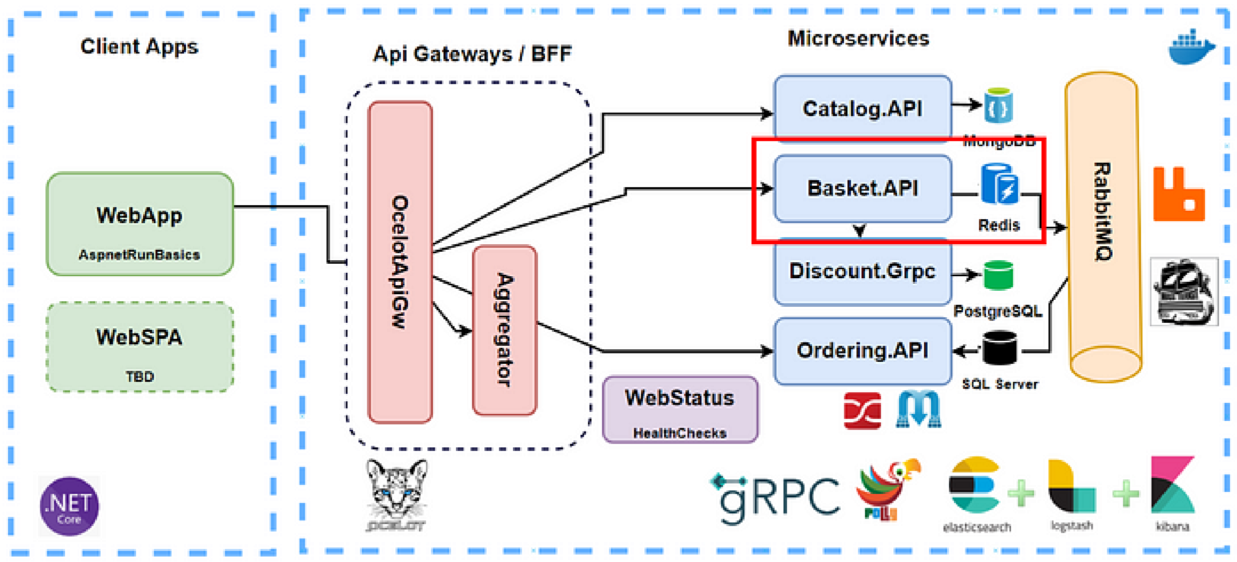
# Using Redis with ASP.NET Core, and Docker Container for Basket Microservices

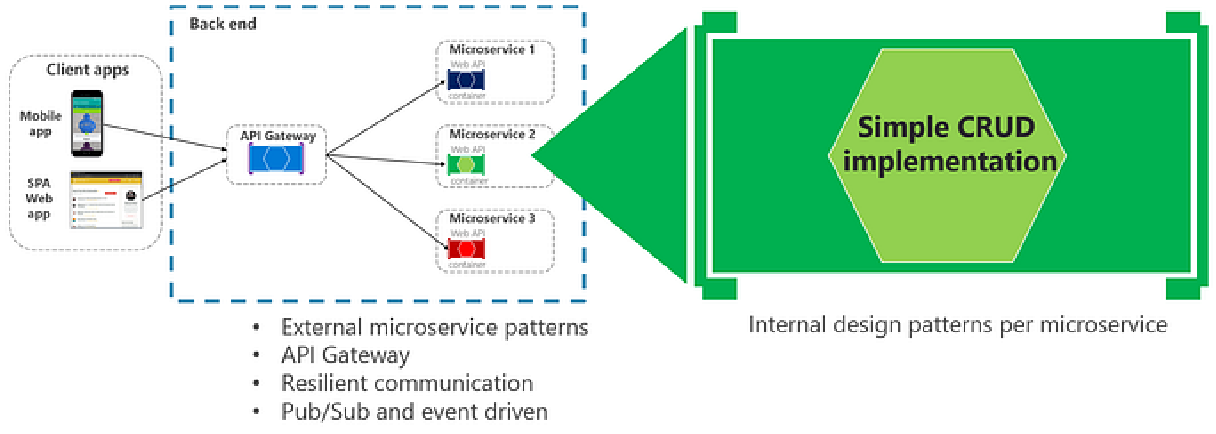
*Building Basket Microservice on .Net platforms which used Asp.Net Web API, Docker, Redis and Swagger. Test microservice with using Postman.*



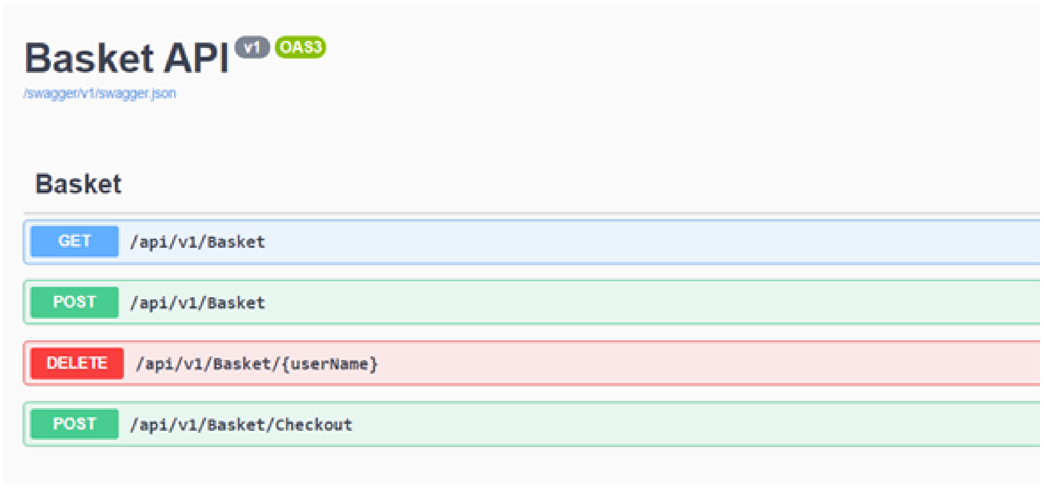
## Introduction

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

By the end of the article, we will have a Web API which implemented CRUD operations over **Basket** and **BasketItem** objects. These objects will be store in **Redis** as a **cache value** so in our case we will use Redis for NoSQL database.



Look at the final swagger application.



```
GET     /api/v1/Basket
POST    /api/v1/Basket
DELETE  /api/v1/Basket/{userName}
POST    /api/v1/Basket/Checkout
```

Developing Basket microservice which includes;

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

In the upcomming articles :

- Consume Discount Grpc Service for inter-service sync communication to calculate product final price
- Publish BasketCheckout Queue with using MassTransit and RabbitMQ

At the end, you'll have a working Web API Microservice running on your local machine.

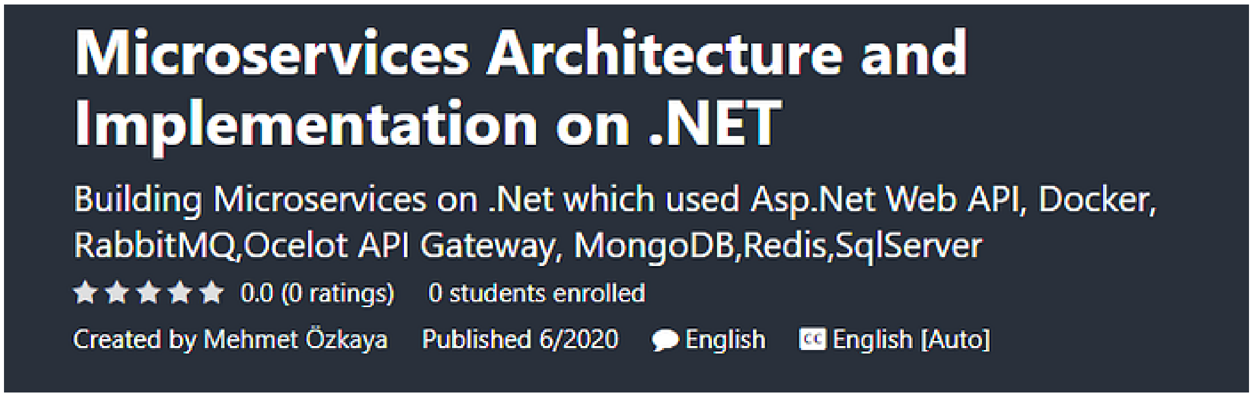
## Background

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 Basket 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

## Redis

Redis, an abbreviation of **Remote Dictionary Server** expression; It positions itself as a data structure server. **Redis** is an open source NoSQL database that originally holds memory.

**Redis** is not just a **simple key-value server**. One of the most important differences among other alternatives is that Redis ability to store and use high level data structures.

As mentioned in the definition of Redis own documentation, redis is not just a simple key-value server. One of the major differences among other alternatives is the ability of Redis to store and use high-level data structures. These data structures are the basic data that most developers are familiar with. structures (list, map, set).

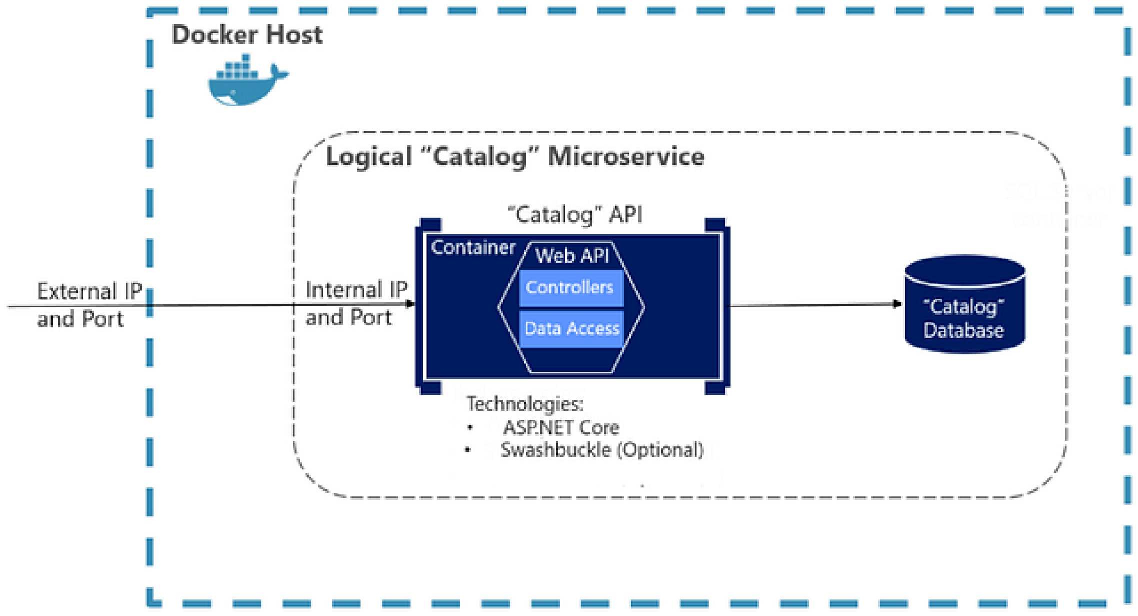
**Advantages**
It is extremely fast because it works synchronously. It supports many data types. It can save data both on RAM and on disk according to the configuration you set. Since it records on the disc, it continues to work with the same data after restart. It has many enterprise features such as Sharding, Cluster, Sentinel, Replication.

**Disadvantages**
Since it does not work asynchronously, you may not be able to reach the performance that asynchronous alternatives reach on a single instance. You will need RAM according to your data size. It does not support complex queries like relational databases. If a transaction receives an error, there is no return.

## Analysis & Design

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

## Data-Driven/CRUD microservice container



We should define our Basket use case analysis. In this part we will create Basket — BasketItem entities. Our main use case are Listing Basket and Items, able to add new item to basket. Also performed CRUD operations on Basket entity.

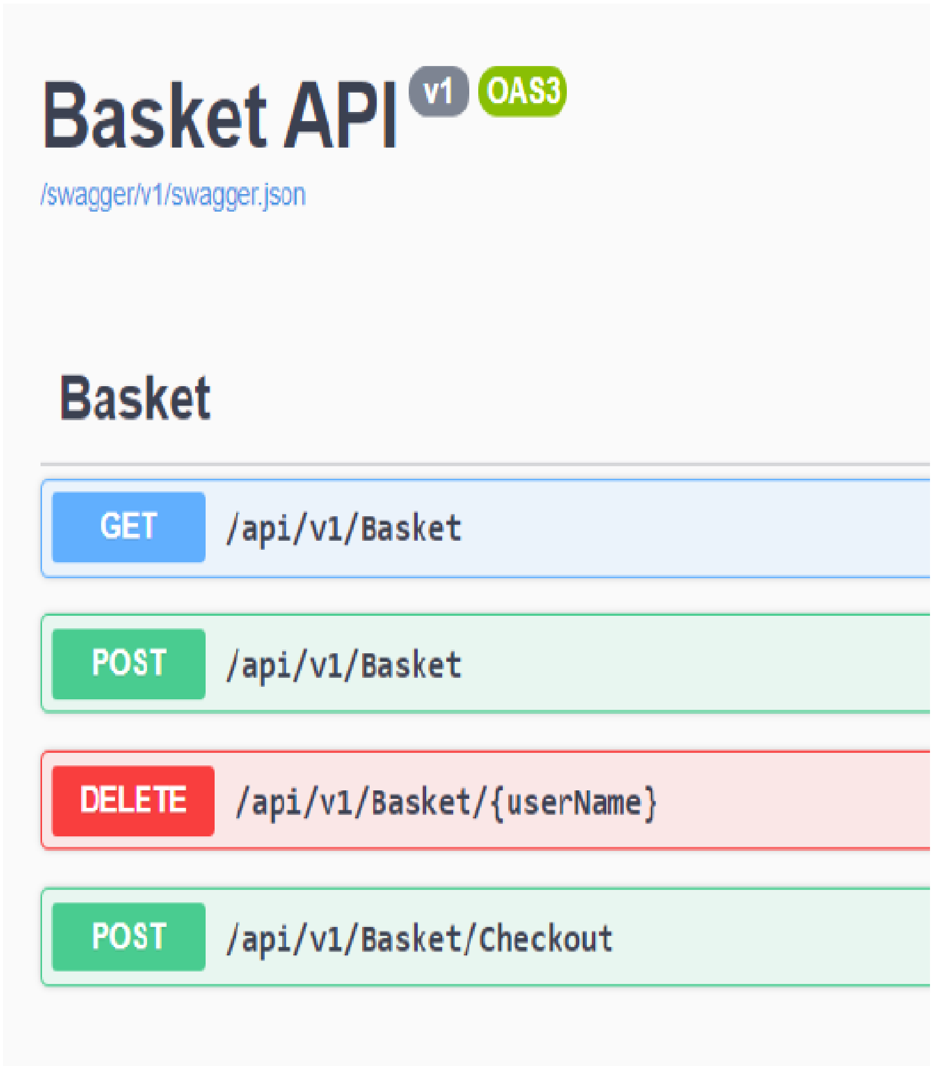
Our main use cases;

- Get Basket and Items with username
- Update Basket and Items (add — remove item on basket)
- Delete Basket
- Checkout Basket

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

| **Method** | **Request URI** | **Use Case** |
|---|---|---|
| GET | api/v1/Basket | Get Basket and Items with username |
| POST | api/v1/Basket | Update Basket and Items (add – remove item on basket) |
| DELETE | api/v1/Basket/{id} | Delete Basket |
| POST | api/v1/Basket/Checkout | Checkout Basket |

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


Basket API v1 OAS3

/swagger/v1/swagger.json

Basket

GET /api/v1/Basket

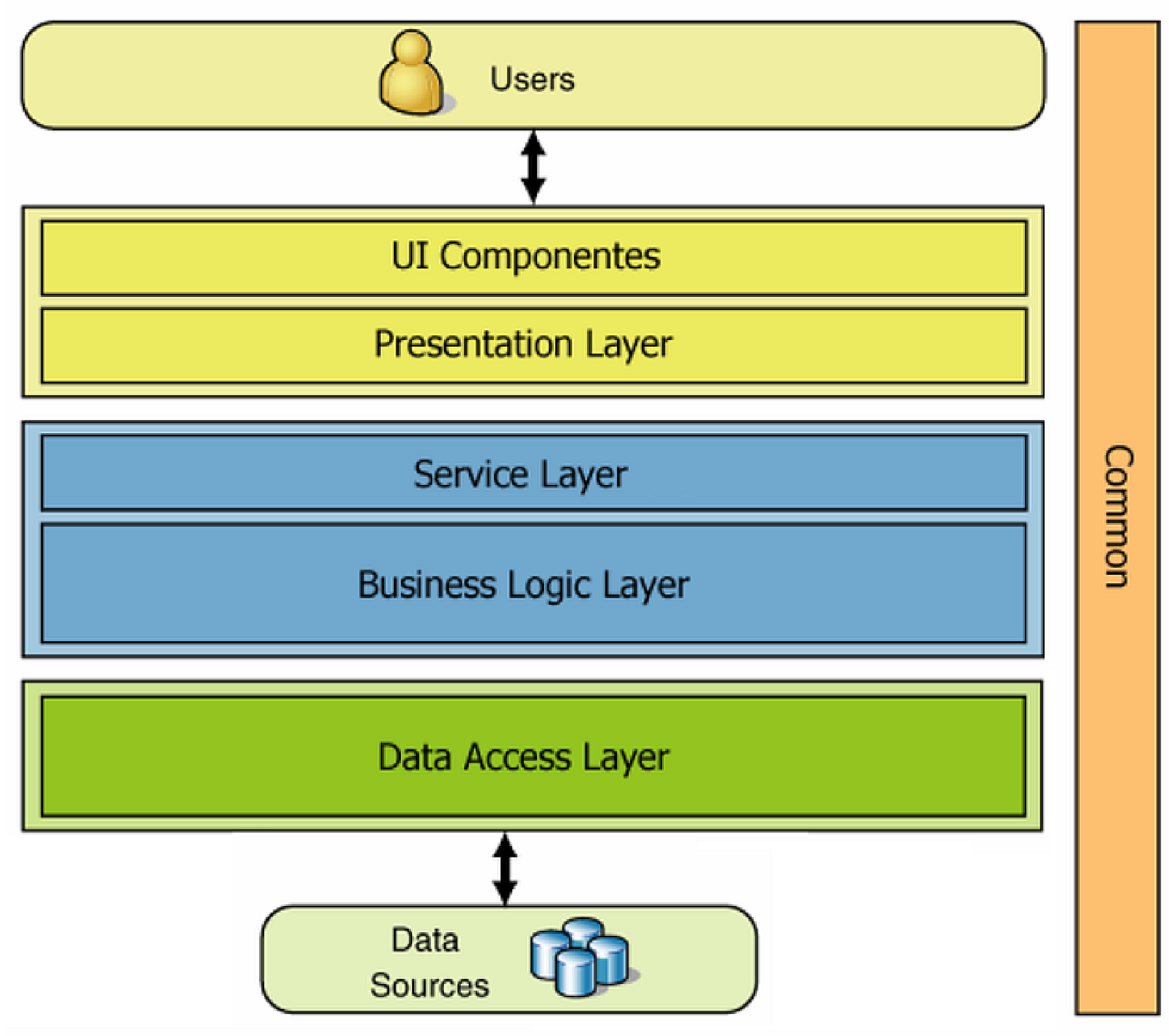
POST /api/v1/Basket

DELETE /api/v1/Basket/{userName}

POST /api/v1/Basket/Checkout

## Architecture of Basket 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.



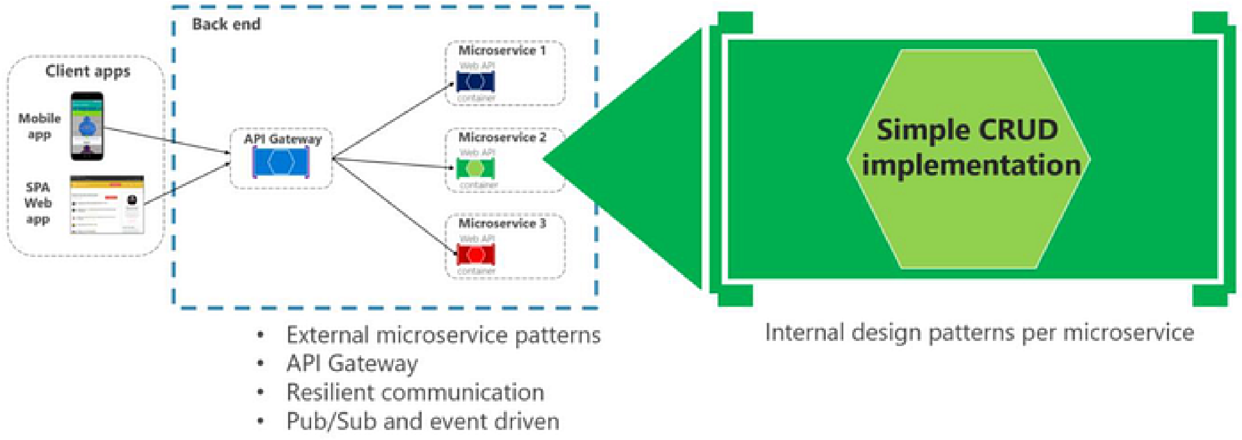
**Data Access Layer:** Only database operations are performed on this layer.
The task of this layer is to add, delete, update and extract data from the database. There is no other operation in this layer other than these operations.

**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.

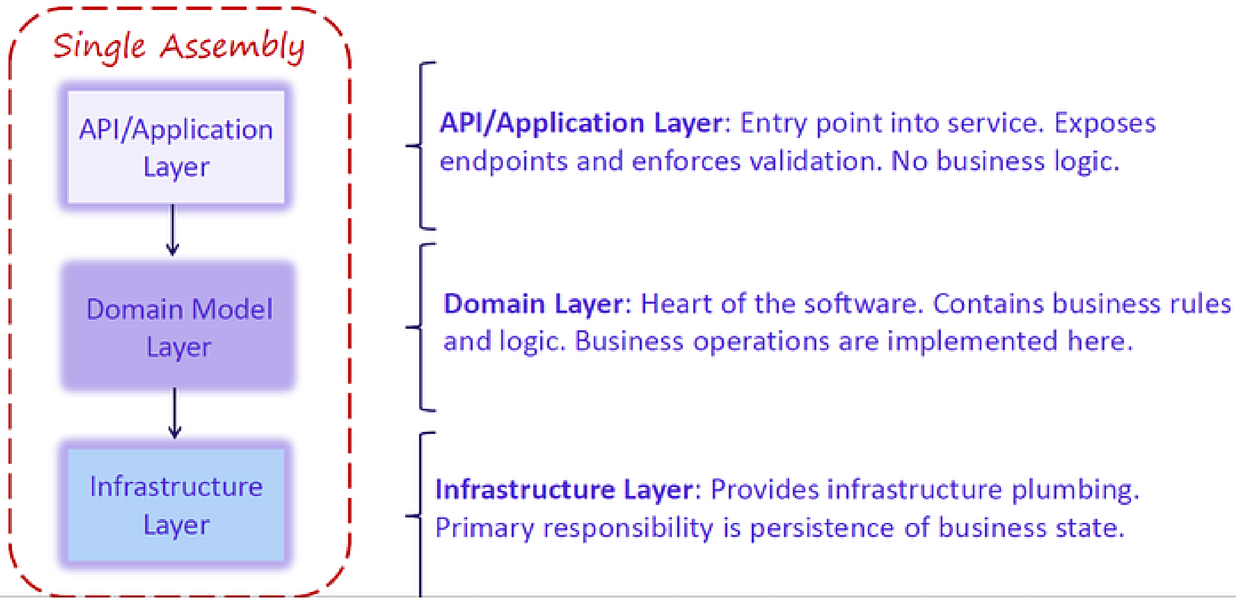
## Simple Data Driven CRUD Microservice

Basket.API microservices will be simple crud implementation on Basket data on Redis 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.


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

## Database Setup with Docker

For Basket microservices, we are going to use no-sql Redis database.

## Setup Redis Database

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

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

**docker pull redis**

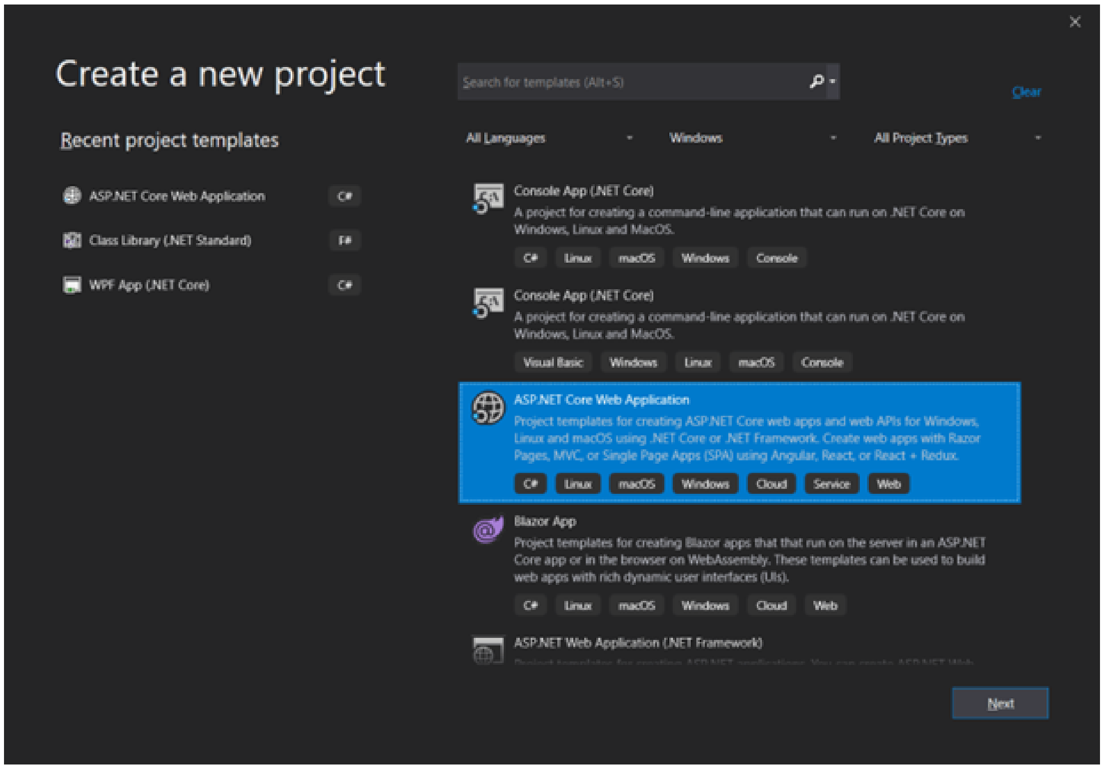
To run database on your docker environment use below command. It will expose 6379 port in your local environment.

**docker run -d -p 6379:6379 — name aspnetrun-redis redis**

## 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.


Create a new project
Search for templates (Alt+S)
Clear
Recent project templates
ASP.NET Core Web Application C#
Class Library (.NET Standard) F#
WPF App (.NET Core) C#
All Languages
Windows
All Project Types
Console App (.NET Core)
C# Linux macOS Windows Console
Console App (.NET Core)
Visual Basic Windows Linux macOS Console
ASP.NET Core Web Application
C# Linux macOS Windows Cloud Service Web
Blazor App
C# Linux macOS Windows Cloud Web
ASP.NET Web Application (.NET Framework)
Next

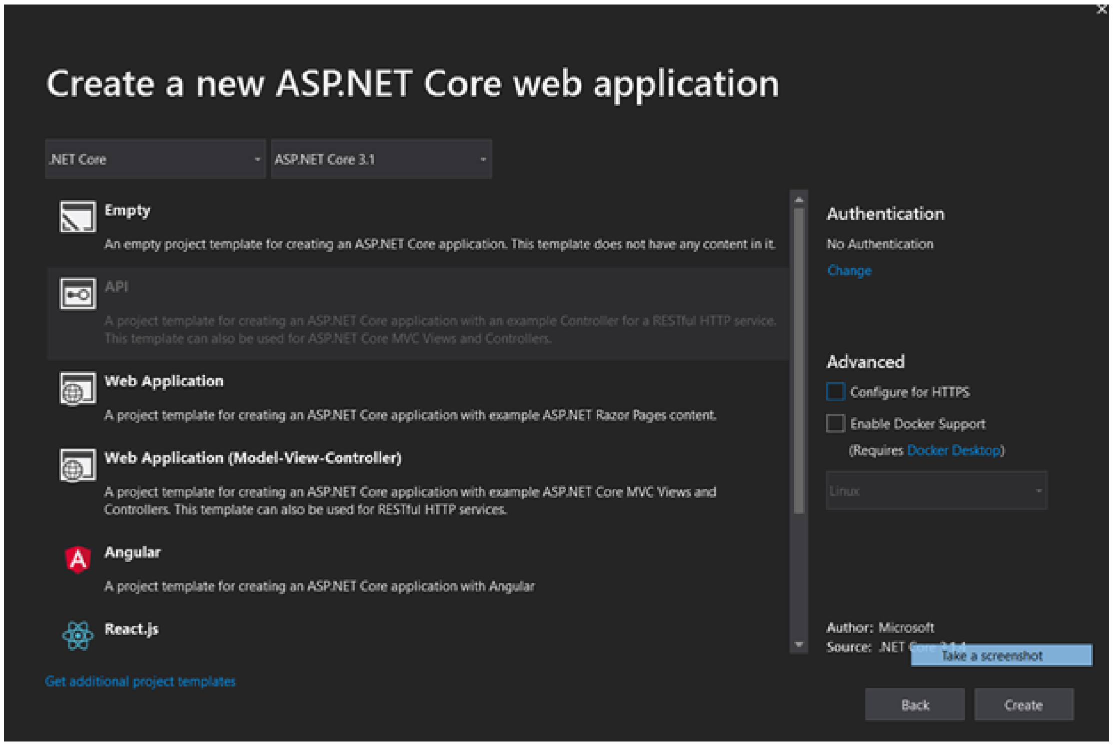
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.

Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).

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.



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 3.1

**Empty**
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

**API**
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

**Web Application**
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

**Web Application (Model-View-Controller)**
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

**Angular**
A project template for creating an ASP.NET Core application with Angular

**React.js**

Get additional project templates

Authentication
No Authentication
Change

Advanced
Configure for HTTPS
Enable Docker Support
(Requires Docker Desktop)
Linux

Author: Microsoft
Source: .NET

Add New Web API project under below location and name;

**src/catalog/Basket.API**

## Library & Frameworks

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

- Microsoft.Extensions.Caching.StackExchangeRedis — To connect redis database
- Newtonsoft.Json — To parse json objects
- Swashbuckle.AspNetCore — To generate swagger index page

## Create Entities

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

Create "**Entities**" folder

Entities -> Add **ShoppingCart** and **ShoppingCartItem** class

```
using System.Collections.Generic;namespace Basket.API.Entities
 {
     public class ShoppingCart
     {
         public string UserName { get; set; }
         public List<ShoppingCartItem> Items { get; set; } = new List<ShoppingCartItem>();public ShoppingCart()
         {
         }public ShoppingCart(string userName)
         {
             UserName = userName;
         }

         public decimal TotalPrice
         {
             get
```

```
            {
                decimal totalprice = 0;
                foreach (var item in Items)
                {
                    totalprice += item.Price * item.Quantity;
                }
                return totalprice;
            }
        }
    }
}
```

**ShoppingCartItem.cs**

```
namespace Basket.API.Entities
{
    public class ShoppingCartItem
    {
        public int Quantity { get; set; }
        public string Color { get; set; }
        public decimal Price { get; set; }
        public string ProductId { get; set; }
        public string ProductName { get; set; }
    }
}
```

## Connect Redis Docker Container from Basket.API Microservice with AddStackExchangeRedisCache into DI

We are going to Connect Redis Docker Container from Basket.API Microservice. This will be the Redis 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 Redis Client.

After that, we have to configure our application to support the Redis cache and specify the port at which Redis is available. To do this, navigate to Startup.cs/ConfigureServices method and add the following.

**Startup.cs/ConfigureServices**

```
services.AddStackExchangeRedisCache(options =>
 {
 options.Configuration = “localhost:6379”;
 });
```

It is good to get this url from the configuration.

Move Configuration appsettings.json

```
“CacheSettings”: {
 “ConnectionString”: “localhost:6379”
 },
```

— Update startup with configuration

```
services.AddStackExchangeRedisCache(options =>
 {
 options.Configuration = Configuration.GetValue<string>(“CacheSettings:ConnectionString”);
 });
```

Now it is ready to use with **IDistributedCache**.
— With this **AddStackExchangeRedisCache extention method**, it is provide to inject any class with **IDistributedCache** and create an instance for us.
We are going to inject this class in a repository classes.

## 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.

The name of the Repository, would not be appreciate for Business Layer but we are building one solution that's why we select this name, you can modify with name of Service, Application, Manager, Helper etc.

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

- Get Basket and Items with username
- Update Basket and Items (add — remove item on basket)
- Delete Basket
- Checkout Basket

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

```
public interface IBasketRepository
    {
        Task<BasketCart> GetBasket(string userName);
        Task<BasketCart> UpdateBasket(BasketCart basket);
        Task<bool> DeleteBasket(string userName);
    }
```

Implementation of Repository interface should be as below code;

```
using Basket.API.Entities;
 using Basket.API.Repositories.Interfaces;
 using Microsoft.Extensions.Caching.Distributed;
 using Newtonsoft.Json;
 using System;
 using System.Threading.Tasks;namespace Basket.API.Repositories
 {
     public class BasketRepository : IBasketRepository
     {
         private readonly IDistributedCache _redisCache;public BasketRepository(IDistributedCache cache)
         {
             _redisCache = cache ?? throw new ArgumentNullException(nameof(cache));
         }public async Task<ShoppingCart> GetBasket(string userName)
         {
             var basket = await _redisCache.GetStringAsync(userName);if (String.IsNullOrEmpty(basket))
                 return null;return JsonConvert.DeserializeObject<ShoppingCart>(basket);
         }

         public async Task<ShoppingCart> UpdateBasket(ShoppingCart basket)
         {
             await _redisCache.SetStringAsync(basket.UserName, JsonConvert.SerializeObject(basket));

             return await GetBasket(basket.UserName);
         }public async Task DeleteBasket(string userName)
         {
             await _redisCache.RemoveAsync(userName);
         }
     }
 }
```

Let me try to explain this methods,
— basically we are using "**IDistributedCache**" object as a redis cache. Because in the startup class, we have configured redis as a distributed cache.
— So when we inject this interface, this will create an redis cache instance.

— After that with using "**IDistributedCache**" object almost every Redis CLI commands include in this context class as a method members.
— so we have implemented this methods into our api requirements. For example **GetStringAsync** , **SetStringAsync**..
— We have used JsonConvert in order to save and extract json objects from redis cache databases. So our basket and basket item structure saving to redis a a json objects.

# 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 **BasketController** class.

```
using Basket.API.Entities;
 using Basket.API.Repositories.Interfaces;
 using Microsoft.AspNetCore.Mvc;
 using System;
 using System.Net;
 using System.Threading.Tasks;namespace Basket.API.Controllers
 {
     [ApiController]
     [Route("api/v1/[controller]")]
     public class BasketController : ControllerBase
     {
         private readonly IBasketRepository _repository;public BasketController(IBasketRepository repository)
         {
             _repository = repository ?? throw new ArgumentNullException(nameof(repository));
         }[HttpGet("{userName}", Name = "GetBasket")]
         [ProducesResponseType(typeof(ShoppingCart), (int)HttpStatusCode.OK)]
         public async Task<ActionResult<ShoppingCart>> GetBasket(string userName)
         {
             var basket = await _repository.GetBasket(userName);
             return Ok(basket ?? new ShoppingCart(userName));
         }[HttpPost]
         [ProducesResponseType(typeof(ShoppingCart), (int)HttpStatusCode.OK)]
         public async Task<ActionResult<ShoppingCart>> UpdateBasket([FromBody] ShoppingCart basket)
         {
             return Ok(await _repository.UpdateBasket(basket));
         }[HttpDelete("{userName}", Name = "DeleteBasket")]
         [ProducesResponseType(typeof(void), (int)HttpStatusCode.OK)]
         public async Task<IActionResult> DeleteBasket(string userName)
         {
             await _repository.DeleteBasket(userName);
             return Ok();
         }
     }
 }
```

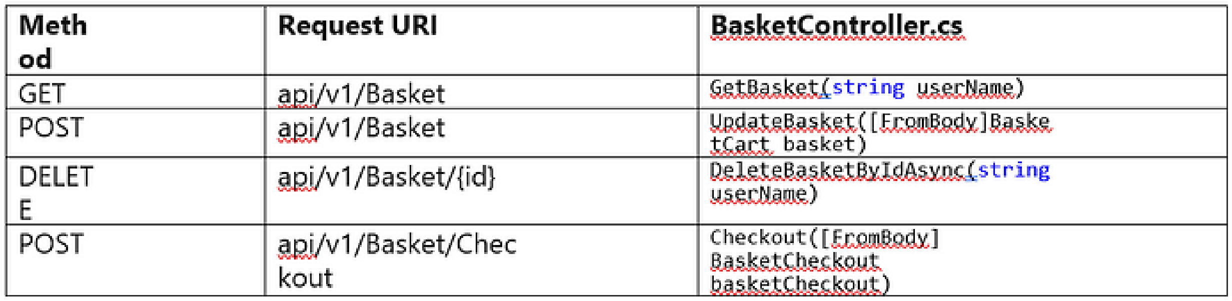
we have injected **IBasketRepository** object and use this object when exposing apis.
— you can see that we have exposed crud api methods over the Basket Controller.

— 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.

## API Routes in Controller Classes

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

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



| Meth od | Request URI | BasketController.cs |
|---|---|---|
| GET | api/v1/Basket | GetBasket(string userName) |
| POST | api/v1/Basket | UpdateBasket([FromBody]BasketCart basket) |
| DELETE | api/v1/Basket/{id} | DeleteBasketByIdAsync(string userName) |
| POST | api/v1/Basket/Checkout | Checkout([FromBody] BasketCheckout basketCheckout) |

## 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”);
 });
 }
```

## Register Redis into AspNet Built-in Dependency Injection Tool

After we have developed the Controller, the most important part is registering objects into aspnet built-in dependency injection tool.
— So in our case, Controller class uses Repository object and Repository object uses IDistributedCache objects. That means we should register both repostiory and IDistributedCache object.

Go to Startup.cs
Register Repository into DI Service collections.

Add Startup Configurations

```
public void ConfigureServices(IServiceCollection services)
 {
 services.AddStackExchangeRedisCache(options =>
 {
 options.Configuration = Configuration.GetValue<string>(“DatabaseSettings:ConnectionString”);
 });

 services.AddScoped<IBasketRepository, BasketRepository>();
```

— For **IDistributedCache**, we have already registered with **AddStackExchangeRedisCache** extention method.

## Run Application

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

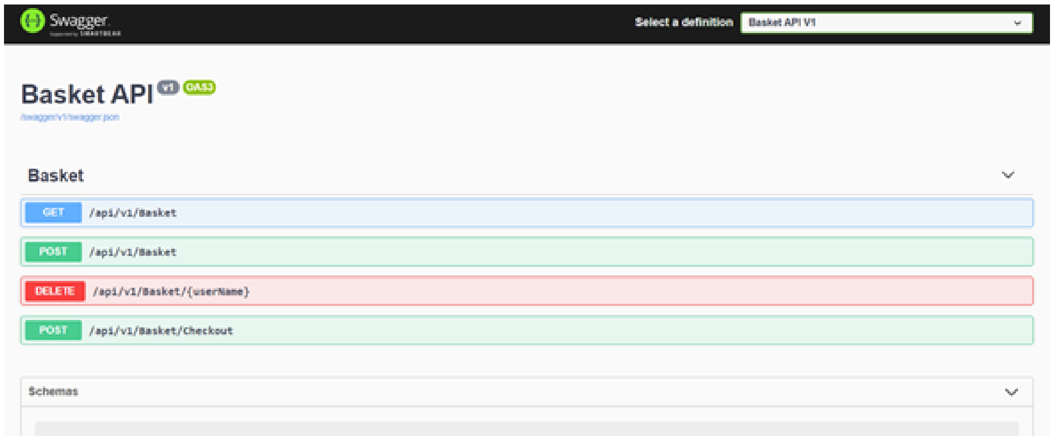
Before running the application, **configure the debug profile**;

*Right Click the project File and Select to Debug section.*

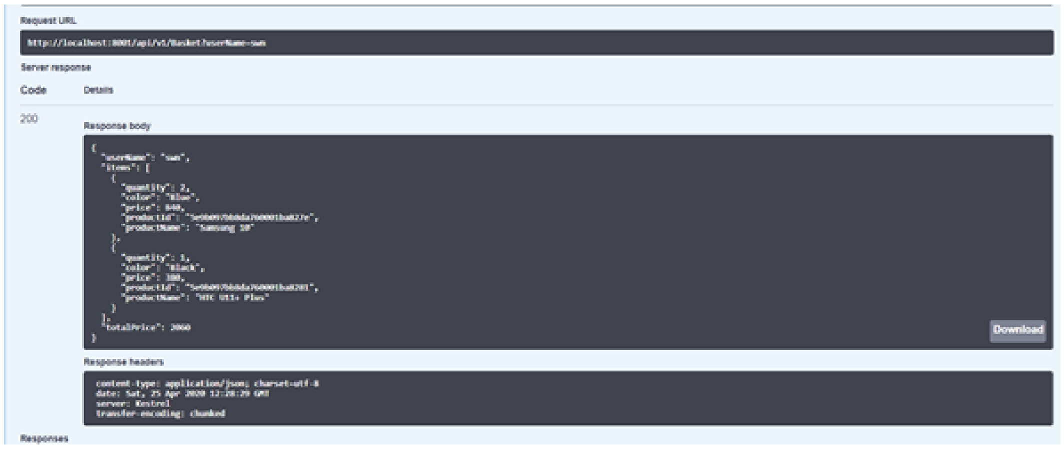
Change Launch browser to **swagger**

Change the App URL to **http://localhost:5001**

Hit **F5** on **Basket.API** project.



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


Request URL

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

Server response

Code Details

200 Response body

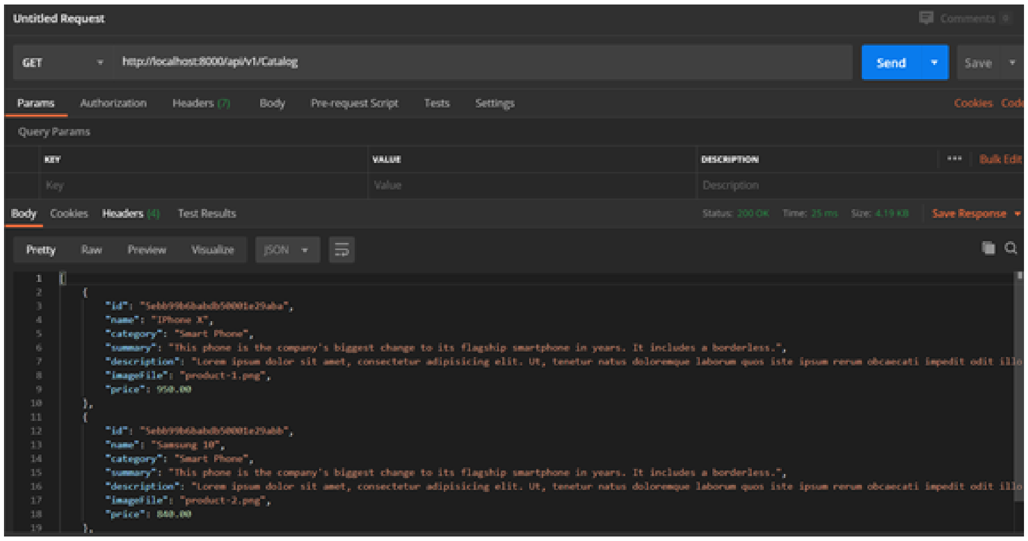
```
{
  "userName": "swn",
  "items": [
    {
      "quantity": 2,
      "color": "Blue",
      "price": 840,
      "productId": "5e9b097bb8da760001ba827e",
      "productName": "Samsung 10"
    },
    {
      "quantity": 1,
      "color": "Black",
      "price": 380,
      "productId": "5e9b097bb8da760001ba8281",
      "productName": "HTC U11+ Plus"
    }
  ],
  "totalPrice": 2060
}
```

Response headers

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

Responses

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



GET http://localhost:8000/api/v1/Catalog

Status: 200 OK Time: 25 ms Size: 4.19 KB

```
[
    {
        "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
    },
```

# 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 **Basket API** project with our **Redis**.

## 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:
    basketdb:
        image: redisbasket.api:
        image: ${DOCKER_REGISTRY-}basketapi
        build:
            context: .
            dockerfile: src/Basket/Basket.API/Dockerfile
```

Basically in **docker-compose.yml file**, created 2 image 1 is for Redis which name is **basketdb**, 2 is web api project which name is **basket.api**.

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

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 :

**Basket API ->** http://localhost:8001/swagger/index.html

**SEE DATA** with Test over Swagger

**/api/v1/Basket**