**What are Microservices?**

Microservices are the architectural approach to build applications from small to large scale applications. With this architectural approach, an application is broken down into the smallest components, independent of each other. Unlike Monolithic architecture, where all the functionalities are targeted to build into a single project/application, Microservices helps to separate functionalities to develop in a more modular way and all modules work together to accomplish the specific targeted tasks.

**Need of a Microservice**

This architectural approach to developing software gives more modularity, being lightweight, and the ability to share similar functionalities across multiple applications. It is a major way of designing and optimizing app development towards a cloud-native model.

To develop quality software, the faster development process for change in existing software features, easy to tackle any runtime issues after deployment into production. As we have challenges in every positive idea, Microservices also have the challenges like managing them aligning them to work without downtimes, because we maintain them in different servers and have to deal with the data and databases they use for transactions.

**Advantages of Microservices**

Microservices give development teams and testers a faster approach through distributed development. Provides the ability to develop multiple Microservices simultaneously. This means more developers working on the same app but different functional modules, at the same time, this results in a less deliverable time of application to the client. Below are some critical points you can observe positively in Microservices.

**Faster Development**

Since the application is designed into smaller modules development time is shortened, a microservices architecture supports a more agile development process. Multiple teams can work on each module/feature to develop and deliver separately.

**High scalability**

As user demand for some services in application grows, you can deploy those services across multiple servers, and infrastructures, to meet your needs. As much as the usage of services increases by end-users, easily the deployment and extending of Microservice can be handled by load balancing.

**Resilient & Independency**

The independent services, after properly developed and orchestrated to not impact one another. This means that one service failed to work the whole app won’t go down, unlike the monolithic application model.

**Easy of Deployment**

Because the microservice-based applications are more modular than traditional monolithic applications, the problems that came with application deployments and application downtimes are reduced.

**Accessibility for Development**

Since the larger app is broken down into smaller modules, developers can easily understand, update, and enhance the new changes, which results in fast development cycles, if planned with agile development methodologies.

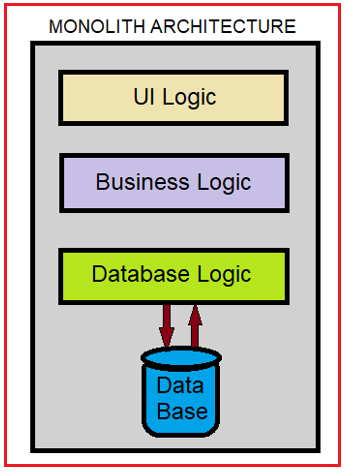
#### **Difference between Monolith Architecture and Microservice.**

##### ****Monolith Architecture****

First, we will understand the traditional way of building applications called Monolith Architecture, which is the widely used architectural pattern while developing applications. Still, we can see these architecture applications exist in the real world, it is designed in a way that the entire application built is ultimately a single piece, no matter how much you try to de-couple them by using Patterns and tiers – 1/2/3. All the services or business functionalities would be tightly coupled within the single Solution. The major thing to note is that while publishing the application, you would have to deploy them to a single server only.

While it is still an effective way to build applications, there are some drawbacks associated with Monolith Architecture. Any small to midscale Applications would do just fine with this Architecture, but when you scale up further to extend the capabilities of existing software, you would have to make a few compromises as well as face whole application downtimes while deploying new versions/bug fixes.

Here is a diagrammatic representation of the Monolith Architecture.



From the above diagram, you can observe different logics or layers you want to integrate with your application are consolidated into a part of the single application itself. Thus, it is a big problem to maintain and add a new feature and deployment of the application in this architecture.

##### ****Microservice Architecture****

As we mentioned using Microservice architecture the application is divided into various components or modules, with each module serving a particular purpose. And these components are called Microservices all to gather. These components are not dependent on the application itself. Each of these components is truly independent in all technical manners. Because of this robust separation, you can have separately dedicated Databases for each component i.e., Microservice as well as can deploy them to separate Hosts & Servers.

##### ****Microservice Vs Monolith Architecture****

|  |  |
| --- | --- |
| **Monolithic** | **Microservices** |
| Dependent & compounded project development of User Interface, Business functional Components, and Database Logics. | Multiple Services are developed with a single purpose each. |
| Single Database due to complex coupled development | Dedicated Database for each microservice due to modularity. |
| Single programming Language dependency for the Backend (Business logic) | Ability to choose multiple technologies/languages for each microservice. |
| The solution has to be deployed into a single Server. Physical Separation of deployment and maintenance is complex. | Each of the Microservice can be deployed anywhere on the web with ease. |
| Different business functionalities technically called modules or components will be tightly coupled to the application itself. | Loosely Coupled Architecture because of the modular approach. |

##### ****Microservices Architecture****

###### ****A brief explanation of Microservice Architecture****

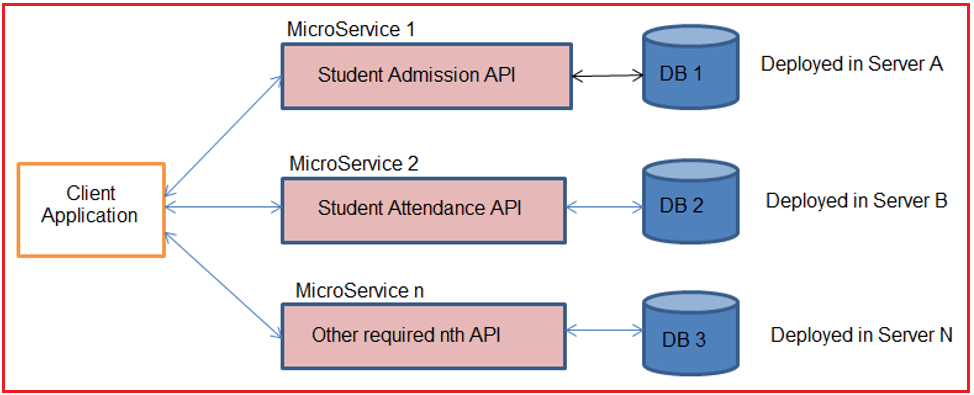
We will discuss this here with a small example. Imagine we are going to build a School Management Application in ASP.NET Core. Which have the functionalities to handle Student Admission and Attendance systems.

In Monolith Approach would be to Build a Single MVC project Solution on Visual Studio and then separate the functionalities via Layers / Tiers. Thus, you would probably have Projects like Data Access Layer, Business Function Layer, View, and so on.

Then these Layers/ Tiers are just at the level of code organization and are efficient only while developing. When you are done with the application, you will have to publish them to a single server where you can no longer see the separation concerns, A small change in any functionality is a heavy impact for the whole application, because for re-deployment to server whole application has to move to downtime and go through several unit tests to make sure that the new fix/enhancement did not break existing other functionalities.

But if you had followed a Microservice Architecture, you would have made Separate Components for Student Admission & Student Attendance, and so on. That means you have 2 different API Projects that are specific to what a Student Admission Endpoint needs and what a Student Attendance Endpoint needs. Now, these different APIs (Microservices) can be deployed anywhere in the Web world and don’t need to get deployed along with the Main Application. You can create a clean separation and physically decouple the Application into various parts.

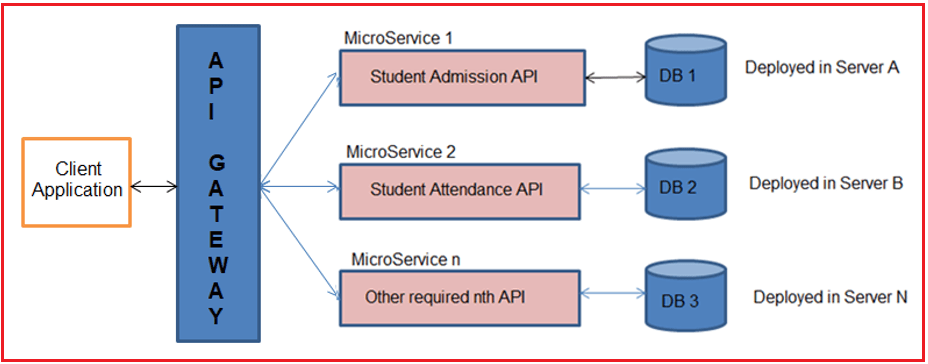
The below diagram illustrates a Microservice Architecture.



In the above picture, we can see the Client UI application either developed in .Net technology or Android or IOS or Angular, etc., will interact with our decoupled APIs called Microservices developed in .NET CORE technology, and they are deployed on multiple servers.

##### ****Understanding API Gateway (Ocelot Gateway with example)****

API Gateway is nothing but a middleware layer of directing incoming HTTP request calls from Client applications to specific Microservice without directly exposing the Microservice details to the Client and returning the responses generated from the respective Microservice.



Ocelot is an Open-Source API Gateway for the .NET/Core Platform which is officially supported by Microsoft. What it does is simple. It mimics masking multiple microservices existing behind that the client does not have to worry about the location of each and every Microservice.

Ocelot is widely used by Microsoft and other tech giants as well for Microservice Management.

##### ****Features of API Gateway****

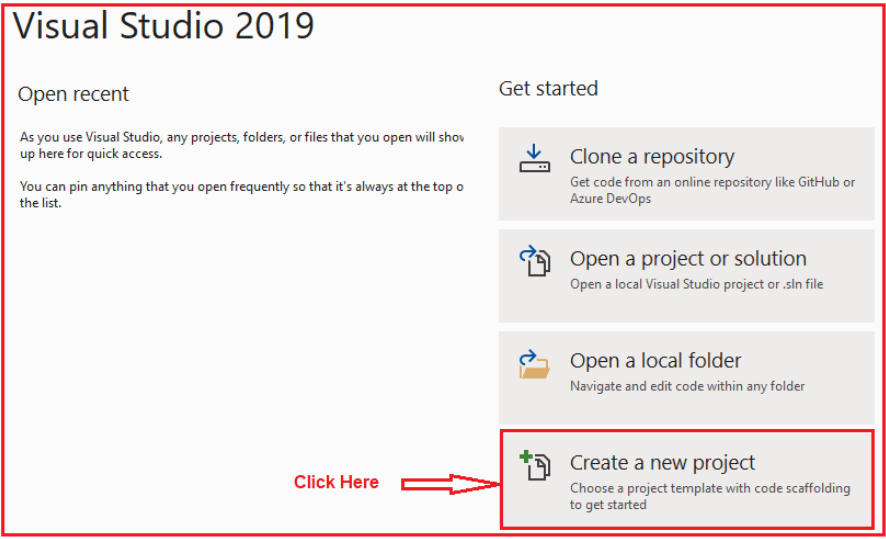
**API virtualization:**API Gateways acts as a single point entry for all the microservices configured, and avoids direct availability of microservices to clients, and hides versioning details of microservices.

**Serves as an additional layer of security microservices:**API gateways prevent malicious attacks by providing an additional layer of protection from attack vectors and hackers like SQL Injection, XML Parser exploits, and denial-of-service (DoS) attacks, and forged form data submissions.

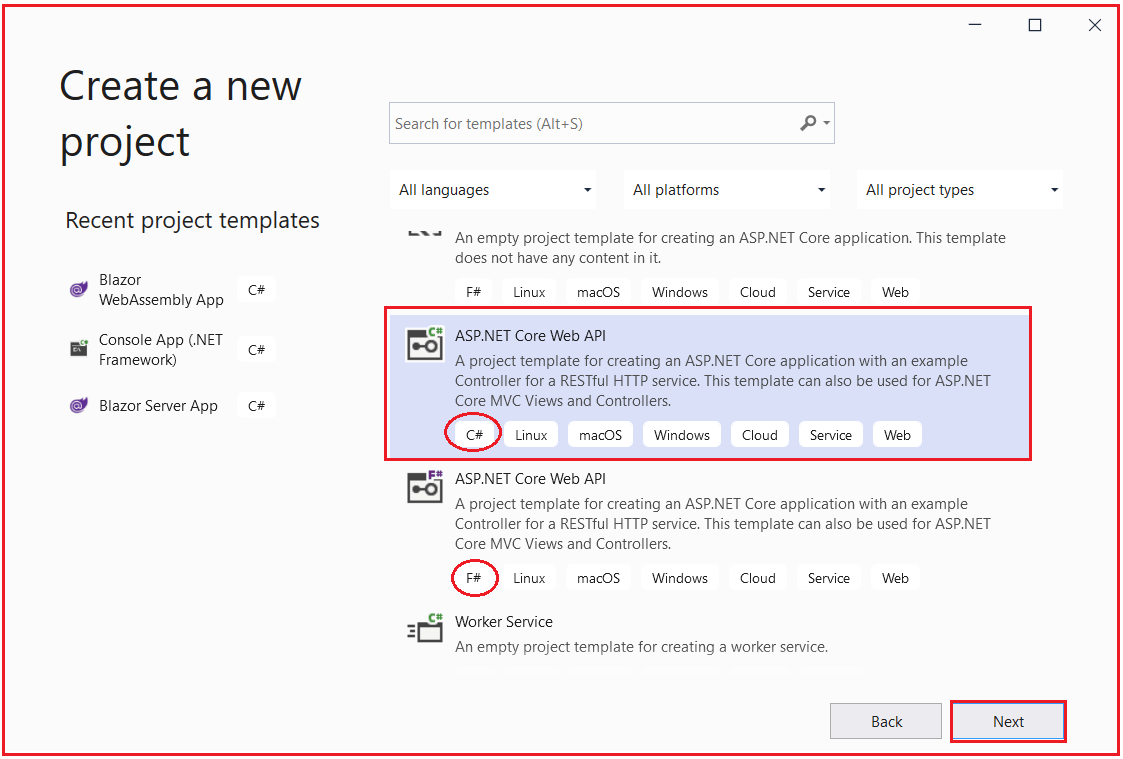
#### **Application Building Steps Wise**

##### ****Creating ASP.NET Core Web API Project Using Visual Studio 2019****

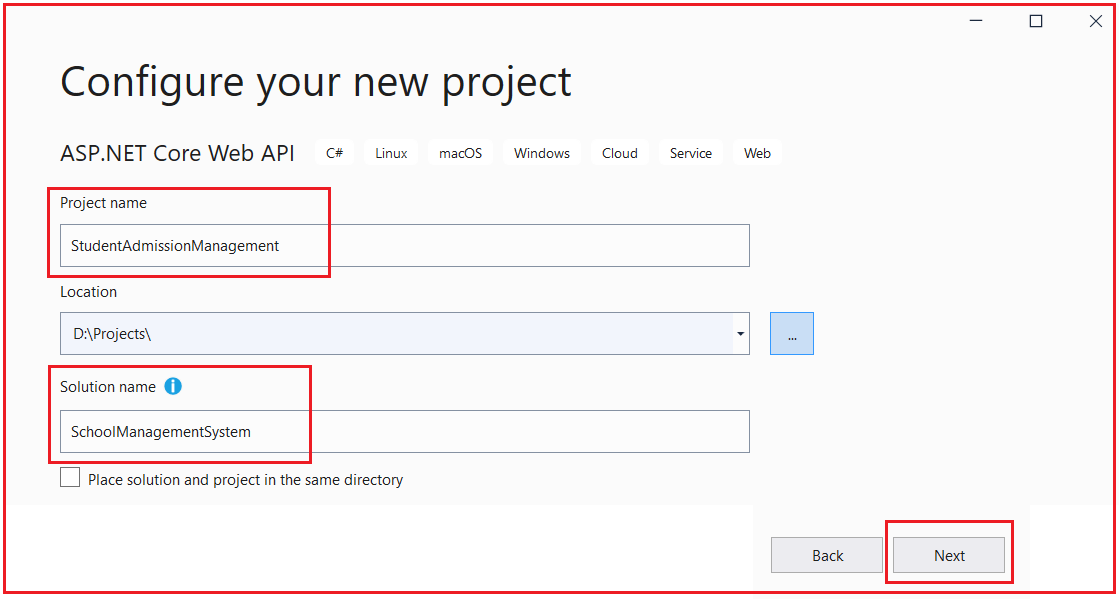
Now, we are going to create the ASP.NET Core Web API project using Visual Studio 2019. So, open Visual Studio 2019 and then click on the Create a new project option as shown in the below image.



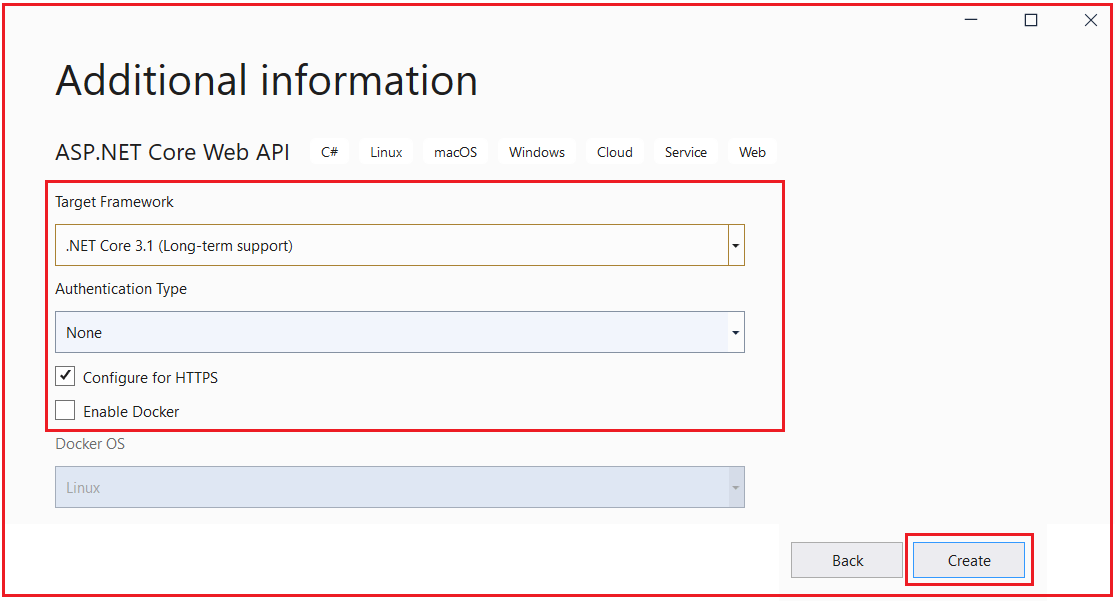
Once you click on the Create a new project option, the following Create a new project window will open. Here, you can find two projects template for creating the **ASP.NET Core Web API**project. One is using C# language and the other one is using F# language. I am going to use C# as the programming language, so I select the project template which uses C# Language as shown in the below image.



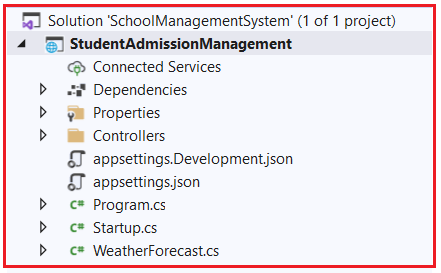
Once you click on the **Next** button, then the configure your new project window will open. Here, you need to specify the Project name Solution name, and the location where you want to create the project. I am providing the Project name as StudentAdmissionManagement, solution name as SchoolManagementSystem, and finally, click on the **Next**button as shown in the below image.



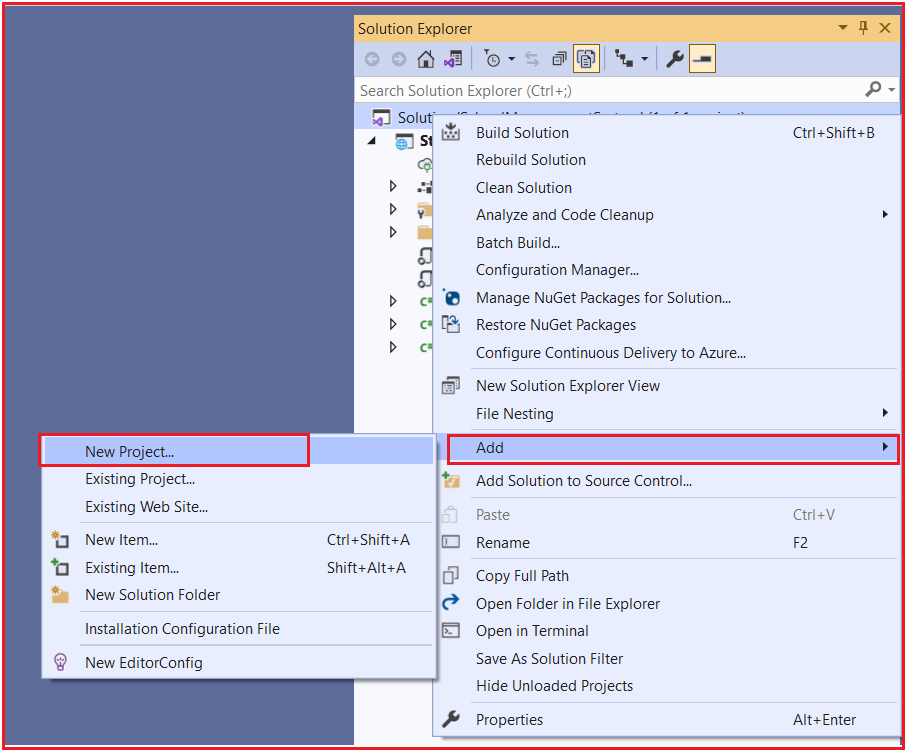
Once you click on the Next button, it will open the Additional Information window. Here, you need to select the Target .NET Framework version. The authentication Types. Whether you want to configure HTTPS and enable Docker. Select .NET Core 3.1 for now because it is having long Term support from Microsoft, select authentication type as None, check the Configure for HTTPS and uncheck the Enable Docker checkboxes and then click on the Create button as shown in the below image.



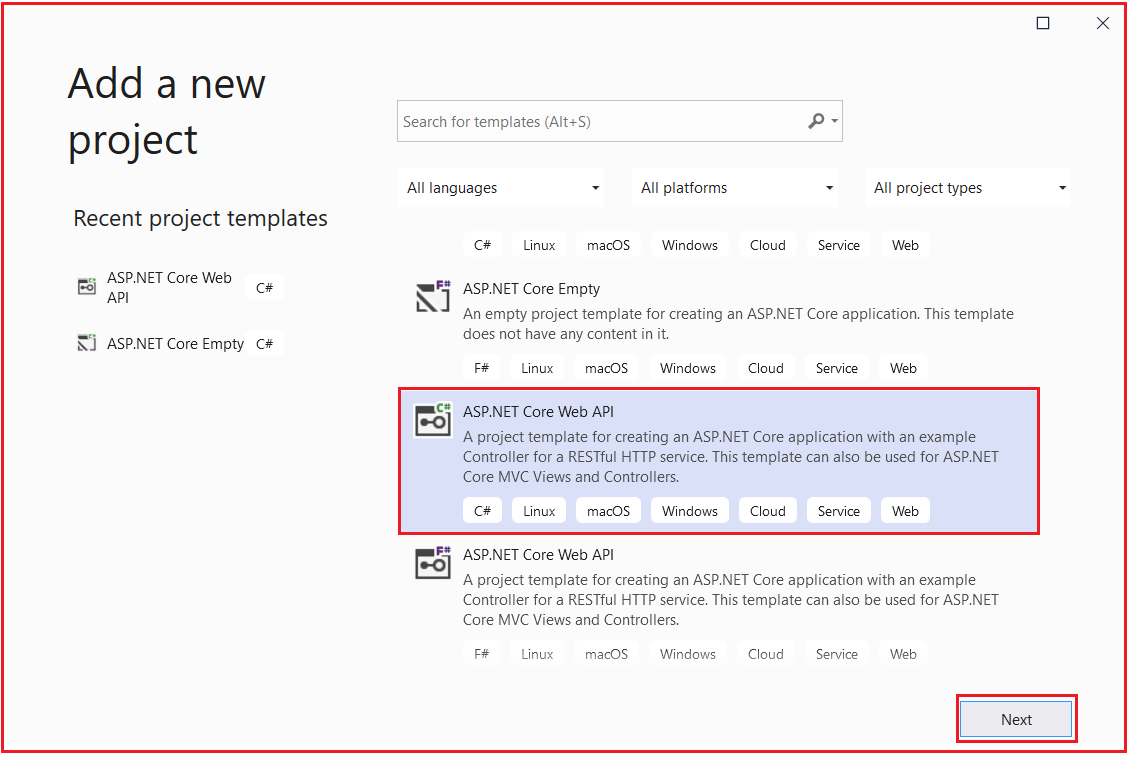
Once you click on the Create button, it will create the ASP.NET Core Web API project with the following file and folder structure. Initially, I have created only one separate project for student admission purposes which can be considered as a single microservice that works for student admission.



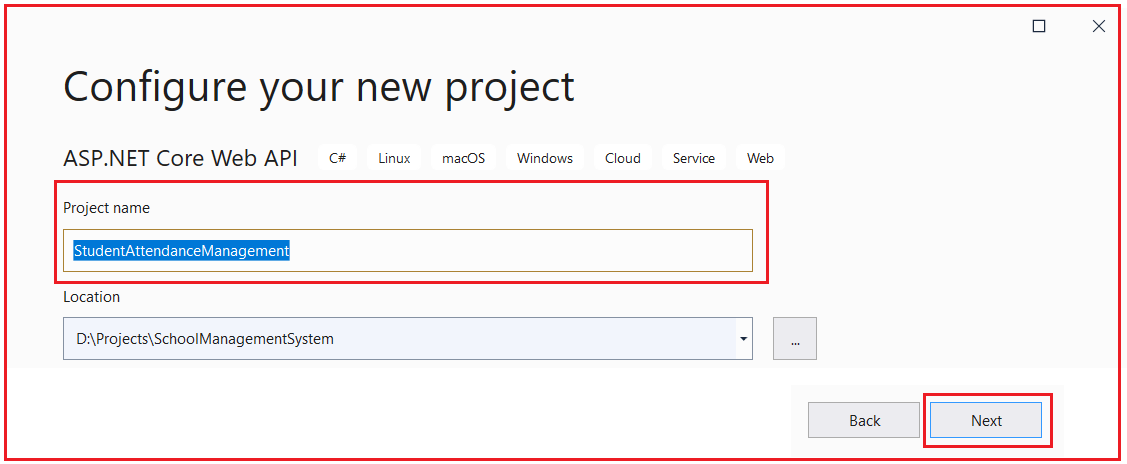
Let us create one more project under the solution which is another microservice for purpose of student attendance. To do so, right-click on the solution and then select Add => New Project option from the context menu as shown in the below image.



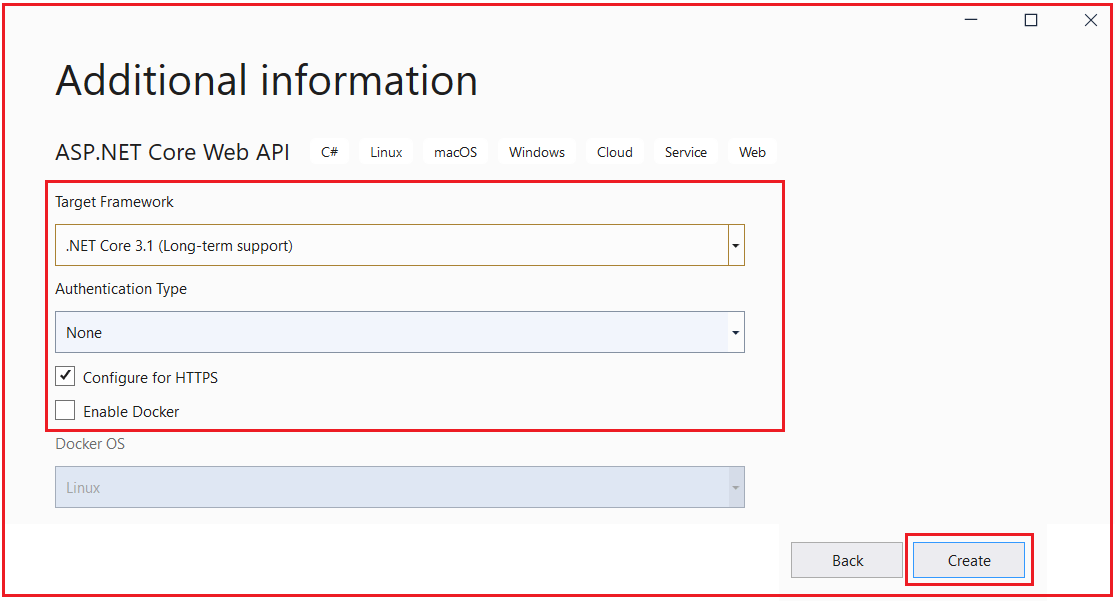
Once you click on the Add => New Project option, it will open the Add New Project window. From this window, select ASP.NET Core Web API (which uses C# language) and click on the Next button as shown in the below image.



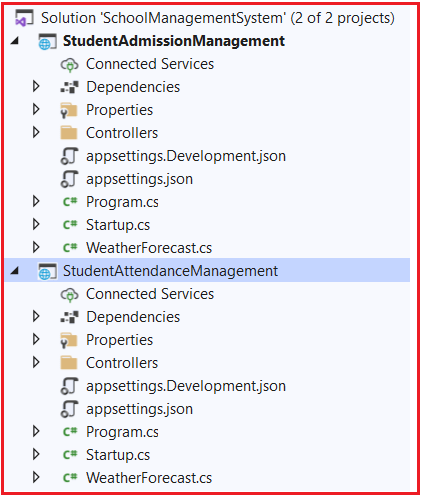
Once you click on the Next button, it will open Configure your new project window. From this window, provide the project name as StudentAttendanceManagement and click on the Next button as shown in the below image.



Once you click on the Next button, it will open the Additional Information window. Here, Select .NET Core 3.1 as Target Framework, select authentication type as None, check the Configure for HTTPS and uncheck the Enable Docker checkboxes and then click on the Create button as shown in the below image.



Once you click on the Create button, then it will add the new project to the existing solution. Now, our solution containing two projects with the following file and folder structure.



Now we have two microservices defined one for Student Admission purposes and the other for Student Attendance purposes.

##### ****Creating Models:****

Now let’s create a model class to hold details of Admission and Attendance in both the projects respectively and create a CRUD operation Controllers in each of project respectively under the Controllers folder. So, right-click on the StudentAdmissionManagement project and add a class file with the name StudentAdmissionDetailsModel.cs and then copy and paste the following code in it.

**using** *System;*

**namespace** *StudentAdmissionManagement*

**{**

**public** **class** StudentAdmissionDetailsModel

**{**

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

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

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

**public** DateTime DateofJoining **{** **get**; **set**; **}**

**}**

**}**

Now, right-click on the StudentAttendanceManagement project and add a class file with the name StudentAttendanceDetailsModel.cs and then copy and paste the following code in it.

**namespace** *StudentAttendanceManagement*

**{**

**public** **class** StudentAttendanceDetailsModel

**{**

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

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

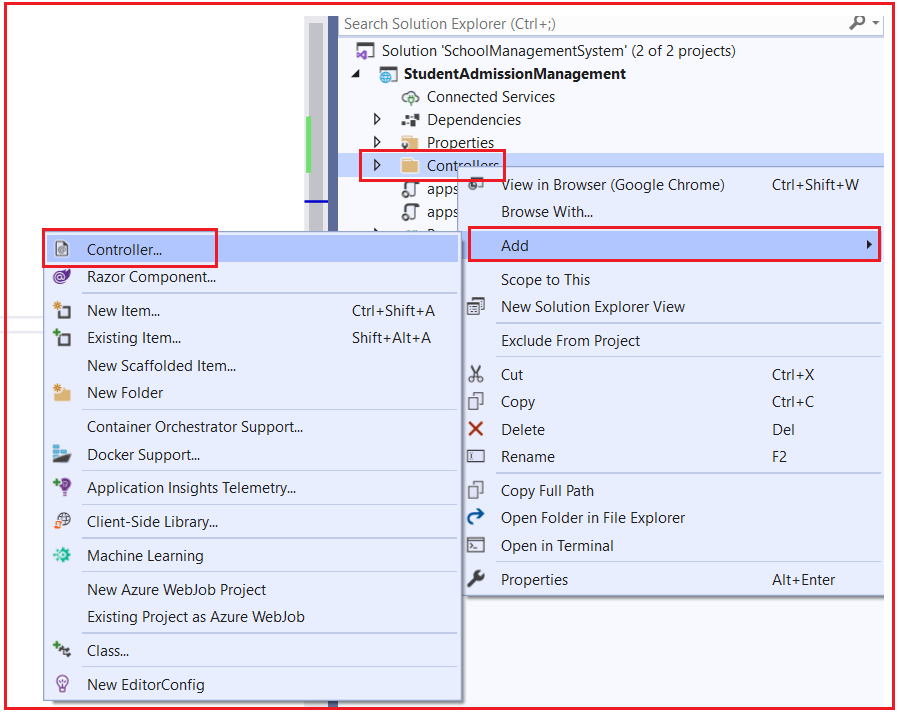
**public** **double** AttendencePercentage **{** **get**; **set**; **}**

**}**

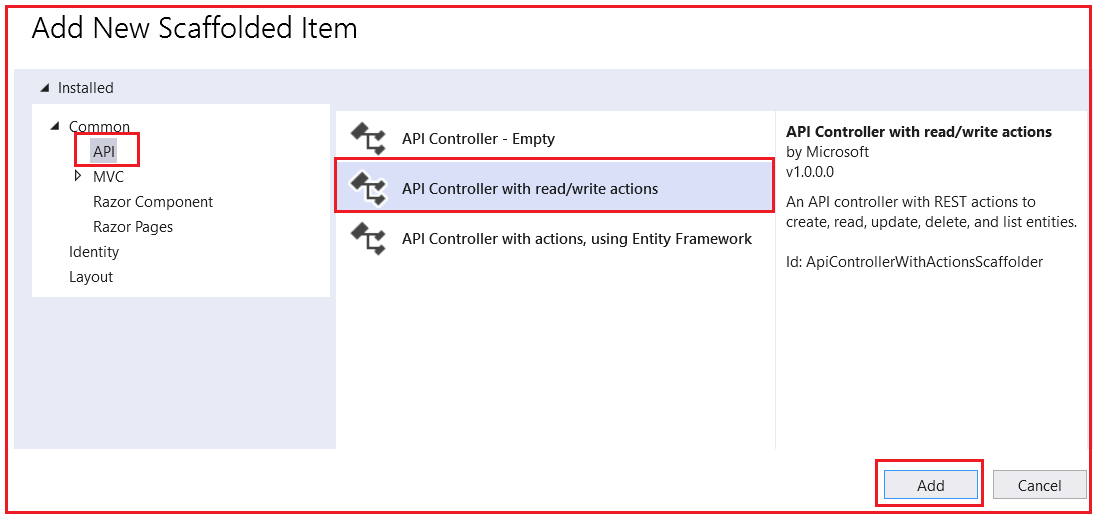
**}**

##### ****Creating Controllers:****

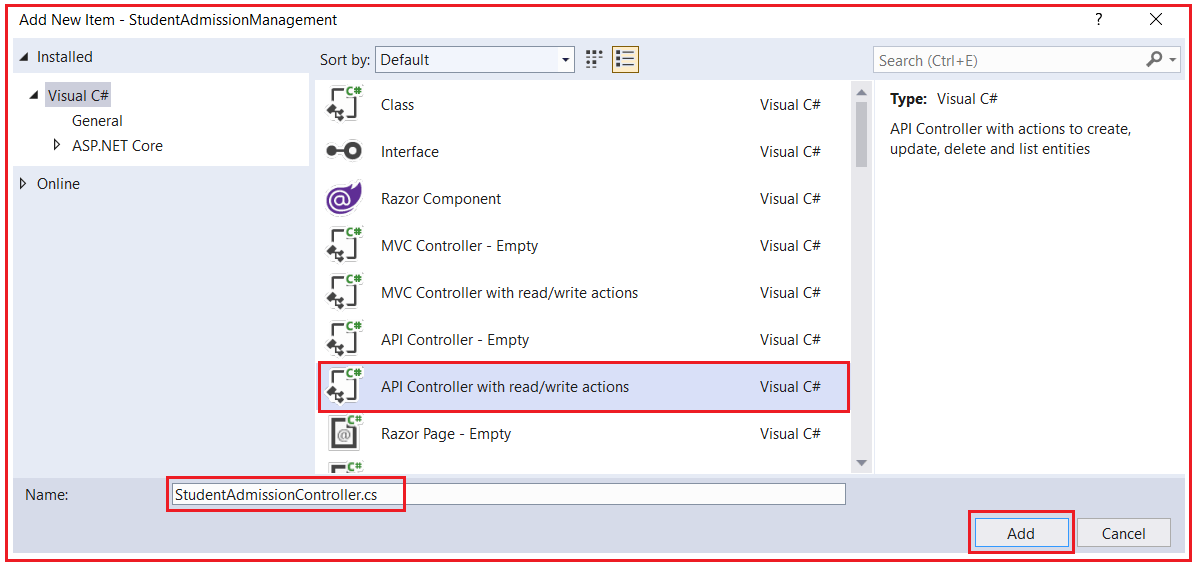
Now Let us create controllers which are endpoints for microservices for defining CRUD operations. Right-click on the Controllers folder of StudentAdmissionManagement and then select Add => Controller option from the context menu as shown in the below image.



Once you select the Add => Controller option, then it will open the Add New Scaffolded Item window. Here, first, select the API template and then select the Add Controller with read/write action template and click on the Add button as shown in the below image.



Once you click on the Add button, from the next window provide the name for your controller. Here, I am providing the name as StudentAdmissionController and click on the Add button as shown in the below image.



Once you click on the Add button, then it will add the StudentAdmissionController within the Controllers folder of your StudentAdmissionManagement project.

For the demonstration purpose, we create a GET endpoint with details of two students which will return details of two students to client request over HTTP GET. So, modify the StudentAdmissionController as follows.

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *StudentAdmissionManagement.Controllers*

**{**

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** StudentAdmissionController : ControllerBase

**{**

// GET: api/<StudentAdmissionController>

**[**HttpGet**]**

**public** IEnumerable**<**StudentAdmissionDetailsModel**>** Get**()**

**{**

StudentAdmissionDetailsModel admissionobj1 = new StudentAdmissionDetailsModel**()**;

StudentAdmissionDetailsModel admissionobj2 = new StudentAdmissionDetailsModel**()**;

admissionobj1.StudentID = 1;

admissionobj1.StudentName = "Adam";

admissionobj1.StudentClass = "X";

admissionobj1.DateofJoining = DateTime.Now;

admissionobj2.StudentID = 2;

admissionobj2.StudentName = "Brad";

admissionobj2.StudentClass = "IX";

admissionobj2.DateofJoining = DateTime.Now;

List**<**StudentAdmissionDetailsModel**>** listofobj = new List**<**StudentAdmissionDetailsModel**>**

**{**

admissionobj1,

admissionobj2

**}**;

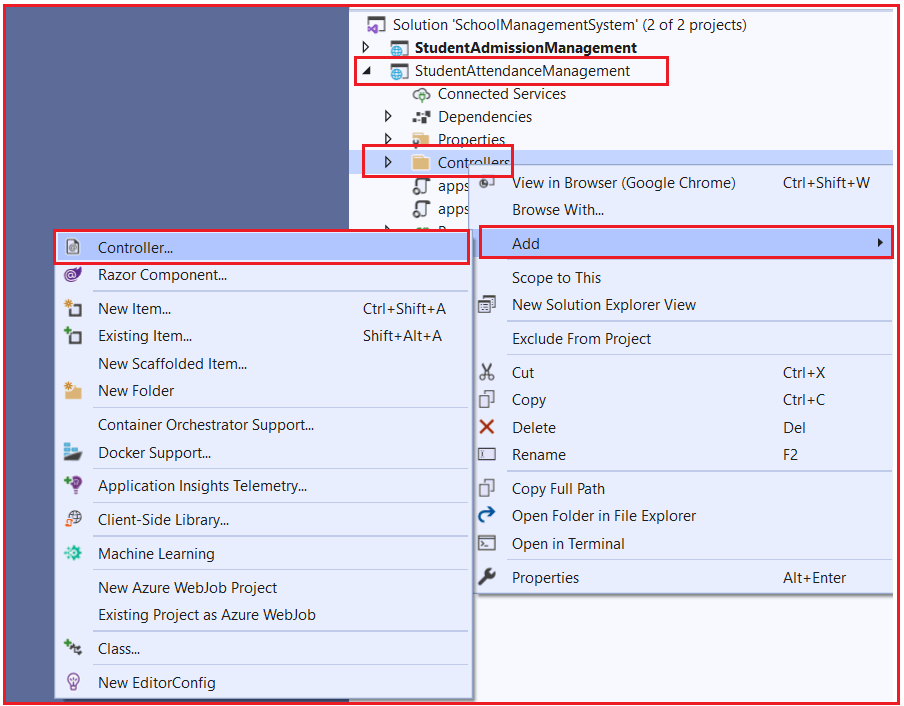
**return** listofobj;

**}**

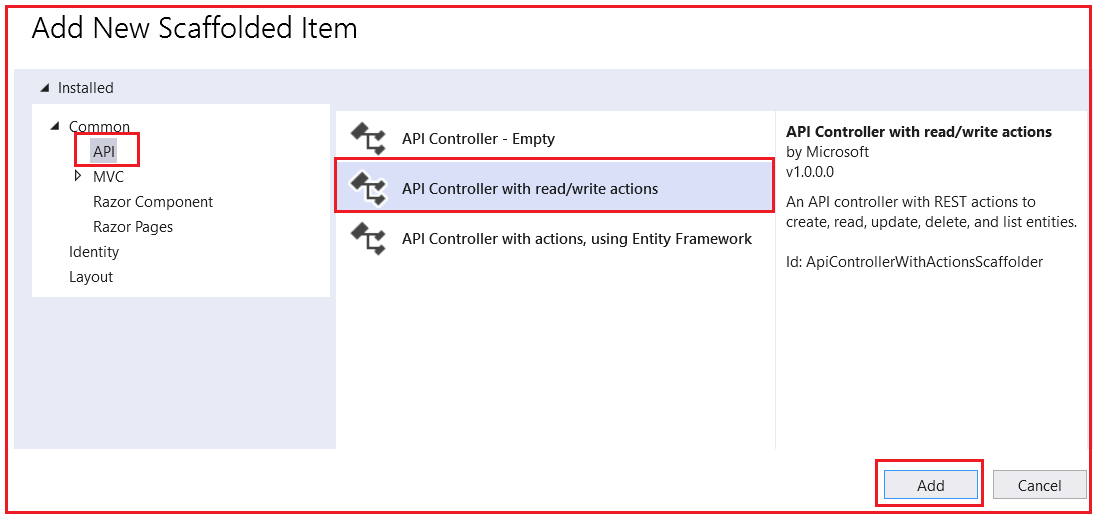
**}**

**}**

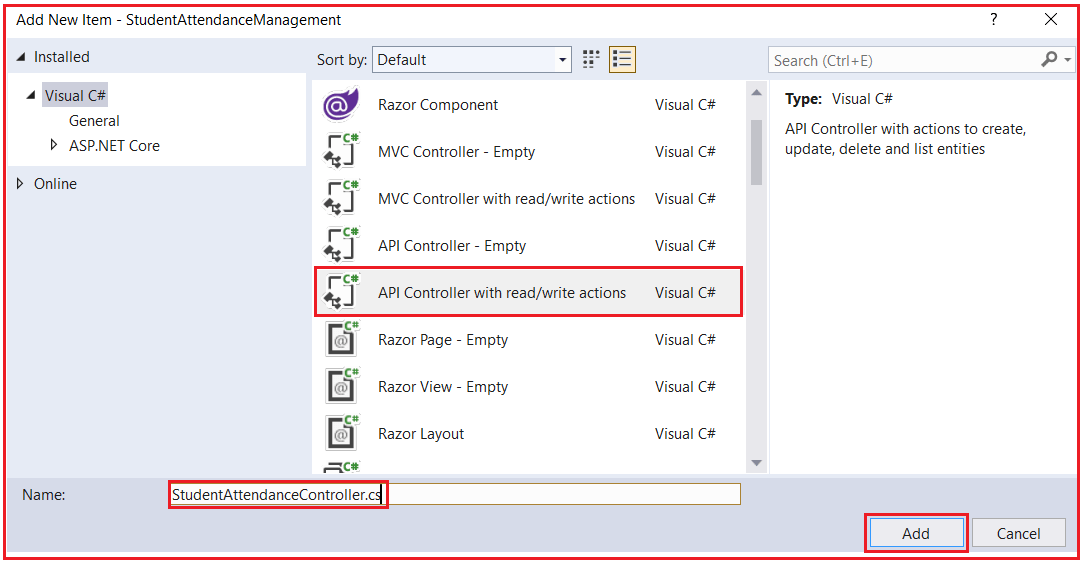
Now, Right-click on the Controllers folder of StudentAttendanceManagement and then select Add => Controller option from the context menu as shown in the below image.



Once you select the Add => Controller option, then it will open the Add New Scaffolded Item window. Here, first, select the API template and then select the Add Controller with read/write actions and click on the Add button as shown in the below image.



Once you click on the Add button, from the next window provide the name for your controller. Here, I am providing the name as StudentAttendanceController and click on the Add button as shown in the below image.



Once you click on the Add button, then it will add the StudentAttendanceController within the Controllers folder of your StudentAttendanceManagement project. For the demonstration purpose, we create a simple Get method for returning the two students’ attendance percentage on HTTP GET requests. So, modify the StudentAttendanceController as follows.

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *StudentAttendanceManagement.Controllers*

**{**

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** StudentAttendanceController : ControllerBase

**{**

// GET: api/<StudentAttendanceController>

**[**HttpGet**]**

**public** IEnumerable**<**StudentAttendanceDetailsModel**>** Get**()**

**{**

StudentAttendanceDetailsModel attendanceObj1 = new StudentAttendanceDetailsModel**()**;

StudentAttendanceDetailsModel attendanceObj2 = new StudentAttendanceDetailsModel**()**;

attendanceObj1.StudentID = 1;

attendanceObj1.StudentName = "Adam";

attendanceObj1.AttendencePercentage = 83.02;

attendanceObj2.StudentID = 2;

attendanceObj2.StudentName = "Brad";

attendanceObj2.AttendencePercentage = 71.02;

List**<**StudentAttendanceDetailsModel**>** listObj = new List**<**StudentAttendanceDetailsModel**>**

**{**

attendanceObj1,

attendanceObj2

**}**;

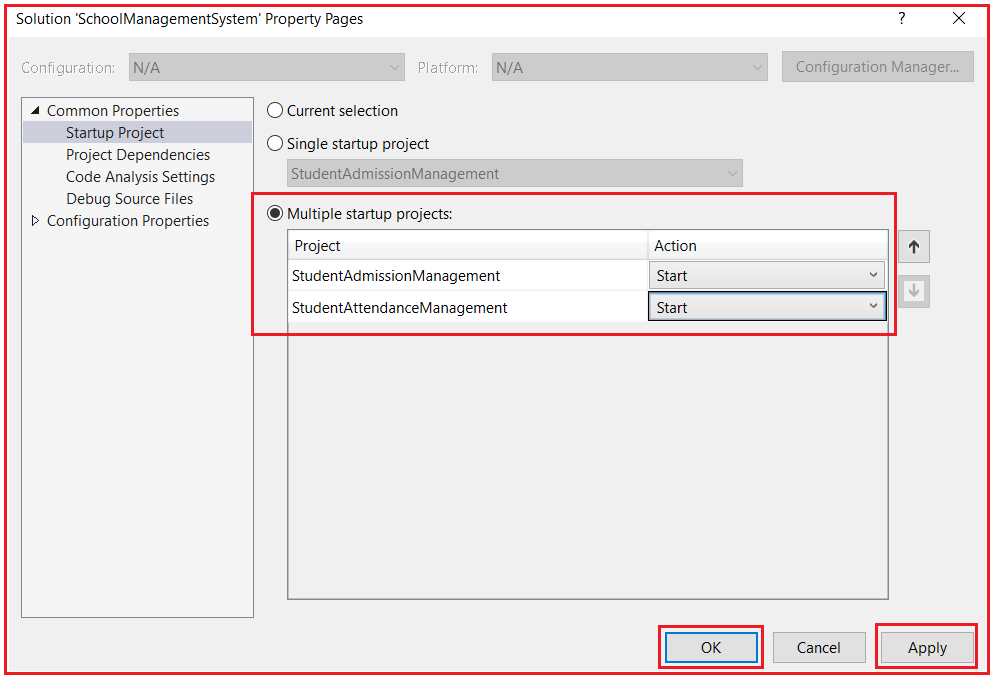
**return** listObj;

**}**

**}**

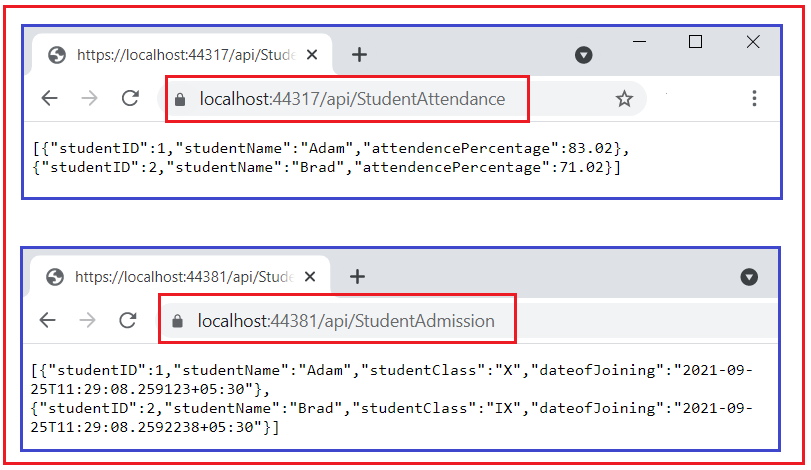
**}**

Let’s build the solution and launch both the Web API services (Microservices) simultaneously. For this, we have to set both the projects as starting projects. For that, open the properties window of Solution and then select multiple startup projects radio button and set the action to start for both the projects as shown in the below image.



##### ****Running Applications and Showing Outputs****

Click on the Start button and then we get two browsers opened and point to the controllers from browsers like below. Browser by default sends GET HTTP Verb.

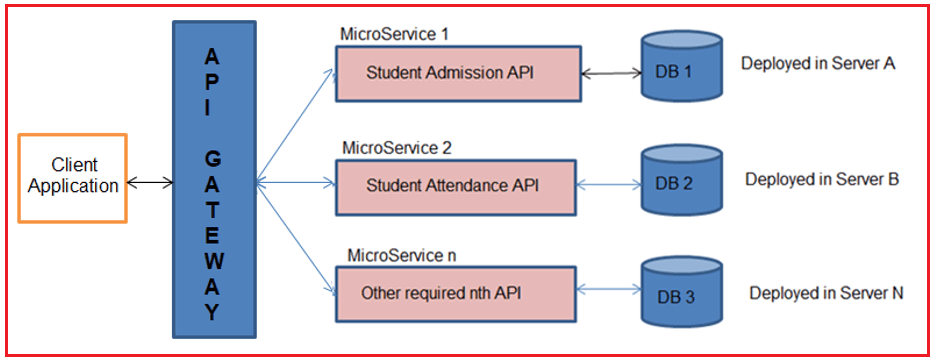


They returned the details as expected. So, this is how microservices are developed and our two microservices are available to serve the user requests over HTTP request and response. To test Microservices with Postman application please refer to the Testing section provided at end of this learning path.

Till now we have observed the implementation of independent APIs (Microservices) which hold separate concerns of business functionalities, one is to get admission details of all the students and the other is to get the attendance percentage of all the students in examples. Which are directly exposed to the Client.

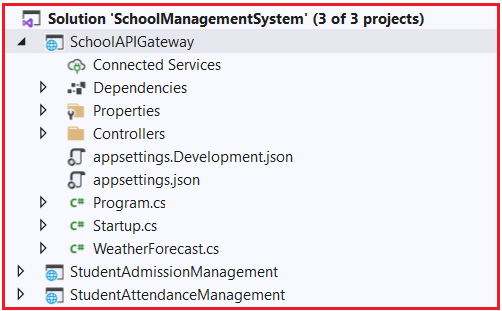
##### ****Creating an API Gateway in ASP.NET Core****

In this section let’s implement an API gateway, which will be responsible to re-direct the incoming request from the client (Browser / Postman App / any Client Application) to the Microservice and returns the response to the client. Observe the following diagram, depicting role of API Gateway,



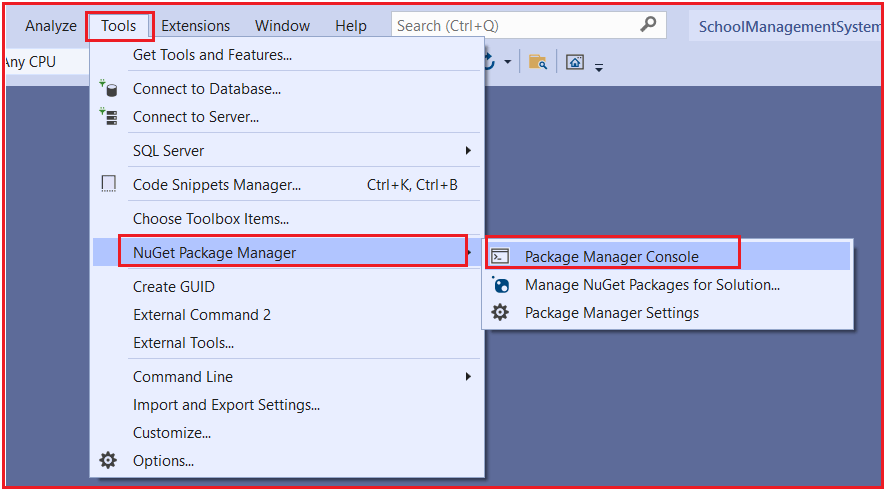
##### ****Creating API Gateway Project using Ocelot****

Let’s create a separate ASP.NET Core Web API project with the name SchoolAPIGateway like how we created projects in the solution earlier. Once the project is created it would look like below in the Solution Explorer window.

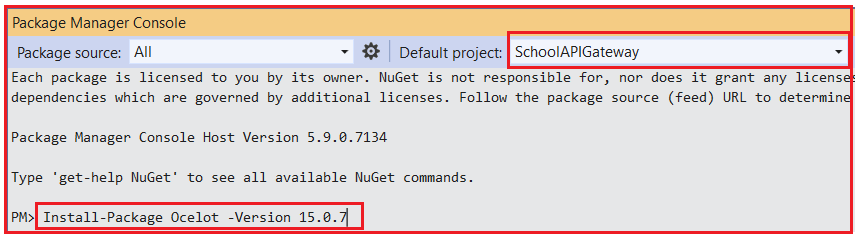


##### ****Installing Ocelot package****

Let’s install the Ocelot Package. This makes the project behave as an API gateway. To do so, open click on Tools => NuGet Package Manager => Package Manager Console option from the context menu as shown in the below image.



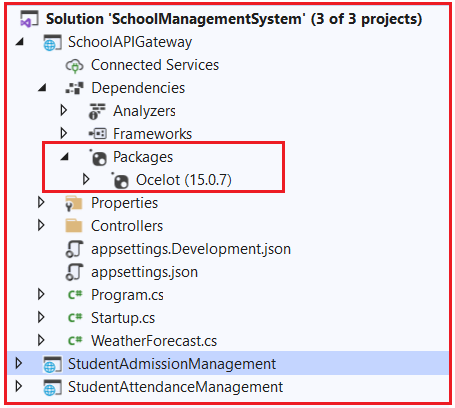
Once you open the Package Manager Console window, choose SchoolAPIGateway as the default project and then type Install-Package Ocelot -Version 15.0.7 and press enter as shown in the below image.



If you face installation issues, due to a mismatch of .NET versions, you can try the different versions of Ocelot, you will find the versions on the official NuGet website.

[**https://www.nuget.org/packages/Ocelot/**](https://www.nuget.org/packages/Ocelot/)

After successful installation, you can see the Ocelot package in the packages folder as shown in the below image.



##### ****Configure API Gateway and Integrate Microservices.****

let’s configure the project and define the Microservices endpoints.

##### ****Configure Ocelot as a middleware****

Let’s configure the Ocelot to work with our application. Go to the Program.cs class file of the SchoolAPIGateway Project and change the CreateHostBuilder() method as follows.

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.Configuration;*

**using** *Microsoft.Extensions.Hosting;*

**using** *Microsoft.Extensions.Logging;*

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *SchoolAPIGateway*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

CreateHostBuilder**(**args**)**.Build**()**.Run**()**;

**}**

**public** **static** IHostBuilder CreateHostBuilder**(string[]** args**)** =**>**

Host.CreateDefaultBuilder**(**args**)**

.ConfigureWebHostDefaults**(**webBuilder =**>**

**{**

webBuilder.UseStartup**<**Startup**>()**;

**})**.ConfigureAppConfiguration**((**HostingContext, config**)** =**>** **{**

config.SetBasePath**(**HostingContext.HostingEnvironment.ContentRootPath**)**

.AddJsonFile**(**"Ocelot.json", optional: **false**, reloadOnChange: **true)**;

**})**;

**}**

**}**

In the above code, we are configuring the app to read the Ocelot.json file from the root directory of the project which we obtained from HostingContext class. The Ocelot reads the route configuration from the JSON config file, through that the ASP.NET Core 3.1 Application is able to access the settings. Remember that we yet to have created the Ocelot.json file. We will be creating it after we have configured the Ocelot Middleware.

In the project SchoolAPIGateway open the Startup.cs class file and add Ocelot to the ConfigureServices method by adding services.AddOcelot() statement. And in the Configure( ) method adds the Ocelot Middleware to the ASP.NET Core Application Pipeline as follows.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.AspNetCore.HttpsPolicy;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.Extensions.Configuration;*

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**using** *Microsoft.Extensions.Logging;*

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**using** *Ocelot.DependencyInjection;*

**using** *Ocelot.Middleware;*

**namespace** *SchoolAPIGateway*

**{**

**public** **class** Startup

**{**

**public** Startup**(**IConfiguration configuration**)**

**{**

Configuration = configuration;

**}**

**public** IConfiguration Configuration **{** **get**; **}**

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

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

**{**

services.AddControllers**()**;

services.AddOcelot**()**;

**}**

// 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.UseHttpsRedirection**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapControllers**()**;

**})**;

app.UseOcelot**()**.Wait**()**;

**}**

**}**

**}**

##### ****Configuring Ocelot Routes in ASP.NET Core****

This is the most important part of configuring Gateway to reroute the incoming requests to a definite API service. In the Ocelot.json file, you have to configure the Upstream / Downstream routes for the API Gateways, which makes Ocelot understand the routes to redirect the requests to microservices. Let’s understand the following terms

1. **DownstreamPathTemplate** is the path of the actual endpoint in the Microservice to which the API Gateway redirects the incoming request from the client.
2. **DownstreamScheme** is the scheme used by Microservice, which is HTTPS.
3. **DownstreamHostAndPorts** defines the location of the Microservice. Where we will add the host details and port number details.
4. **UpstreamPathTemplate** is the path that is directed to Ocelot API Gateway exposed to send requests from the client.
5. **UpstreamHttpMethod** is the supported HTTP Method by the API Gateway such as GET/PUT/POST/DELETE. Based on the Incoming HTTP Method, Ocelot forwards a similar HTTP method request to the microservice as well.

Upstream and Downstream are the two terms that you have to be clear with. Upstream Request is nothing but the Request sent by the Client to the API Gateway and Downstream request is the request sent to the specific Microservice by the API Gateway. All these are visualized from the API Gateway perspective.

Let’s add a basic route setting so that you can understand how it works. We will start with Student Admission and Student Attendance Microservices which we built earlier. Let’s say the client wants to get all the student’s admission and student attendance details via the API Gateway this time.

We have the URL for Student Admission Microservice which we built earlier and tested in Browser

**https://localhost:44381/api/StudentAdmission ->** Downstream URL

**https://localhost:44317/api/StudentAttendance ->** Downstream URL

Create a new Ocelot.json file in the root directory of the SchoolAPIGateway Project. This file carries the routing configurations needed for Ocelot. To do so add a JSON file with the name Ocelot.json into the root directory of the SchoolAPIGateway Project and then copy and paste the following code into it.

**{**

"ReRoutes": **[**

**{**

"DownstreamPathTemplate": "/api/StudentAttendance"**,**

"DownstreamScheme": "https"**,**

"DownstreamHostAndPorts": **[**

**{**

"Host": "localhost"**,**

"Port": 44317

**}**

**],**

"UpstreamPathTemplate": "/apigateway/AttendanceService"**,**

"UpstreamHttpMethod": **[** "GET"**,** "PUT"**,** "POST" **]**

**},**

**{**

"DownstreamPathTemplate": "/api/StudentAdmission"**,**

"DownstreamScheme": "https"**,**

"DownstreamHostAndPorts": **[**

**{**

"Host": "localhost"**,**

"Port": 44381

**}**

**],**

"UpstreamPathTemplate": "/apigateway/AdmissionService"**,**

"UpstreamHttpMethod": **[** "GET"**,** "PUT"**,** "POST"**]**

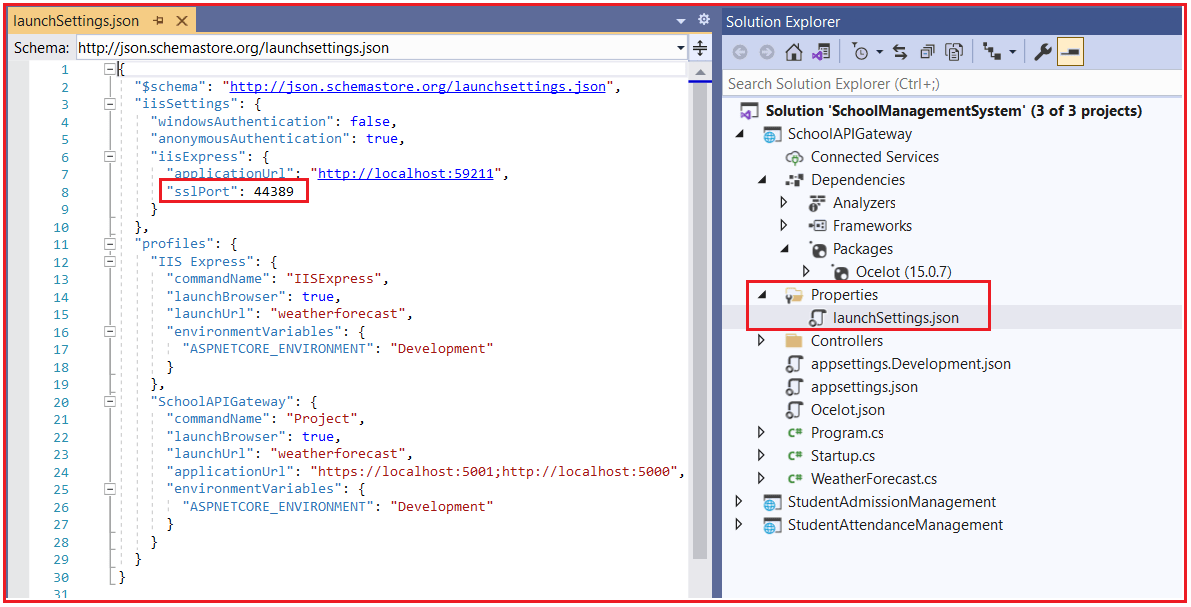
**}**

**]**

**}**

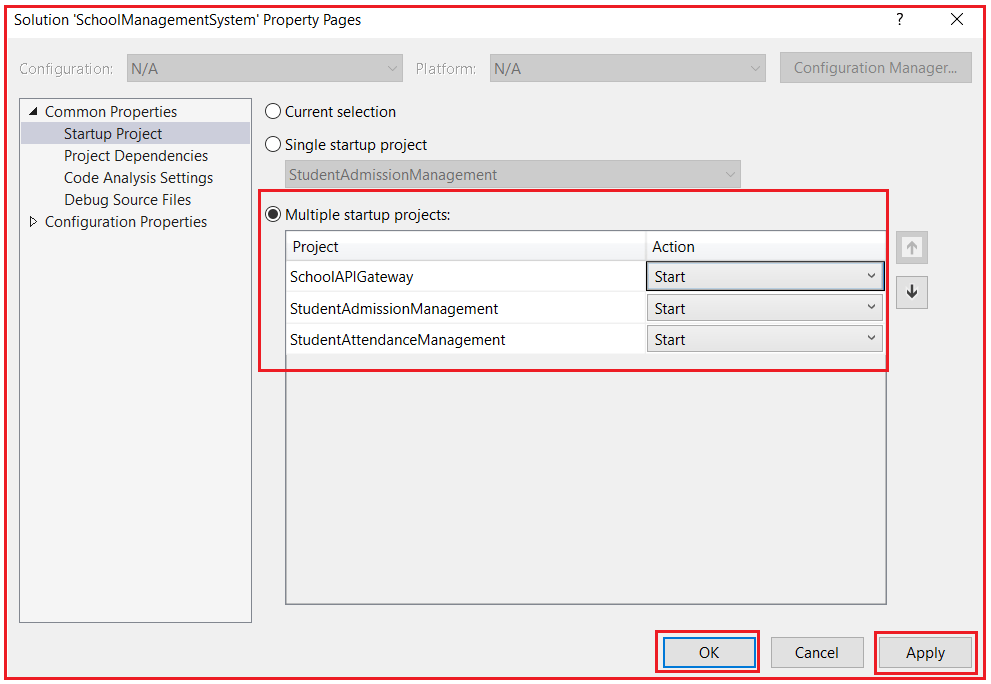
Ocelot reads the JSON file and takes Array of Route Objects. As the first element in the array, let’s configure the Student Attendance Microservice with access to GET, PUT and POST HTTP Verbs. Here in the example project we are currently using only GET HTTP Verb. And another Microservice Student Admission is also added similar to Student Attendance.

Let’s test it now. As per our configuration, we have to request the SchoolAPIGateway project defined Gateway API endpoints in UpstreamPathTemplate that is located at localhost:44389 which you can find in launchSettings.json file in the same project i.e. SchoolAPIGateway as shown in the below image.

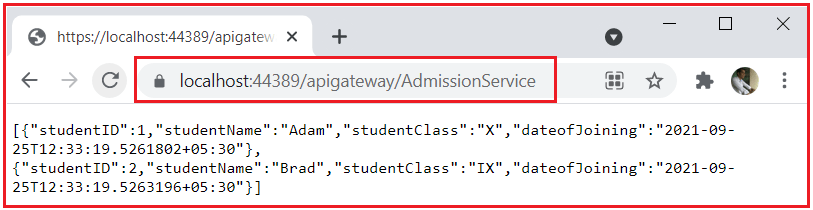


Build the entire solution. Now, there is one thing to change. We have 3 APIs now. Let’s set up the solution so that all three APIs get launched when you run the application. Because we will need all the APIs actively working to test the functionality.

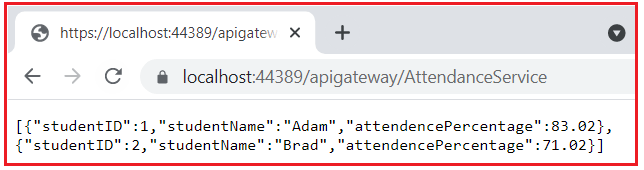
To launch Multiple Startup Projects, Go to the Solution Properties window. Select the Multiple Startup Projects radio button and set the Start option in the Action dropdown for each project and then click on the Apply and OK button as shown in the below image.



At the route **https://localhost:44389/apigateway/AdmissionService**, we are expecting to get the results directly from the microservice which is located at localhost:44381. Now, navigate to **apigateway/AdmissionService** and you will get the following result as expected.



And also, for /apigateway/AttendanceService in API Gateway you will get results from Student Attendance Microservice as expected as shown in the below image.



We have successfully developed API Gateways and created a Microservice Architecture in ASP.NET Core.

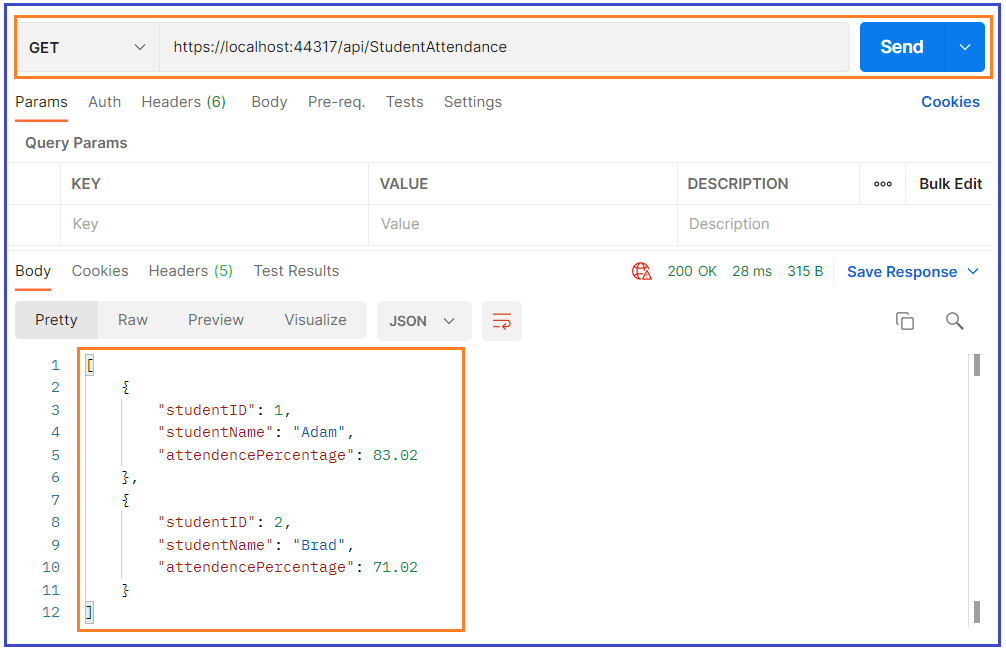
##### ****Testing Microservices using POSTMAN application****

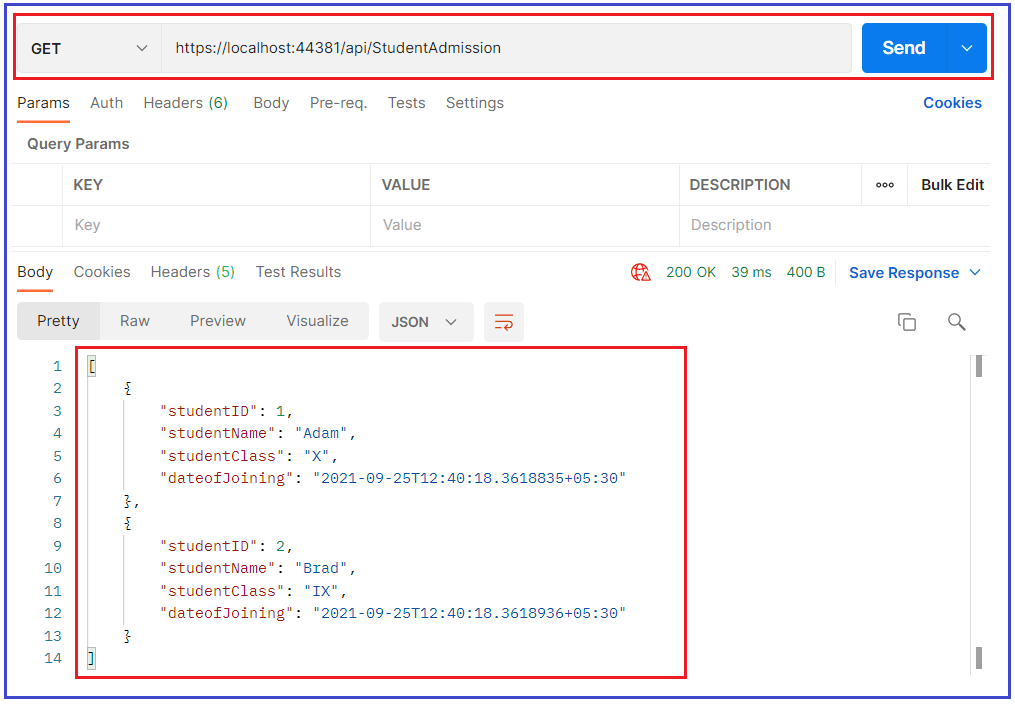
Postman is a web API testing tool that is easy to understand and effective in testing all kinds of HTTP Verbs like POST, GET, PUT, DELETE and etc., along with multiple kinds of request headers, Using Postman we can submit HTTP requests on behalf of actual client application where we are using this microservices / Web APIs. This is the official link where you can download the Postman application.

[**https://www.postman.com/downloads/**](https://www.postman.com/downloads/)

##### ****Testing Microservices****

Make sure that the Student Attendance and Student Admission microservices are running in visual studio and are available to serve the client HTTP requests before requesting from the postman to get the expected outputs. I have given the Student Attendance microservice URL directly into the Postman and clicked the send button. We got the expected output with status 200 OK which means the client request is successfully resolved.

 And now testing the other microservice Student Admission, I have given the URL with GET request and we received the expected result with status 200 OK.

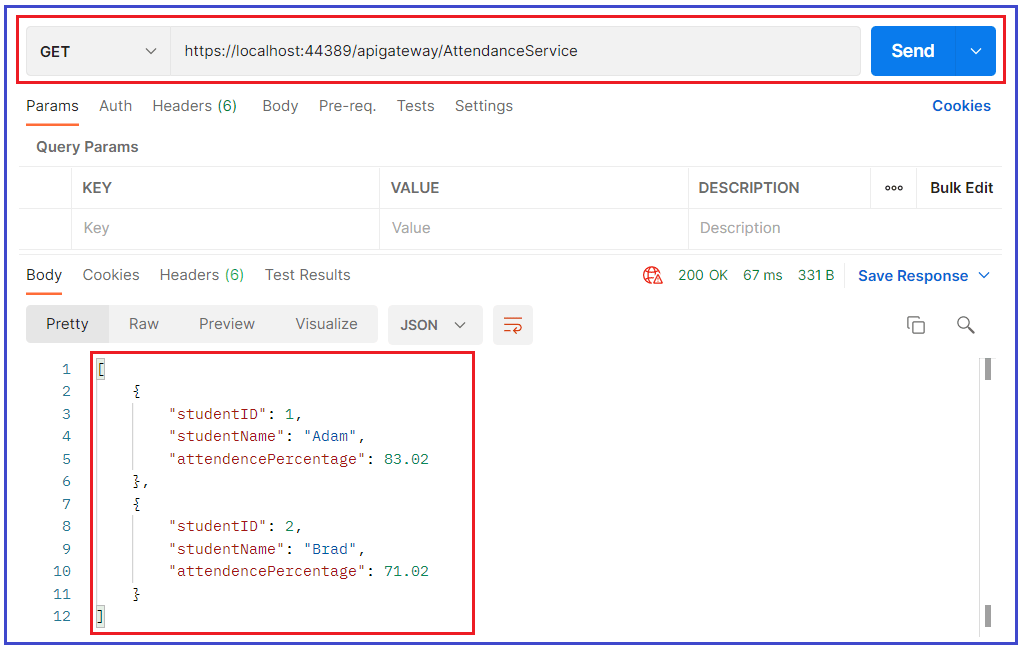


##### ****Testing the Microservice through API Gateway****

Now we will try to test the API gateway URLs which we defined for calling Student Attendance microservice and Student Admission Microservice.

Make sure that all three projects are running, SchoolAPIGateway project, Student Admission Management, and Student Attendance Management, so that the First request will be sent to SchoolAPIGateway it will redirect to the respective microservice and gets a response.

I have given the API gateway URL defined to get Student Attendance details in Postman with GET and sent the request. We received the expected output.



Similarly, I have initiated the request to API gateway URL defined to get Student Admission details in Postman with GET and sent request, We received expected output.

