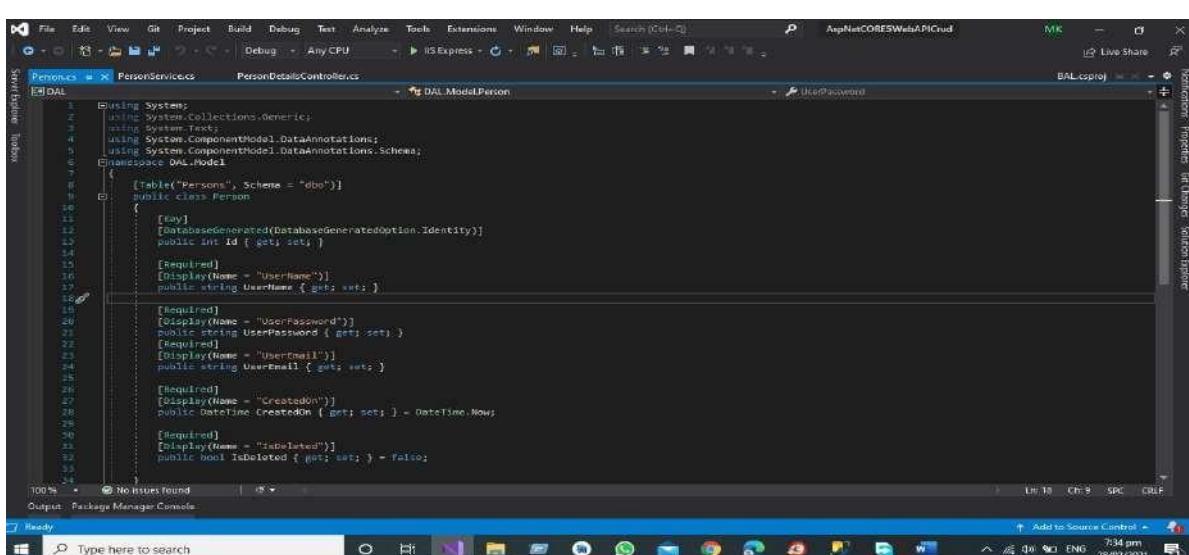
All the Data in Data Access Layer and All the Crud Operations are in Data Access Layer for Business Logic we have Business Access layer in which we will do our All Operations Related to our Business.

Data Access Layer

The Given below picture is the structure of our project according to the Our Initial steps.



Step 3 First, Add the Model Person in the Data Access Layer



Code is Given Below

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace DAL.Model

{
    [Table("Persons", Schema = "dbo")]
    public class Person
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

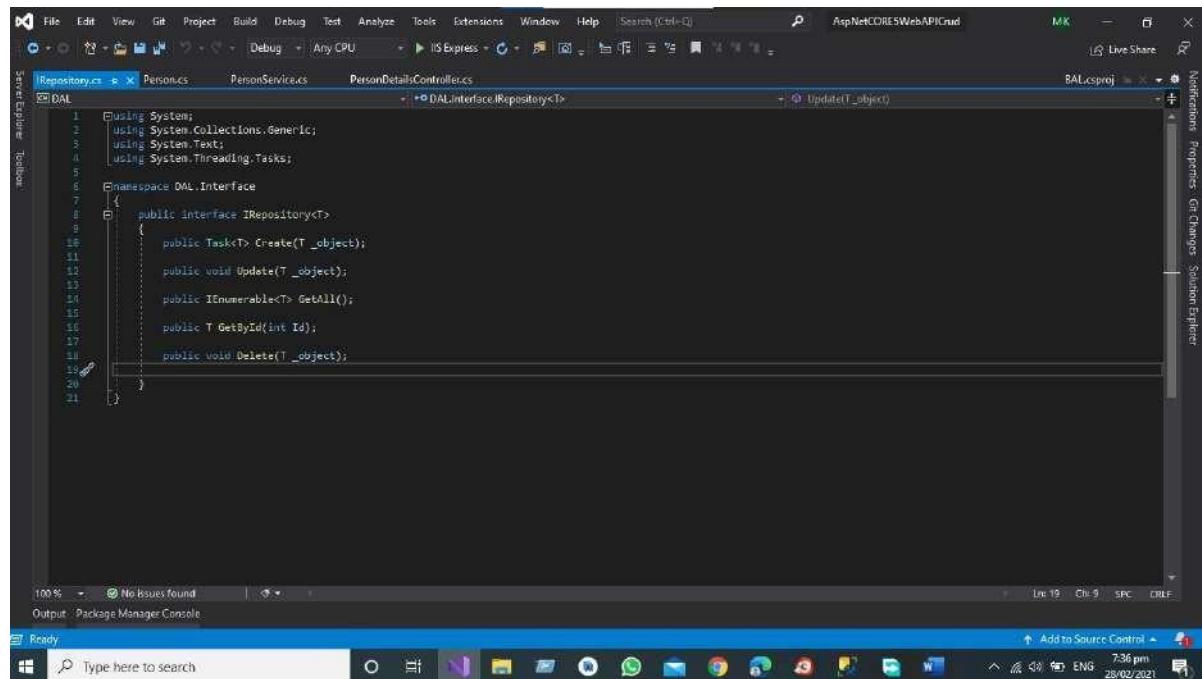
        [Required]
        [Display(Name = "UserName")]
        public string UserName { get; set; }

        [Required]
        [Display(Name = "UserPassword")]
        public string UserPassword { get; set; }
        [Required]
        [Display(Name = "UserEmail")]
        public string UserEmail { get; set; }

        [Required]
        [Display(Name = "CreatedOn")]
        public DateTime CreatedOn { get; set; } = DateTime.Now;

        [Required]
        [Display(Name = "IsDeleted")]
        public bool IsDeleted { get; set; } = false;
    }
}
```

Step 4 Add the Enter face for all the operation you want to perform.



The screenshot shows the Visual Studio IDE interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and Search (Ctrl+F). The title bar says "AspNetCORE5WebAPI.csproj". The left sidebar has "Server Explorer", "Toolbox", and "Solution Explorer". The main code editor window displays the file "Repository.cs" with the following code:

```
1  Using System;
2  Using System.Collections.Generic;
3  Using System.Text;
4  Using System.Threading.Tasks;
5
6  Namespace DAL.Interface
7  {
8      Public Interface IRepository<T>
9      {
10          Public Task<T> Create(T _object);
11          Public void Update(T _object);
12          Public IEnumerable<T> GetAll();
13          Public T GetById(int Id);
14          Public void Delete(T _object);
15      }
16  }
```

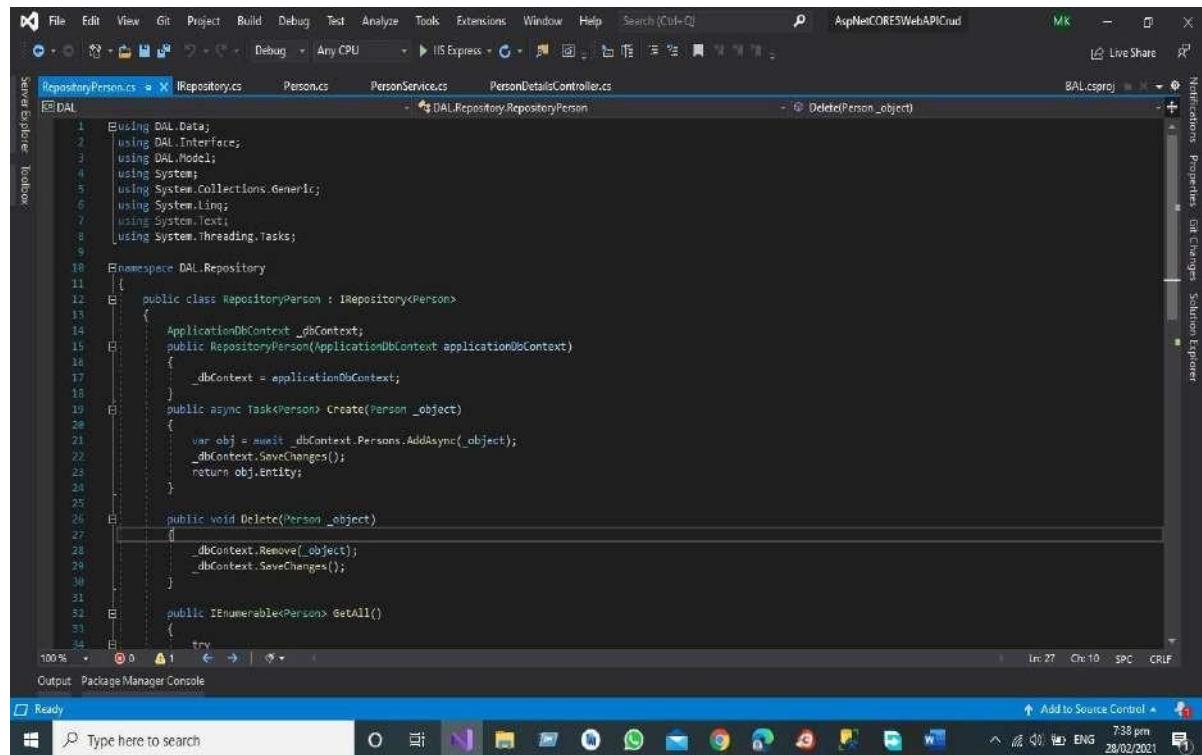
The status bar at the bottom shows "100% No Issues found", "Output: Package Manager Console", "Ready", "Type here to search", and system information like "Line 19 Col 9 SPC CRLF", "7:36 pm 28/02/2021", and "ENG".

Code is Given Below

```
using System;
using System.Collections.Generic; using
System.Text;
using System.Threading.Tasks;

namespace DAL.Interface
{
    public interface IRepository<T>
    {
        public Task<T> Create(T _object);
        public void Update(T _object);
        public IEnumerable<T> GetAll();
        public T GetById(int Id);
        public void Delete(T _object);
    }
}
```

Step 5 Apply the Repository Pattern in you project Add the Class Repository Person.



The screenshot shows the Visual Studio IDE with the code editor open to RepositoryPerson.cs. The code implements the Repository pattern for a Person entity. It includes using statements for DAL.Data, DAL.Interface, DAL.Model, System, System.Collections.Generic, System.Linq, System.Text, and System.Threading.Tasks. The class RepositoryPerson implements IRepository<Person>. It has a constructor taking an ApplicationDbContext parameter and setting _dbContext to it. It contains methods for Create, Delete, and GetAll. The Delete method uses _dbContext.Remove and _dbContext.SaveChanges. The GetAll method uses _dbContext.People and returns an IEnumerable<Person>. The code is annotated with comments explaining the purpose of each section.

```
1  using DAL.Data;
2  using DAL.Interface;
3  using DAL.Model;
4  using System;
5  using System.Collections.Generic;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9
10 namespace DAL.Repository
11 {
12     public class RepositoryPerson : IRepository<Person>
13     {
14         ApplicationDbContext _dbContext;
15         public RepositoryPerson(ApplicationDbContext applicationDbContext)
16         {
17             _dbContext = applicationDbContext;
18         }
19         public async Task<Person> Create(Person _object)
20         {
21             var obj = await _dbContext.People.AddAsync(_object);
22             _dbContext.SaveChanges();
23             return obj.Entity;
24         }
25
26         public void Delete(Person _object)
27         {
28             _dbContext.Remove(_object);
29             _dbContext.SaveChanges();
30         }
31
32         public IEnumerable<Person> GetAll()
33         {
34             try
35             {
36                 return _dbContext.People;
37             }
38             catch (Exception ex)
39             {
40                 throw;
41             }
42         }
43     }
44 }
```

Code is Given Below

```
using DAL.Data; using DAL.Interface; using DAL.Model; using System;
using System.Collections.Generic; using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace DAL.Repository
{
    public class RepositoryPerson : IRepository<Person>
    {
        ApplicationDbContext _dbContext;
        public RepositoryPerson(ApplicationDbContext applicationDbContext)
        {
            _dbContext = applicationDbContext;
```

```

}

public async Task<Person> Create(Person _object)
{
    var obj = await _dbContext.Persons.AddAsync(_object);
    _dbContext.SaveChanges(); return obj.Entity;
}

public void Delete(Person _object)
{
    _dbContext.Remove(_object);
    _dbContext.SaveChanges();
}

public IEnumerable<Person> GetAll()
{
    try
    {
        return _dbContext.Persons.Where(x => x.IsDeleted == false).ToList();
    }
    catch (Exception ee)
    {
        throw;
    }
}

public Person GetById(int Id)
{
    return _dbContext.Persons.Where(x => x.IsDeleted == false && x.Id == Id).FirstOrDefault();
}

```

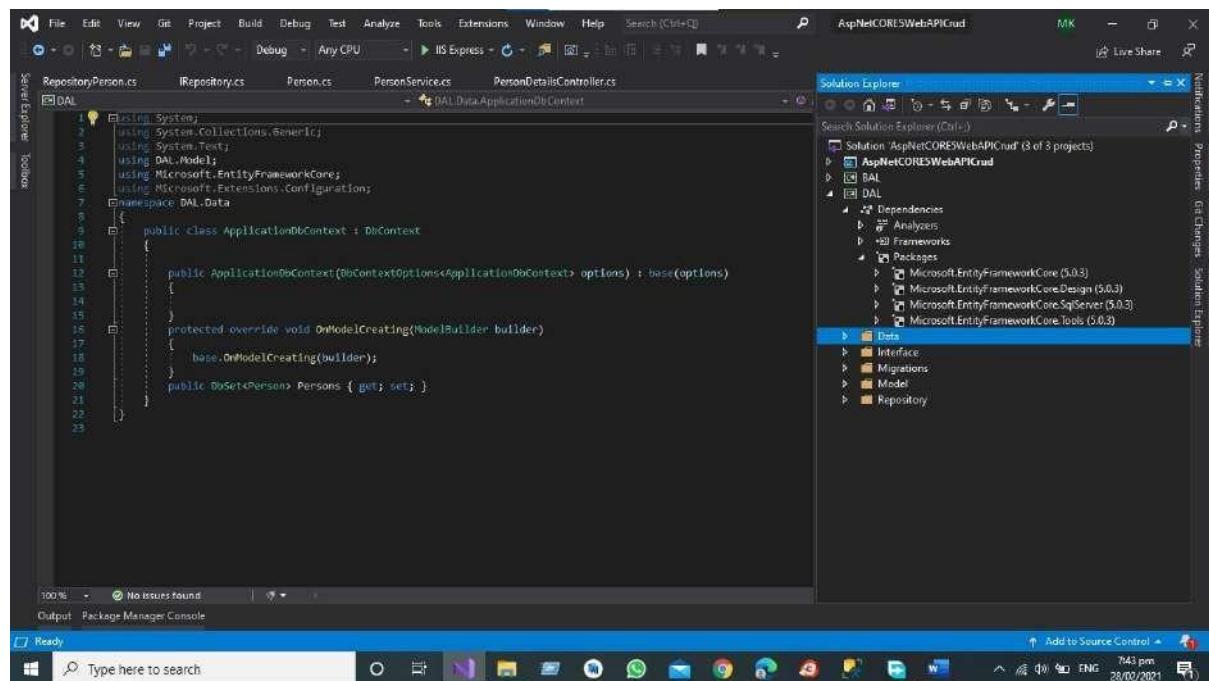
```

public void Update(Person _object)
{
    _dbContext.Persons.Update(_object);
    _dbContext.SaveChanges();
}

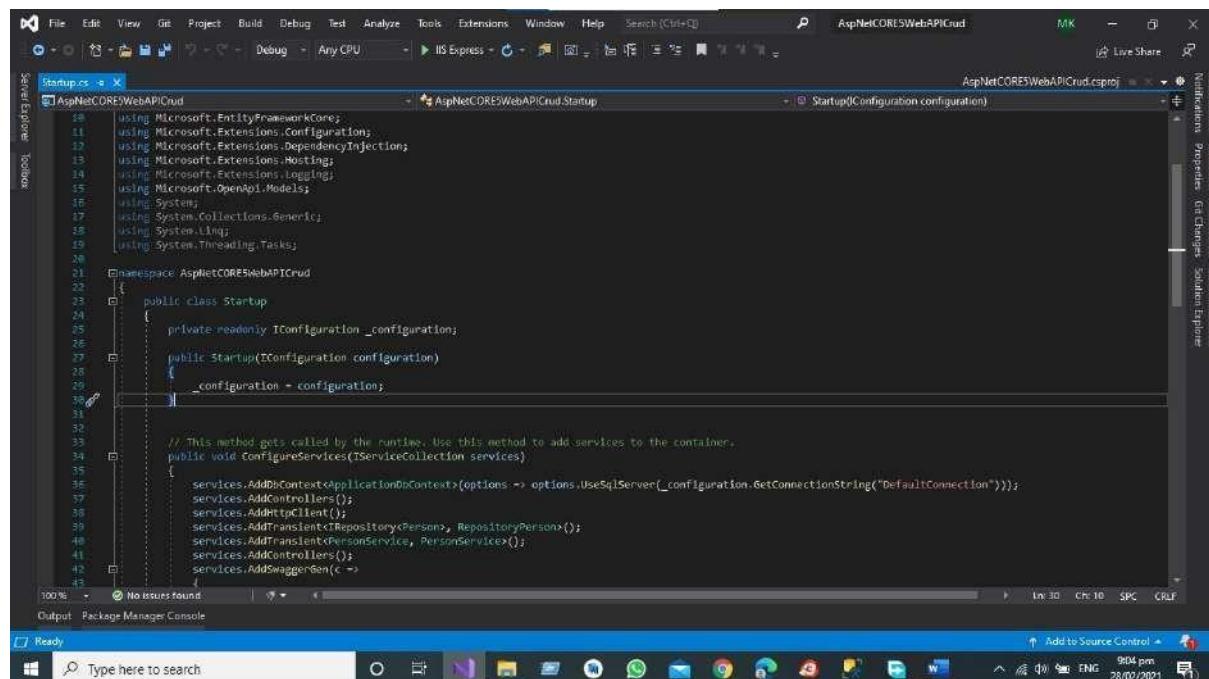
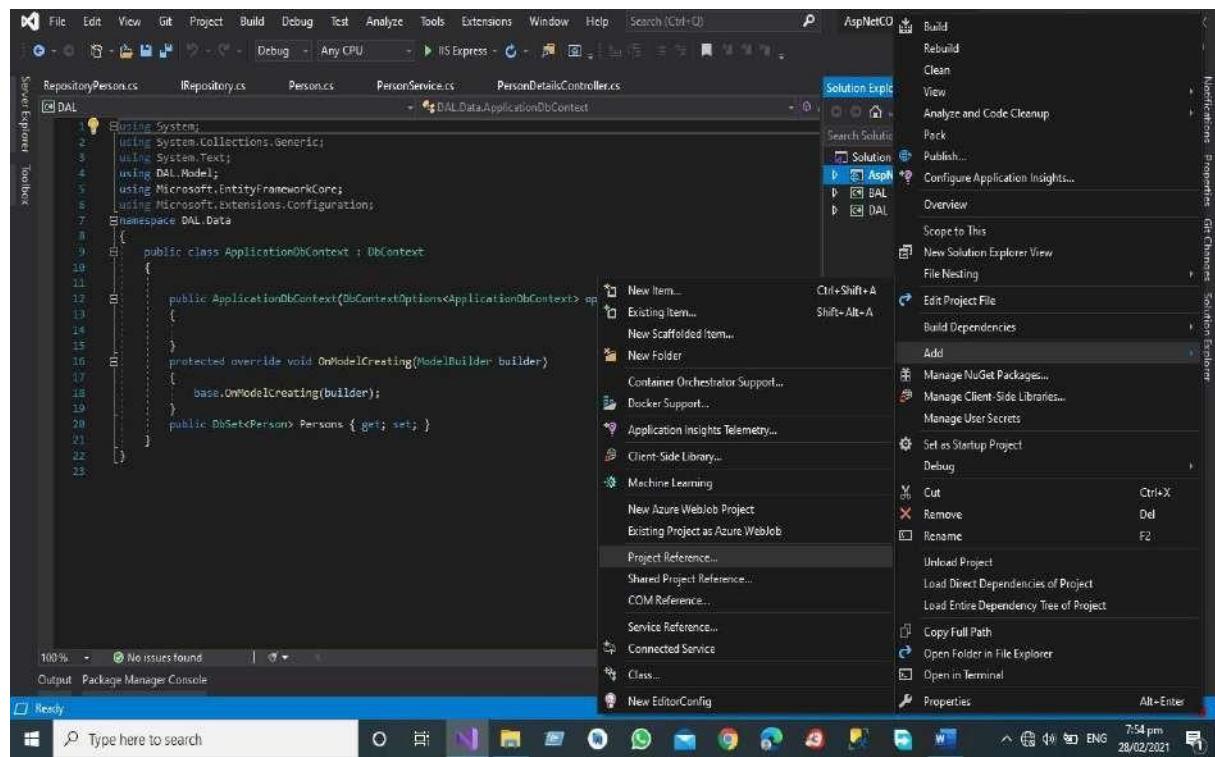
```

Step 6

Add the Data Folder and then add the Db Context Class set the database connection using SQL Server before starting the migration check these four packages in All of Your Project



Then Add the project references in All of Project Directory Like in API Project and BAL Project Like



Now Modify the Start-up Class in Project Solution.

Code of the start-up file is given below

```
using BAL.Service;
using DAL.Data;
using DAL.Interface;
using DAL.Model;
using DAL.Repository;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
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;
namespace AspNetCORE5WebAPICrud
{
    public class Startup
    {
        private readonly IConfiguration _configuration;

        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }
    }
}
```

```

    // This method gets called by the runtime. Use this method to add services
    to the container.

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(_configuration.GetConnectionString("DefaultConnection")));

        services.AddControllers();
        services.AddHttpClient();
        ;

        services.AddTransient< IRepository<Person>, RepositoryPerson>();
        services.AddTransient< PersonService, PersonService>();
        services.AddControllers();

        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title = "AspNetCORE5WebAPICrud",
Version = "v1" });
        });
    }

    // This method gets called by the runtime. Use this method to configure
    the HTTP request pipeline.

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
        }
    }

```

```

        app.UseSwaggerUI(c =>c.SwaggerEndpoint("/swagger/v1/swagger.json",
    "AspNetCORE5WebAPICrud v 1"));

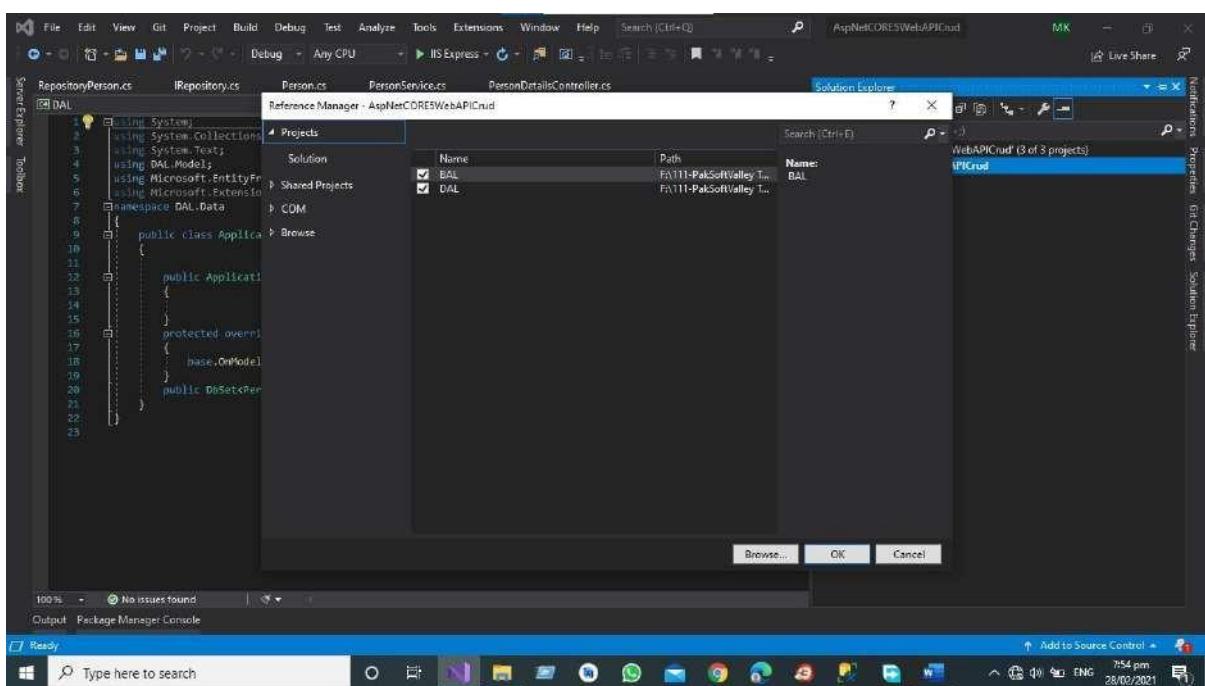
    }

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();

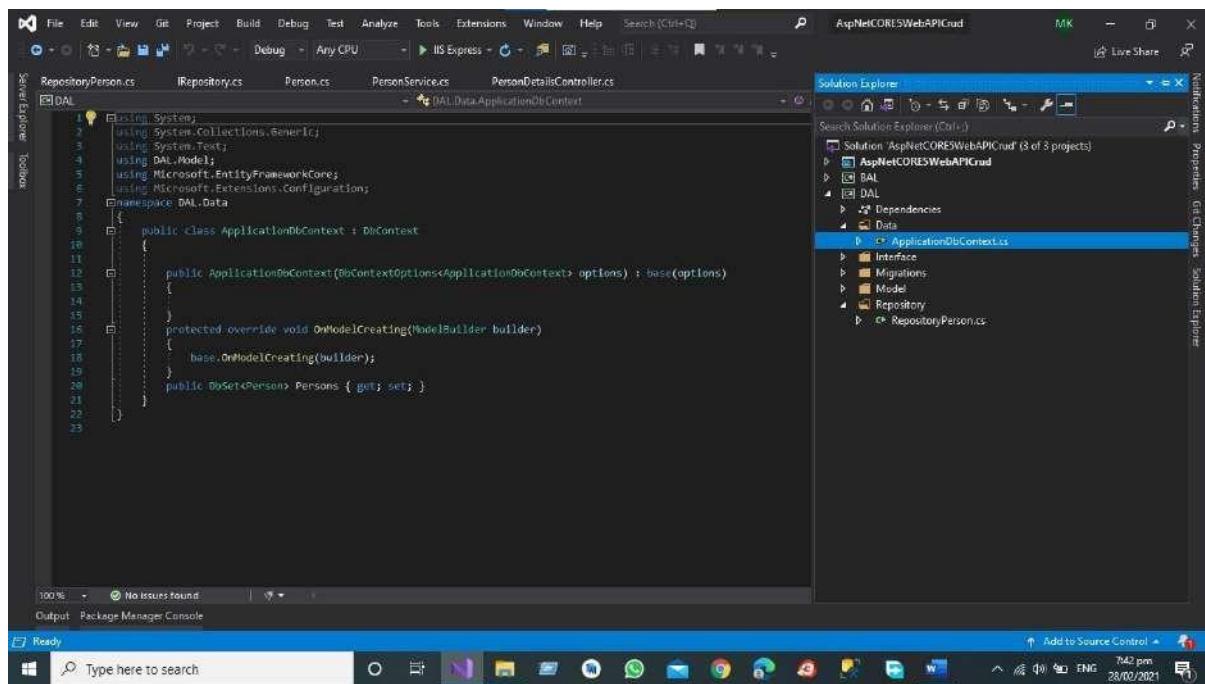
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}

}

```



Then Add the DB Set Property in your Application Db Context Class.



Code of Db Context Class is Given Below

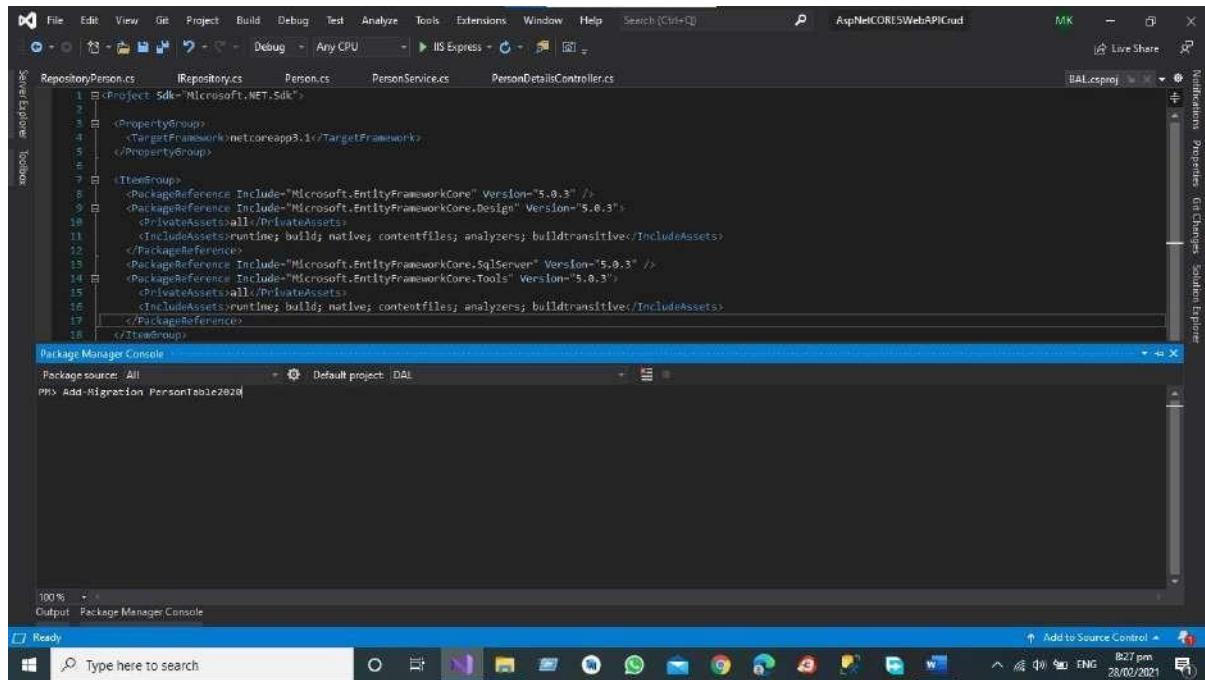
```

using System;
using System.Collections.Generic; using
System.Text;
using DAL.Model;
using Microsoft.EntityFrameworkCore; using
Microsoft.Extensions.Configuration;
namespace DAL.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
        {
        }
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
        public DbSet<Person> Persons { get; set; }
    }
}
public class ApplicationDbCont ext ( DbContextOptions<Appl i onDbCont ext > opt i ons) : base(options)
{
}
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
}
public DbSet<Person> Persons { get; set; }
}

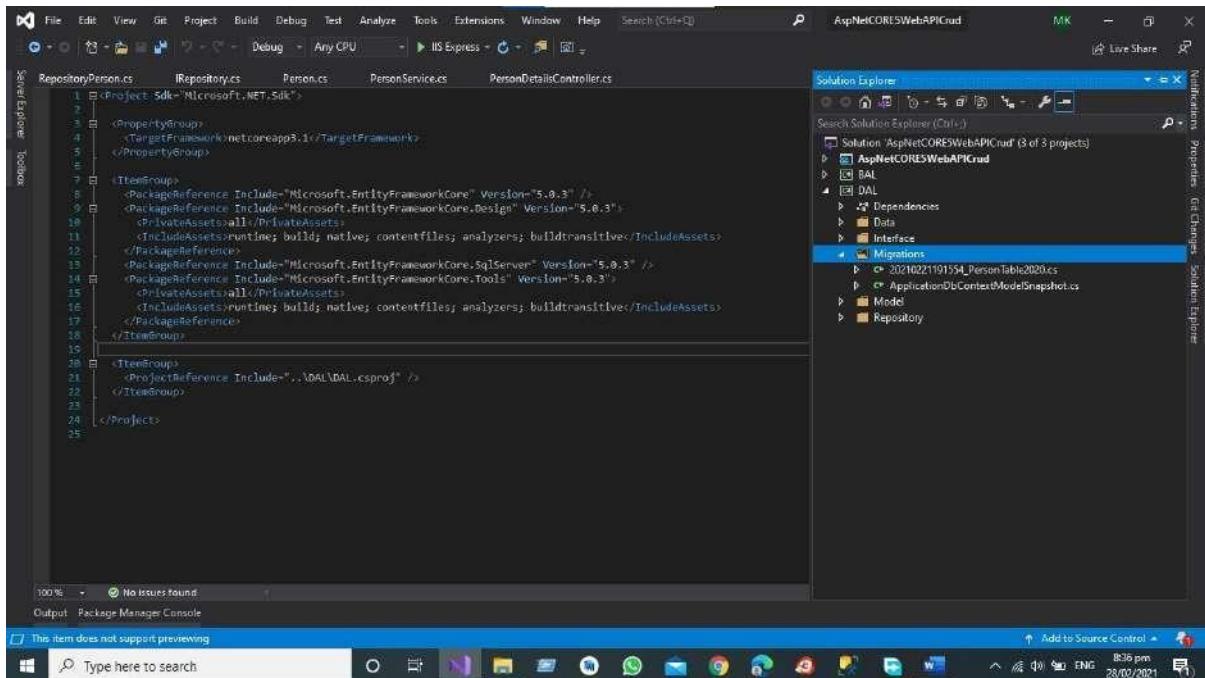
```

}

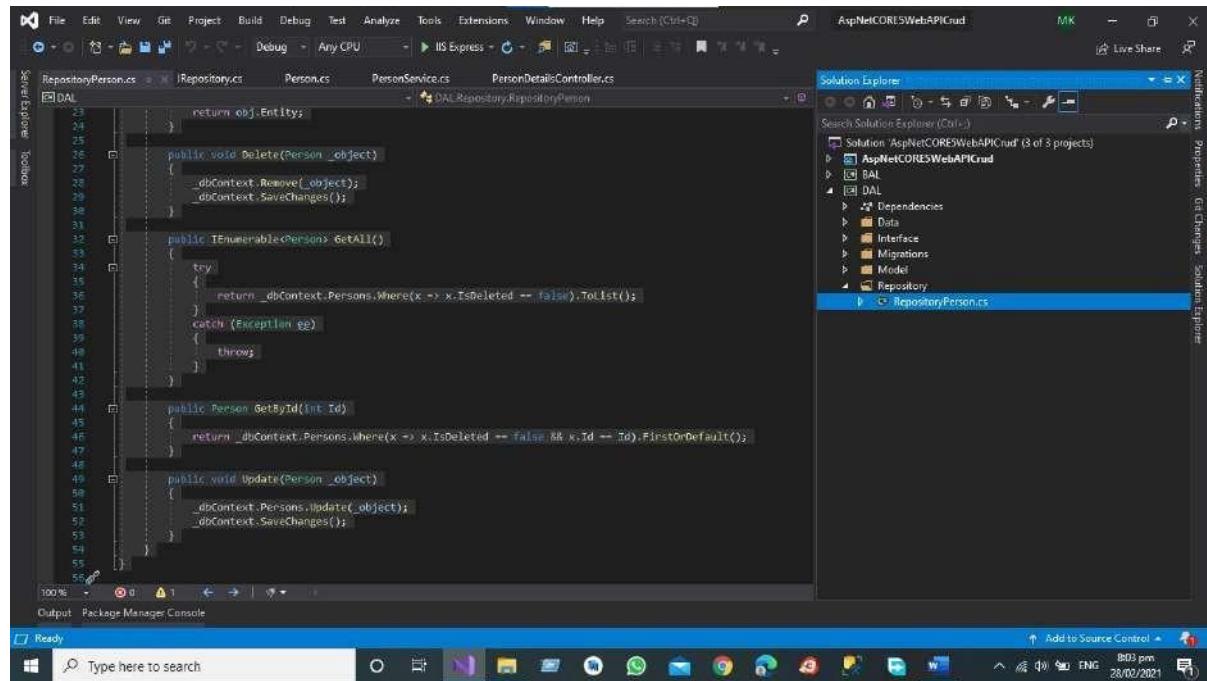
Step 7 Now start the migration Command for Creating the Table in Database.



After Executing the command, we have the migration folder in our Data Access Layer that contain the Definition of Person Table



Step 8 Add the Repository Person class then Implement the Interface and Perform the suitable Action according to the action define in Interface.



Code ID Given Below.

```

using DAL.Data;
using DAL.Interface;
using DAL.Model; using
System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DAL.Repository
{
    public class RepositoryPerson : IRepository<Person>
    {
        ApplicationDbContext _dbContext;
        public RepositoryPerson(ApplicationDbContext applicationContext)
        {
            _dbContext = applicationContext;
        }
    }
}

```

```

public async Task<Person> Create(Person _object)
{
    var obj = await _dbContext.Persons.AddAsync(_object);
    _dbContext.SaveChanges();
    return obj.Entity;
}

public void Delete(Person _object)
{
    _dbContext.Remove(_object);
    _dbContext.SaveChanges();
}

public IEnumerable<Person> GetAll()
{
    try
    {
        return _dbContext.Persons.Where(x => x.IsDeleted == false).ToList();
    }
    catch (Exception ee)
    {
        throw;
    }
}

public Person GetById(int Id)
{
    return _dbContext.Persons.Where(x => x.IsDeleted == false && x.Id == Id).FirstOrDefault();
}

public void Update(Person _object)
{
    _dbContext.Persons.Update(_object);
    _dbContext.SaveChanges();
}

```

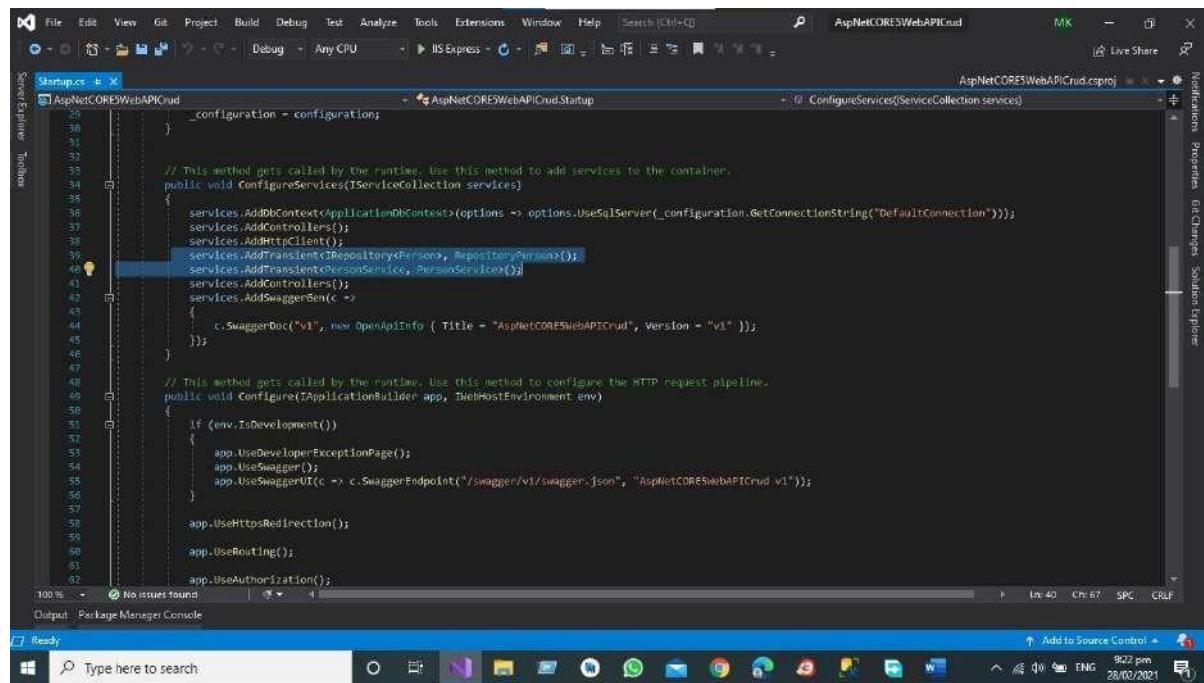
```

        }
    }
}

```

Step 10

Again, Modify the Start Up Class for Adding Transient to register the service business Layer and Data Access Layers Classes.



Code of the Start-up is given below.

```

using BAL.Service;
using DAL.Data; using
DAL.Interface; using
DAL.Model;

using DAL.Repository;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;

```

```

using Microsoft.EntityFrameworkCore;
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;

namespace AspNetCORE5WebAPICrud

{
    public class Startup

    {
        private readonly IConfiguration _configuration;

        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }

        // This method gets called by the runtime. Use this method to add services to the
        // container.
        public void ConfigureServices(IServiceCollection services)
        {

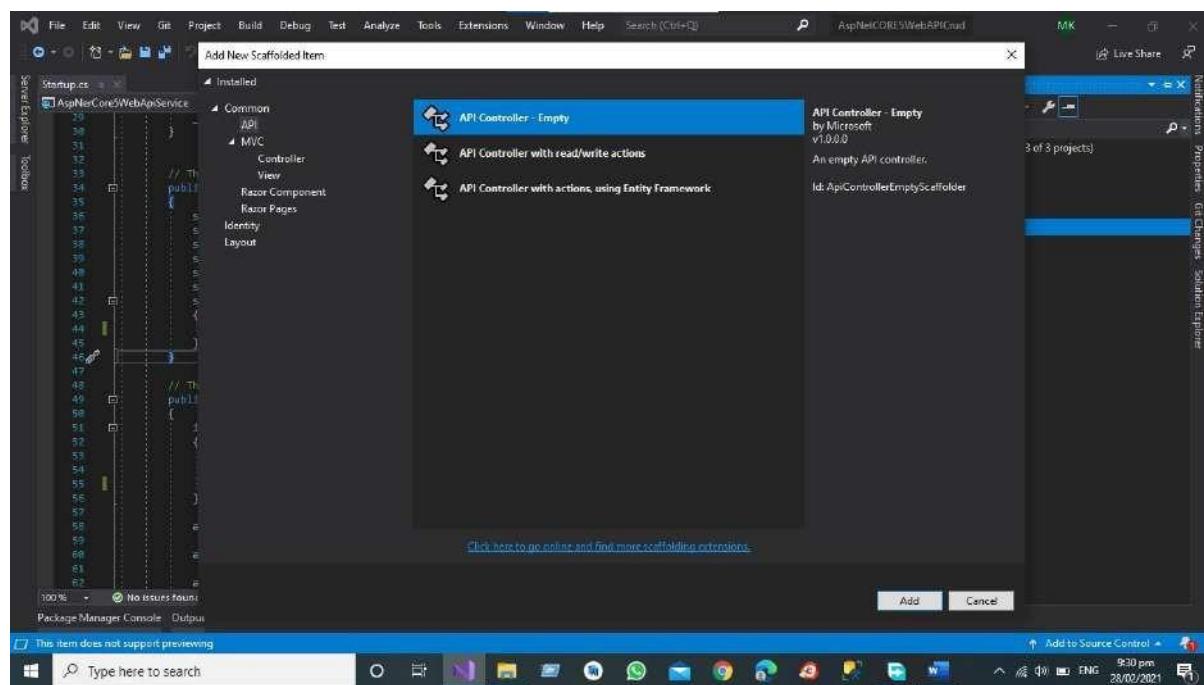
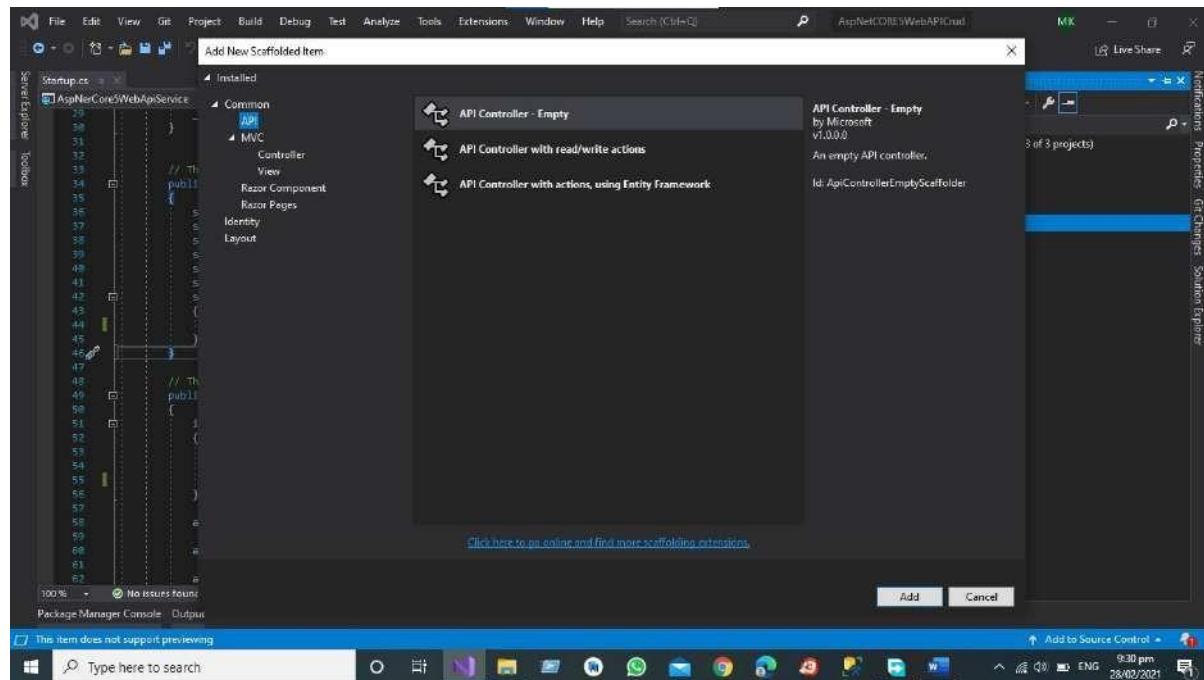
            services.AddDbContext<ApplicationContext>(options
                options.UseSqlServer(_configuration.GetConnectionString("DefaultConnection")));
            services.AddControllers();
            services.AddHttpClient();
            services.AddTransient<PersonService, PersonService>(); services.AddControllers();
            services.AddSwaggerGen(c =>
            {
        }
    }
}

```

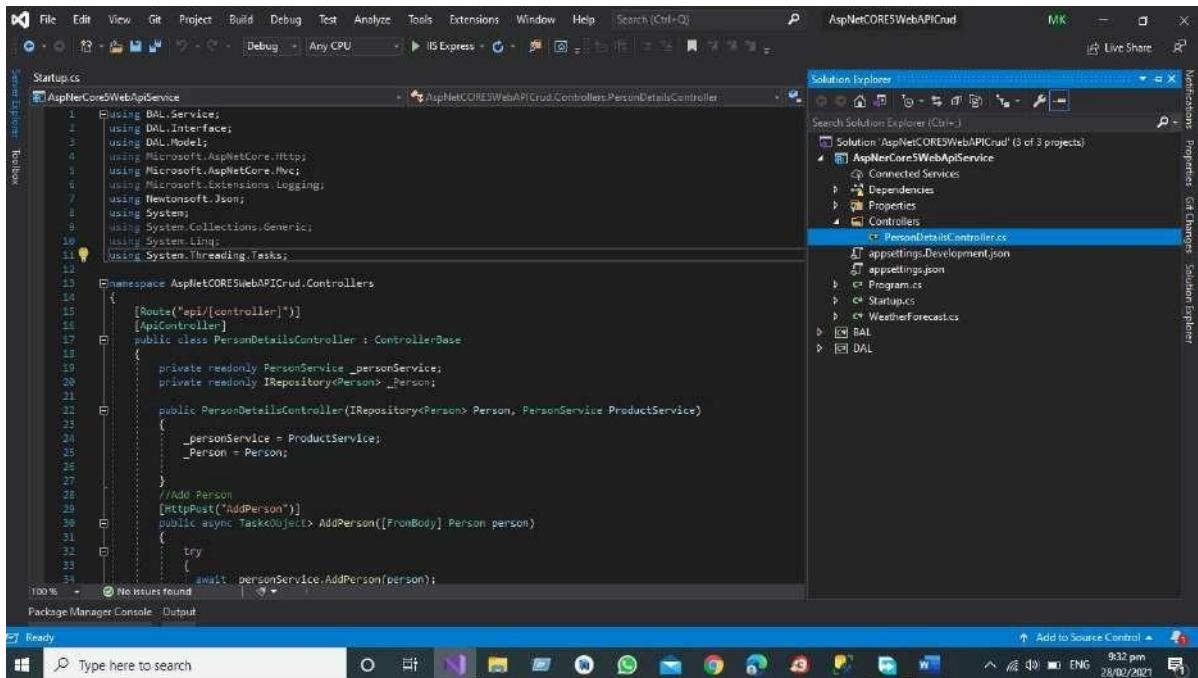
```
c.SwaggerDoc("v1", new OpenApiInfo { Title = "AspNetCORE5WebAPICrud", Version  
= "v1" }));  
}  
  
// This method gets called by the runtime. Use this method to configure the HTTP  
request pipeline.  
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
        app.UseSwagger();  
        app.UseSwagger UI ( c => c.SwaggerEndpoint("/swagger/v1/swagger.json",  
"AspNetCORE5WebAPICrud v 1"));  
    }  
  
    app.UseHttpsRedirection();  
    app.UseRouting();  
    app.UseAuthorization();  
  
    app.UseEndpoints(endpoints =>  
    {  
        endpoints.MapControllers();  
    } );  
}
```

Step 11

Add the Person Details API Controller for GET PUT POST AND DELETE end points



After Adding the Controller in the API project



Code of the Controller for Person Details is Given Below

```
using BAL.Service; using DAL.Interface;
using DAL.Model;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace AspNetCORE5WebAPICrud.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class PersonDetailsController : ControllerBase
    {
        private readonly PersonService _personService;
        private readonly IRepository<Person> _Person;

        public PersonDetailsController(IRepository<Person> Person, PersonService ProductService)
        {
            _personService = ProductService;
            _Person = Person;
        }

        //Add Person
        [HttpPost("AddPerson")]
        public async Task<Object> AddPerson([FromBody] Person person)
        {
            try
            {
                await _personService.AddPerson(person);
            }
            catch (Exception ex)
            {
                return new { error = ex.Message };
            }
        }
    }
}
```

```

}

//Add Person [HttpPost("AddPerson")]
public async Task<Object> AddPerson([FromBody] Person person)
{
    try
    {
        await _personService.AddPerson(person); return true;
    }
    catch (Exception)
    {

        return false;
    }
}

//Delete Person [HttpDelete("DeletePerson")]
public bool DeletePerson([FromBody] String UserEmail)
{
    try
    {

        _personService.DeletePerson(UserEmail); return true;

    catch (Exception)
    {

        return false;
    }
}

//Delete Person [HttpPut("UpdatePerson")]
public bool UpdatePerson([FromBody] String UserEmail)
{
    try
    {

        _personService.UpdatePerson(UserEmail); return true;

    catch (Exception)
    {

        return false;
    }
}

```

```

//GET All Person by Name [HttpGet("GetAllPersonByName")]
public Object GetAllPersonByName(string UserEmail)
{
    var data = _personService.GetPersonByUserName(UserEmail);
    var json = JsonConvert.SerializeObject(data, Formatting.Indented, new JsonSerializerSettings()
    {
        ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore
    });
    return json;
}

//GET All Person [HttpGet("GetAllPersons")]
public Object GetAllPersons()
{
    var data = _personService.GetAllPersons();
    var json = JsonConvert.SerializeObject(data, Formatting.Indented,
        new JsonSerializerSettings()
    {
        ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore
    });
    return json;
}

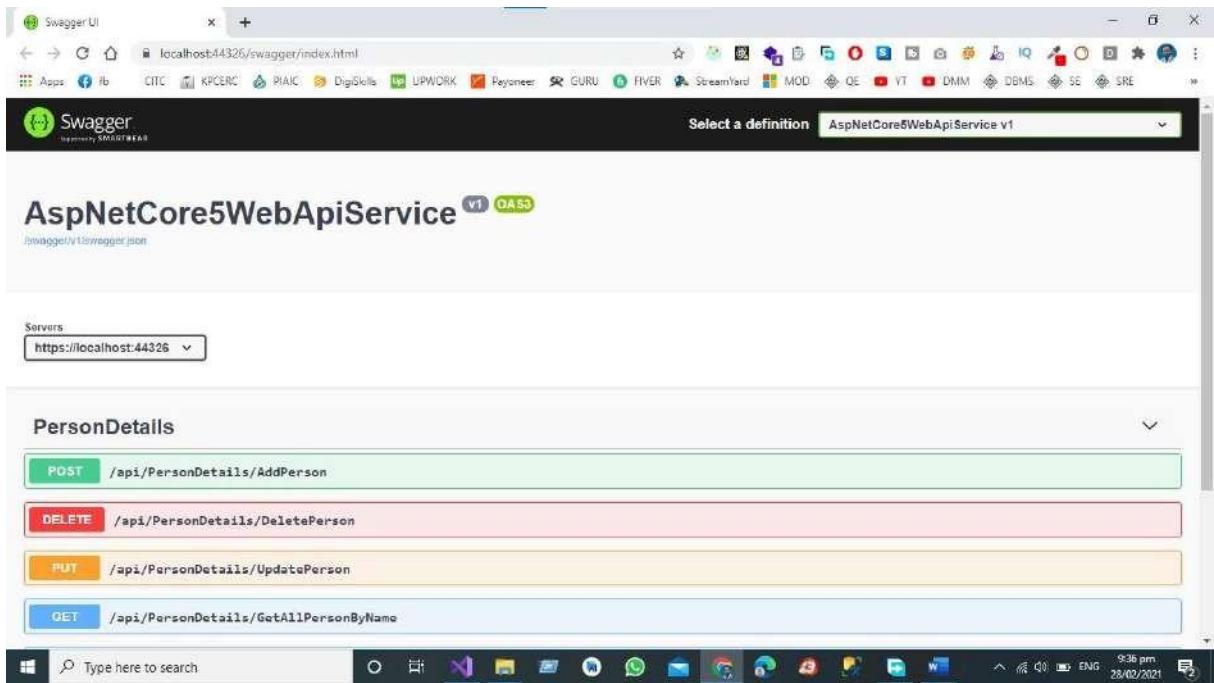
```

RESTful Web Service Results

To get the results of APIS We need to start the visual studio debugging process and then using swagger API Testing we will test all end points of Project.

Debug the project using visual studio

Swagger is Already Implemented in Our Project for Documenting the web service in this Project
We have Separate Endpoints for GET POST DELETE AND PUT



For Add the Person we have Add Person End Point that give us the JSON Object for sending the Complete JSON Object in the Database.

POST Endpoint in Web API Service

If we want to Create the New Person in the Database, we must send the JSON Object to the End Point then data is posted by Swagger UI to POST End Point in the Controller Then Create Person Object Creates the Complete Person Object in the Database.

PersonDetails

POST /api/PersonDetails/AddPerson

Parameters

No parameters

Request body

application/json

Example Value: Schema

```
{
  "id": 0,
  "userName": "string",
  "userPassword": "string",
  "userEmail": "string",
  "createdOn": "2021-02-28T16:30:30.104Z",
  "isDeleted": true
}
```

Responses

Code Description Links

200 Success No links

DELETE Endpoint in Web API Service

For deletion of person, we have a separate end point delete employee from the database, we must send the JSON object to the end point then data is posted by swagger UI to call the end point delete person in the controller then person object will be deleted from the database.

200 Success No links

DELETE /api/PersonDetails/DeletePerson

Parameters

No parameters

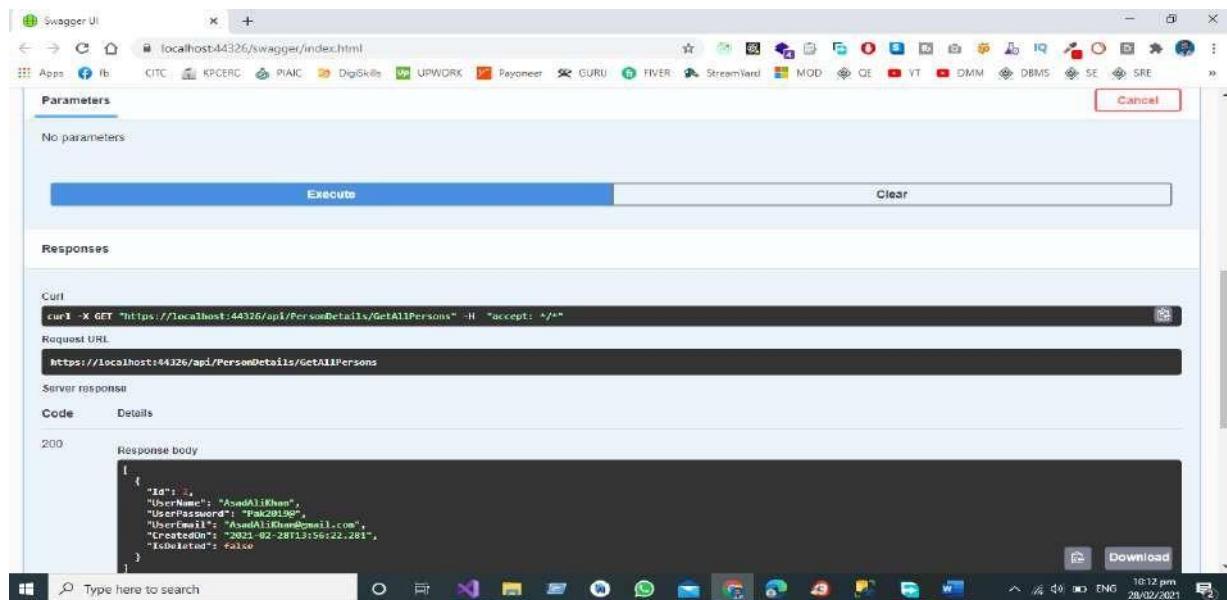
Request body

application/json

"string"

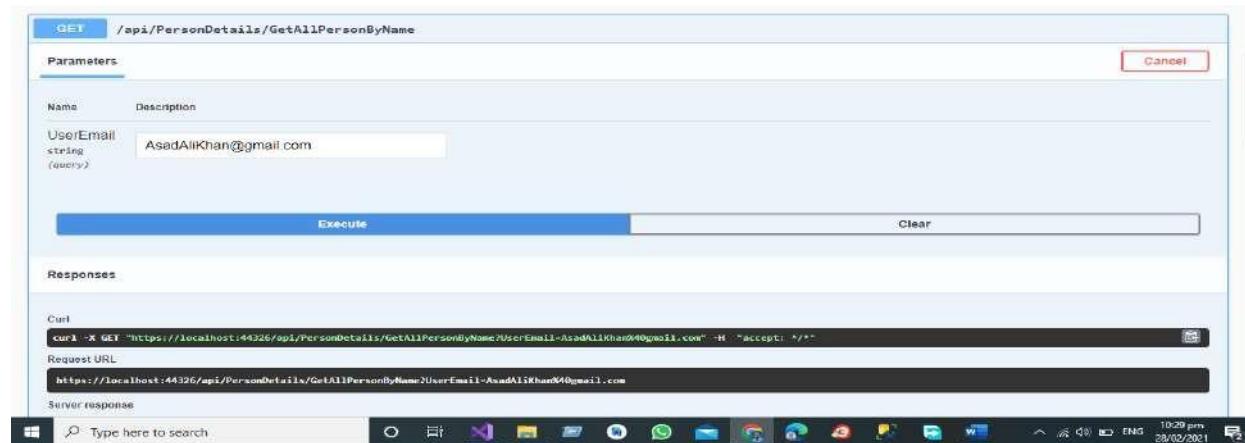
GET All Endpoint in Web API Service

We have Separate End Points for All the operations like Get All the Person Records from the Database by Hitting the GET End Point that send the request to controller and in controller we have the separate End Point for Fetching all the record from the database.



GET End Point for Getting the Record by User Email

We have Separate End Points for All the operations like Get the Record of the Person by User Email from the Database by Hitting the GET End Point that send the request to controller and in controller we have the separate End Point for Fetching the record based on User Email from the database.



The screenshot shows the Swagger UI interface for a .NET Core Web API. A successful GET request is displayed, returning a 200 OK status code. The response body contains a JSON object representing a person record:

```
{
  "Id": 2,
  "UserName": "AsadAliKhan",
  "UserPassword": "Pak2019",
  "UserEmail": "AsadAliKhan@gmail.com",
  "CreatedOn": "2021-02-28T15:56:22.281",
  "UpdatedOn": "2021-02-28T15:56:22.281"
}
```

The response headers include standard HTTP headers like Content-Type, Content-Encoding, and Date.

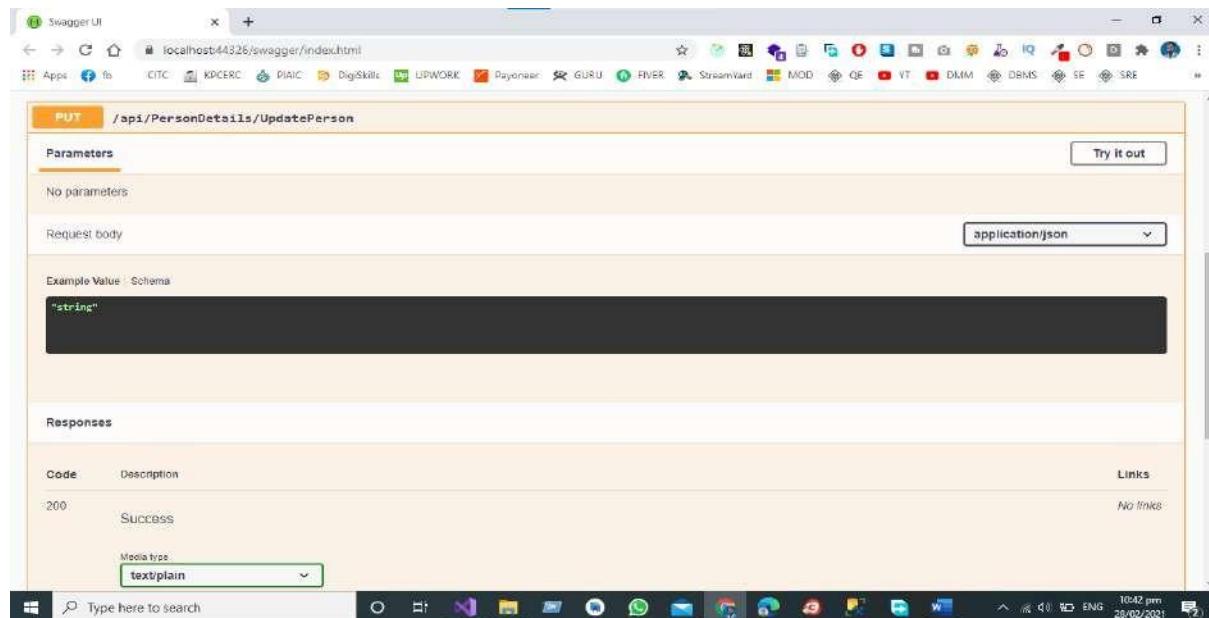
Delete End Point for Deleting the Record by User Email

We have Separate End Points for All the operations like Delete the Record of the Person by User Email from the Database by Hitting the DELETE End Point that send the request to controller and in controller we have the separate End Point for Deleting the record based on User Email from the database.

The screenshot shows the Swagger UI interface for the DELETE operation of the /api/PersonDetails/DeletePerson endpoint. The method is set to POST, but the URL path is correct. The operation description is "/api/PersonDetails/DeletePerson". The parameters section shows "No parameters". The request body is defined as application/json, containing the string "string".

Update End Point for Updating the Record by User Email

We have Separate End Points for All the operations like Update the Record of the Person by User Email from the Database by Hitting the UPDTEA End Point that send the request to controller and in controller we have the separate End Point for Updating the record based on User Email from the database.



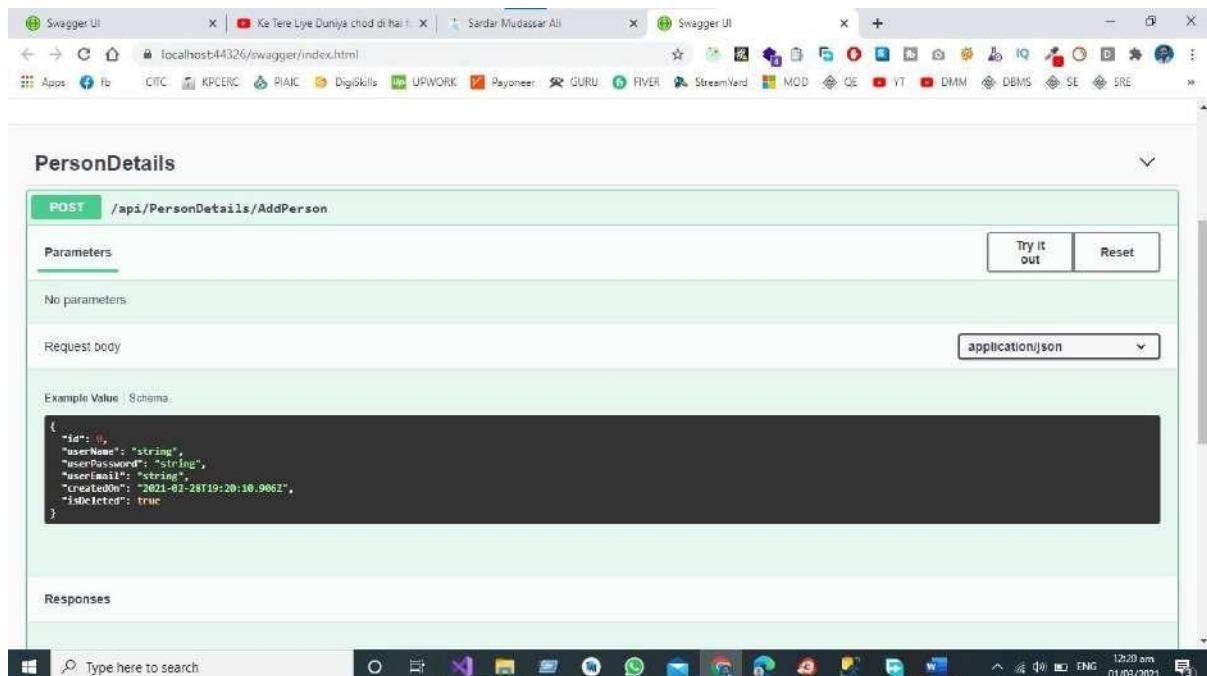
Testing the RESTful Web Service

Test the project using visual studio for that we need to follow given below steps

Step 1 Run the Project

Run the project then in the browser Swagger UI will be shown to the Tester

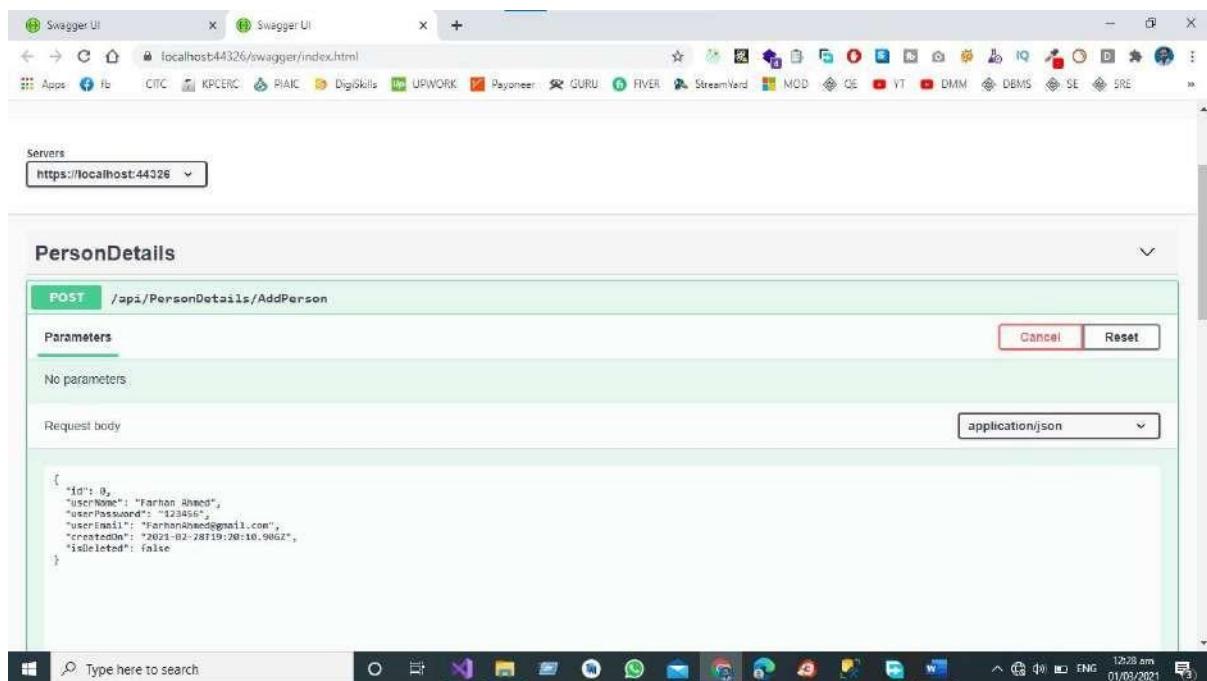
Step 2 Test POST End Point



The screenshot shows the Swagger UI interface for testing a POST request to the '/api/PersonDetails/AddPerson' endpoint. The 'Parameters' section indicates 'No parameters'. The 'Request body' section specifies 'application/json' and contains an example JSON object:

```
{
  "id": 0,
  "userName": "string",
  "userPassword": "string",
  "userEmail": "string",
  "createdOn": "2021-02-28T19:20:10.906Z",
  "isDeleted": true
}
```

Now Click on Try Out Button then and then send the JSON Object to the **Add Person** End Point that is already Define in Our Controller.



The screenshot shows the same Swagger UI interface after clicking the 'Try it out' button. The 'Responses' section is now visible, and the 'Cancel' button has replaced the original 'Try it out' button. The JSON object remains the same as in the previous screenshot.

We have sent the JSON Object to the End Point of API as shown in the figure and the Response Body for this call is true means that our One Object is Created in in Database.

The screenshot shows the Swagger UI interface for a .NET Core Web API. The URL in the browser is `localhost:44326/swagger/index.html`. The main panel displays the response to a POST request to the endpoint `/api/PersonDetails/AddPerson`. The response code is 200, and the response body is `true`. The response headers include standard CORS headers and a timestamp. The bottom status bar shows the date as 01/03/2021 and the time as 12:28 am.

Step 3 Test GET End Point

The screenshot shows the Swagger UI interface for a .NET Core Web API. The URL in the browser is `localhost:44326/swagger/index.html`. The main panel displays the response to a GET request to the endpoint `/api/PersonDetails/GetAllPersons`. The response code is 200, and the response body is `Success`. The bottom status bar shows the date as 01/03/2021 and the time as 12:29 am.

Click the Tryout Button to get the Get All Record.

The screenshot shows the Swagger UI interface for a .NET Core Web API. The top navigation bar has two tabs: 'Swagger UI' and 'Swagger UI'. The main content area is titled 'Responses' under the 'curl' section. It displays a curl command to get all persons: `curl -X GET "https://localhost:44326/api/PersonDetails/GetAllPersons" -H "Accept: */*`. Below this is a 'Request URL' input field containing `https://localhost:44326/api/PersonDetails/GetAllPersons`. The 'Server response' section shows a 'Code' tab selected, displaying a 200 status code response body. The response body is a JSON array of two objects, each representing a person with properties like Id, UserName, UserPassword, UserEmail, and CreationDate. There is also a 'Download' button next to the response body. At the bottom of the response section, there are response headers: `access-control-allow-methods: GET,PUT,POST,DELETE,HEAD,OPTIONS`, `access-control-allow-origin: https://localhost:44326`, and `content-encoding: gzip`. The bottom of the window shows a Windows taskbar with various pinned icons and the date/time as 12:33 am 01/03/2021.

The screenshot shows the Swagger UI interface for the 'PersonDetails' endpoint. The top navigation bar has two tabs: 'Swagger UI' and 'Swagger UI'. The main content area is titled 'PersonDetails'. It lists two operations: 'POST /api/PersonDetails/AddPerson' and 'DELETE /api/PersonDetails/DeletePerson'. Under the 'DELETE' operation, there is a 'Parameters' section with a table showing a parameter named 'UserEmail' of type 'string' with a description '(query)'. To the right of the table is a 'Try it out' button. Below the operations is a 'Responses' section with a table showing a 200 status code response with a 'Success' description. The bottom of the window shows a Windows taskbar with various pinned icons and the date/time as 12:33 am 01/03/2021.

UserEmail
string
(query)

Code	Description	Links
200	SUCCESS	No links

Media type:
text/plain
Controls accept header.
Example Value | Schema
true

PUT /api/PersonDetails/UpdatePerson

GET /api/PersonDetails/GetAllPersonByName

Type here to search

1237 am ENG 01/03/2021

Step 4 Test **DELETE** End Point

Now Click on Try Out Button then and then send the JSON Object to the **Delete** End Point that is already Define in Our Controller.

200 Success

DELETE /api/PersonDetails/DeletePerson

Parameters

Name	Description
UserEmail string (query)	FarhanAhmed@gmail.com

Execute Clear

Responses

Curl:
curl -X DELETE "https://localhost:44326/api/PersonDetails/DeletePerson?UserEmail=FarhanAhmed%40gmail.com" -H "accept: text/plain"

Request URL:
https://localhost:44326/api/PersonDetails/DeletePerson?UserEmail=FarhanAhmed%40gmail.com

We have sent the User Email to the End Point of API as shown in the figure and the Response Body for this call is true means that our One Object is Deleted in in Database.

Step 5 Test GET by User Email End

The screenshot shows the Swagger UI interface for an API. The URL is `localhost:44326/swagger/index.html`. The main content area displays the `GET /api/PersonDetails/GetAllPersonByName` endpoint. Under 'Parameters', there is one parameter named `UserEmail` with type `string (query)` and value `UserEmail`. Under 'Responses', a 200 status code is listed with the description `Success` and a note 'No links'. At the bottom, there are buttons for `POST /api/PersonDetails/AddPerson`, `DELETE /api/PersonDetails/DeletePerson`, and `PUT /api/PersonDetails/UpdatePerson`. The `PUT` button is highlighted in orange. Below these is the `GET /api/PersonDetails/GetAllPersonByName` button, which is also highlighted in orange. The Windows taskbar at the bottom shows various pinned icons.

Point

Now Click on Try Out Button then and then send the JSON Object to the **GET** End Point that is already Define in Our Controller for getting Record on the bases of Person Email.

The screenshot shows the Swagger UI interface for an API. The URL is `localhost:44326/swagger/index.html`. The main content area displays the `PUT /api/PersonDetails/UpdatePerson` endpoint. Under 'Parameters', there is one parameter named `UserEmail` with type `string (query)` and value `AsadAliKhani@gmail.com`. Below the parameters is a large blue 'Execute' button. At the bottom, there is a terminal window showing a curl command: `curl -X GET "https://localhost:44326/api/PersonDetails/GetAllPersonByName?UserEmail=AsadAliKhani@gmail.com" -H "accept: */*`. The Windows taskbar at the bottom shows various pinned icons.

We get the Data from the Database According to the User Gmail Id

```
curl -X GET "https://localhost:44326/api/PersonDetails/GetAllPersonByName?UserEmail=AsadAliKhan%40gmail.com" -H "accept: */*"
Request URL
https://localhost:44326/api/PersonDetails/GetAllPersonByName?UserEmail=AsadAliKhan%40gmail.com
Server response
Code Details
200 Response body
[{"Id": 1, "UserName": "AsadAliKhan", "UserPassword": "P@ss2019@*", "UserEmail": "AsadAliKhan@gmail.com", "CreatedOn": "2021-02-28T13:56:22.281", "IsDeleted": false}]
Response headers
access-control-allow-methods: GET,PUT,POST,DELETE,HEAD,OPTIONS
access-control-allow-origin: https://localhost:44326
content-encoding: gzip
content-type: text/plain; charset=utf-8
date: Sun, 28 Feb 2021 19:46:16 GMT
server: Microsoft-IIS/10.0
vary: Accept-Encoding
x-powered-by: ASP.NET
```

Conclusion

As a Software Engineer My Opinion about Restful web services are a lightweight maintainable and scalable service that is built on the REST Architecture. Restful Web service exposes API from your application in a secure uniform stateless manner to the calling client can perform predefined operations using the restful service the protocols for the REST is HTTP and Rest is a Representational State Transfer. Representational state transfer (REST) is a de-facto standard for a software Architecture or Interactive applications that use Web Service a Web Service that follows this standard is called RESTful such web must provide its web resources in a textual representation and allow them to be read and modified with stateless protocol and predefined set of operations this standard allow interoperability between a Machine and the Internet that provide these services. REST Architecture is alternative to SOAP and SOAP Architecture is the way to access a Web Service.

Complete Git Hub Project URL

[Click here for complete project access](#)

Chapter 2 – API Development in Asp.Net Core Using Three Tier Architecture

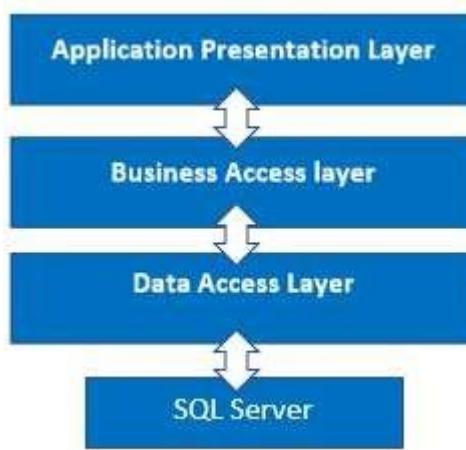
- ✓ Introduction
- ✓ Project Structure
- ✓ Presentation Layer (PL)
- ✓ Business Access Layer (BAL)
- ✓ Data Access Layer (DAL)
- ✓ Add the References to the Project
- ✓ Data Access Layer
- ✓ Contacts
- ✓ Data
- ✓ Migrations
- ✓ Models
- ✓ Repository
- ✓ Business Access Layer
- ✓ Services
- ✓ Presentation layer
- ✓ Advantages of 3-Tier Architecture
- ✓ Dis-Advantages of 3-Tier Architecture
- ✓ Conclusion
- ✓ Complete GitHub Project URL

Introduction

In this chapter, we'll look at three-tier architecture and how to incorporate the Data Access Layer and Business Access Layer into a project, as well as how these layers interact.

Project Structure

We use a three-tier architecture in this project, with a data access layer, a business access layer, and an application presentation layer.



Presentation Layer (PL)

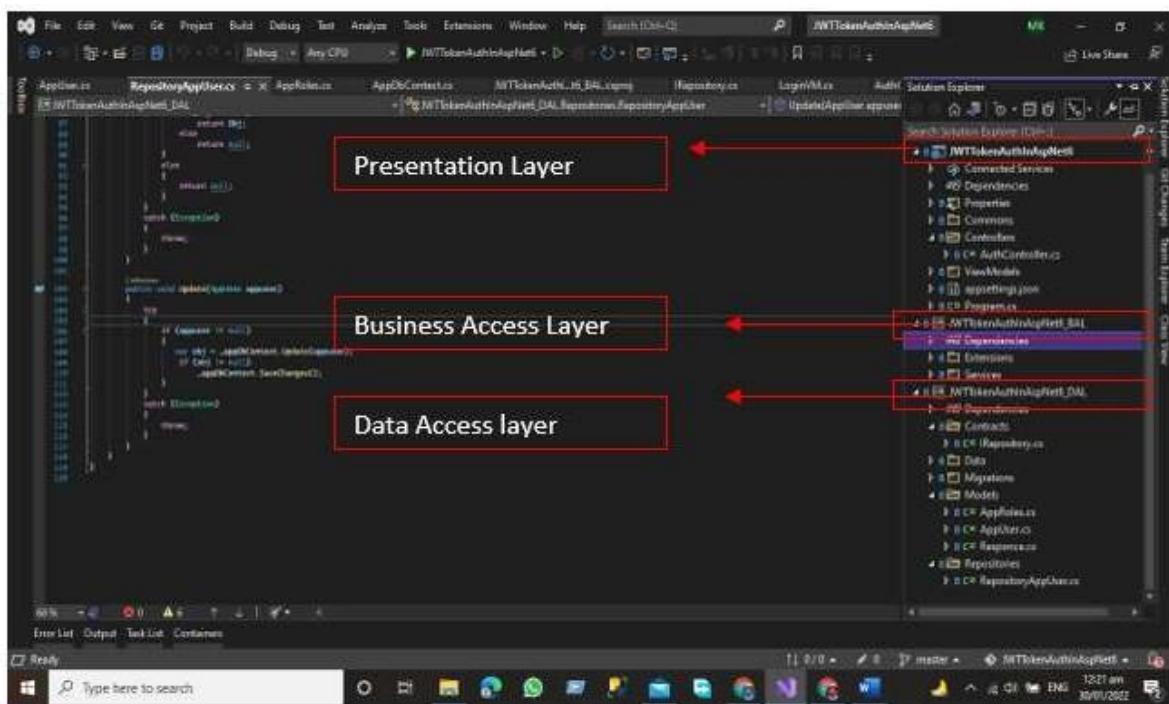
The Presentation layer is the top-most layer of the 3-tier architecture, and its major role is to display the results to the user, or to put it another way, to present the data that we acquire from the business access layer and offer the results to the front-end user.

Business Access Layer (BAL)

The logic layer interacts with the data access layer and the presentation layer to process the activities that lead to logical decisions and assessments. This layer's primary job is to process data between other layers.

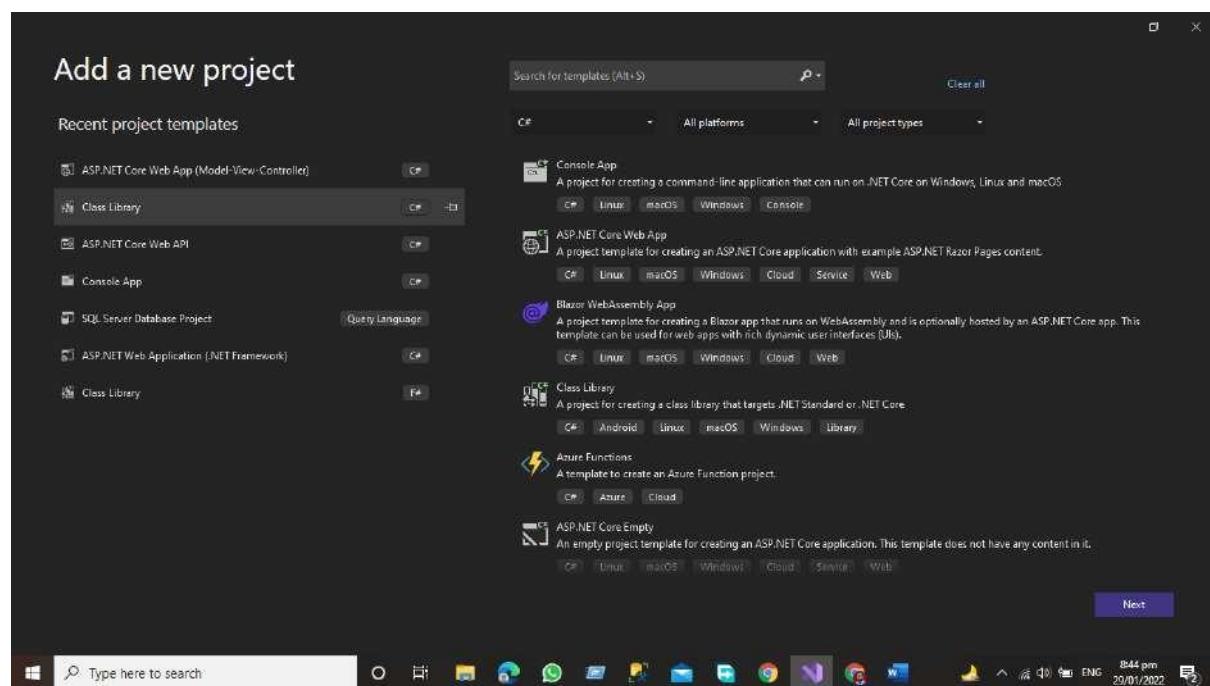
Data Access Layer (DAL)

The main function of this layer is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.

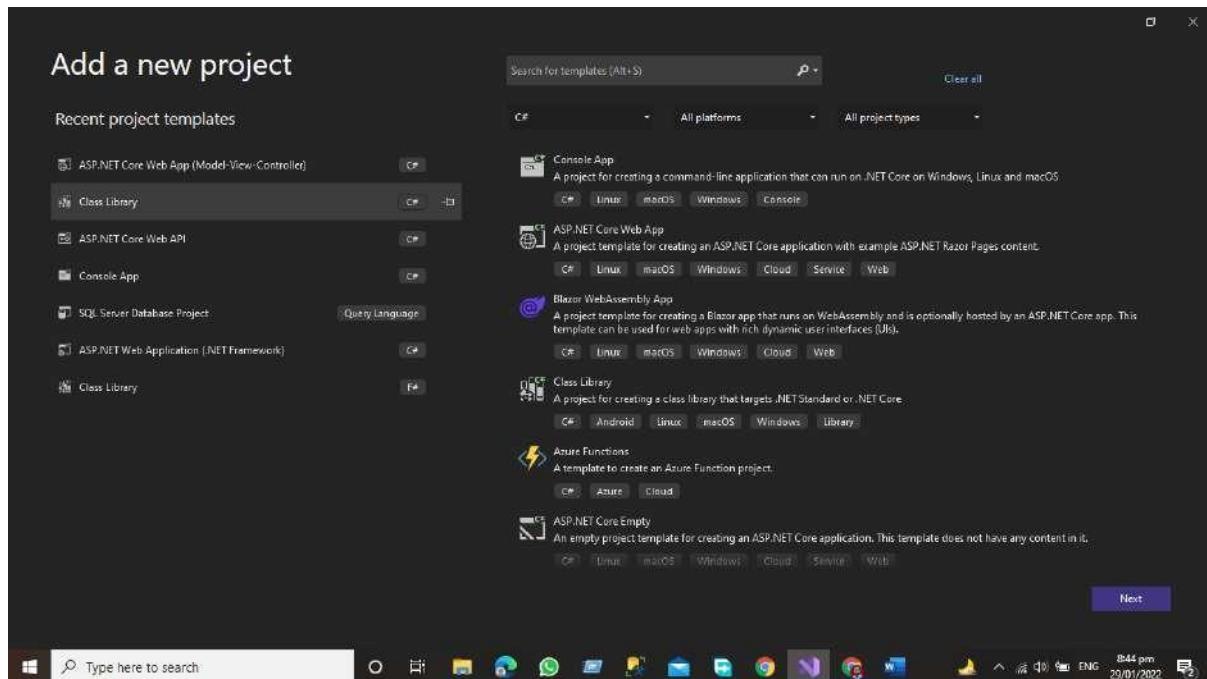


Steps to Follow for configuring these layer

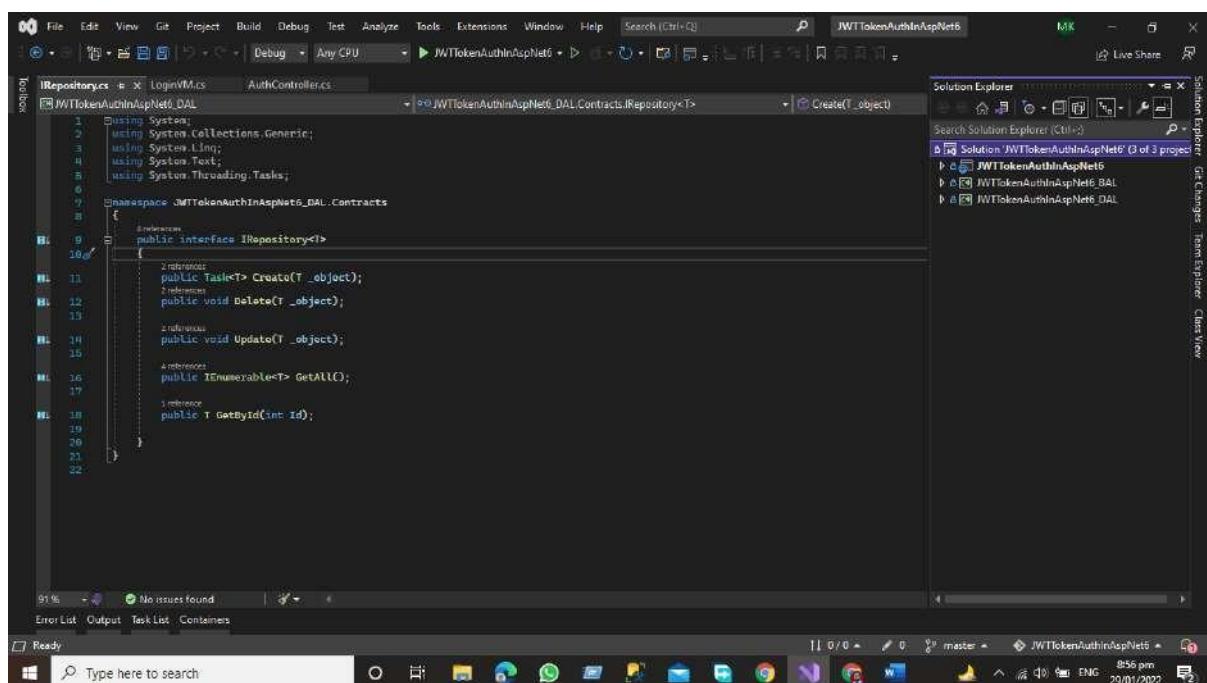
- Add the Class Library project of Asp.net for Data Access Layer
- Right Click on the project and then go to the add the new project window and then add the Asp.net Core class library project.



- After Adding the Data Access layer project now, we will add the Business access layer folder
- Add the Class library project of Asp.Net Core for Business Access
- Right Click on the project and then go to the add the new project window and then add the Asp.net Core class library project.



After adding the business access layer and data access layer, we must first add the references of the Data Access layer and Business Access layer, so that the project structure can be seen.

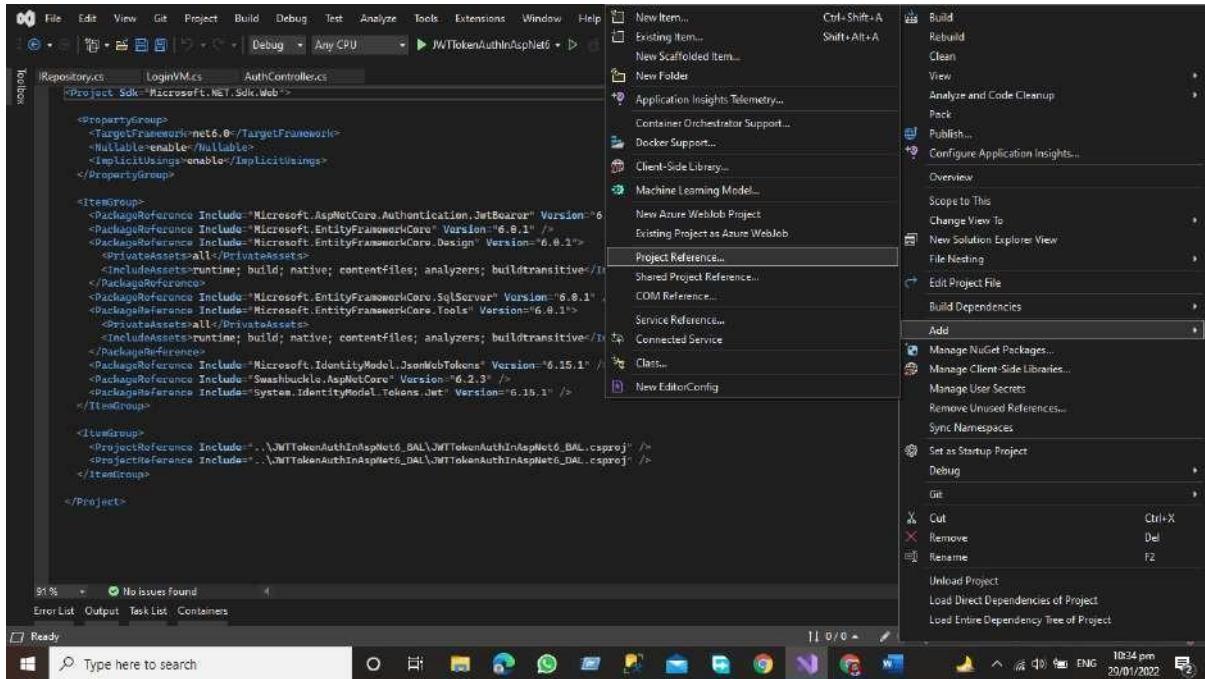


As you can see, our project now has a Presentation layer, Data Access Layer, and Business Access layer.

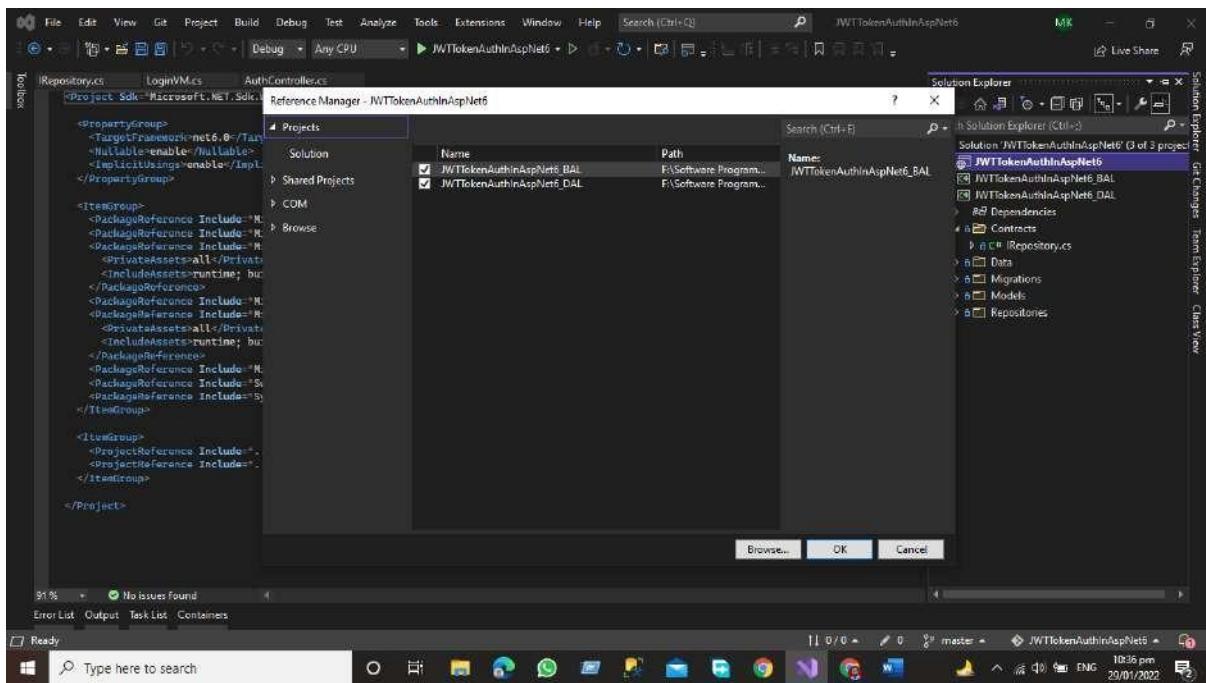
Add the References to the Project

Now, in the Presentation Layer, add the references to the Data Access Layer and the Business Access Layer.

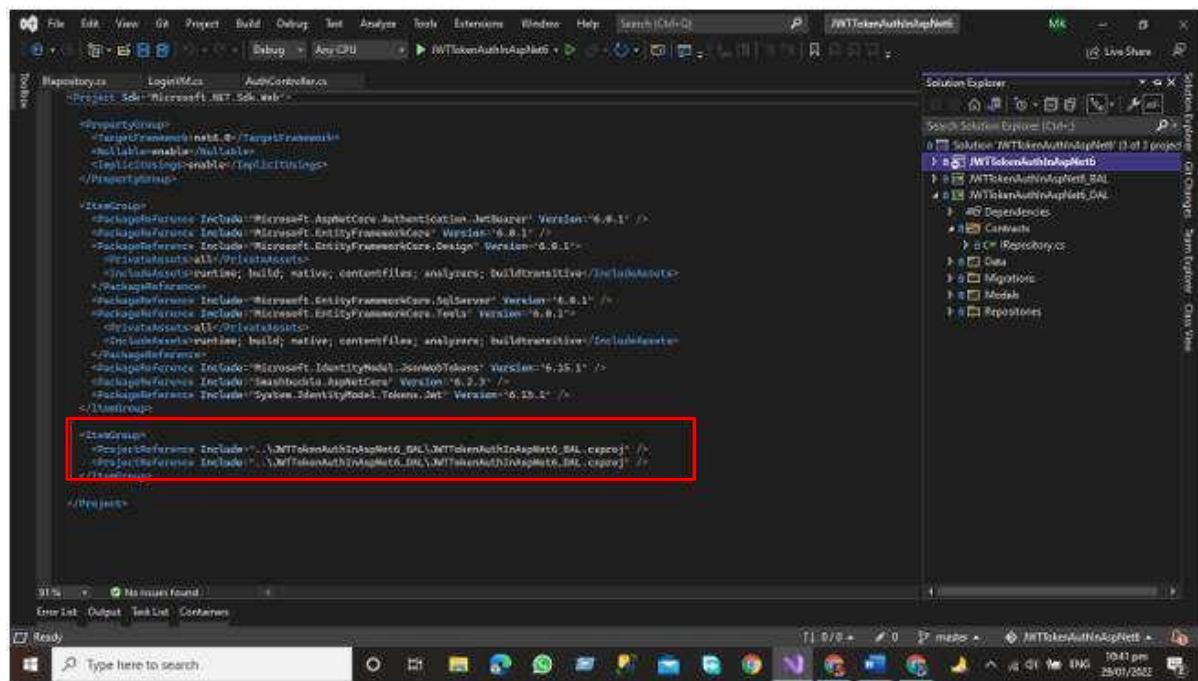
Right Click on the Presentation layer and then click on add => project Reference



Add the References by Checking both the checkboxes

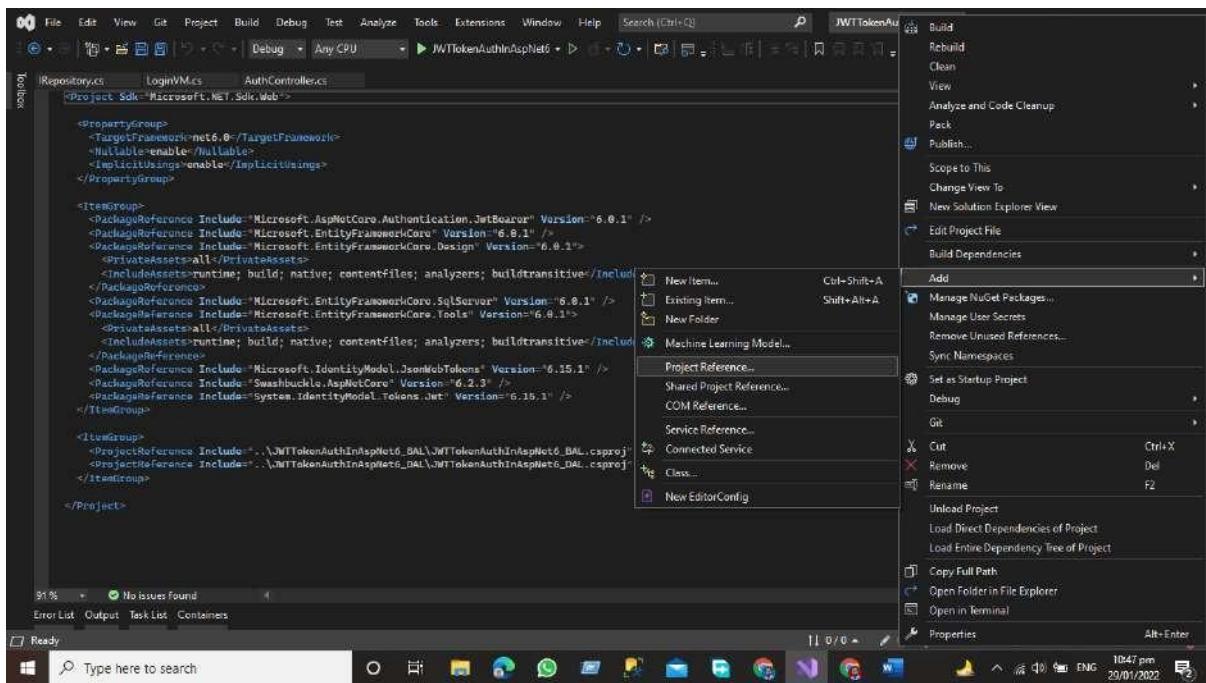


References have been added for both DAL and BAL

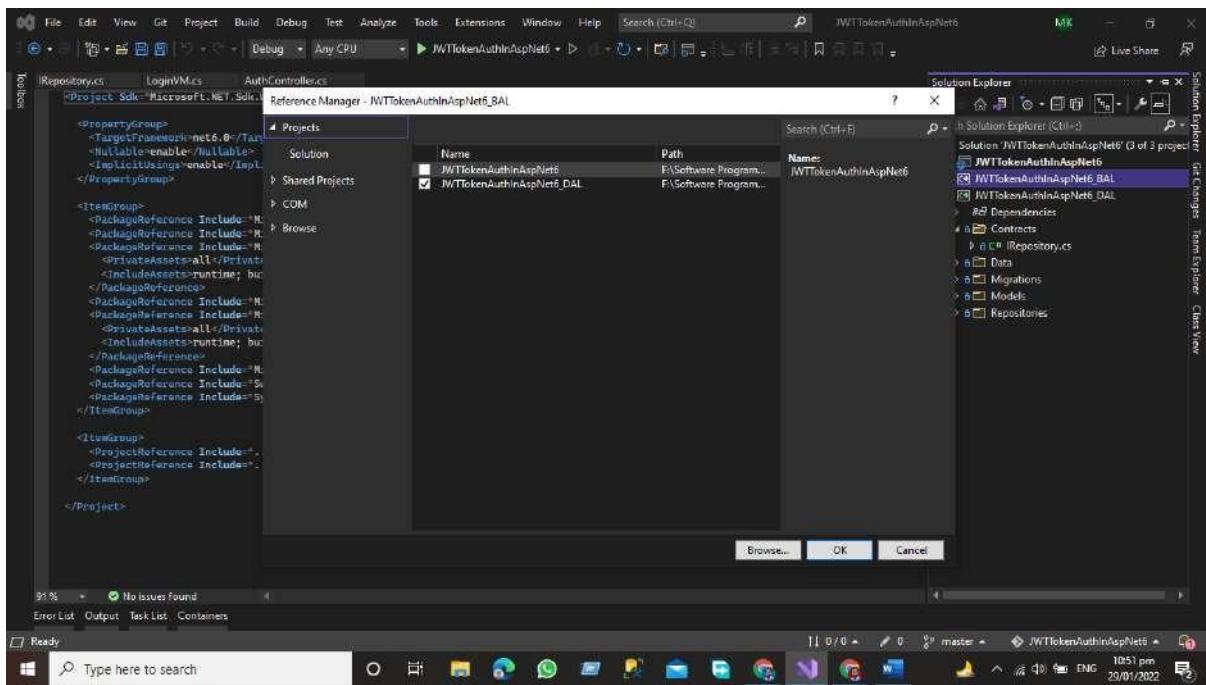


Now we'll add the Data Access layer project references to the Business layer.

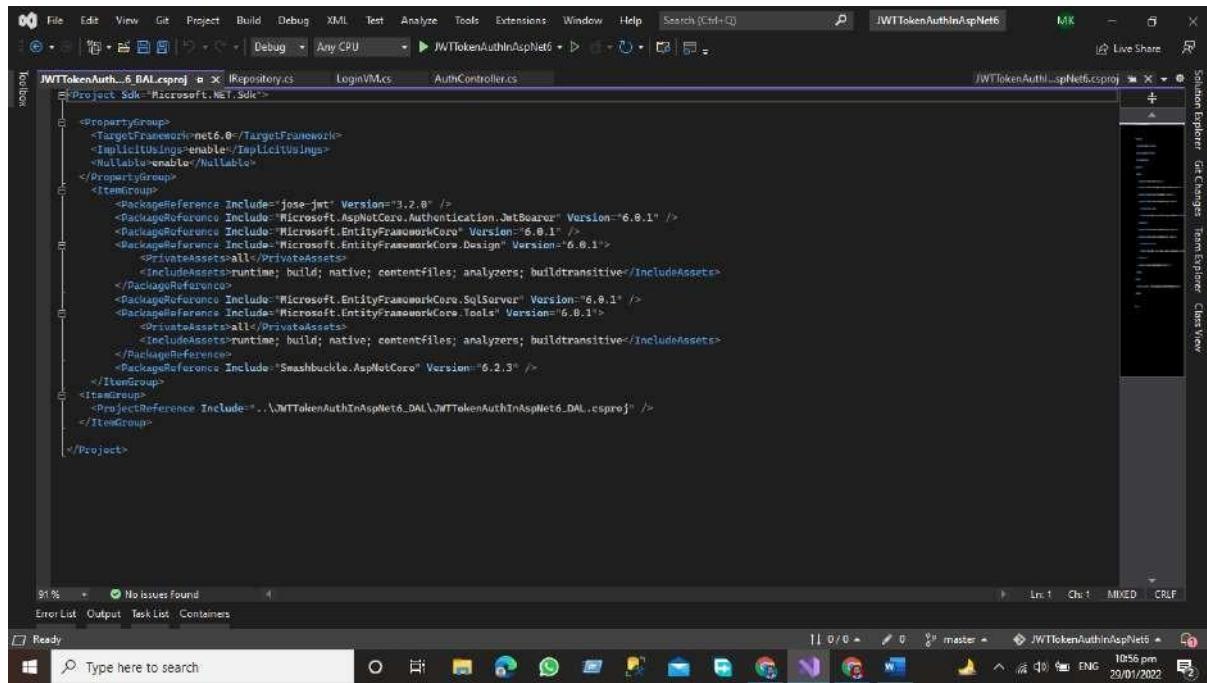
- Right Click on the Business access layer and then Click Add Button => Project References



“Remember that we have included the DAL and BAL references in the project, so don't add the Presentation layer references again in the business layer. We only need the DAL Layer references in the business layer”



Project References of DAL Has Been Added to BAL Layer



```
<PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
</PropertyGroup>
<ItemGroup>
    <PackageReference Include="jose-jwt" Version="3.2.0" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1" />
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PackageReference>
        <IncludeAssets>build; native</IncludeAssets>
        <PrivateAssets>all</PrivateAssets>
        <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Smashbuckle.AspNetCore" Version="6.2.3" />
</ItemGroup>
<ItemGroup>
    <ProjectReference Include=".\\JWTTokenAuthInAspNet6.DAL\\JWTTokenAuthInAspNet6.DAL.csproj" />
</ItemGroup>
</Project>
```

Data Access Layer

The Data Access Layer contains methods that assist the Business Access Layer in writing business logic, whether the functions are linked to accessing or manipulating data. The Data Access Layer's major goal is to interface with the database and the Business Access Layer in our project.

the structure will be like this.

In this Layer we have the Following Folders Add these folders to your Data access layer

- Contacts
- Data
- Migrations
- Models
- Repositories

Contacts

In the Contract Folder, we define the interface that has the following function that performs the desired functionalities with the database like

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

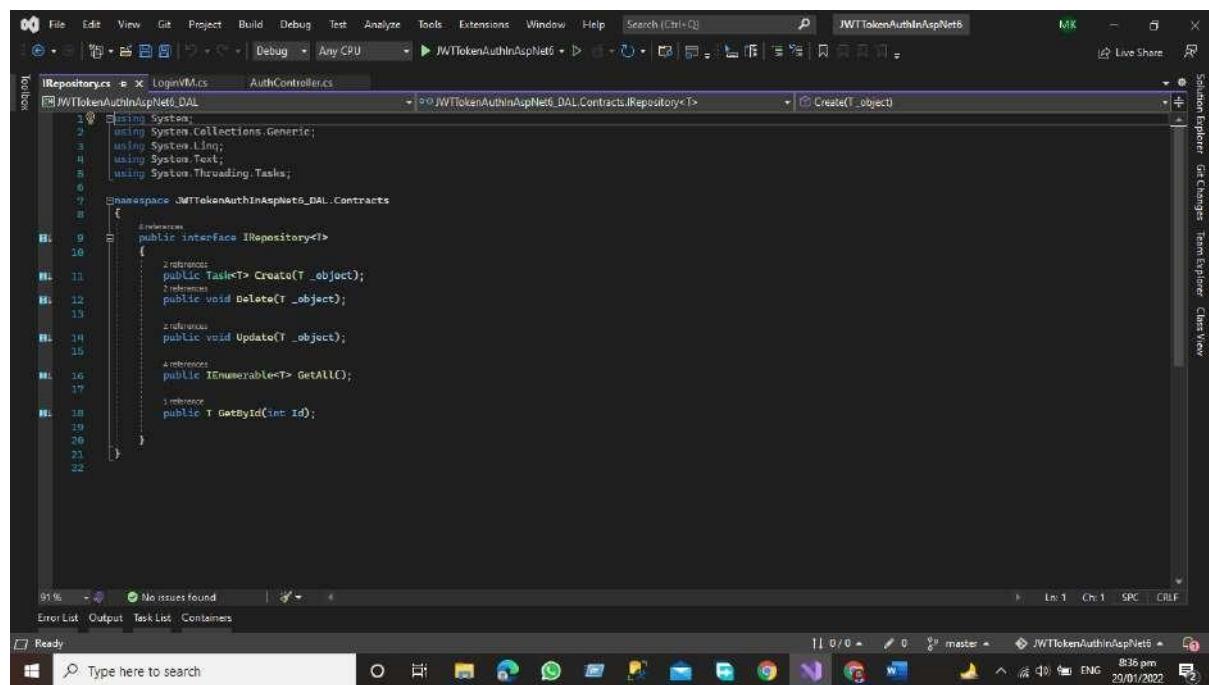
namespace JWTTokenAuthInAspNet6_DAL.Contracts
{
    public interface IRepository<T>
    {
        public Task<T> Create(T _object);
        public void Delete(T _object);

        public void Update(T _object);

        public IEnumerable<T> GetAll();

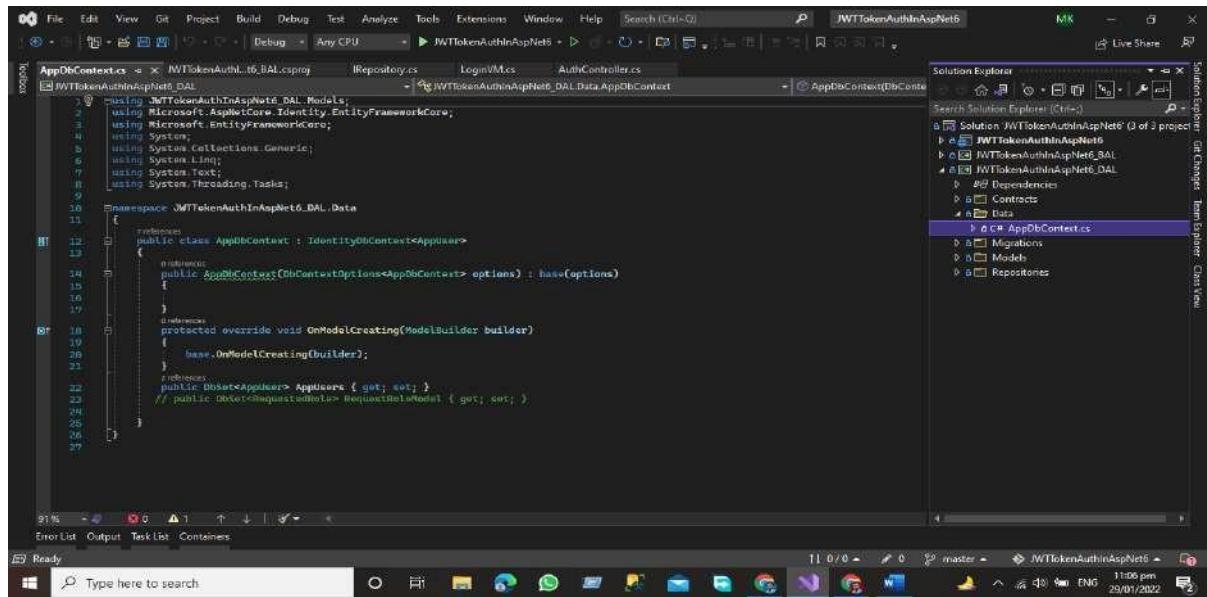
        public T GetById(int Id);
    }
}

```



Data

In the Data folder, we have Db Context Class this class is very important for accessing the data from the database.



The screenshot shows the Visual Studio IDE with the solution 'JWTTokenAuthInAspNet6' open. The 'AppDbContext.cs' file is selected in the Solution Explorer. The code in the editor defines a DbContext named 'AppDbContext' that inherits from 'IdentityDbContext<AppUser>'. It includes methods for overriding model creation and getting DbSet<AppUser>. The Solution Explorer also shows other projects like 'JWTTokenAuthInAspNet6' and 'JWTTokenAuthInAspNet6.DAL' along with their respective folders like Contracts, Data, Migrations, Models, and Repositories.

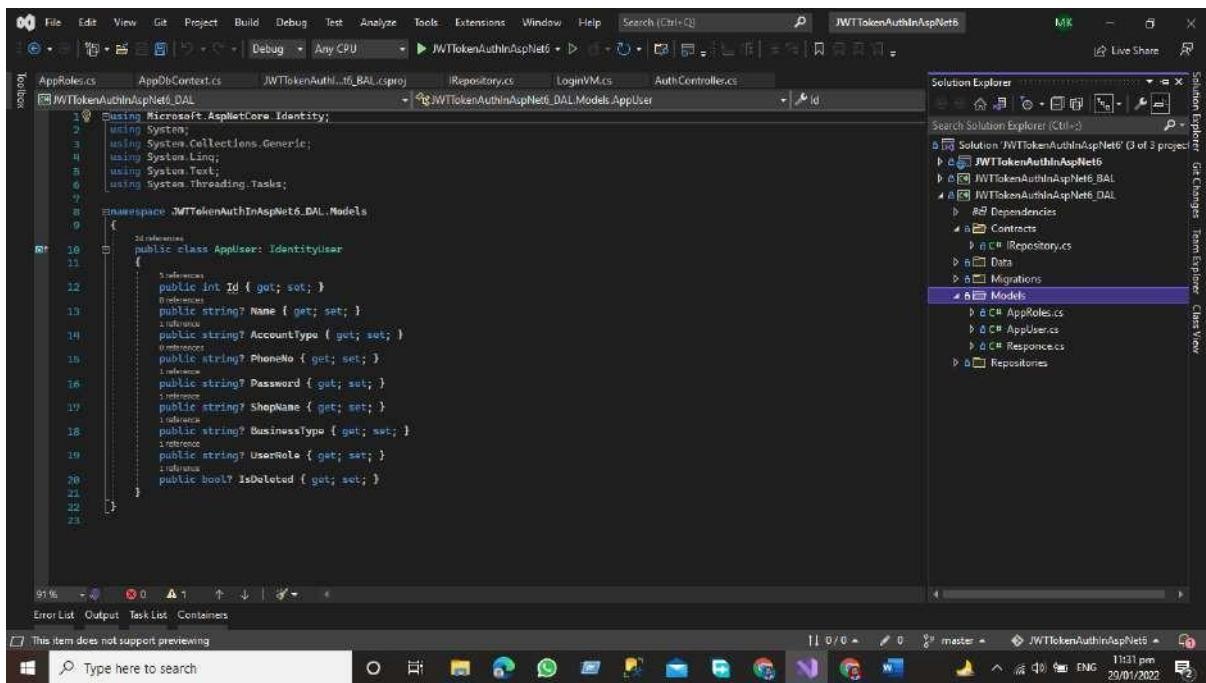
```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace JWTTokenAuthInAspNet6.DAL.Models
7  {
8      using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
9      using Microsoft.EntityFrameworkCore;
10     using System;
11     using System.Collections.Generic;
12     using System.Linq;
13     using System.Text;
14     using System.Threading.Tasks;
15
16     public class AppDbContext : IdentityDbContext<AppUser>
17     {
18         public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
19         {
20         }
21
22         protected override void OnModelCreating(ModelBuilder builder)
23         {
24             base.OnModelCreating(builder);
25         }
26
27         public DbSet<AppUser> AppUsers { get; set; }
28         // public DbSet<AppRequestedHolds> RequestedHolds { get; set; }
29     }
30 }
```

Migrations

The Migration Folder contains information on all the migrations we performed during the construction of this project. The Migration Folder contains information on all the migrations we performed during the construction of this project.

Models

Our application models, which contain domain-specific data, and business logic models, which represent the structure of the data as public attributes, business logic, and methods, are stored in the Model folder.



```

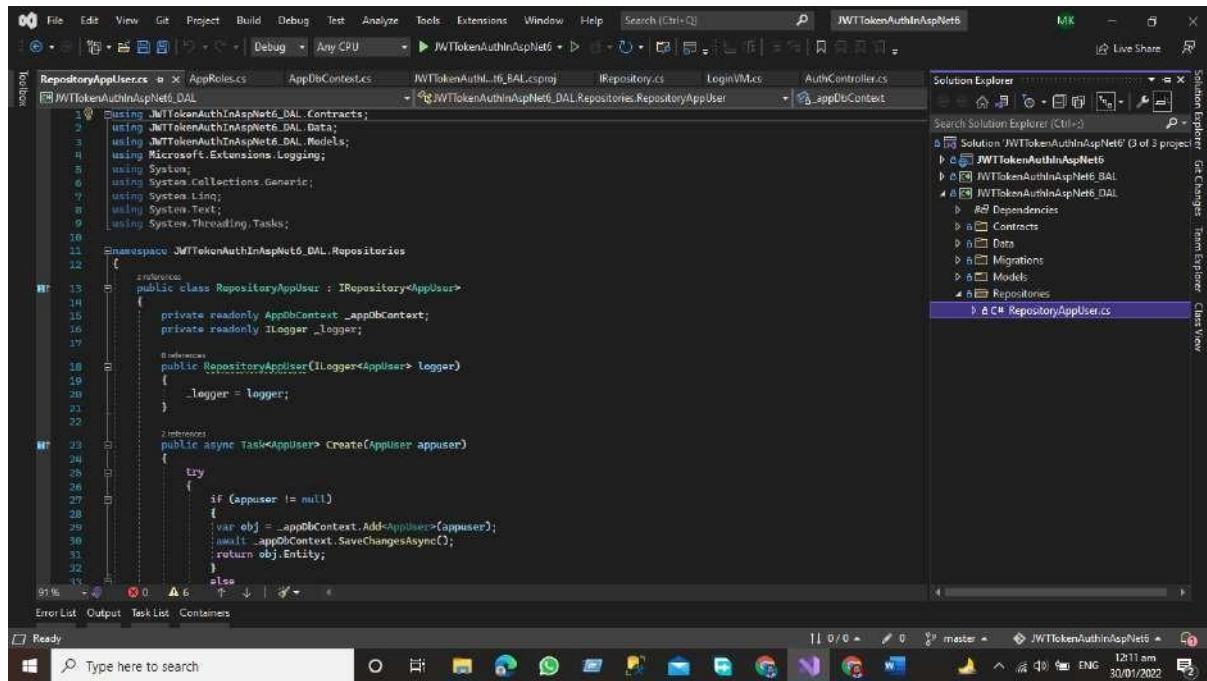
using Microsoft.AspNetCore.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6.DAL.Models
{
    public class AppUser: IdentityUser
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? AccountType { get; set; }
        public string? PhoneNo { get; set; }
        public string? Password { get; set; }
        public string? ShopName { get; set; }
        public string? BusinessType { get; set; }
        public string? UserRole { get; set; }
        public bool? IsDeleted { get; set; }
    }
}

```

Repository

In the Repository folder, we add the repository classes against each model. We write the CRUD function that communicates with the database using the entity framework. We add the repository class that inherits our Interface that is present in our contract folder.



```
using JWTTokenAuthInAspNet6.DAL.Contracts;
using JWTTokenAuthInAspNet6.DAL.Data;
using JWTTokenAuthInAspNet6.DAL.Models;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6.DAL.Repositories
{
    public class RepositoryAppUser : IRepository<AppUser>
    {
        private readonly AppDbContext _appDbContext;
        private readonly ILogger _logger;

        public RepositoryAppUser(ILogger<AppUser> logger)
        {
            _logger = logger;
        }

        public async Task<AppUser> Create(AppUser appuser)
        {
            try
            {
                if (appuser != null)
                {
                    var obj = _appDbContext.Add<AppUser>(appuser);
                    await _appDbContext.SaveChangesAsync();
                    return obj.Entity;
                }
                else
            }
        }
    }
}
```

```

public async Task<AppUser> Create(AppUser appuser)
{
    try
    {
        if (appuser != null)
        {
            var obj = _appDbContext.Add<AppUser>(appuser);
            await _appDbContext.SaveChangesAsync();
            return obj.Entity;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void Delete(AppUser appuser)
{
    try
    {
        if(appuser!=null)
        {
            var obj = _appDbContext.Remove(appuser);
            if (obj!=null)
            {
                _appDbContext.SaveChangesAsync();
            }
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public IEnumerable<AppUser> GetAll()

```

```

{
    try
    {
        var obj = _appDbContext.AppUsers.ToList();
        if (obj != null)
            return obj;
        else
            return null;
    }
    catch (Exception)
    {
        throw;
    }
}

public AppUser GetById(int Id)
{
    try
    {
        if(Id!=null)
        {
            var Obj = _appDbContext.AppUsers.FirstOrDefault(x => x.Id == Id);
            if (Obj != null)
                return Obj;
            else
                return null;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void Update(AppUser appuser)
{
    try
    {

```

```
if (appuser != null)
{
    var obj = _appDbContext.Update(appuser);
    if (obj != null)
        _appDbContext.SaveChanges();
}
}

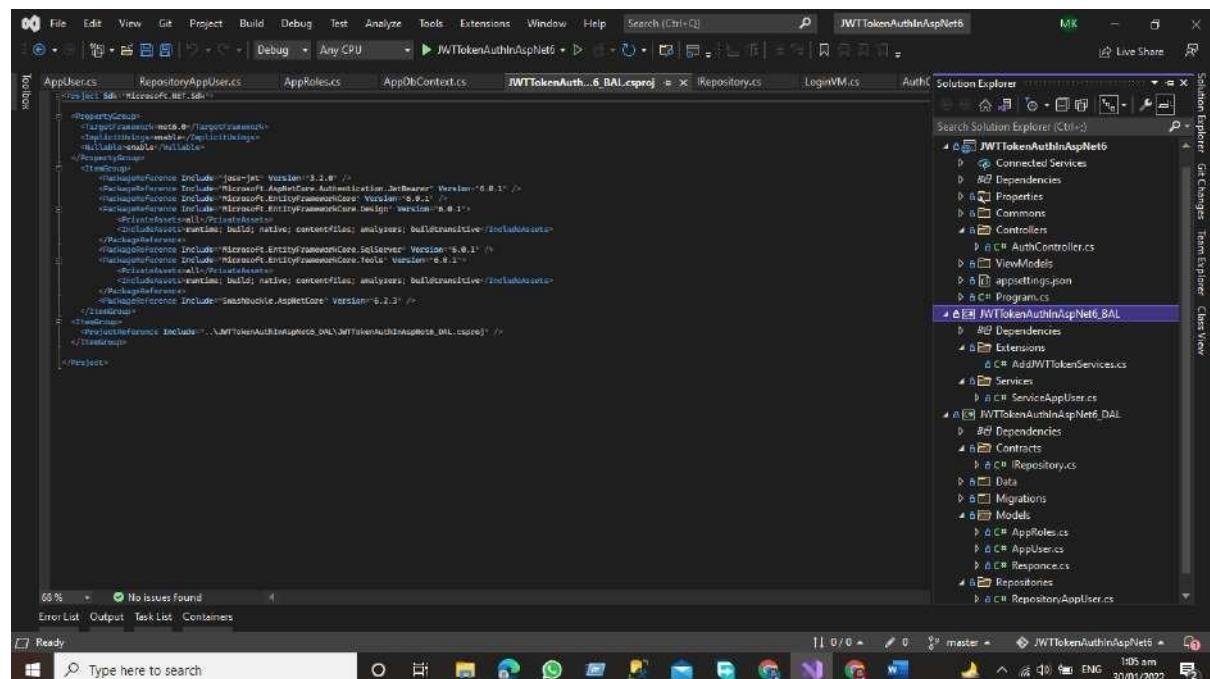
catch (Exception)
{
    throw;
}
}

}
```

Business Access Layer

The business layer communicates with the data access layer and the presentation layer logic layer process the actions that make the logical decision and evaluations the main function of this layer is to process the data between surrounding layers.

Our Structure of the project in the Business Access Layer will be like this.



In this layer, we will have two folders

- Extensions Method Folder
 - And Services Folder

In the business access layer, we have our services that communicate with the surrounding layers like the data layer and Presentation Layer.

Services

Services define the application's business logic. We develop services that interact with the data layer and move data from the data layer to the presentation layer and from the presentation layer to the data access layer.

Example

```
using JWTTOKENAuthInAspNet6_DAL.Contracts;
using JWTTOKENAuthInAspNet6_DAL.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTOKENAuthInAspNet6_BAL.Services
{
    public class ServiceAppUser
    {
        public readonly IRepository<AppUser> _repository;
        public ServiceAppUser(IRepository<AppUser> repository)
        {
            _repository = repository;
        }
        //Create Method
        public async Task<AppUser> AddUser(AppUser appUser)
        {
            try
            {
                if (appUser == null)
                {
                    throw new ArgumentNullException(nameof(appUser));
                }
                else
                {
                    return await _repository.Create(appUser);
                }
            }
            catch (Exception)
```

```

    {
        throw;
    }
}

public void DeleteUser(int Id)

{
    try
    {
        if(Id != 0)
        {
            var obj = _repository.GetAll().Where(x => x.Id == Id).FirstOrDefault();
            _repository.Delete(obj);
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void UpdateUser(int Id)

{
    try
    {
        if(Id != 0)
        {
            var obj = _repository.GetAll().Where(x => x.Id == Id).FirstOrDefault();
            if(obj != null)
                _repository.Update(obj);
        }
    }
    catch (Exception)
    {

        throw;
    }
}

```

```

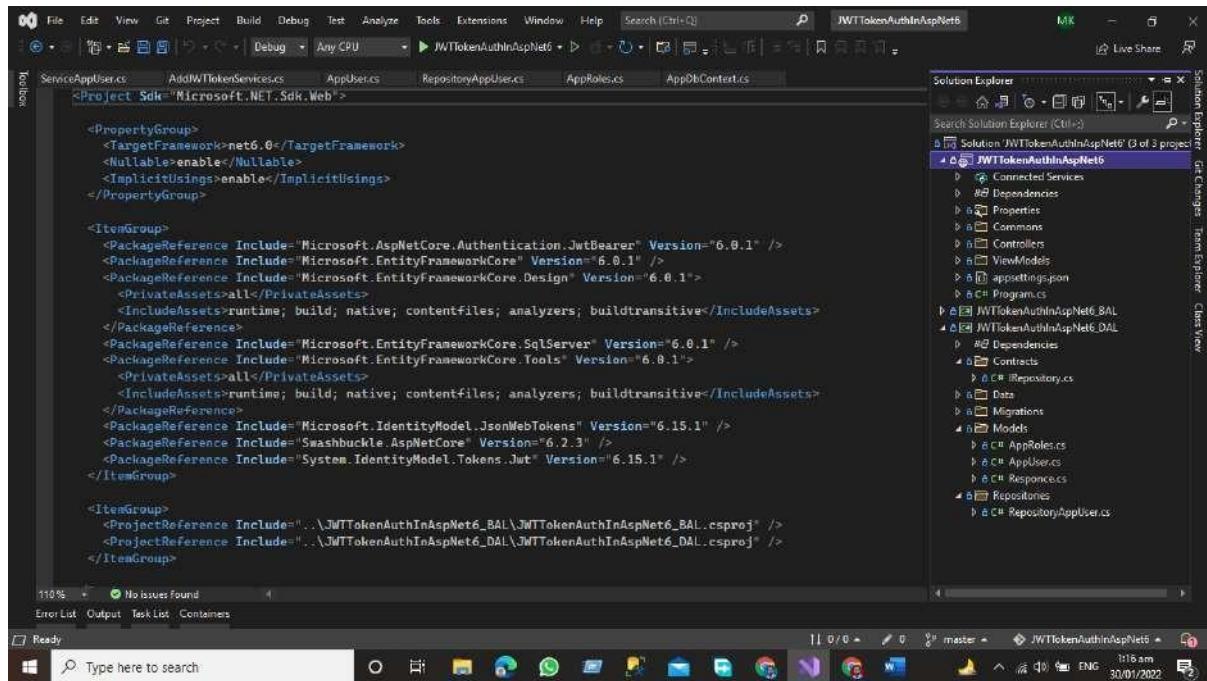
public IEnumerable<AppUser> GetAllUser()
{
    try
    {
        return _repository.GetAll().ToList();
    }
    catch (Exception)
    {
        throw;
    }
}
}

```

Presentation layer

The top-most layer of the 3-Tier architecture is the Presentation layer the main function of this layer is to give the results to the user or in simple words to present the data that we get from the business access layer and give the results to the front-end user.

In the presentation layer, we have the default template of the asp.net core application here OUR structure of the application will be like this.



The presentation layer is responsible to give the results to the user against every HTTP request. Here we have a controller that is responsible for handling the HTTP request and communicates with the business layer against each HTTP request get the get data from the associated layer and then present it to the User.

Advantages of 3-Tier Architecture

- It makes the logical separation between the presentation layer, business layer, and data layer.
- Migration to a new environment is fast.
- In This Model, each tier is independent so we can set the different developers on each tier for the fast development of applications
- Easy to maintain and understand the large-scale applications
- The business layer is in between the presentation layer and data layer the application is more secured because the client will not have direct access to the database.
- The process data that is sent from the business layer is validated at this level.
- The Posted data from the presentation layer will be validated at the business layer before Inserting or Updating the Database
- Database security can be provided at BAL Layer
- Define the business logic one in the BAL Layer and then share it among the other components at the presentation layer
- Easy to Apply for Object-oriented Concepts
- Easy to update the data provided quires at DAL Layer

Dis-Advantages of 3-Tier Architecture

- It takes a lot of time to build the small part of the application
- Need Good understanding of Object-Oriented programming

Conclusion

In this chapter, we have learned about 3-tier Architecture and how the three-layer exchange data between them how we can add the Data Access Layer and Business access layer in the project and studied how these layers communicate with each other.

Presentation Layer (PL)

The top-most layer of the 3-Tier architecture is the Presentation layer the main function of this layer is to give the results to the user or in simple words to present the data that we get from the business access layer and give the results to the front-end user.

Business Access Layer (BAL)

The logic layer communicates with the data access layer and the presentation layer logic layer process the actions that make the logical decision and evaluations the main function of this layer is to process the data between surrounding layers.

Data Access Layer (DAL)

The main function of this application is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.

Complete Git Hub Project URL

[Click here for complete project access](#)

Chapter 3- API Development in Asp.Net Using Onion Architecture

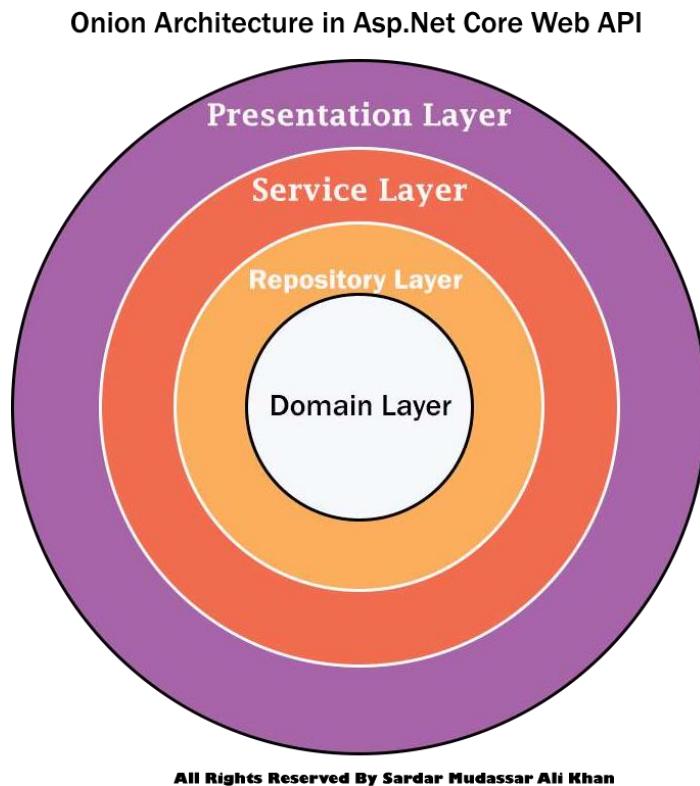
- ✓ Introduction
- ✓ What is Onion architecture
- ✓ Layers in Onion Architecture
- ✓ Domain Layer
- ✓ Repository Layer
- ✓ Service Layer
- ✓ Presentation Layer
- ✓ Advantages of Onion Architecture
- ✓ Implementation of Onion Architecture
- ✓ Project Structure.
- ✓ Domain Layer
- ✓ Models Folder
- ✓ Data Folder
- ✓ Repository Layer
- ✓ Service Layer
- ✓ I Custom Service
- ✓ Custom Service
- ✓ Department Service
- ✓ Student Service
- ✓ Result Service
- ✓ Subject GPA Service
- ✓ Presentation Layer
- ✓ Dependency Injection
- ✓ Modify Program.cs File
- ✓ Controllers
- ✓ Output
- ✓ Conclusion

Introduction

In this chapter, we will cover the onion architecture using the Asp.Net 6 Web API. Onion architecture term introduced by Jeffrey Palermo in 2008 this architecture provides us a better way to build applications using this architecture our applications are better testable maintainable and dependable on infrastructures like databases and services. Onion architecture solves common problems like coupling and separation of concerns.

What is Onion architecture

In onion architecture, we have the domain layer, repository layer, service layer, and presentation layer. Onion architecture solves the problem that we face during the enterprise applications like coupling and separations of concerns. Onion architecture also solves the problem that we confront in three-tier architecture and N-Layer architecture. In Onion architecture, our layer communicates with each other using interfaces.



Layers in Onion Architecture

Onion architecture uses the concept of the layer, but it is different from N-layer architecture and 3-Tier architecture.

Domain Layer

This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

Repository Layer

The repository layer act as a middle layer between the service layer and model objects we will maintain all the database migrations and database context Object in this layer. We will add the interfaces that consist the of data access pattern for reading and writing operations with the database.

Service Layer

This layer is used to communicate with the presentation and repository layer. The service layer holds all the business logic of the entity. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Presentation Layer

In the case of the API Presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

Advantages of Onion Architecture

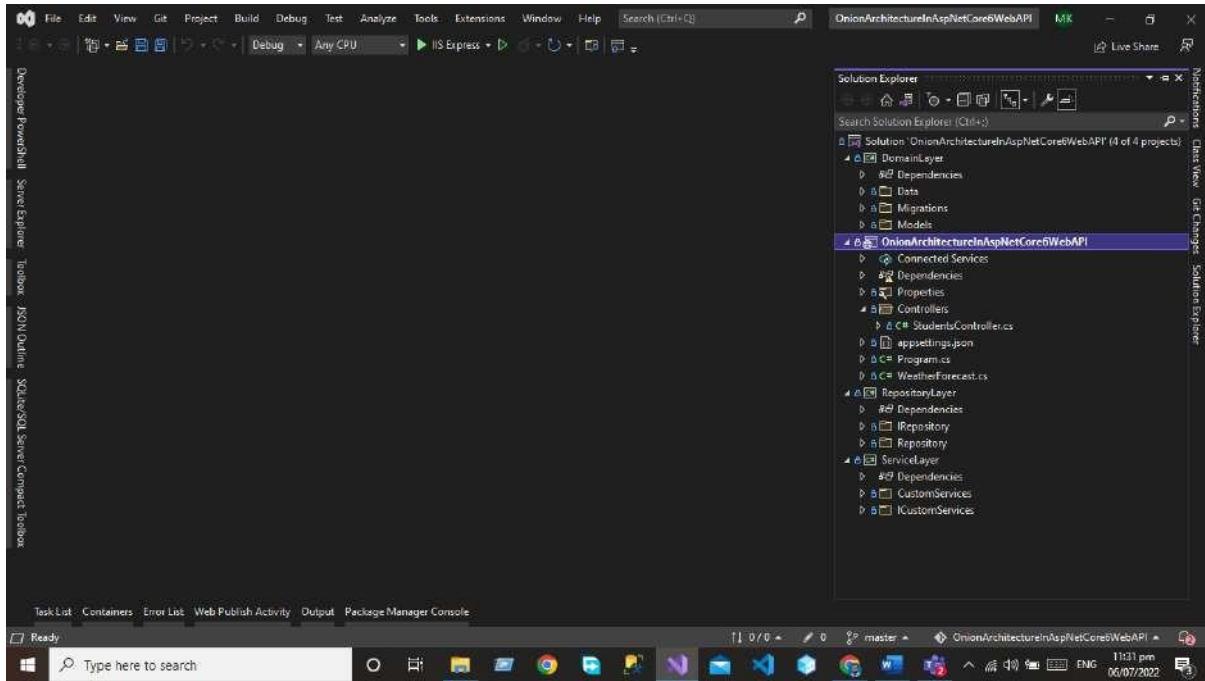
- Onion architecture provides us with the better maintainability of code because code depends on layers.
- It provides us better testability for unit tests we can write the separate test cases in layers without affecting the other module in the application.
- Using the onion architecture our application is loosely coupled because our layers communicate with each other using the interface.
- Domain entities are the core and center of the architecture and have access to databases and UI Layer.
- A Complete implementation would be provided to the application at run time.
- The external layer never depends on the external layer.

Implementation of Onion Architecture

Now we are going to develop the project using the Onion Architecture

First, you need to create the Asp.net Core web API project using visual studio. After creating the project, we will add our layer to the project after adding all the layers our project structure will look like this.

Project Structure.

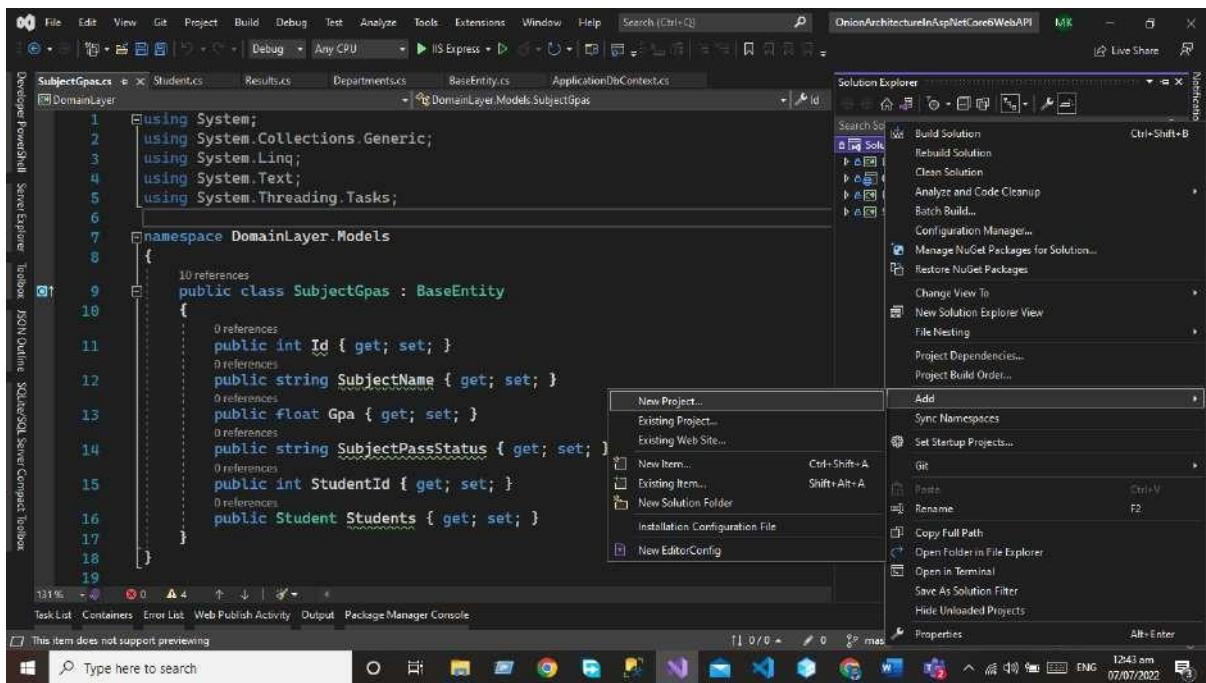


Domain Layer

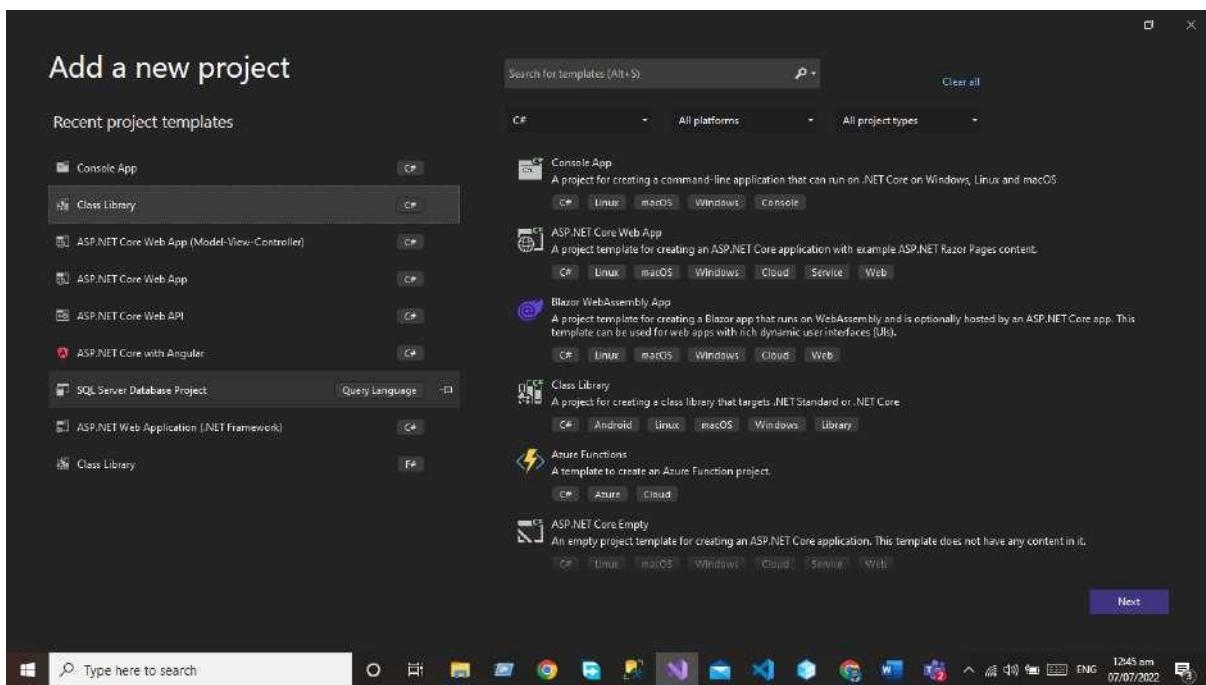
This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

For the Domain layer, we need to add the library project to our application.

Right click on the Solution and then click on add option.



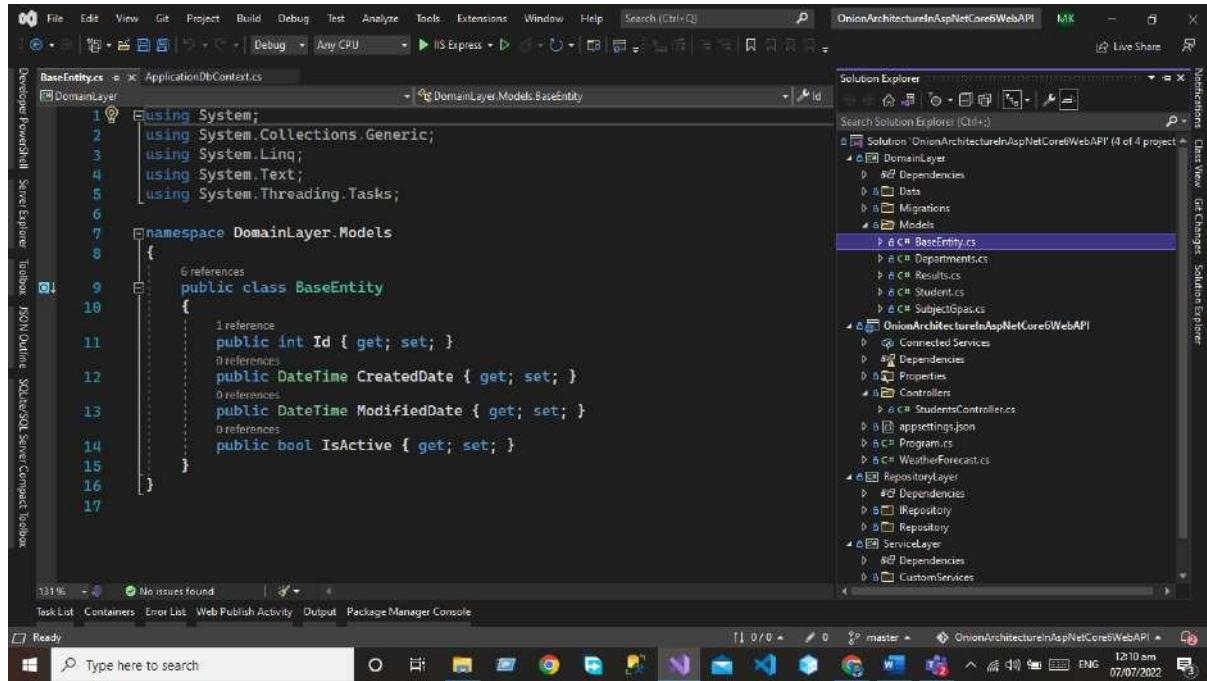
Add the library project to your solution



Name this project as Domain Layer

Models Folder

First, you need to add the Models folder that will be used to create the database entities. In the Models folder, we will create the following database entities.



Base Entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class BaseEntity
    {
        public int Id { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime ModifiedDate { get; set; }
        public bool IsActive { get; set; }
    }
}
```

Department

Department entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Departments : BaseEntity
    {
        public int Id { get; set; }
        public string DepartmentName { get; set; }
        public int StudentId { get; set; }
        public Student Students { get; set; }
    }
}
```

Result

Result Entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Resultss : BaseEntity
    {
        [Key]
        public int Id { get; set; }
        public string? ResultStatus { get; set; }
        public int StudentId { get; set; }
        public Student Students { get; set; }
    }
}
```

Students

Student Entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Student : BaseEntity
    {
        [Key]
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Address { get; set; }
        public string? Emial { get; set; }
        public string? City { get; set; }
        public string? State { get; set; }
        public string? Country { get; set; }
        public int? Age { get; set; }
        public DateTime? BirthDate { get; set; }

    }
}
```

SubjectGpa

Subject GPA Entity will extend the Base Entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class SubjectGpas : BaseEntity
    {
        public int Id { get; set; }
        public string SubjectName { get; set; }
    }
}
```

```

    public float Gpa { get; set; }
    public string SubjectPassStatus { get; set; }
    public int StudentId { get; set; }
    public Student Students { get; set; }
}
}

```

Data Folder

Add the Data in the domain that is used to add the database context class. The database context class is used to maintain the session with the underlying database using which you can perform the CRUD operation.

Write click on the application and create the class ApplicationDbContext.cs

```

using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
        {

        }

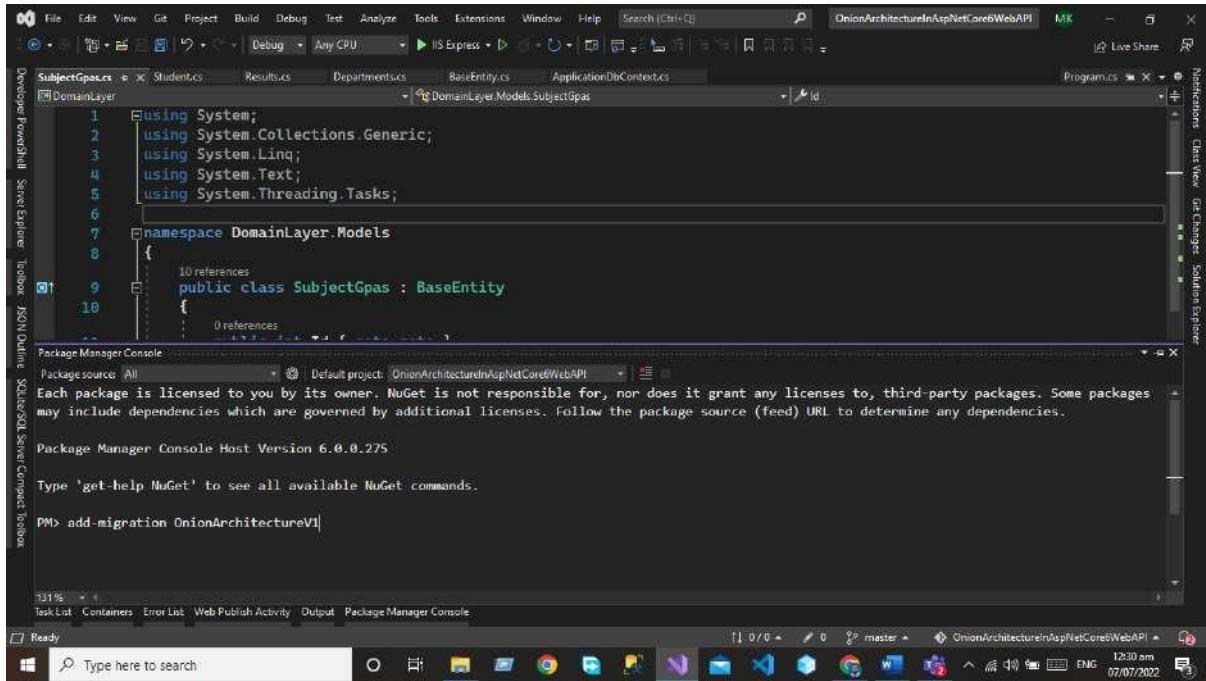
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Departments> Departments { get; set; }
        public DbSet<SubjectGpas> SubjectGpas { get; set; }
        public DbSet<Results> Results { get; set; }
    }
}

```

After Adding the DB Set properties we need to add the migration using the package manager console and run the command Add-Migration.

```
add-migration OnionArchitectureV1
```

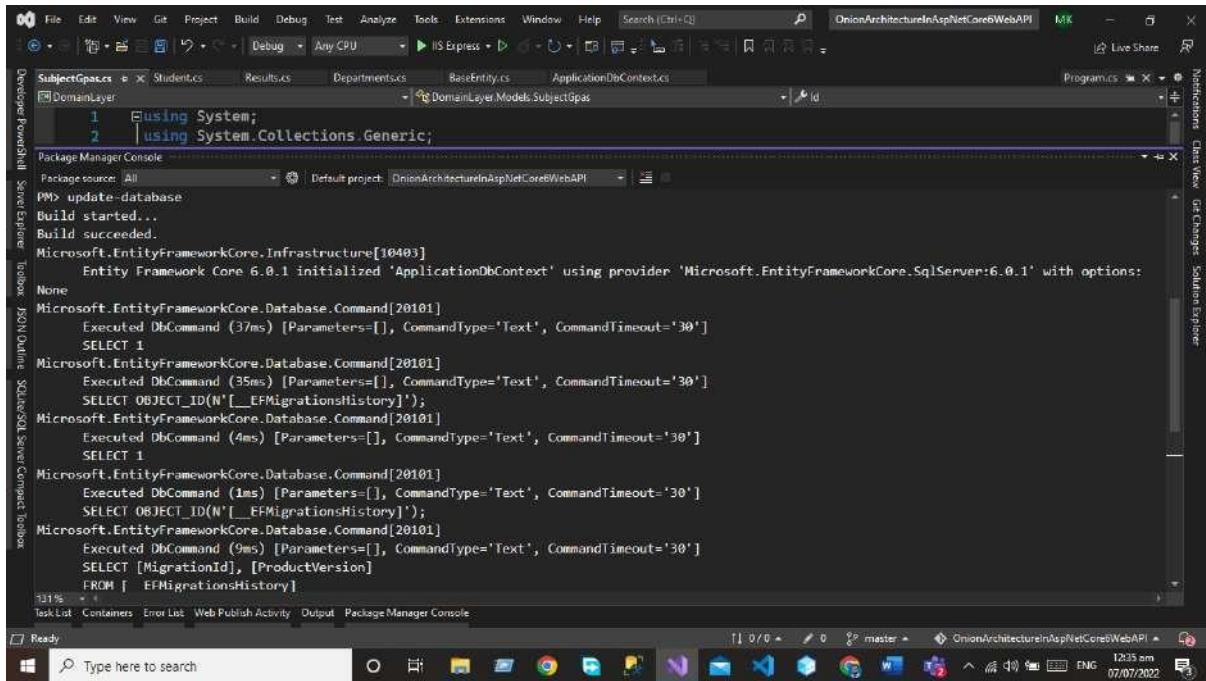


The screenshot shows the Visual Studio IDE. In the top navigation bar, the project 'OnionArchitectureInAspNetCoreWebAPI' is selected. The code editor displays a C# file named 'SubjectGpas.cs' which contains the following code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DomainLayer.Models
8  {
9      public class SubjectGpas : BaseEntity
10  }
```

Below the code editor is the 'Package Manager Console' window. It shows the command 'PM> add-migration OnionArchitectureV1' entered and executed. The output window displays the message: 'Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.'

After executing the commands now, you need to update the database by executing the **update-database** Command



The screenshot shows the Visual Studio IDE. The 'Package Manager Console' window now displays the output of the 'PM> update-database' command. The output includes logs from Entity Framework Core, such as 'Build started...', 'Build succeeded.', and various database migration logs like 'Microsoft.EntityFrameworkCore.Infrastructure[10403]' and 'Microsoft.EntityFrameworkCore.Database.Command[20101]'. The command 'SELECT 1' is shown being executed multiple times.

Migration Folder

After executing both commands now you can see the migration folder in the domain layer.

The screenshot shows the Microsoft Visual Studio interface. The code editor displays a C# class named `SubjectGpas.cs` located in the `DomainLayer/Models` namespace. The class inherits from `BaseEntity.cs` and contains properties for `Id`, `SubjectName`, `Gpa`, `SubjectPassStatus`, `StudentId`, and a collection of `Students`. The Solution Explorer on the right shows the project structure with four main layers: `DomainLayer`, `RepositoryLayer`, `ServiceLayer`, and `API Layer`. The `Migrations` folder under `DomainLayer` is selected. The status bar at the bottom indicates the file path `OnionArchitectureInAspNetCore6WebAPI/DomainLayer/Models/SubjectGpas.cs`, the branch `master`, and the date `07/07/2022 12:36 am`.

```

1  Using System;
2  Using System.Collections.Generic;
3  Using System.Linq;
4  Using System.Text;
5  Using System.Threading.Tasks;
6
7  Namespace DomainLayer.Models
8  {
9      public class SubjectGpas : BaseEntity
10     {
11         public int Id { get; set; }
12         public string SubjectName { get; set; }
13         public float Gpa { get; set; }
14         public string SubjectPassStatus { get; set; }
15         public int StudentId { get; set; }
16         public Student Students { get; set; }
17     }
18 }
19

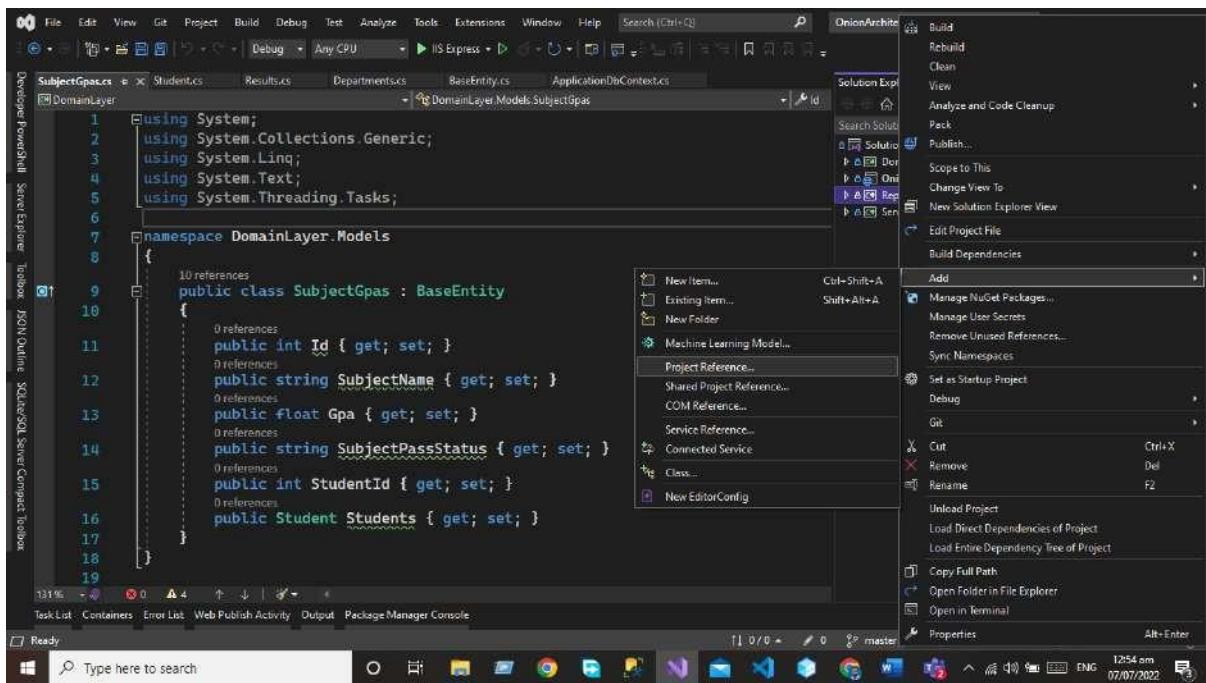
```

Repository Layer

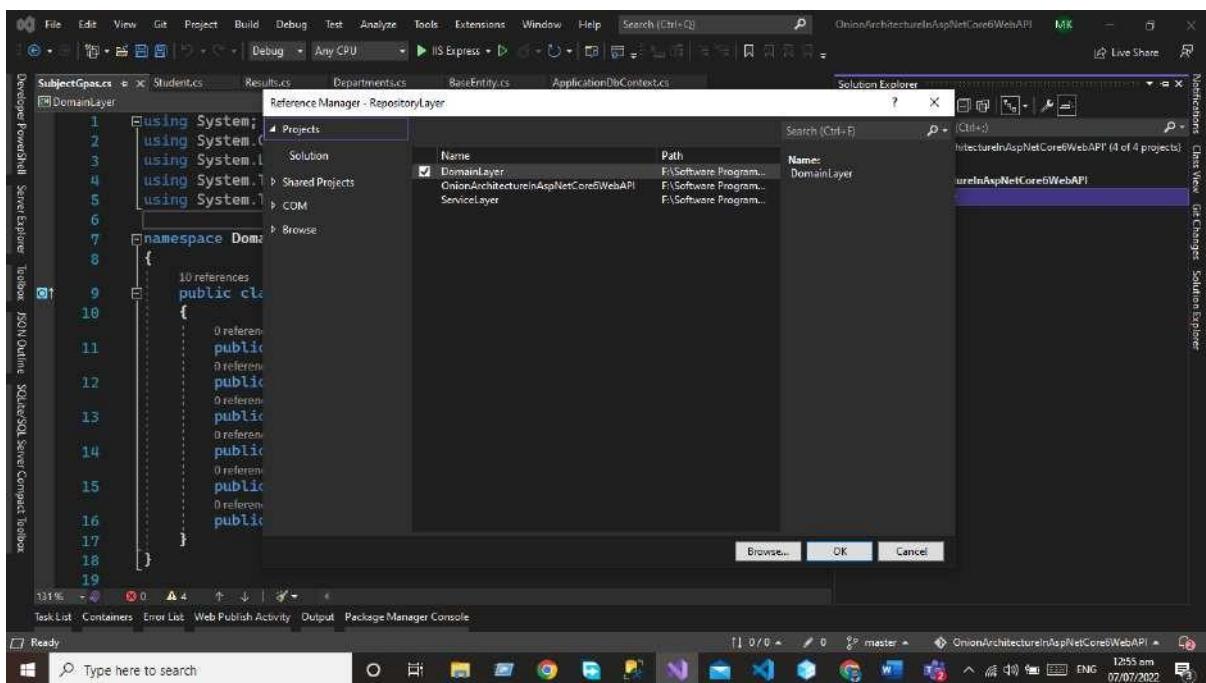
The repository layer act as a middle layer between the service layer and model objects we will maintain all the database migrations and database context Object in this layer. We will add the interfaces that consist the of data access pattern for reading and writing operations with the database.

Now we will work on Repository Layer we will follow the same project as we did for the Domain layer add the library project in your applicant and give a name to that project Repository layer.

But here we need to add the project reference of the Domain layer in the repository layer. Write click on the project and then click the Add button after that we will add the project references in the Repository layer.



Click on project reference now and select the Domain layer.



Now we need to add the two folders.

- I Repository
- Repository

IRepository

IRepository folder contains the generic interface using this interface we will create the CRUD operation for our all entities.

Generic Interface will extend the Base-Entity for using some properties. The Code of the generic interface is given below.

```
using DomainLayer.Models;
```

```
namespace RepositoryLayer.IRepository
{
    public interface IRepository<T> where T: BaseEntity
    {
        IEnumerable<T> GetAll();
        T Get(int Id);
        void Insert(T entity);
        void Update(T entity);
        void Delete(T entity);
        void Remove(T entity);
        void SaveChanges();
    }
}
```

Repository

Now we create the Generic repository that extends the Generic IRepository Interface. The Code of the generic repository is given below.

```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RepositoryLayer.Repository
{
    public class Repository<T> : IRepository<T> where T : BaseEntity
    {
        #region property
        private readonly ApplicationDbContext _applicationDbContext;
```

```

private DbSet<T> entities;
#endregion

#region Constructor
public Repository(ApplicationDbContext applicationDbContext)
{
    _applicationDbContext = applicationDbContext;
    entities = _applicationDbContext.Set<T>();
}
#endregion

public void Delete(T entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
    entities.Remove(entity);
    _applicationDbContext.SaveChanges();
}

public T Get(int Id)
{
    return entities.SingleOrDefault(c => c.Id == Id);
}

public IEnumerable<T> GetAll()
{
    return entities.AsEnumerable();
}

public void Insert(T entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
    entities.Add(entity);
    _applicationDbContext.SaveChanges();
}

public void Remove(T entity)

```

```

{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
    entities.Remove(entity);
}

public void SaveChanges()
{
    _applicationDbContext.SaveChanges();
}

public void Update(T entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
    entities.Update(entity);
    _applicationDbContext.SaveChanges();
}

}
}

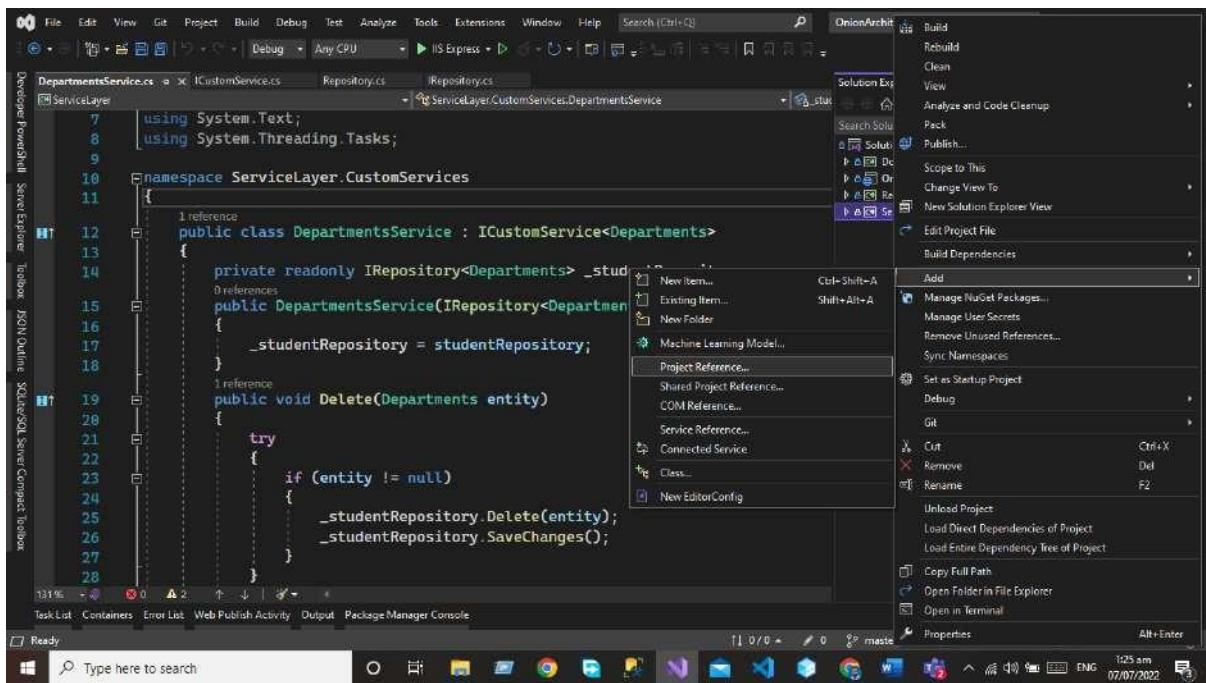
```

We will use this repository in our service layer.

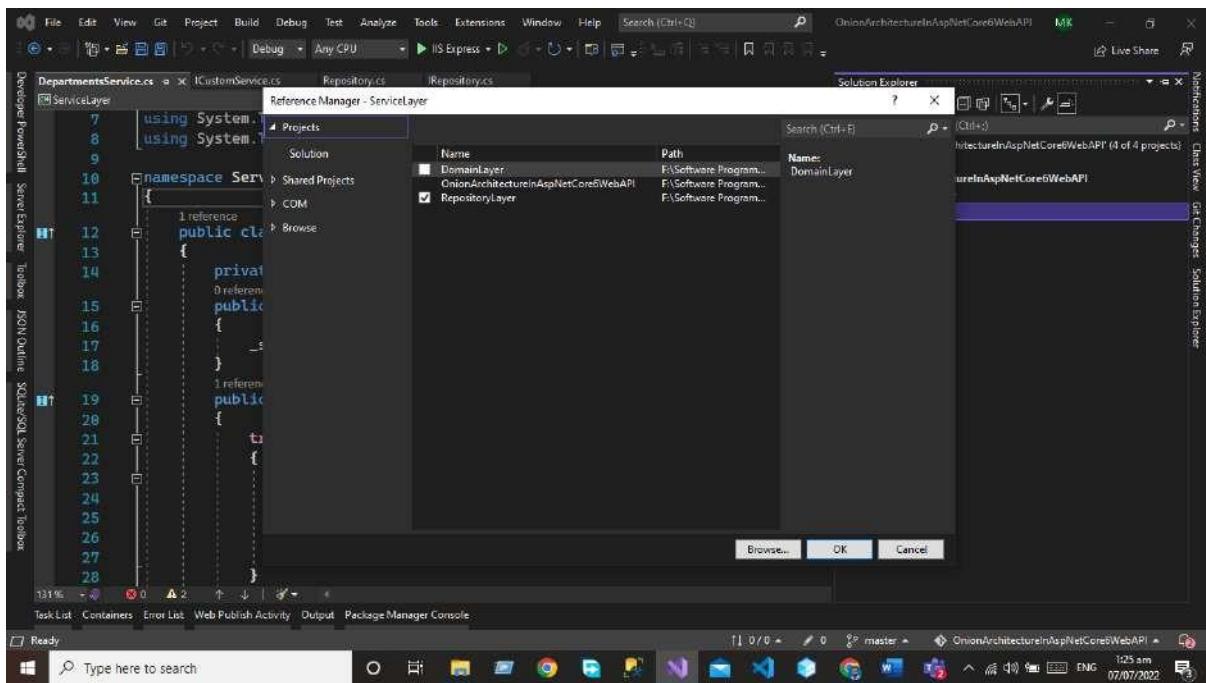
Service Layer

This layer is used to communicate with the presentation and repository layer. The service layer holds all the business logic of the entity. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Now we need to add a new project to our solution that will be the service layer we will follow the same process for adding the library project in our application but here we need some extra work after adding the project we need to add the reference of the **Repository Layer**.



Now our service layer contains the reference of the repository layer.



In the Service layer, we will create the two folders.

- ICustomServices
- CustomServices

ICustom Service

Now in the ICustomServices folder, we will create the ICustomServices Interface this interface holds the signature of the method we will implement these methods in the customs service code of the ICustomServices Interface given below.

```
using DomainLayer.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.ICustomServices
{
    public interface ICustomService<T> where T : class
    {
        IEnumerable<T> GetAll();
        T Get(int Id);
        void Insert(T entity);
        void Update(T entity);
        void Delete(T entity);
        void Remove(T entity);

    }
}
```

Custom Service

In the custom service folder, we will create the custom service class that inherits the ICustomService interface code of the custom service class given below. We will write the crud for our all entities.

For every service, we will write the CRUD operation using our generic repository.

Department Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class DepartmentsService : ICustomService<Departments>
    {
        private readonly IRepository<Departments> _studentRepository;
        public DepartmentsService(IRepository<Departments> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {
                throw;
            }
        }

        public Departments Get(int Id)
        {
            try
            {
                var obj = _studentRepository.Get(Id);
                if (obj != null)
                {
                    return obj;
                }
                else
                {
                    return null;
                }
            }
        }
    }
}

```

```

        }

        catch (Exception)
        {

            throw;
        }
    }

public IEnumerable<Departments> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Insert(Departments entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {
}

```

```

        throw;
    }
}

public void Remove(Departments entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Remove(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Update(Departments entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Update(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}
}

```

Student Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class StudentService : ICustomService<Student>
    {
        private readonly IRepository<Student> _studentRepository;
        public StudentService(IRepository<Student> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Student entity)
        {
            try
            {
                if(entity!=null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {
                throw;
            }
        }

        public Student Get(int Id)
        {
            try
            {
                var obj = _studentRepository.Get(Id);
                if(obj!=null)
```

```

        {
            return obj;
        }
        else
        {
            return null;
        }

    }
    catch (Exception)
    {

        throw;
    }
}

public IEnumerable<Student> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Insert(Student entity)
{
    try
    {
        if(entity != null)

```

```

        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }

    catch (Exception)
    {
        throw;
    }
}

public void Remove(Student entity)
{
    try
    {
        if(entity!=null)
        {
            _studentRepository.Remove(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void Update(Student entity)
{
    try
    {
        if(entity!=null)
        {
            _studentRepository.Update(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {
        throw;
    }
}

```

```

        }
    }
}
}

Result Service
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class ResultService : ICustomService<Resultss>
    {
        private readonly IRepository<Resultss> _studentRepository;
        public ResultService(IRepository<Resultss> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Resultss entity)
        {
            try
            {
                if(entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {
                throw;
            }
        }

        public Resultss Get(int Id)
    }
}

```

```

    {
        try
        {
            var obj = _studentRepository.Get(Id);
            if (obj != null)
            {
                return obj;
            }
            else
            {
                return null;
            }
        }
        catch (Exception)
        {

            throw;
        }
    }

    public IEnumerable<Resultss> GetAll()
    {
        try
        {
            var obj = _studentRepository.GetAll();
            if (obj != null)
            {
                return obj;
            }
            else
            {
                return null;
            }
        }
        catch (Exception)
        {

            throw;
        }
    }

```

```

public void Insert(Resultss entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Remove(Resultss entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Remove(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Update(Resultss entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Update(entity);
            _studentRepository.SaveChanges();
        }
    }
}

```

```
    }  
    catch (Exception)  
    {  
  
        throw;  
    }  
}  
}  
}
```

Subject GPA Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class SubjectGpasService : ICustomService<SubjectGpas>
    {
        private readonly IRepository<SubjectGpas> _studentRepository;
        public SubjectGpasService(IRepository<SubjectGpas> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(SubjectGpas entity)
        {
            try
            {
                if(entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

```

```

        throw;
    }
}

public SubjectGpas Get(int Id)
{
    try
    {
        var obj = _studentRepository.Get(Id);
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public IEnumerable<SubjectGpas> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

```

```

        throw;
    }

}

public void Insert(SubjectGpas entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Remove(SubjectGpas entity)
{
    try
    {
        if(entity != null)
        {
            _studentRepository.Remove(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Update(SubjectGpas entity)
{
    try
    {

```

```
if (entity != null)
{
    _studentRepository.Update(entity);
    _studentRepository.SaveChanges();
}

}

catch (Exception)
{
    throw;
}

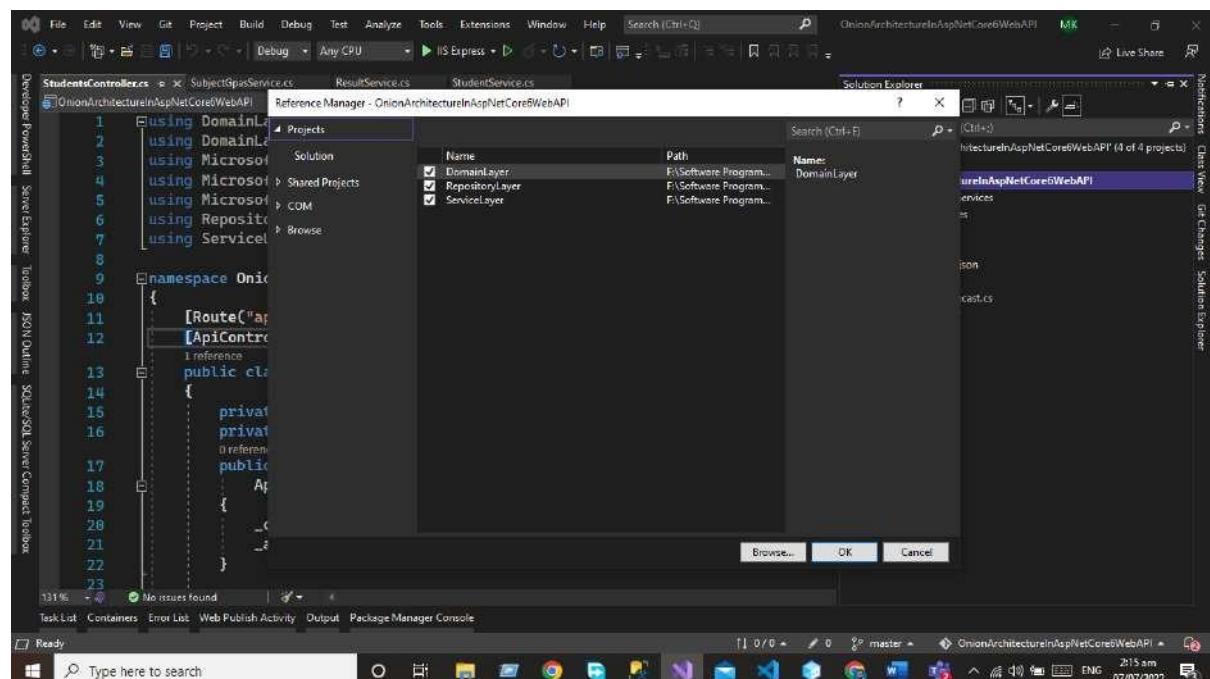
}
}
```

Presentation Layer

The presentation layer is our final layer that presents the data to the front-end user on every HTTP request.

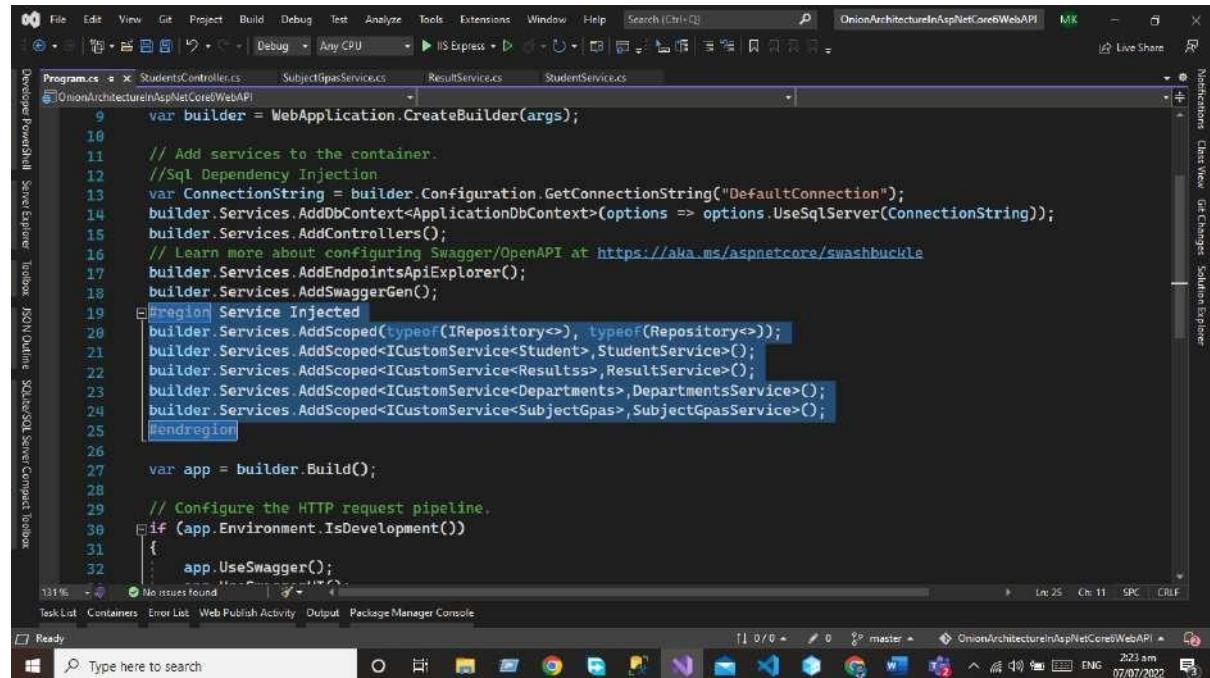
In the case of the API presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIs.

The presentation layer is the default Asp.net core web API project Now we need to add the project references of all the layers as we did before



Dependency Injection

Now we need to add the dependency Injection of our all services in the program.cs class



```
9 var builder = WebApplication.CreateBuilder(args);
10 // Add services to the container.
11 //Sql Dependency Injection
12 var ConnectionString = builder.Configuration.GetConnectionString("DefaultConnection");
13 builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(ConnectionString));
14 builder.Services.AddControllers();
15 // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
16 builder.Services.AddEndpointsApiExplorer();
17 builder.Services.AddSwaggerGen();
18 #region Service Injected
19 builder.Services.AddScoped(typeof IRepository<>, typeof Repository<>);
20 builder.Services.AddScoped<ICustomService<Student>, StudentService>();
21 builder.Services.AddScoped<ICustomService<Resultss>, ResultService>();
22 builder.Services.AddScoped<ICustomService<Departments>, DepartmentsService>();
23 builder.Services.AddScoped<ICustomService<SubjectGpas>, SubjectGpasService>();
24 #endregion
25
26 var app = builder.Build();
27
28 // Configure the HTTP request pipeline.
29 if (app.Environment.IsDevelopment())
30 {
31     app.UseSwagger();
32 }
```

Modify Program.cs File

The Code of the Startup Class is Given below

```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using RepositoryLayer.Repository;
using ServiceLayer.CustomServices;
using ServiceLayer.ICustomServices;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
//Sql Dependency Injection
var ConnectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(ConnectionString));
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
#region Service Injected
```

```

builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped<ICustomService<Student>, StudentService>();
builder.Services.AddScoped<ICustomService<Resultss>, ResultService>();
builder.Services.AddScoped<ICustomService<Departments>, DepartmentsService>();
builder.Services.AddScoped<ICustomService<SubjectGpas>, SubjectGpasService>();
#endregion

var app = builder.Build();

// Configure the HTTP request pipeline.
if(app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

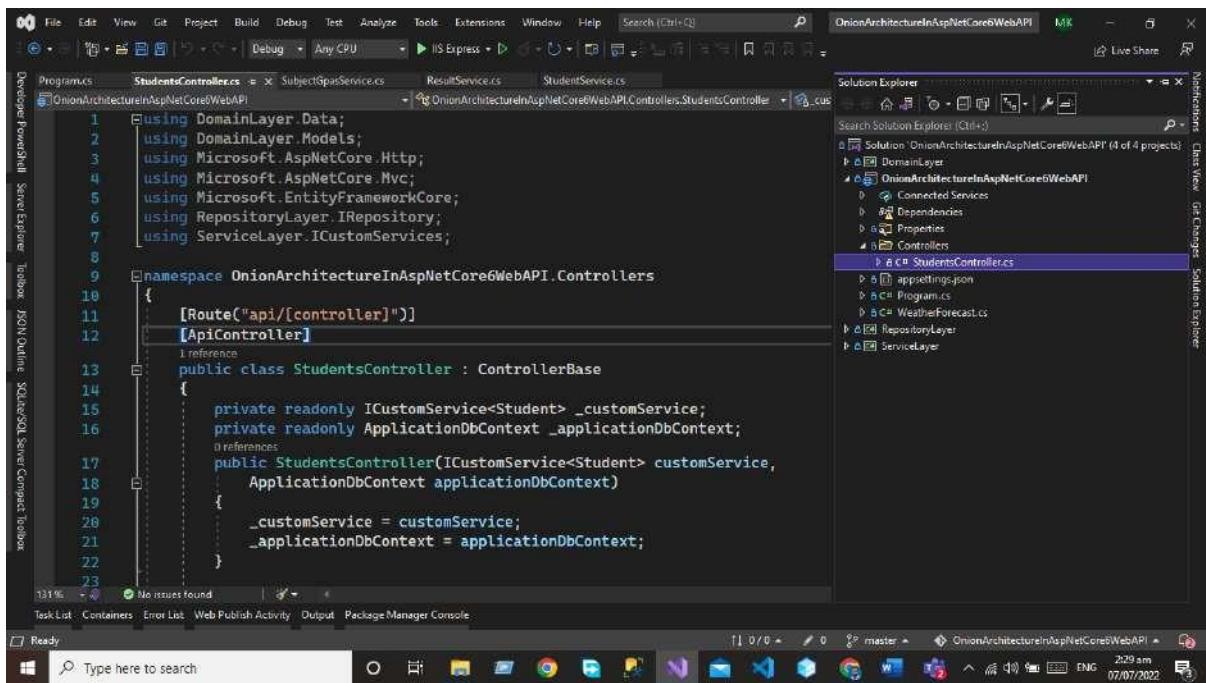
app.MapControllers();

app.Run();

```

Controllers

Controllers are used to handle the HTTP request. Now we need to add the student controller that will interact with our service layer and display the data to the users.



The Code of the Controller is given below.

```

using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;

namespace OnionArchitectureInAspNetCore6WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StudentsController : ControllerBase
    {
        private readonly ICustomService<Student> _customService;
        private readonly ApplicationDbContext _applicationDbContext;
        public StudentsController(ICustomService<Student> customService,
            ApplicationDbContext applicationDbContext)
        {
            _customService = customService;
            _applicationDbContext = applicationDbContext;
        }

        [HttpGet(nameof(GetStudentById))]
    }
}

```

```

public IActionResult GetStudentById(int Id)
{
    var obj = _customService.Get(Id);
    if (obj == null)
    {
        return NotFound();
    }
    else
    {
        return Ok(obj);
    }
}

[HttpGet(nameof(GetAllStudent))]
public IActionResult GetAllStudent()
{
    var obj = _customService.GetAll();
    if(obj == null)
    {
        return NotFound();
    }
    else
    {
        return Ok(obj);
    }
}

[HttpPost(nameof(CreateStudent))]
public IActionResult CreateStudent(Student student)
{
    if (student!=null)
    {
        _customService.Insert(student);
        return Ok("Created Successfully");
    }
    else
    {
        return BadRequest("Something went wrong");
    }
}

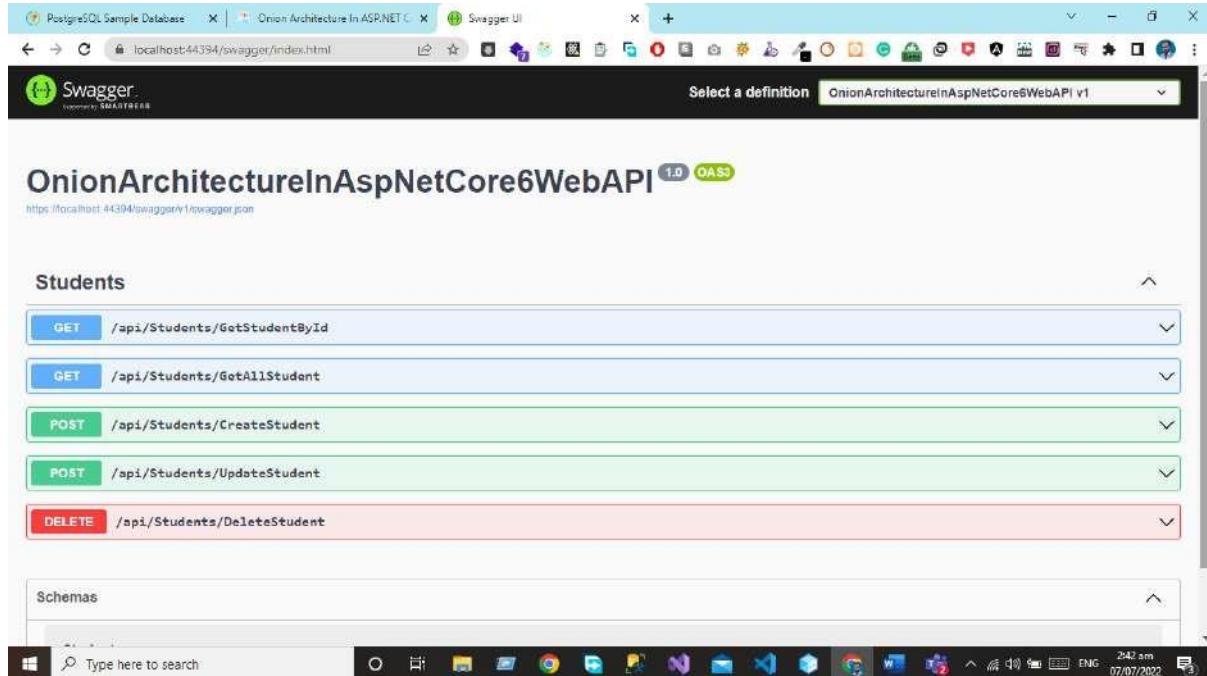
[HttpPost(nameof(UpdateStudent))]
public IActionResult UpdateStudent(Student student)
{
    if(student!=null)

```

```
{  
    _customService.Update(student);  
    return Ok("Updated SuccessFully");  
}  
  
else  
{  
    return BadRequest();  
}  
  
}  
  
[HttpDelete(nameof>DeleteStudent))]  
public IActionResult DeleteStudent(Student student)  
{  
    if(student!=null)  
    {  
        _customService.Delete(student);  
        return Ok("Deleted Successfully");  
    }  
    else  
    {  
        return BadRequest("Something went wrong");  
    }  
}  
}  
}
```

Output

Now we will run the project and will see the output using the swagger.

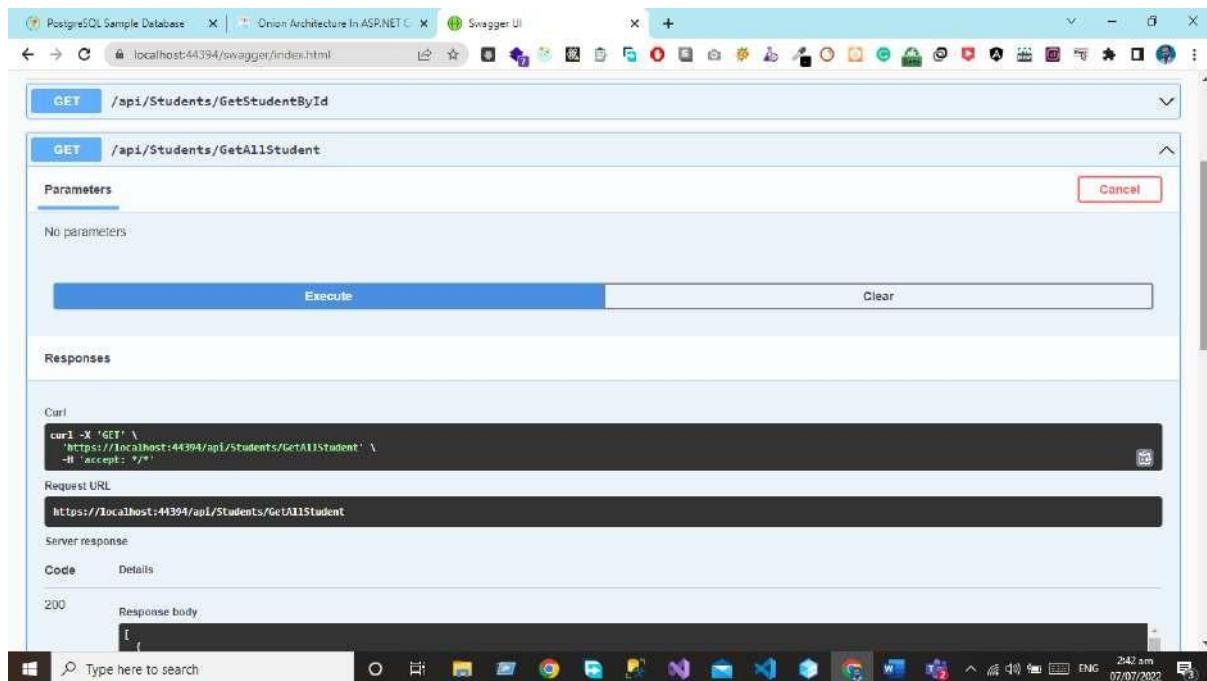


The screenshot shows the Swagger UI interface for the `OnionArchitectureInAspNetCore6WebAPI` project. The top navigation bar includes tabs for `PostgreSQL Sample Database`, `Onion Architecture In ASP.NET Core`, and `Swagger UI`. The main content area is titled `OnionArchitectureInAspNetCore6WebAPI` (v1) and `OAS3`. Below this, the `Students` endpoint group is shown with the following operations:

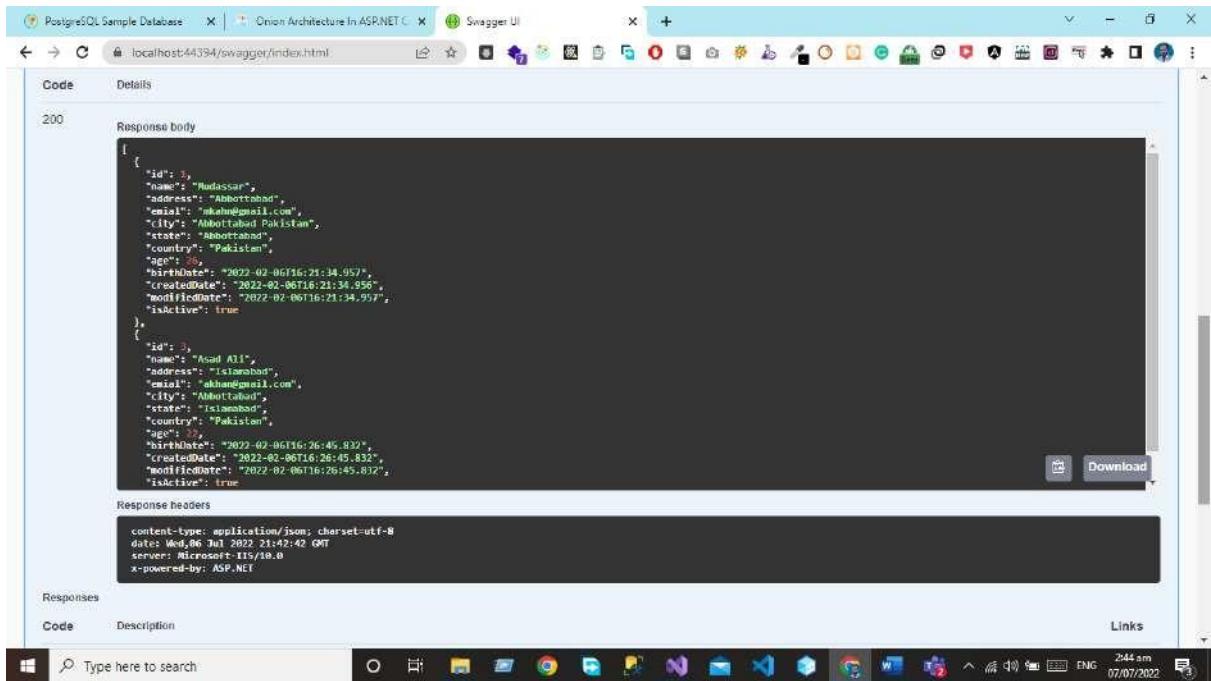
- `GET /api/Students/GetStudentById`
- `GET /api/Students/GetAllStudent` (highlighted in blue)
- `POST /api/Students/CreateStudent`
- `POST /api/Students/UpdateStudent`
- `DELETE /api/Students/DeleteStudent`

Below the operations, there is a section for `Schemas`. At the bottom of the interface is a Windows taskbar with various pinned icons and a search bar.

Now we can see when we hit the GetAllStudent Endpoint we can see the data of students from the database in the form of JSON projects.



The screenshot shows the execution details for the `GET /api/Students/GetAllStudent` endpoint. The `Parameters` section indicates "No parameters". The `Responses` section shows a `Curl` command and a `Request URL` of `https://localhost:44394/api/Students/GetAllStudent`. The `Server response` section is currently empty, showing a `Code` of `200` and a `Response body` field containing an empty JSON object [].



Conclusion

In this chapter, we have implemented the Onion architecture using the Entity Framework and Code First approach. We have now the knowledge of how the layer communicates with each other's in onion architecture and how we can write the Generic code for the Interface repository and services. Now we can develop our project using onion architecture for API Development OR MVC Core Based projects.

Complete GitHub project URL

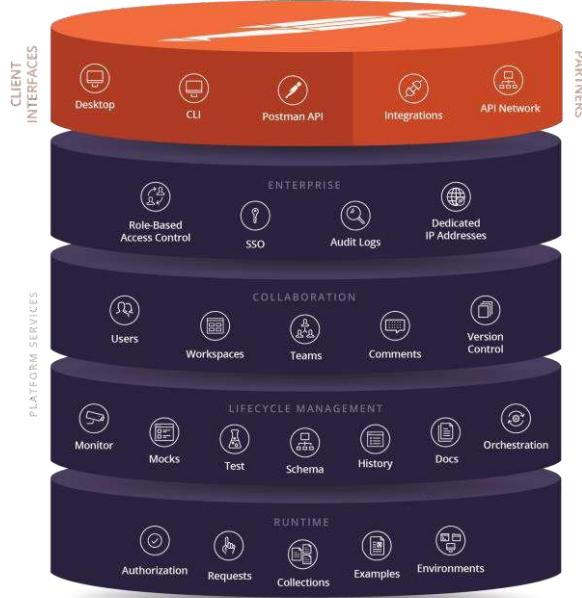
[Click here for complete project](#)

Chapter 4- API Testing Using Postman

- ✓ Introduction
- ✓ How to Test API Web Service in Asp Net Core 5 Using Postman
- ✓ Test Post Call in Postman
- ✓ Test Get Call in Postman
- ✓ Test Put Call in Postman
- ✓ Test Delete Call in Postman
- ✓ Conclusion

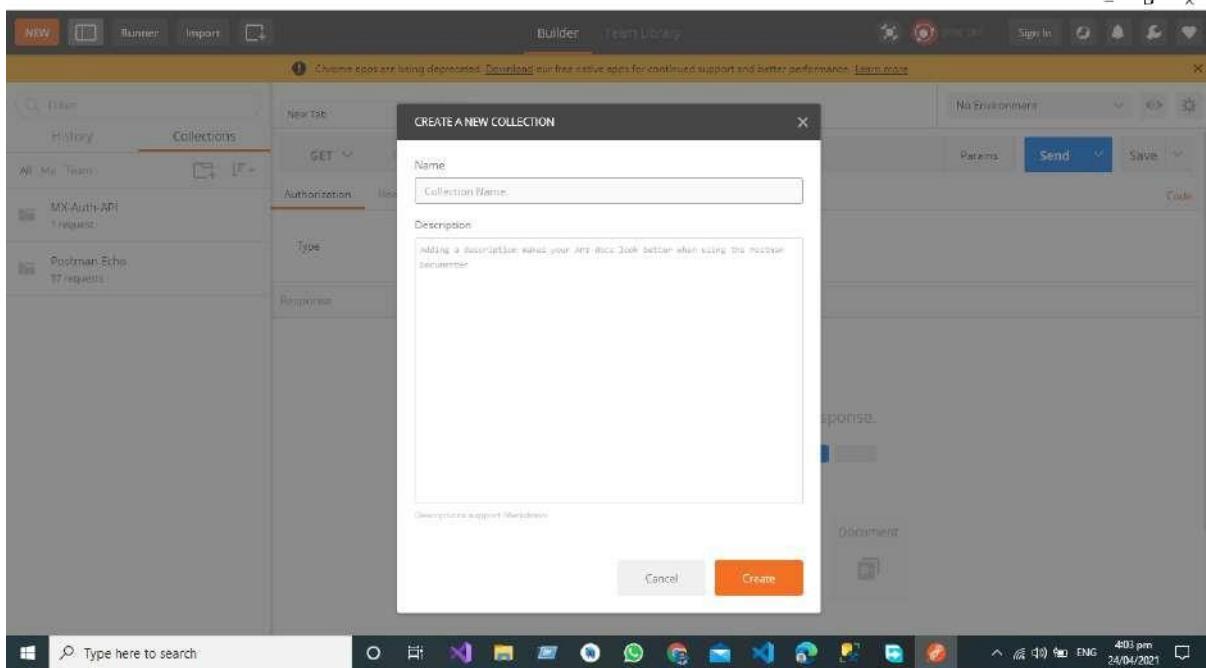
Introduction

In this article I will cover the complete procedure of How to test web API Service using Postman It is a simplest and very beautiful way to test and document your web service. Postman is a collaboration platform for API development. Postman's functions simplify every step of building an API and streamline collaboration so you can create better APIs—quicker.

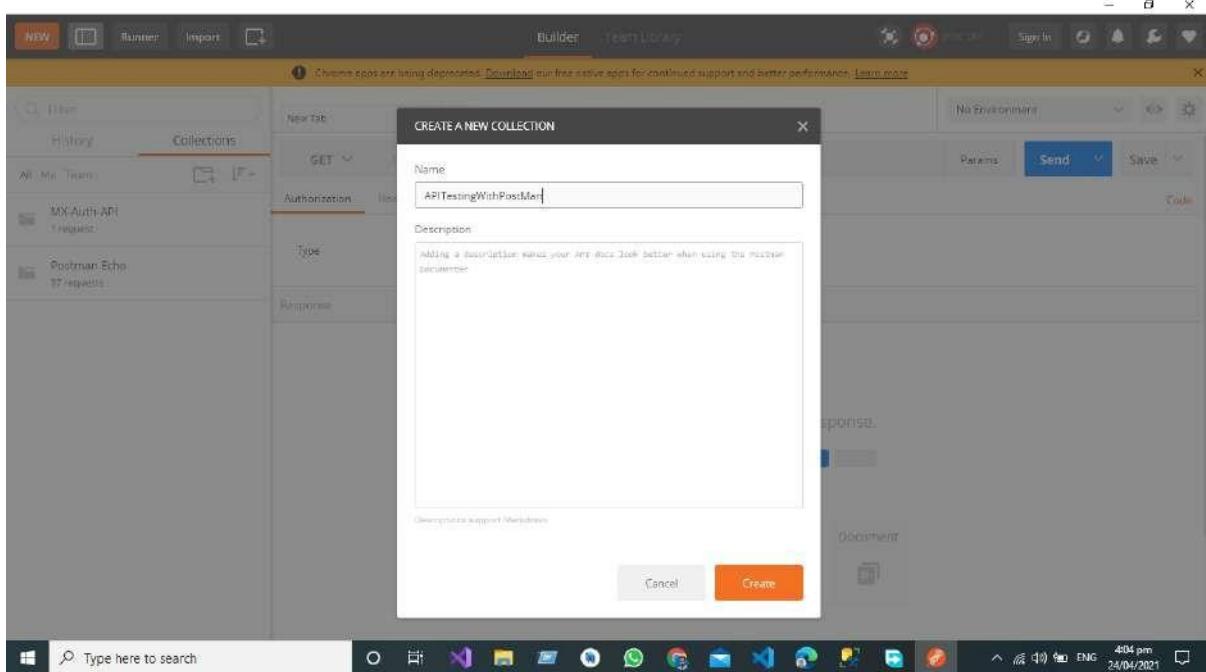


How to Test API Web Service in Asp Net Core 5 Using Postman

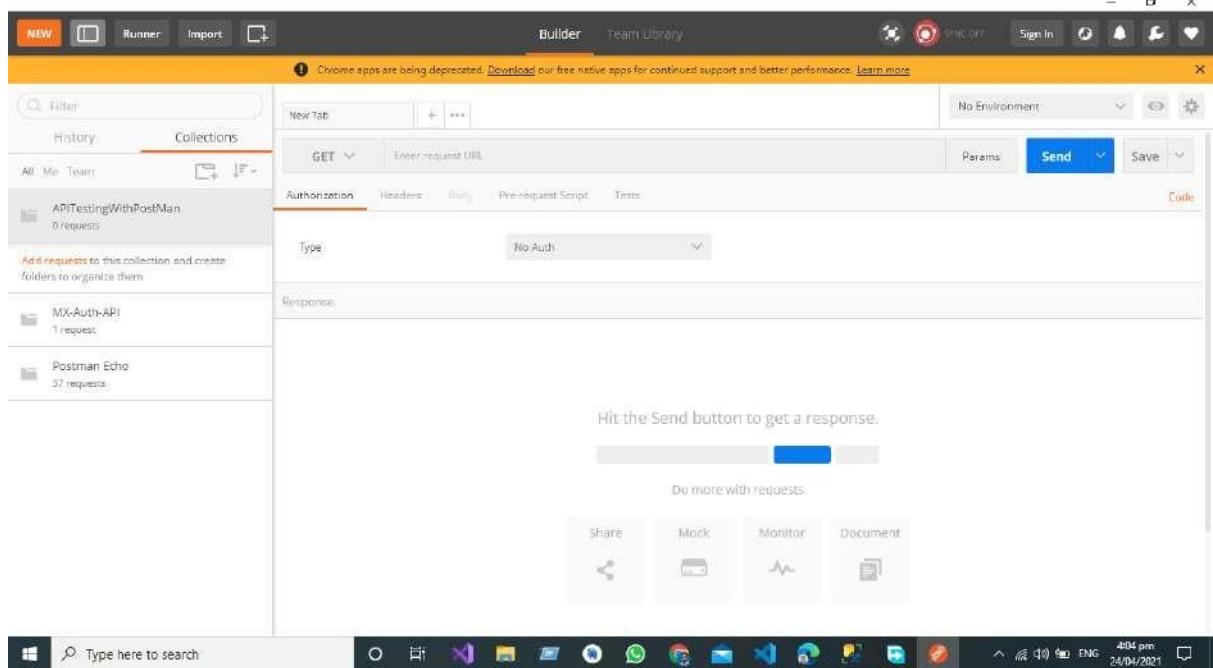
In Postman for API Testing the professional approach is make the collection for API Requests for collecting the responses like GET PUT POST AND DELETE Request Response. Open the post man Click on Create Collection.



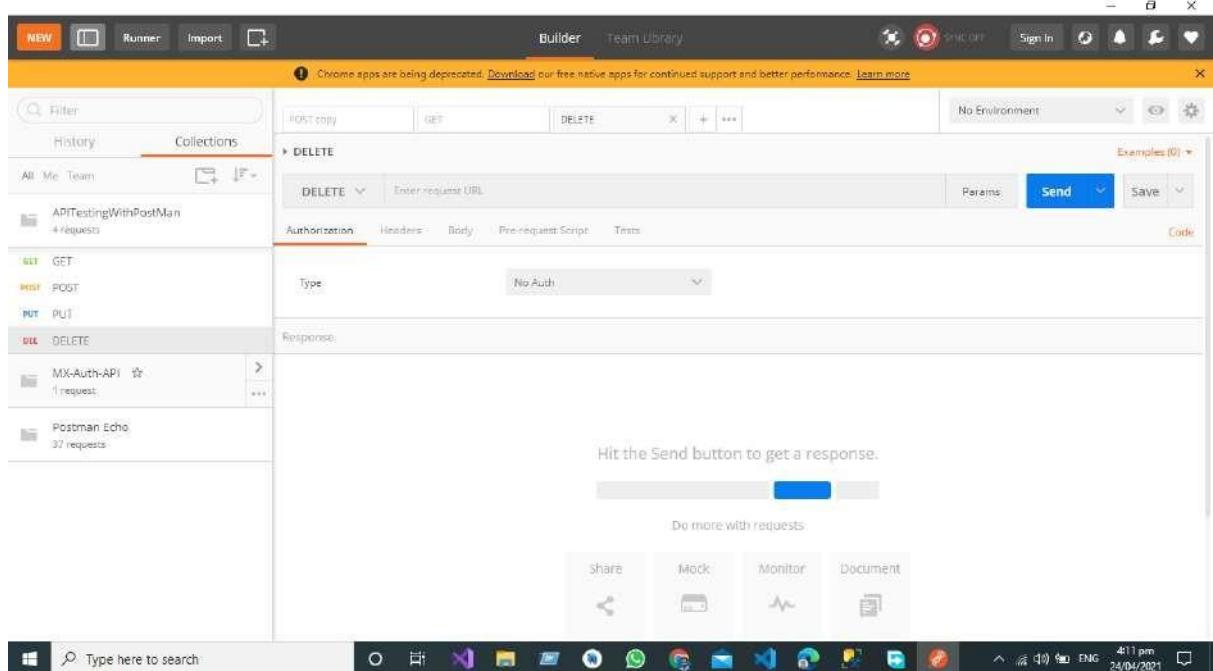
Name the Collection in Postman.



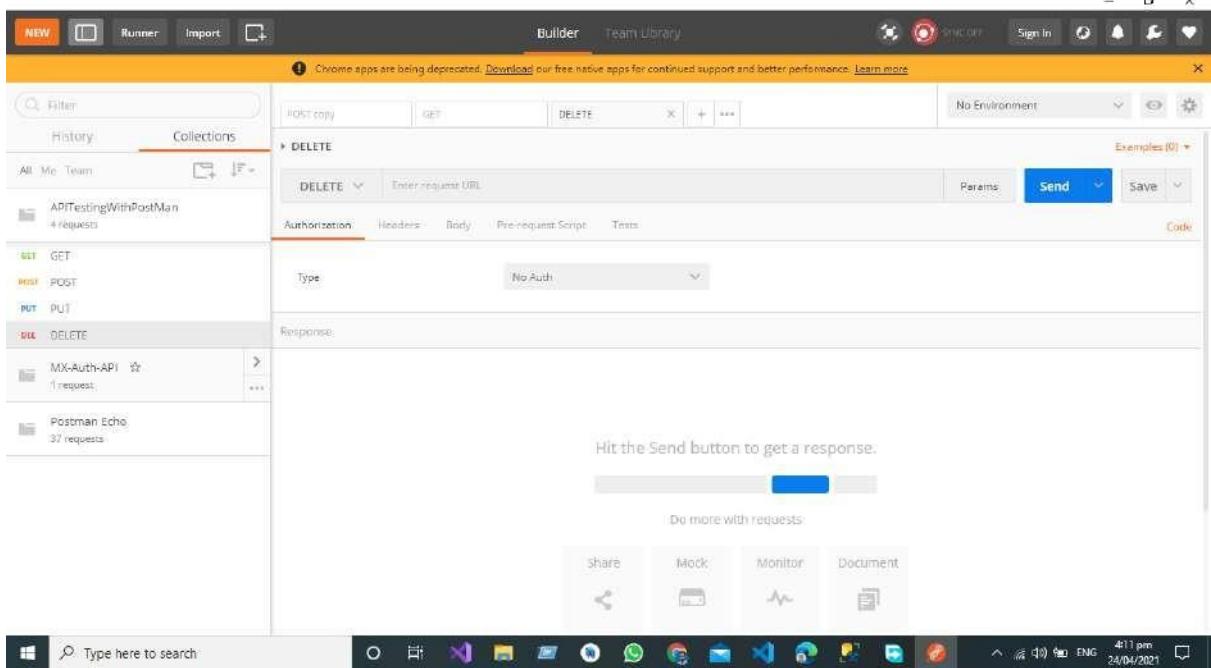
Now the collection is ready for collecting the API Request Response.



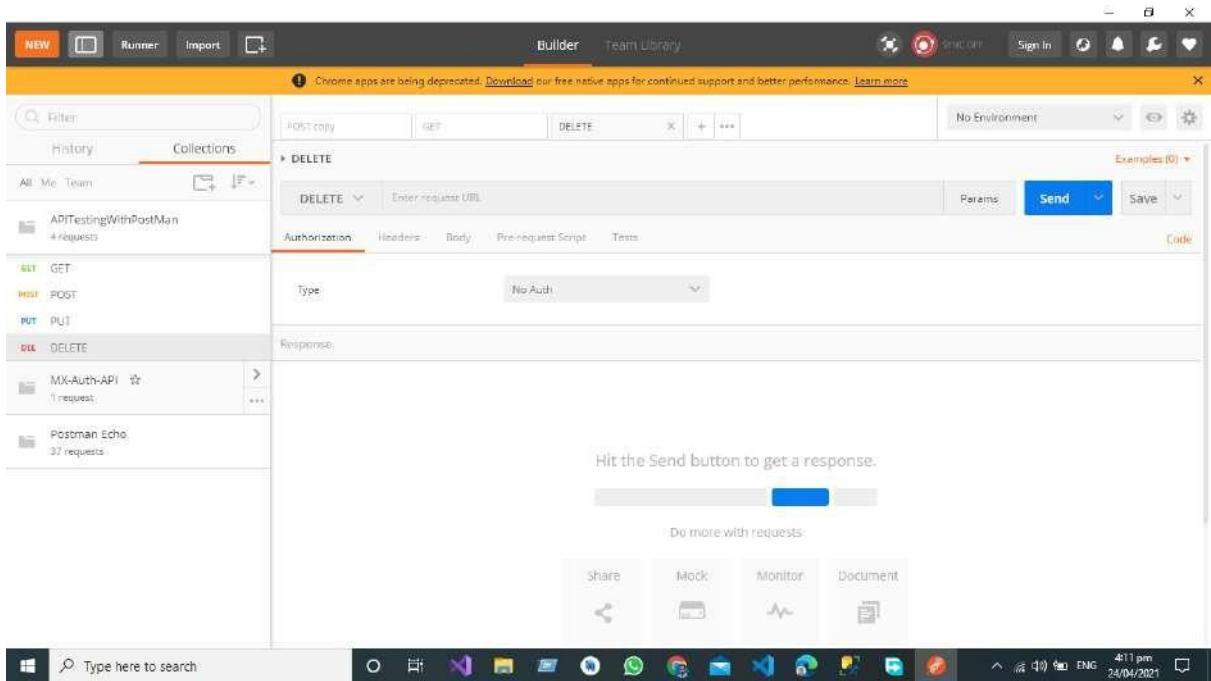
Add the API Request Responses Like PUT,POST,DELETE AND GET,



We are going to test the following use case for this article. In this article I am going to test the **POST, GET, PUT, DELETE, AND UPDATE**, Endpoints of Web Service.



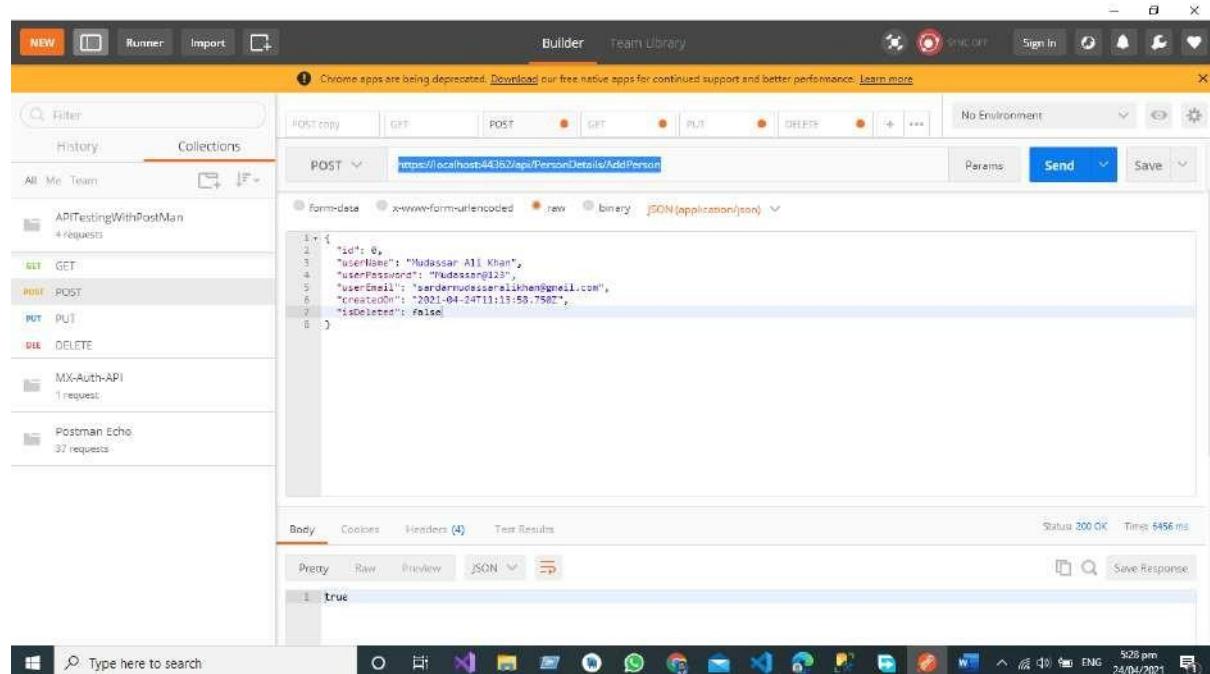
Here in this example we are going to test the Web Service using Postman Developed using the Asp.Net Core 5 Web API



Test Post Call in Postman

Here we are going to send the JSON object with Following Properties

```
{  
    "id": 0,  
    "userName": "string",  
    "userPassword": "string",  
    "userEmail": "string",  
    "createdOn": "2021-04-16T20:38:41.062Z",  
    "isDeleted": true  
}
```



The screenshot shows the Postman application interface. In the top navigation bar, 'Builder' is selected. The main area shows a POST request to 'https://localhost:44362/api/PersonDetails/AddPerson'. The 'Body' tab is active, displaying the JSON payload:

```
1  {  
2      "id": 0,  
3      "userName": "Mudassar Ali Khan",  
4      "userPassword": "Mudassar@123",  
5      "userEmail": "sardernuksarsikhan@gmail.com",  
6      "createdOn": "2021-04-24T11:13:50.750Z",  
7      "isDeleted": false  
8  }
```

The response at the bottom shows a status of 200 OK and a time of 5456 ms. The response body is 'true'.

Now Click on Raw and then Select the JSON

The screenshot shows the Postman application interface. In the top navigation bar, there are tabs for NEW, Runner, Import, and Builder. The Builder tab is active. The main area shows a POST request to the URL <https://localhost:44362/api/PersonDetails/AddPerson>. The request body is set to JSON (application/json) and contains the following JSON object:

```
1. {  
2.     "id": 6,  
3.     "Name": "Muhammad Ali Khan",  
4.     "UserPassword": "Vudessa@123",  
5.     "Email": "sardernuksaraisikhan@gmail.com",  
6.     "CreatedOn": "2021-04-24T10:13:59.750Z",  
7.     "IsDeleted": false  
8. }
```

The response section shows a status of 200 OK with a time of 5456 ms. The response body is displayed as 'true'. The bottom of the screen shows the Windows taskbar with various pinned icons.

If we want to Create the New Person in the Database, we must send the JSON Object to the End Point then data is posted by Postman UI to POST End Point in the Controller Then Create Person Object Creates the Complete Person Object in the Database.

The screenshot shows the Postman application interface. In the top navigation bar, there are tabs for NEW, Runner, Import, and Builder. The Builder tab is active. The main area shows a POST request to the URL <https://localhost:44362/api/PersonDetails/AddPerson>. The request body is set to JSON (application/json) and contains the following JSON object:

```
1. {  
2.     "id": 6,  
3.     "Name": "Muhammad Ali Khan",  
4.     "UserPassword": "Vudessa@123",  
5.     "Email": "sardernuksaraisikhan@gmail.com",  
6.     "CreatedOn": "2021-04-24T10:13:59.750Z",  
7.     "IsDeleted": false  
8. }
```

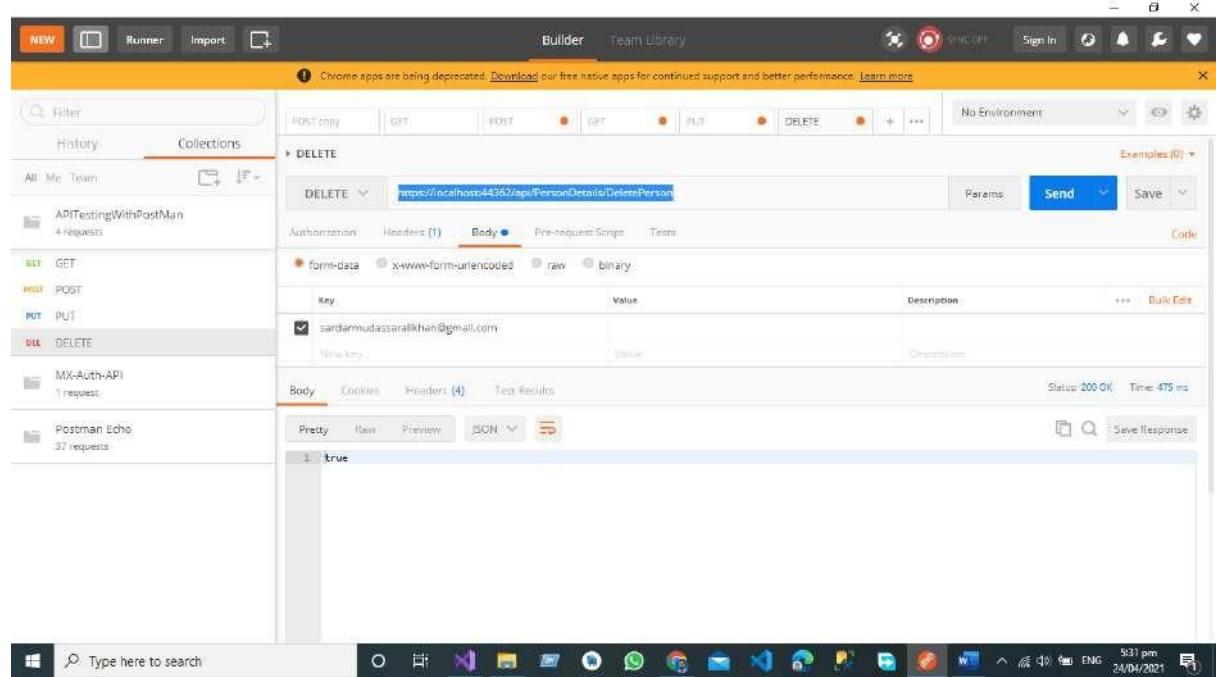
The response section shows a status of 200 OK with a time of 523 pm. The response body is displayed as 'true'. The bottom of the screen shows the Windows taskbar with various pinned icons.

Now you can see that our API give us the server response == true so it indicates that a new record has been created in the data base against our JSON Object that we have sent using Postman.

Test Delete Call in Postman

For Deletion of Person, we have a separate End Point Delete Employee from the Database, we must send the JSON Object to the End Point then data is posted by Postman UI to call the End Point DELETE Person in the Controller Then Person Object will be Deleted from the Database.

In Deletion of the record just select body and the select the form data pass the id value the for the record you want to delete from the database.



The screenshot shows the Postman application interface. In the center, there is a request configuration window for a 'DELETE' operation. The URL is set to `https://localhost:44363/api/PersonDetails/s/DeletePerson`. Under the 'Body' tab, the 'form-data' option is selected, and a single key-value pair is defined: 'Email' with the value 'sardarmudassarali Khan@gmail.com'. The response section at the bottom shows the raw JSON output: `true`.

We are going to delete the record against the given email if the API server response equal to true then our record deleted from the database successfully.

The screenshot shows the Postman application interface. In the top navigation bar, there are tabs for 'Builder' and 'Team Library'. On the right side, there are icons for 'Sync Off', 'Sign In', and various notifications. Below the navigation, a message says 'Chrome apps are being deprecated. Download our free native apps for continued support and better performance. Learn more.' A toolbar at the top has buttons for 'POST copy', 'GET', 'POST', 'PUT', 'DELETE', and others. The URL 'https://localhost:44362/api/PersonDetails/DeletePerson' is entered in the address bar. To the right of the URL are 'Params', 'Send', and 'Save' buttons. The main area shows a table with columns 'Key', 'Value', and 'Description'. A row is selected with the key 'sardarmudassarali Khan@gmail.com' and value 'true'. Below the table, tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results' are visible. The 'Test Results' tab shows a status of 'Status: 200 OK' and 'Time: 475 ms'. At the bottom, there are buttons for 'Pretty', 'Raw', 'Preview', 'JSON', and 'Code'. The status bar at the bottom right shows '5:03 pm 24/04/2021'.

In Above two pictures you can see that after hitting the Delete End point our server response is true it indicates that our record has been delete from the database against our desired Email.

Test Get Call in Postman

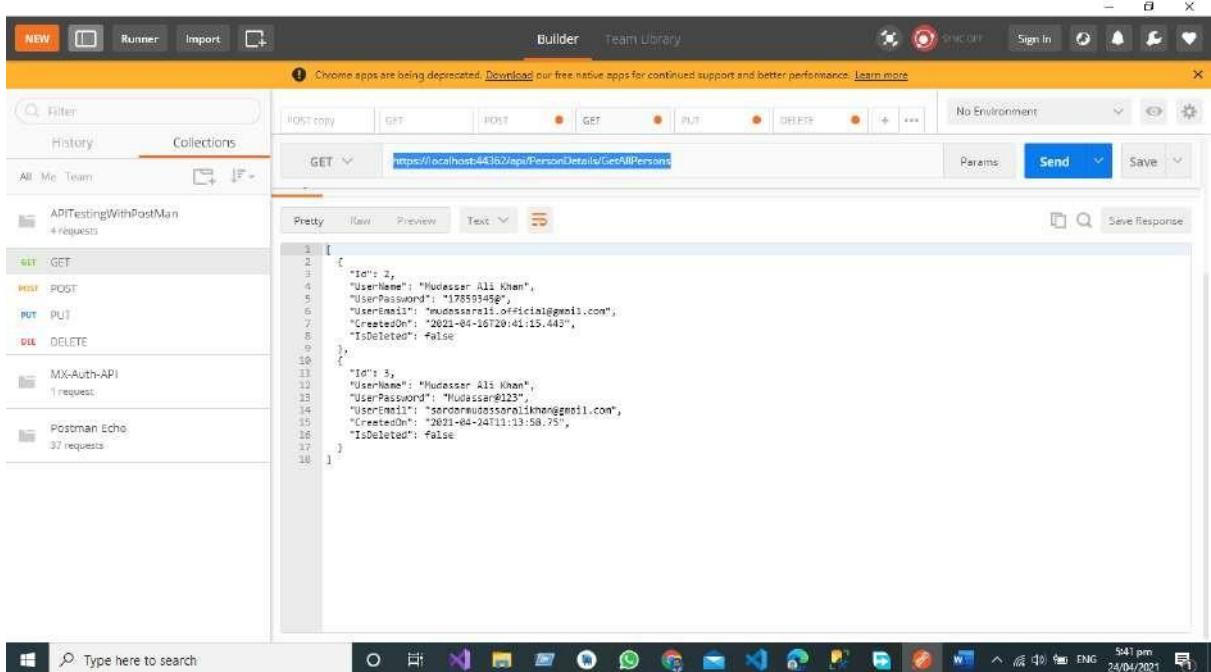
If we want to get the record of all the person's data from the database, we will hit the GET end point in our Web Service using Postman after the completion of server request all the data will be given to us by our GetAllPerson End point.

The screenshot shows the Postman application interface. The top navigation bar and toolbar are identical to the previous screenshot. The URL 'https://localhost:44362/api/PersonDetails/GetAllPersons' is entered in the address bar. The 'GET' button is highlighted in green. The 'Authorization' tab is selected under the 'Headers' section. The 'Body' tab is selected, showing a JSON response. The response is a list of objects, each representing a person's details:

```
[{"Id": 2, "UserName": "Mudassar Ali Khan", "UserPassword": "1234567890", "UserEmail": "mudasserali.oficial@gmail.com", "CreatedOn": "2021-04-16T20:41:15.443", "IsDeleted": false}, {"Id": 3, "UserName": "Mudassar Ali Khan", "UserPassword": "Mudassar123", "UserEmail": "xandermudasseralikhan@gmail.com", "CreatedOn": "2021-04-24T11:13:58.75", "IsDeleted": false}]
```

The 'Test Results' tab shows a status of 'Status: 200 OK' and 'Time: 2937 ms'. The status bar at the bottom right shows '5:40 pm 24/04/2021'.

Hit the GET Endpoint for getting all record from the database.



The screenshot shows the Postman application interface. In the left sidebar, there's a list of collections: 'APITestingWithPostMan' (4 requests), 'MX-Auth-API' (1 request), and 'Postman Echo' (37 requests). The main area shows a GET request to 'https://localhost:44382/api/PersonDetails/GetAllPersons'. The response is displayed in 'Pretty' format as a JSON array:

```
[{"id": 2, "UserName": "Mudasser Ali Khan", "UserPassword": "17859345@", "UserEmail": "mudassarali.official@gmail.com", "CreatedOn": "2021-04-16T20:41:15.443Z", "IsDeleted": false}, {"id": 3, "UserName": "Mudasser Ali Khan", "UserPassword": "Mudasser@123", "UserEmail": "sardarmudassarali.khan@gmail.com", "CreatedOn": "2021-04-24T11:13:58.75Z", "IsDeleted": false}]
```

After the successful execution of Web service all the record is visible to us in the form of JSON object in database we have Only One Record so that

Test Put Call in Postman

In put call we are going to update the record in database. We have Separate End Points for All the operations like Update the Record of the Person by User Email from the Database by Hitting the UPDTAE End Point that send the request to controller and in controller we have the separate End Point for Updating the record based on User Email from the database.

We are going to update the given below JSON Object in the database

```
{  
    "id": 0,  
    "userName": "Mudassar Ali Khan",  
    "userPassword": "17859345@",  
    "userEmail": "mudassarali.official@gmail.com",  
    "createdOn": "2021-04-16T20:41:15.443Z",  
    "isDeleted": false  
}
```

To given below JSON object

```
{  
    "id": 0,  
    "userName": "Sardar Mudassar Ali Khan",  
    "userPassword": "SardarMudassarAliKhan@123",  
    "userEmail": "sardarmudassarali.khan@gmail.com",  
    "createdOn": "2021-04-24T11:13:58.75Z",  
    "isDeleted": false  
}
```

```

"userPassword": "17859345@",
"userEmail": "mudassarali.official@gmail.com",
"createdOn": "2021-04-16T20:41:15.443Z",
"isDeleted": false
}

```

The screenshot shows the Postman application interface. On the left, there's a sidebar with a collection named 'APITestingWithPostMan' containing four requests: GET, POST, PUT, and DELETE. The main area shows a 'PUT' request to the URL `https://localhost:44362/api/PersonDetails/UpdatePerson`. The 'Body' tab is selected, displaying a JSON payload:

```

1. {
2.     "id": 0,
3.     "userName": "Sardar Mudassar Ali Khan",
4.     "userPassword": "Mudasser@123",
5.     "userEmail": "SardarMudassarAliKhan@gmail.com",
6.     "createdOn": "2021-04-16T20:41:15.443Z",
7.     "isDeleted": false
8. }

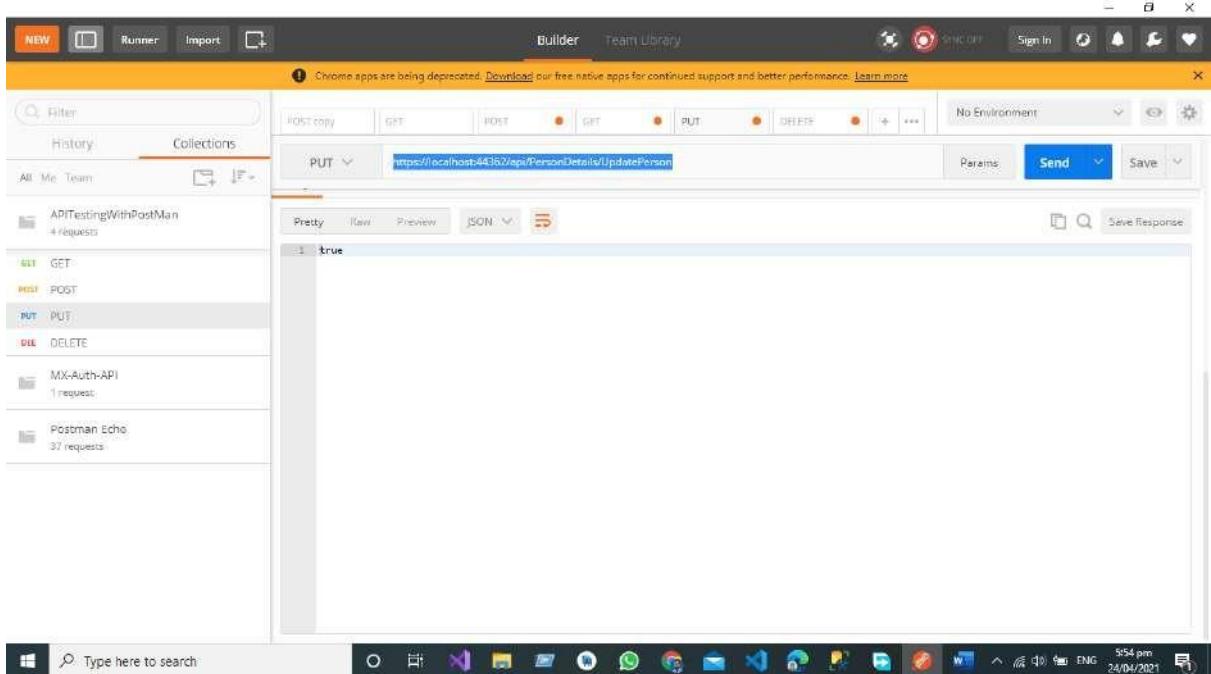
```

Below the body, the response status is shown as 'status 200 OK' and 'Time 2160 ms'. The bottom of the screen shows a Windows taskbar with various icons.

We have sent the API JSON Response to the PUT Endpoint for Updating the Record in the database

The screenshot shows the Postman application interface again. The left sidebar remains the same with the 'APITestingWithPostMan' collection. The main area now displays the response to the PUT request. The 'Body' tab is selected, showing the response body as 'true', which indicates the update was successful. The bottom of the screen shows a Windows taskbar with various icons.

After the successful submission of data, you can see that our server response is true it indicates that our data has been successfully submitted in the database



Conclusion

As a Software Engineer My Conclusion about the Postman is the best Open API for Testing the RESTful API as a Developer I have a great experienced with Postman for testing the restful API sending the Complete JSON Object and then getting the response in a professional way. It is easy to implement in Asp.net core Web API Project and after the configuration Developers can enjoy the beauty of Postman Open API.

Chapter No 5 API Testing Using Swagger

- ✓ Introduction
- ✓ How to Test API Web Service in Asp Net Core 5 Using Swagger
- ✓ Test Post Call in Postman
- ✓ Test Get Call in Postman
- ✓ Test Put Call in Postman
- ✓ Test Delete Call in Postman
- ✓ Conclusion

Introduction

In this article I will cover the complete procedure of How to test web API Service using Swagger Open API It is the simplest and very beautiful way to test and document your web service.

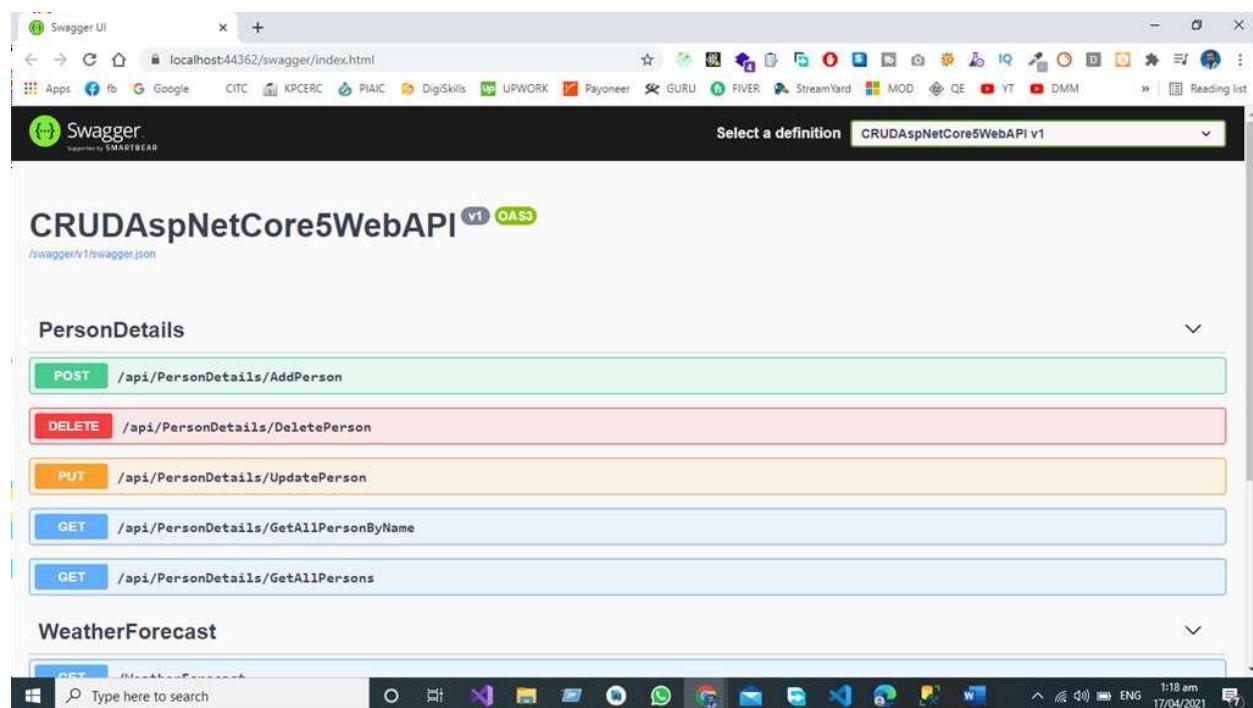
Swagger (Open API) is a language-agnostic specification for describing and documenting the REST API. Swagger Allows both the Machine and Developer to understand the working and capabilities of the Machine without direct access to the source code of the project the main objectives of swagger (Open API) are to:

Minimize the workload to connect with Micro services.

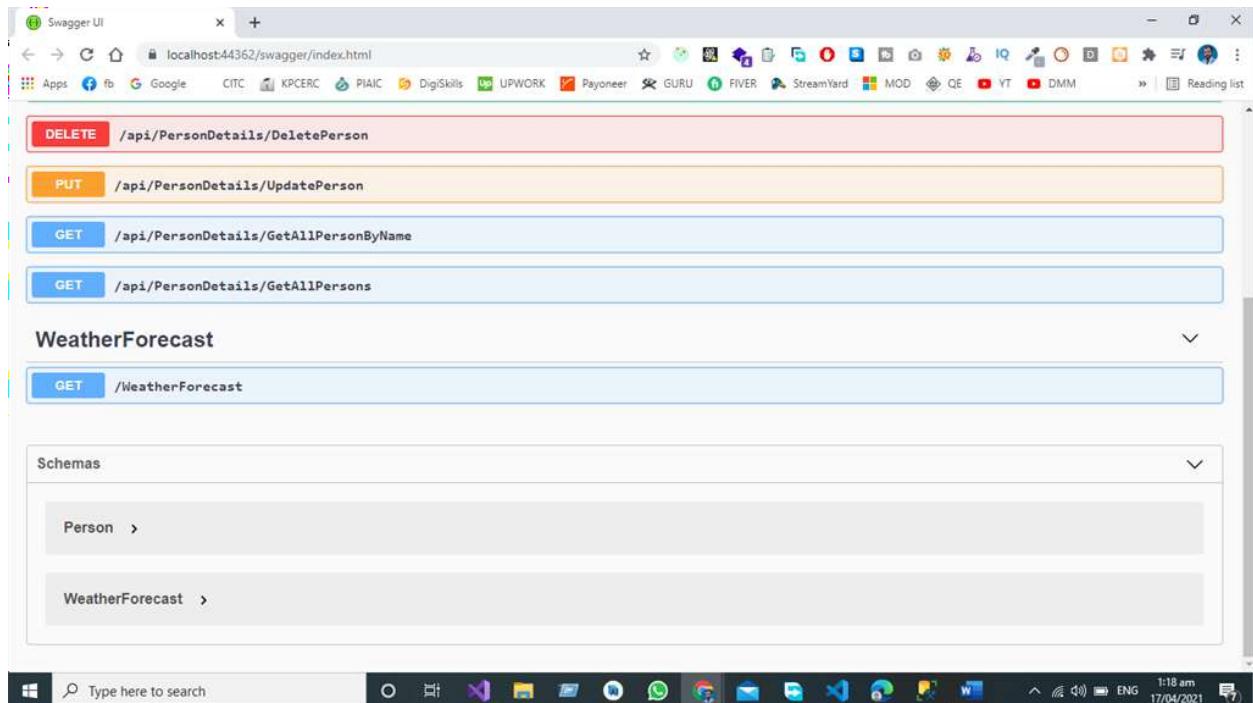
Reduce the Time Needed to accurately document the Micro services.

How to Test API Web Service in Asp Net Core 5 Using Swagger

We are going to test the following use case for this article. In this article I am going to test the **POST, GET, PUT, DELETE, AND UPDATE**, Endpoints of Web Service.



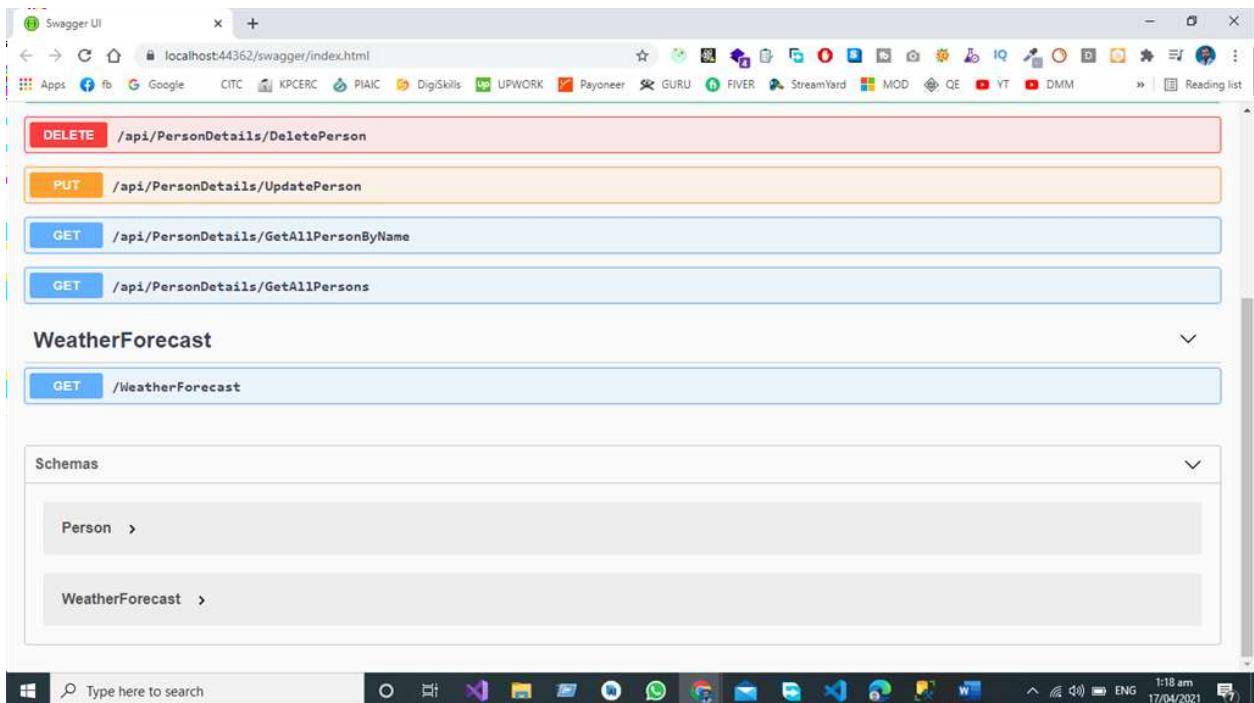
Here in this example, we are going to test the Web Service using Swagger Developed using the Asp.Net Core 5 Web API.



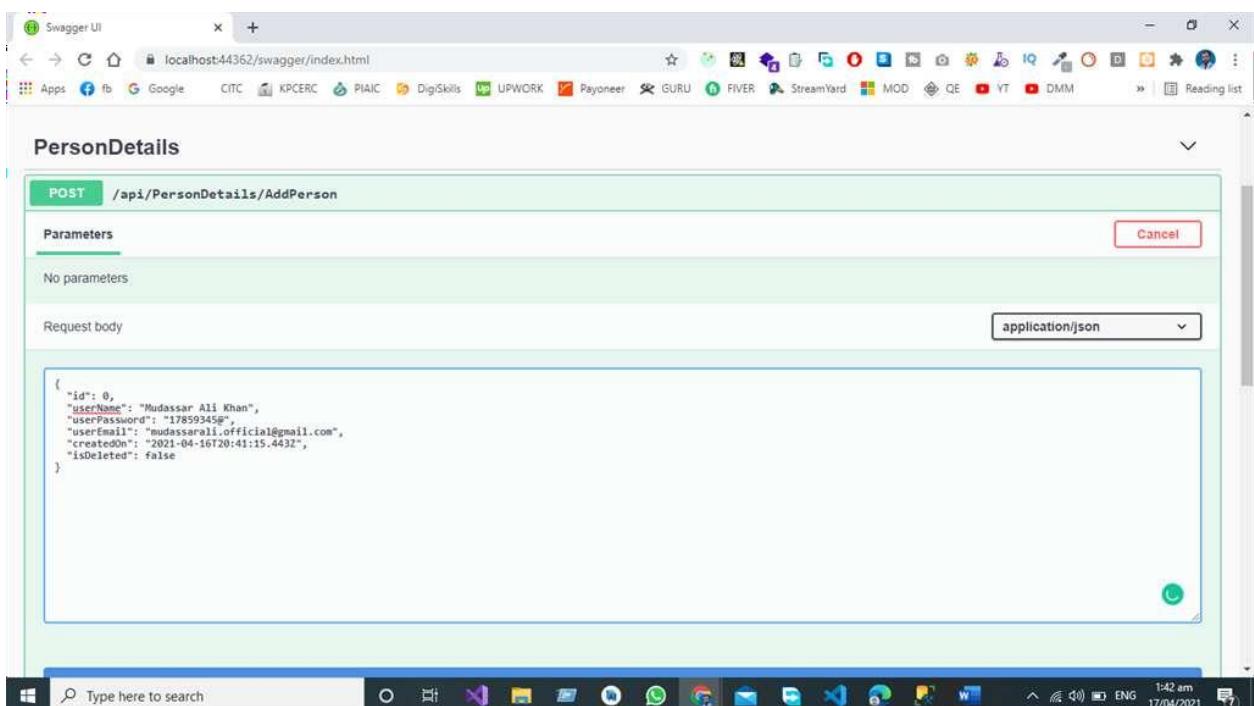
Test POST Call in Swagger

Here we are going to send the JSON object with the Following Properties

```
1. {  
2.   "id": 0,  
3.   "userName": "string",  
4.   "userPassword": "string",  
5.   "userEmail": "string",  
6.   "createdOn": "2021-04-16T20:38:41.062Z",  
7.   "isDeleted": true  
8. }
```



Now click the try-out button.



If we want to Create the New Person in the Database, we must send the JSON Object to the End Point then data is posted by Swagger UI to POST End Point in the Controller then Create Person Object Creates the Complete Person Object in the Database.

The screenshot shows the Swagger UI interface for a .NET Core API. A successful POST request to the endpoint `/api/PersonDetails/AddPerson` has been made. The response code is 200, and the response body contains the value `true`. The response headers are also displayed, including standard HTTP headers like `Content-Type` and `Date`, as well as ASP.NET specific headers like `X-Powered-By`.

Now you can see that our API gives us the server response == true so it indicates that a new record has been created in the database against our JSON Object that we have sent using Swagger.

Test DELETE Call in Swagger

For Deletion of Person, we have a separate End Point Delete Employee from the Database, we must send the JSON Object to the End Point then data is posted by Swagger UI to call the End Point DELETE Person in the Controller Then Person Object will be deleted from the Database.

The screenshot shows the Swagger UI interface for a .NET Core Web API. The URL is `localhost:44362/swagger/index.html`. The main window displays the `DELETE /api/PersonDetails/DeletePerson` endpoint. In the 'Parameters' section, there is a single parameter named 'UserEmail' of type 'string (query)' with the value 'mudassarali.official@gmail.com'. Below this is a large blue 'Execute' button. Under the 'Responses' section, a single entry for code 200 is listed with the description 'Success' and a 'Links' column showing 'No links'. The media type is set to 'text/plain'. The bottom status bar indicates the system is at 1:59 am on 17/04/2021.

Now we are going to delete the record against a certain Id

The screenshot shows the execution of the `DELETE /api/PersonDetails/DeletePerson` endpoint. The 'Curl' section contains the command `curl -X DELETE "https://localhost:44362/api/PersonDetails/DeletePerson?UserEmail=mudassarali.official%40gmail.com" -H "accept: text/plain"`. The 'Request URL' is `https://localhost:44362/api/PersonDetails/DeletePerson?UserEmail=mudassarali.official%40gmail.com`. The 'Server response' section shows a 200 OK status with a response body of 'true'. The 'Response headers' section lists standard HTTP headers including 'access-control-allow-methods: GETPUTDELETEHEADOPTIONS', 'access-control-allow-origin: https://localhost:44362', 'content-length: 4', 'content-type: application/json; charset=utf-8', 'date: Fri 16 Apr 2021 20:59:52 GMT', 'server: Microsoft-IIS/10.0', and 'x-powered-by: ASP.NET'. The bottom status bar indicates the system is at 1:59 am on 17/04/2021.

We are going to delete the record against the given email if the API service equal to true then our record deleted from the database successfully.

The screenshot shows the Swagger UI interface for a .NET Core Web API. At the top, a status bar displays various application icons. Below it, the main content area shows a successful DELETE operation response:

```
access-control-allow-methods: GETPUTDELETEHEADOPTIONS  
access-control-allow-origin: https://localhost:44362  
content-length: 4  
content-type: application/json; charset=utf-8  
date: Fri 16 Apr 2021 20:59:52 GMT  
server: Microsoft-IIS/10.0  
x-powered-by: ASP.NET
```

Under the "Responses" section, a table lists the response code (200) and description ("Success"). The "Links" column indicates "No links".

Below the table, there is a dropdown for "Media type" set to "text/plain", with a note below it stating "Controls Accept header". There are also "Example Value" and "Schema" buttons.

At the bottom of the main content area, there are three buttons for other endpoints:

- PUT** /api/PersonDetails/UpdatePerson
- GET** /api/PersonDetails/GetAllPersonByName
- GET** /api/PersonDetails/GetAllPersons

The bottom of the screen shows the Windows taskbar with various pinned icons and the system tray.

In the above two pictures, you can see that after hitting the Delete Endpoint our server response is true it indicates that our record has been deleted from the database against our desired Email.

Test GET Call in Swagger

If we want to get the record of all the personal data from the database, we will hit the GET endpoint in our Web Service using swagger after the completion of the server request all the data will be given to us by our GETAllPerson Endpoint.

The screenshot shows the Swagger UI interface for a .NET Core Web API. At the top, a status bar displays various application icons. Below it, the main content area shows a successful GET operation response for the "/api/PersonDetails/GetAllPersons" endpoint:

```
Code Description  
200 Success
```

Under the "Responses" section, a table lists the response code (200) and description ("Success"). The "Links" column indicates "No links".

Below the table, there is a "Parameters" section with a note "No parameters" and a "Cancel" button. There is also a large blue "Execute" button.

At the bottom of the main content area, there is a "Responses" section for the "WeatherForecast" endpoint:

```
Code Description  
200 Success
```

Below the responses, there are three buttons for other endpoints:

- GET** /WeatherForecast

The bottom of the screen shows the Windows taskbar with various pinned icons and the system tray.

Now we are going to fetch all the record from the database.

```
curl -X GET "https://localhost:44362/api/PersonDetails/GetAllPersons" -H "accept: */*"
```

Request URL
https://localhost:44362/api/PersonDetails/GetAllPersons

Server response

Code Details

200 Response body

```
[{"Id": 2, "UserName": "Mudassar Ali Khan", "UserPassword": "178593456", "UserEmail": "mudassarali.official@gmail.com", "CreatedOn": "2021-04-16T20:41:15.443", "IsDeleted": false}]
```

Response headers

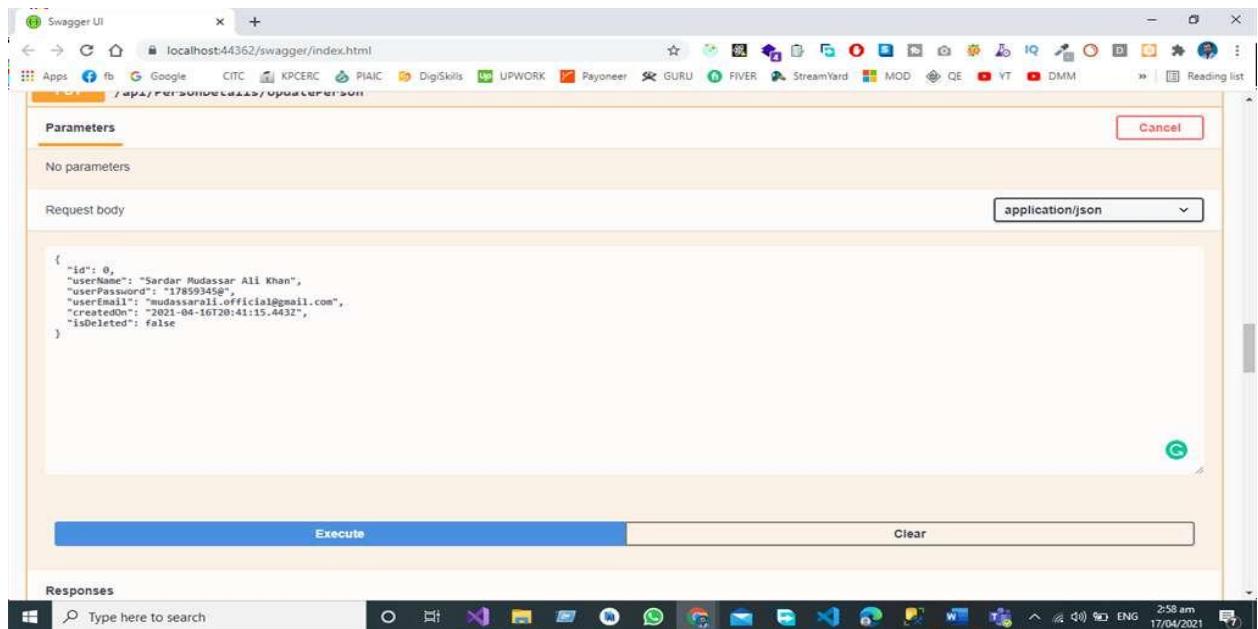
```
access-control-allow-methods: GETPUTPOSTDELETEHEADOPTIONS
access-control-allow-origin: https://localhost:44362
content-encoding: gzip
content-length: 270
content-type: text/plain; charset=utf-8
date: Fri, 16 Apr 2021 21:37:00 GMT
server: Microsoft-IIS/10.0
vary: Accept-Encoding
x-powered-by: ASP.NET
```

After the successful execution of Web service, all the record is visible to us in the form of a JSON object in database we have Only One Record so that

Test PUT Call in Swagger

In put-call, we are going to update the record in the database. We have Separate End Points for All the operations like Update the Record of the Person by User Email from the Database by Hitting the UPDATE End Point that sends the request to the controller and in the controller we have a separate End Point for Updating the record based on User Email from the database.

We are going to update the given below JSON Object in the database.



```
1. {
2.   "id": 0,
3.   "userName": "Mudassar Ali Khan",
4.   "userPassword": "17859345@",
5.   "userEmail": "mudassarali.official@gmail.com",
6.   "createdOn": "2021-04-16T20:41:15.443Z",
7.   "isDeleted": false
8. }
```

To give below JSON Object

```
1. {
2.   "id": 0,
3.   "userName": "Sardar Mudassar Ali Khan",
4.   "userPassword": "17859345@",
5.   "userEmail": "mudassarali.official@gmail.com",
6.   "createdOn": "2021-04-16T20:41:15.443Z",
7.   "isDeleted": false
8. }
```

Swagger UI

localhost:44362/swagger/index.html

Parameters

No parameters

Request body

application/json

```
{ "id": 0, "userName": "Sardar Mudassar Ali Khan", "userPassword": "178593456", "userEmail": "mudassarali.official@gmail.com", "createdOn": "2021-04-16T20:41:15.443Z", "isDeleted": false }
```

Execute Clear

Responses

After the successful submission of data, you can see that our server response is true it indicates that our data has been successfully submitted to the database

Swagger UI

localhost:44362/swagger/index.html

Responses

Curl

```
curl -X PUT "https://localhost:44362/api/PersonDetails/UpdatePerson" -H "accept: text/plain" -H "Content-Type: application/json" -d "{\"id\":0,\"userName\":\"Sardar Mudassar Ali Khan\",\"userPassword\":\"178593456\",\"userEmail\":\"mudassarali.official@gmail.com\",\"createdOn\":\"2021-04-16T20:41:15.443Z\",\"isDeleted\":false}"
```

Request URL

https://localhost:44362/api/PersonDetails/UpdatePerson

Server response

Code Details

200 Response body

true

Download

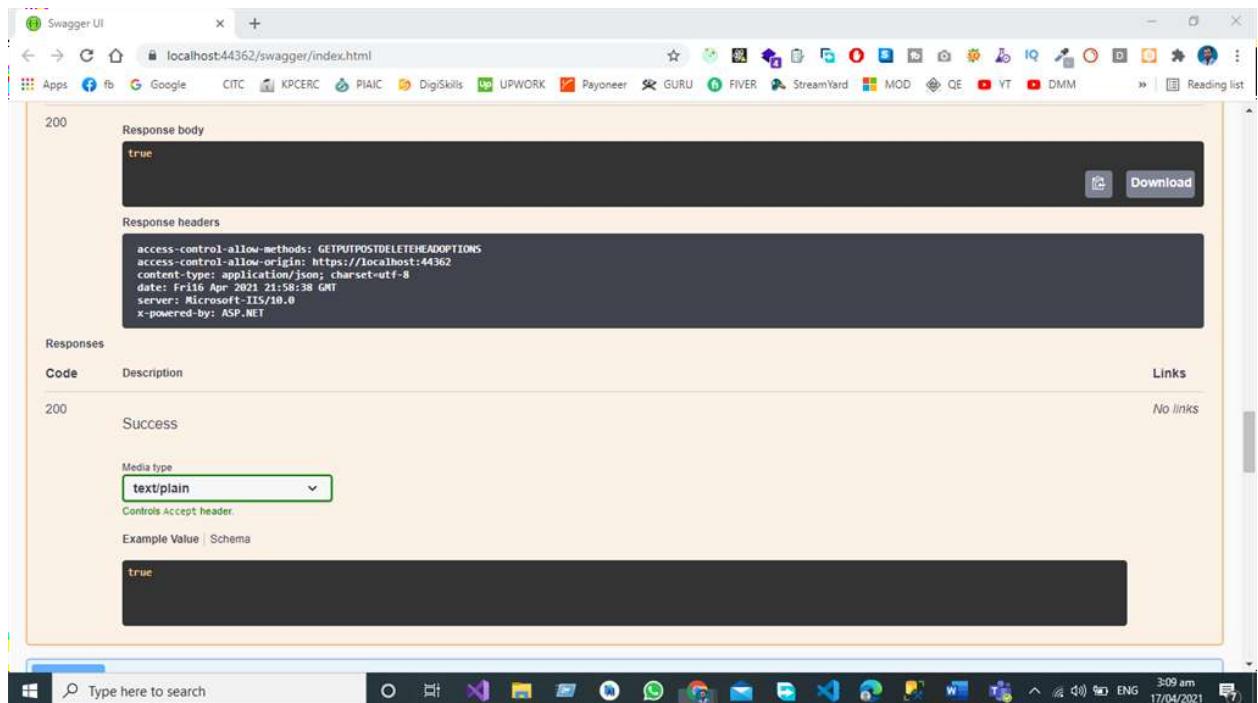
Response headers

```
access-control-allow-methods: GETPUTDELETEHEADOPTIONS
access-control-allow-origin: https://localhost:44362
content-type: application/json; charset=utf-8
date: Fri, 16 Apr 2021 23:58:38 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
```

Responses

Code Description

Links



Summary

Swagger (Open API) is a language-agnostic specification for describing and documenting the REST API. Swagger Allows both the Machine and Developer to understand the working and capabilities of Machine without direct access to the source code of the project the main objectives of swagger (Open API) are too,

Minimize the workload to connect with Micro services.

Reduce the Time Needed to accurately document the Micro services.

Swagger is the Interface Description Language for Describing the RESTful APIS Expressed using JSON. Swagger is used to gather with a set of open-source software tools to Design build documents and use Restful Web Services Swagger includes automated documentation code generation and test the generation. Three main components of Swashbuckle Swashbuckle.AspNetCore.Swagger is the Object Model and Middleware to expose swagger Document as JSON End Points.

Swashbuckle.AspNetCore.SwaggerGen.Swagger Generator that builds Swagger Document Objects Directly from your routes controllers and models this package is combined with swagger endpoints middleware to automatically expose the Swagger JSON. Swashbuckle.AspNetCore.SwaggerUI is an Embedded version of the Swagger UI Tool it explains swagger JSON to build a rich customizable practical for explaining the Web API Functionality it includes a built-in test harnessed for the public Methods.

Conclusion

Now My Conclusion about the Swagger Open API Swagger is the best Open API for Documenting the Restful API as a Developer I have great experience with Swagger for testing the restful API sending the Complete JSON Object and then getting the response in an efficient way. It is easy to implement in Asp.net core Web API Project and after the configuration, Developers can enjoy the beauty of Swagger Open API.

Chapter 6- Application Deployment on Microsoft Azure Cloud.

- ✓ Introduction
- ✓ Deployment Procedure
- ✓ Server Configuration
- ✓ Deployment Steps
- ✓ Conclusion

Introduction

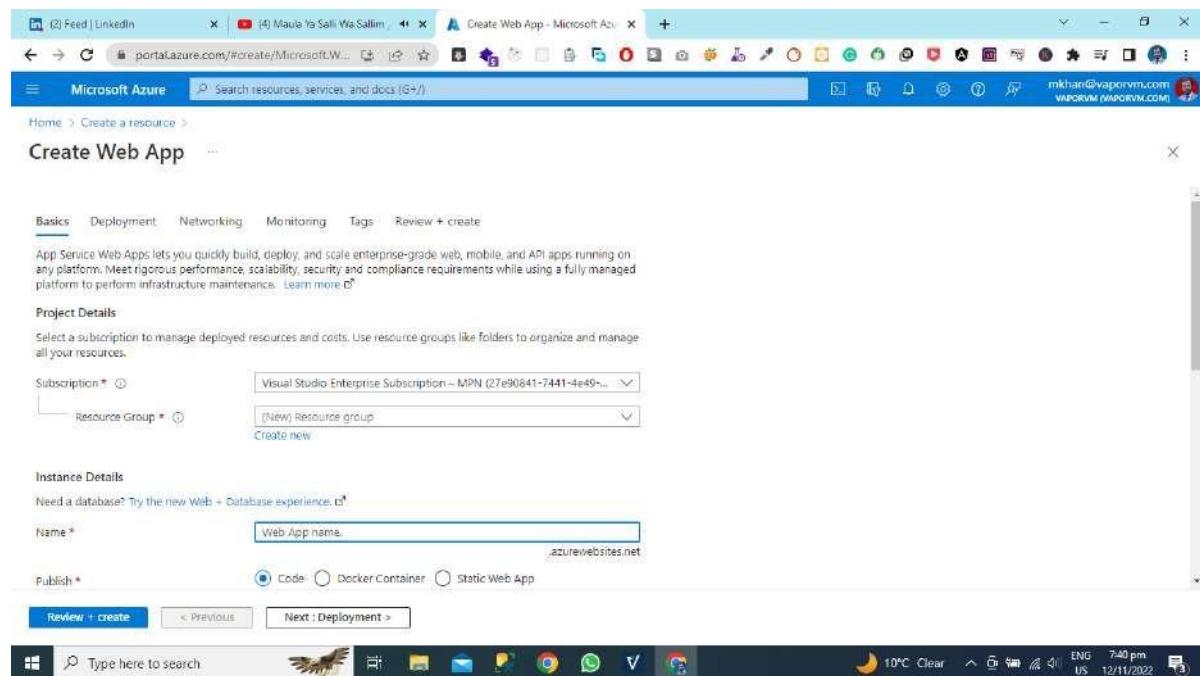
App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance.

Open Your Azure portal

First you need to open your Microsoft azure account after that go to home and then select the subscription where you want to create azure web app.

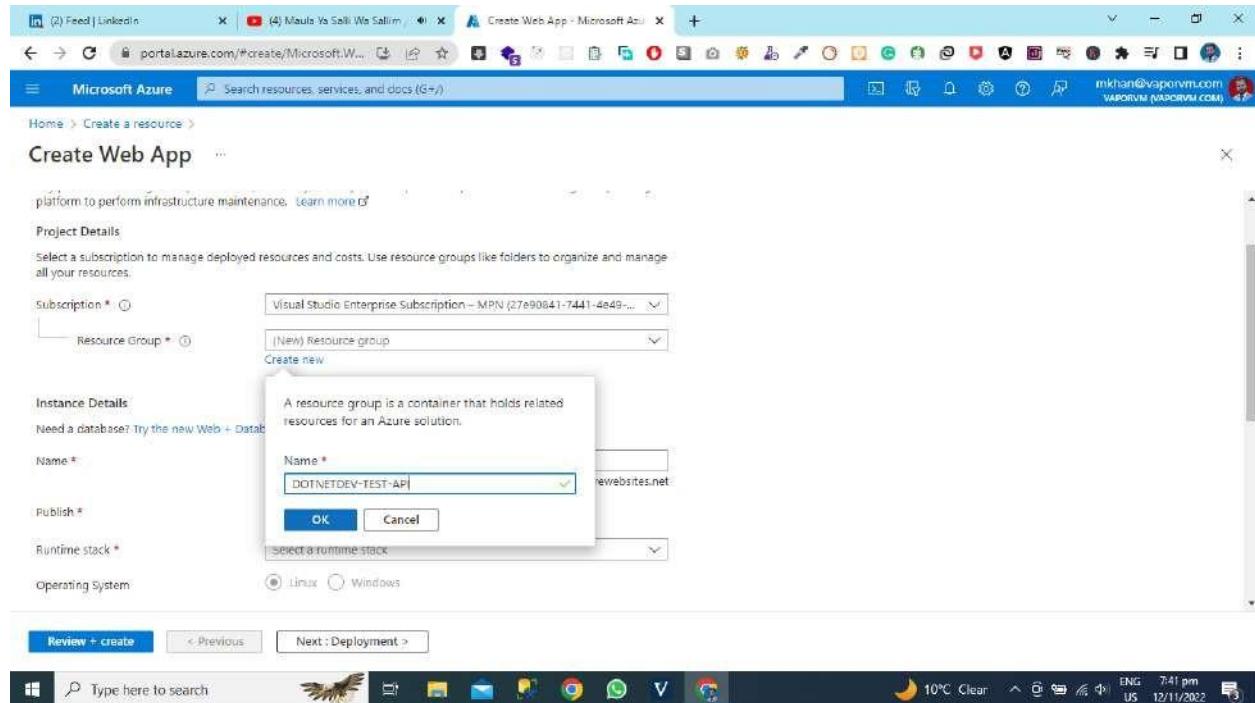
Create Azure Web App

Select the azure app from resources then follow the steps that are necessary for the azure application Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.



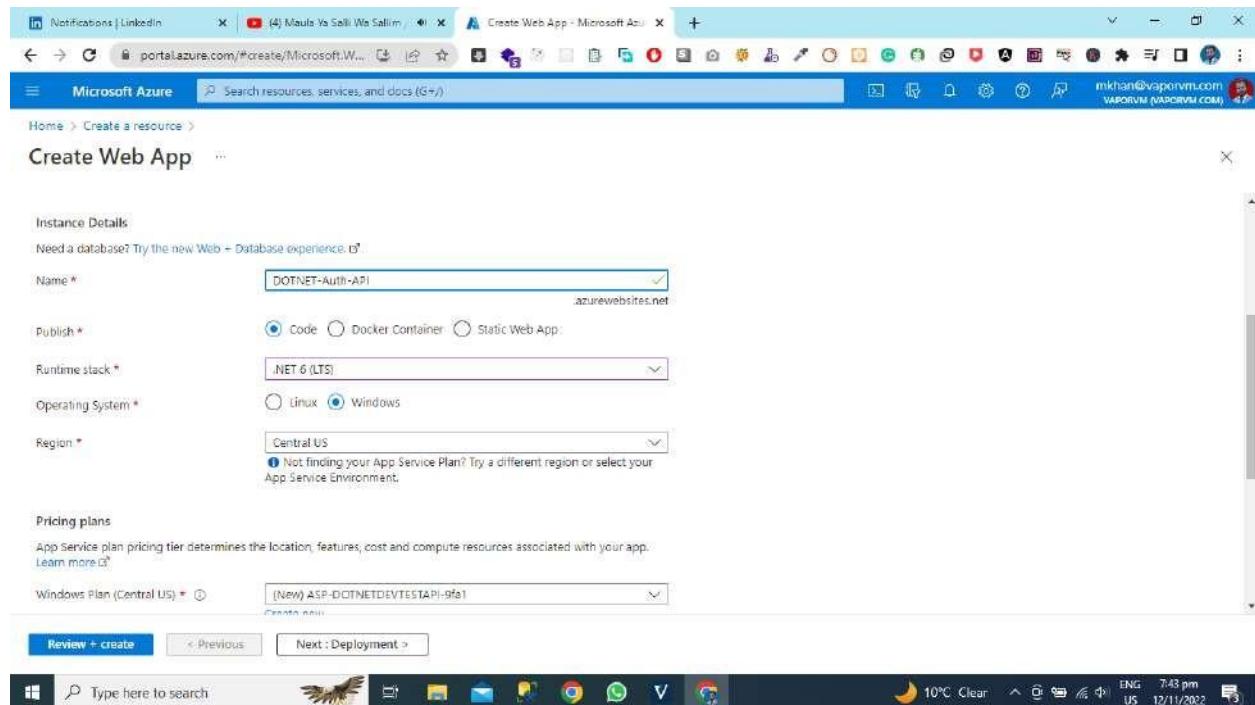
Create the Azure Resource Group.

Now Select the Azure Subscription and create the Azure resource group.



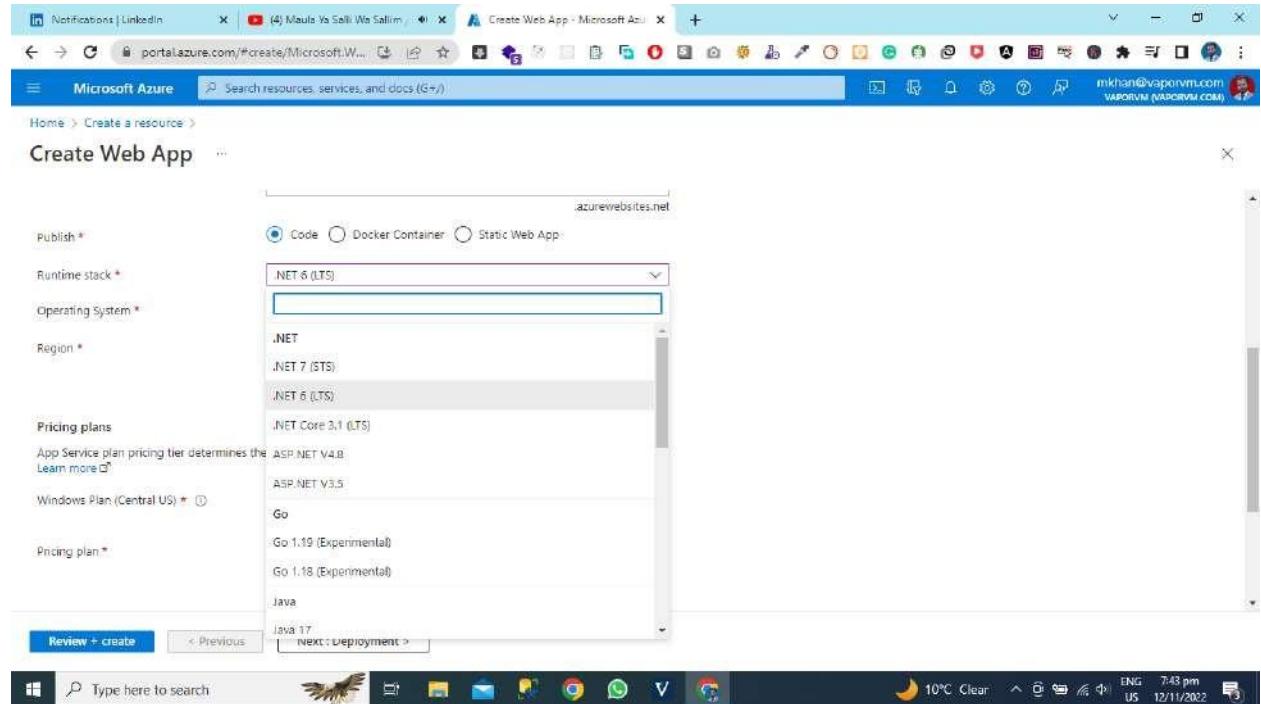
Select the Name of Your Application

Select the Azure App Name and I am selecting my web app name is DOTNET-Auth-API



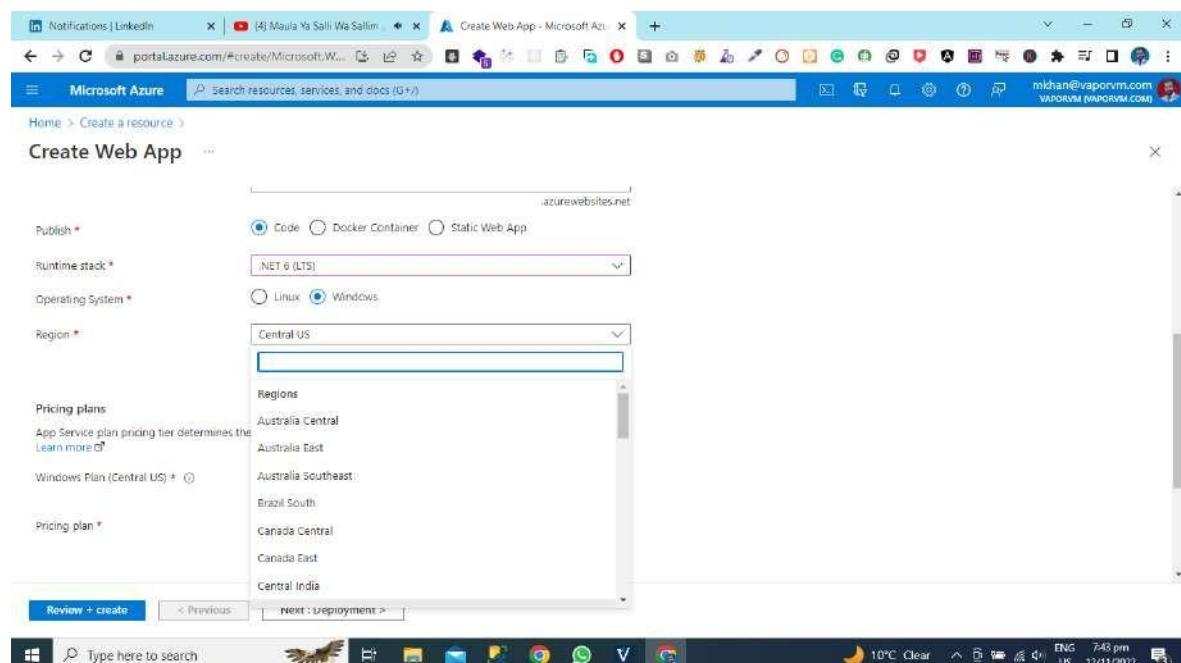
Select the Runtime Stack

After selecting the name and Recourse group now select the code version for your application and my application is using Microsoft Azure Asp.Net 6 so I am selecting .NET 6 Latest stable Version.



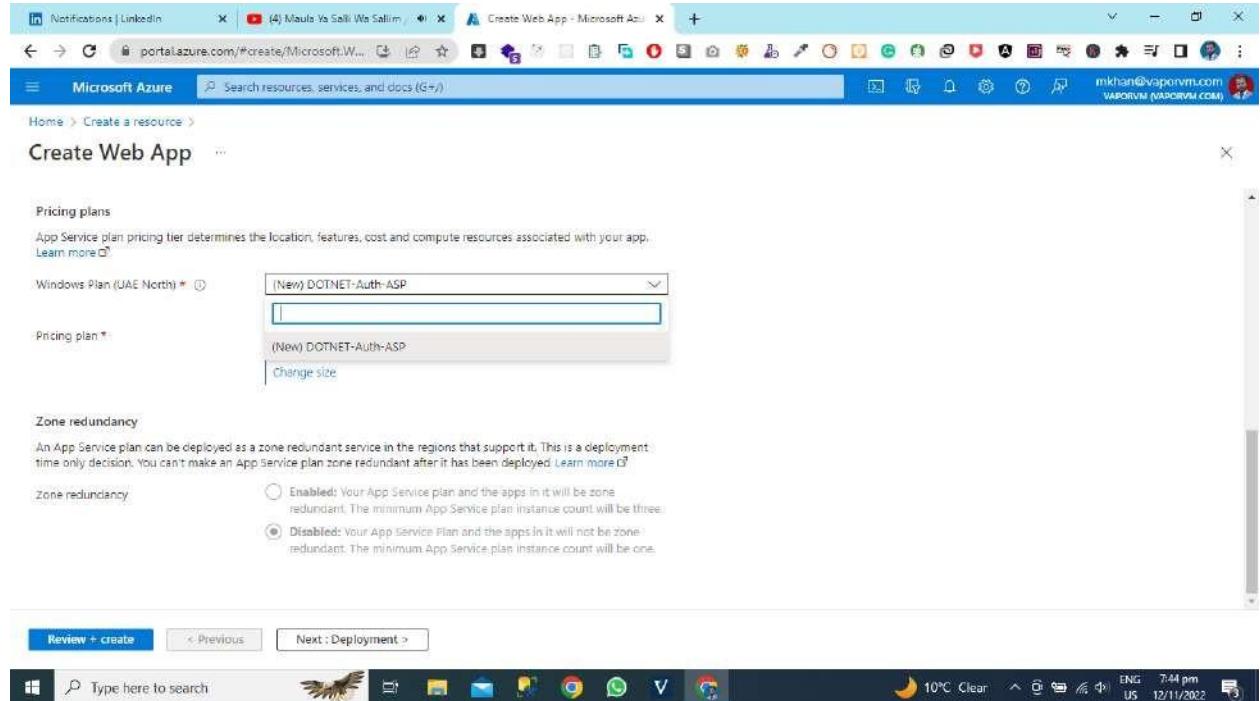
Select the Azure Region

After Selecting the Azure Code Runtime and then select the azure region where you want to deploy your application azure regions are very important for application cost and application deployment cost. So I am selecting **Central US**.



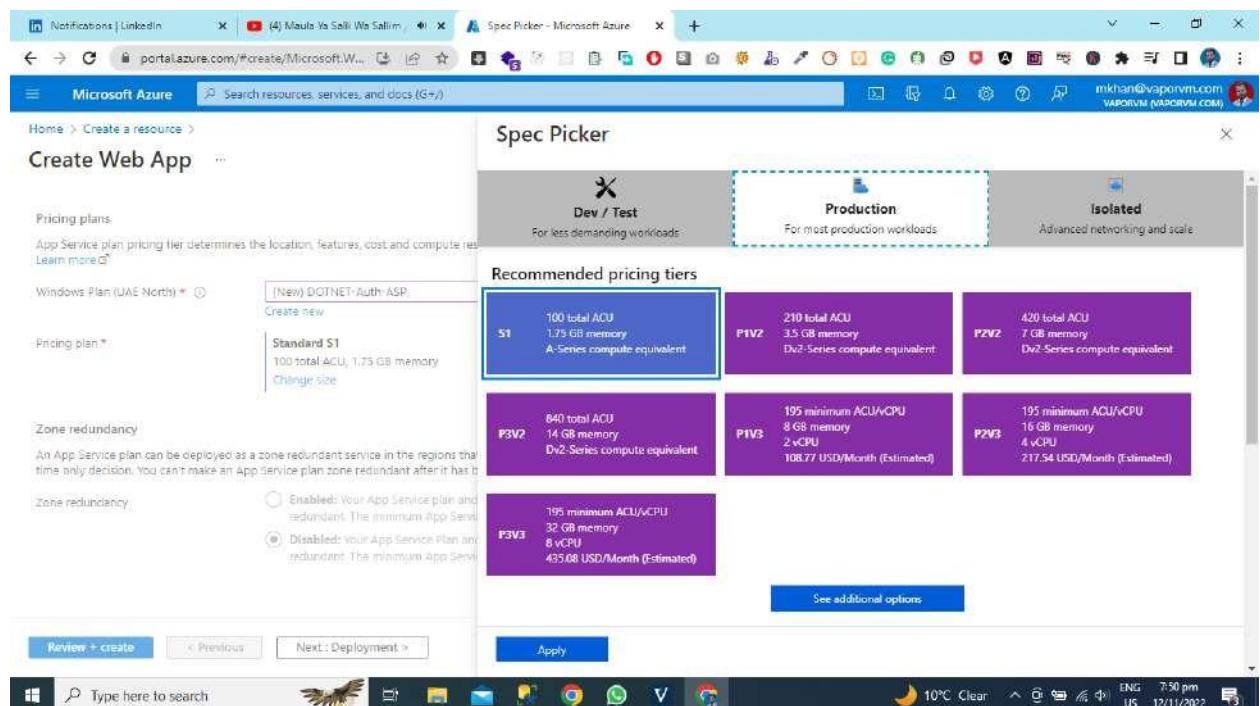
Create App Service Plan

An App Service plan defines a set of compute resources for a web app to run. These compute resources are analogous to the server farm in conventional web hosting. One or more apps can be configured to run on the same computing resources (or in the same App Service plan).



Select the pricing plan.

Now Select the pricing Tier Dev/Test, Production, Isolated



Now our purpose of application is development and testing so we are selecting Dev/Test

The screenshot shows the Microsoft Azure portal interface for creating a new resource. The top navigation bar includes links for Notifications, LinkedIn, YouTube, and the Spec Picker - Microsoft Azure. The main title is 'Create a resource > Create Web App'. On the left, there's a 'Pricing plans' section with a dropdown set to '(New) DOTNET-Auth-ASP' and a 'Standard S1' plan selected. The 'Zone redundancy' section has two options: 'Enabled' (selected) and 'Disabled'. On the right, the 'Spec Picker' window is open, showing three tiers: 'Dev / Test' (selected), 'Production', and 'Isolated'. Under 'Recommended pricing tiers', three options are listed: F1 Shared infrastructure (1 GB memory, 60 minutes/day compute), D1 Shared infrastructure (1 GB memory, 240 minutes/day compute), and B1 100 total ACU (1.75 GB memory, A-Series compute equivalent). Below the tiers, there's a 'See additional options' button. To the right, under 'Included hardware', it lists 'Memory' (Memory available to run applications deployed and running in the App Service plan) and 'Storage' (1 GB disk storage shared by all apps deployed in the App Service plan). At the bottom of the Spec Picker, there are 'Review + create', 'Previous', 'Next : Deployment >', and 'Apply' buttons.

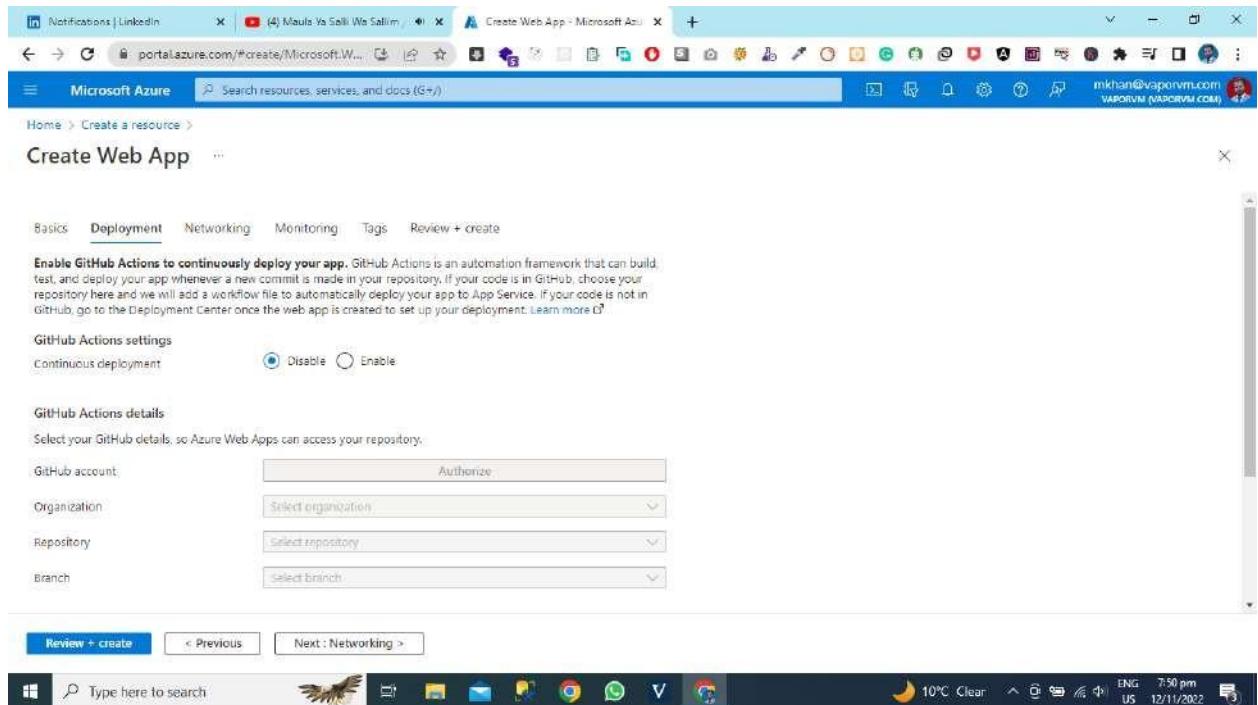
Select the Recommended Pricing Tier

Here you can see the recommended pricing tier and we are selecting the BI Green

This screenshot shows the same 'Spec Picker' interface as the previous one, but with a different selection. The 'BI' tier (B1) is now highlighted with a dashed green border. The 'Included features' section on the right shows 'Custom domains / SSL' and 'Manual scale'. The 'Included hardware' section shows 'Azure Compute Units (ACU)' and 'Memory'. The rest of the interface remains the same, including the 'Review + create', 'Previous', 'Next : Deployment >', and 'Apply' buttons at the bottom.

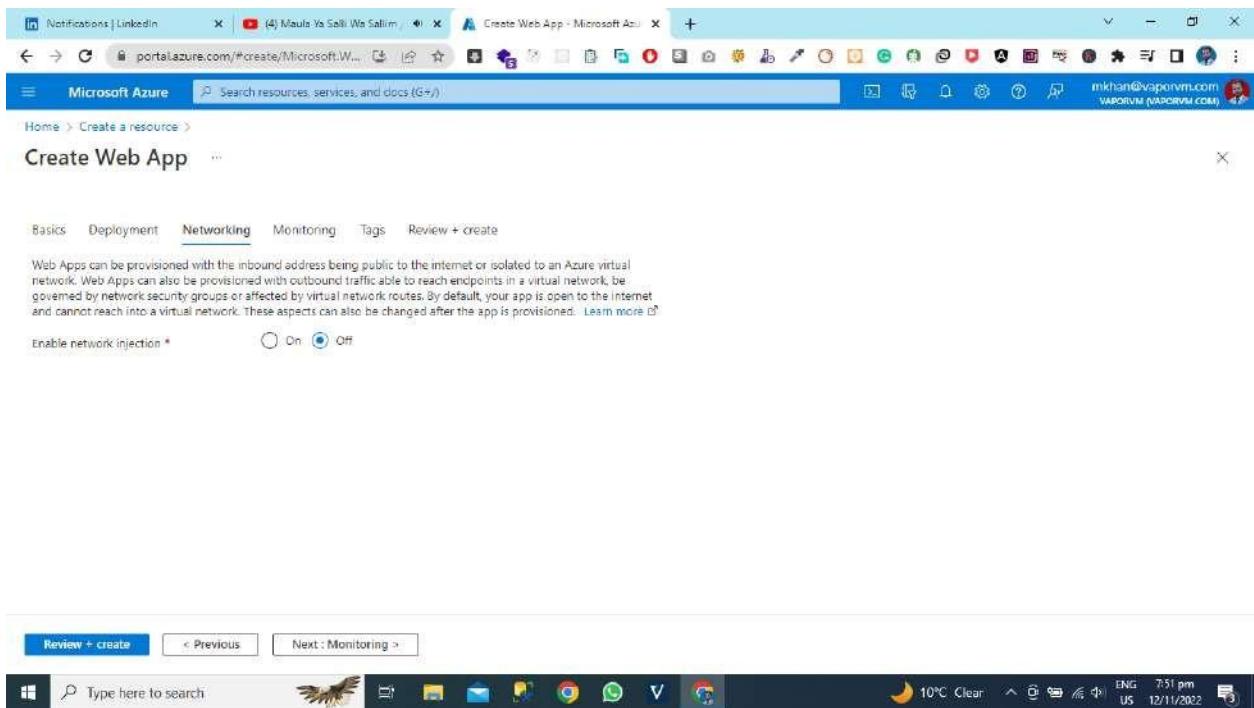
Select the Deployment Option.

Enable GitHub Actions to continuously deploy your app. GitHub Actions is an automation framework that can build, test, and deploy your app whenever a new commit is made in your repository. If your code is in GitHub, choose your repository here and we will add a workflow file to automatically deploy your app to App Service. If your code is not in GitHub, go to the Deployment Center once the web app is created to set up your deployment.



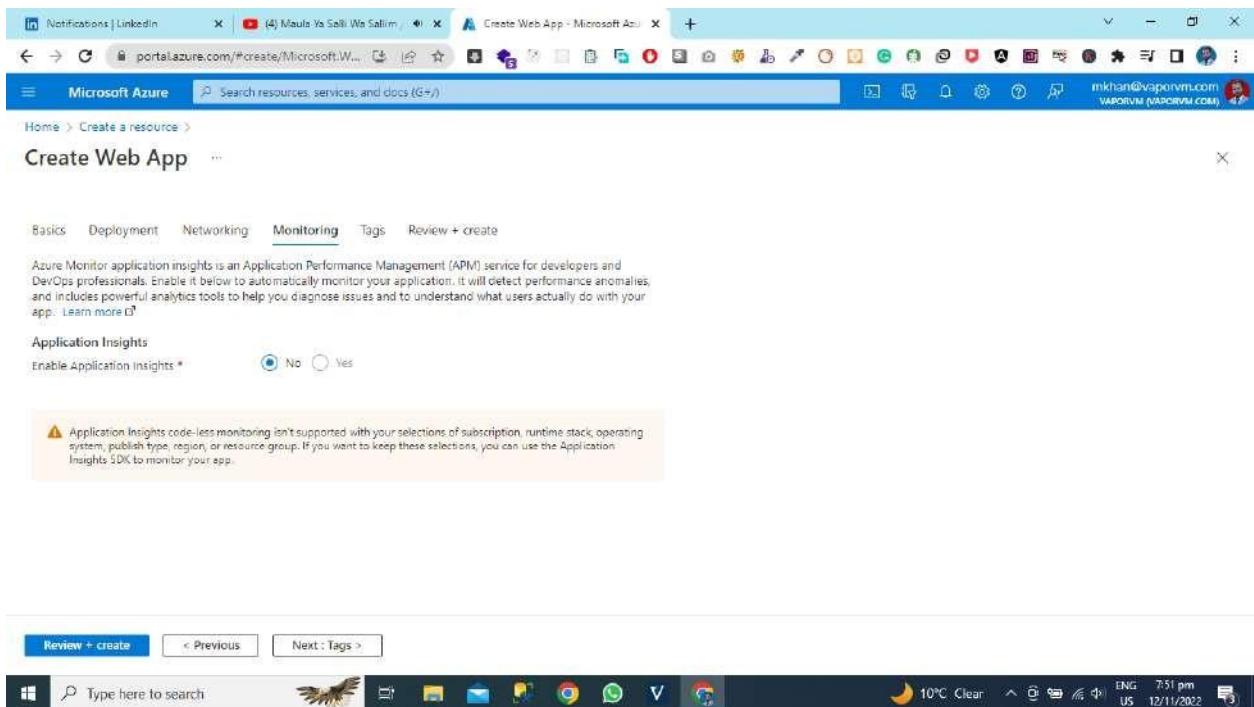
Select the Net Working Options

Web Apps can be provisioned with the inbound address being public to the internet or isolated to an Azure virtual network. Web Apps can also be provisioned with outbound traffic able to reach endpoints in a virtual network, be governed by network security groups or affected by virtual network routes. By default, your app is open to the internet and cannot reach into a virtual network. These aspects can also be changed after the app is provisioned.



Select the Azure Monitoring Options

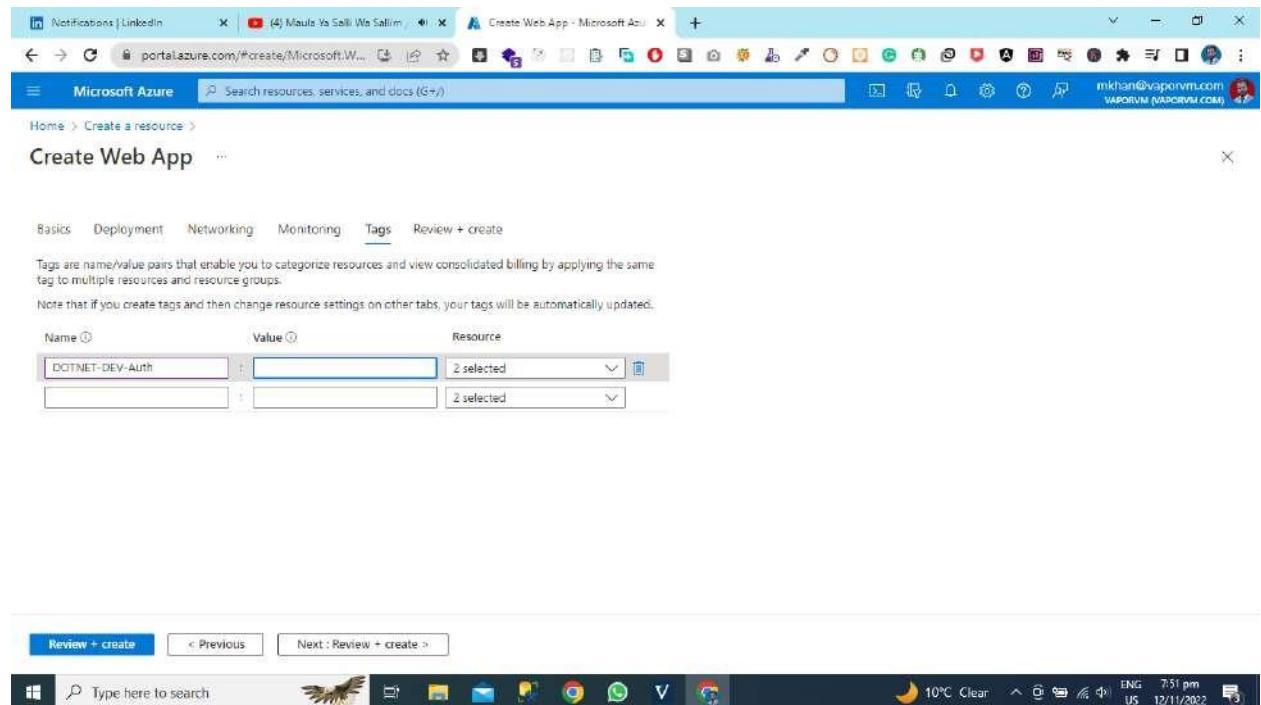
Azure Monitor application insights is an Application Performance Management (APM) service for developers and DevOps professionals. Enable it below to automatically monitor your application. It will detect performance anomalies, and includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app.



Select the Tags Options

Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups.

Note that if you create tags and then change resource settings on other tabs, your tags will be automatically updated.



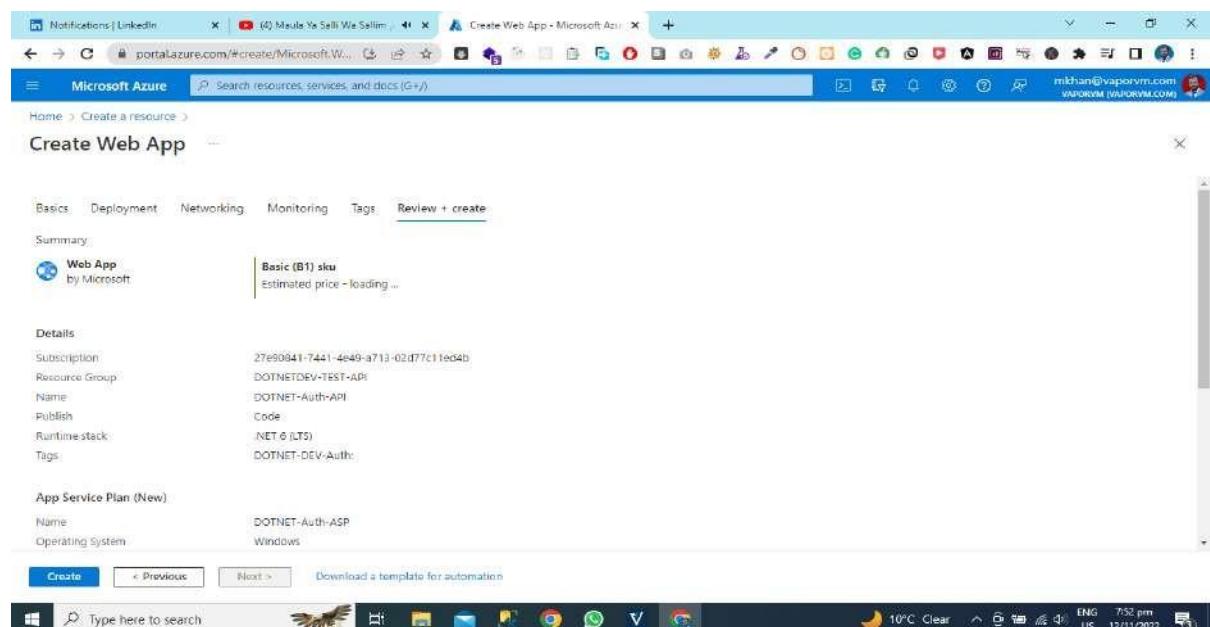
The screenshot shows the Microsoft Azure portal's 'Create Web App' wizard. The 'Tags' tab is currently selected. Two tags are listed:

Name	Value	Resource
DOTNET-DEV-Auth	2 selected	
		2 selected

Below the tags, there are 'Review + create' and 'Next : Review + create >' buttons. The status bar at the bottom shows the date and time as 12/11/2022, 7:51 pm.

Review and Create

Now Review all your Steps and Create the Azure App.



The screenshot shows the Microsoft Azure portal's 'Create Web App' wizard with the 'Review + create' tab selected. The page displays the following information:

Summary:

- Web App by Microsoft
- Basic (B1) sku
- estimated price - loading ...

Details:

Subscription	27e90841-7441-4e49-a713-03d77c11ed4b
Resource Group	DOTNETDEV-TEST-APL
Name	DOTNET-Auth-API
Publish	Code
Runtime stack	.NET 6 (LTS)
Tags	DOTNET-DEV-Auth

App Service Plan (New):

Name	DOTNET-Auth-ASP
Operating System	Windows

At the bottom, there are 'Create', 'Previous', and 'Next >' buttons. The status bar at the bottom shows the date and time as 12/11/2022, 7:52 pm.

Now hit the `create` button and then Azure app creation process has will start.

App Service Plan (New)

Name:	DOTNET-Auth-ASP
Operating System:	Windows
Region:	UAE North
SKU:	Basic
Size:	Small
ACU:	100 total ACU
Memory:	1.75 GB memory
Tags:	DOTNET-DEV-Auth;

Monitoring

Application Insights	Not enabled
----------------------	-------------

Deployment

Continuous deployment	Not enabled / Set up after app creation
-----------------------	---

Create < Previous Next > Download a template for automation

You can see in the given below picture is application deployment process is in progress.

Microsoft.Web-WebApp-Portal-740951f4-a0dd | Overview

Deployment

Deployment is in progress

Deployment name:	Microsoft.Web-WebApp-Portal-740951f4-a0dd	Start time:	11/12/2022, 7:52:14 PM
Subscription:	Visual Studio Enterprise Subscription - MPN (27e908...)	Correlation ID:	0bb7fcda-007e-4f8c-8427-bf87fa5...
Resource group:	DOTNETDEV-TEST-API		

Deployment details

Resource	Type	Status	Operation details
No results.			

Give feedback

Tell us about your experience with deployment

Microsoft Defender for Cloud

Free Microsoft tutorials

Work with an expert

Now you can see that our deployment process succeeded and we can go to the Created resource.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Notifications' (LinkedIn), 'Search resources, services, and docs (G+)', and the user's email 'mkhan@vaporvm.com'. The main content area displays the 'Microsoft.Web-WebApp-Portal-740951f4-a0dd | Overview' page. A prominent green checkmark indicates 'Your deployment is complete'. Deployment details show: Deployment name: Microsoft.Web-WebApp-Portal-740951f4-a0dd, Subscription: Visual Studio Enterprise Subscription – MPN (27e909...), Resource group: DOTNETDEV-TEST-API, Start time: 11/1, Correlation ID: f1. Below this, 'Deployment details' and 'Next steps' sections are visible. To the right, a 'Notifications' sidebar shows a successful deployment event: 'Deployment succeeded' for 'Deployment : Microsoft.Web-WebApp-Portal-740951f4-a0dd' to resource group 'DOTNETDEV-TEST-API'. The event was successful and occurred 14 minutes ago. At the bottom, there are 'Go to resource' and 'Pin to dashboard' buttons. The taskbar at the bottom of the screen shows the Windows Start button, a search bar, pinned icons for File Explorer, Mail, and Edge, and system status indicators like battery level and network connection.

Our Azure App Creation Process Has Been Done.

The screenshot shows the Microsoft Azure portal interface. The navigation bar is identical to the previous one. The main content area displays the 'DOTNET-Auth-API - Microsoft Azure' overview page. The left sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Microsoft Defender for Cloud', 'Events (preview)', 'Deployment', 'Quickstart', 'Deployment slots', 'Deployment Center', 'Settings', 'Configuration', 'Authentication', and 'Application Insights (preview)'. The 'Properties' tab is selected in the center pane. Key details shown include: Resource group (move) : DOTNETDEV-TEST-API, Status : Running, Location (move) : UAE North, Subscription (move) : Visual Studio Enterprise Subscription – MPN, Subscription ID : 27e90841-7441-4e49-a713-02d7?c11ed4b, Tags (edit) : DOTNET-DEV-Auth, Name : DOTNET-Auth-API, Publishing model : Code, Runtime Stack : Dotnet - v6.0, URL : https://dotnet-auth-api.azurewebsites.net, App Service Plan : DOTNET-Auth-ASP, Operating System : Windows, and Health Check : Not Configured. The 'Monitoring' tab is also visible. The taskbar at the bottom of the screen is identical to the previous one.

Now you need to hit the Azure App URL.

The screenshot shows the Microsoft Azure portal interface. The main title bar says 'DOTNET-Auth-API - Microsoft Azure'. The left sidebar has sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment, Quickstart, Deployment slots, Deployment Center, Configuration, and Authentication. The 'Overview' section is selected. It displays details such as Resource group (move) : DOTNETDEV-TEST-APPI, Status : Running, Location (move) : UAE North, Subscription (move) : Visual Studio Enterprise Subscription – MPN, Subscription ID : 27e90841-7441-4e49-a713-02d77c11ed4b, and Tags (edit) : DOTNET-DEV-Auth. Below this are tabs for Properties, Monitoring, Logs, Capabilities, and Notifications. On the right, there are sections for Application Insights (Name: Enable Application Insights) and Hosting (Plan Type: App Service plan). The bottom navigation bar includes 'Deployment Center' and 'Quickstart'.

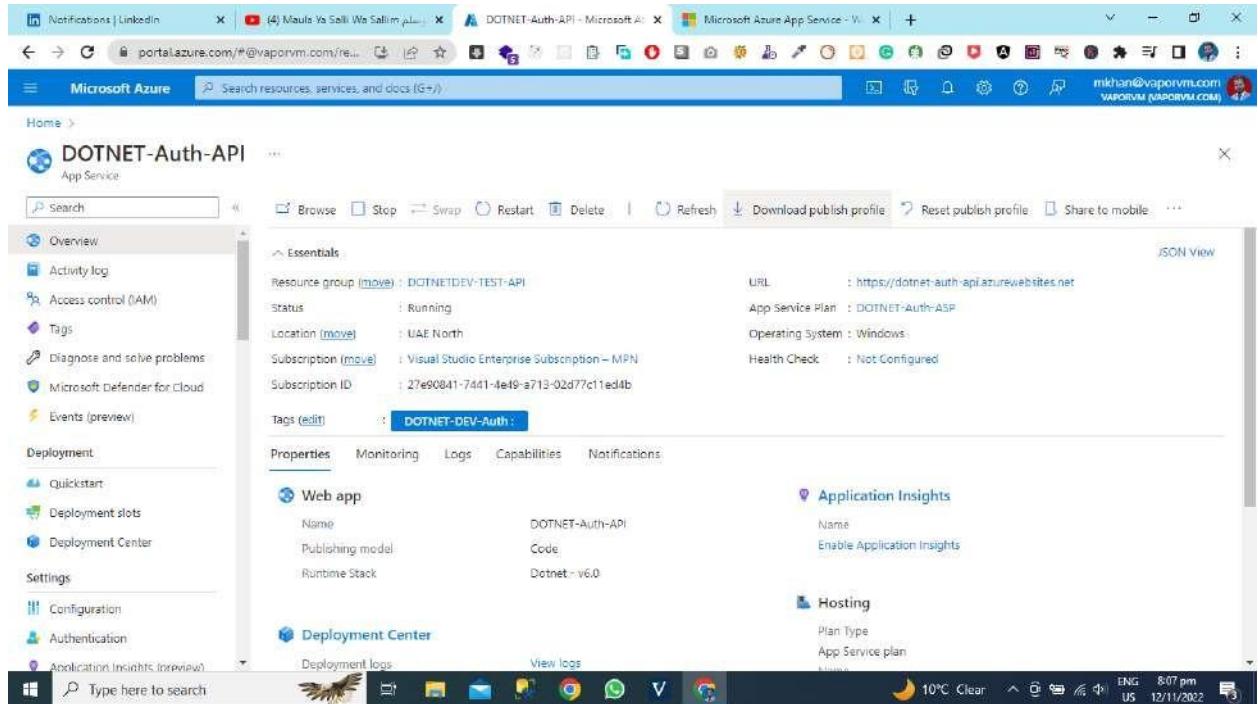
Application Status

Now You Can See that Our Application is Live on the Azure Cloud.

The screenshot shows a web browser window with the URL 'dotnet-auth-api.azurewebsites.net'. The page title is 'Microsoft Azure'. The main content area says 'Your web app is running and waiting for your content'. It includes a message: 'Your web app is live, but we don't have your content yet. If you've already deployed, it could take up to 5 minutes for your content to show up, so come back soon.' To the right is an illustration of a computer monitor displaying a globe, with two blue speech bubbles containing code snippets '</>' and '< />'. Below this, there are links for 'Supporting Node.js, Java, .NET and more'. At the bottom, there are sections for 'Haven't deployed yet?' and 'Starting a new web site?'. The bottom navigation bar includes 'Deployment Center' and 'Quickstart'.

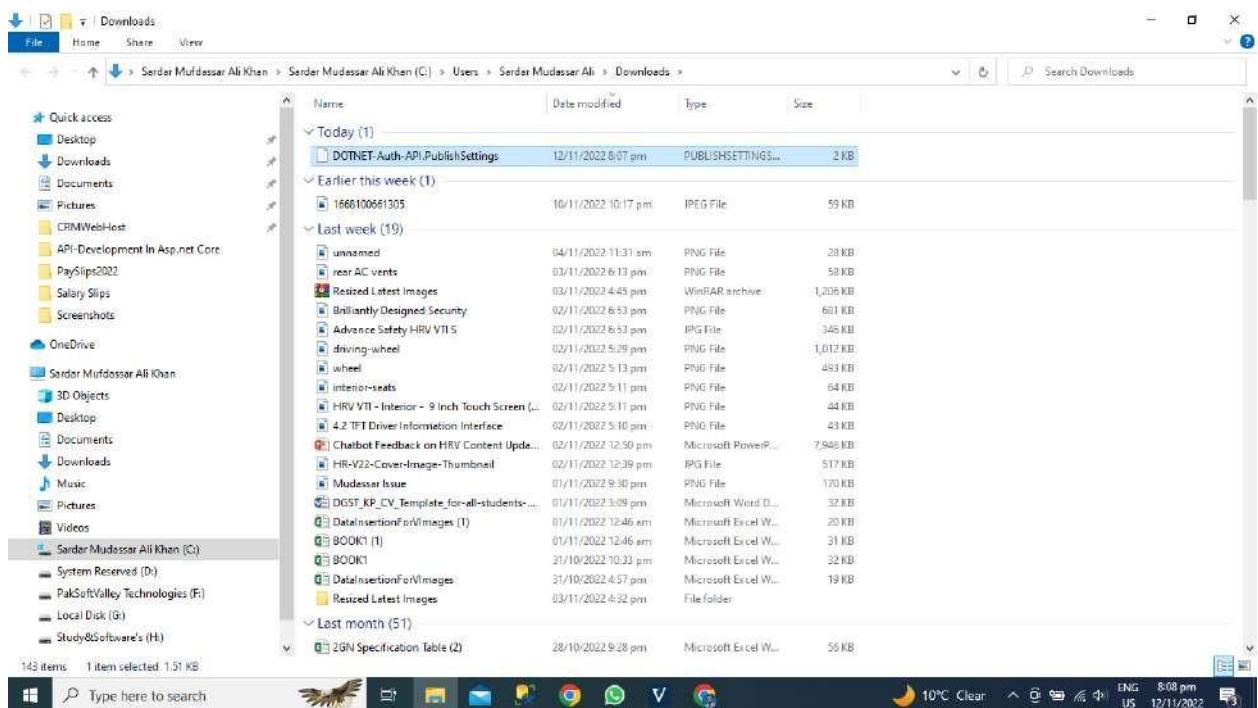
Get the publish profile.

Now you need to download the publish profile for deployment of Asp.net core application using Visual Studio.



The screenshot shows the Microsoft Azure portal interface. In the center, there's a detailed view of the 'DOTNET-Auth-API' app service. On the left, a sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, and Events (preview). Below this, sections for Deployment (Quickstart, Deployment slots, Deployment Center) and Settings (Configuration, Authentication, Application Insights (preview)) are visible. The main content area displays the app service's configuration, including its Resource group (DICTNET-DEV-TEST-API), Status (Running), Location (UAE North), Subscription (Visual Studio Enterprise Subscription – MPN), and Subscription ID (27e90841-7441-4e49-a713-02d77c1ed4b). A 'Tags (edit)' section shows the tag 'DOTNET-DEV-Auth'. Below this, tabs for Properties, Monitoring, Logs, Capabilities, and Notifications are present. To the right, sections for Web app (Name: DOTNET-Auth-API, Publishing model: Code, Runtime Stack: Dotnet - v6.0), Application Insights (Name: Enable Application Insights), and Hosting (Plan Type: App Service plan) are shown. At the bottom, a 'Deployment Center' section includes links for Deployment logs and View logs. The top navigation bar shows several tabs including 'Notifications | LinkedIn', 'Maula Ya Sali Wa Salim pl...', 'DOTNET-Auth-API - Microsoft App...', 'Microsoft Azure App Service - W...', and 'mikhani@vaporvm.com VAPORVM.COM'. The status bar at the bottom right shows the date (12/11/2022), time (8:07 pm), and location (ENG US).

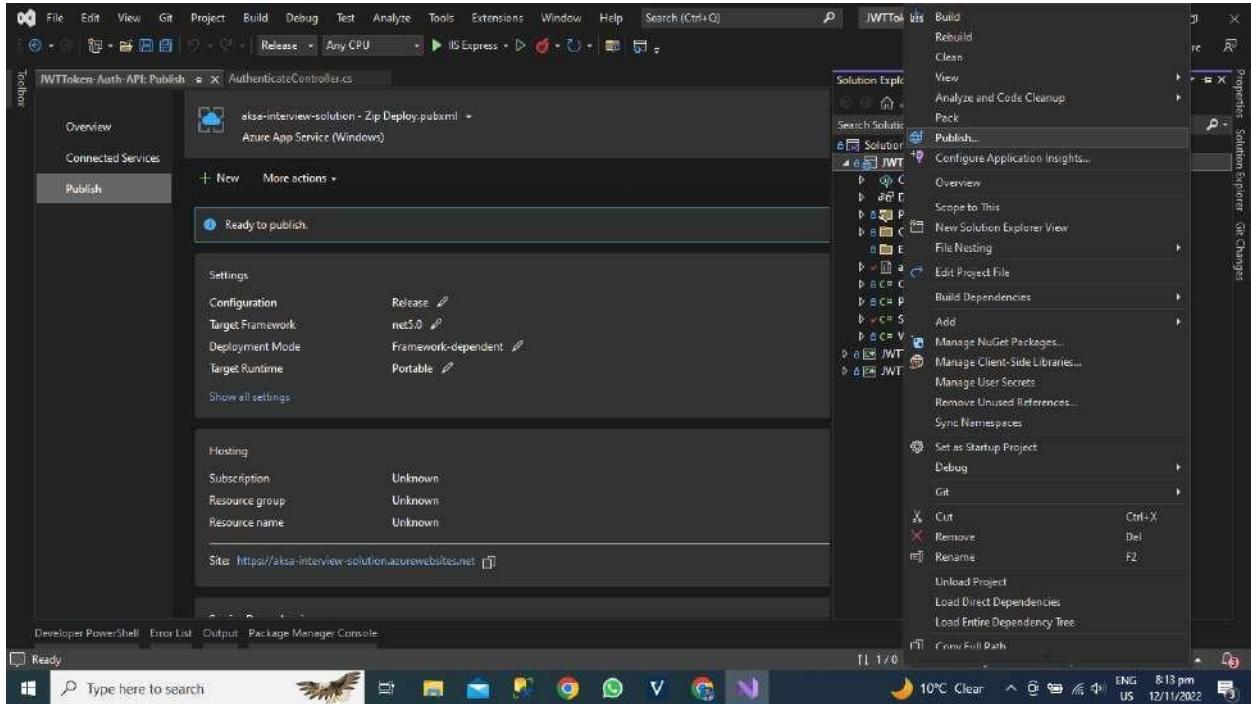
We have Downloaded the Our Application Publish Profile



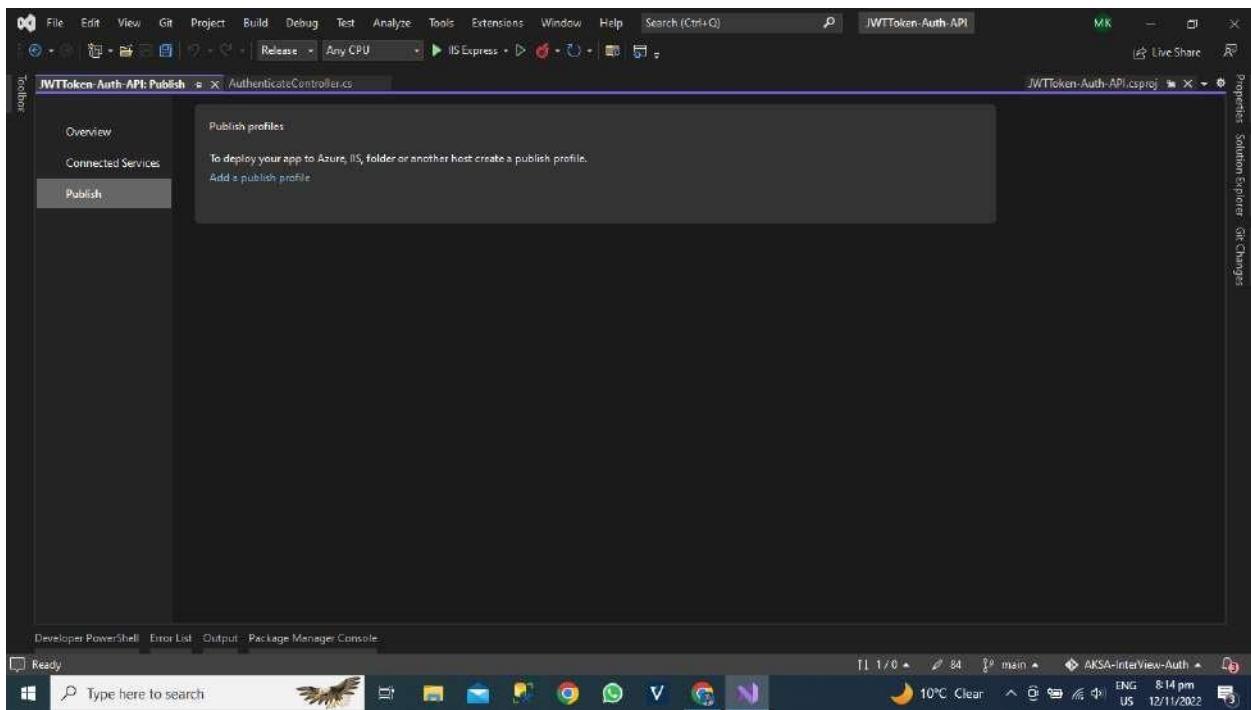
The screenshot shows a Windows File Explorer window with the path 'Sardar Mufaddass Ali Khan > Sardar Mufaddass Ali Khan (C) > Users > Sardar Mufaddass Ali > Downloads'. The 'Downloads' folder contains several files and folders, including 'DOTNET-Auth-API.PublishSettings' (2 KB), '1668100661305' (59 KB), and numerous smaller files and folders from the previous week. The left sidebar shows a tree view of the user's OneDrive and local drives. The status bar at the bottom right shows the date (12/11/2022), time (8:08 pm), and location (ENG US).

Publish the App Using Visual Studio

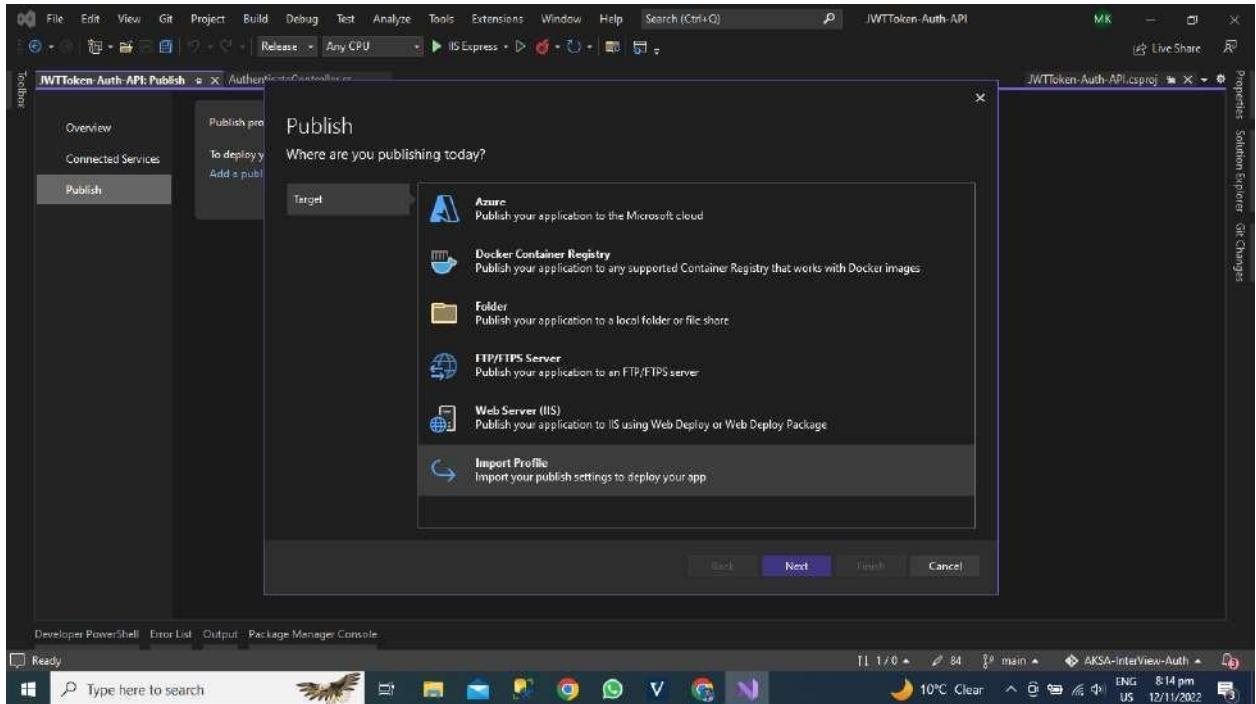
Now we will open the visual studio and then open the Auth API Project after the Right click on project select the publish Option.



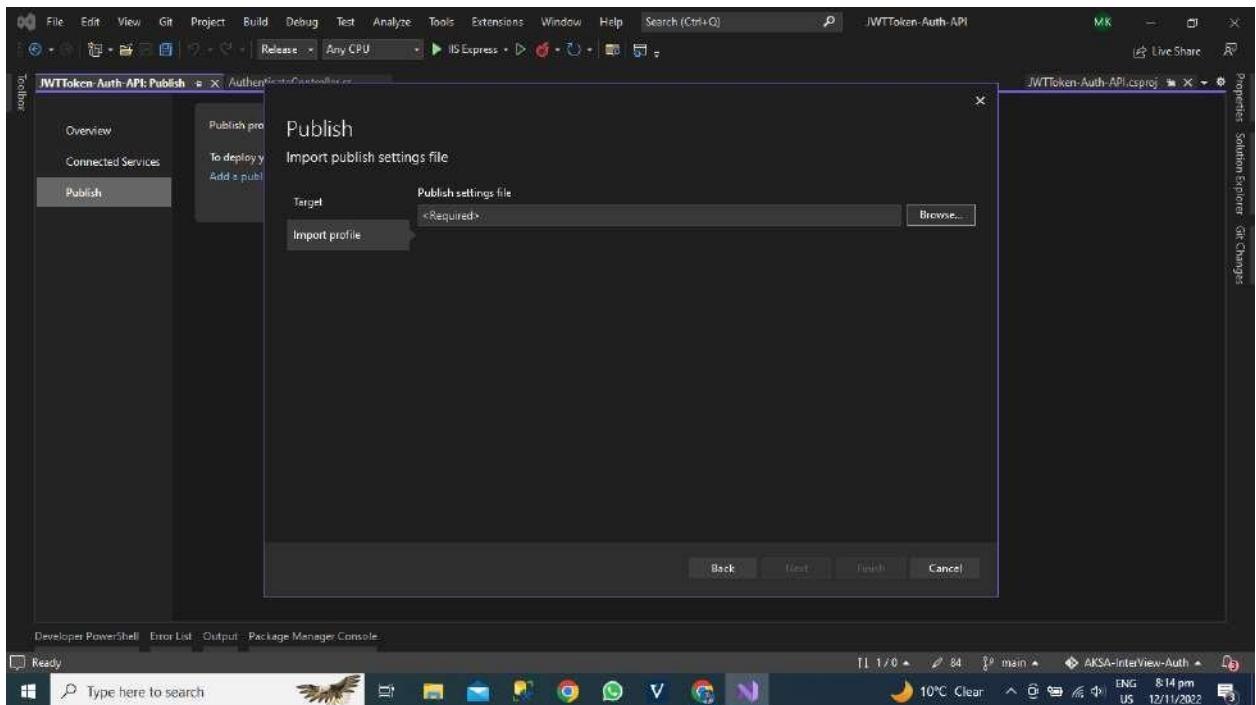
A new window will be visible in front of you and you can see the we have the option for Add Publish Profile. Click the button add publish profile.



Now you can see that we have the option for Import Profile and we will import the profile that we have downloaded from the Azure App Service portal. Click the Import Profile Option.

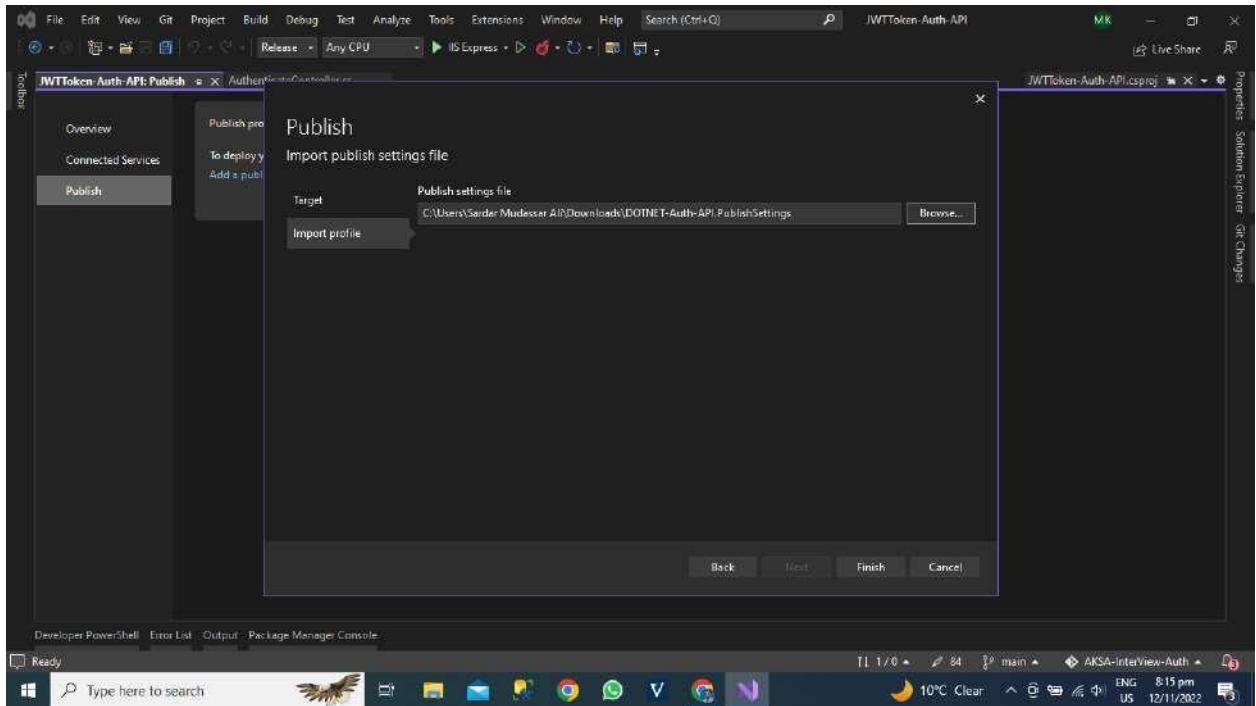


Browse the file for Azure App Publish Profile where you have saved the file.

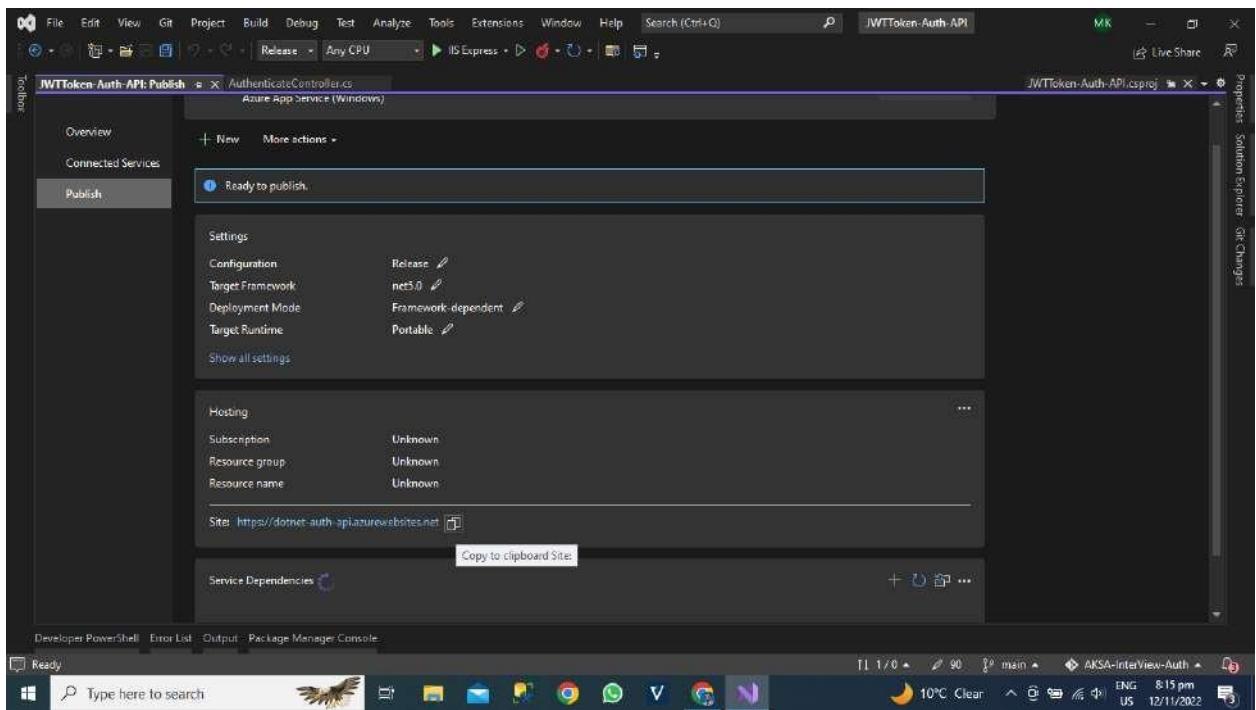


Select the Publish Profile

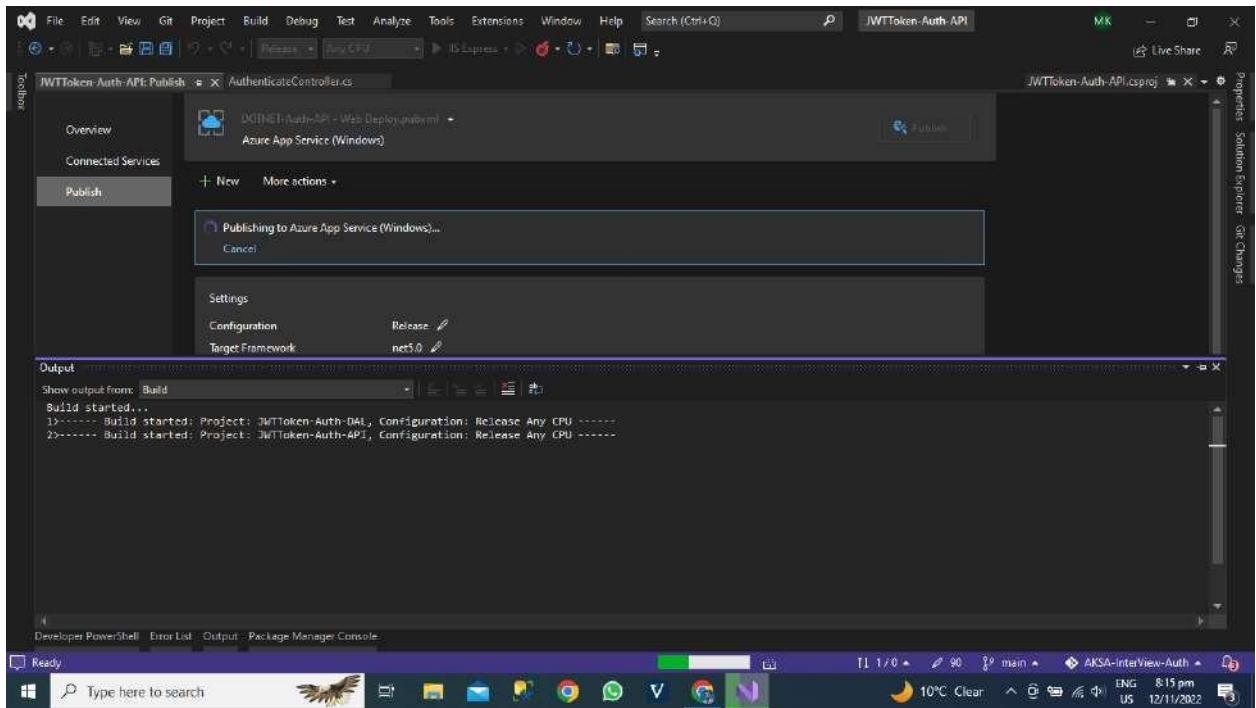
I have selected the file from the downloads folder now click the finish button after some processing visual studio will get the all information the azure portal for our Azure App.



Now you can see that visual studio get all the information about our Azure App you can see the URL of our application.

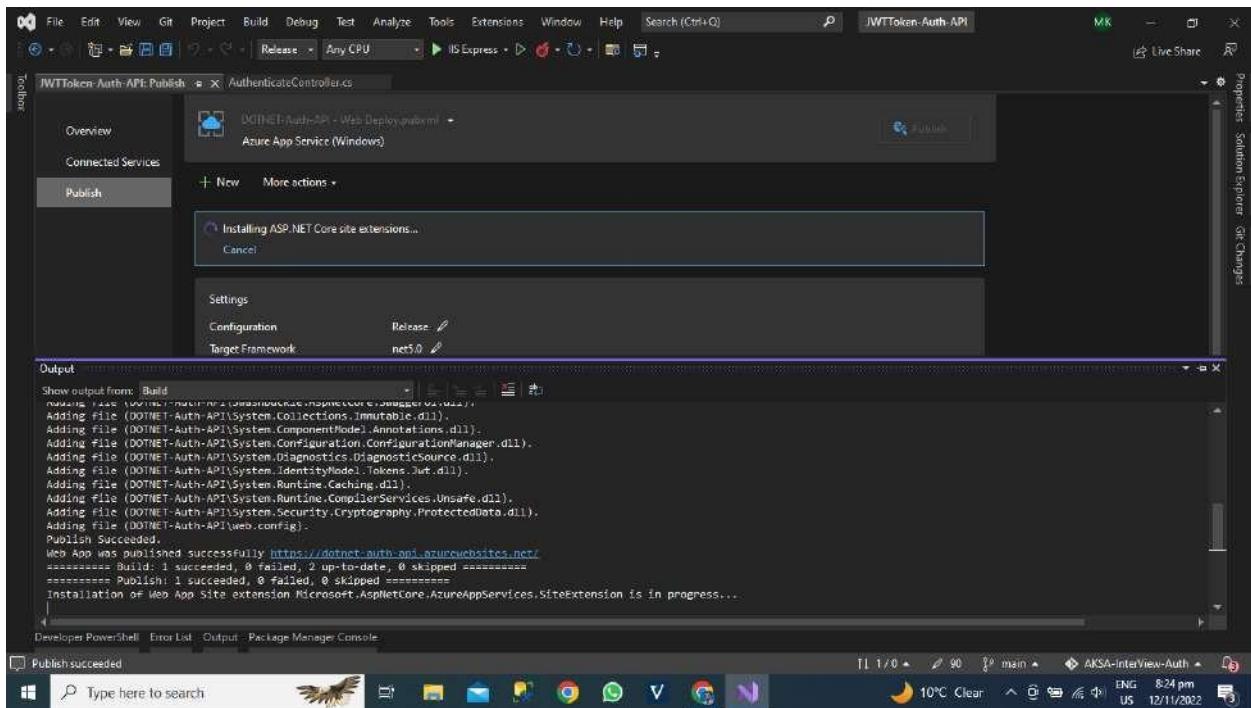


Now Hit the Publish Button after that publishing of our application on azure cloud has been started.



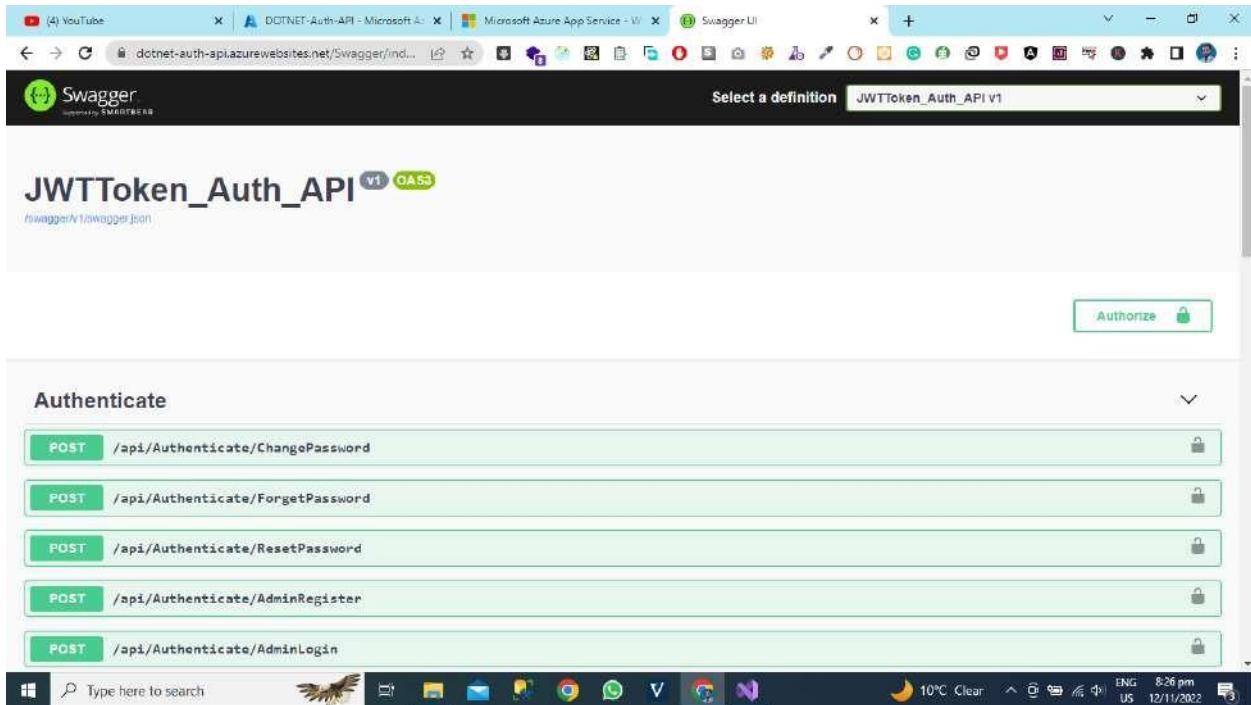
Deployment Status

Our Publication on Azure Cloud has been done. You can see the publication status successful.



Application Deployed on Azure Cloud

Now Hit the URL Of Azure App after URL Just Write URL OF YOUR
APPLICATION/Swagger You can see that our all APIS Are Now Live on Azure Cloud Using
Azure App service.



Chapter 7- Application Deployment on Server.

- ✓ Introduction
- ✓ Deployment Procedure
- ✓ Server Configuration
- ✓ Deployment Steps
- ✓ Conclusion

Introduction

In this chapter, we will learn about how to deploy the asp.net core 5 websites on a server using visual studio 2019.

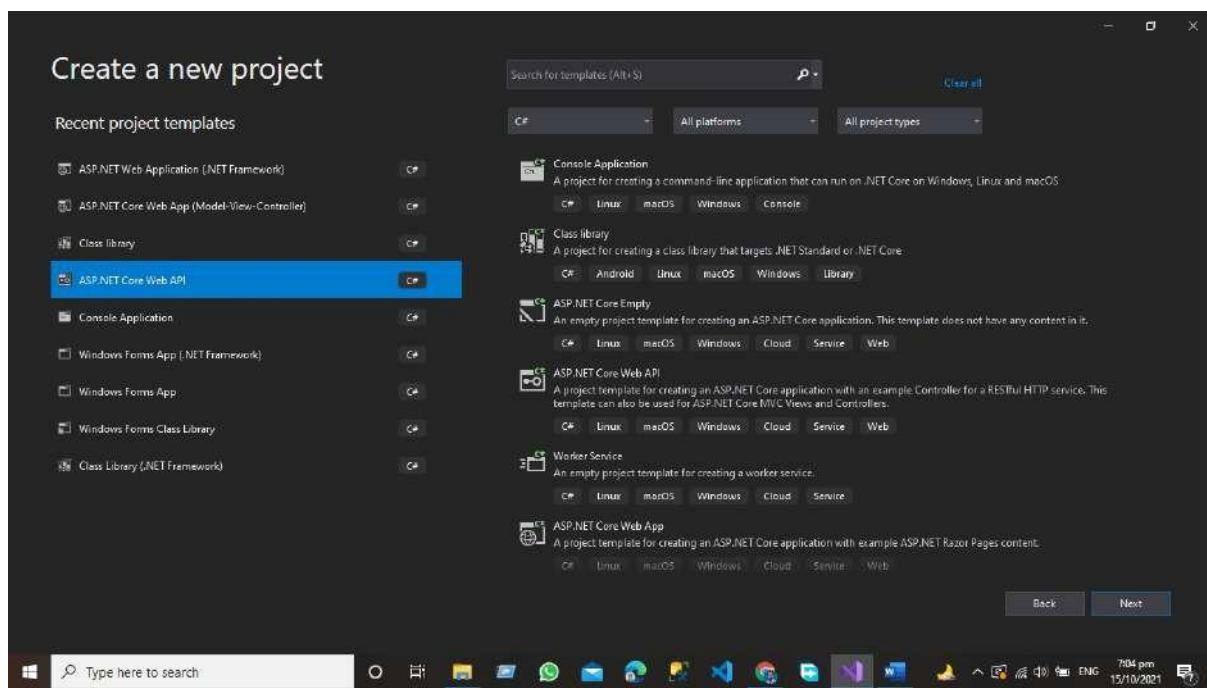
Step 1 Account Setup on Server

The first step for deployment of asp.net core web API is to create an account on the hosting provider website that supports the asp.net web application framework and then follow the steps in the given chapter.

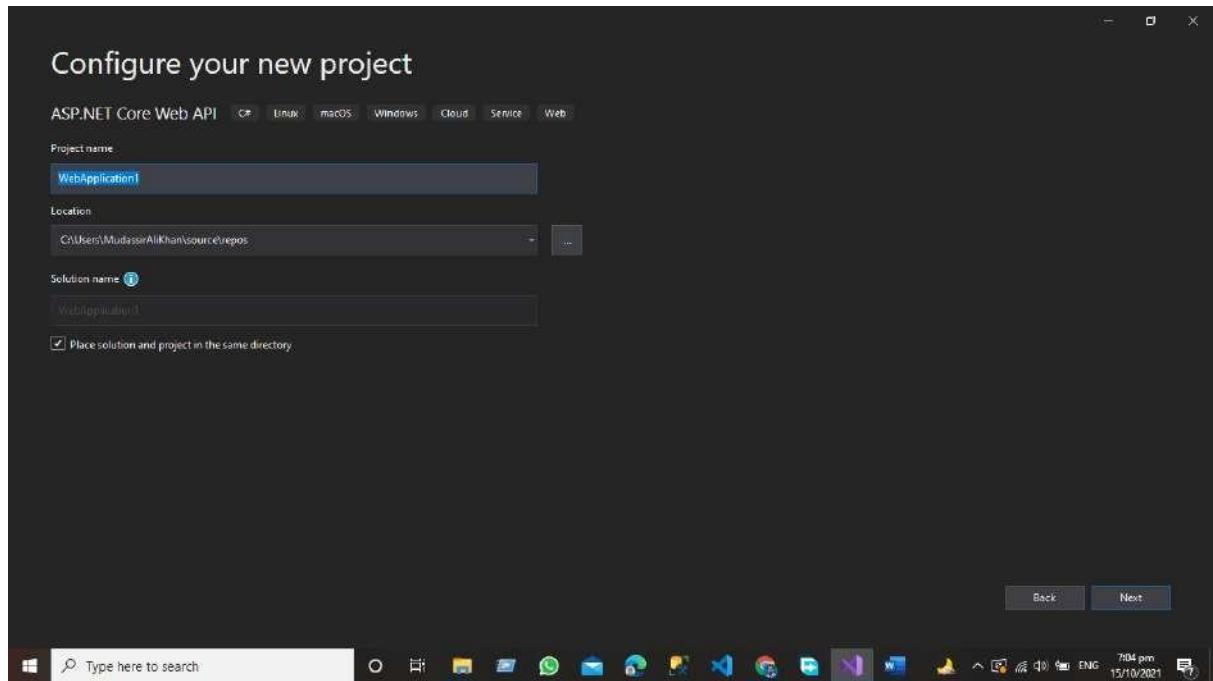
- Create a website in the hosting panel.
- Then Create the SQL Server Website using the Hosting Panel

Step 2 Create the project asp.net Core 5 Web API using Visual Studio

Create the project using visual studio and select the asp.net core version .Net 5.0 Current.



Give the Name of your project that you want to develop My Project is ONP (Online Prescription Application)



Step 3 Set the Live Server Database Connection string in your application appsetting.json file

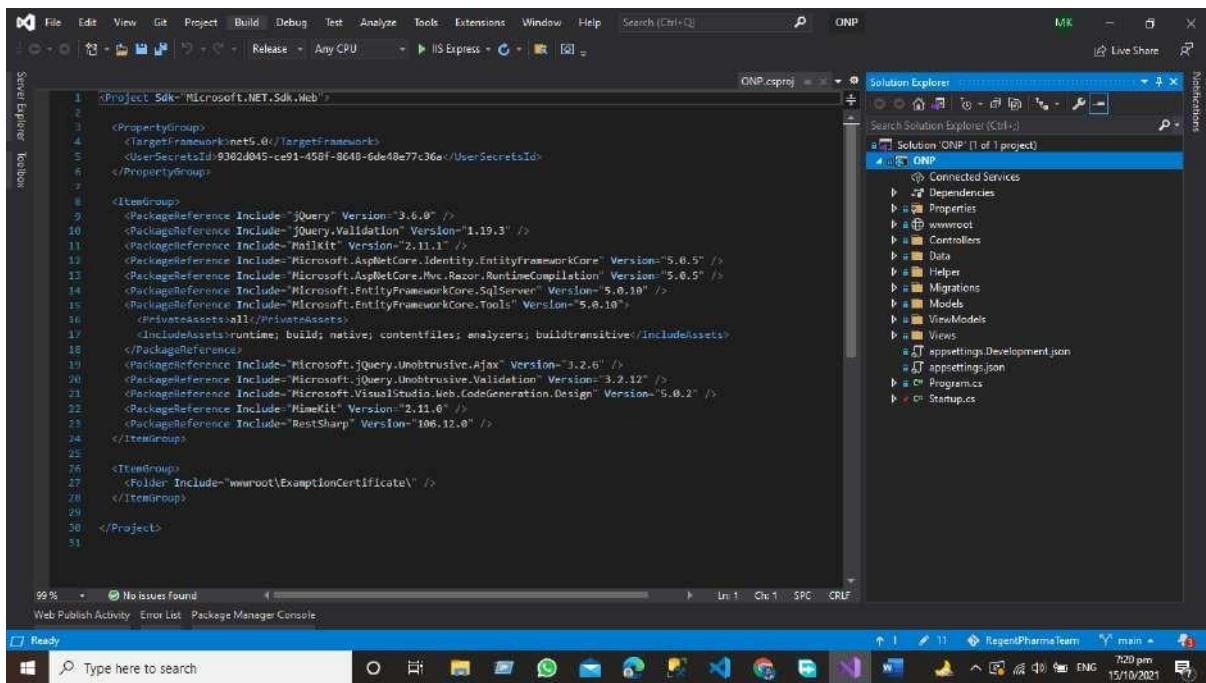
{

```
"DefaultConnection": "Data Source=SQL5108.site4now.net;Initial Catalog=YOUR ONLINE DB NAME;User Id=YOUR ONLINE DB USERNAME;Password=YOUR ONLINE DB PASSWORD"

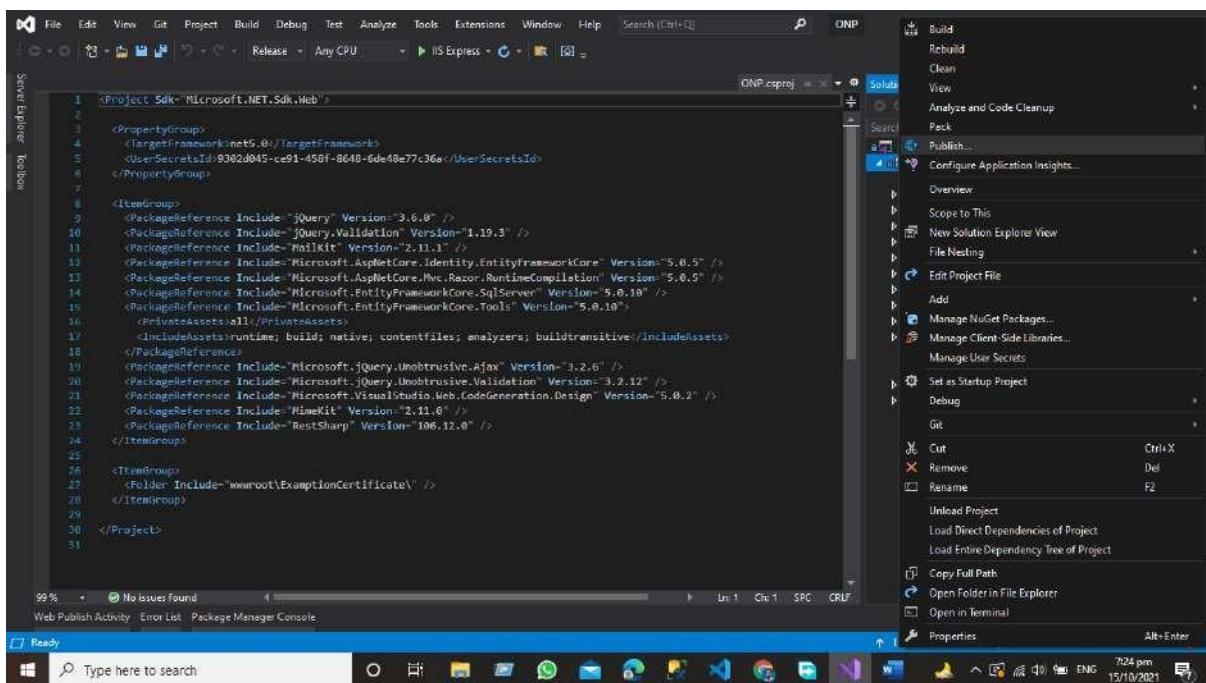
},
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft": "Warning",
    "Microsoft.Hosting.Lifetime": "Information"
  }
},
"AllowedHosts": "*"
}
```

Step 4 Publish Your Project

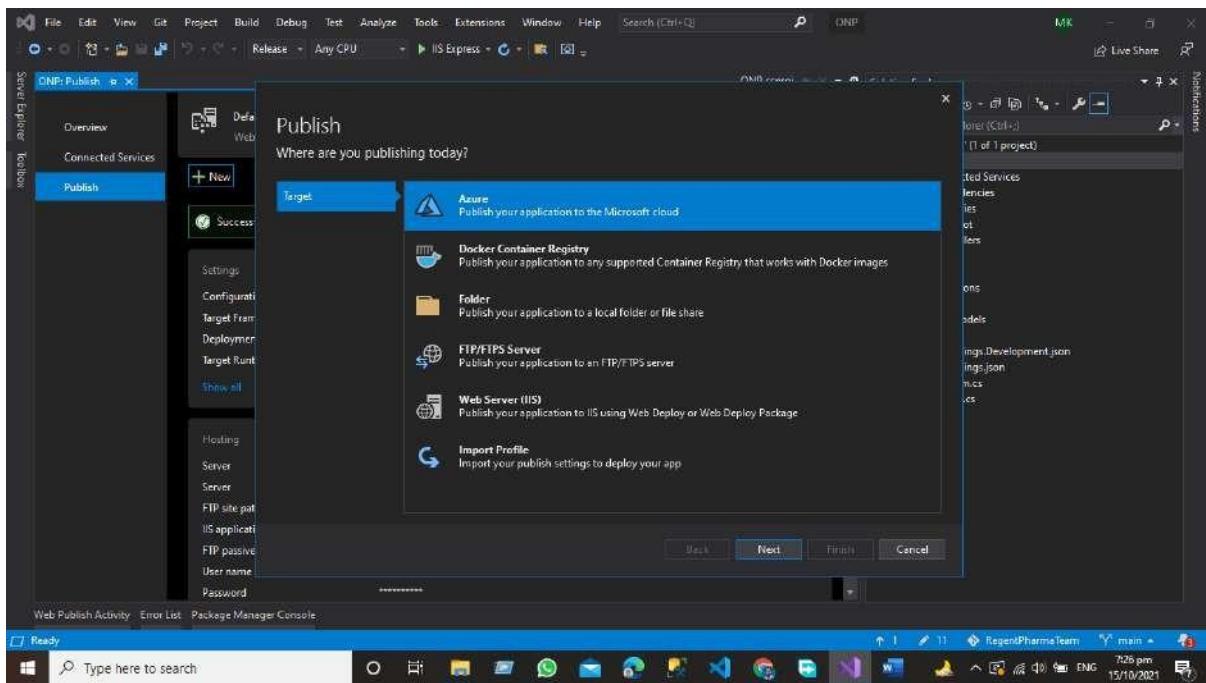
After Creating your project and Final Testing from QA Section now it's time to deploy our website Now go to solution explorer in visual studio 2019.



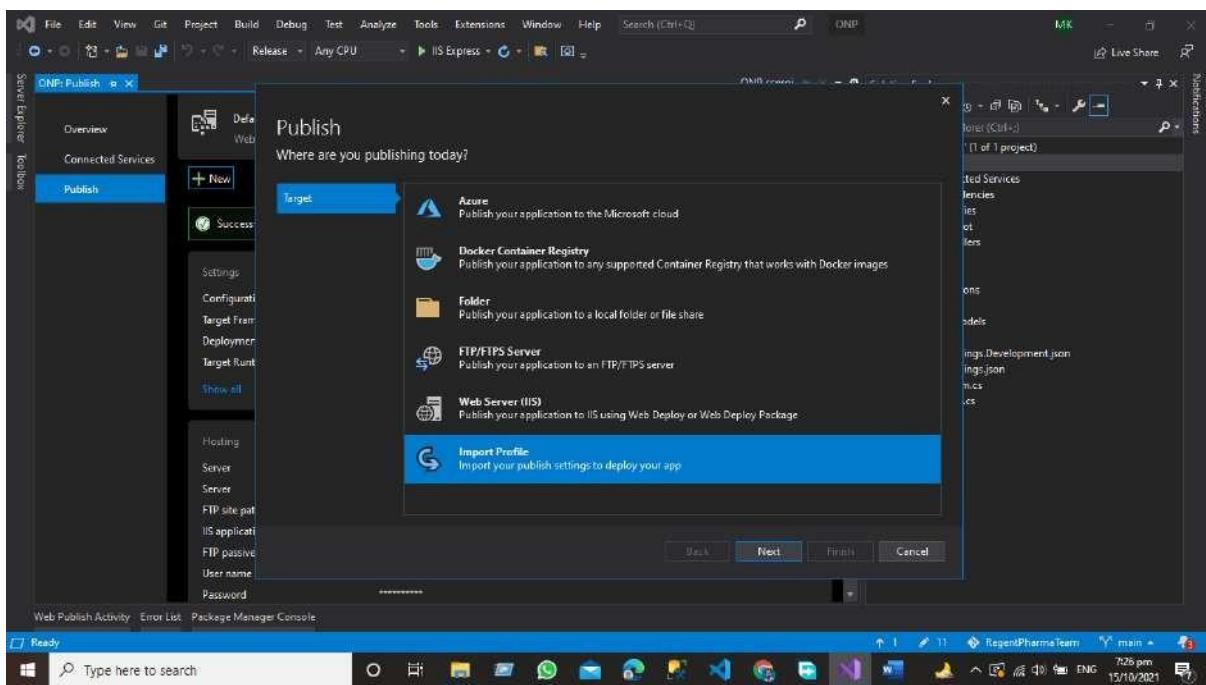
Now Right Click on your Project and select the option to publish.



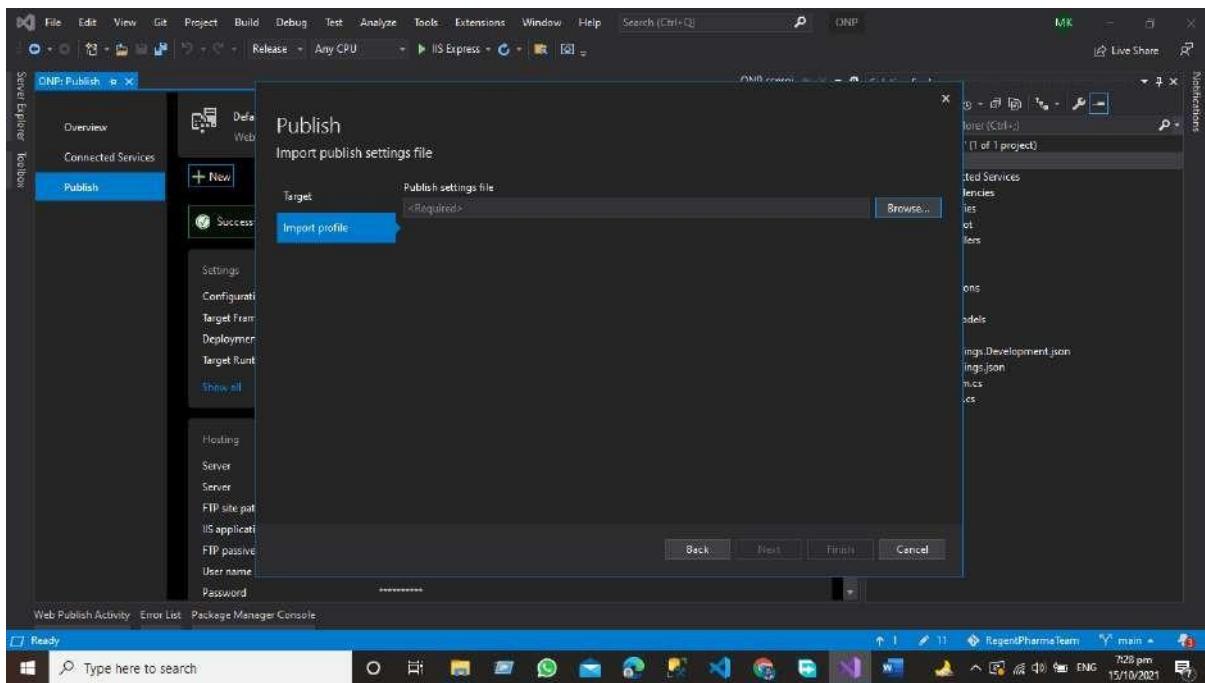
Click the Publish option a new window will appear now select the project publication option here you have 5 to 6 options given below.



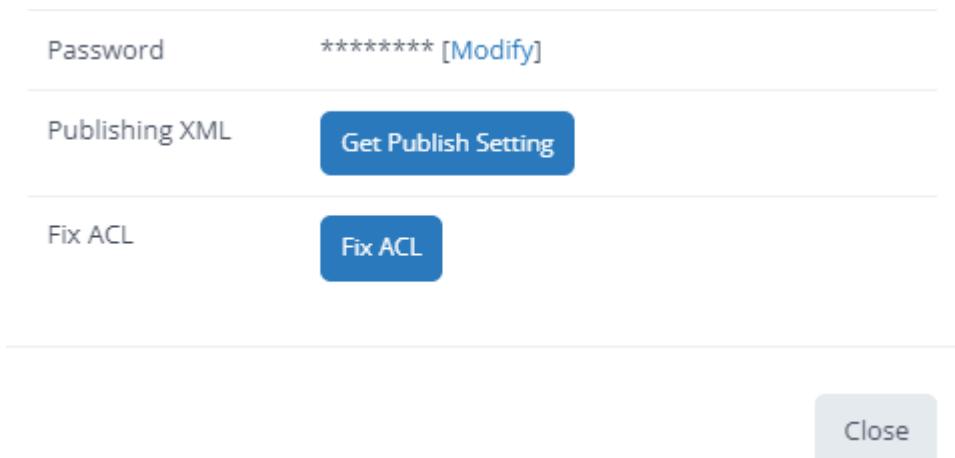
Now select the option Import to publish profile.



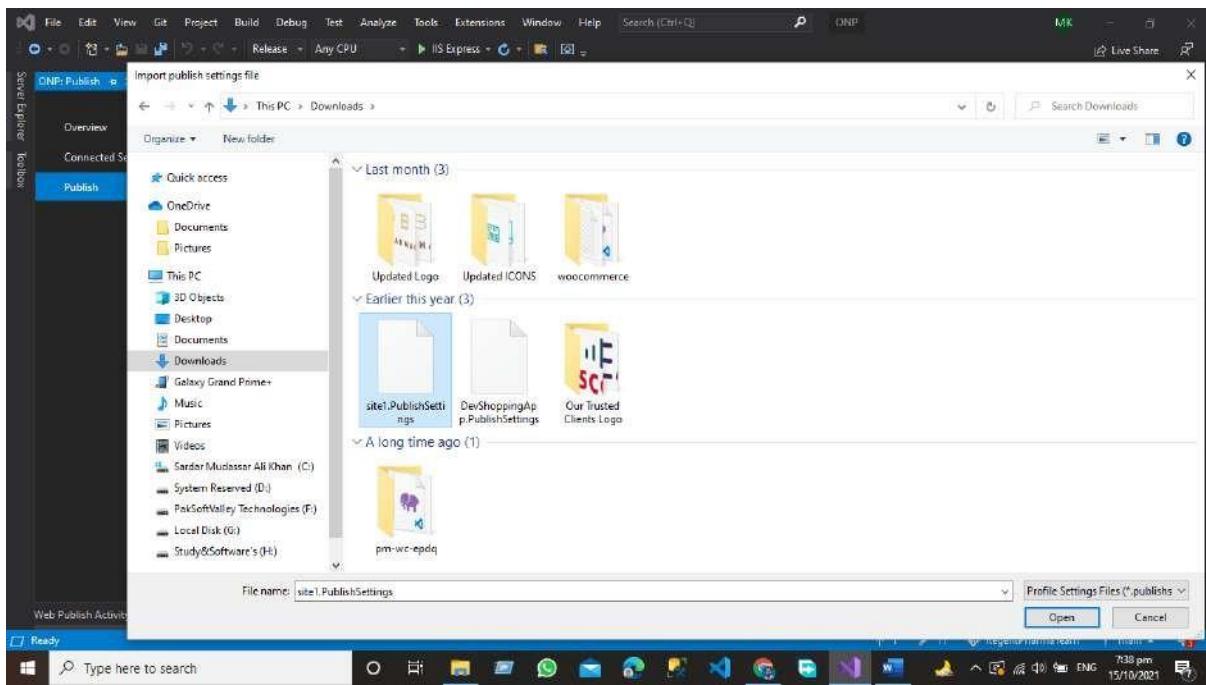
After Selecting the Option Import Profile click next.



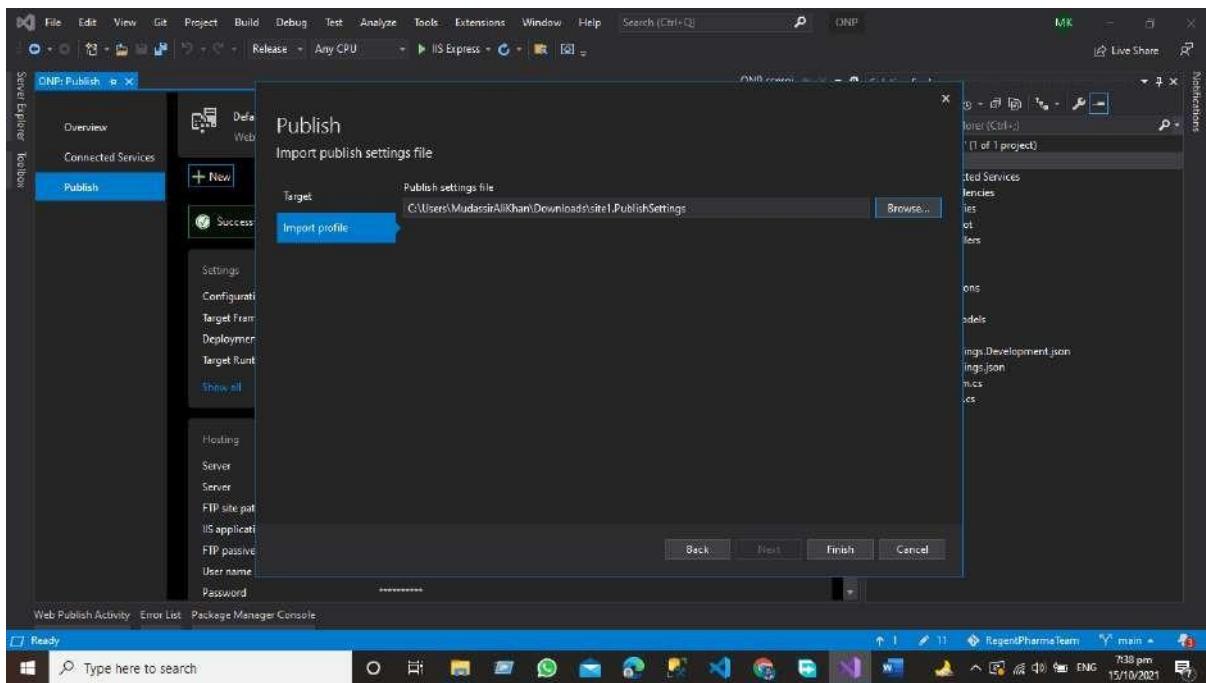
But here question will raise in your mind how to get the publish profile just login to the hosting proving server that supports Asp.net core website applications and then go to the deployment option and get the publish profile file and download it in your local system.



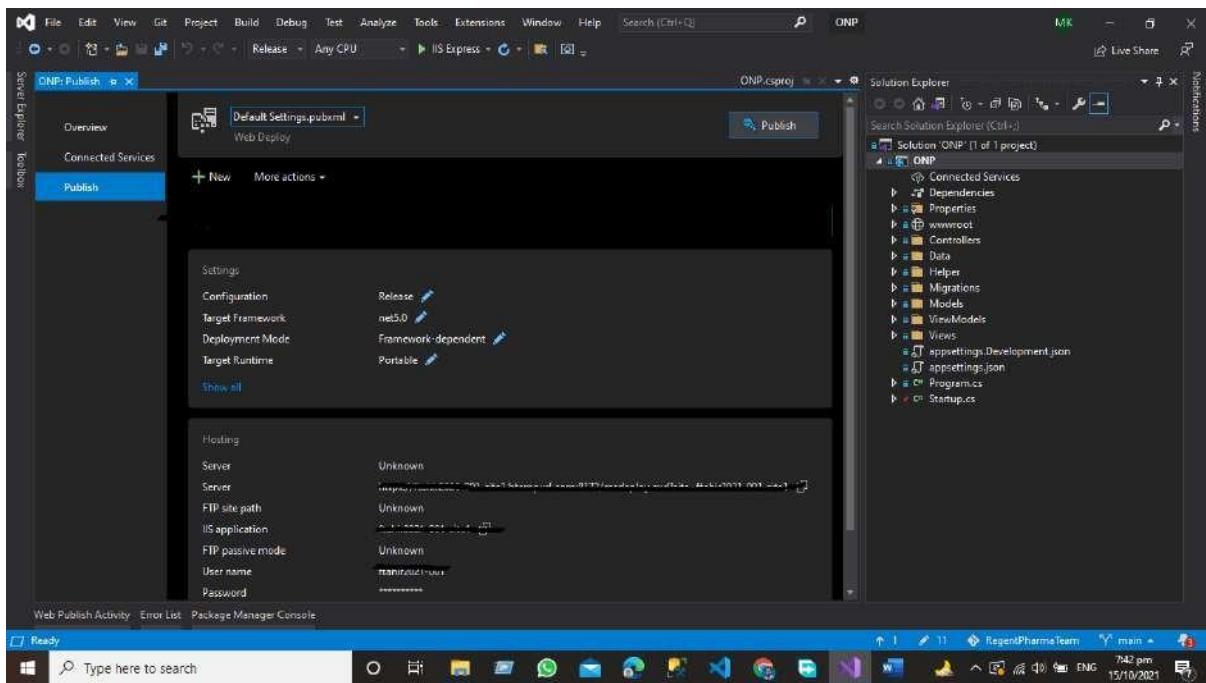
You all get this type of option in your control panel just get the publish profile from and down that file in your system.



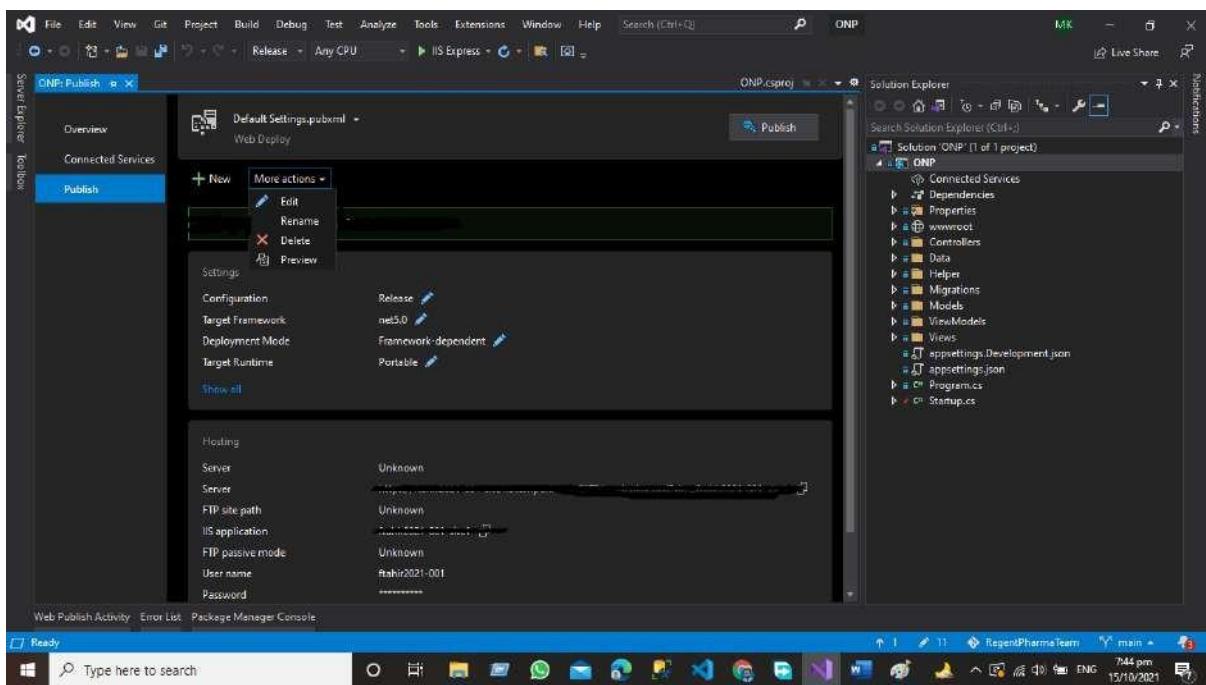
Then upload the file in the visual studio using the browse option from the downloaded location.



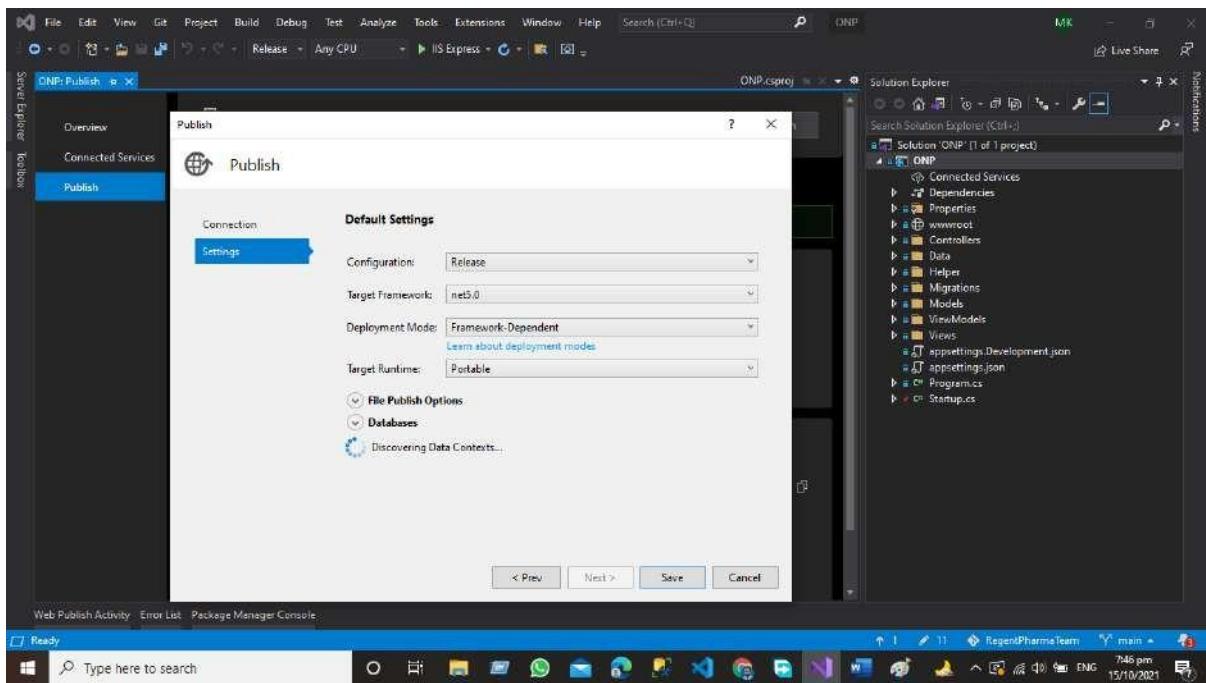
After Uploading the publish profile setting file in the visual studio just click on the Finish option and you will get given the below window.



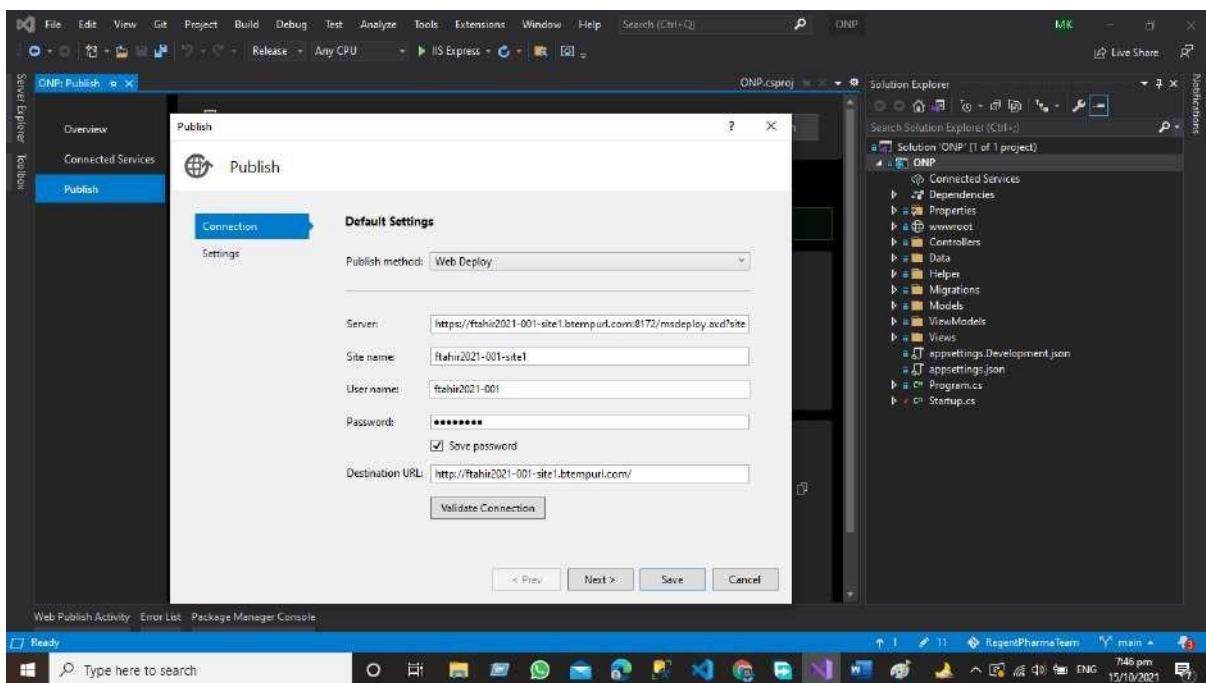
Now Click on More Actions.



Click the Edit Option.



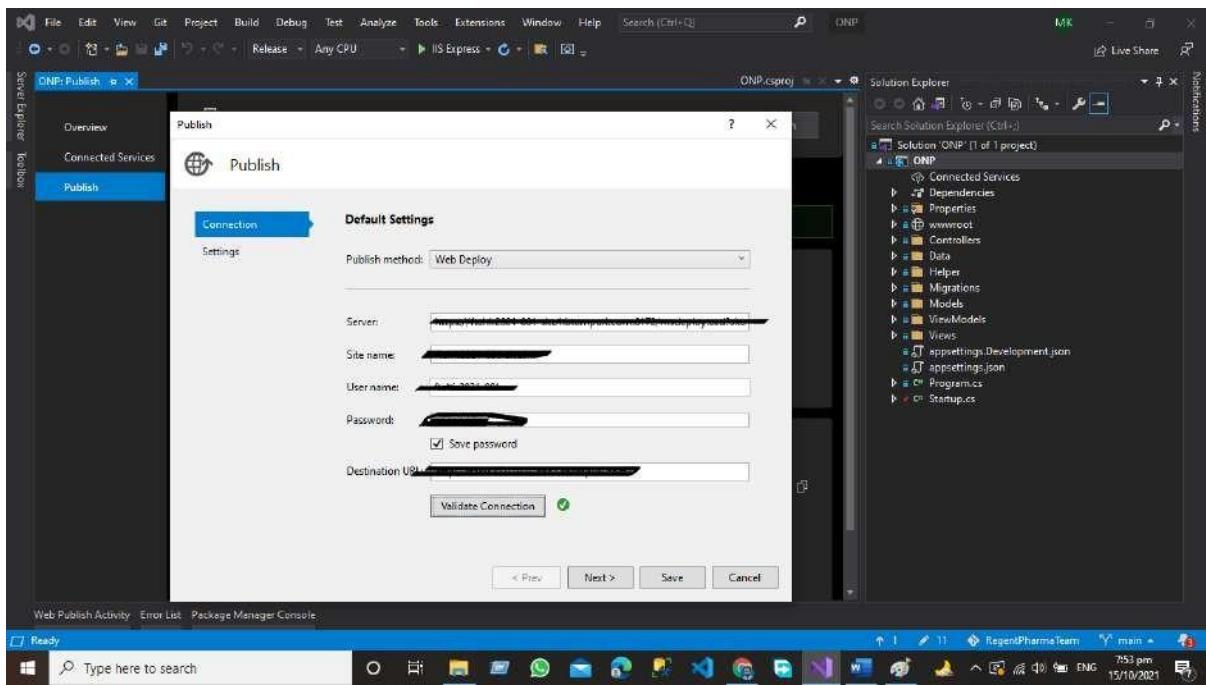
Click On Connection.



All the information will automatically upload fetched from the uploaded publish profile and this information will automatically save in the visual studio for future deployments.

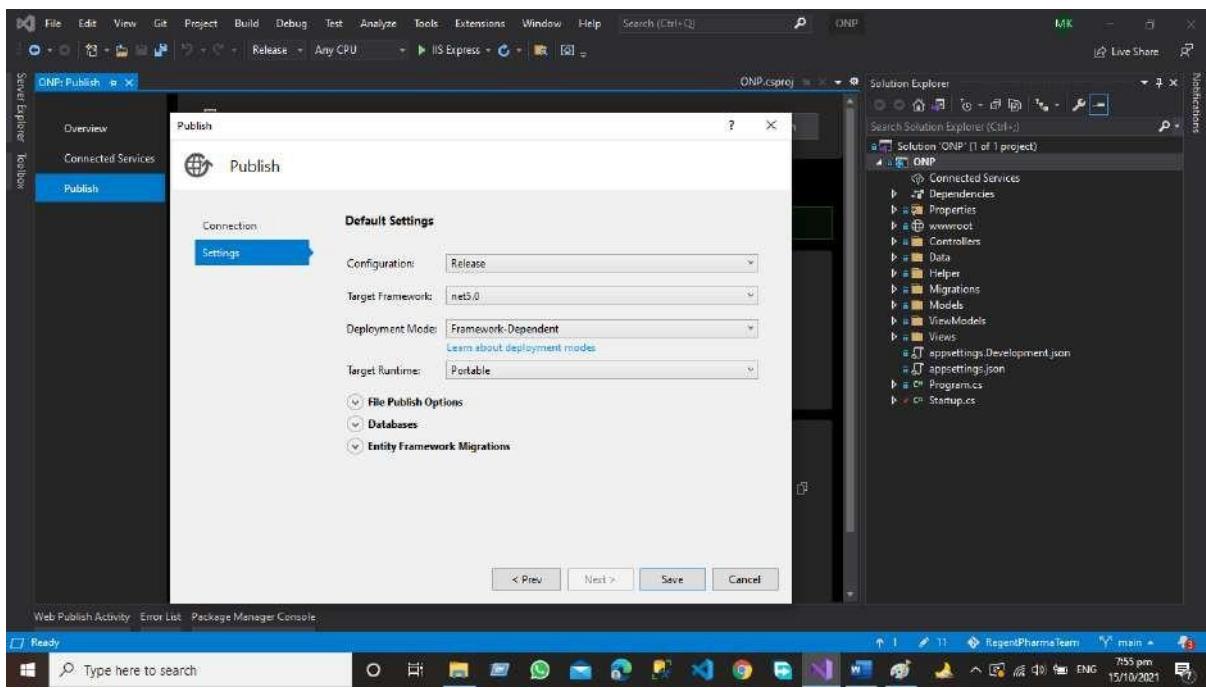
Step 5 Validate the Connection

In this step, visual studio validates the connection with the server and then you can publish your application on the server.

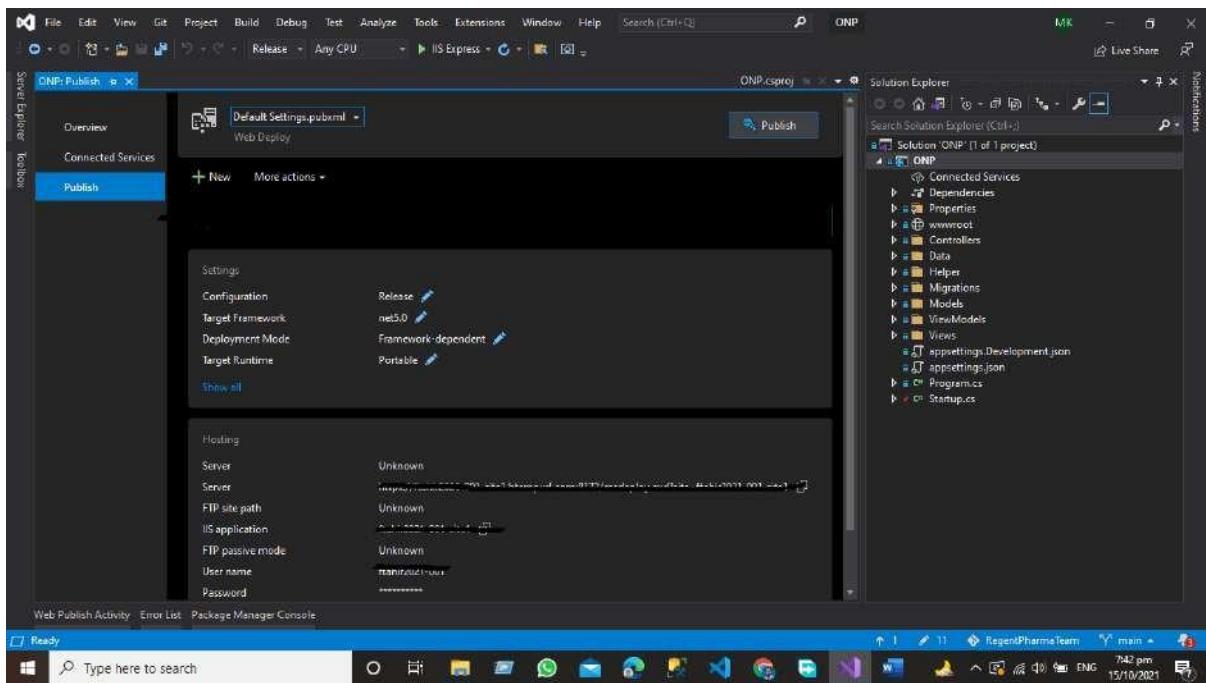


Now you can see that our connection has been validated from visual studio with the server now we are ready to publish the application.

Now Click on Next

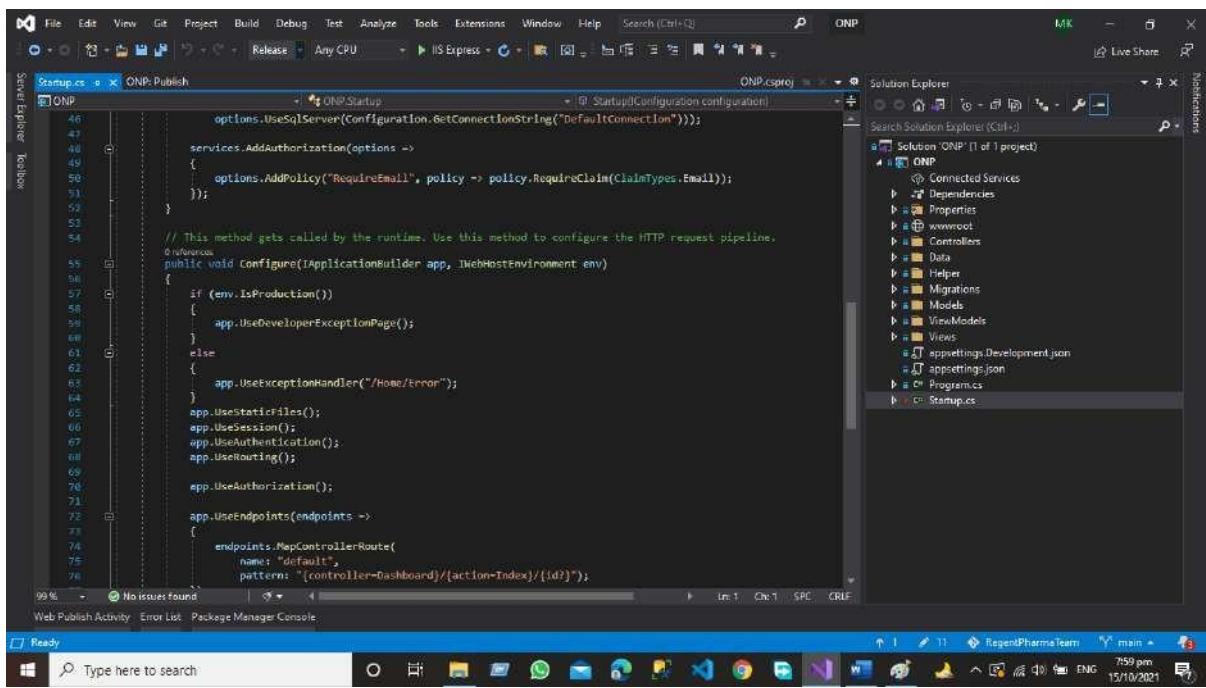


Click Save the Setting

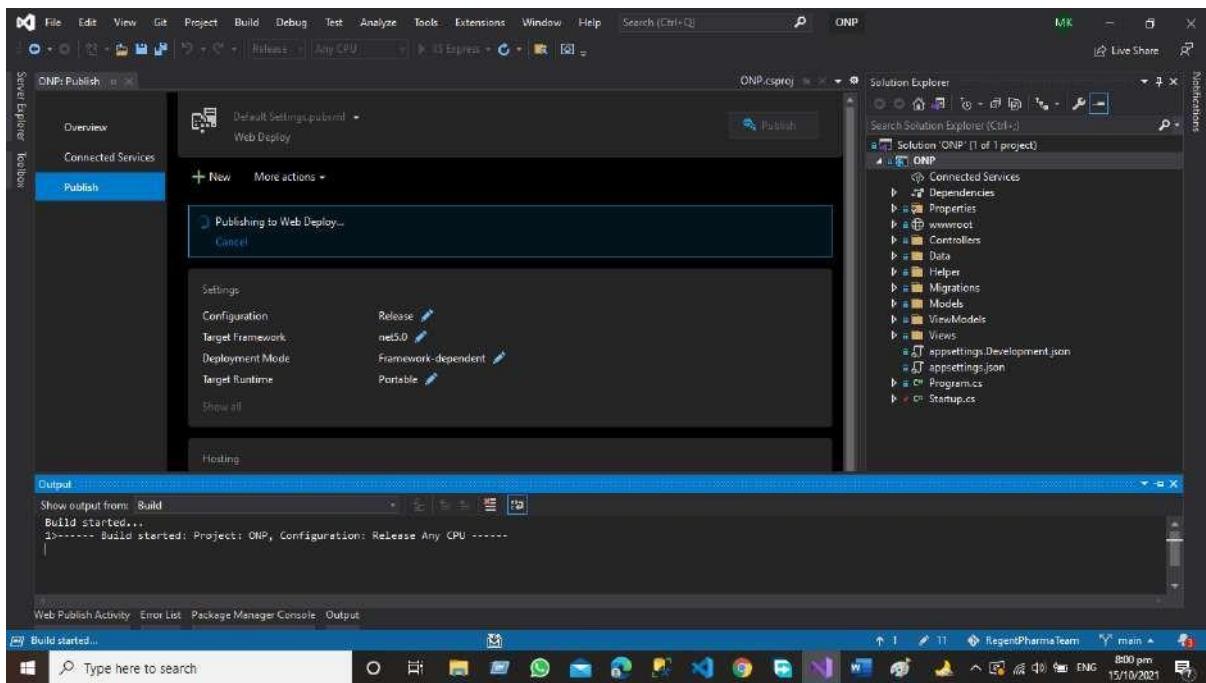


Now Click on Publish

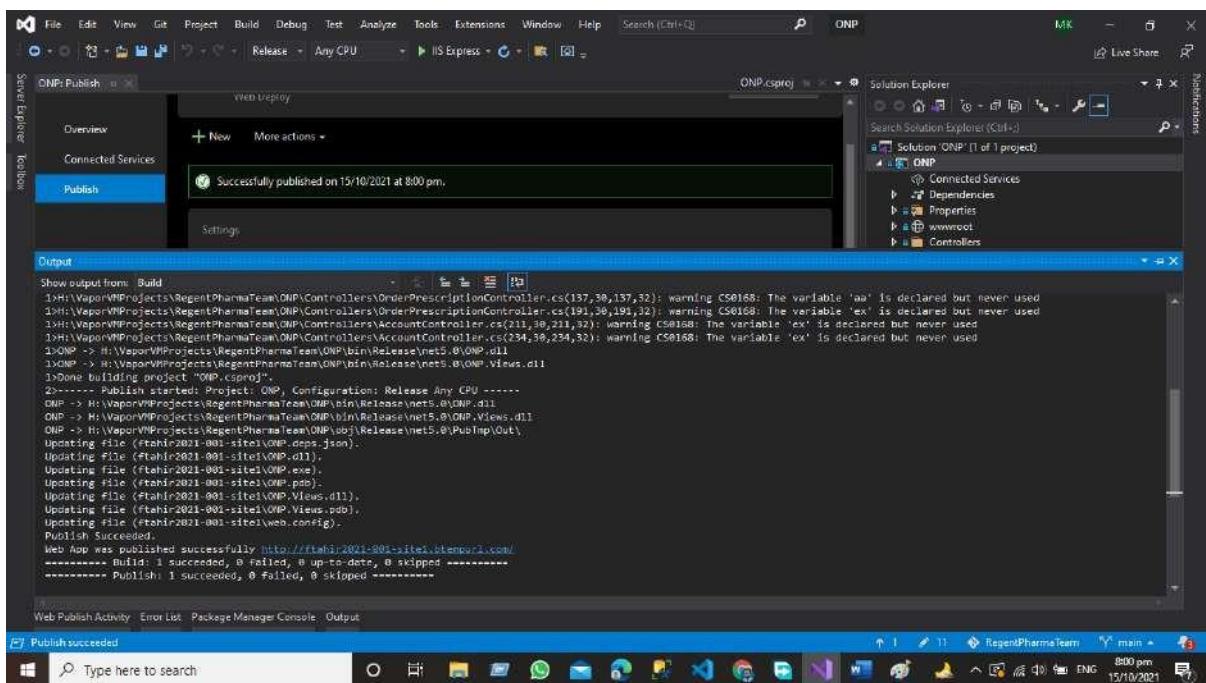
But before the publication just checks your application should be in release mode and the Webhost environment should be in production mode.



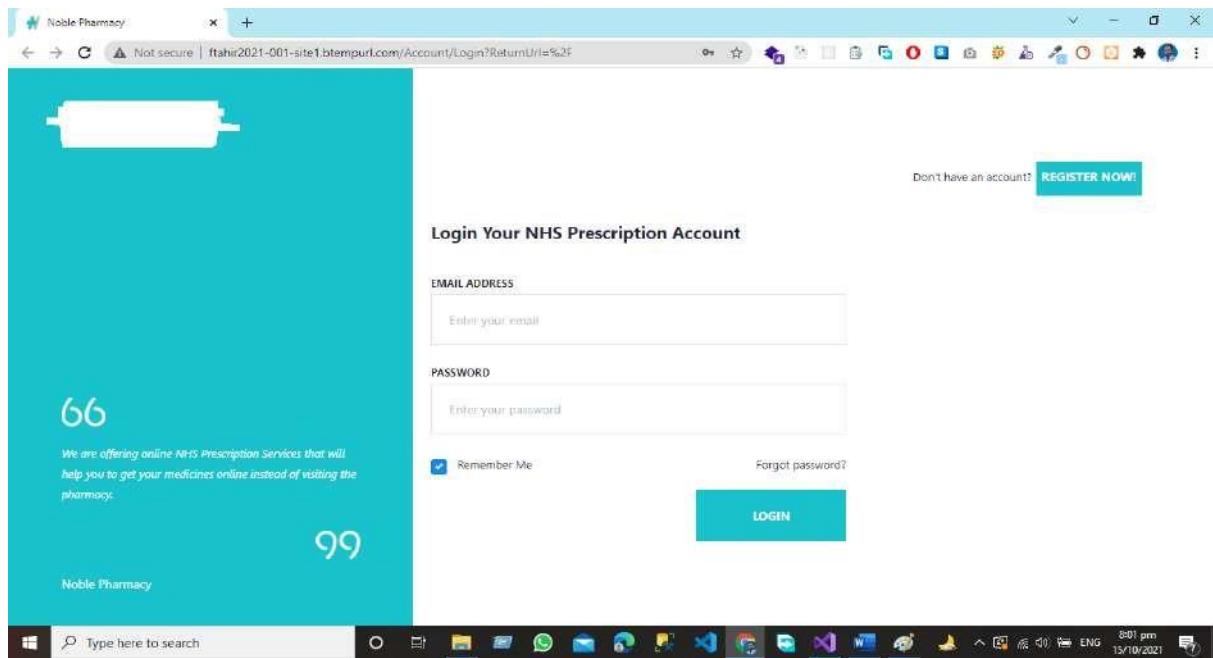
Publication of the website has been started.



Our Deployment of the website on the server has been successfully done.



Now we are live on the website.



Conclusion

We learned how to publish our asp.net Core web application using Visual Studio in this chapter, and in the next post, we'll learn how to deploy our application using FileZilla Software FTP.

“Happy Coding”

References

1. <https://www.c-sharpcorner.com/>
2. <https://docs.microsoft.com/en-us/>
3. <https://google.com/>
4. <https://www.castsoftware.com/>
5. <https://www.guru99.com/restful-web-services.html>

Feedback

If you want to submit your feedback regarding the book theme book content or any other suggestion for improvement, then scan the QR Code and submit your feedback.



Thank You!!!

References