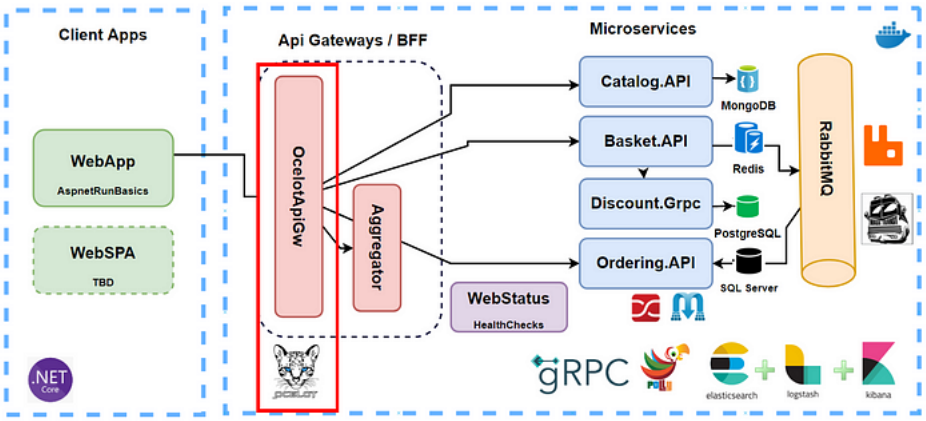
Building Ocelot API Gateway Microservices with ASP.NET Core and Docker Container

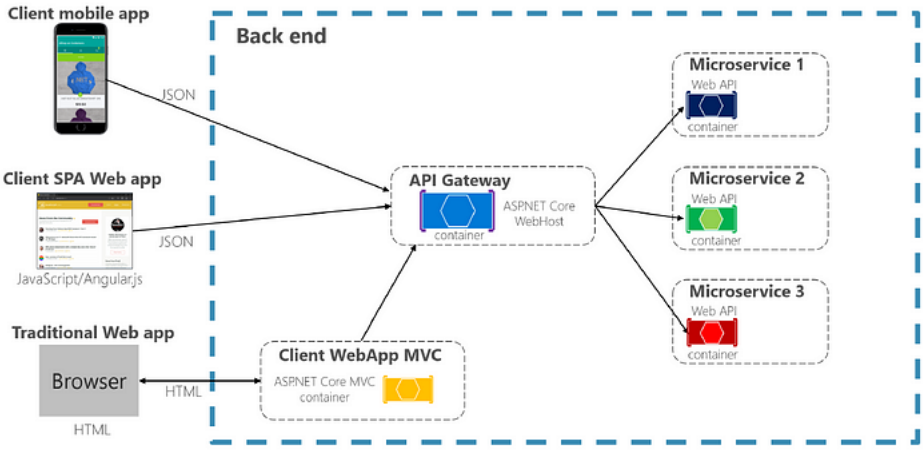
Building Ocelot API Gateway Microservice on .Net platforms which used Asp.Net Web Application, Docker, Ocelot. Test microservice with applying Gateway Routing Pattern.



Introduction

In this article we will show how to perform **API Gateway** microservices operations on ASP.NET Core Web application using **Ocelot Api Gateway** library.

By the end of the section, we will have a empty **Web project** which implemented **Ocelot API Gateway routing operations** over the **Catalog, Discount, Basket and Ordering microservices.**



Look at the final appearance of application.

You'll learn how to Create **Ocelot API Gateway microservice** which includes;

- Implement API Gateways with **Ocelot**
- ASP.NET Core Empty Web application
- **Ocelot Routing, UpStream, DownStream**
- **Ocelot Configuration**
- Sample microservices/containers to **reroute** through the API Gateways
- Run multiple different **API Gateway/BFF container types**

- The **Gateway Aggregation Pattern** in Shopping.Aggregator
- **Containerize** Ocelot Microservices using **Docker Compose**

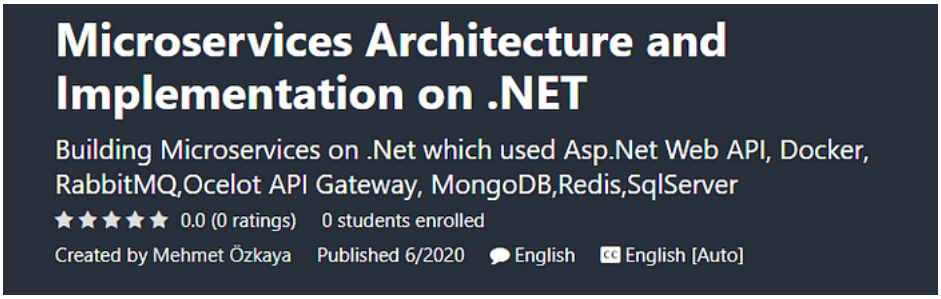
Background

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 Api Gateway microservice from that overall e-commerce microservice architecture.

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

The Gateway Routing pattern

Gateway Routing pattern main objective is Route requests to multiple services using a single endpoint. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.

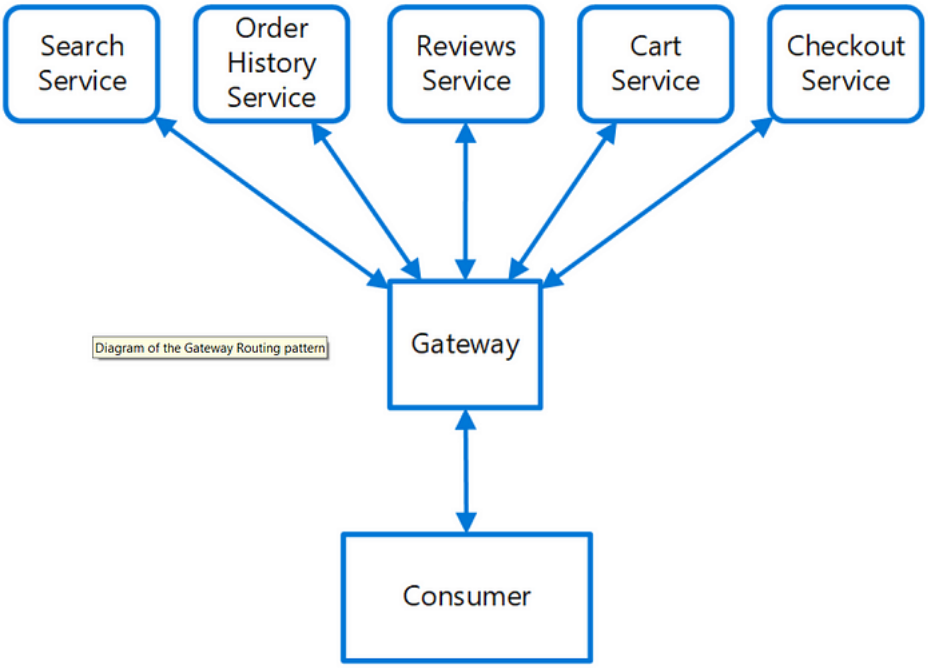


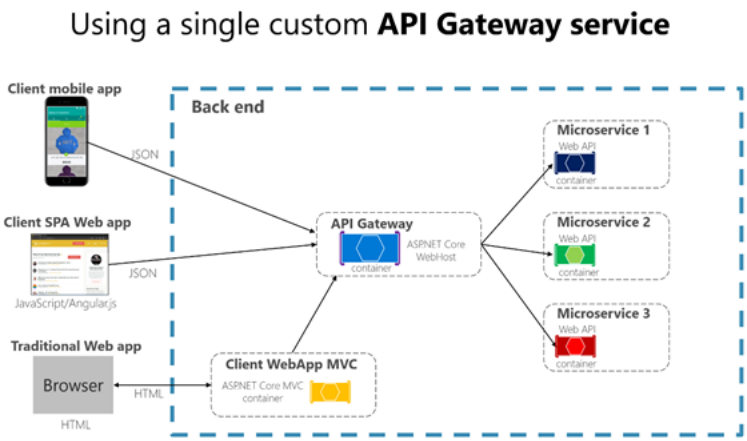
Diagram of the Gateway Routing pattern

When a client needs to consume multiple services, setting up a separate endpoint for each service and having the client manage each endpoint can be challenging. For example, an e-commerce application might provide services such as search, reviews, cart, checkout, and order history.

The solution is to place a gateway in front of a set of applications, services, or deployments. Use application Layer 7 routing to route the request to the appropriate instances. With this pattern, the client application only needs to know about and communicate with a single endpoint.
If a service is consolidated or decomposed, the client does not necessarily require updating. It can continue making requests to the gateway, and only the routing changes. So this pattern is the ancestor of API Gateway Pattern.

Api Gateway Design Pattern

We can simply define our internal services as the proxy layer hanging out.



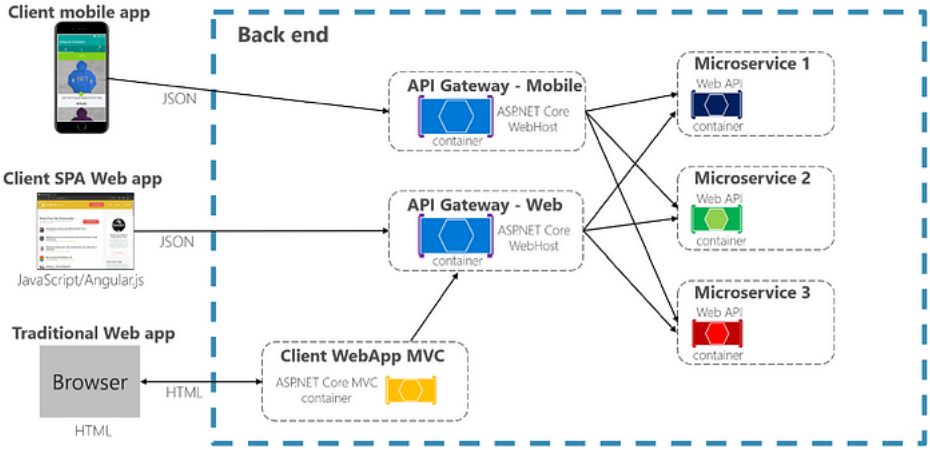
When the user throws a request from the application, he doesn't know what's going on behind. Api-gateway may be going to dozens of microservices inside to respond to this request. Right here, we have the api-gateway pattern to figure out what we need about how to fetch and aggregate data.

This pattern is a service that provides a single-entry point for certain groups of microservices. It's similar to the Facade pattern from object-oriented design, but in this case, it's part of a distributed system. The API Gateway pattern is also sometimes known as the **"Backend For Frontend" (BFF)** because you build it while thinking about the needs of the client app.

Therefore, the API gateway sits between the client apps and the microservices. It acts as a reverse proxy, routing requests from clients to services. It can also provide other cross-cutting features such as authentication, SSL termination, and cache.

Backend for Frontend Pattern — BFF

When splitting the API Gateway tier into multiple API Gateways, if your application has multiple client apps, that can be a primary pivot when identifying the multiple API Gateways types, so that you can have a different facade for the needs of each client app.



This case is a pattern named **“Backend for Frontend” (BFF)** where each API Gateway can provide a different API tailored for each client app type, possibly even based on the client form factor by implementing specific adapter code which underneath calls multiple internal microservices.

Main features in the API Gateway

Reverse proxy or gateway forwarding, API Gateway offers a reverse proxy for forwarding or forwarding requests (layer 7 routing, usually HTTP requests) to endpoints of internal microservices.

As part of the gateway model, request aggregation, you can aggregate multiple client requests into a single client request, often targeting multiple internal microservices.
With this approach, the client application sends a single request to the API Gateway, which sends several requests to internal microservices and then collects the results and sends everything back to the client application.

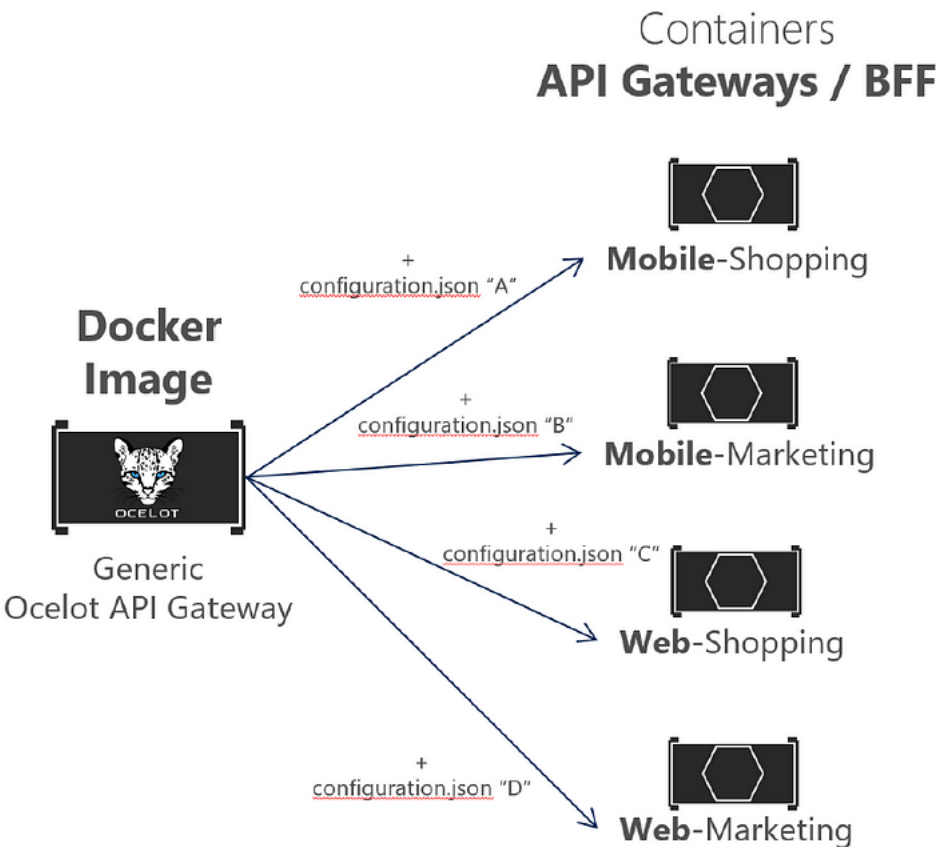
Common Feature;

Cross-cutting concerns or gateway offloading
Authentication and authorization
Service discovery integration
Response caching
Retry policies, circuit breaker, and QoS
Rate limiting and throttling
Load balancing
Logging, tracing, correlation
Headers, query strings, and claims transformation
IP allowlisting

Ocelot API Gateway

Ocelot is basically a set of middlewares that you can apply in a specific order.

Ocelot is a lightweight API Gateway, recommended for simpler approaches. Ocelot is an Open Source .NET Core-based API Gateway especially made for microservices architectures that need unified points of entry into their systems. It’s lightweight, fast, and scalable and provides routing and authentication among many other features.



The main reason to choose Ocelot for our reference application is because Ocelot is a .NET Core lightweight API Gateway that you can deploy into the same application deployment environment where you're deploying your microservices/containers, such as a Docker Host, Kubernetes, etc. And since it's based on .NET Core, it's cross-platform allowing you to deploy on Linux or Windows.
Ocelot is designed to work with ASP.NET Core only. You install Ocelot and its dependencies in your ASP.NET Core project with Ocelot's NuGet package, from Visual Studio.

Analysis & Design

This project will be the REST APIs which basically perform Routing operations on Catalog, Basket and Ordering microservices.

We should define our Ocelot API Gateway use case analysis.

Our main use cases;

- Route Catalog APIs with /Catalog path
- Route Basket APIs with / Basket path
- Route Discount APIs with / Discount path
- Route Ordering APIs with /Ordering path

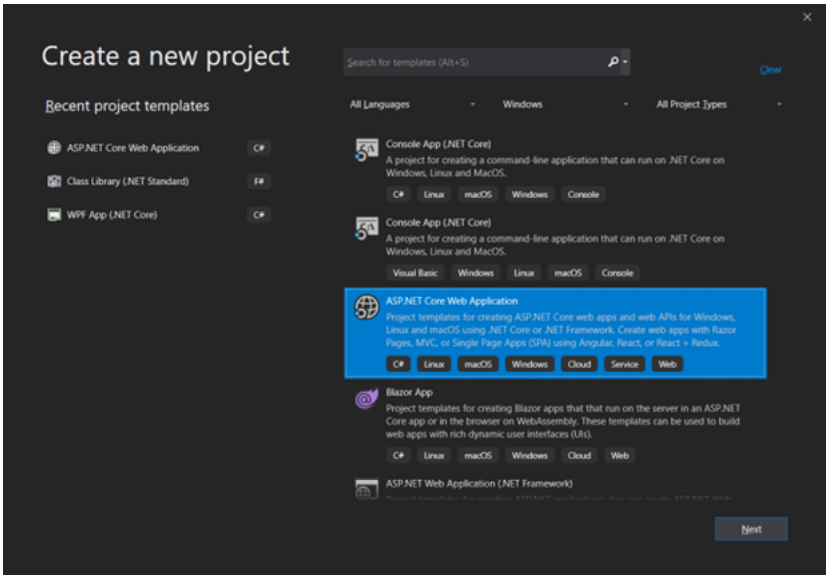
Along with this we should design our APIs according to REST perspective.

Method	Request URI	Use Case
GET/POST	/Catalog	Route /api/v1/Catalog apis
GET	/Catalog/{id}	Route /api/v1/Catalog apis
GET/POST	/Basket	Basket /api/v1/Basket apis
POST	/Basket/Checkout	Basket /api/v1/Basket apis
GET	/Order	Order /api/v1/Order apis

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.

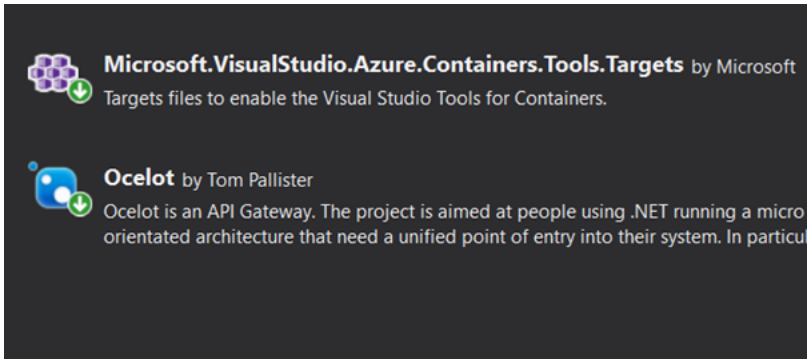
Add New **Blank Web** project under below location and name;

src/ApiGateway/APIGateway

Library & Frameworks

For API Gateway microservices, we have to libraries in our Nuget Packages,

- Ocelot — API Gateway library



Configure Ocelot in Startup.cs

In order to configure and use Ocelot in Asp.Net Core project, we should define Ocelot methods into **Startup.cs.**

Go to Class -> Startup.cs

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddOcelot();
        }// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public async void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }app.UseRouting();app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });//ocelot
            await app.UseOcelot();
        }
```

Configuration Json File Definition of Ocelot

In order to use routing function of Ocelot, we should give the **configuration json** file when Asp.Net Web application start.

Go to Class -> Program.cs

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                 .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config.AddJsonFile($"ocelot.{hostingContext.HostingEnvironment.EnvironmentName}.json", true, true);
                })
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
```

We have added configuration file according to Environment name.
Since I changed the environment variable as a Local, application will pick the ocelot.Local.json configurations.

Ocelot.Local.json File Routing API

In order to use routing we sould create Ocelot.Local.json and put json objects.

The important part here is that each element we define in the **Routes** series represents a service.

In the **DownstreamPathTemplate** field, we define the url directory of the api application. In **UpstreamPathTemplate**, we specify which path the user writes to this api url.

Create **Ocelot.Local.json** file.

```
{
   "Routes": [
     //Catalog API
     {
       "DownstreamPathTemplate": "/api/v1/Catalog",
       "DownstreamScheme": "http",
       "DownstreamHostAndPorts": [
         {
           "Host": "localhost",
           "Port": "8000"
         }
       ],
       "UpstreamPathTemplate": "/Catalog",
       "UpstreamHttpMethod": [ "GET", "POST", "PUT" ]
     },
     {
       "DownstreamPathTemplate": "/api/v1/Catalog/{id}",
       "DownstreamScheme": "http",
       "DownstreamHostAndPorts": [
         {
           "Host": "localhost",
           "Port": "8000"
         }
       ],
       "UpstreamPathTemplate": "/Catalog/{id}",
       "UpstreamHttpMethod": [ "GET", "DELETE" ]
     },
     {
       "DownstreamPathTemplate": "/api/v1/Catalog/GetProductByCategory/{category}",
       "DownstreamScheme": "http",
       "DownstreamHostAndPorts": [
         {
           "Host": "localhost",
           "Port": "8000"
         }
       ],
       "UpstreamPathTemplate": "/Catalog/GetProductByCategory/{category}",
       "UpstreamHttpMethod": [ "GET" ]
     },
     //Basket API
     {
       "DownstreamPathTemplate": "/api/v1/Basket/{userName}",
       "DownstreamScheme": "http",
       "DownstreamHostAndPorts": [
         {
           "Host": "localhost",
           "Port": "8001"
         }
       ],
       "UpstreamPathTemplate": "/Basket/{userName}",
       "UpstreamHttpMethod": [ "GET", "DELETE" ]
     },
     {
       "DownstreamPathTemplate": "/api/v1/Basket",
       "DownstreamScheme": "http",
       "DownstreamHostAndPorts": [
         {
           "Host": "localhost",
           "Port": "8001"
         }
       ],
       "UpstreamPathTemplate": "/Basket",
       "UpstreamHttpMethod": [ "POST" ]
     },
     {
```

```
      "DownstreamPathTemplate": "/api/v1/Basket/Checkout",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "8001"
        }
      ],
      "UpstreamPathTemplate": "/Basket/Checkout",
      "UpstreamHttpMethod": [ "POST" ]
    },
    //Discount API
    {
      "DownstreamPathTemplate": "/api/v1/Discount/{productName}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "8002"
        }
      ],
      "UpstreamPathTemplate": "/Discount/{productName}",
      "UpstreamHttpMethod": [ "GET", "DELETE" ]
    },
    {
      "DownstreamPathTemplate": "/api/v1/Discount",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "8002"
        }
      ],
      "UpstreamPathTemplate": "/Discount",
      "UpstreamHttpMethod": [ "PUT", "POST" ]
    },
    //Order API
    {
      "DownstreamPathTemplate": "/api/v1/Order/{userName}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "8004"
        }
      ],
      "UpstreamPathTemplate": "/Order/{userName}",
      "UpstreamHttpMethod": [ "GET" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5010"
  }
}
```

These **routes definition** provide to open api to outside of the system and redirect these request into internal api calls.
Of course these requests will have token so Ocelot api gateway also carry this token when calling to internal systems.
Also as you can see that we had summarized the api calls remove api path and use only /Catalog path when exposing apis for the client application.

We have Develop and configured our Ocelot Api Gateway Microservices with seperating environment configurations.

Run Application

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

Before we start, make sure that docker-compose microservices running properly. Start Docker environment;

```
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 docker urls :

Catalog
http://localhost:8000/swagger/index.html
Basket
http://localhost:8001/swagger/index.html
Discount
http://localhost:8002/swagger/index.html
Ordering
http://localhost:8004/swagger/index.html

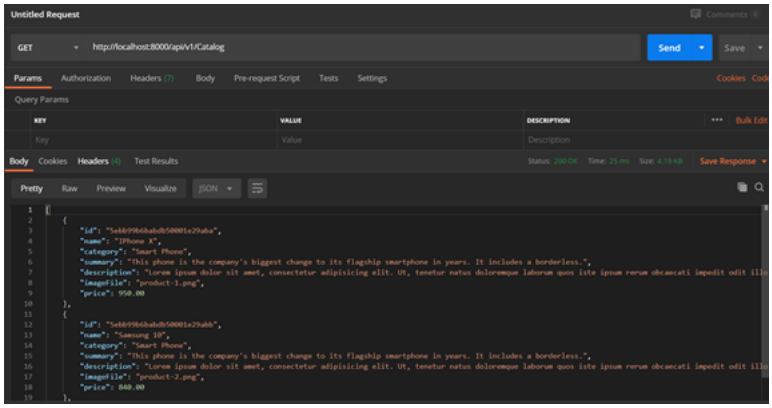
Before running the application, **configure the debug profile**;

Right Click the project File and Select to Debug section.

Change the App URL to **http://localhost:7000**

Hit **F5** on **APIGateway** project.

Exposed the APIGateways in our Microservices, you can **test** it over the chrome or **Postman**.



Rate Limiting in Ocelot Api Gateway with Configuring Ocelot.json File

We are going to do Rate Limiting in Ocelot Api Gateway with Configuring Ocelot.json File.

Rate Limiting Ocelot
https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html

— Example configuration

```
“RateLimitOptions”: {
 “ClientWhitelist”: [],
 “EnableRateLimiting”: true,
 “Period”: “5s”,
 “PeriodTimespan”: 1,
 “Limit”: 1
}
```

After that we can try on Catalog route configuration.

EDIT Get Catalog

```
“Routes”: [
 //Catalog API
 {
 “DownstreamPathTemplate”: “/api/v1/Catalog”,
 “DownstreamScheme”: “http”,
 “DownstreamHostAndPorts”: [
 {
 “Host”: “localhost”,
 “Port”: “8000”
 }
 ],
 “UpstreamPathTemplate”: “/Catalog”,
 “UpstreamHttpMethod”: [ “GET”, “POST”, “PUT” ],
 “RateLimitOptions”: {
 “ClientWhitelist”: [],
 “EnableRateLimiting”: true,
 “Period”: “5s”,
 “PeriodTimespan”: 1,
 “Limit”: 1
 }
 },
```

— Test on Postman

GET
http://localhost:5010/Catalog

Second Call ERROR
API calls quota exceeded! maximum admitted 1 per 5s.

Response Caching in Ocelot Api Gateway with Configuring Ocelot.json File

We are going to do response Caching in Ocelot Api Gateway with Configuring Ocelot.json File.

Response Caching
https://ocelot.readthedocs.io/en/latest/features/caching.html

— Example configuration
```
“FileCacheOptions”: { “TtlSeconds”: 30 }
```

After that, we should add required nuget package;

Add Nuget Package
```
Install-Package Ocelot.Cache.CacheManager
```

Modify DI Ocelot

Startup.cs
```
public void ConfigureServices(IServiceCollection services)
 {
 services.AddOcelot().AddCacheManager(settings => settings.WithDictionaryHandle()); – CHANGED !!
 }
```

Add Configuration ocelot.json

— how many seconds keep cache
```
“FileCacheOptions”: { “TtlSeconds”: 30 }
```

— UPDATE Catalog API
```
“Routes”: [
 //Catalog API
 {
 “DownstreamPathTemplate”: “/api/v1/Catalog”,
 “DownstreamScheme”: “http”,
 “DownstreamHostAndPorts”: [
 {
 “Host”: “localhost”,
 “Port”: “8000”
 }
 ],
 “UpstreamPathTemplate”: “/Catalog”,
 “UpstreamHttpMethod”: [ “GET”, “POST”, “PUT” ],
 “FileCacheOptions”: { “TtlSeconds”: 30 } – – – – – – ADDED
 },
```

30 second cache.

— Second call will comes from cache.

As you can see that, we have perform response caching in Ocelot Api Gateway with Configuring Ocelot.json File.

Configure Ocelot Json For Docker Development Environment in Ocelot Api Gateway

We are going to Configure Ocelot Json For Docker Development Environment in Ocelot Api Gateway.

Before I am adding to ocelot.Development.json file, let me clarify our environments.
— When we create any aspnet web application, you can see the environment value is Local

— Change env is Development

Right Click OcelotApiGw
Properties
Debug
ASPNETCORE_ENVIRONMENT = Local
change
ASPNETCORE_ENVIRONMENT = Development

Also if we check the docker-compose override file, for every configuration we set **ASPNETCORE_ENVIRONMENT=Development.**
```
ocelotapigw:
 container_name: ocelotapigw
 environment:
 – ASPNETCORE_ENVIRONMENT=Development
```

That’s why we can say that our Development environment should be full docker environment with docker names of the containers.
We only configured for our local environment and tested successfully.
So now we are going to configure Ocelot for Docker Development environment in order to Run and Debug Ocelot Api Gw on docker environment.

ocelot.Development.json
```
{
 “Routes”: [
 //Catalog API
 {
```

```
“DownstreamPathTemplate”: “/api/v1/Catalog”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “catalog.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Catalog”,
“UpstreamHttpMethod”: [ “GET”, “POST”, “PUT” ],
“FileCacheOptions”: { “TtlSeconds”: 30 }
},
{
“DownstreamPathTemplate”: “/api/v1/Catalog/{id}”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “catalog.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Catalog/{id}”,
“UpstreamHttpMethod”: [ “GET”, “DELETE” ]
},
{
“DownstreamPathTemplate”: “/api/v1/Catalog/GetProductByCategory/{category}”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “catalog.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Catalog/GetProductByCategory/{category}”,
“UpstreamHttpMethod”: [ “GET” ]
},
//Basket API
{
“DownstreamPathTemplate”: “/api/v1/Basket/{userName}”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “basket.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Basket/{userName}”,
“UpstreamHttpMethod”: [ “GET”, “DELETE” ]
},
{
“DownstreamPathTemplate”: “/api/v1/Basket”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “basket.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Basket”,
“UpstreamHttpMethod”: [ “POST” ]
},
{
“DownstreamPathTemplate”: “/api/v1/Basket/Checkout”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “basket.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Basket/Checkout”,
“UpstreamHttpMethod”: [ “POST” ],
“RateLimitOptions”: {
“ClientWhitelist”: [],
“EnableRateLimiting”: true,
“Period”: “3s”,
“PeriodTimespan”: 1,
“Limit”: 1
}
},
//Discount API
{
“DownstreamPathTemplate”: “/api/v1/Discount/{productName}”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “discount.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Discount/{productName}”,
“UpstreamHttpMethod”: [ “GET”, “DELETE” ]
},
{
“DownstreamPathTemplate”: “/api/v1/Discount”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
```

```
“Host”: “discount.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Discount”,
“UpstreamHttpMethod”: [ “PUT”, “POST” ]
},
//Order API
{
“DownstreamPathTemplate”: “/api/v1/Order/{userName}”,
“DownstreamScheme”: “http”,
“DownstreamHostAndPorts”: [
{
“Host”: “ordering.api”,
“Port”: “80”
}
],
“UpstreamPathTemplate”: “/Order/{userName}”,
“UpstreamHttpMethod”: [ “GET” ]
}
],
“GlobalConfiguration”: {
“BaseUrl”: “http://localhost:5010”
}
}
```

As you can see that, we have Configured Ocelot Json For Docker Development Environment in Ocelot Api Gateway.

Run Application on Docker with Database

Since here we developed ASP.NET Core Web project for APIGateway microservices. Now it’s time to make **docker** the **APIGateway** project with our **existing microservices** image.

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 Ocelot 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:
  ocelotapigw:
    image: ${DOCKER_REGISTRY-}ocelotapigw
    build:
      context: .
      dockerfile: ApiGateways/OcelotApiGw/Dockerfile
```

Docker-compose.override.yml

```
version: ‘3.4’services:
    ocelotapigw:
    container_name: ocelotapigw
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
    depends_on:
        - catalog.api
        - basket.api
        - discount.api
        - ordering.api
    ports:
      - "8010:80"
```

Basically in **docker-compose.yml file**, created one image for **apigateway**.

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 :

Let me test Ocelot Api Gw Routing Features in docker environment

TEST OVER OCELOT

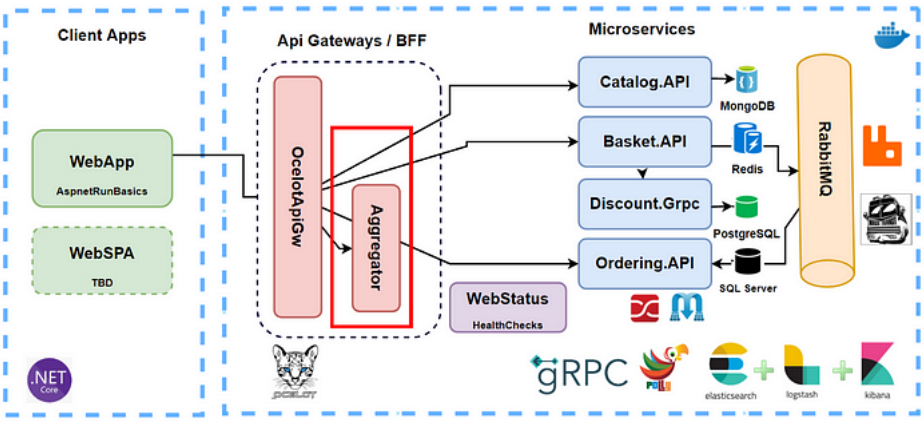
Test
Open Postman
Catalog

```
GET
http://localhost:8010/Catalog
http://localhost:8010/Catalog/6022a3879745eb1bf118d6e2
http://localhost:8010/Catalog/GetProductByCategory/Smart PhoneBasket
GET
http://localhost:8010/Basket/swnPOST
http://localhost:8010/Basket
{
“UserName”: “swn”,
“Items”: [
{
“Quantity”: 2,
“Color”: “Red”,
“Price”: 33,
“ProductId”: “5”,
“ProductName”: “5”
},
{
“Quantity”: 1,
“Color”: “Blue”,
“Price”: 55,
“ProductId”: “3”,
“ProductName”: “5”
}
]
}POST
http://localhost:8010/Basket/Checkout{
“userName”: “swn”,
“totalPrice”: 0,
“firstName”: “swn”,
“lastName”: “swn”,
“emailAddress”: “string”,
“addressLine”: “string”,
“country”: “string”,
“state”: “string”,
“zipCode”: “string”,
“cardName”: “string”,
“cardNumber”: “string”,
“expiration”: “string”,
“cvv”: “string”,
“paymentMethod”: 1
}See from Rabbit Management Dashboard
http://localhost:15672DELETE
http://localhost:8010/Basket/swnDiscount
GET
http://localhost:8010/Discount/IPhone XDELETE
http://localhost:8010/Discount/IPhone XOrder
GET
http://localhost:8010/Order/swn
```

As you can see that, we have tested Ocelot API Gw with Routing features on docker environment.

Develop Shopping.Aggregator microservices with Applying Gateway Aggregation Pattern

We are going to Develop Shopping.Aggregator microservices with Applying Gateway Aggregation Pattern.



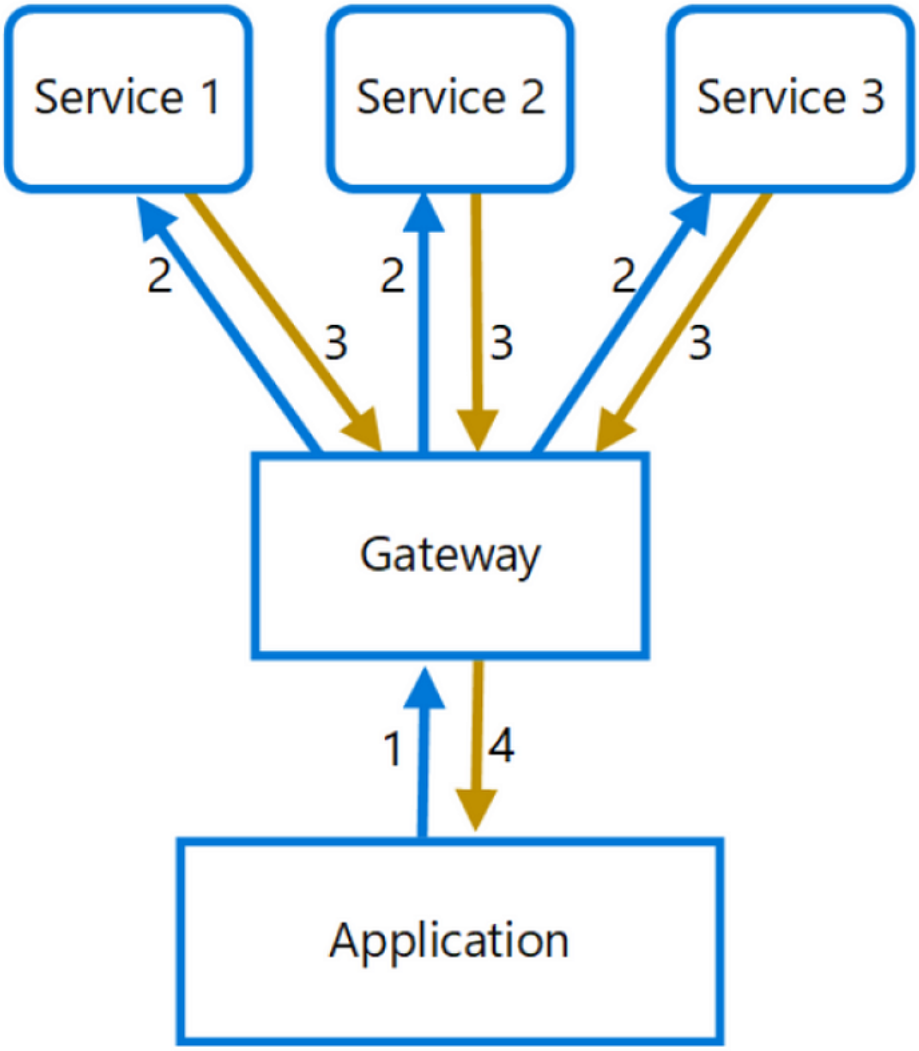
Shopping.Aggregator Microservice cover the;

- Aggregate multiple client requests using **HTTP Client Factory**
- Targeting **multiple internal microservices** into a single client request
- Client app sends a single request to the API Gateway that dispatches several requests to the internal microservices
- Then **aggregates** the results and sends everything back to the client app

We are going to routing operations over the Catalog, Basket and Ordering microservices. Client only send the username of 1 api exposing from shopping.aggregetor microservices.
Reduce chattiness between the client apps and the backend API
Implement the Gateway aggregation pattern in Shopping.Aggregator
Similar to Custom api Gateway implementation.

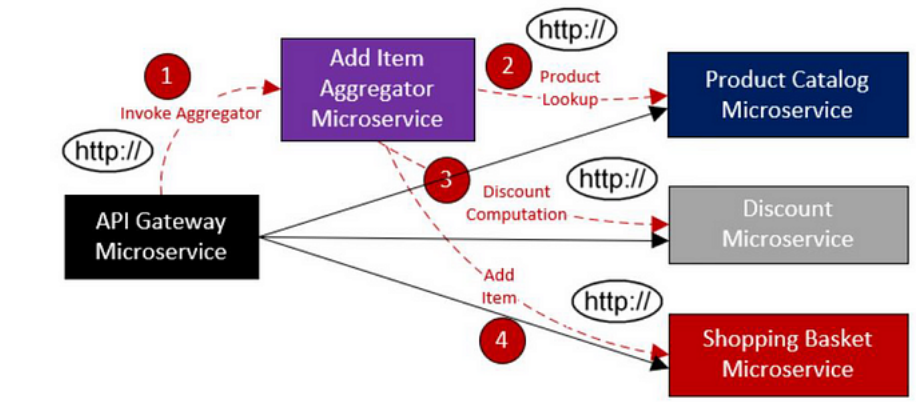
The Gateway Aggregation pattern

Use a gateway to aggregate multiple individual requests into a single request.
This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.



To perform a single task, a client may have to make multiple calls to various backend services. An application that relies on many services to perform a task must expend resources on each request. When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls.
This chattiness between a client and a backend can adversely impact the performance and scale of the application. Microservice architectures have made this problem more common, as applications built around many smaller services naturally have a higher amount of cross-service calls.

As a Solution, use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.



As you can see in this picture, We are going to develop Shopping.Aggregator Microservices with implementing Gateway Aggregation pattern.

This **Shopping.Aggregator Microservices** expose only 1 api to the client applications with taking user Name information. And Aggregate multiple client requests with consuming **Catalog, Basket** and **Ordering** internal microservices.

Client app sends a single request to the API Gateway that dispatches several requests to the internal microservices. Then aggregates the results and sends everything back to the client app. We are going to routing operations over the Catalog, Basket and Ordering microservices.
Client only send the username of 1 api exposing from shopping.aggragetor microservices.

This will Reduce chattiness between the client apps and the backend API.

Developing Service Classes for Consuming Internal Microservices in Shopping.Aggreation Microservice

We are going to Develop Service Classes for Consuming Internal Microservices in Shopping.Aggreation.

Before we start, remember that we have added url configurations in appsettings json file.

Addding Configurations

```
“ApiSettings”: {
 “CatalogUrl”: “http://localhost:8001",
 “BasketUrl”: “http://localhost:8002",
 “OrderingUrl”: “http://localhost:8004"
 },
```

Create Services Folder — for consuming apis

Create folder
Services

Add service interfaces;

```
ICatalogServicepublic interface ICatalogService
 {
 Task<IEnumerable<CatalogModel>> GetCatalog();
 Task<IEnumerable<CatalogModel>> GetCatalogByCategory(string category);
 Task<CatalogModel> GetCatalog(string id);
 }IBasketServicepublic interface IBasketService
 {
 Task<BasketModel> GetBasket(string userName);
 }IOrderServicepublic interface IOrderService
 {
 Task<IEnumerable<OrderResponseModel>> GetOrdersByUserName(string userName);
 }
```

Add service Implementation classes but no implementation only write methods with empty bodies.

```
CatalogService

 public class CatalogService : ICatalogService
 {
 private readonly HttpClient _client;public CatalogService(HttpClient client)
 {
 _client = client ?? throw new ArgumentNullException(nameof(client));
 }
```

In this service classes we need to consume apis. So we need **HttpClient**. In order to use it, inject **HttpClient** object with Client Factory in Startup.cs

Register HttpClient to AspNet DI
We are going to register HttpClient definitions in aspnet built-in dependency injection.
With AddHttpClient method we can pass the type objects as a generic tpyes. By this way httpclient factory manange httpclient creation operation inside of this

types.
This is called type-based **HttpClient Factory** registiration.

Startup.cs;

```
services.AddHttpClient<ICatalogService, CatalogService>(c =>
 c.BaseAddress = new Uri(Configuration[“ApiSettings:CatalogUrl”]));services.AddHttpClient<IBasketService, BasketService>(c =>
 c.BaseAddress = new Uri(Configuration[“ApiSettings:BasketUrl”]));services.AddHttpClient<IOrderService, OrderService>(c =>
 c.BaseAddress = new Uri(Configuration[“ApiSettings:OrderingUrl”]));
```

We have registered 3 microservices integrations with giving base addresses
We have registered service classes and http client object will manage by factory by this type configuration dependency.

Now ready to Implementation Service Classes;

```
CatalogServicepublic class CatalogService : ICatalogService
 {
 private readonly HttpClient _client;public CatalogService(HttpClient client)
 {
 _client = client ?? throw new ArgumentNullException(nameof(client));
 }public async Task<IEnumerable<CatalogModel>> GetCatalog()
 {
 var response = await _client.GetAsync(“/api/v1/Catalog”);
 return await response.ReadContentAs<List<CatalogModel>>();
 }public async Task<CatalogModel> GetCatalog(string id)
 {
 var response = await _client.GetAsync($”/api/v1/Catalog/{id}”);
 return await response.ReadContentAs<CatalogModel>();
 }public async Task<IEnumerable<CatalogModel>> GetCatalogByCategory(string category)
 {
 var response = await _client.GetAsync($”/api/v1/Catalog/GetProductByCategory/{category}”);
 return await response.ReadContentAs<List<CatalogModel>>();
 }
 }BasketServicepublic class BasketService : IBasketService
 {
 private readonly HttpClient _client;public BasketService(HttpClient client)
 {
 _client = client ?? throw new ArgumentNullException(nameof(client));
 }public async Task<BasketModel> GetBasket(string userName)
 {
 var response = await _client.GetAsync($”/api/v1/Basket/{userName}”);
 return await response.ReadContentAs<BasketModel>();
 }
 }OrderServicepublic class OrderService : IOrderService
 {
 private readonly HttpClient _client;public OrderService(HttpClient client)
 {
 _client = client ?? throw new ArgumentNullException(nameof(client));
 }public async Task<IEnumerable<OrderResponseModel>> GetOrdersByUserName(string userName)
 {
 var response = await _client.GetAsync($”/api/v1/Order/{userName}”);
 return await response.ReadContentAs<List<OrderResponseModel>>();
 }
 }
```

After developed Service classes, we should expose api with creating Controller class.

Develop Controller for Shopping Aggregator

ShoppingController

```
[ApiController]
 [Route(“api/v1/[controller]”)]
 public class ShoppingController : ControllerBase
 {
 private readonly ICatalogService _catalogService;
 private readonly IBasketService _basketService;
 private readonly IOrderService _orderService;public ShoppingController(ICatalogService catalogService, IBasketService basketService, IOrderService or
 {
 _catalogService = catalogService ?? throw new ArgumentNullException(nameof(catalogService));
 _basketService = basketService ?? throw new ArgumentNullException(nameof(basketService));
 _orderService = orderService ?? throw new ArgumentNullException(nameof(orderService));
 }[HttpGet(“{userName}”, Name = “GetShopping”)]
 [ProducesResponseType(typeof(ShoppingModel), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<ShoppingModel>> GetShopping(string userName)
 {
 var basket = await _basketService.GetBasket(userName);foreach (var item in basket.Items)
 {
 var product = await _catalogService.GetCatalog(item.ProductId);// set additional product fields
 item.ProductName = product.Name;
 item.Category = product.Category;
 item.Summary = product.Summary;
 item.Description = product.Description;
 item.ImageFile = product.ImageFile;
 }var orders = await _orderService.GetOrdersByUserName(userName);var shoppingModel = new ShoppingModel
 {
 UserName = userName,
 BasketWithProducts = basket,
 Orders = orders
 };

 return Ok(shoppingModel);
 }}
```

As you can see that, first we call basket microservices, in order to get basket information with a given username.

- After that we call catalog microservices for every basket item into the basket and get product detail information and enrich our aggregator basket item data with product informations.
- Lastly we call Order microservices to get user existing orders
- Finally return to aggregated ShoppningModel to the client.

You can create your own **BFF — backend for frontend microservices** for aggreagotr or api gateway operations.
These are 2 design patterns

- **Gateway Aggregator**
- **Gateway Routing**

So that means you can create your custom api gateway with the same steps of **Shopping.Aggreation Microservices.**