Lab 10

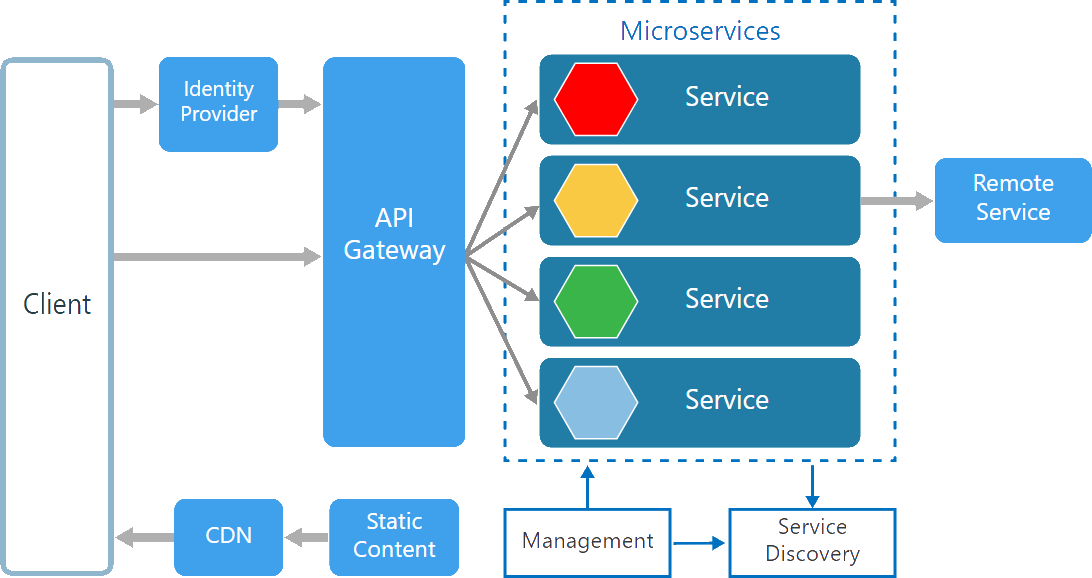
Microservice using ASP.NET Core

# Microservices

The term microservices portrays a software development style that has grown from contemporary trends to set up practices those are meant to increase the speed and efficiency of developing and managing software solutions at scale. Microservices is more about applying a certain number of principles and architectural patterns an architecture. Each microservice lives independently, but on the other hand, also all rely on each other. All microservices in a project get deployed in production at their own pace, on- premise on the cloud, independently, living side by side.

# Microservices Architecture

The following picture from [Microsoft Docs](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices) shows the microservices architecture style.



There are various components in a microservices architecture apart from microservices themselves.

**Management**. Maintains the nodes for the service.

**Identity Provider**. Manages the identity information and provides authentication services within a distributed network.

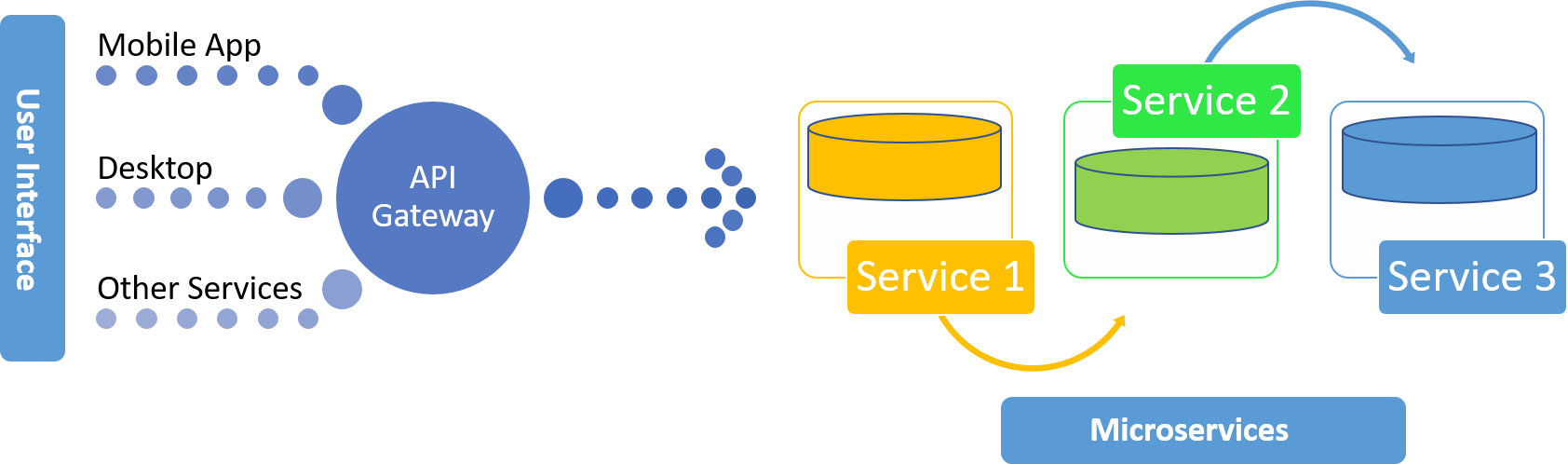
**Service Discovery**. Keeps track of services and service addresses and endpoints.

**API Gateway**. Serves as client’s entry point. Single point of contact from the client which in turn returns responses from underlying microservices and sometimes an aggregated response from multiple underlying microservice

**CDN**. A content delivery network to serve static resources for e.g. pages and web content in a distributed network

**Static Content** The static resources like pages and web content

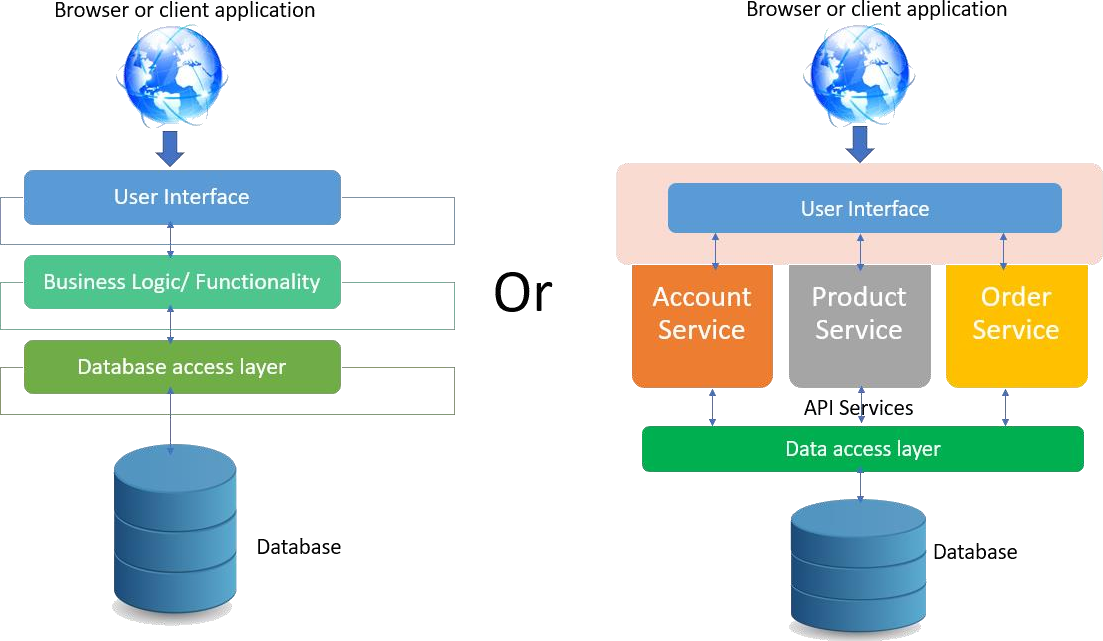
Microservices are deployed independently with their own database per service so the underlying microservices look as shown in the following picture.



# Monolithic vs Microservices Architecture

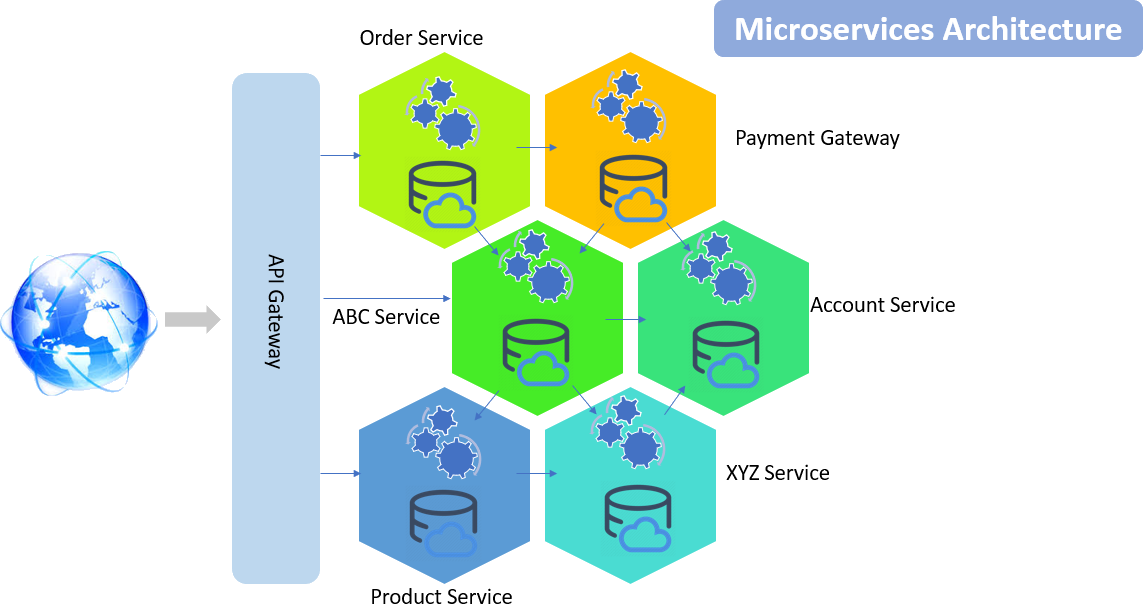
Monolithic applications are more of a single complete package having all the related needed components and services encapsulated in one package.

Following is the diagrammatic representation of monolithic architecture being package completely or being service based.



Microservice is an approach to create small services each running in their own space and can communicate via messaging. These are independent services directly calling their own database.

Following is the diagrammatic representation of microservices architecture.



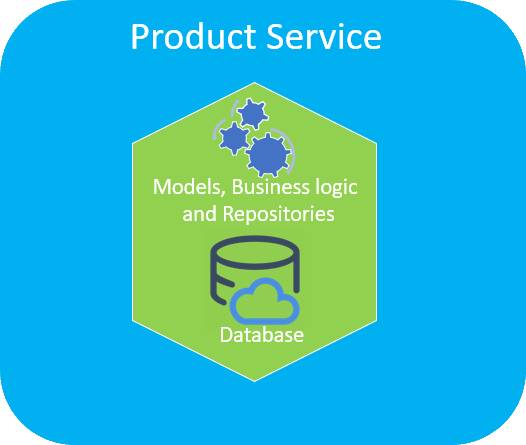
In monolithic architecture, the database remains the same for all the functionalities even if an approach of service-oriented architecture is followed, whereas in microservices each service will have their own database.

# Docker Containers and Docker installation

Containers like Dockers and others slice the operating system resources for e.g. the network stack, processes namespace, file system hierarchy and the storage stack. Dockers are more like virtualizing the operating system.

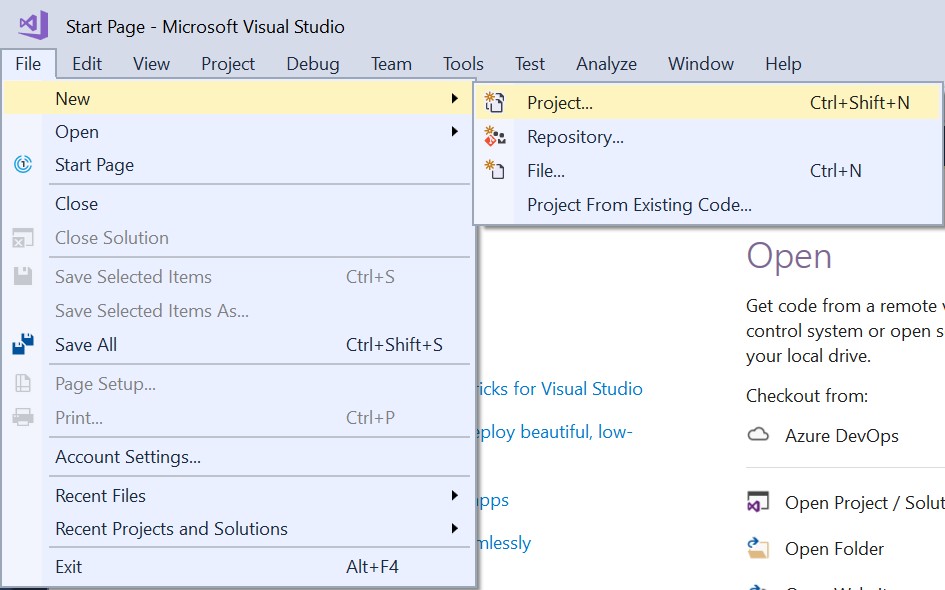
# Microservice using ASP.NET Core

This section will demonstrate how to create a Product microservice using ASP.NET Core step by step with the help of pictures. The service would be built using ASP.NET Core 2.1 and Visual Studio 2017. Asp.NET Core comes integrated with VS 2017. This service will have its own dbcontext and database with the isolated repository so that the service could be deployed independently.

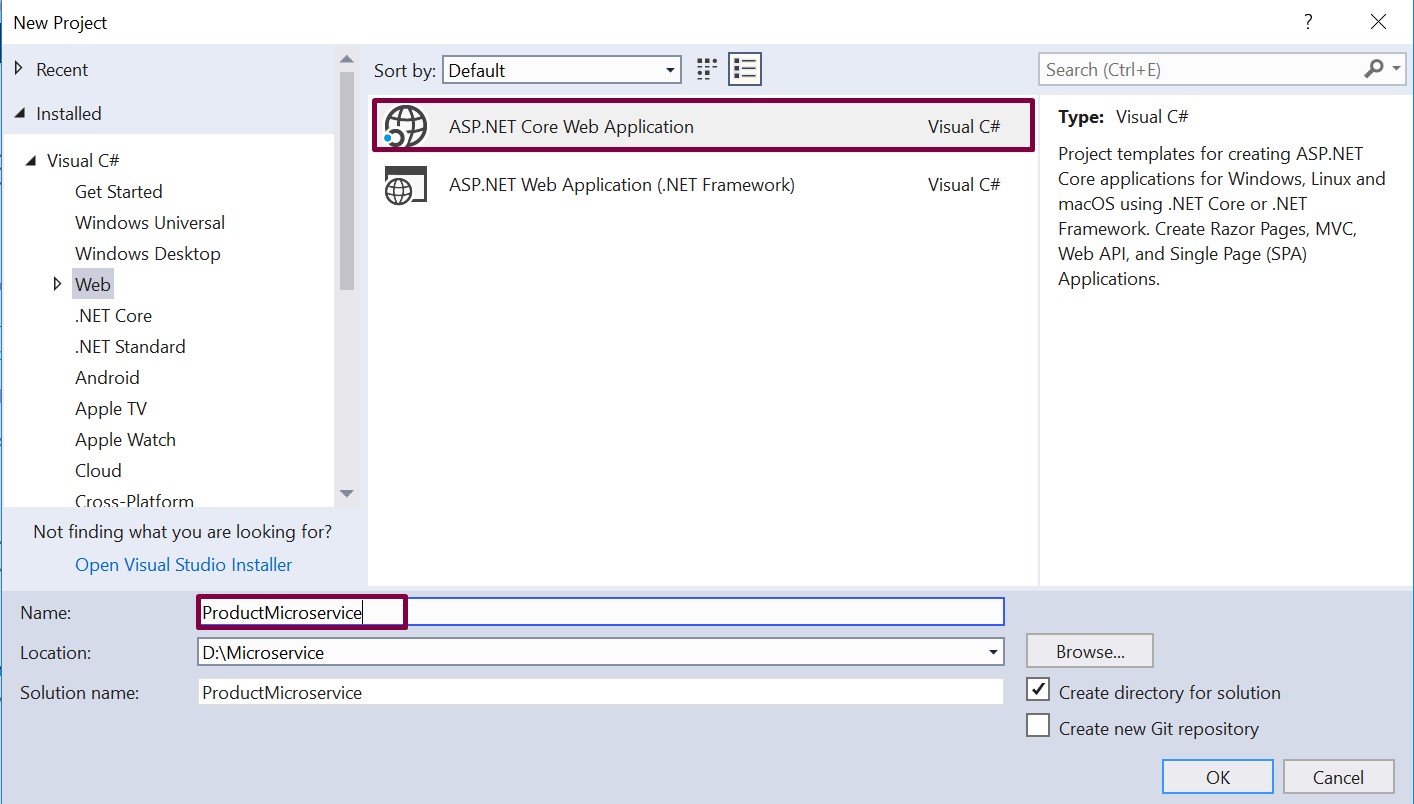


## Creating an Asp.NET Core Application Solution

1. Open the Visual Studio and add a new project.

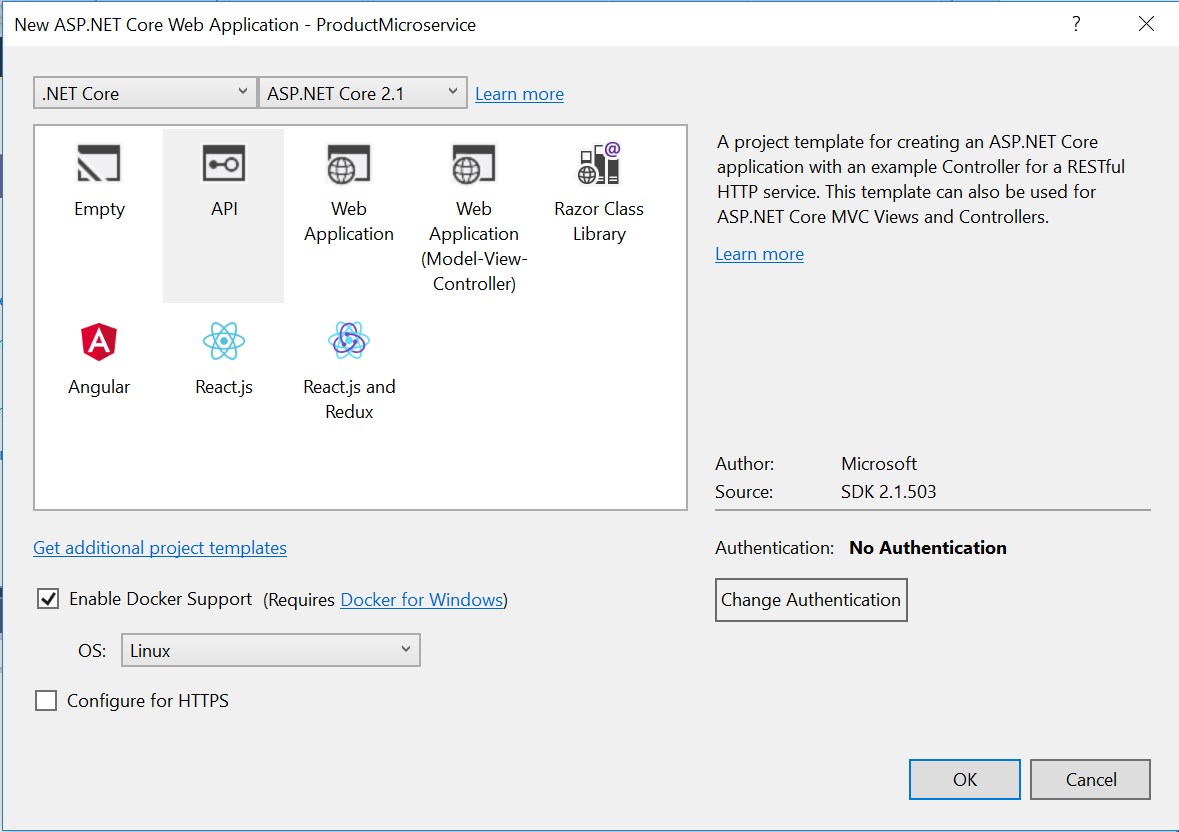


1. Choose the application as ASP.NET Core Web Application and give it a meaningful name.

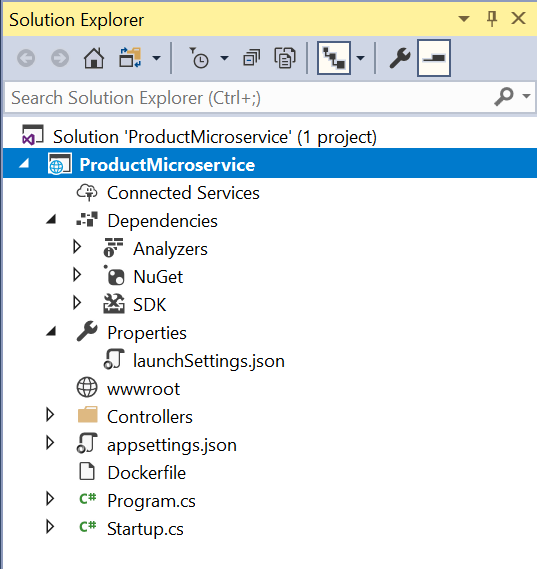


1. Next, choose API as the type of the project and make sure that “Enable Docker Support” option is

selected with OS type as Linux.

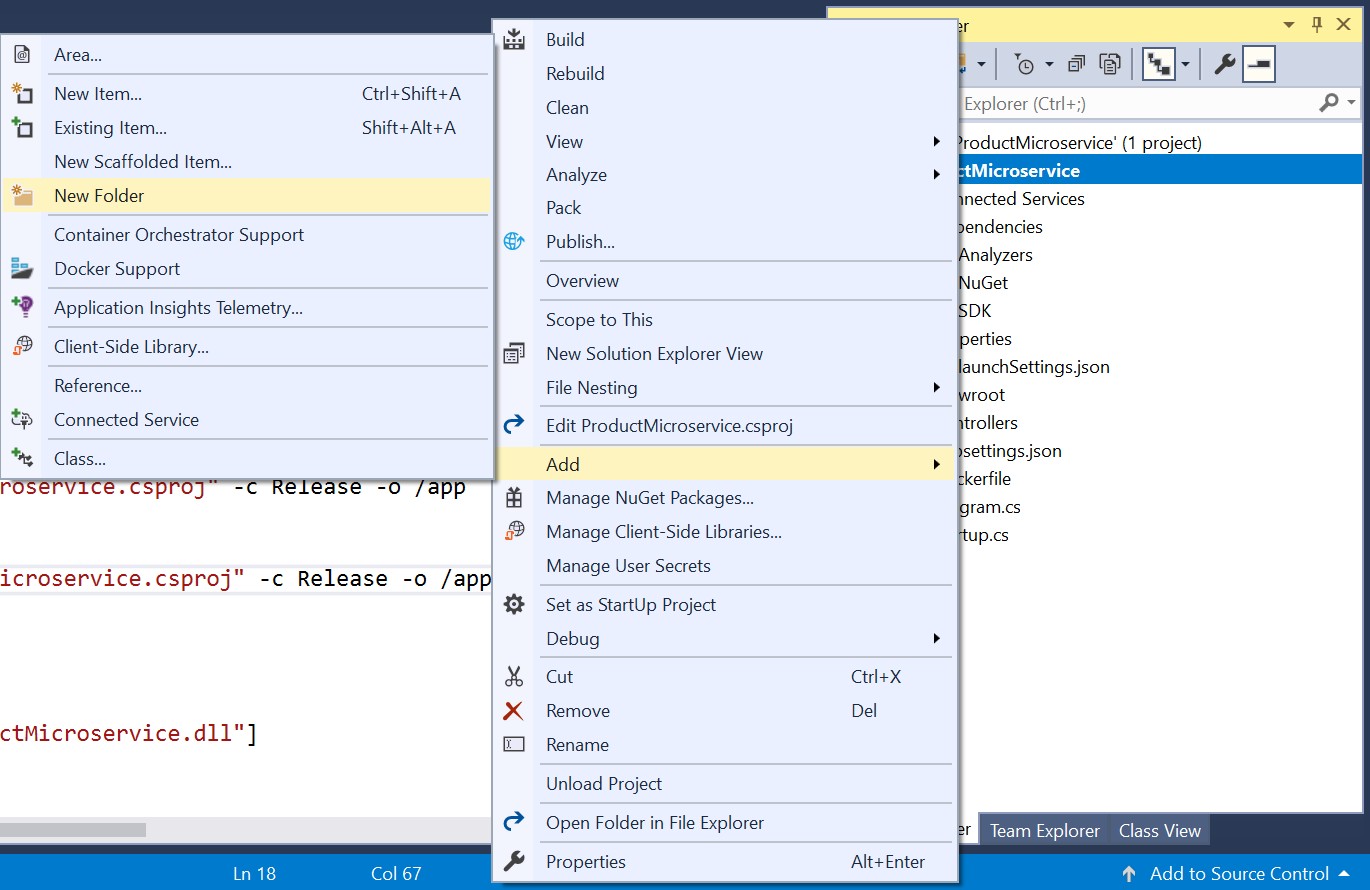


1. The solution will look as shown below.

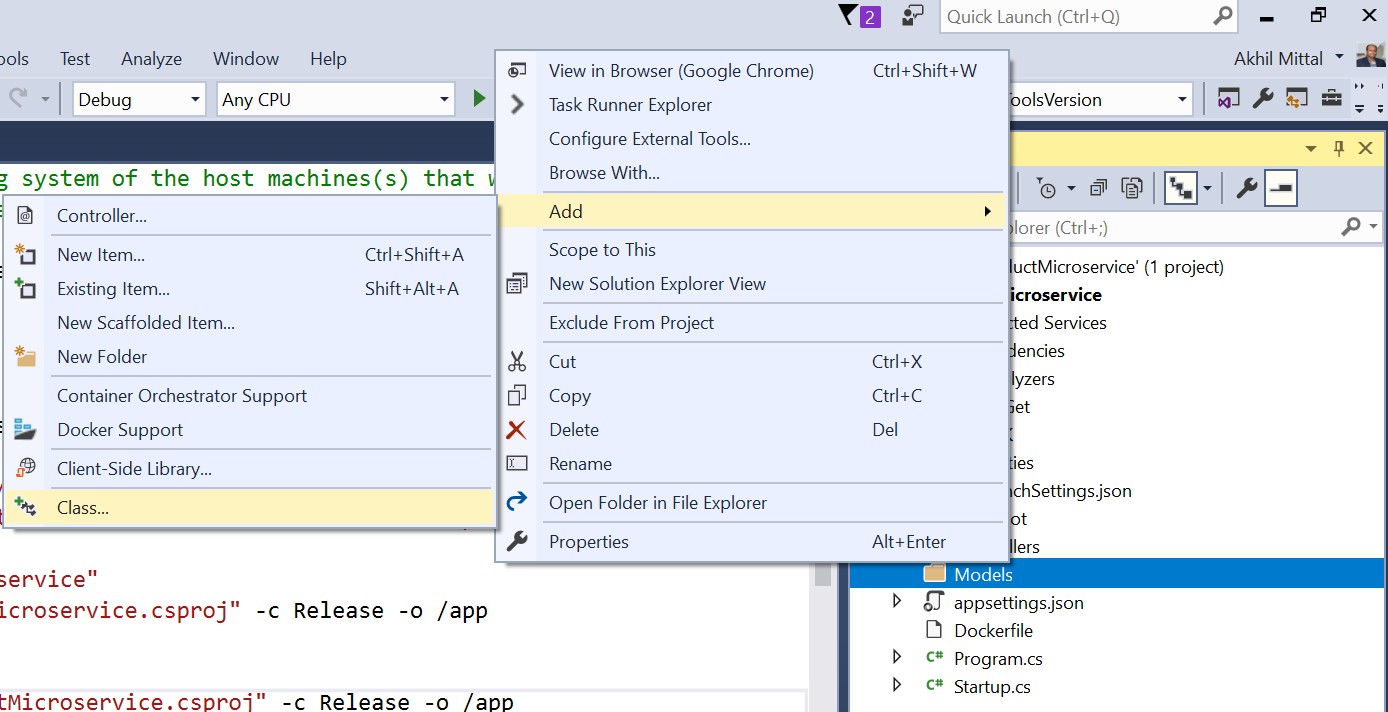


## Adding Models

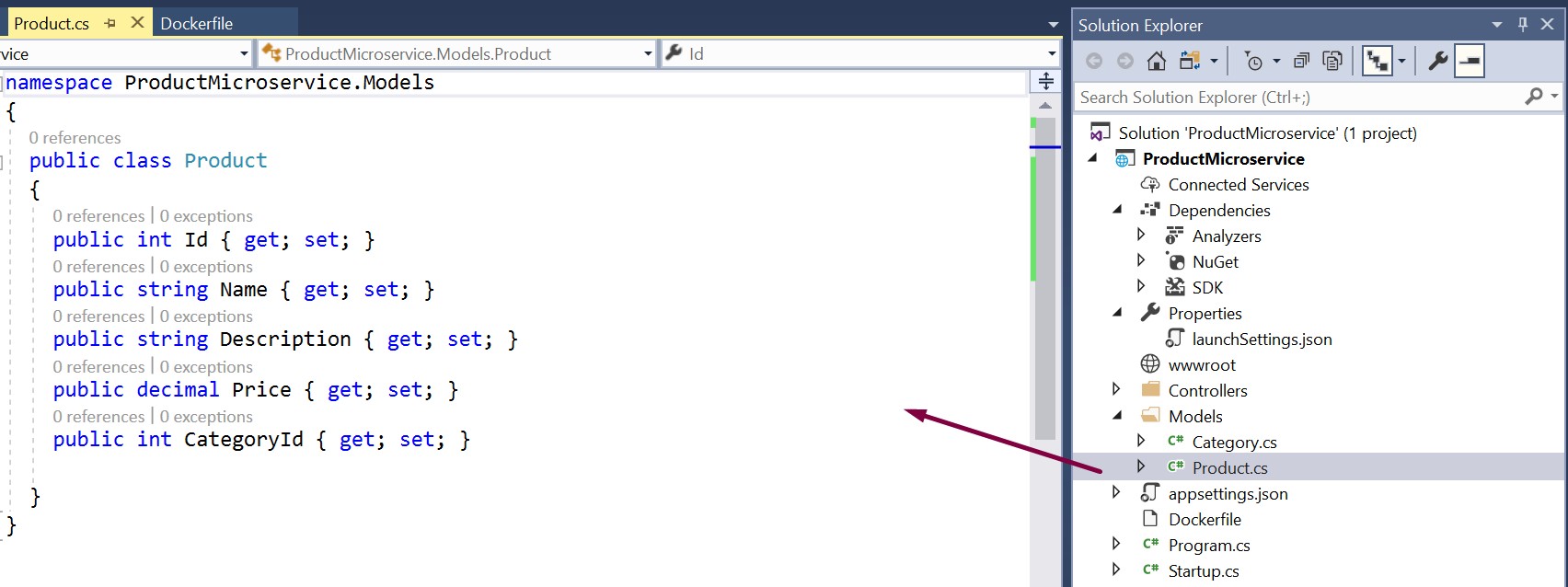
1. Add a new folder named “Model” to the project.



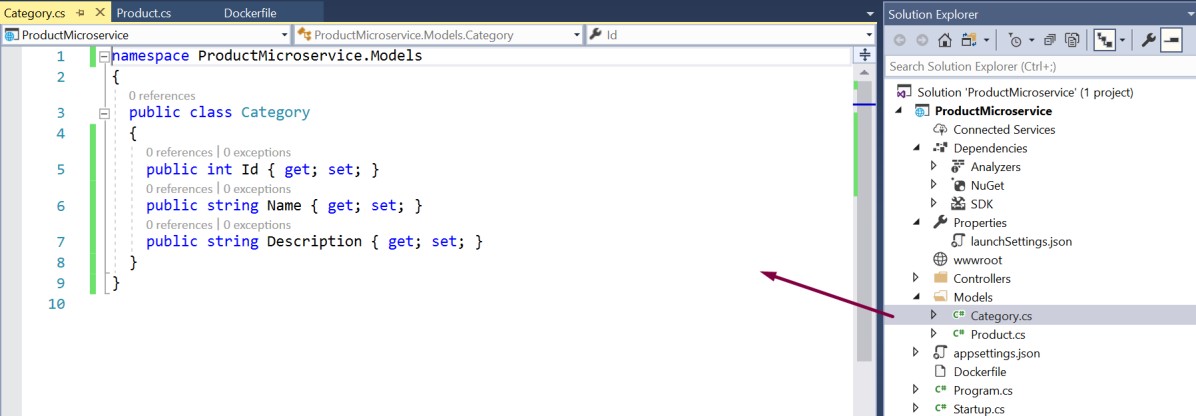
1. In the Models folder, add a class named Product.



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

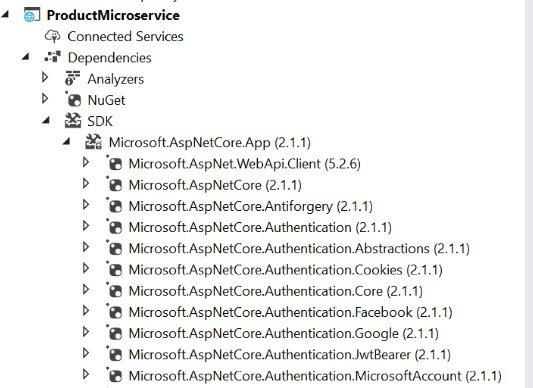


1. Similarly, add Category model.



## Enabling EF Core

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



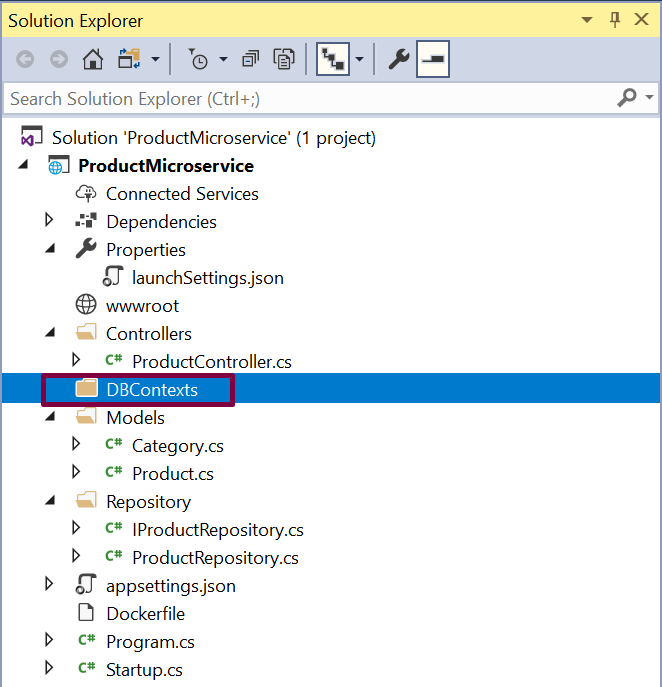
Microsoft.EntityFrameworkCore.SqlServer (2.1.1) should be the package inside the downloaded SDK’s. If it

is not present, it could be explicitly added to the project via Nuget Packages.

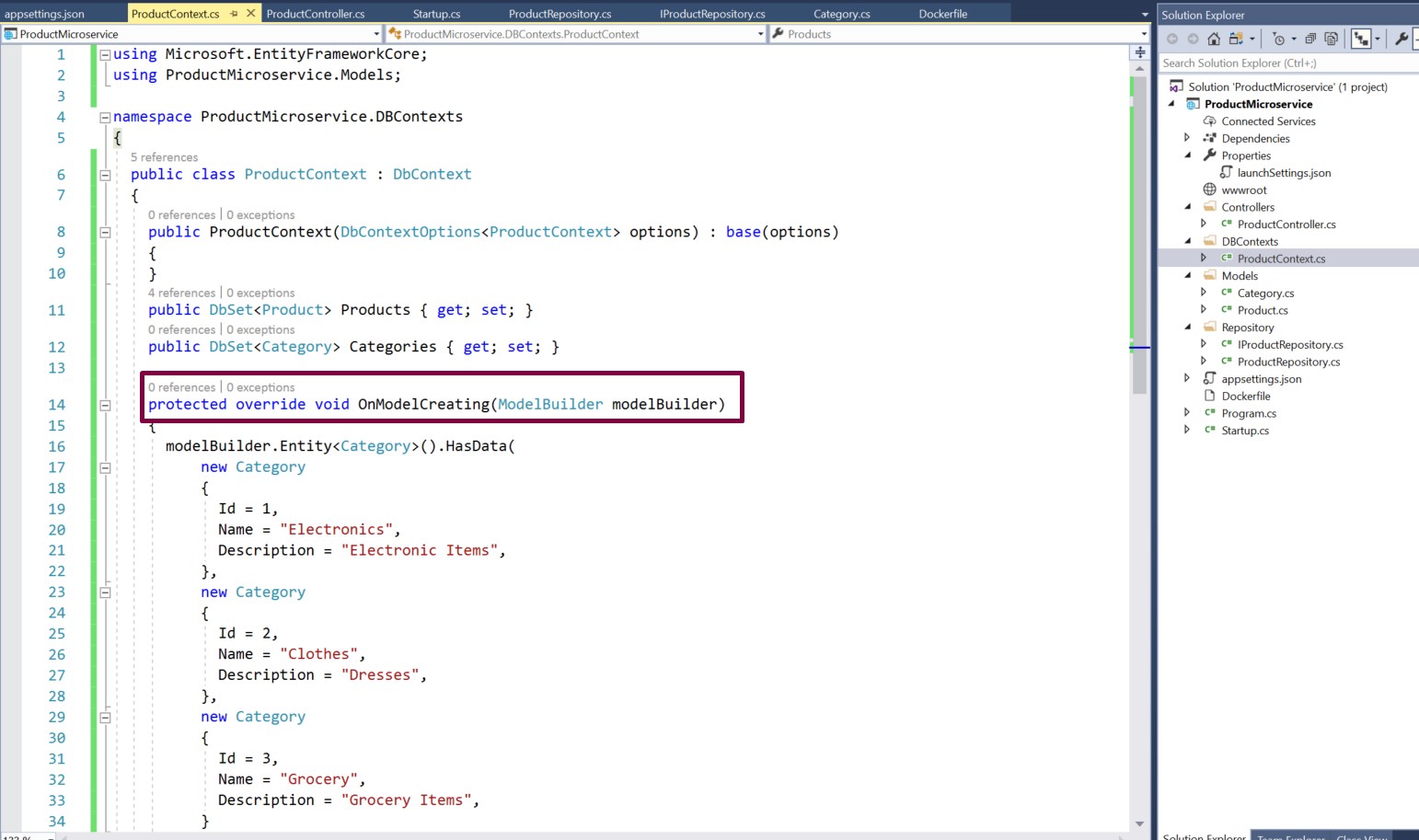
## Adding EF Core DbContext

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

1. Add a new folder named DBContexts to the project.



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



### ProductContext code:

using Microsoft.EntityFrameworkCore; using ProductMicroservice.Models;

namespace ProductMicroservice.DBContexts

{

public class ProductContext : DbContext

{

public ProductContext(DbContextOptions<ProductContext> options) : base(options)

{

}

public DbSet<Product> Products { get; set; } public DbSet<Category> Categories { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Category>().HasData( new Category

{

Id = 1,

Name = "Electronics", Description = "Electronic Items",

},

new Category

{

Id = 2,

Name = "Clothes", Description = "Dresses",

},

new Category

{

Id = 3,

Name = "Grocery", Description = "Grocery Items",

}

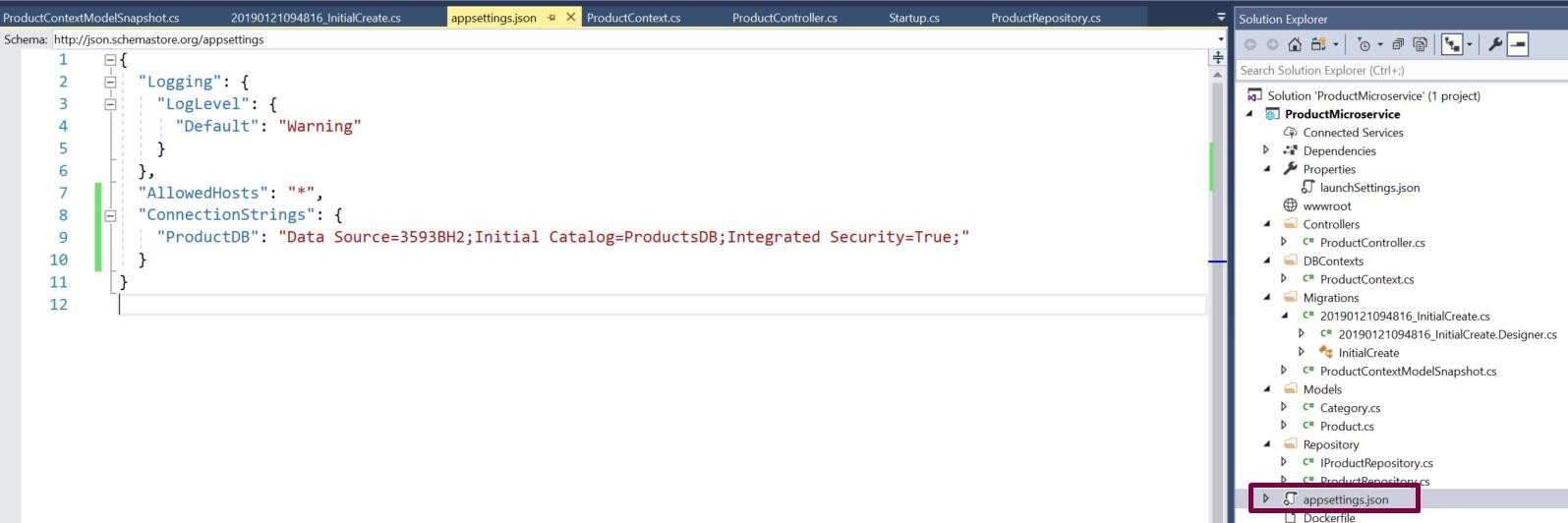
);

}

}

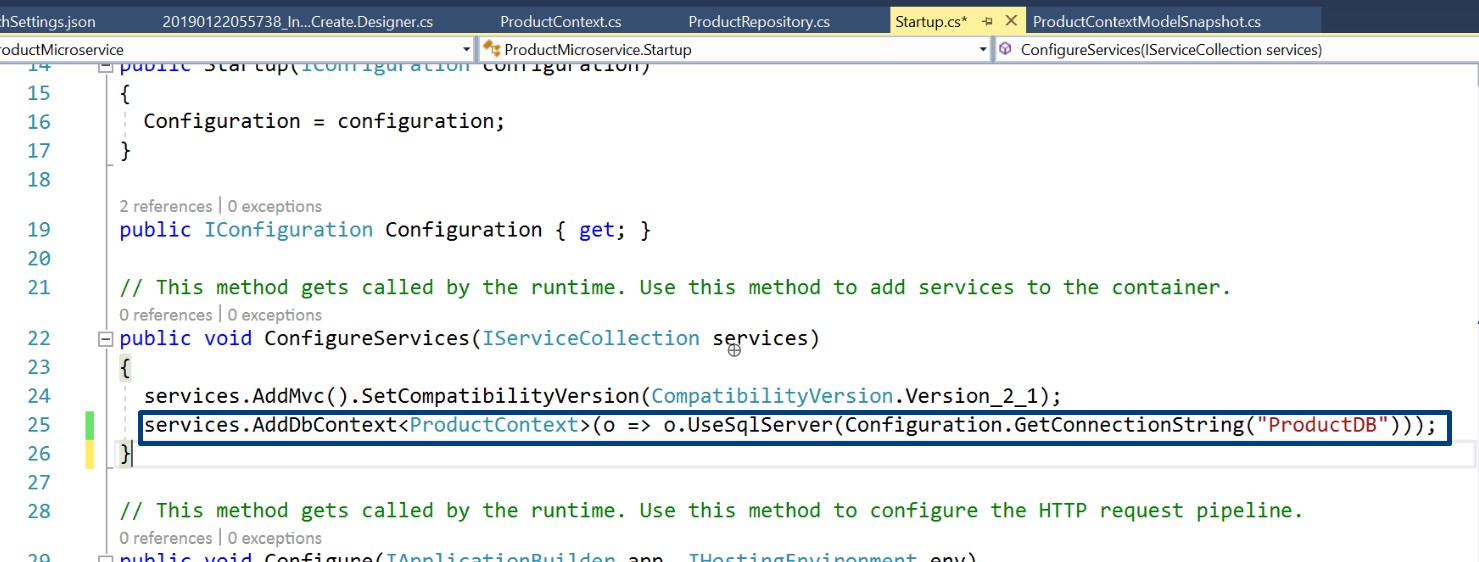
}

1. Add a connection string in the appsettings.json file.



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

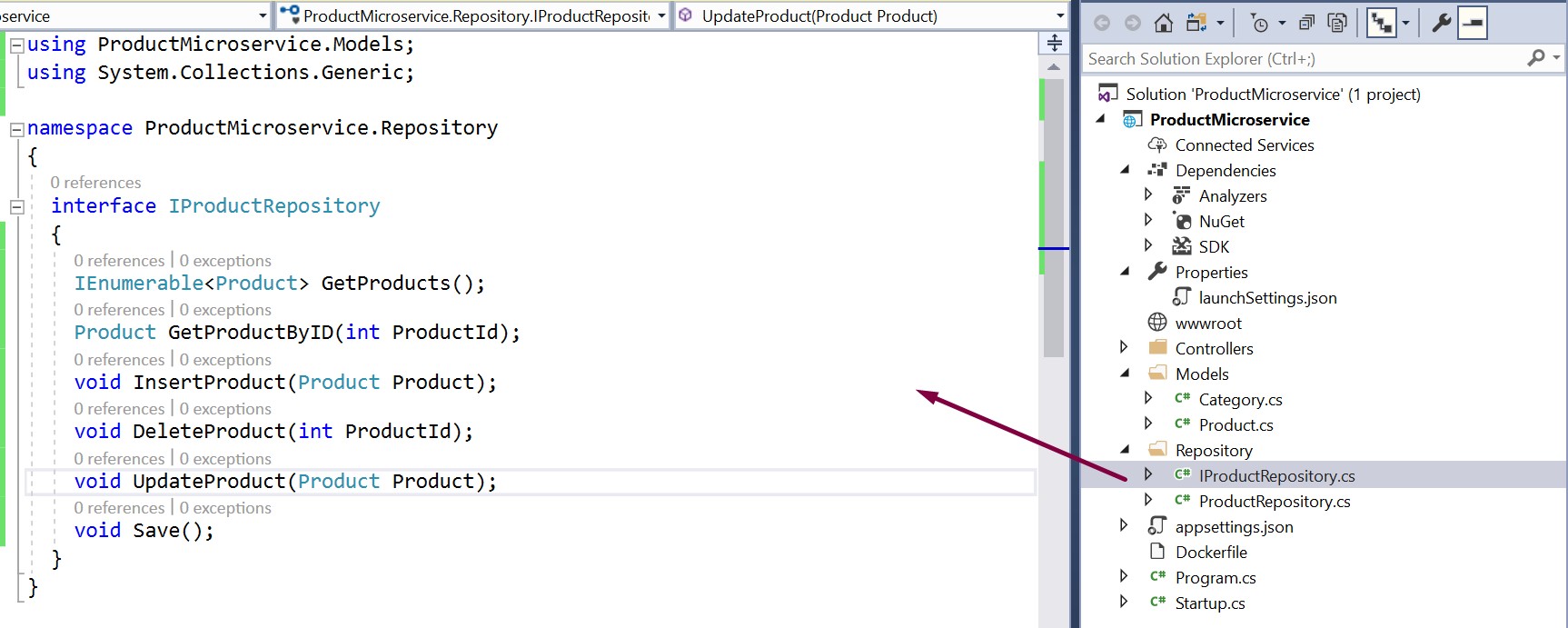
services.AddDbContext<ProductContext>(o => o.UseSqlServer(Configuration.GetConnectionString("ProductDB"))); under ConfigureServices method. Note that in the GetConnectionString method the name of the key of the connection string is passed that was added in appsettings file.



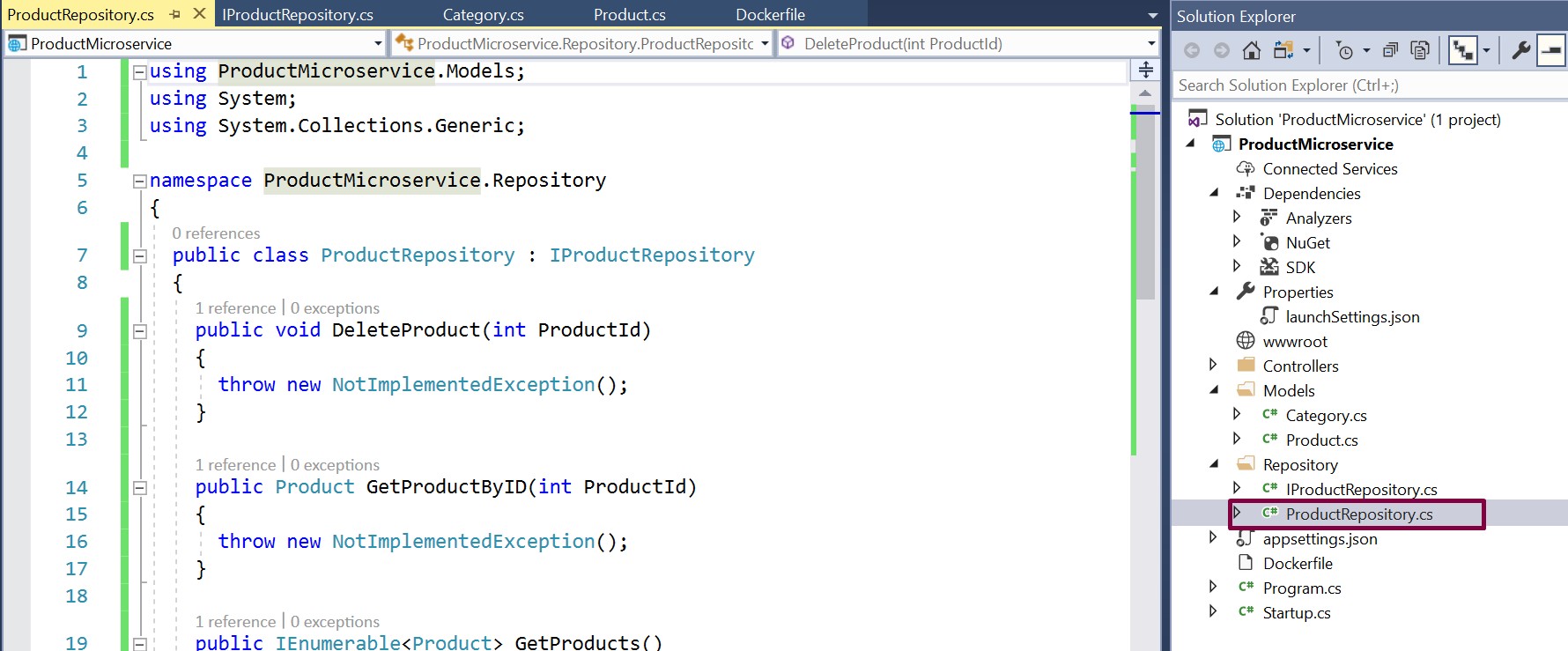
## Adding Repository

Repository works as a micro component of microservice that encapsulates the data access layer and helps in data persistence and testability as well.

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



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



1. Add the implementation for the methods via accessing context methods.

### ProductRepository.cs:

using Microsoft.EntityFrameworkCore; using ProductMicroservice.DBContexts; using ProductMicroservice.Models; using System;

using System.Collections.Generic; using System.Linq;

namespace ProductMicroservice.Repository

{

public class ProductRepository: IProductRepository

{

private readonly ProductContext \_dbContext;

public ProductRepository(ProductContext dbContext)

{

\_dbContext = dbContext;

}

public void DeleteProduct(int productId)

{

var product = \_dbContext.Products.Find(productId);

\_dbContext.Products.Remove(product); Save();

}

public Product GetProductByID(int productId)

{

return \_dbContext.Products.Find(productId);

}

public IEnumerable<Product> GetProducts()

{

return \_dbContext.Products.ToList();

}

public void InsertProduct(Product product)

{

\_dbContext.Add(product); Save(); }

public void Save()

{

\_dbContext.SaveChanges();

}

public void UpdateProduct(Product product)

{

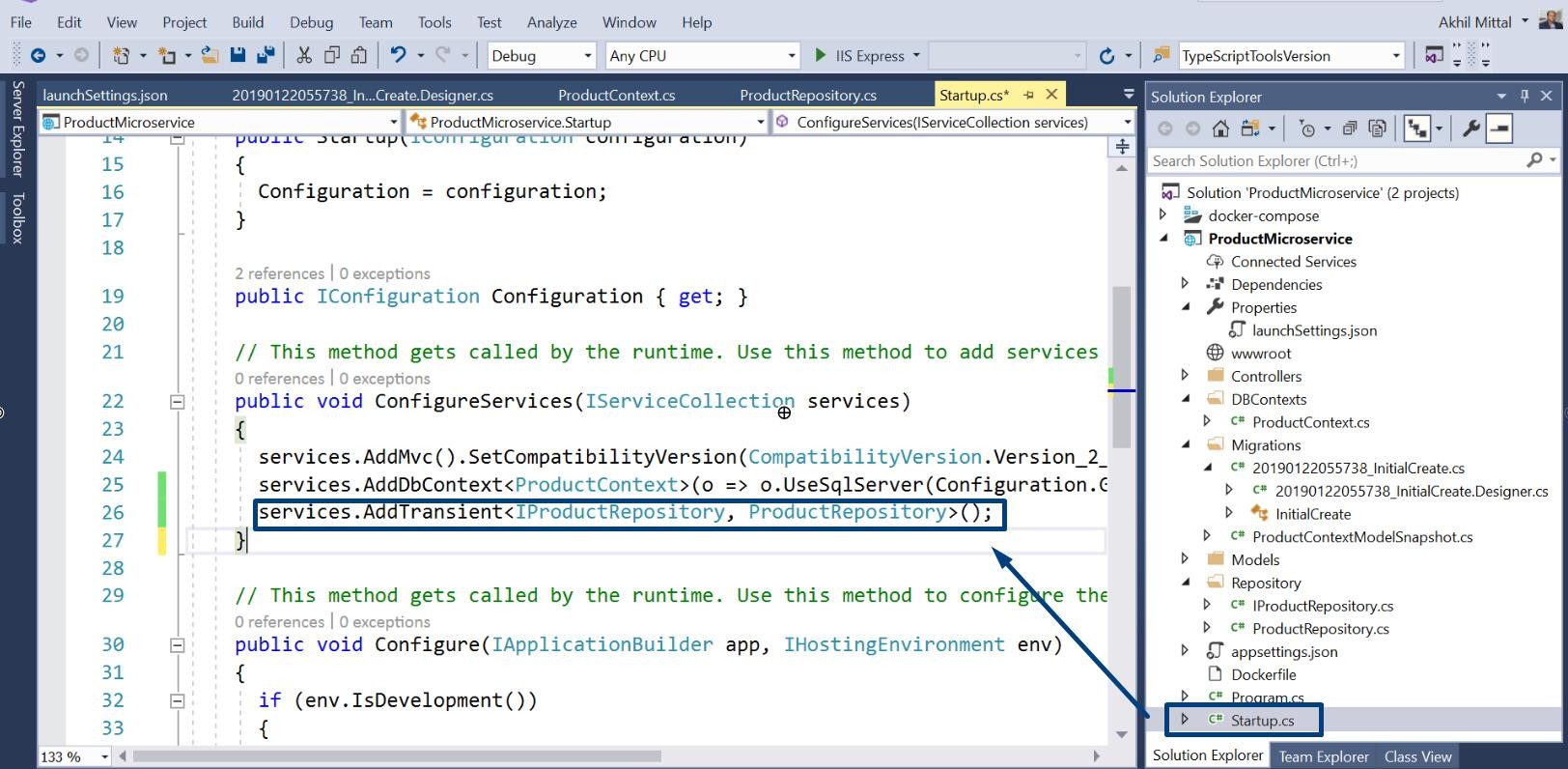
\_dbContext.Entry(product).State = EntityState.Modified; Save();

}

}

}

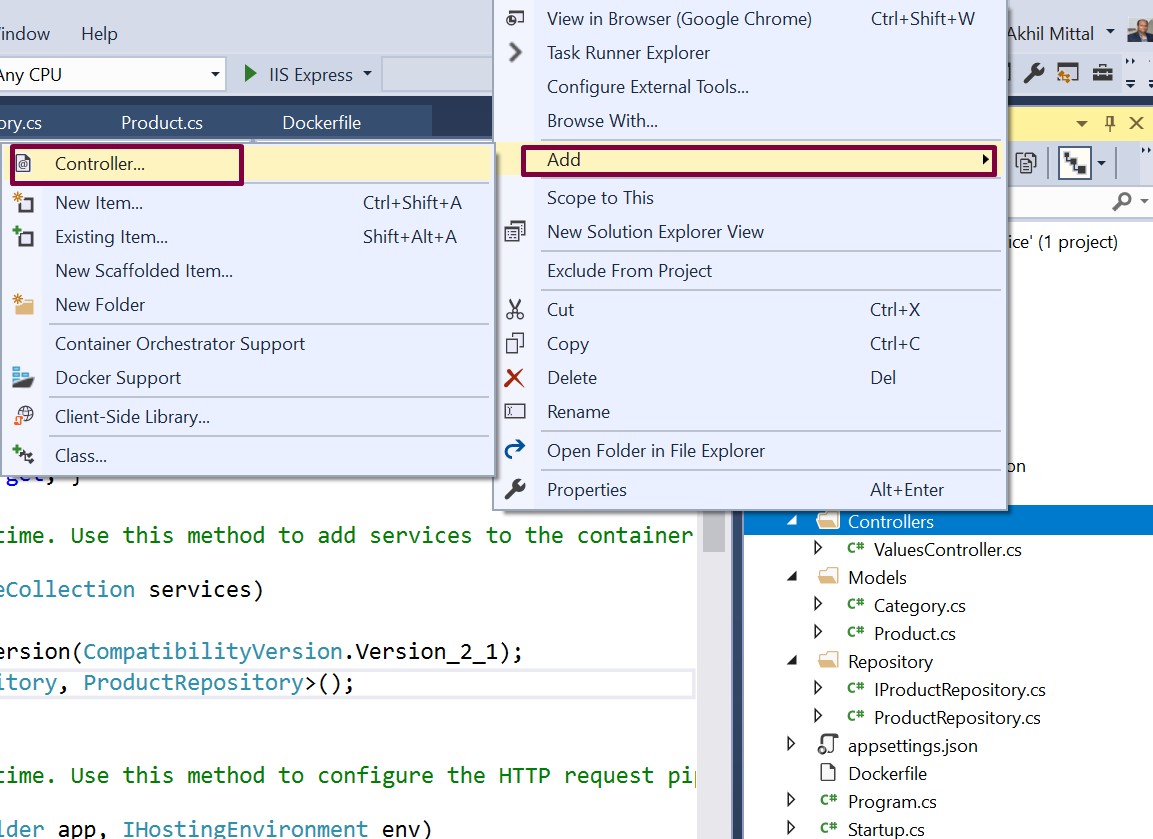
1. Open the Startup class in the project and add the code as services.AddTransient<IProductRepository, ProductRepository>(); inside ConfigureServices method so that the repository’s dependency is resolved at a run time when needed.



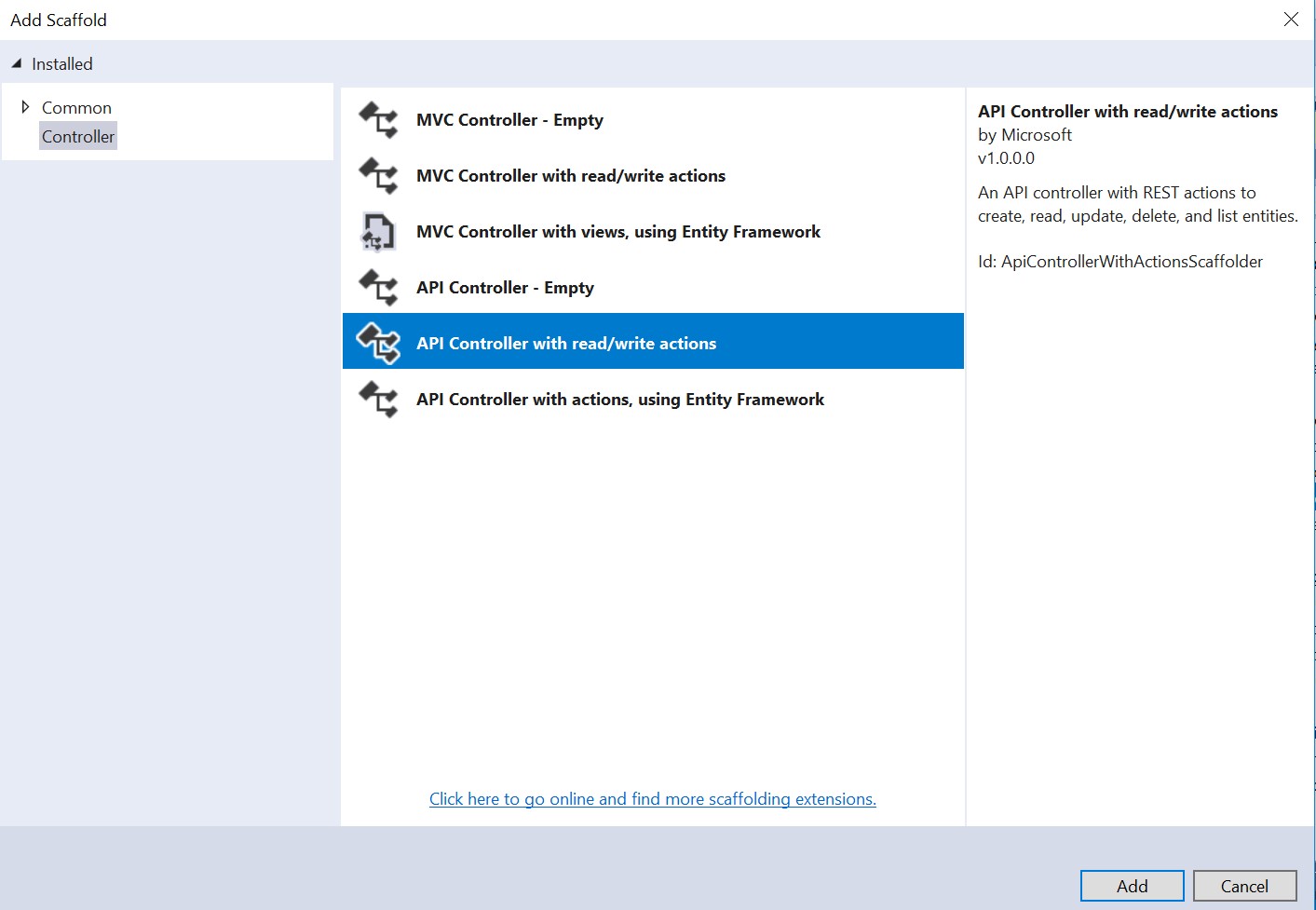
## Adding Controller

The microservice should have an endpoint for which a controller is needed which exposes the HTTP methods to the client as endpoints of the service methods.

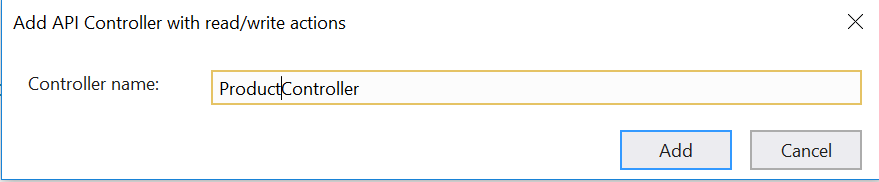
1. Right click on the Controllers folder and add a new Controller as shown below.



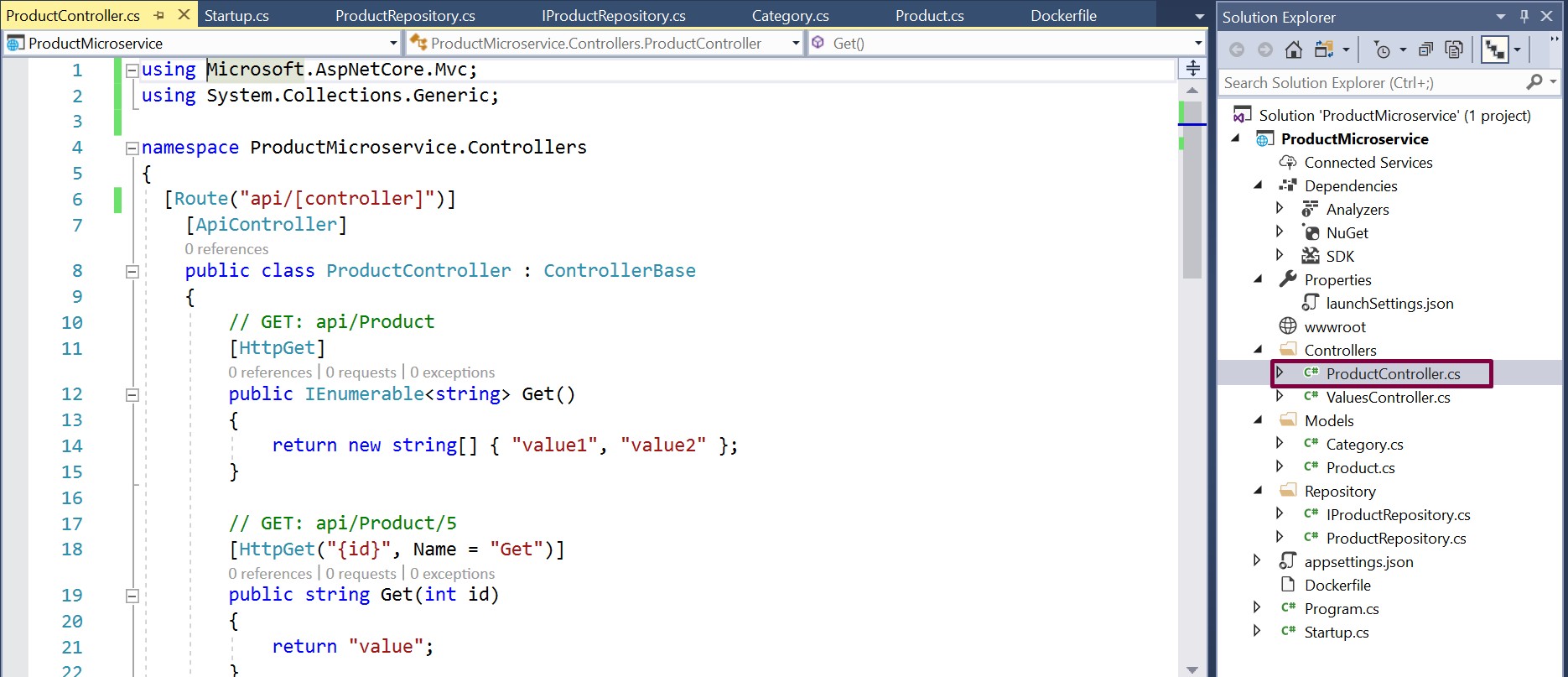
1. Select the option “API Controller with read/write actions” to add the controller.



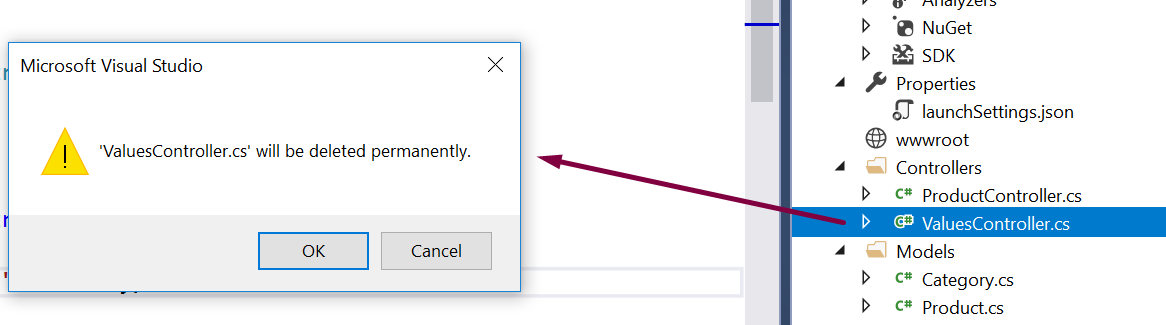
1. Give the name of the controller as ProductController.



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



1. ValuesController can be deleted as it is not needed.



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

### ProductController.cs:

using Microsoft.AspNetCore.Mvc; using ProductMicroservice.Models; using ProductMicroservice.Repository; using System;

using System.Collections.Generic; using System.Transactions;

namespace ProductMicroservice.Controllers

{

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

public class ProductController : ControllerBase

{

private readonly IProductRepository \_productRepository;

public ProductController(IProductRepository productRepository)

{

\_productRepository = productRepository;

}

[HttpGet]

public IActionResult Get()

{

var products = \_productRepository.GetProducts(); return new OkObjectResult(products);

}

[HttpGet("{id}", Name = "Get")] public IActionResult Get(int id)

{

var product = \_productRepository.GetProductByID(id); return new OkObjectResult(product);

}

[HttpPost]

public IActionResult Post([FromBody] Product product)

{

using (var scope = new TransactionScope())

{

\_productRepository.InsertProduct(product); scope.Complete();

return CreatedAtAction(nameof(Get), new { id = product.Id }, product);

}

}

[HttpPut]

public IActionResult Put([FromBody] Product product)

{

if (product != null)

{

using (var scope = new TransactionScope())

{

\_productRepository.UpdateProduct(product); scope.Complete();

return new OkResult();

}

}

return new NoContentResult();

}

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

\_productRepository.DeleteProduct(id); return new OkResult();

}

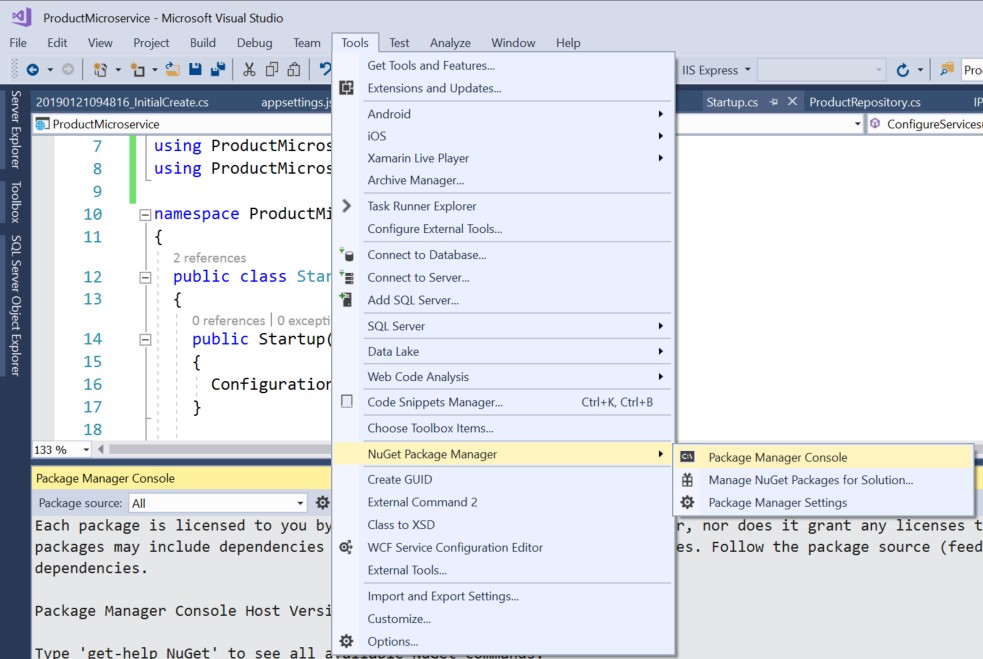
}

}

## Entity Framework Core Migrations

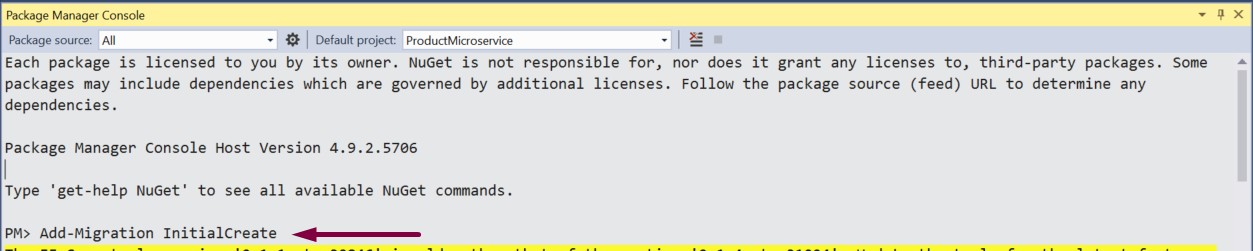
Migrations allow us to provide code to change the database from one version to another.

1. Open Package Manager Console.

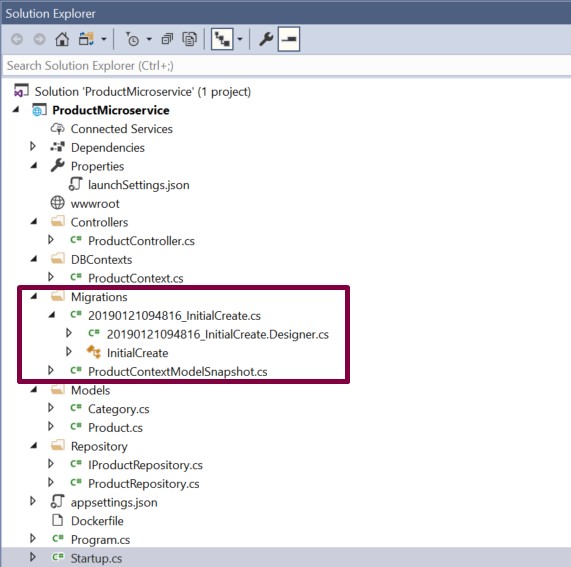


1. To enable the migration, type the command, Add-Migration and give that a meaningful name for

e.g. InitialCreate and press enter.



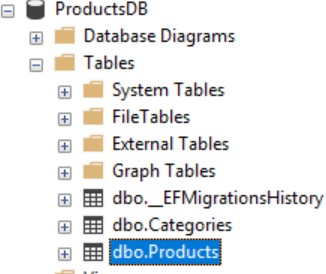
1. Once the command is executed, if we look at our solution now, we see there's a new Migrations folder. And it contains two files. One, a snapshot of our current context model. Feel free to check the files. The files are very much self-explanatory.



1. To ensure that migrations are applied to the database there's another command for that. It's called the **update-database** command. If executed, the migrations will be applied to the current database.



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



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

