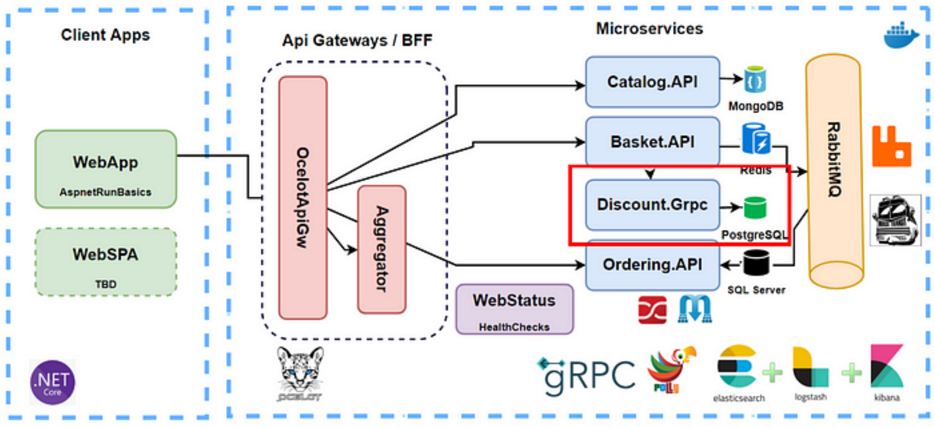
Microservices Using ASP.NET, PostgreSQL, Dapper Micro-ORM and Docker Container

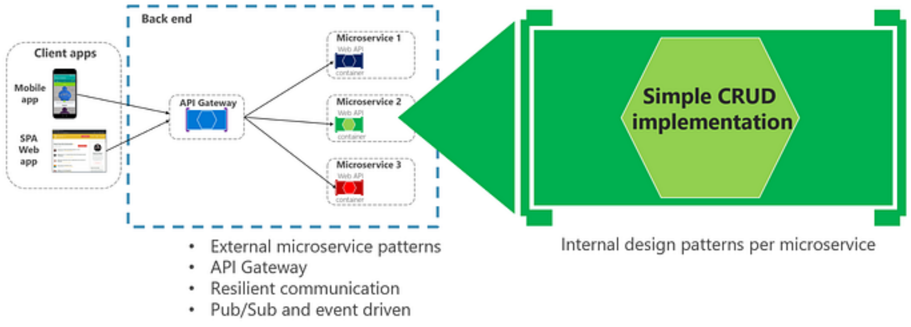
Building Discount Microservice on .Net platforms which used Asp.Net Web API, Docker, PostgreSQL, Dapper Micro-ORM and Swagger. Test microservice with using Postman.



Introduction

In this article we will show **how to perform Discount Microservices operations** on ASP.NET Core Web application using **PostgreSQL, Dapper micro-orm, Docker Container** and **Swagger**.

By the end of the article, we will have a Web API which implemented CRUD operations over **Coupon** objects. These objects will be store in **PostgreSQL database** and retrieved data with using Dapper micro-orm tool.



Developing Discount microservice which includes;

- **ASP.NET Web API** application
- **REST API** principles, **CRUD** operations
- **PostgreSQL database** connection and containerization
- **Repository Pattern** implementation
- Using **Dapper for micro-orm** implementation to simplify data access and ensure high performance
- **PostgreSQL** database connection and **containerization**

We will Analysis and Architecting of Discount Microservices, applying N-Layer architecture. Containerize Discount Microservices with PostgreSQL database using **Docker Compose.**

In the upcoming sections :

- ASP.NET Grpc Server application
- Build a Highly Performant inter-service **gRPC Communication** with Discount Microservice
- Exposing Grpc Services with creating Protobuf messages

So in this section, we are going to Develop **Discount.API Microservices** with **PostgreSQL**.

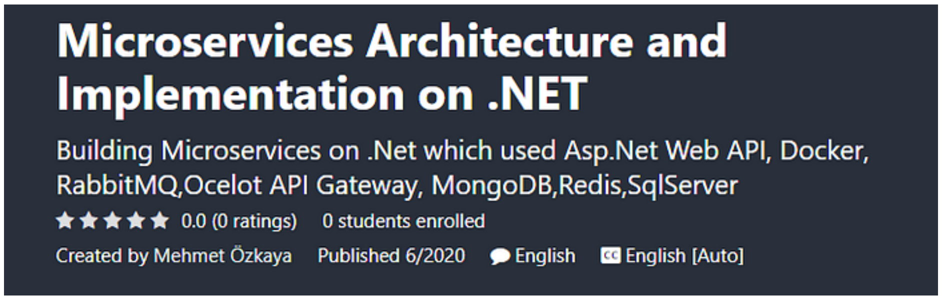
Background

You can follow the previous article which explains overall microservice architecture of this example.

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

We will focus on Discount microservice from that overall e-commerce microservice architecture.

Step by Step Development w/ Udemy Course


Microservices Architecture and Implementation on .NET

Building Microservices on .Net which used Asp.Net Web API, Docker, RabbitMQ,Ocelot API Gateway, MongoDB,Redis,SqlServer

★★★★★ 0.0 (0 ratings) 0 students enrolled

Created by Mehmet Özkaya Published 6/2020 English English [Auto]

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

PostgreSQL in Discount Microservices

PostgreSQL is an open source and completely free object relational database system with powerful features and advantages. Taking advantage of the security, storability and scalability features of the SQL language, PostgreSQL is also used as a database manager in many areas.

PostgreSQL is one of the most accepted database management systems in the industry today. Because it offers users the advantages of successful data architecture, **data accuracy**, powerful feature set, and open source. PostgreSQL is supported by many major operating systems such as UNIX, Linux, MacOS and Windows.

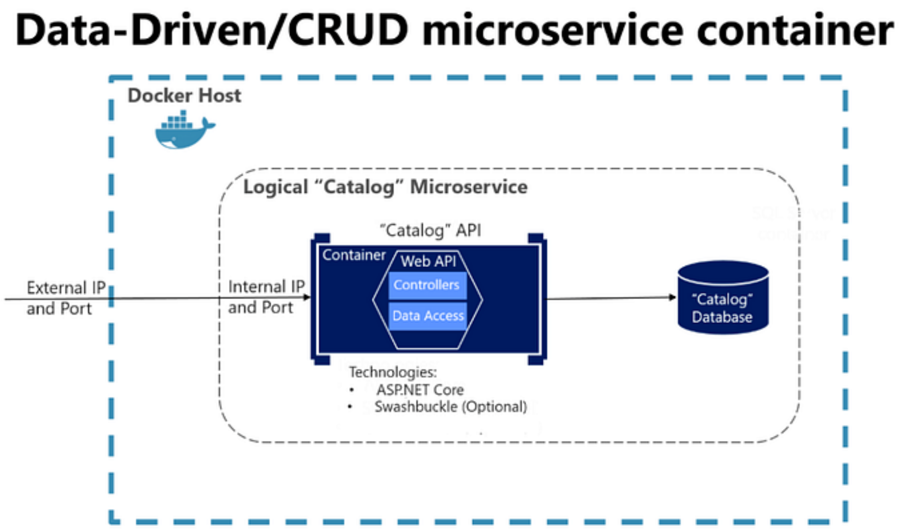
In terms of performance, PostgreSQL has been found to be more successful compared to other commercial or open source databases.
In the face of some database systems, it is fast in some areas, but slow in others.

Compared to PostgreSQL, MySQL and databases in the same class, it is slower in **INSERT / UPDATE transactions** because it works transaction-based.
In some cases, PostgreSQL and MySQL have significant advantages in terms of features, reliability and flexibility. Aside from the highlights of PostgreSQL, this service is offered completely free of charge by open source developers.

For Discount microservices, we are going to store discount coupon data information into PostgreSQL database in a Coupon table.

Analysis & Design

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



We should define our Discount use case analysis. In this part we will create Coupon entities. Our main use case are Get Coupon by ProductName, able to add new item to Discount. Also performed CRUD operations on Coupom entity.

Our main use cases;

- Get Coupon with productname
- Update Coupon
- Delete Coupon
- Checkout Coupon

Architecture of Discount 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.

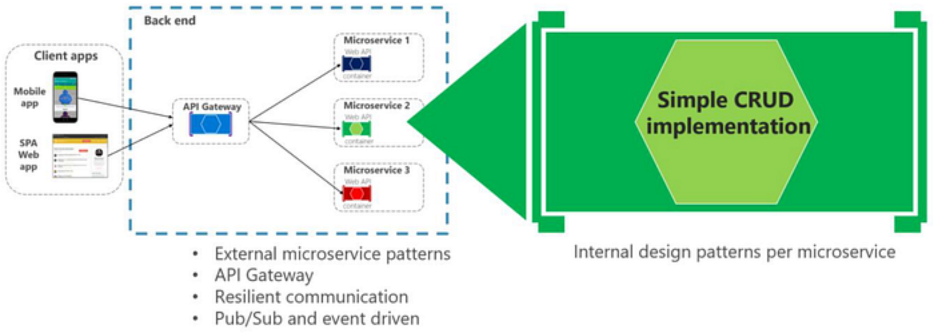
Data Access Layer: Only database operations are performed on this layer.
The task of this layer is to add, delete, update and extract data from the database. There is no other operation in this layer other than these operations.

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.

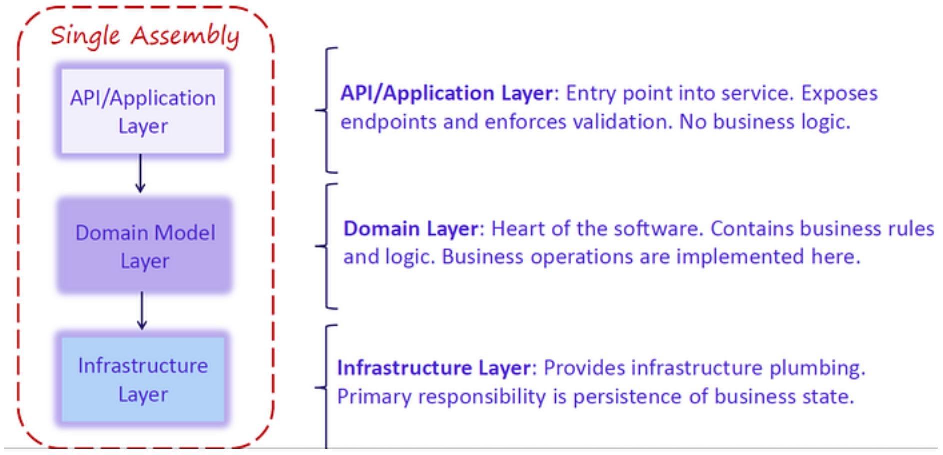
Simple Data Driven CRUD Microservice

Discount.API microservices will be simple crud implementation on Coupom data on PostgreSQL databases.


Back end
Client apps
Mobile app
SPA Web app
API Gateway
Microservice 1
Microservice 2
Microservice 3
Simple CRUD implementation
Internal design patterns per microservice
• External microservice patterns
• API Gateway
• Resilient communication
• Pub/Sub and event driven

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.


Single Assembly
API/Application Layer
Domain Model Layer
Infrastructure Layer
API/Application Layer: Entry point into service. Exposes endpoints and enforces validation. No business logic.
Domain Layer: Heart of the software. Contains business rules and logic. Business operations are implemented here.
Infrastructure Layer: Provides infrastructure plumbing. Primary responsibility is persistence of business state.

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 — PostgreSQL entity
- Data — PostgreSQL data context
- Repositories — PostgreSQL repos
- Controllers — api classes

Setup PostgreSQL Docker Database for Discount.API Microservices

We are going to Setup PostgreSQL Docker Database for Discount.API Microservices.

First, We should go to **DockerHub** and find Postgres official image.

Postgres DockerHub;
https://hub.docker.com/_/postgres

This time we are going to write directly on docker-compose file.

docker-compose.yml
```
version: ‘3.4’services:
 catalogdb:
 image: mongobasketdb:
 image: redis:alpinediscountdb: — — ADDED
 image: postgres
…volumes:
 portainer_data:
 postgres_data: — — ADDED
```

—
Check environment variables - https://hub.docker.com/_/postgres
pgdata username etc..
—

docker-compose.override.yml
```
discountdb:
 container_name: discountdb
 environment:
 — POSTGRES_USER=admin
 — POSTGRES_PASSWORD=admin1234
 — POSTGRES_DB=DiscountDb
 restart: always
 ports:
 — “5432:5432”
 volumes:
 — postgres_data:/var/lib/postgresql/data/
```

We have added postgresql database in our dc file. We have not started yet, because we also need to manage portal of postresql.

Setup pgAdmin Management Portal for PostgreSQL Database for Discount.API Microservices

We are going to Setup pgAdmin Manage Portal for PostgreSQL Database for Discount.API Microservices.

pgAdmin is one of the popular and feature rich Open Source administration and development platform for PostgreSQL.
We will use pgAdmin for managing PostgreSQL Discount database creation and add same records into that table for Discount microservices.

We should go to DockerHub and find pgAdmin image.
— now its time to add manage portal of postresql whic is pgadmin.

Check env variables
https://www.pgadmin.org/docs/pgadmin4/latest/container_deployment.html

Run a simple container over port 80, setting some configuration options:
```
docker pull dpage/pgadmin4
 docker run -p 80:80 \
 -e 'PGADMIN_DEFAULT_EMAIL=user@domain.com' \
 -e 'PGADMIN_DEFAULT_PASSWORD=SuperSecret' \
 -e 'PGADMIN_CONFIG_ENHANCED_COOKIE_PROTECTION=True' \
 -e 'PGADMIN_CONFIG_LOGIN_BANNER="Authorised users only!"' \
 -e 'PGADMIN_CONFIG_CONSOLE_LOG_LEVEL=10' \
 -d dpage/pgadmin4
```

After getting these informations, its time to add this into our docker-compose file with postgreSQL
write dc

docker-compose.yml
```
version: '3.4'
…discountdb:
 image: postgrespgadmin: — — — — — — — — — — ADDED
 image: dpage/pgadmin4volumes:
 portainer_data:
 postgres_data:
 pgadmin_data: — — — — — — — ADDED
```

docker-compose.override.yml
```
pgadmin:
 container_name: pgadmin
 environment:
 — PGADMIN_DEFAULT_EMAIL=admin@aspnetrun.com
 — PGADMIN_DEFAULT_PASSWORD=admin1234
 restart: always
 ports:
 — "5050:80"
 volumes:
 — pgadmin_data:/root/.pgadmin
```

So now we had postgres database and also pgadmin management tool of postgres.

RUN with below command on that location;
```
docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d
 docker-compose -f docker-compose.yml -f docker-compose.override.yml down
```

Check portainer containers
localhost:9000

Check pgAdmin

localhost:5050

Login system with
— PGADMIN_DEFAULT_EMAIL=admin@aspnetrun.com
— PGADMIN_DEFAULT_PASSWORD=admin1234

After taht we can manage our postgresql database with adding new server and connect to our discountdb database.
```
Add New Server
 General
 name — DiscountServerConnection
 hostname — discountdb — this should be the docker container name of postresql
 username — admin
 password — admin1234

 See dc-override of postresql
 discountdb:
 container_name: discountdb
 environment:
 — POSTGRES_USER=admin
 — POSTGRES_PASSWORD=admin1234
```

See "DiscountDb" database under servers.
You can create tables under schemas.

As you can see that, we have added **pgAdmin management portal** of postgresql database in our dc file. And run our docker-compose file and see that we can manage our postgresql database in pgAdmin portal like adding new server and see our Discount db.

Create Coupon Table in the DiscountDb of PostgreSQL Database with pgAdmin Management Portal

We are going to Create Coupon Table in the DiscountDb of PostgreSQL Database with pgAdmin Management.

See “DiscountDb” database under servers.
You can create tables under schemas.

Open in pgadmin
Tools — Query Tool

Create Table with below script

```
Tools — Query ToolCREATE TABLE Coupon(
 ID SERIAL PRIMARY KEY NOT NULL,
 ProductName VARCHAR(24) NOT NULL,
 Description TEXT,
 Amount INT
 );
```

Now our Postresql database and table is ready and loaded with predefined data.

Developing Discount.API Microservices Creating Entities

We should create entity objects for redis definitions.

Create “**Entities**” folder

Entities
Add Coupon class

```
namespace Discount.API.Entities
 {
 public class Coupon
 {
 public int Id { get; set; }
 public string ProductName { get; set; }
 public string Description { get; set; }
 public int Amount { get; set; }
 }
 }
```

Developing Repository Pattern Connect PostgreSQL use Dapper on Discount.API Microservice

We are going to Developing Repository Pattern Connect PostgreSQL use Dapper on Discount.API Microservice.
We will encapsulate data related objects in the repository classes by this way there is an abstraction with the repository layer.

First of all, we should install required nuget packages

Install Nuget Package
Open Package Manager Console — PMC
Select Project — Discount.API

```
 Run Command:
 Install-Package Npgsql
 Install-Package Dapper
```

If required Update Packages
Update-Package -ProjectName Discount.API

Dapper

Dapper is a Micro-ORM (Object Relationship Mapper) tool developed by the Stack Overflow team as LightWeight, published open source on Github. It has support for most databases (SQL Server, MySQL, PosgreSQL so on).

In Ado.Net, we perform our queries or procedures using SqlDataReader, SqlCommand etc. objects. Dapper takes the burden of writing these objects from us. We can do our filtering by using generic and extension methods. By writing less code, we can execute our queries in a short time and convert them to the type we want. The biggest feature is it works very fast speed and very close to Ado.Net speed.

Create Business Layer

After that, We should create Create “**Repositories**” and **IDiscountRepository** interface.

Repositories Folder

Add Interface — **IDiscountRepository**
```
namespace Discount.API.Repositories.Interfaces
 {
 public interface IDiscountRepository
 {
 Task<Coupon> GetDiscount(string productName);Task<bool> CreateDiscount(Coupon coupon);
 Task<bool> UpdateDiscount(Coupon coupon);
 Task<bool> DeleteDiscount(string productName);
 }
 }
```

After that, we can implement this interface with using NpgsqlConnection objects.

Create Repository — **DiscountRepository**
```
namespace Basket.API.Repositories
{
 public class DiscountRepository : IDiscountRepository
 {
 private readonly IConfiguration _configuration;public DiscountRepository(IConfiguration configuration)
 {
 _configuration = configuration ?? throw new ArgumentNullException(nameof(configuration));
 }public async Task<Coupon> GetDiscount(string productName)
 {
 using var connection = new NpgsqlConnection(_configuration.GetValue<string>(“DatabaseSettings:ConnectionString”));

 var coupon = await connection.QueryFirstOrDefaultAsync<Coupon>
 (“SELECT * FROM Coupon WHERE ProductName = @ProductName”, new { ProductName = productName });

 if (coupon == null)
 return new Coupon { ProductName = “No Discount”, Amount = 0, Description = “No Discount Desc” };
 return coupon;
 }public async Task<bool> CreateDiscount(Coupon coupon)
 {
 using var connection = new NpgsqlConnection(_configuration.GetValue<string>(“DatabaseSettings:ConnectionString”));var affected =
 await connection.ExecuteAsync
 (“INSERT INTO Coupon (ProductName, Description, Amount) VALUES (@ProductName, @Description, @Amount)”,
 new { ProductName = coupon.ProductName, Description = coupon.Description, Amount = coupon.Amount });if (affected == 0)
 return false;return true;
 }public async Task<bool> UpdateDiscount(Coupon coupon)
 {
 using var connection = new NpgsqlConnection(_configuration.GetValue<string>(“DatabaseSettings:ConnectionString”));var affected = await connection.Exe
 (“UPDATE Coupon SET ProductName=@ProductName, Description = @Description, Amount = @Amount WHERE Id = @Id”,
 new { ProductName = coupon.ProductName, Description = coupon.Description, Amount = coupon.Amount, Id = coupon.Id });if (affected == 0)
 return false;return true;
 }public async Task<bool> DeleteDiscount(string productName)
 {
 using var connection = new NpgsqlConnection(_configuration.GetValue<string>(“DatabaseSettings:ConnectionString”));var affected = await connection.Exe
 new { ProductName = productName });if (affected == 0)
 return false;return true;
 }
 }
}
```

Basically we are using “**NpgsqlConnection**” object as a **postgresql** connection. This will create connection for us with using connection string. After that with using “connection” object and methods, we are using **Dapper** sql commands. This will be the fastest way to use **micro-orm** to **retrieve data** from databases.

Create DiscountController Class for Discount.API Microservice

First of all, We should create DiscountController into Controller class.

Controller + Startup.cs
```
[ApiController]
 [Route(“api/v1/[controller]”)]
 public class DiscountController : ControllerBase
 {
 private readonly IDiscountRepository _repository;public DiscountController(IDiscountRepository repository)
 {
 _repository = repository ?? throw new ArgumentNullException(nameof(repository));
 }

 [HttpGet(“{productName}”, Name = “GetDiscount”)]
 [ProducesResponseType(typeof(Coupon), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<Coupon>> GetDiscount(string productName)
 {
 var discount = await _repository.GetDiscount(productName);
 return Ok(discount);
 }[HttpPost]
 [ProducesResponseType(typeof(Coupon), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<Coupon>> CreateDiscount([FromBody] Coupon coupon)
 {
 await _repository.CreateDiscount(coupon);
 return CreatedAtRoute(“GetDiscount”, new { productName = coupon.ProductName }, coupon);
 }[HttpPut]
 [ProducesResponseType(typeof(Coupon), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<Coupon>> UpdateDiscount([FromBody] Coupon coupon)
 {
 return Ok(await _repository.UpdateDiscount(coupon));
 }[HttpDelete("{productName}", Name = "DeleteDiscount")]
 [ProducesResponseType(typeof(void), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<bool>> DeleteDiscount(string productName)
 {
 return Ok(await _repository.DeleteDiscount(productName));
 }
 }
```

we have injected **IDiscountRepository** object and use this object when exposing apis.
You can see that we have exposed crud api methods over the DiscountController.

Also you can see the anotations.
ProducesResponseType — provide to restrictions on api methods. produce only including types of objects.
Route — provide to define custom routes. Mostly using if you have one more the same http method.
httpget/put/post — send http methods.

Test and Run Discount Microservice

We are going to Test and Run Discount Microservice. Now the **Discount microservice** Web API application ready to run.

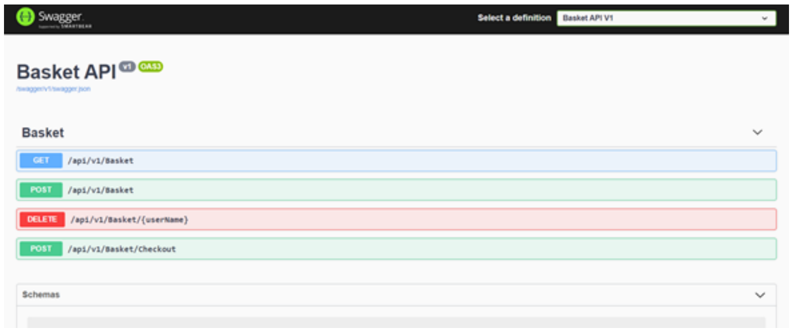
Before running the application, **configure the debug profile**;

Right Click the project File and Select to Debug section.

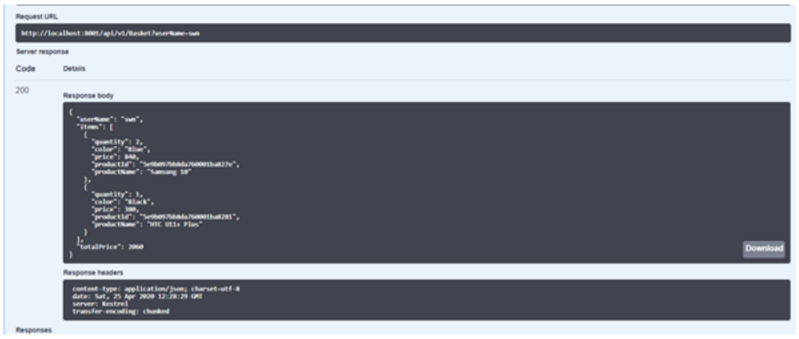
Change Launch browser to **swagger**

Change the App URL to **http://localhost:5002**

Hit **F5** on **Discount.API** project.



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



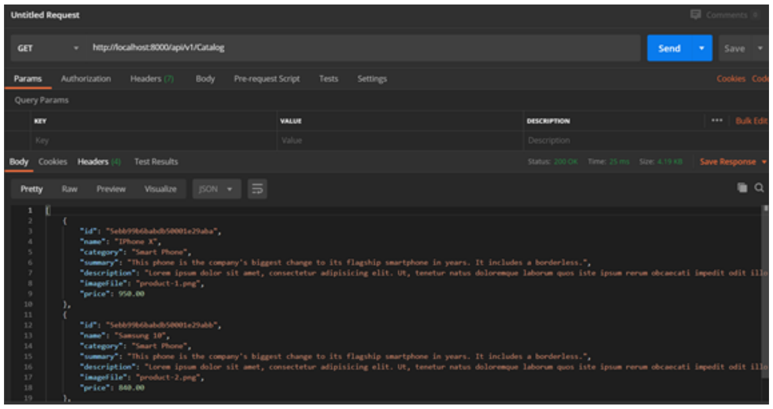
```
http://localhost:8001/api/v1/Basket?userName=swn
```

200

```
{
  "userName": "swn",
  "items": [
    {
      "quantity": 2,
      "color": "Blue",
      "price": 840,
      "productId": "5e9b097bbdda760001ba827e",
      "productName": "Samsung 10"
    },
    {
      "quantity": 1,
      "color": "Black",
      "price": 380,
      "productId": "5e9b097bbdda760001ba8281",
      "productName": "HTC U11+ Plus"
    }
  ],
  "totalPrice": 2060
}
```

```
content-type: application/json; charset=utf-8
date: Sat, 25 Apr 2020 12:28:29 GMT
server: Kestrel
transfer-encoding: chunked
```

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



```
[
    {
        "id": "5ebb99b6babdb50001e29aba",
        "name": "IPhone X",
        "category": "Smart Phone",
        "summary": "This phone is the company's biggest change to its flagship smartphone in years. It includes a borderless.",
        "description": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ut, tenetur natus doloremque laborum quos iste ipsum rerum obcaecati impedit odit illo
        "imageFile": "product-1.png",
        "price": 950.00
    },
    {
        "id": "5ebb99b6babdb50001e29abb",
        "name": "Samsung 10",
        "category": "Smart Phone",
        "summary": "This phone is the company's biggest change to its flagship smartphone in years. It includes a borderless.",
        "description": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ut, tenetur natus doloremque laborum quos iste ipsum rerum obcaecati impedit odit illo
        "imageFile": "product-2.png",
        "price": 840.00
    },
```

http://localhost:5002/swagger/index.html

See CRUD operations on Swagger
— -
Set productName

productName — IPhone X
productName — Samsung 10

```
Get Success :
 {
 “id”: 1,
 “productName”: “IPhone X”,
 “description”: “IPhone Discount”,
 “value”: 150
 }

 Create
 {
 “productName”: “Huawei Plus”,
 “description”: “test new product”,
 “value”: 550
 }Update
 {
 “productName”: “Huawei Plus”,
 “description”: “test update”,
 “value”: 200
 }Delete
 productName — IPhone X
 productName — Samsung 10
 productName — Huawei Plus
```

As you can see that, we have test our Discount microservices is working fine.