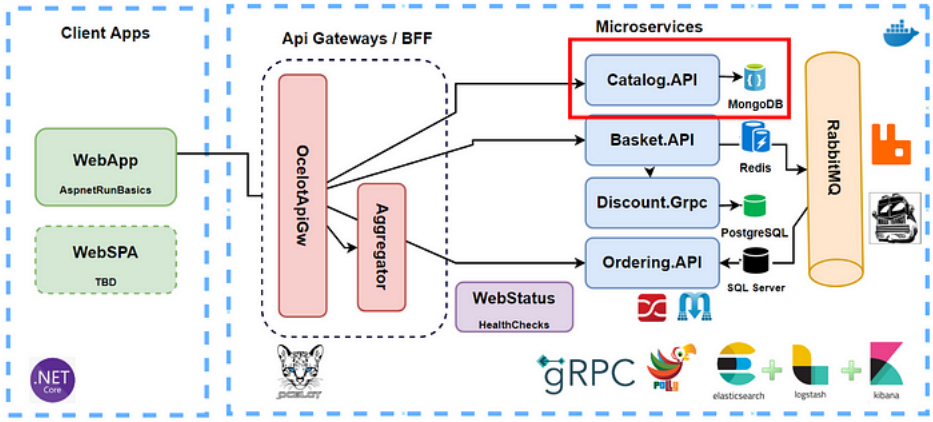
Microservices Using ASP.NET Core, MongoDB and Docker Container

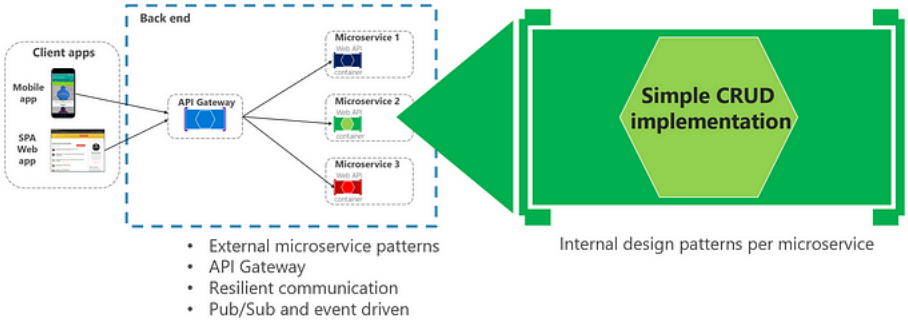
Building Catalog Microservice on .Net platforms which used Asp.Net Web API, Docker, MongoDB and Swagger. Test microservice with using Postman.



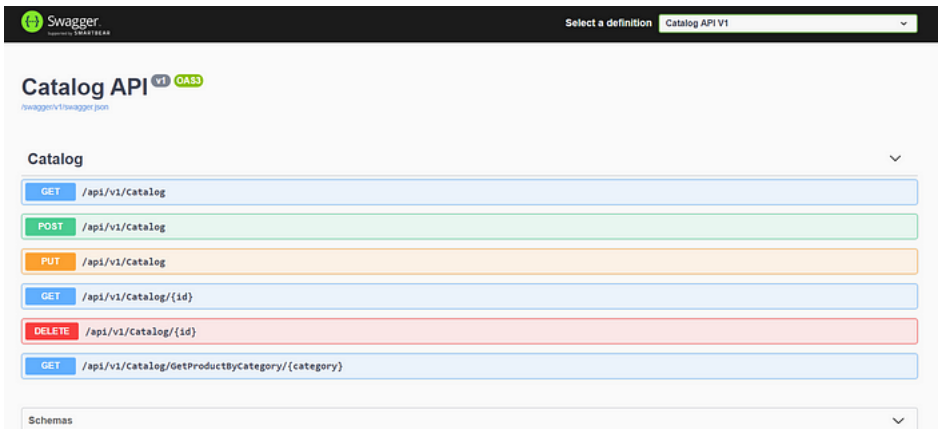
Introduction

In this article we will show **how to perform Catalog Microservices operations** on ASP.NET Core Web application using **MongoDB, Docker Container** and **Swagger**.

By the end of the article, we will have a Web API which implemented CRUD operations over Product and Category documents on MongoDB.



Look at the final swagger application.



Developing Catalog microservice which includes;

- **ASP.NET Core Web API** application
- **REST API** principles, CRUD operations
- **Mongo DB NoSQL** database connection and containerization
- N-Layer implementation with **Repository Pattern**
- **Swagger** Open API implementation
- **Dockerfile** implementation

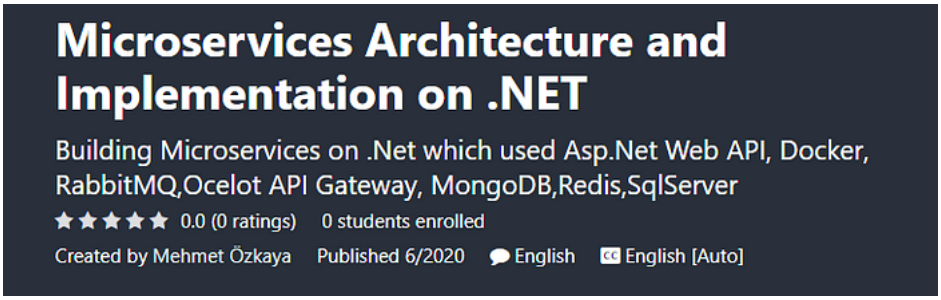
At the end of article, we will have a Web API Microservice which implemented CRUD operations over Product and Category documents on MongoDB.

Background

You can follow the previous article which explains overall microservice architecture of this example. We will focus on Catalog microservice from that overall e-commerce microservice architecture.

Check for the previous article which explained overall microservice architecture of this repository.

Step by Step Development w/ Udemy Course



Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

Source Code

Get the Source Code from AspnetRun Microservices Github — Clone or fork this repository, if you like don't forget the star. If you find or ask anything you can directly open issue on repository.

Prerequisites

- Install the .NET Core 5 or above SDK
- Install Visual Studio 2019 v16.x or above
- Docker Desktop

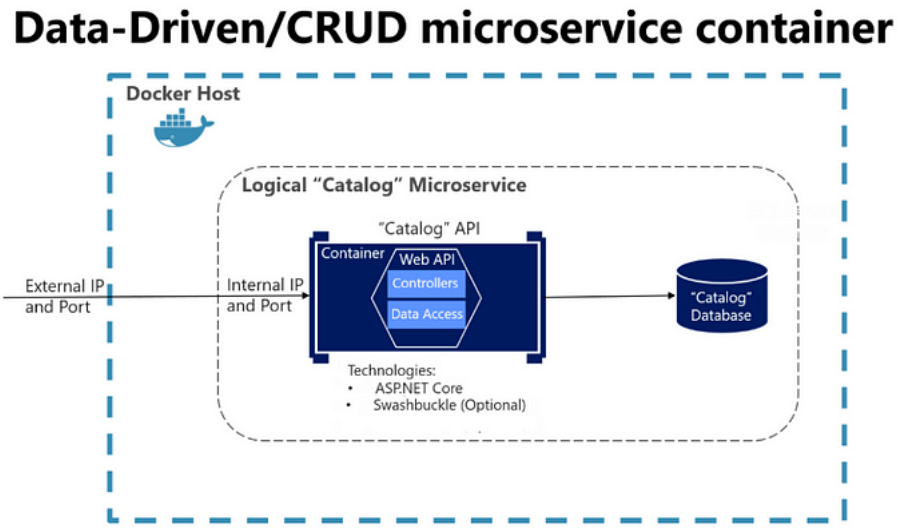
MongoDB

MongoDB introduces us as an open source, **document-oriented database** designed for ease of development and scaling. Every record in **MongoDB** is actually a document. Documents are stored in MongoDB in JSON-like Binary JSON **(BSN)** format. **BSON** documents are objects that contain an ordered list of the elements they store. Each element consists of a domain name and a certain type of value.

It is a **document-based NoSQL** database. It keeps the data structurally in Json format and in documents. Queries can be written in any field or by range. If we compare the structures in MongoDB with the structures in their relational databases, it uses **Collection** instead of Tables and uses **Documents** instead of rows.

Analysis & Design

This project will be the REST APIs which basically perform CRUD operations on Catalog databases.



We should define our Catalog use case analysis. In this part we will create Product — Category entities. Our main use case are Listing Products and Categories, able to search products. Also performed CRUD operations on Product entity.

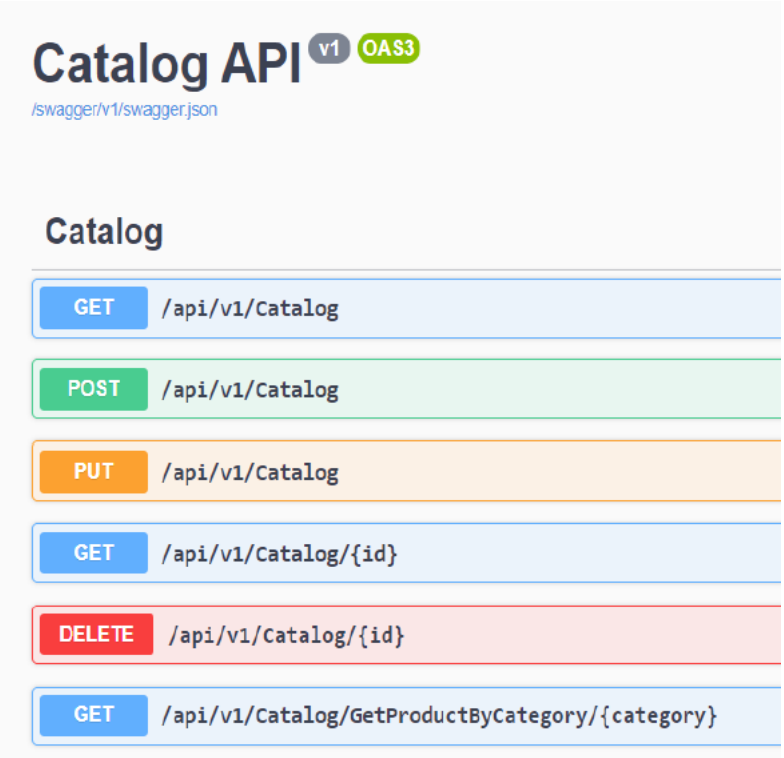
Our main use cases;

- Listing Products and Categories
- Get Product with product Id
- Get Products with category
- Create new Product
- Update Product
- Delete Product

Along with this we should design our APIs according to REST perspective.

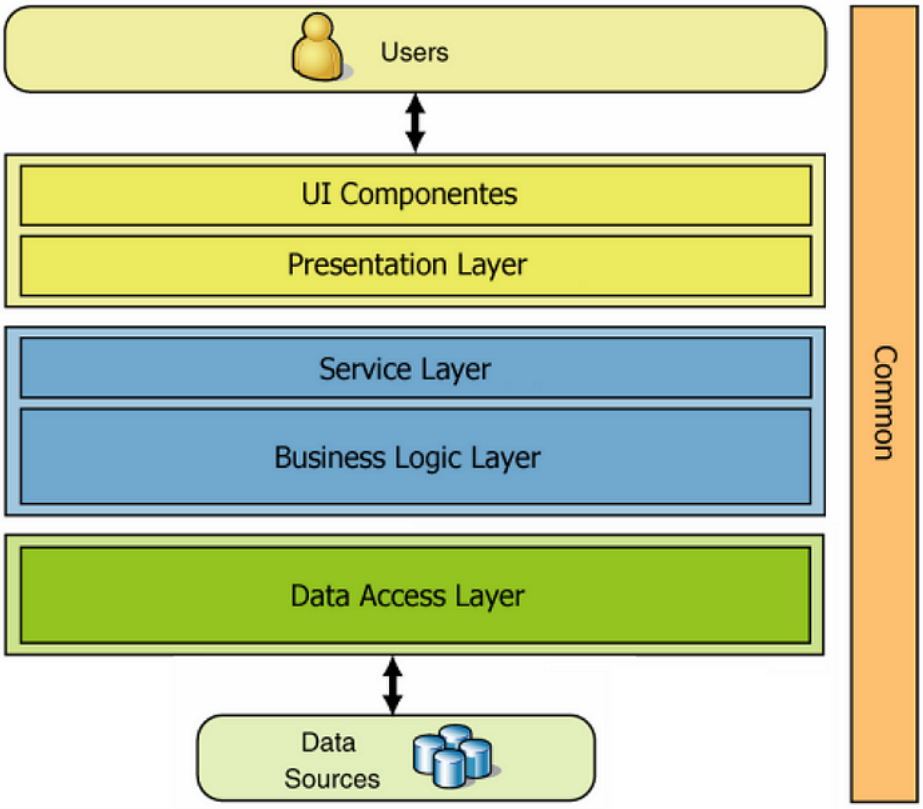
Method	Request URI	Use Case
GET	api/v1/Catalog	Listing Products and Categories
GET	api/v1/Catalog/{id}	Get Product with product Id
GET	api/v1/Catalog/ GetProductByCategory /{category}	Get Products with category
POST	api/v1/Catalog	Create new Product
PUT	api/v1/Catalog	Update Product
DELETE	api/v1/Catalog/{id}	Delete Product

According the analysis, we are going to create swagger output of below;



Architecture of Catalog microservices

We are going to use traditional **N-Layer architecture**. Layered architecture basically consists of 3 layers. These 3 layers are generally the ones that should be in every project. You can create a layered structure with more than these 3 layers, which is called multi-layered architecture.



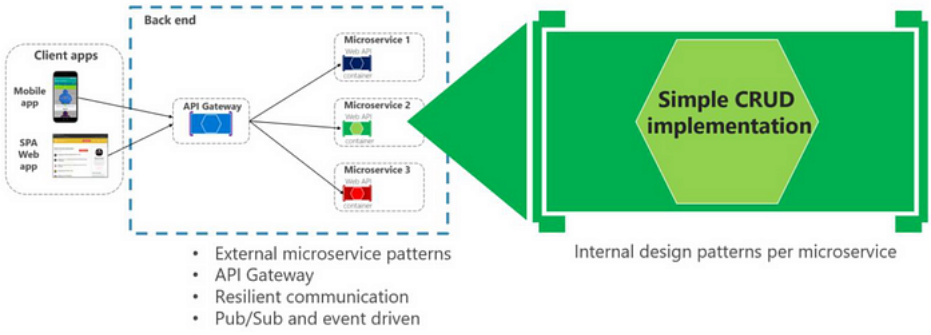
Data Access Layer: Only database operations are performed on this layer.
The task of this layer is to add, delete, update and extract data from the database. There is no other operation in this layer other than these operations.

Business Layer: We implement business logics on this layer. This layer is will process the data taken by Data Access into the project.
We do not use the Data Access layer directly in our applications.
The data coming from the user first goes to the Business layer, from there it is processed and transferred to the Data Access layer.

Presentation Layer: This layer is the layer on which the user interacts.
It could be in Windows form, on the Web, or in a Console application.
The main purpose here is to show the data to the user and to transmit the data from the user to the Business Layer and Data Access.

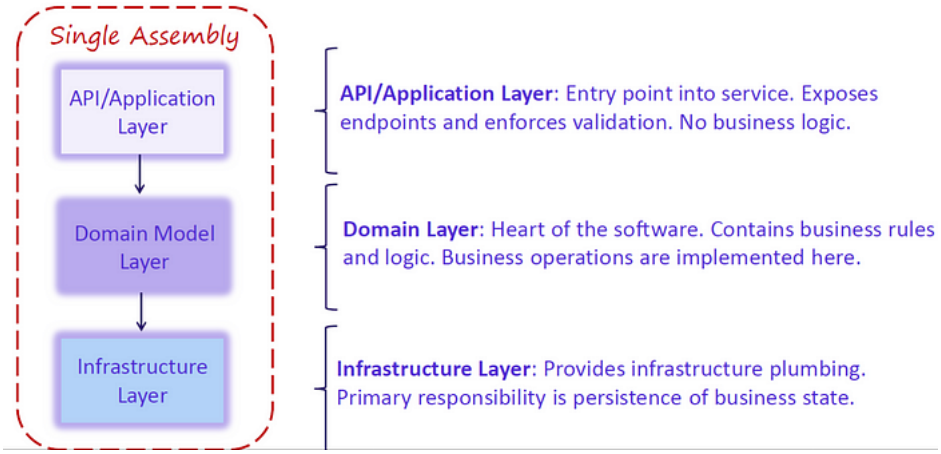
Simple Data Driven CRUD Microservice

Catalog.API microservices will be simple crud implementation on Product data on Mongo databases.



You can apply any internal design pattern per microservices.
Since we have 1 project, so we are going to separate this layers with using folders into the project.
But for the ordering microservices we also separate layers with projects with using clean arch and CQRS implementation.

So we don't need to separate layers in different assemblies.



If we look at the project structure, we are planning to create this layers,

- Domain Layer — Contains business rules and logic.
- Application Layer — Expose endpoints and validations. API layer will be Controller classes.
- Infrastructure Layer — responsible by persistence operations.

Project Folder Structure

- Entities — mongo entity
- Data — mongo data context
- Repositories — mongo repos
- Controllers — api classes

Database Setup with Docker

For Catalog microservices, we are going to use no-sql MongoDB database.

Setup Mongo Database

Here is the docker commands that basically download Mongo DB in your local and use db collections.

In order to download mongo db from docker hub use below commands;

docker pull mongo

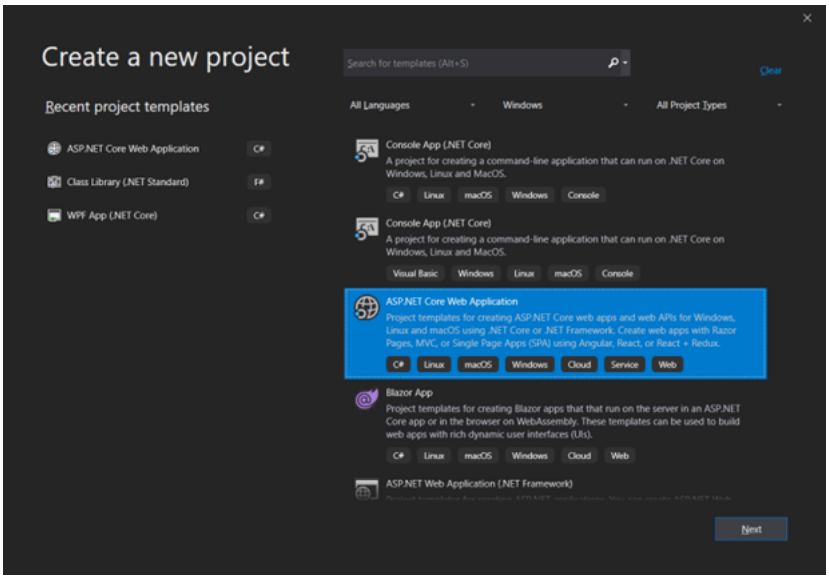
To run database on your docker environment use below command. It will expose 27017 port in your local environment.

docker run -d -p 27017:27017 — name aspnetrun-mongo mongo

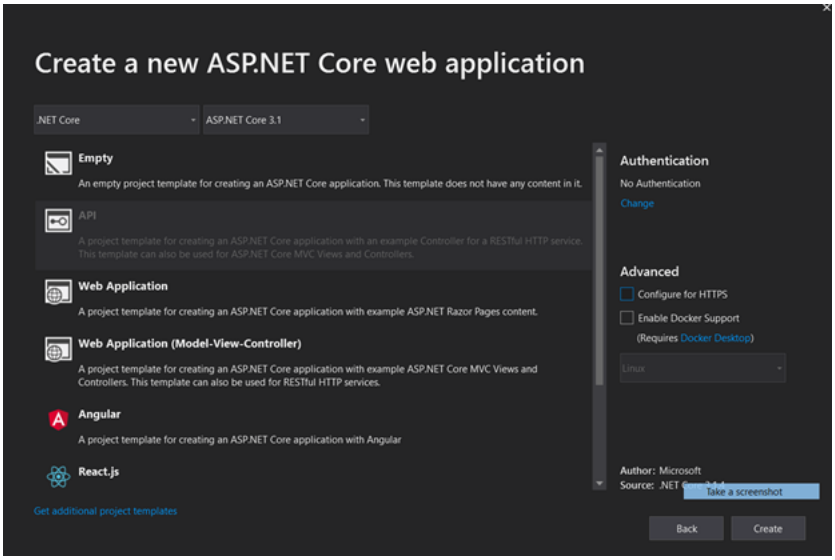
Starting Our Project

Create new web application with visual studio.

First, open **File -> New -> Project**. Select ASP.NET Core Web Application, give your project a name and select OK.



In the next window, select .Net Core and ASP.Net Core latest version and select **Web API** and then uncheck "**Configure for HTTPS**" selection and click OK. This is the default Web API template selected. Unchecked for https because we don't use https for our api's now.



Add New Web API project under below location and name;

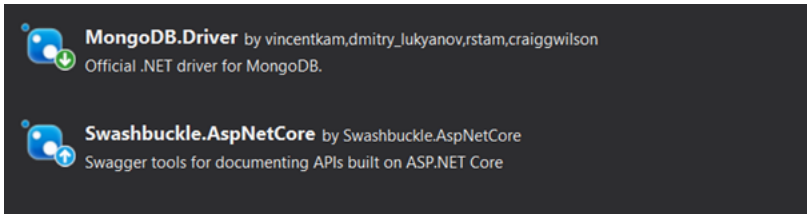
src/catalog/Catalog.API

Library & Frameworks

For Catalog microservices, we have to libraries in our Nuget Packages,

1- Mongo.DB.Driver — To connect mongo database

2- Swashbuckle.AspNetCore — To generate swagger index page



Create Entities

Create **Entities** folder into your project. This will be the MongoDB collections of your project. In this section, we will use the MongoDB Client when connecting the database. That's why we write the entity classes at first.

Add New Class -> Product

```
public class Product
{
[BsonId]
[BsonRepresentation(BsonType.ObjectId)]
public string Id { get; set; }[BsonElement("Name")]
public string Name { get; set; }public string Category { get; set; }public string Summary { get; set; }public string Description { get; set; }public s
}
```

There is **Bson** annotations which provide to mark properties for the database mapping. I.e. BsonId is primary key for Product collection.

Create Data Layer

Create **Data** folder into your project. This will be the MongoDB collections of your project.

In order to manage these entities, we should create a data structure. To work with a database, we will use this class with the MongoDb Client. In order to wrapper this classes we will create that provide data access over the **Context** classes.

To store these entities, we start with **ICatalogContext** interface.

Create **ICatalogContext** class.

```
public interface ICatalogContext
{
IMongoCollection<Product> Products { get; }
}
```

Basically, we expect from our db context object is **Products** collections.

Continue with implementation and create **CatalogContext** class.

```
public class CatalogContext : ICatalogContext
 {
 public CatalogContext(ICatalogDatabaseSettings settings)
 {
 var client = new MongoClient(settings.ConnectionString);
 var database = client.GetDatabase(settings.DatabaseName);Products = database.GetCollection<Product>(settings.CollectionName);
 CatalogContextSeed.SeedData(Products);
 }public IMongoCollection<Product> Products { get; }
 }
```

In this class, constructor initiate **MongoDB connection** with using **MongoClient** library. And load the **Products** collection.

The code getting connection string from settings. In Asp.Net Core this configuration stored **appsettings.json** file;

```
"CatalogDatabaseSettings": {
"ConnectionString": "mongodb://localhost:27017",
"DatabaseName": "CatalogDb",
"CollectionName": "Products"
},
```

At this point, you can put your configurations according to your dockerize mongodb. Default port was **27017** that's why we use same port.

Register DataContext into ASP.NET Dependency Injection

You should register this repository classes into **ASP.NET Built-in Dependency Injection** engine. That means we should **recognize these classes** into asp.net core in order to use from frontend side in web application.

Open Startup.cs -> Go To Method ConfigureAspnetRunServices -> put your dependencies;

Mongo DB context object should register in DI when starting the application. Ensure that **DbContext** object into **ConfigureServices** method is configured properly.

```
public void ConfigureServices(IServiceCollection services)
{
#region Project Dependencies
services.AddScoped<ICatalogContext, CatalogContext>();
#endregion
}
```

Create Business Layer

For the Business Logic Layer, we should create a new folder which name could be the Service — Application — Manager — Repository in order to manage business operations with using data access layer objects.

For the Business Logic Layer, we are using **Repository** folder in order to manage business operations with using data access layer objects.

According to our main use cases we will create **interface** and **implementation** classes in our business layer.

- Listing Products and Categories
- Get Product with product Id
- Get Products with category
- Create new Product
- Update Product
- Delete Product

So, let's create/open a **Repository folder** and create a new interface to **IProductRepository** class in order to manage Product related requests.

```
public interface IProductRepository
    {
        Task<IEnumerable<Product>> GetProducts();
        Task<Product> GetProduct(string id);
        Task<IEnumerable<Product>> GetProductByName(string name);
        Task<IEnumerable<Product>> GetProductByCategory(string categoryName);Task CreateProduct(Product product);
        Task<bool> UpdateProduct(Product product);
        Task<bool> DeleteProduct(string id);
    }
```

Let's implement these interfaces with using Data layer objects. In our case Data layer represents from Mongo Client library so we should use **DBContext** object. In order to use **CatalogContext** object which represent us DB Layer, the constructor should use **dependency injection** to **inject** the database context(**CatalogContext**) into the **ProductRepository** class.

```
public class ProductRepository : IProductRepository
    {
        private readonly ICatalogContext _context;public ProductRepository(ICatalogContext context)
        {
            _context = context ?? throw new ArgumentNullException(nameof(context));
        }public async Task<IEnumerable<Product>> GetProducts()
        {
            return await _context
                            .Products
                            .Find(p => true)
                            .ToListAsync();
        }public async Task<Product> GetProduct(string id)
        {
            return await _context
                           .Products
                           .Find(p => p.Id == id)
                           .FirstOrDefaultAsync();
        }public async Task<IEnumerable<Product>> GetProductByName(string name)
        {
            FilterDefinition<Product> filter = Builders<Product>.Filter.ElemMatch(p => p.Name, name);return await _context
                            .Products
                            .Find(filter)
                            .ToListAsync();
        }public async Task<IEnumerable<Product>> GetProductByCategory(string categoryName)
        {
            FilterDefinition<Product> filter = Builders<Product>.Filter.Eq(p => p.Category, categoryName);return await _context
                            .Products
                            .Find(filter)
                            .ToListAsync();
        }public async Task CreateProduct(Product product)
        {
            await _context.Products.InsertOneAsync(product);
        }public async Task<bool> UpdateProduct(Product product)
        {
            var updateResult = await _context
                                        .Products
                                        .ReplaceOneAsync(filter: g => g.Id == product.Id, replacement: product);return updateResult.IsAcknowledged
                    && updateResult.ModifiedCount > 0;
        }public async Task<bool> DeleteProduct(string id)
        {
            FilterDefinition<Product> filter = Builders<Product>.Filter.Eq(p => p.Id, id);DeleteResult deleteResult = await _context
                                               .Products
                                               .DeleteOneAsync(filter);return deleteResult.IsAcknowledged
                && deleteResult.DeletedCount > 0;
        }
    }
```

*Basically, In **ProductRepository class**, we managed all business-related actions with using **CatalogContext** object. You can put all business logics into these functions in order to manage one place.*

Don't forget to add below references into your **repository implementations** class;

using MongoDB.Driver;

By this library, we use Mongo operations over the **Products** collections. (**Find, InsertOne, ReplaceOne, DeleteOne** methods)

Register Repository into ASP.NET Dependency Injection

You should register this repository classes into **ASP.NET Built-in Dependency Injection** engine. That means we should **recognize these classes** into asp.net core in order to use from frontend side in web application.

Open Startup.cs -> Go To Method ConfigureAspnetRunServices -> put your dependencies;

```
public void ConfigureServices(IServiceCollection services)
{
…
#region Project Dependencies
services.AddScoped<ICatalogContext, CatalogContext>();
services.AddScoped<IProductRepository, ProductRepository>();
#endregion
}
```

Create Presentation Layer

Since created a **Web API** template for ASP.NET Core project, the presentation layer will be **Controller** classes which produce **API layer**.

Locate the **Controller folder** and create **CatalogController** class.

```
[ApiController]
    [Route("api/v1/[controller]")]
    public class CatalogController : ControllerBase
    {
        private readonly IProductRepository _repository;
        private readonly ILogger<CatalogController> _logger;public CatalogController(IProductRepository repository, ILogger<CatalogController> logger
        {
            _repository = repository ?? throw new ArgumentNullException(nameof(repository));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        }[HttpGet]
        [ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)]
        public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
        {
            var products = await _repository.GetProducts();
            return Ok(products);
        }[HttpGet("{id:length(24)}", Name = "GetProduct")]
        [ProducesResponseType((int)HttpStatusCode.NotFound)]
        [ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]
        public async Task<ActionResult<Product>> GetProductById(string id)
        {
            var product = await _repository.GetProduct(id);if (product == null)
            {
                _logger.LogError($"Product with id: {id}, not found.");
                return NotFound();
            }return Ok(product);
        }[Route("[action]/{category}", Name = "GetProductByCategory")]
        [HttpGet]
        [ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)]
        public async Task<ActionResult<IEnumerable<Product>>> GetProductByCategory(string category)
        {
            var products = await _repository.GetProductByCategory(category);
            return Ok(products);
        }[HttpPost]
        [ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]
        public async Task<ActionResult<Product>> CreateProduct([FromBody] Product product)
        {
            await _repository.Create(product);return CreatedAtRoute("GetProduct", new { id = product.Id }, product);
        }[HttpPut]
        [ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]
        public async Task<IActionResult> UpdateProduct([FromBody] Product product)
        {
            return Ok(await _repository.Update(product));
        }[HttpDelete("{id:length(24)}", Name = "DeleteProduct")]
        [ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]
        public async Task<IActionResult> DeleteProductById(string id)
        {
            return Ok(await _repository.Delete(id));
        }
    }
```

In this class we are creating API with data through business layer. Before we should pass **IProductRepository** class into **constructor** of class in order to use repository related functions inside of the API calls.

API Routes in Controller Classes

In Controller class can manage to provide below **routes** as intended methods in **CatalogController.cs**.

Along with this we developed our APIs according below list.

Method	Request URI	CatalogController.cs
GET	api/v1/Catalog	`GetProducts()`
GET	api/v1/Catalog/{id}	`GetProduct(string id)`
GET	api/v1/Catalog/GetProductByCategory/{category}	`GetProductByCategory(string category)`
POST	api/v1/Catalog	`CreateProduct([FromBody] Product product)`
PUT	api/v1/Catalog	`UpdateProduct([FromBody] Product value)`
DELETE	api/v1/Catalog/{id}	`DeleteProductById(string id)`

Swagger Implementation

Swagger is dynamic used by the software world is a widely used dynamic document creation tool that is widely accepted. Its implementation within .Net Core projects is quite simple.

Implementation of Swagger

1- Let's download and install the **Swashbuckle.AspNetCore package** to the web api project via nuget.

2- Let's add the swagger as a service in the **ConfigureServices** method in the Startup.cs class of our project.

```
public void ConfigureServices(IServiceCollection services)
 {
 …
 #region Swagger Dependenciesservices.AddSwaggerGen(c =>
 {
 c.SwaggerDoc("v1", new OpenApiInfo { Title = "Catalog API", Version = "v1" });
 });#endregion
}
```

3- After that we will use this added service in the Configure method in Startup.cs.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
 {
…app.UseSwagger();
 app.UseSwaggerUI(c =>
 {
 c.SwaggerEndpoint("/swagger/v1/swagger.json", "Catalog API V1");
 });
 }
```

Run Application

Now the **Catalog microservice** Web API application ready to run.

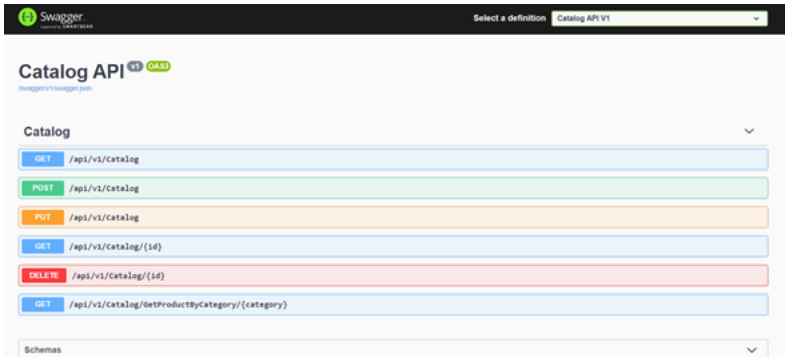
Before running the application, **configure the debug profile**;

Right Click the project File and Select to Debug section.

Change Launch browser to **swagger**

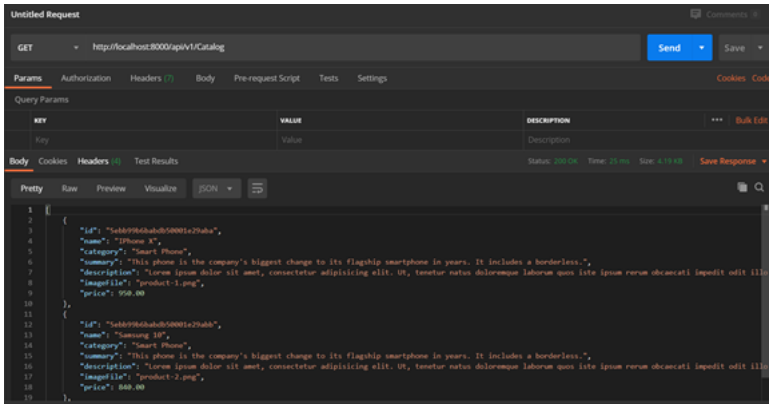
Change the App URL to **http://localhost:5000**

Hit **F5** on **Catalog.API** project.



Exposed the Product APIs in our Catalog Microservices, you can **test** it over the **Swagger GUI**.

You can also test it over the **Postman** as below way.



Above image is example of test Get Catalog method.

Run Application on Docker with Database

Since here we developed ASP.NET Core Web API project for Catalog microservices. Now it's time to make **docker** the **Catalog API** project with our **MongoDB**.

Add Docker Compose and Dockerfile

Normally you can add only Dockerfile for make dokerize the Web API application but we will integrate our API project with MongoDB docker image, so we should create docker-compose file with Dockerfile of API project.

Right Click to Project -> Add -> ..Container Orchestration Support

Continue with default values.

Dockerfile and **docker-compose** files are created.

Docker-compose.yml is a command-line file used during development and testing, where necessary definitions are made for multi-container running applications.

Docker-compose.yml

```
version: '3.4'services:
 catalogdb:
 image: mongocatalog.api:
 image: ${DOCKER_REGISTRY-}catalogapi
 build:
 context: .
 Dockerfile: src/Catalog/Catalog.API/Dockerfile
```

Docker-compose.override.yml

```
version: '3.4'services:
 catalogdb:
 container_name: catalogdb
 restart: always
 volumes:
 - ${WEBAPP_STORAGE_HOME}/site:/data/db
 ports:
 - "27017:27017"catalog.api:
 container_name: catalogapi
 environment:
 - ASPNETCORE_ENVIRONMENT=Development
 - "CatalogDatabaseSettings:ConnectionString=mongodb://catalogdb:27017"
 depends_on:
 - catalogdb
 volumes:
 - ${HOME}/.microsoft/usersecrets/:/root/.microsoft/usersecrets
 - ${HOME}/.aspnet/https:/root/.aspnet/https/
```

```
ports:
– “8000:80”
```

Basically in **docker-compose.yml file**, created 2 image 1 is for mongoDb which name is **catalogdb**, 2 is web api project which name is **catalog.api**.

After that we **configure** these images into **docker-compose.override.yml file.**

In override file said that;

- **Catalogdb** which is mongo database will be open **27017** port.
- **Catalog.api** which is our developed web API project depend on **catalogdb** and open port on **8000** and we override to connection string with **catalogdb**.

Run below command on top of project folder which include **docker-compose.yml** files.

docker-compose -f docker-compose.yml -f docker-compose.override.yml up –build

That’s it!

You can check microservices as below urls :

Catalog API -> http://localhost:8000/swagger/index.html

SEE DATA with Test over Swagger

/api/v1/Catalog