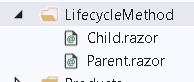
**Lifecycle Methods**

1. SetParameterAsync

* First method to invoked
* Called before any parameter value is assigned to their respective properties.
* Also called each time new or updated parameters are received
* Useful for Parameter validation

Demo :

1. Add New Component Parent and child



1. Add Code to child component

<h3>Child</h3>

<h4> SetParameter Async</h4>

<p> @FirstNameFromParent</p>

@code {

[Parameter]

public string FirstNameFromParent { get; set; }

public override Task SetParametersAsync(ParameterView parameters)

{

foreach (var item in parameters)

{

Console.WriteLine(item.Name + " " + item.Value);

}

return base.SetParametersAsync(parameters);

}

}

1. Pass Data from Parent Component

@page "/parent"

<h3>Parent</h3>

<h4> SetParameterAsync : </h4>

<label> FirstName</label>

<input **@bind**="FirstName" />

<**Child** **FirstNameFromParent**="@FirstName">

</**Child**>

@code {

string FirstName = "Sachin";

}

1. OnInit or OnInitAsync

* This methods execute when the component is ready to start
* In this , initial parameters are available in their respective properties.
* Invoked once after OnInit is finished. Any operations that require the component to rerender should be placed here.

Demo : OnInit and OnInitAsync

1. Add Title variale in child component and methods

// Initialization

string Title = String.Empty;

// OnInit

protected override void OnInitialized()

{

Title = "Title is Empty";

base.OnInitialized();

}

protected override async Task OnInitializedAsync()

{

await Task.Delay(3000);

Title = "Update Title";

}

1. Check the Output .. Data will update after 3 sec
2. OnParameterSet and OnParameterSetAsync

* By the time this methods are called , all parameters have been assigned to their respective properties.
* They both get executed for the first time on initialiation and each time parameter values updated by the parent.
* Invoked After SetParametersAsync and OnInit are finished and then each time new or updated parameters are received.

1. Go to child component and add following code

// OnParameter Set

[Parameter]

public int RandNumber { get; set; } = 0;

int count = 0;

int randNumWithDelay = 0;

protected override void OnParametersSet()

{

count++;

}

protected override async Task OnParametersSetAsync()

{

await Task.Delay(2000);

randNumWithDelay = RandNumber;

//return base.OnParametersSetAsync();

}

1. Display data using table in child component

<h4> On Parameter Set</h4>

<table class="table">

<tr>

<th> Parameter Set call</th>

<th>Random No</th>

<th> RandomNo with Delay</th>

</tr>

<tr>

<td>@count</td>

<td>@RandNumber</td>

<td>@randNumWithDelay</td>

</tr>

</table>

1. Generate Random number and pass it to Child from Parent . Make the following code changes in Parent.

<**Child**

**FirstNameFromParent**="@FirstName"

**RandNumber**="@randomNumber">

</**Child**>

<button **@onclick**="RandomNumberGen" class="btn btn-primary">

Generate Random Number</button>

@code {

string FirstName = "Sachin";

// OnParameter Set

int randomNumber = 0;

Random random = new Random();

public void RandomNumberGen()

{

randomNumber = random.Next(100);

}

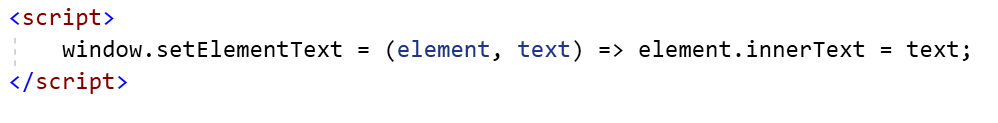
}

1. Check the Output:

OnAfterRender and OnAfterRenderAsync

* These Methods are essential when you need to perform additional initialization steps using rendered content . such as initializaing JS libraries.
* Use this stage to perform additional initialization steps using the rendered content, such as activating third-party JavaScript libraries that operate on the rendered DOM elements.

1. Add JS code in \_Host.cshtml



1. Add New Component and add a code to call this JS library function

@page "/micro-modal"

@using Microsoft.JSInterop

@inject IJSRuntime JSRuntime

<h2>Micro Modal</h2>

<div **@ref**="divElement"> Text During Render</div>

@code {

int count = 0;

private ElementReference divElement;

protected override void OnAfterRender(bool firstRender)

{

count++;

Console.WriteLine($"OnAfterRender execution count: {count}");

}

protected override async Task OnAfterRenderAsync(bool firstRender)

{

await Task.Delay(3000);

await JSRuntime.InvokeVoidAsync(

"setElementText", divElement, "Text after render");

}

}

ShouldRender

* Method returns a Boolean value
* If the value is true , the UI is re-rendered
* Its used to suppress subsequent rendering of the component.

For Example.

1. We have simple counter component is used to trigger UI refresh when the value count changes
2. When the ShouldRender method gets invoked , we can use suppress further renderting.
3. Add following code in Parent.razor

// Should Render

bool shouldRender = true;

int count = 0;

public void TurnOff()

{

shouldRender = false;

}

public void TurnOn()

{

shouldRender = true;

}

public void IncrementCount()

{

count++;

}

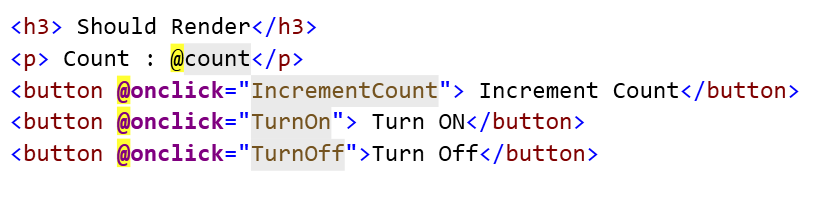
protected override bool ShouldRender()

{

return shouldRender;

}

1. Add a Display logic



1. Check the Output . updates will work only if ShouldRender is ON

**What is gRPC**

* gRPC is an Open Source RPC technology
* Developed by Google back in 2015
* Roughly 7 to 10 times faster than Standard WebAPI.
* Supports Bi-Directional communication and everything you need , from enterprise solution to another youtube data streaming solution.

**What is HTTP2 protocol**

HTTP/2 is fully multiplexed. This means that HTTP/2 can send multiple requests for data in parallel over a single TCP connection. This is the most advanced feature of the HTTP/2 protocol because it allows you to download web files via ASync mode from one server. Most modern browsers limit TCP connections to one server

Server Push  
The server can asynchronously send resources to the client's cache for future use.

Binary protocol  
Data will be transmitted as binary frames. Not as text form in the HTTP/1.1.

gRPC does the same thing as WEB API where it takes request message and gives back response to caller over web network . But key difference are

1. REST works on HTTP protocol while gRPC works on HTTP 2 protocol
2. REST depends on the Endpoint for data communication by the client while gRPC depends on Service implementation at the Protocol buffers.
3. REST uses JSON that is heavier and human readable while gRPC uses binaries which is more faster than any other existing form of payload data type.
4. Possible to fetch data directly by accessing the API endpoint while gRPC client is needed to fetch data.

**Protocol Buffers**

* Way to Serialize data , much more simpler , compact and faster.
* Protos support huge variety of language including C#, Python , GO, Java etc
* Designed in a way that it is much faster than other formats, thanks to smaller sizes due to serialized binary strings.

**How gRPC Works**

The flow starts with creating a gRPC Server. Next, you create proto files that not only contains definitions of the response and requests but also services that has to implemented by the server. Once the proto files are ready, you would have to add a reference to these files as a service in both the client and server projects.

As soon as the client invokes the Services defined within the Proto file and requests to the port where the server is running, the Service implementation at the server side gets invokes and retuns the data in binary format which will be further de-serialized to the response object that is defined in the configuration file (proto). This is the basic idea. You will be more clear with the concept when we start implementing.

gRPC defines 4 Types of Service Methods

1. Unary :- Client Send a Single Request to the Server and get a Single Response back.
2. Server Streaming :- Client Sends a request to the Server and gets a stream to read a sequence of messages back.
3. Client Streaming :- Client writes a sequence of messages and sends them to the Server.
4. Bidirectional Streaming :- Both sides send a sequence of messages using read-write stream.

Note : Metadata : Information about a particular RPC call such as authentication details in the form of key value pairs.

[Core concepts, architecture and lifecycle | gRPC](https://grpc.io/docs/what-is-grpc/core-concepts/)

Unary RPC

* Once the Client calls a Stub method , the Server is notified that RPC has been invoked with the Clients Metadata for this call, method name .
* Server performs whatever works is necessary to create and populate response. Response is then returned to the client with status details and optional trailing metadata.
* If the response status is OK then the client get the response which completes the call on the client side.

Server Streaming RPC

* Its Similar to unary RPC except server returns a stream of messages in response to clients request.
* After sending all its messages the server status details and trailing metadata send to the client. The client completes once it has all the servers messages.

Client Streaming RPC

* Similar to Unary RPC, But here client sends a stream of messages to the server instead of a single messages
* Server responds with a single messages.

Package : package specifier affect the generated code depends on the language you are using. For C# it’s a namespace while its used as package name in java.

Bi-directional streaming RPC

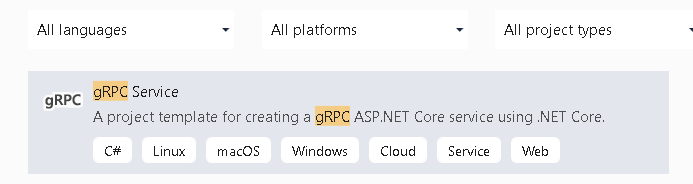
* The call is initiated by the client invoking the method and server receiving client metadata , method name and deadline. The server can choose to send back its initial metadata or wait for the client to start streaming messages.
* Client and server side stream processing is application specific also 2 streams are independent and client and server can read and write messages in any order.

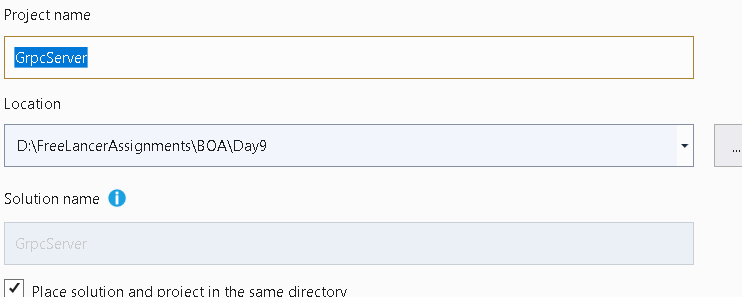
What is ServerSidecontext

* Context for a server-side call.

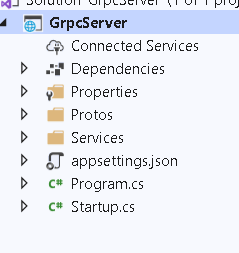
Demo 1:

1. Add New grpc Project name it as gRPC Server





Project Structure



Greet.proto : Interface description language

Services : Implementation of Service

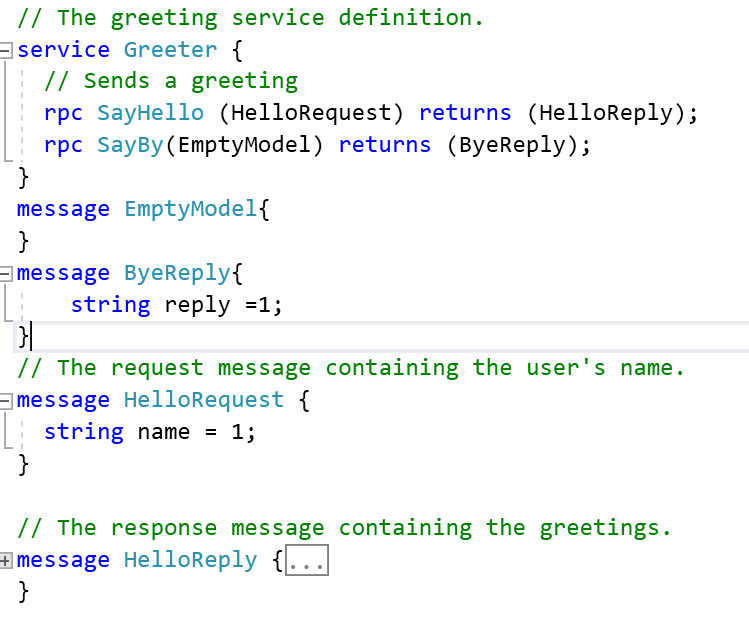
appSetting.json : Configuration

Program.cs

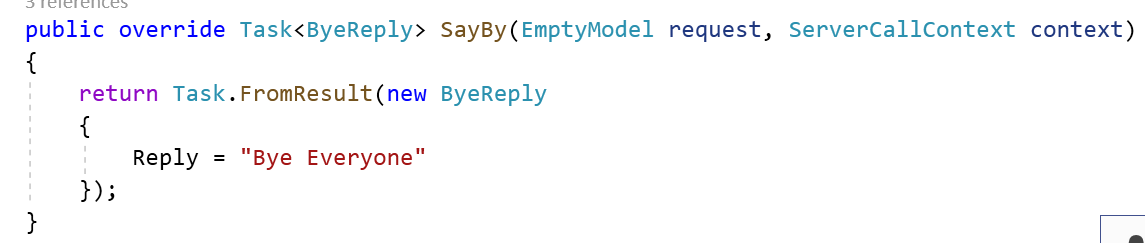
Startup.cs

gRPC Client

1. Grpc.Net.Client : Contains NET Core Client
2. Google PRotobuf : Contains Protobuf message APIS for C#
3. Grpc.Tools : C# tooling support , use to generate class assets at client and server side.
4. Add Following code in proto classes

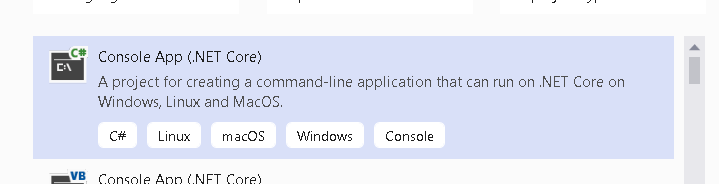


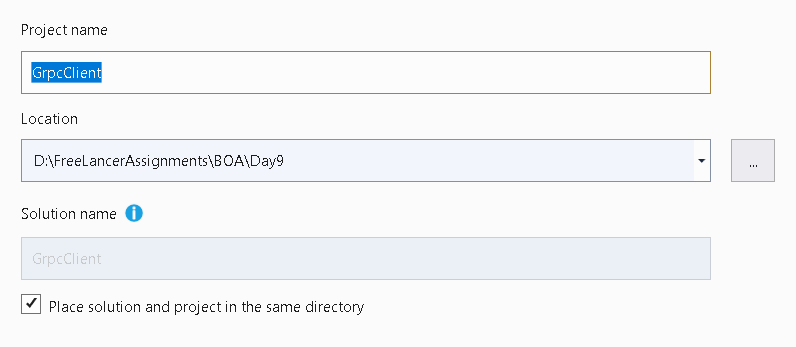
1. Implement it in C# Greeter Service



Create a Client that can consume this service

1. Add New Console Application **GrpcClient**





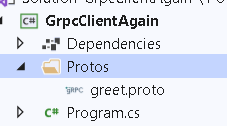
1. Install following libraries

Install-Package Grpc.Net.Client

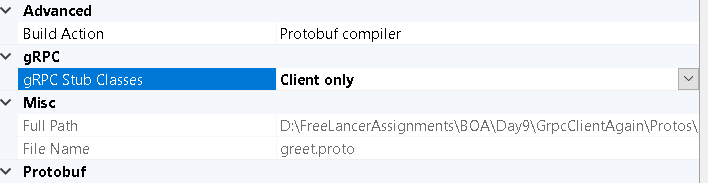
Install-Package Google.Protobuf

Install-Package Grpc.Tools

1. Create Protos folder and copy and paste .proto file from Server to Client and Change its namespace to the current project namespace.



1. Right Click on the proto file and Change its build action to proto compiler and Client only



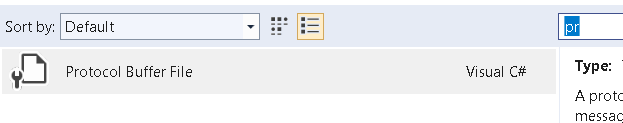
1. Create the Greeter Client



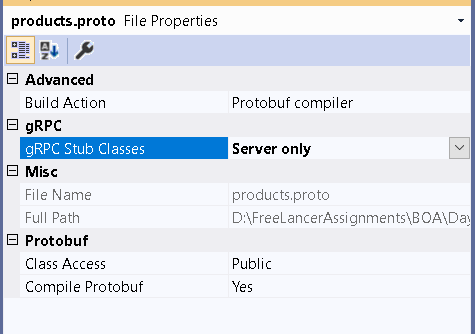
1. Check the Output

Demo 2: Create One more Service that is ProductService

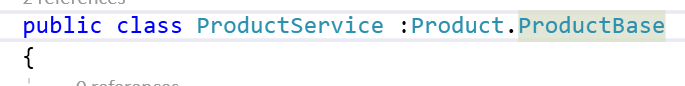
1. Add new Proto file



1. Add Compilation for the proto



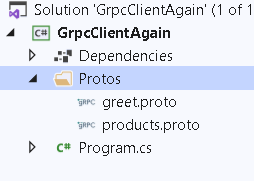
1. This will generate Stub classes



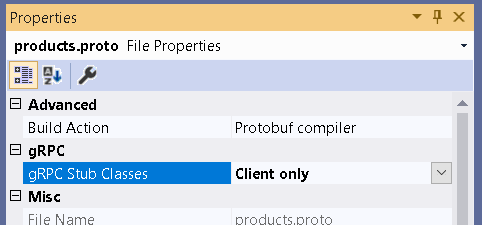
1. Add a code for ProductByID



1. Copy Products.proto file in the client and change the namespace



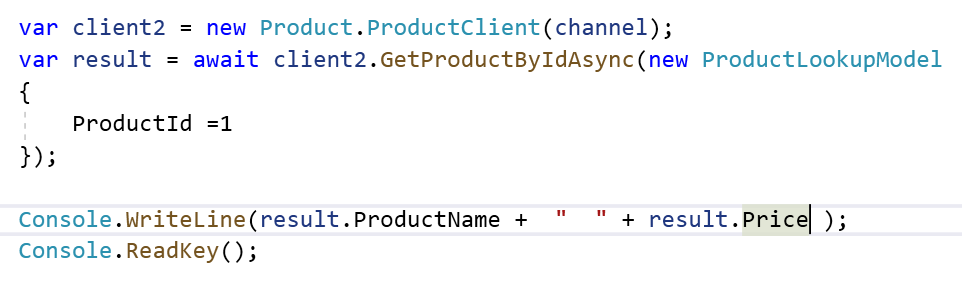
1. Go to Properties of protofile and set its properties



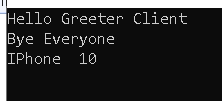
1. Register ProductService in Configure



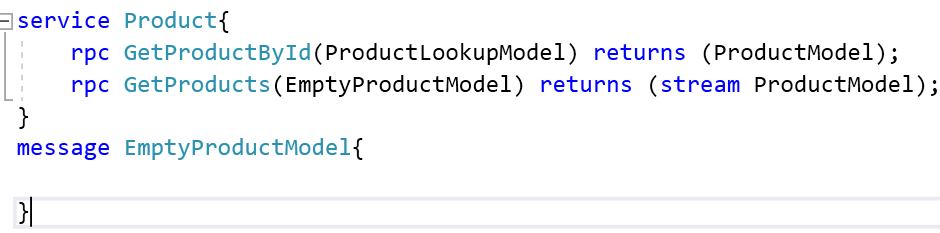
1. Add Following code in a client



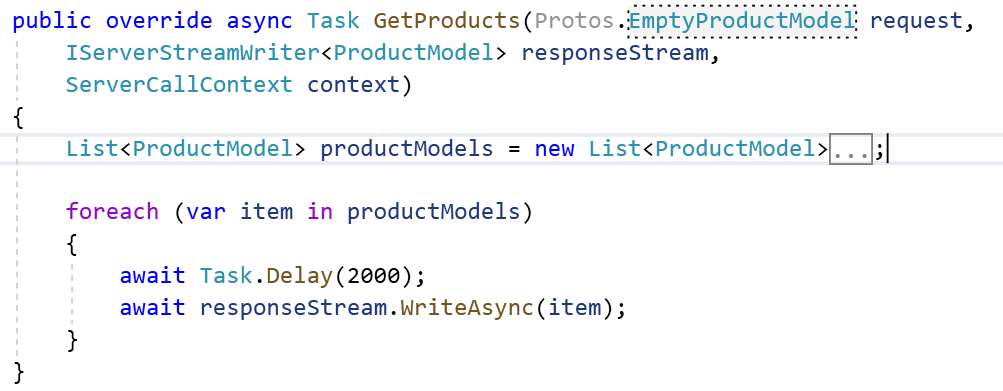
1. Run the client



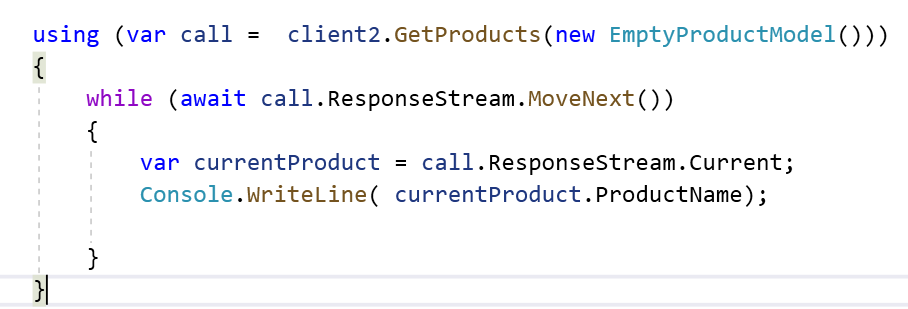
1. Add GetProducts in the Server



1. Implement it



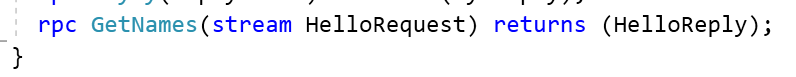
1. Copy content new file into Client proto file.
2. Add a code in the client



1. Check the Output

Demo Client Streaming

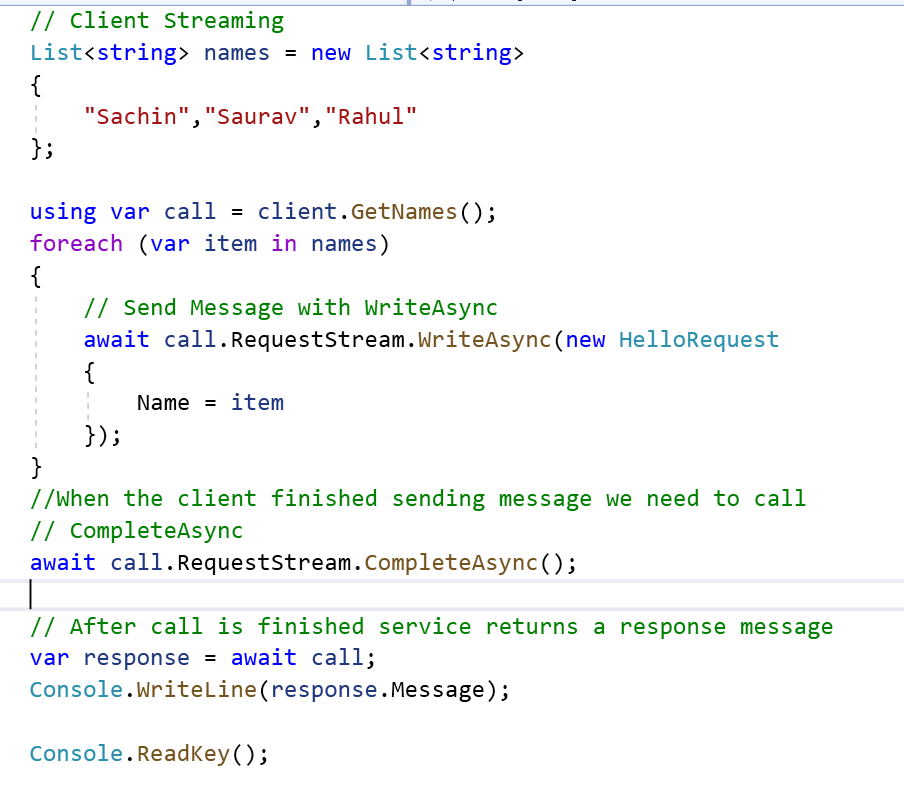
1. Add Method in Server



1. Implement the same



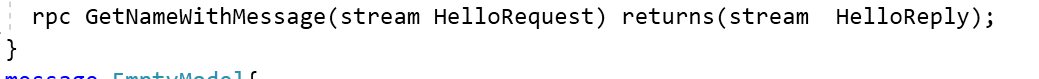
1. Copy the Content of Greet.proto and paste it into Greet.Client
2. Call the GetNames from the Client



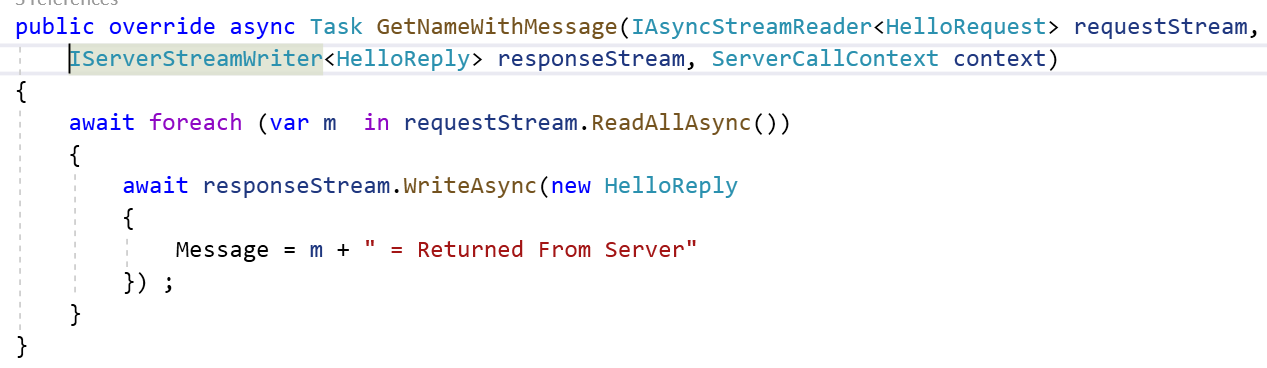
1. Check the Output

Bi-directional Streaming

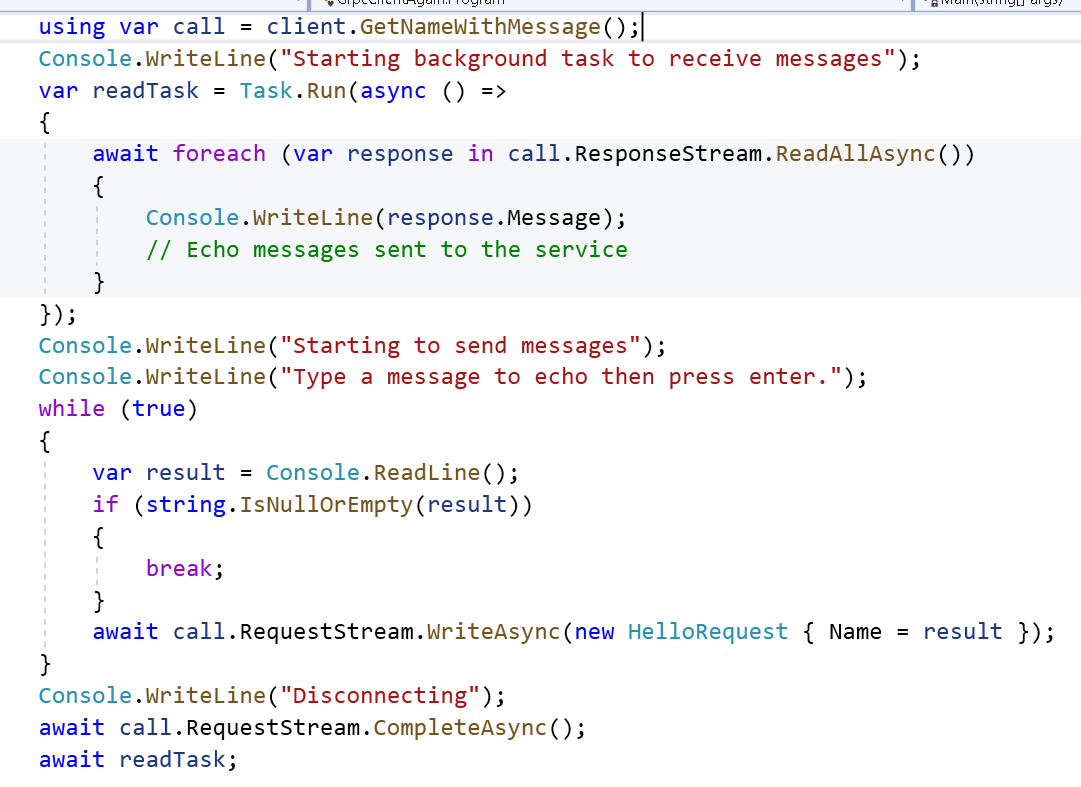
1. Add a Method in Greeter Service



1. Implement it

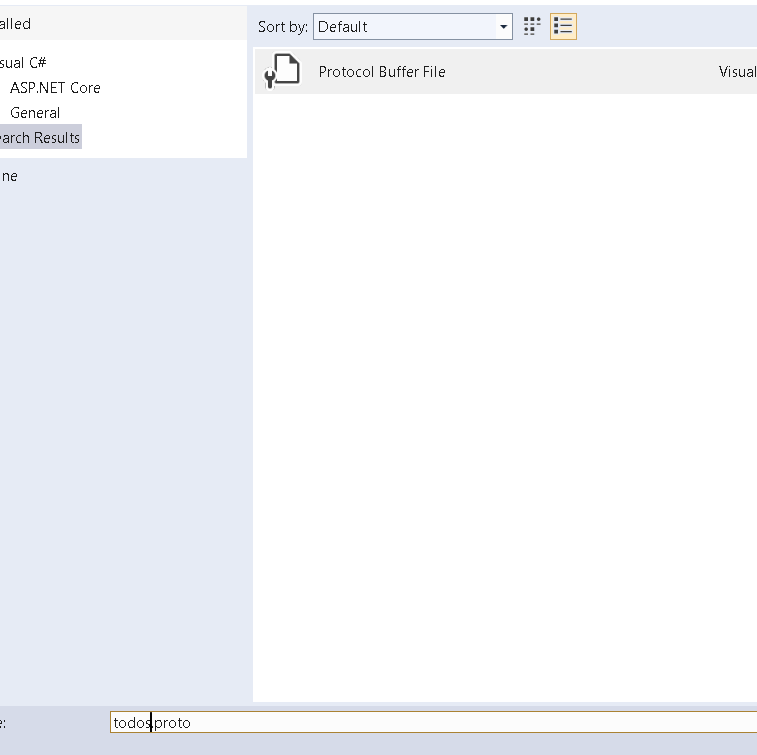


1. Call this method at client side



Connecting to Database

1. Add Proto file

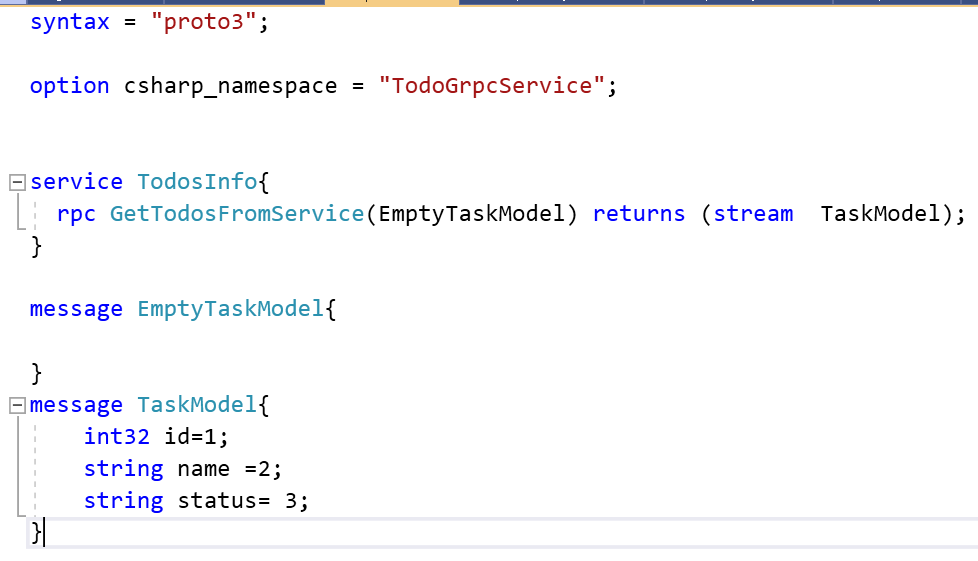


How to Connect to Database

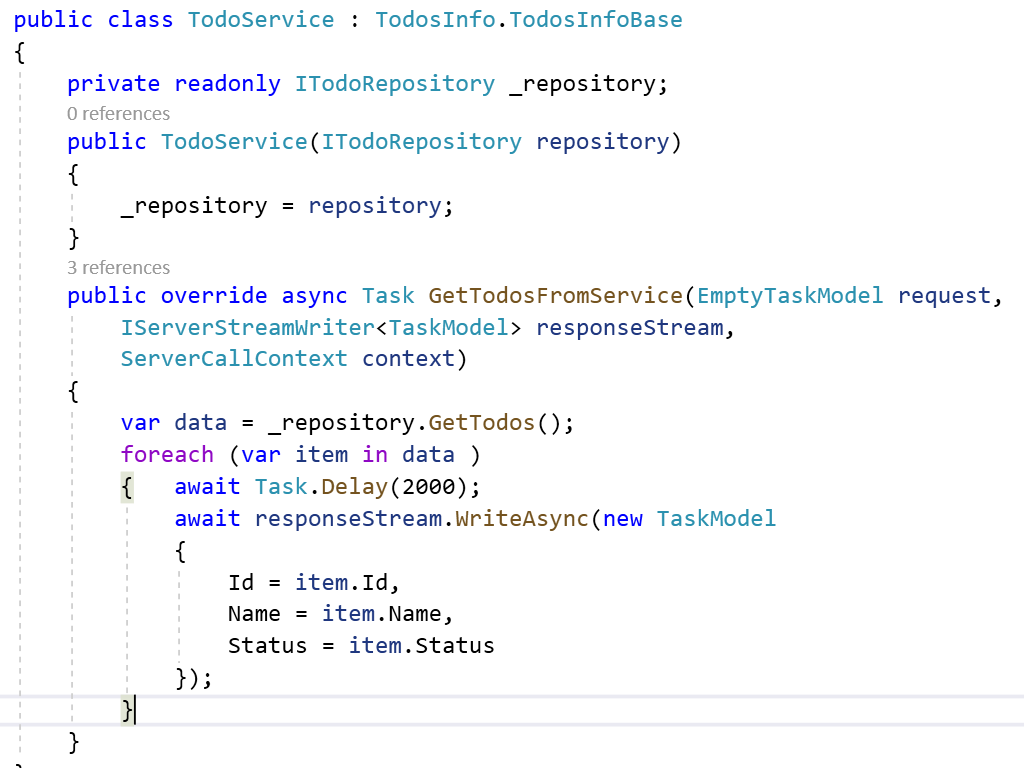
1. Add todo.proto file



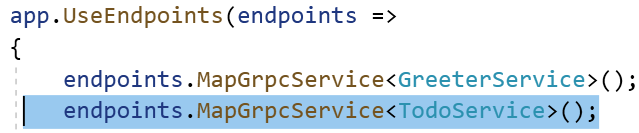
1. Add proto



1. Right Click on proto file and set it as Client only and Proto Compiler
2. Add TodoService



1. Add Service in Startup.cs



1. Add Service object in ConfigureService



1. Create Client , Copy proto file and set its compiler and make it client only



1. Check the Output

Migration

<https://docs.microsoft.com/en-us/dotnet/core/porting/>

* Porting from .NET to .NET is relatively straightforward
* Complexity of Project dictates how much work you will do after the initial migration of the project files.
* Projects where app model is available in .Net Core Such as libraries, Console apps and desktop apps usually require little change
* Project that require a new app model such as moving to ASP.Net Core from ASP.Net require more work.

Unavailble Tech

1.Remoting

2. CAS

3. WF and WCF

Consider following thing before migrating to Windows Forms

1. Project Files for .Net Core use a different format than .Net framework
2. Your project may use an API that isn’t available in .Net Core
3. Third party controls and libraries may not have been ported to .NET and remain only available to .NET framework.
4. Your project uses a technology that is no longer available in .NET

Migration Tools Available for Porting application to .Net Core

1. .**Net Portability Analyzer**

* ToolChain that can generate a report of how portable your code is between .Net and .Net Core

It can be used in 2 ways

1. Command line tool
2. Visual Studio Extension
3. **Platform Compatibility analyzer**

* he analyzer identifies all APIs that are not cross-platform
* Inform developers when they use platform specific APIS from call sites where the API might not be available.
* It identifies all APIS that are not cross platform.

For Example :

Console.WindowWidth works on Windows but not on Linux and macos.

<https://docs.microsoft.com/en-us/dotnet/standard/analyzers/api-analyzer>

1. **.Net API analyzer**

* Help detect platform compatibility issues in cross platform apps and libraries.
* .Net API analyses whether or not you are using an API that will throw PlatformNotSupportException at run time.
* It discovers Potential compatibilities risks for C# APIS on different platforms and detects calls to deprecated APIs
* Use ful for C# dev during dev
* This is the APIS that all are not supported in .Net Core

<https://docs.microsoft.com/en-us/dotnet/core/compatibility/unsupported-apis>

* Its available as Nuggets package MicrosoftDotNet.Analyzers.Compability
* When you add it in project it automatically monitors code and indicates problematic API usage.
* Its still in pre-release version.

1. **Try-Convert :**

* Tool that can convert a project or entire solution the .Net SDK.
* IT can move desktop apps to .Net Core.
* Its may reject many project types that are incompatible with .NET core.
* This tool is not recommended if your project has custom tasks, targets or imports.
* This tool work only on windows.
* Class libraries or code with no platform specific code , this tool will help a lot.
* How it works : <https://github.com/dotnet/try-convert>

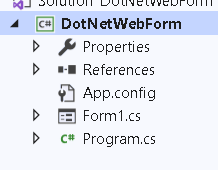
Demo 1 : Migration From Windows Forms to .Net Windows Forms

Demo 1: How to Migrate Application From Windows Forms to .Net Core Windows App

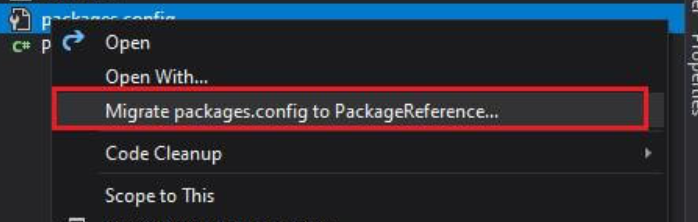
<https://docs.microsoft.com/en-us/dotnet/desktop/winforms/migration/?view=netdesktop-5.0&preserve-view=true>

Migrating .Net Windows App to .Net Core App

1. Go to DornetwindowsForm Application

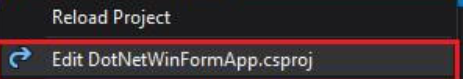


1. Install Entity Framework
2. Right Click on Packages reference and Select Migrate Option



1. Packages information will be move to .csproject so this file will not be available
2. Right Click and unload the project and Select EditProject





1. Cut the content from this file and keep it in notepad
2. Paste this 4 settings in the file

<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

<PropertyGroup>

<OutputType>WinExe</OutputType>

<TargetFramework>netcoreapp3.1</TargetFramework>

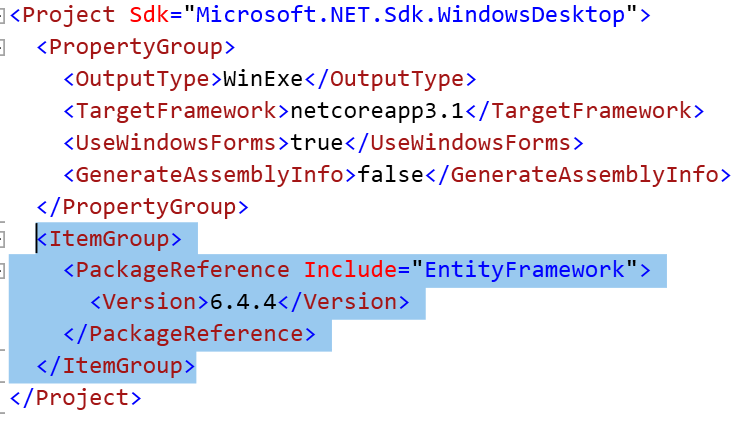
<UseWindowsForms>true</UseWindowsForms>

<GenerateAssemblyInfo>false</GenerateAssemblyInfo>

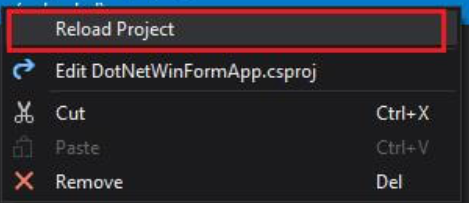
</PropertyGroup>

</Project>

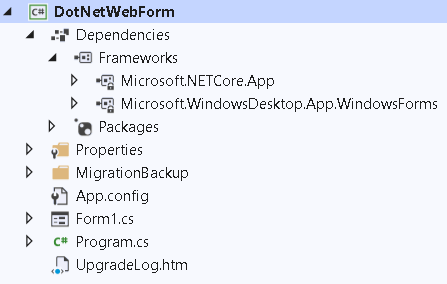
1. Add a Packages information into it



1. Reload the Project



1. Check the References



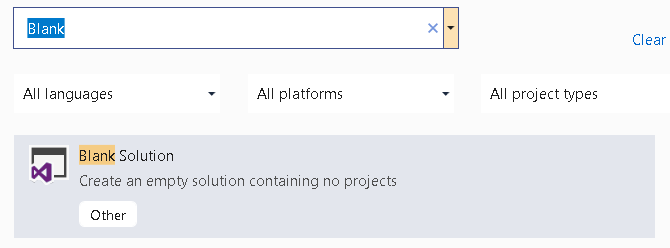
Demo 2: Migration From Web API 2 to .Net Core Web API

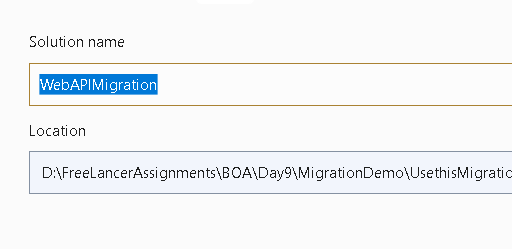
Migrating .Net Web API to .Net Core API : [Migrate from ASP.NET Web API to ASP.NET Core | Microsoft Docs](https://docs.microsoft.com/en-us/aspnet/core/migration/webapi?view=aspnetcore-5.0)

1. Open ProductWebAPI application

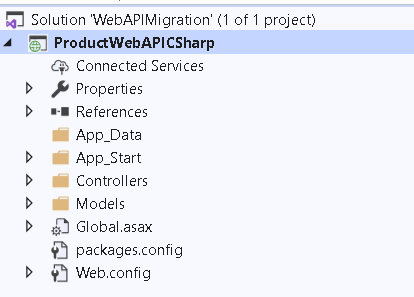
D:\FreeLancerAssignments\BOA\Day9\MigrationDemo\UsethisMigration\ProductWebAPICSharp

1. Run this and type <http://localhost/api/products>
2. Migrate this application to ASP.Net Core
3. Open Visual Studio and Create new Blank Solution Name it as WebAPI Migration

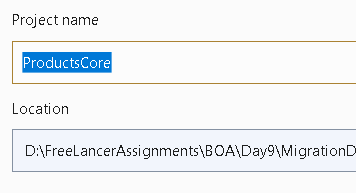


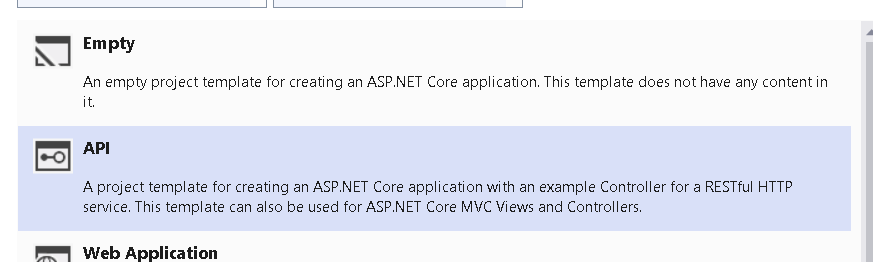


1. Add Existing Project

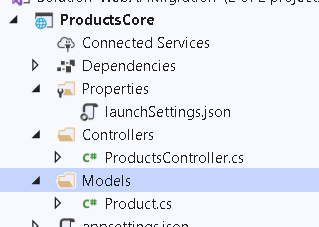


1. Add New ASP.Net Core Project and add API and Name it as ProductCore





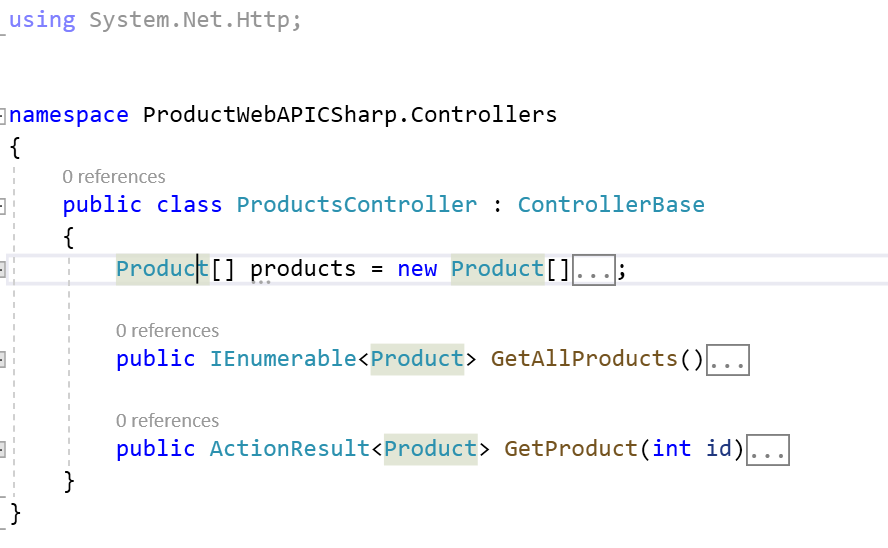
1. Remove Weather Related files
2. Copy Controller and product model from the API to core project



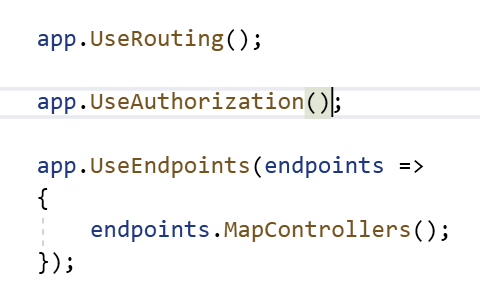
1. Go to ProductController

Following thing don’t exist in ASP.Net core so remove it and replace

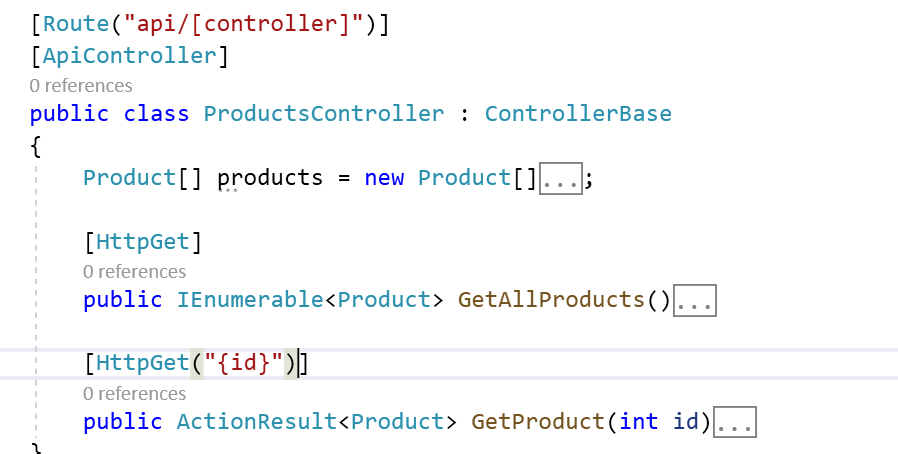
1. APIController = ControllerBase
2. System.Web.Http = using Microsoft.AspNetCore.Mvc
3. IHttpActionResult Interface :ActionResult<Product>
4. It look like this



1. Replace the Routing Configuration : ASP.Net Core doesn’t include webapi config so configure routes for the same in ASP.Net Core



1. Mark the ProductsController with following attributes. Route , HttpGet and ApiController



1. Change the Launch Setting



1. Check the Output

