Université Saint Joseph
École supérieure d’ingénieurs de Beyrouth (ESIB)

3^eme CCE

Applications Integration at the Enterprise

Prepared by:

ABI KHALIL Melody (160144)
EL KHOURY Alain (152067)

Table of Contents

I. Abstract 1
II. Introduction 1
III. Microservices 2
1- What is Monolithic Architecture? 2
2- What is Microservices Architecture? 2
3- Advantages & benefits 3
IV. Microservices in C# 4
1- .NET Core 4
2- C# 4
3- MS SQL Server 4
4- Redis 5
V. Swagger 6
VI. Docker 7
1- Containerization 7
2- Docker 7
3- Comparing Containers and Virtual Machines 8
4- Fundamental Docker concepts 9
VII. RabbitMQ 11
1- Message Broker 11
2- What is RabbitMQ? 12
VIII. Project implementation 13
1- Catalog Microservice 13
2- Basket microservice 14
3- Ordering microservice 15
4- How we built the microservices 16
5- API Gateway 21
6- Swagger 23
IX. Steps to run the project 24
X. Results & Conclusion 24
X. References 9

Table of Figures

Figure 1: Picture from Microsoft Docs showing the microservices architecture style 2
Figure 2: VM Schema 8
Figure 3: Containerized Applications schema 8
Figure 4: Fundamental parts of Docker 9
Figure 5: Docker schema 10
Figure 6: Schema showing the role of a Message broker 11
Figure 7: Screenshot from Swagger UI for the Catalog service 14
Figure 8:Screenshot from Swagger UI for the Basket service 14
Figure 9: Screenshot from Swagger UI for the Ordering service 15
Figure 10: screenshot of adding a singleton service 16
Figure 11: Screenshot of adding CORS middleware 16
Figure 12: Screenshot from OrderingContext class 18
Figure 13: Screenshot of OrderCompletedEvent class 19
Figure 14: Screenshot of docker-compose file 20
Figure 15: Screenshot to add configuration for API Gateway 21
Figure 16: Screenshot showing ReRoutes 22
Figure 17: Screenshot of adding AddSwaggerGen 23
Figure 18: Screenshot of Swagger configuration for Catalog microservice 23
Figure 19: Screenshot from Swagger of GET catalog item 24
Figure 20: Screenshot from Swagger of POST catalog item 25

I. Abstract

This purpose of the project is to develop a microservice architecture based application using .NET Core (C#) and Docker.

This project is a proof of concept related to developing Microservices in C# and Docker.

II. Introduction

The application is a shoe E-Commerce website (like Amazon) that is implemented as a multi-container application. Each container is a microservice developed using .NET Core.

The application consists of 3 microservices:

1. Catalog microservice
2. Ordering microservice
3. Basket microservice

Also, there is an API gateway as an entry point to the internal microservices.

III. Microservices

1- What is Monolithic Architecture?

Monolithic means composed all in one piece. A monolithic application is one which is self-contained. All components of the application must be present in order for the code to work.

Take the case of a typical 3-tier traditional web application built in three parts: a user interface, a database, and a server-side application. This server-side application is called a monolith, which is further divided into 3 layers — presentation, business layer, and data layer. The entire code is maintained in the same codebase. In order for the code to work, it is deployed as a single unit. Any small change requires the entire application to be built and deployed.

2- What is Microservices Architecture?

Microservice architecture, or simply microservices, is a distinctive method of developing software systems that tries to focus on building single-function modules with well-defined interfaces and operations. The trend has grown popular in recent years as Enterprises look to become more Agile and move towards a DevOps and continuous testing.

The important point at this stage is that each independent service has a business boundary which can be independently developed, tested, deployed, monitored, and scaled. These can be even developed in different programming languages.

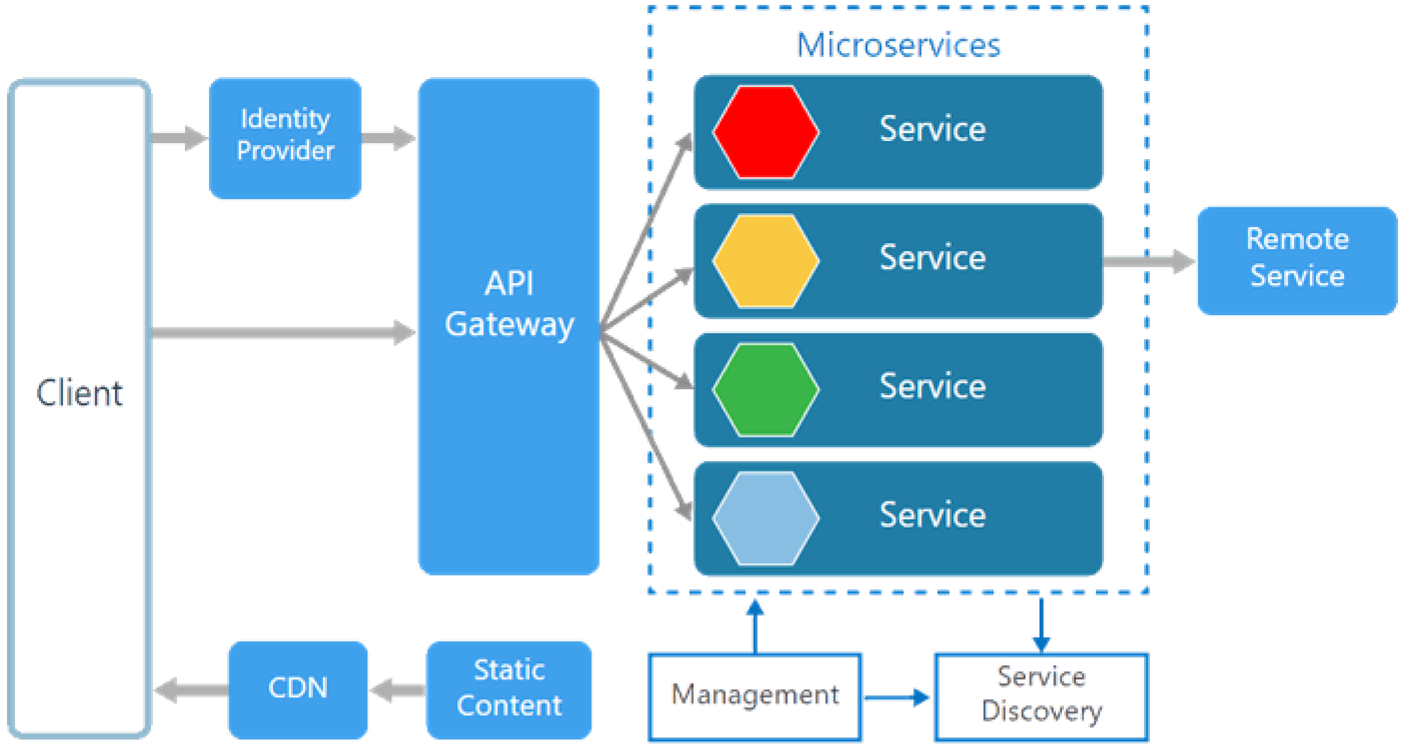


Figure 1: Picture from Microsoft Docs showing the microservices architecture style

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

- Management. Maintains the nodes for the service.
- Identity Provider. Manages the identity information and provides authentication services within a distributed network.
- Service Discovery. Keeps track of services and service addresses and endpoints.
- API Gateway. Serves as client's entry point. Single point of contact from the client which in turn returns responses from underlying microservices and sometimes an aggregated response from multiple underlying microservices.
- CDN. A content delivery network to serve static resources for e.g. pages and web content in a distributed network
- Static Content The static resources like pages and web content

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

3- Advantages & benefits

Microservices have many benefits for Agile and DevOps teams - as Martin Fowler points out, Netflix, eBay, Amazon, Twitter, PayPal, and other tech stars have all evolved from monolithic to microservices architecture. Unlike microservices, a monolith application is built as a single, autonomous unit. This make changes to the application slow as it affects the entire system. A modification made to a small section of code might require building and deploying an entirely new version of software.

Microservices solve these challenges of monolithic systems by being as modular as possible. In the simplest form, they help build an application as a suite of small services, each running in its own process and are independently deployable. These services may be written in different programming languages and may use different data storage techniques. While this results in the development of systems that are scalable and flexible, it needs a dynamic makeover.

Microservices are often connecta via APIs, and can leverage many of the same tools and solutions that have grown in the RESTful and web service ecosystem. Testing these APIs can help validate the flow of data and information throughout the microservice deployment.

In a nutshell, choosing microservices architecture has several benefits:

- Independently develop and deploy services
- Speed and agility
- Better code quality
- Code created/organized around business functionality
- Increased productivity
- Easier to scale
- Freedom (in a way) to choose the implementation technology/language

IV. Microservices in C#

1- .NET Core

.NET Core is a free and open-source, managed computer software framework for Windows, Linux, and macOS operating systems. It is an open source, cross platform successor to .NET Framework. The project is primarily developed by Microsoft and released under the MIT License.

.NET Core fully supports C# and F# and partially supports Visual Basic .NET.

2- C#

C# is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.
It was developed around 2000 by Microsoft as part of its .NET initiative, and later approved as an international standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2018).
C# is one of the programming languages designed for the Common Language Infrastructure (CLI).

C# was designed by Anders Hejlsberg, and its development team is currently led by Mads Torgersen. The most recent version is 8.0, which was released in 2019 alongside Visual Studio 2019 version 16.3.

3- MS SQL Server

Microsoft SQL Server is a relational database management system developed by Microsoft. As a database server, it is a software product with the primary function of storing and retrieving data as requested by other software applications—which may run either on the same computer or on another computer across a network (including the Internet).

a. What is SQL Server?

- It is a software, developed by Microsoft, which is implemented from the specification of RDBMS.
- It is also an ORDBMS.
- It is platform dependent.
- It is both GUI and command based software.
- It supports SQL (SEQUEL) language which is an IBM product, non-procedural, common database and case insensitive language.

b. Usage of SQL Server

- To create databases.
- To maintain databases.
- To analyze the data through SQL Server Analysis Services (SSAS).
- To generate reports through SQL Server Reporting Services (SSRS).
- To carry out ETL operations through SQL Server Integration Services (SSIS).

4- Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

Atomic operations can be running on these types, like appending to a string; incrementing the value in a hash; pushing an element to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

In order to achieve its outstanding performance, Redis works with an in-memory dataset. Depending on the use case, one can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log. Persistence can be optionally disabled, if one just need a feature-rich, networked, in-memory cache.

Redis also supports trivial-to-setup master-slave asynchronous replication, with very fast non-blocking first synchronization, auto-reconnection with partial resynchronization on net split.

Other features include:

-Transactions
-Pub/Sub
-Lua scripting
-Keys with a limited time-to-live
-LRU eviction of keys
-Automatic failover

Redis can be used from most programming languages out there.

V. Swagger

Swagger is an open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful web services. While most users identify Swagger by the Swagger UI tool, the Swagger toolset includes support for automated documentation, code generation, and test-case generation.

Swagger UI allows anyone to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from the OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

Here are some advantages of Swagger:

Dependency Free:
The UI works in any development environment, be it locally or in the web

Human Friendly:
Allow end developers to effortlessly interact and try out every single operation your API exposes for easy consumption

Easy to Navigate:
Quickly find and work with resources and endpoints with neatly categorized documentation

All Browser Support:
Cater to every possible scenario with Swagger UI working in all major browsers

Fully Customizable:
Style and tweak your Swagger UI the way you want with full source code access

Complete OAS Support:
Visualize APIs defined in Swagger 2.0 or OAS 3.0

VI. Docker

1- Containerization

Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image. The containerized application can be tested as a unit and deployed as a container image instance to the host operating system (OS).

Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies. Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.

Containers also isolate applications from each other on a shared OS. Containerized applications run on top of a container host that in turn runs on the OS (Linux or Windows). Containers therefore have a significantly smaller footprint than virtual machine (VM) images

Another benefit of containerization is scalability. You can scale out quickly by creating new containers for short-term tasks. From an application point of view, instantiating an image (creating a container) is similar to instantiating a process like a service or web app. For reliability, however, when you run multiple instances of the same image across multiple host servers, you typically want each container (image instance) to run in a different host server or VM in different fault domains.

In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the whole application lifecycle workflow. The most important benefit is the environment's isolation provided between Dev and Ops.

2- Docker

a. What is Docker?

Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises. Docker is also a company that promotes and evolves this technology, working in collaboration with cloud, Linux, and Windows vendors, including Microsoft.

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

b. What is Docker used for?

Docker is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains. For developers, it means that they can focus on writing code without worrying about the system that it will ultimately be running on. It also allows them to get a head start by using one of thousands of programs already designed to run in a Docker container as a part of their application. For operations staff, Docker gives flexibility and potentially reduces the number of systems needed because of its small footprint and lower overhead.

3- Comparing Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

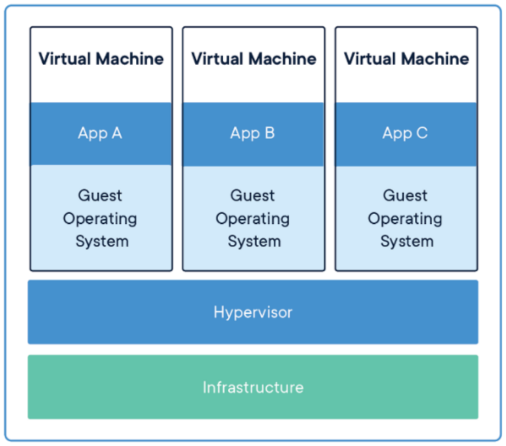


Figure 2: VM Schema

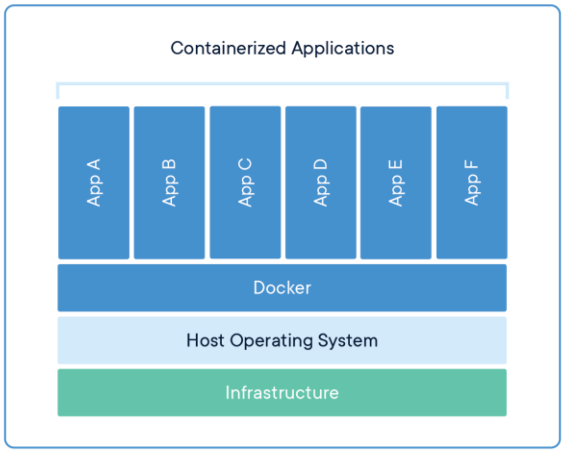


Figure 3: Containerized Applications schema

4- Fundamental Docker concepts

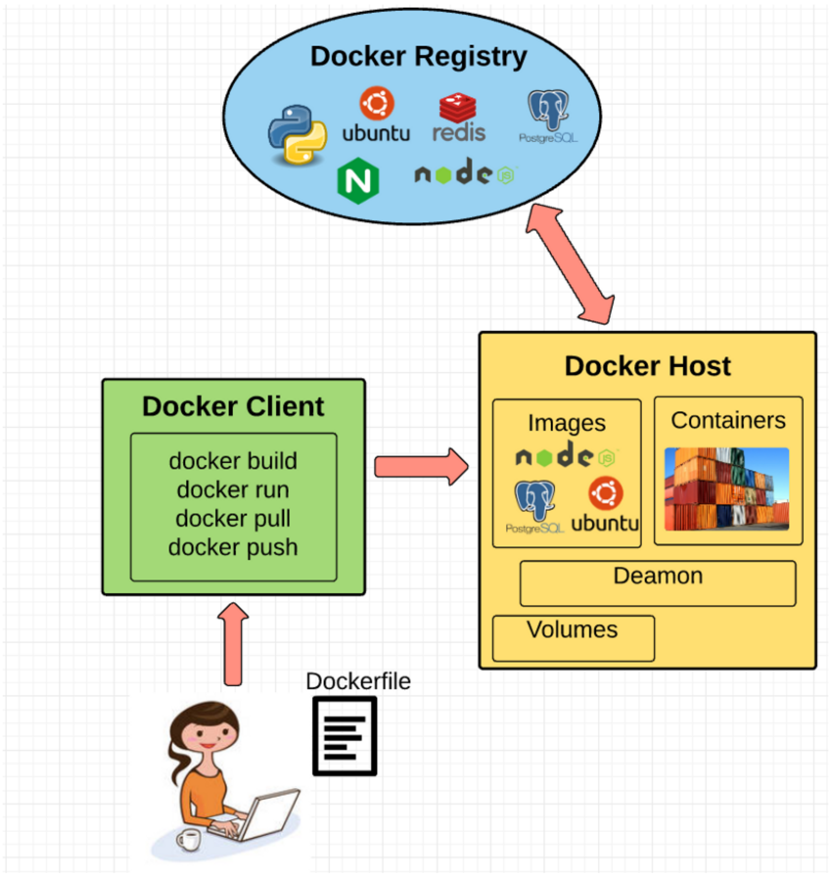


Figure 4: Fundamental parts of Docker

Docker Engine

Docker engine is the layer on which Docker runs. It's a lightweight runtime and tooling that manages containers, images, builds, and more. It runs natively on Linux systems and is made up of:

1. A Docker Daemon that runs in the host computer.
2. A Docker Client that then communicates with the Docker Daemon to execute commands.
3. A REST API for interacting with the Docker Daemon remotely.

Docker Client

The Docker Client is what you, as the end-user of Docker, communicate with. Think of it as the UI for Docker. For example, when you do… You are communicating to the Docker Client, which then communicates your instructions to the Docker Daemon.

Docker Daemon

The Docker daemon is what actually executes commands sent to the Docker Client — like building, running, and distributing your containers. The Docker Daemon runs on the host machine, but as a user, you never communicate directly with the Daemon. The Docker Client can run on the host machine as well, but it's not required to. It can run on a different machine and communicate with the Docker Daemon that's running on the host machine.

Dockerfile

A Dockerfile is where you write the instructions to build a Docker image.
These instructions can be:

• RUN apt-get y install some-package: to install a software package
• EXPOSE 8000: to expose a port
• ENV ANT_HOME /usr/local/apache-ant to pass an environment variable

Docker Image

Images are read-only templates that you build from a set of instructions written in your Dockerfile. Images define both what you want your packaged application and its dependencies to look like *and* what processes to run when it's launched.

The Docker image is built using a Dockerfile. Each instruction in the Dockerfile adds a new "layer" to the image, with layers representing a portion of the images file system that either adds to or replaces the layer below it. Layers are key to Docker's lightweight yet powerful structure.

Volumes

Volumes are the "data" part of a container, initialized when a container is created. Volumes allow you to persist and share a container's data. Even if you destroy, update, or rebuild your container, the data volumes will remain untouched. When you want to update a volume, you make changes to it directly. (As an added bonus, data volumes can be shared and reused among multiple containers, which is pretty neat.)

Docker Containers

A Docker container, as discussed above, wraps an application's software into an invisible box with everything the application needs to run. That includes the operating system, application code, runtime, system libraries, and etc. Docker containers are built off Docker images. Since images are read-only, Docker adds a read-write file system over the read-only file system of the image to create a container.

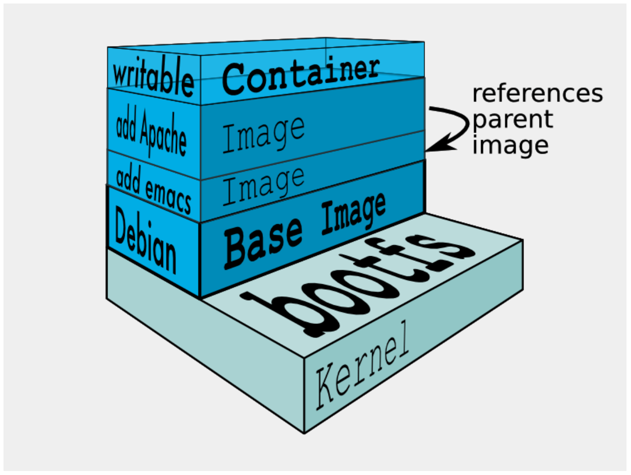


Figure 5: Docker schema

VII. RabbitMQ

1- Message Broker

A message broker is "a program that translates a message to a formal messaging protocol of the sender, to the formal messaging protocol of the receiver"

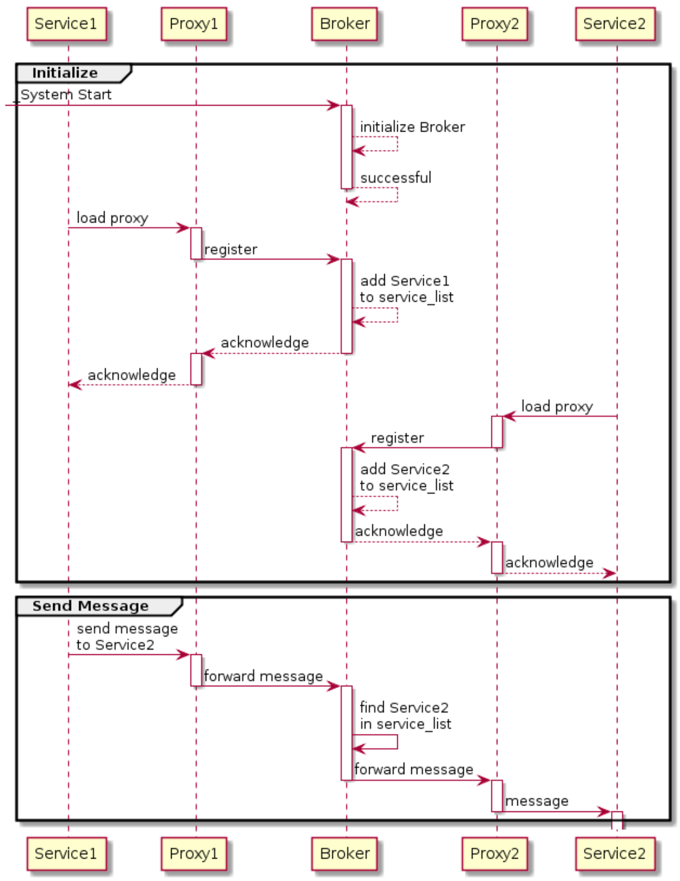


Figure 6: Schema showing the role of a Message broker

The primary purpose of a broker is to take incoming messages from applications and perform some action on them. Message brokers can decouple end-points, meet specific non-functional requirements, and facilitate reuse of intermediary functions. For example, a message broker may be used to manage a workload queue or message queue for multiple receivers, providing reliable storage, guaranteed message delivery and perhaps transaction management.

2- What is RabbitMQ?

RabbitMQ is an open-source message-broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols.

RabbitMQ is also a way to exchange the data between different platform applications such as a message sent from .Net application can be read by a Node.js application or Java application.

The RabbitMQ is built on Erlang general-purpose programming language and it is also used by WhatsApp for messaging.

RabbitMQ is lightweight and easy to deploy on available premises and it supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

RabbitMQ will support multiple operating systems and programming languages. RabbitMQ has provided various client libraries for following programming languages.

- .Net
- Java
- Spring Framework
- Ruby
- Python
- PHP
- Objective-C and Swift
- JavaScript
- GO
- Perl

What is AMQP?

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented and the features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP was designed with the following main characteristics as goals:

- Security
- Reliability
- Interoperability
- Standard
- Open

VIII. Project implementation

ShoesOnContainers is an online market place for shoes, where the user can view and order shoes from different brands.

The application consists of multiple subsystems; its architecture proposes a microservice oriented architecture implementation with multiple autonomous microservices each one owning its own database.

HTTP is the communication protocol between the client and microservices, and asynchronous message based communication between microservices. Message queues are handled with RabbitMQ.

The architecture also includes an implementation of the API Gateway pattern to publish a simplified API to hide the complexity of the internal microservices from the client.

This API Gateway is based on Ocelot and it is deployed as autonomous microservices/containers.

The project was built by following a Udemy course called "ASP.NET Core 2.0 E-Commerce website based on microservices".

The application consists of 3 different microservices:

a. Catalog microservice:
   placing products in shopping cart, updating shopping cart

b. Ordering microservice:
   saving orders

c. Basket microservice:
   saving basket

Each microservice has its own database, and is placed in a separate Docker container.

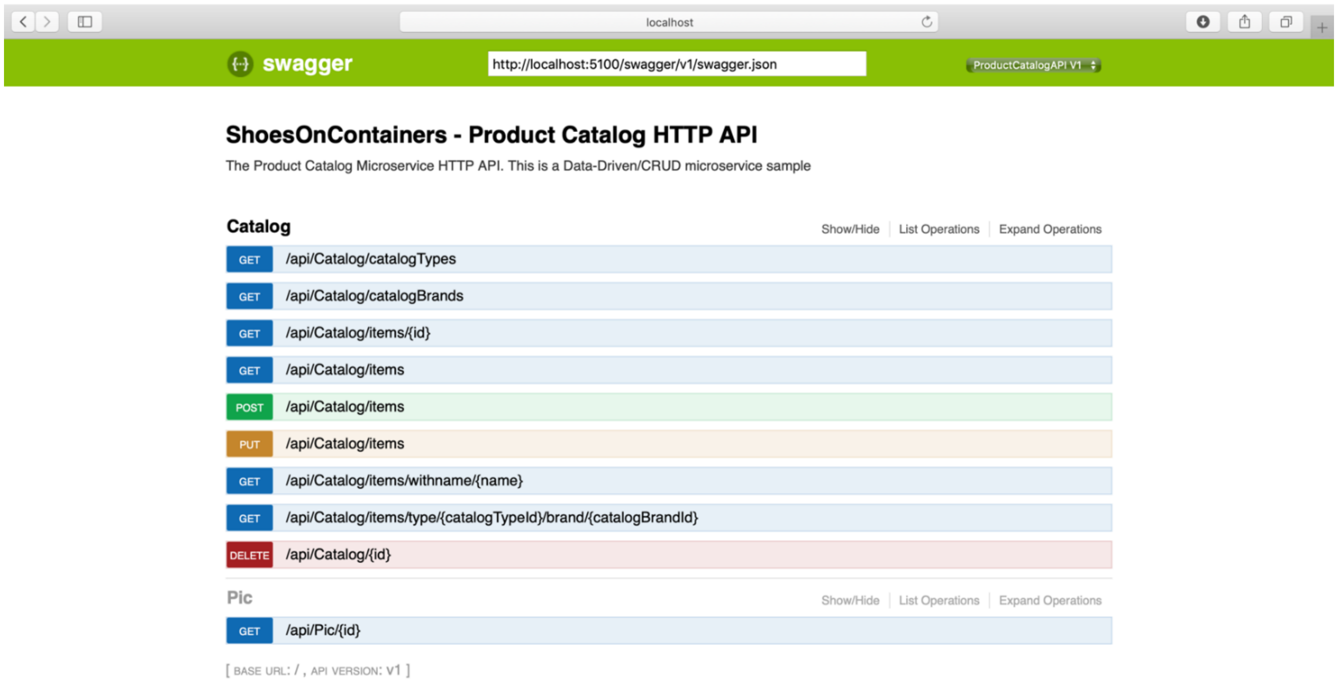
All of the Web APIs of the microservices are documented and the API endpoints are tested via a user interface called Swagger UI.

1- Catalog Microservice

This microservice deals with the catalog items of this online shop.

The job of this microservice is to CREATE, READ, UPDATE or DELETE (CRUD operations) items with the help of its own database.
The database used for this microservice is a SQL database: MSSQL database server from an official Microsoft Linux image using a Docker container.

- Filters item from database based on some criteria.
- Responds with paginated items list.
- Relies on entity framework core for relational object mapping and data access needs.


localhost

swagger http://localhost:5100/swagger/v1/swagger.json ProductCatalogAPI V1

ShoesOnContainers - Product Catalog HTTP API

The Product Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample

Catalog Show/Hide | List Operations | Expand Operations

GET /api/Catalog/catalogTypes
GET /api/Catalog/catalogBrands
GET /api/Catalog/items/{id}
GET /api/Catalog/items
POST /api/Catalog/items
PUT /api/Catalog/items
GET /api/Catalog/items/withname/{name}
GET /api/Catalog/items/type/{catalogTypeId}/brand/{catalogBrandId}
DELETE /api/Catalog/{id}

Pic Show/Hide | List Operations | Expand Operations

GET /api/Pic/{id}

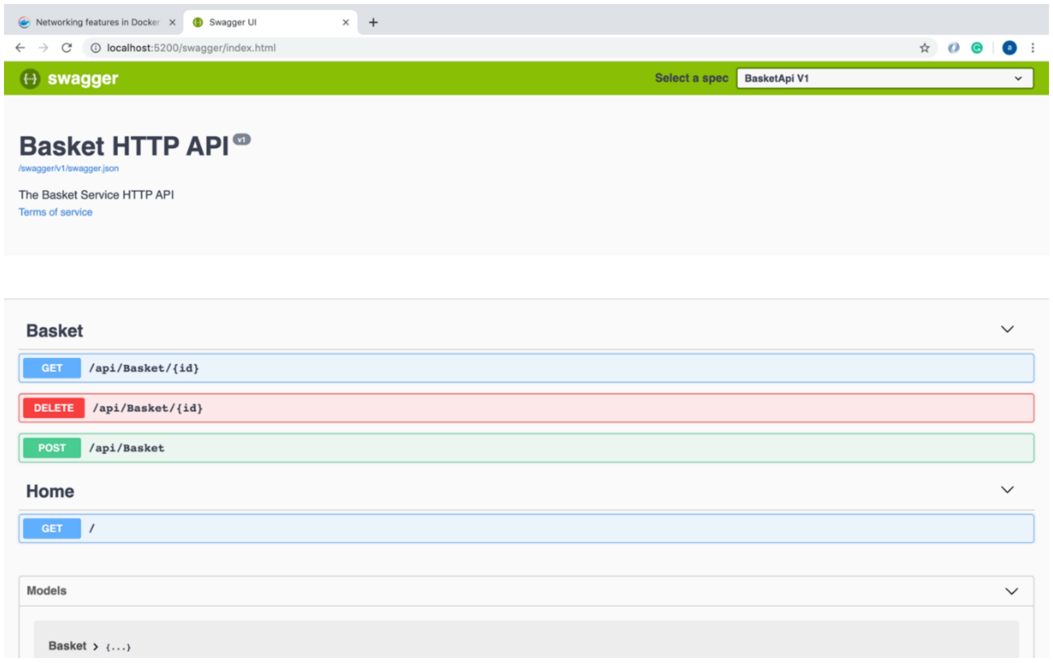
[BASE URL: / , API VERSION: v1]

Figure 7: Screenshot from Swagger UI for the Catalog service

2- Basket microservice

This microservice is responsible for keeping the shopping basket data in a Redis storage rather than a session storage. By using this storage, we will be able to scale the storage without a problem.

- Redis will be running inside a Docker container
- Relies on entity framework core for the relational object mapping and data access needs.
- When the order is completed, a message is published to the RabbitMQ channel and the basket microservice will catch this message and will delete all the products inside the basket


Networking features in Docker | Swagger UI

localhost:5200/swagger/index.html

swagger Select a spec BasketApi V1

Basket HTTP API v1

/swagger/v1/swagger.json

The Basket Service HTTP API

Terms of service

Basket

GET /api/Basket/{id}
DELETE /api/Basket/{id}
POST /api/Basket

Home

GET /

Models

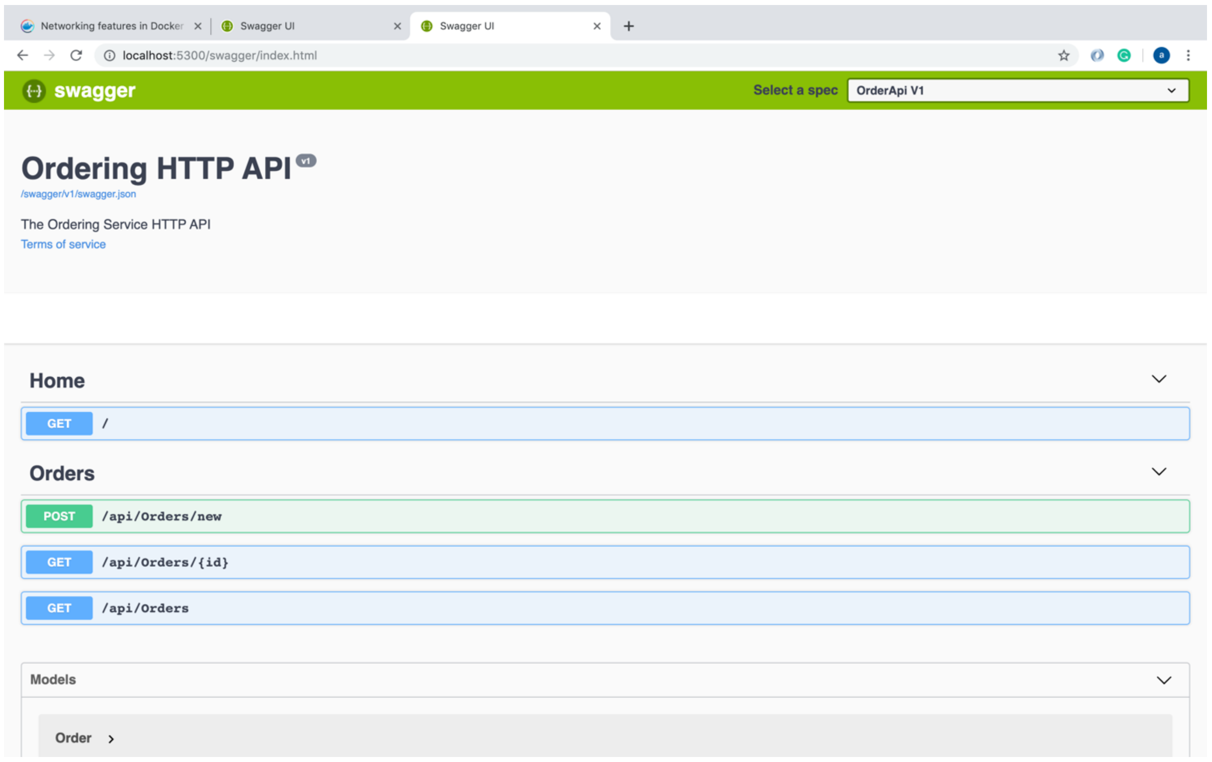
Basket > {...}

Figure 8:Screenshot from Swagger UI for the Basket service

3- Ordering microservice

This microservice is responsible of keeping the orders data in a SQL database (using the same MSSQL database server we used for the Catalog microservice)

- Relies on entity framework core for the relational object mapping and data access needs.
- When the order is completed, we will publish a message to the RabbitMQ channel and the basket microservice will catch this message and will delete all the products inside the basket



Ordering HTTP API

/swagger/v1/swagger.json

The Ordering Service HTTP API

Terms of service

```
GET  /
POST /api/Orders/new
GET  /api/Orders/{id}
GET  /api/Orders
```

Figure 9: Screenshot from Swagger UI for the Ordering service

4- How we built the microservices

a. Basket microservice

When the user adds a product to their basket, this microservice stores these products inside a Redis database.

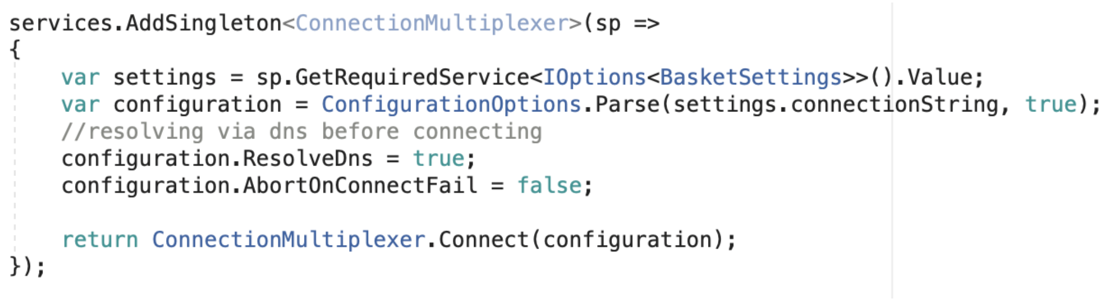
This microservice depends on the following packages:

- "StackExchange.Redis" Version="2.0.601"
- “Autofac.Extensions.DependencyInjection" Version="4.2.0"
- "MassTransit" Version="4.0.1.1381-develop"
- "MassTransit.Autofac" Version="4.0.1.1381-develop"
- "MassTransit.RabbitMQ" Version="4.0.1.1381-develop"

Steps to create the Basket microservice:

1. Singleton service

We need to add a singleton service that represents an inter-related group of connections to Redis servers inside Configureservices method of Startup class.



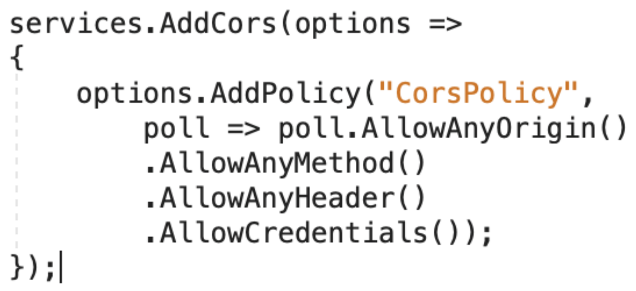
```
services.AddSingleton<ConnectionMultiplexer>(sp =>
{
    var settings = sp.GetRequiredService<IOptions<BasketSettings>>().Value;
    var configuration = ConfigurationOptions.Parse(settings.connectionString, true);
    //resolving via dns before connecting
    configuration.ResolveDns = true;
    configuration.AbortOnConnectFail = false;

    return ConnectionMultiplexer.Connect(configuration);
});
```

Figure 10: screenshot of adding a singleton service

2. Adding cross-origin resource sharing service

We add cross-origin resource sharing service inside Configureservices method of Startup class.



```
services.AddCors(options =>
{
    options.AddPolicy("CorsPolicy",
        poll => poll.AllowAnyOrigin()
        .AllowAnyMethod()
        .AllowAnyHeader()
        .AllowCredentials());
});
```

Figure 11: Screenshot of adding CORS middleware

And then Add this CORS middleware to our web application pipeline to allow cross domain requests, you can do this by using the “UseCors” function of the IApplicationBuilder instance inside Configure method of Startup class.

3. Model creation

Now we need to create our data models.

In this microservice we need to represent two things, the basket item and the basket itself. So we need to create two C# classes, Basket and BasketItem.

Basket:
  -integer: customerId
  -list of BasketItem: items

BasketItem:
  -integer: id
  -integer: productId
  -string: productName
  -decimal: unitPrice
  -integer: quantity

We also need to create an interface (IBasketRepository) that specifies what every database should be able to do.

- Get the basket of any user asynchronously.
- Get a list of users that they have a basket.
- Update any basket asynchronously.
- Delete any basket asynchronously.

4. Redis

At this point, we must create a class that implements the previous interface IBasketRepository. This class uses a Redis database.

5. Creation of APIs

Now we can create our APIs; In ASP.NET Web API, a controller is a class that handles HTTP requests. The public methods of the controller are called action methods or simply actions. When the Web API framework receives a request, it routes the request to an action.

In this microservice we only need one controller to:

- Get basket by Id
- Post a basket (create a new one)
- Delete basket by id

Also we need to inject IBasketRepository instance as a dependency for the BasketController.

b. Ordering microservice

This microservice is responsible for storing information on each order that happens in a SQL database (using an MSSQL database server in a Docker image).

This microservice depends on the following packages:

- “Autofac.Extensions.DependencyInjection" Version="4.2.0"
- "MassTransit" Version="4.0.1.1381-develop"
- "MassTransit.Autofac" Version="4.0.1.1381-develop"
- "MassTransit.RabbitMQ" Version="4.0.1.1381-develop"

<u>Steps to create the ordering microservice:</u>

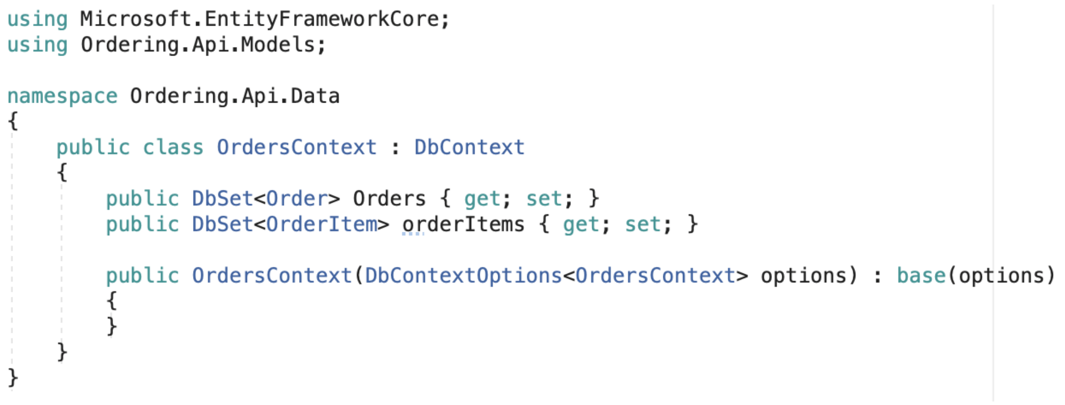
1. Creation of entities

An entity in Entity Framework is a class that maps to a database table, these classes should derive from DBContext class.

We need to store two things, an order and an orderItem (an order has multiple orderItems)

2. Creation of context class

The context class represents a session with the underlying database using which you can perform CRUD operations.



```
using Microsoft.EntityFrameworkCore;
using Ordering.Api.Models;

namespace Ordering.Api.Data
{
    public class OrdersContext : DbContext
    {
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> orderItems { get; set; }

        public OrdersContext(DbContextOptions<OrdersContext> options) : base(options)
        {
        }
    }
}
```

Figure 12: Screenshot from OrderingContext class

3. Registration of context

Now we need to register OrdersContext as a service inside the configureServices method of Startup class.

4. Database migrations

A data model changes during development and gets out of sync with the database. We can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data. The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

As a first step, we need to create a migration. To do that we need to run the following command in order to generate code that can create the database.

```
dotnet ef migrations add InitialMigration -o Data/Mirations -c Ordering.Api.Data.
OrdersContext
```

5. Adding cross-origin resource sharing service

6. Creation of APIs

Now we can create our APIs; In this microservice we only need one controller to:

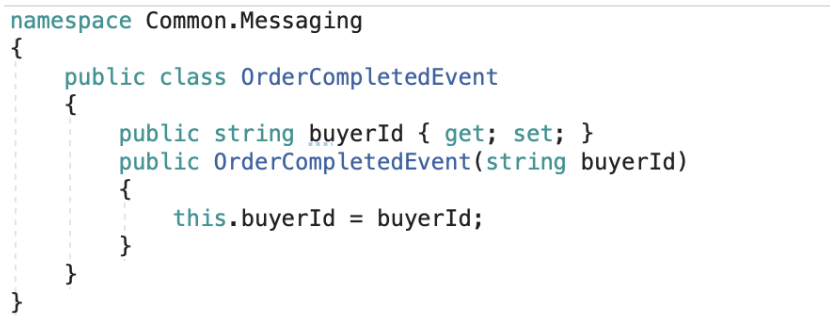
- Get order by Id
- Create an order
- Delete an order by id
- Get all orders

Also we need to inject OrdersContext instance as a dependency for the OrdersController.

7. Messaging

Once a user orders some products, we need to clear their basket. In order to do that, the ordering microservice will send a message to the basket microservice using RabbitMQ.

First, we need to define the message that we want to send; we need the buyerId in order to know which basket we should clear, so the message will contain the buyer Id.



```
namespace Common.Messaging
{
    public class OrderCompletedEvent
    {
        public string buyerId { get; set; }
        public OrderCompletedEvent(string buyerId)
        {
            this.buyerId = buyerId;
        }
    }
}
```

Figure 13: Screenshot of OrderCompletedEvent class

Once we create an order (POST) we will publish a message of type OrderCompletedEvent to all subscribed consumers. In the basket microservice, we will create a class that derives the IConsumer interface; this interface defines a class that is a consumer of a message. Then we implement the “Consume” function, this function will be called every time someone publishes a message of type OrderCompleted. This function will take the buyerId value from the message and it will clear the corresponding basket.

c. Catalog microservice

This microservice’s implementation is very close to that of the ordering microservice. They both use the same MS SQL server but different databases and are developed in the same way

d. Dockerization of microservices

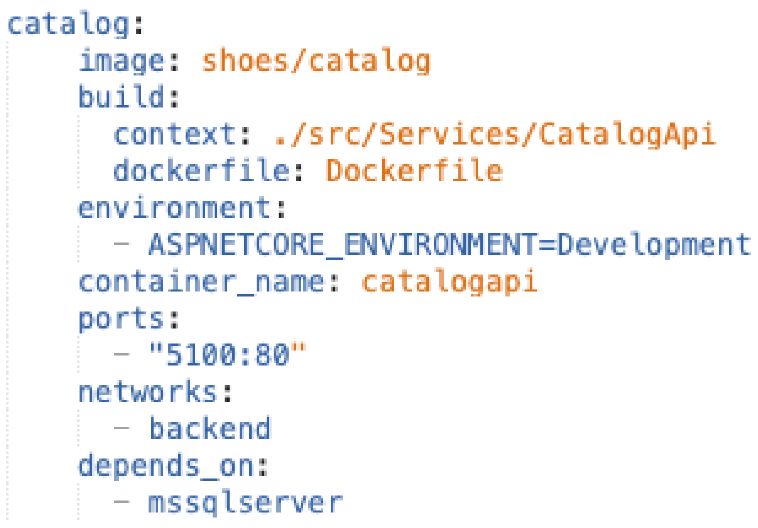
Each microservice in ShoesOnContainers is dockerized. These microservices need to communicate with each other so they need to be in the same Docker network.

The Compose file is a YAML file defining services, networks and volumes.
A service definition contains configuration that is applied to each container started for that service.

In ShoesOnContainers, we have 7 services that need to be inside a Docker image, these services are the following:

- Catalog
- Basket
- Mssqlserver
- Redis
- Ordering
- Gateway
- Rabbitmq

Here is an example for the catalog service on how to define it inside Docker compose file. The same goes for all the other services.



```
catalog:
    image: shoes/catalog
    build:
      context: ./src/Services/CatalogApi
      dockerfile: Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
    container_name: catalogapi
    ports:
      - "5100:80"
    networks:
      - backend
    depends_on:
      - mssqlserver
```

Figure 14: Screenshot of docker-compose file

<u>Dockerization steps:</u>

a. Specify the image name (image)
b. Define the configuration options that are applied at build time. (build)
c. Define the environment variables
d. Specify the container name
e. Specify the external and internal port of the Docker container
f. Specify in which network this Docker container will be
g. Specify which container should be available before running this service.

5- API Gateway

a. Why?

We don't need an API gateway, but using a Gateway pattern sure simplifies things.

In a Microservices architecture, there's a couple of different strategies we can follow:
-We can either have separate APIs that individual consumers need to be aware of,
-Or we can present a unified API via an API Gateway.

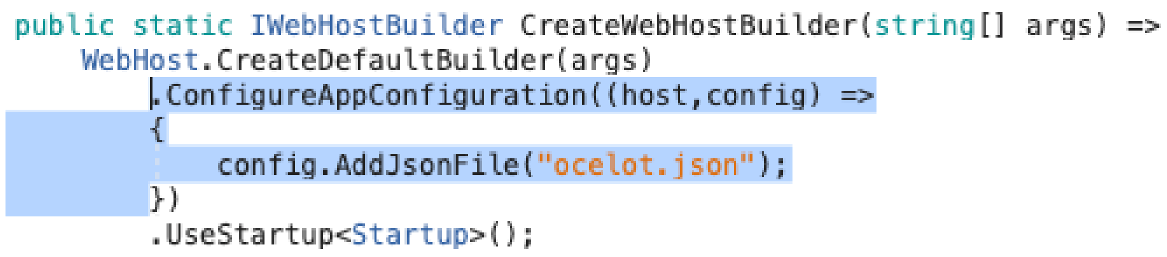
If we're presenting your API via a unified gateway, we will need an API Gateway platform.We developed this API Gateway using Ocelot, an open-source .NET Core API Gateway.

b. How?

First, we create a microservice. At this point, we need to do 2 things:

1.Add ocelot.json to give Ocelot directions for microservice redirects
2.Add Ocelot plumbing in the ApiGateway ASP.NET Core Web API Project

To add the configuration, we simply add an 'ocelot.json' file. Then, we need to make sure our ApiGateway project knows about it. This can be done using the following lines of code inside Program class:



```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((host,config) =>
        {
            config.AddJsonFile("ocelot.json");
        })
        .UseStartup<Startup>();
```

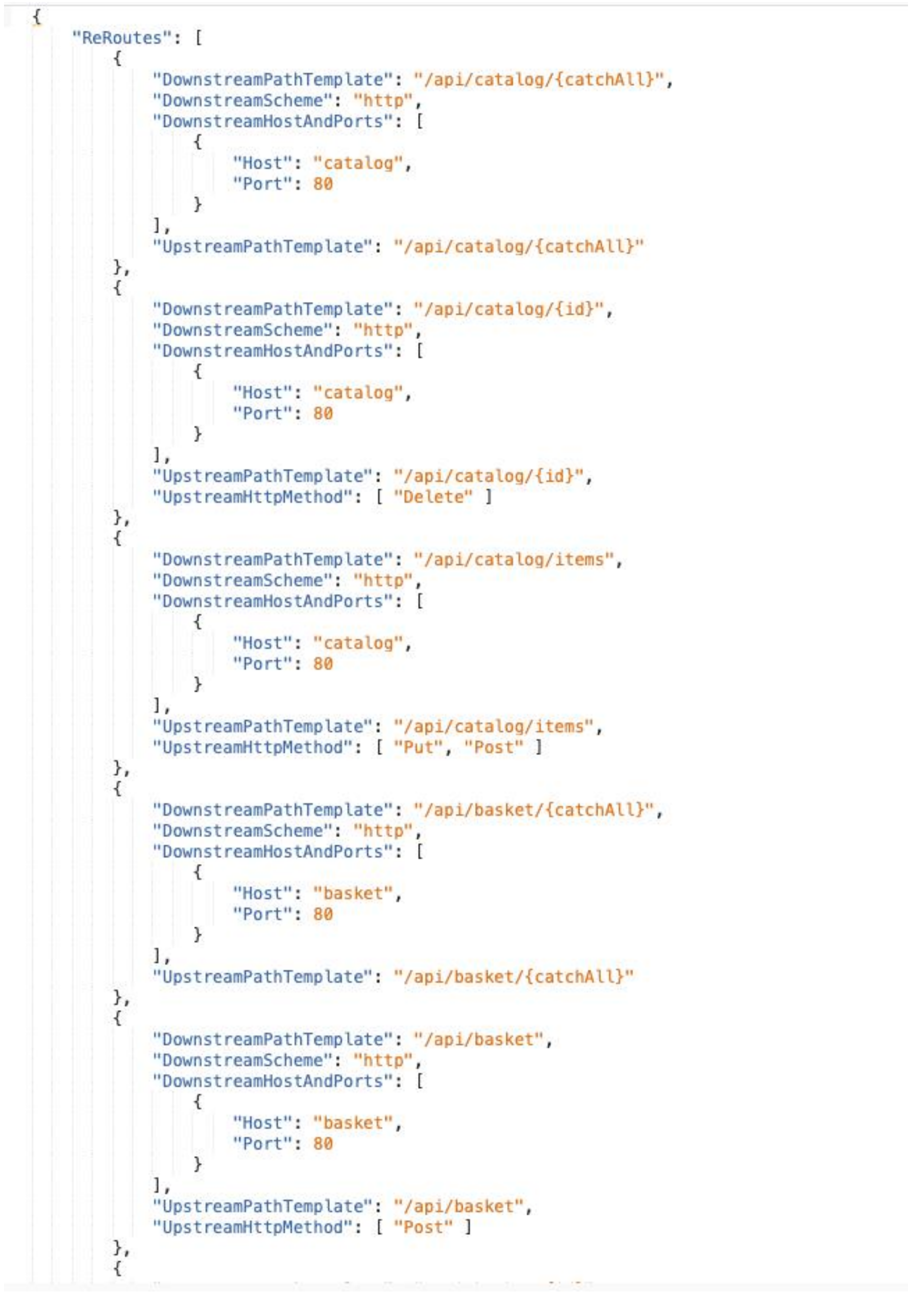
Figure 15: Screenshot to add configuration for API Gateway

Once we do that, we configured the ocelot.json like. The following are important things to note:

1. The GlobalConfiguration allows us to define the BaseUrl that Ocelot is going to listen to. In our case, we configured it to use localhost:5000

2. The ReRoutes array allows us to define one or more redirects. The downstream paths represent the paths to the microservices. The upstream paths represent how the users will access these microservices via the API Gateway.

3. We can do a wildcard redirect for each api by providing a variable name as shown. In our case, we've used a variable name {catchAll} that will catch all incoming traffic on /api/catalog/ and redirect it to api/catalog/ on 80 on host "catalog" because the catalog microservice will be running inside a Docker container.



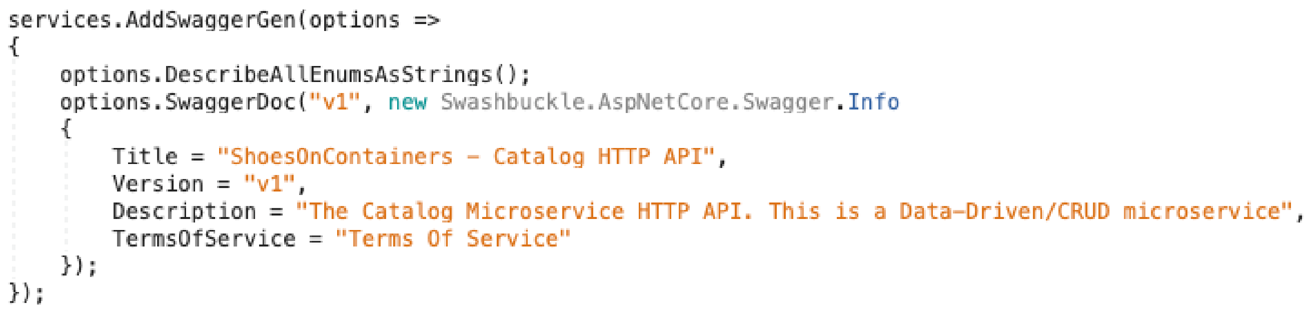
```
{
    "ReRoutes": [
        {
            "DownstreamPathTemplate": "/api/catalog/{catchAll}",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "catalog",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/catalog/{catchAll}"
        },
        {
            "DownstreamPathTemplate": "/api/catalog/{id}",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "catalog",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/catalog/{id}",
            "UpstreamHttpMethod": [ "Delete" ]
        },
        {
            "DownstreamPathTemplate": "/api/catalog/items",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "catalog",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/catalog/items",
            "UpstreamHttpMethod": [ "Put", "Post" ]
        },
        {
            "DownstreamPathTemplate": "/api/basket/{catchAll}",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "basket",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/basket/{catchAll}"
        },
        {
            "DownstreamPathTemplate": "/api/basket",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "basket",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/basket",
            "UpstreamHttpMethod": [ "Post" ]
        },
        {
```

Figure 16: Screenshot showing ReRoutes

6- Swagger

First, we need to add “Swashbuckle.AspNetCore” package to our microservice.

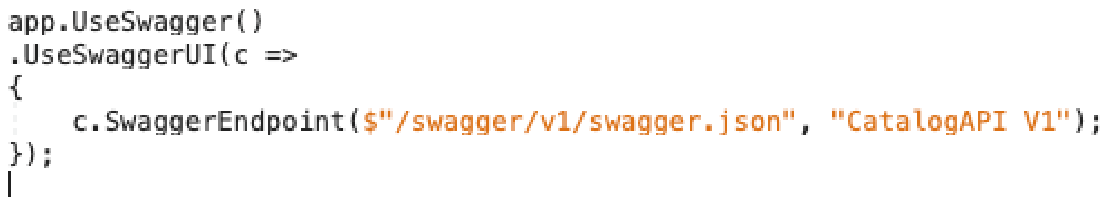
After the package has been installed we have to configure it in we Startup.cs file. Inside ConfigureServices we need to add a swagger method called AddSwaggerGen and inside it, we will insert our documentation.



```
services.AddSwaggerGen(options =>
{
    options.DescribeAllEnumsAsStrings();
    options.SwaggerDoc("v1", new Swashbuckle.AspNetCore.Swagger.Info
    {
        Title = "ShoesOnContainers - Catalog HTTP API",
        Version = "v1",
        Description = "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice",
        TermsOfService = "Terms Of Service"
    });
});
```

Figure 17: Screenshot of adding AddSwaggerGen

The next step is to add the SwaggerUI, we will configure and add it in the Configure method in our Startup.cs file. First we add the method UseSwagger, and after that, we add the method UseSwaggerUI and inside it, we add the url to access swagger.



```
app.UseSwagger()
.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint($"/swagger/v1/swagger.json", "CatalogAPI V1");
});
```

Figure 18: Screenshot of Swagger configuration for Catalog microservice

Now we can access swagger using the following url: http://localhost:5100/swagger/

IX. Steps to run the project

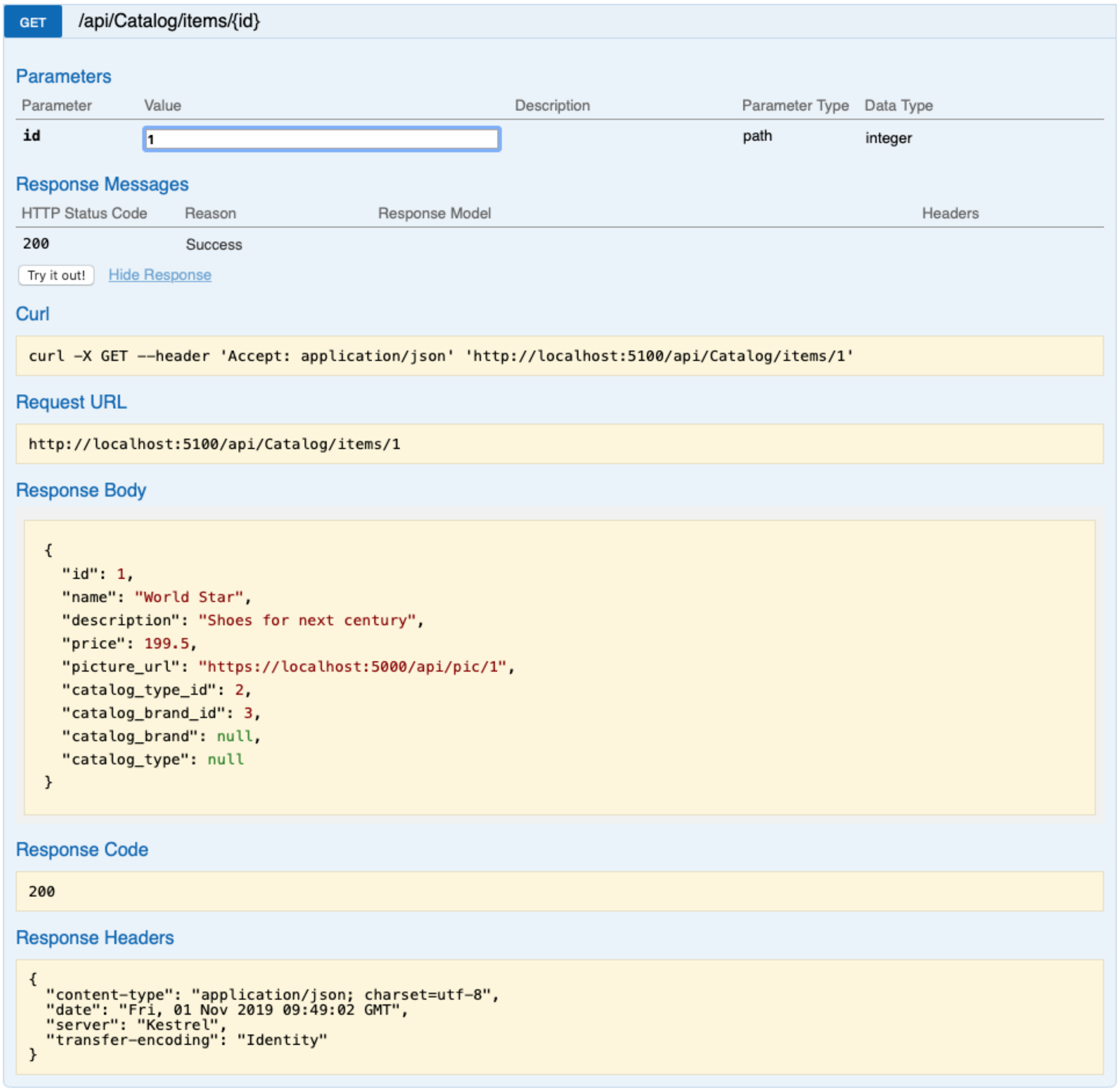
To see how you can run the application, please refer to the Readme on this Github repo.

X. Results & Conclusion

In conclusion, we have managed to build an online shop for shoes using 3 different and completed separated microservices developed in .NET.

The user can first look through the products to choose one, and then he can add it to their basket.

For example, we will show you how you can get an item or add an item to the catalog



Parameter	Value	Description	Parameter Type	Data Type
id	1		path	integer

HTTP Status Code	Reason	Response Model	Headers
200	Success		

```
curl -X GET --header 'Accept: application/json' 'http://localhost:5100/api/Catalog/items/1'
```

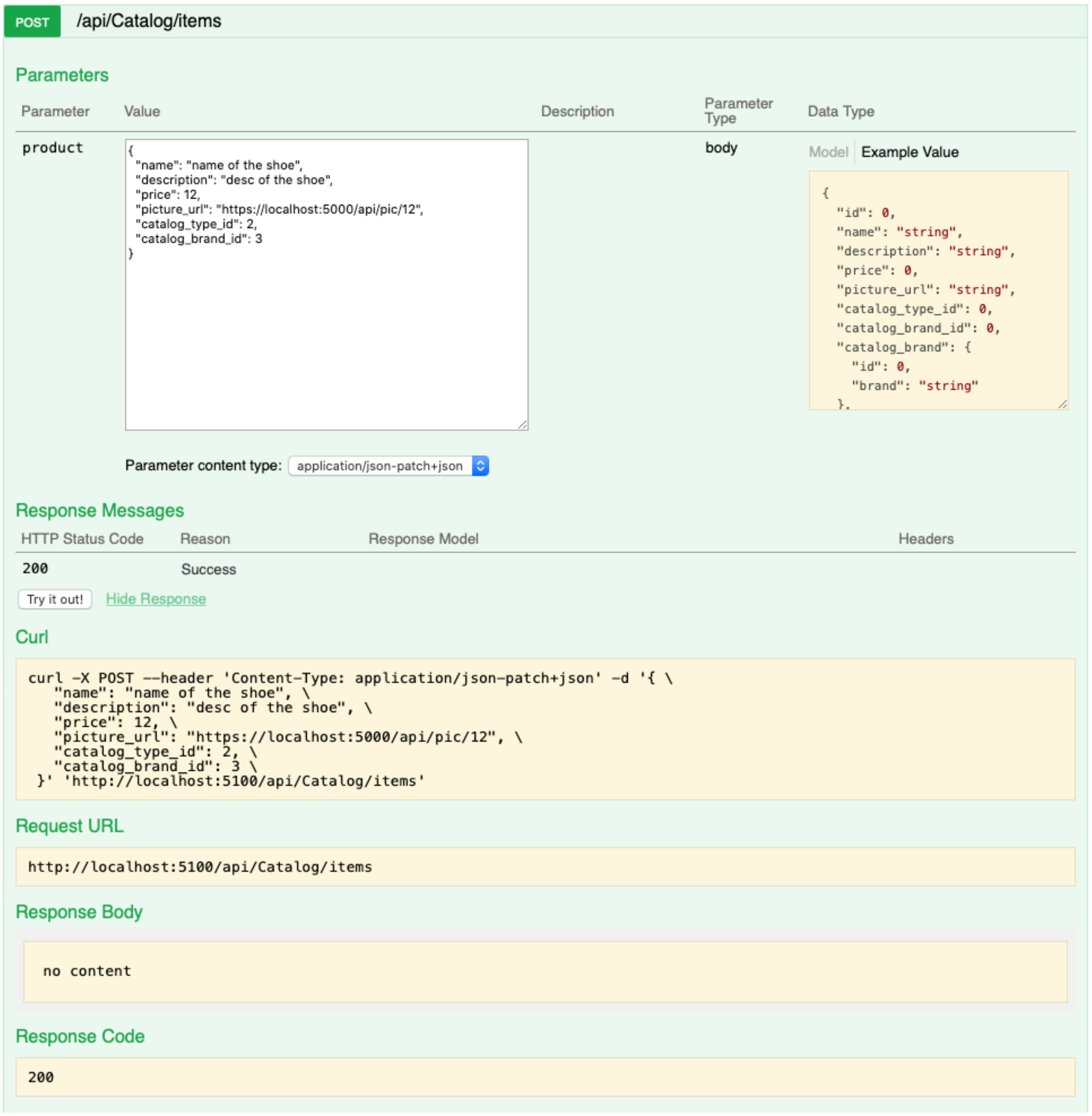
```
http://localhost:5100/api/Catalog/items/1
```

```
{
  "id": 1,
  "name": "World Star",
  "description": "Shoes for next century",
  "price": 199.5,
  "picture_url": "https://localhost:5000/api/pic/1",
  "catalog_type_id": 2,
  "catalog_brand_id": 3,
  "catalog_brand": null,
  "catalog_type": null
}
```

```
200
```

```
{
  "content-type": "application/json; charset=utf-8",
  "date": "Fri, 01 Nov 2019 09:49:02 GMT",
  "server": "Kestrel",
  "transfer-encoding": "Identity"
}
```

Figure 19: Screenshot from Swagger of GET catalog item



POST /api/Catalog/items

Parameters

Parameter	Value	Description	Parameter Type	Data Type
product	{ "name": "name of the shoe", "description": "desc of the shoe", "price": 12, "picture_url": "https://localhost:5000/api/pic/12", "catalog_type_id": 2, "catalog_brand_id": 3 }		body	Model Example Value

```
{
"name": "name of the shoe",
"description": "desc of the shoe",
"price": 12,
"picture_url": "https://localhost:5000/api/pic/12",
"catalog_type_id": 2,
"catalog_brand_id": 3
}
```

Example Value:

```
{
  "id": 0,
  "name": "string",
  "description": "string",
  "price": 0,
  "picture_url": "string",
  "catalog_type_id": 0,
  "catalog_brand_id": 0,
  "catalog_brand": {
    "id": 0,
    "brand": "string"
  },
```

Parameter content type: application/json-patch+json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Curl

```
curl -X POST --header 'Content-Type: application/json-patch+json' -d '{ \
   "name": "name of the shoe", \
   "description": "desc of the shoe", \
   "price": 12, \
   "picture_url": "https://localhost:5000/api/pic/12", \
   "catalog_type_id": 2, \
   "catalog_brand_id": 3 \
 }' 'http://localhost:5100/api/Catalog/items'
```

Request URL

```
http://localhost:5100/api/Catalog/items
```

Response Body

```
no content
```

Response Code

```
200
```

Figure 20: Screenshot from Swagger of POST catalog item

XI. References

1. “Why Use Microservices?”, DZone. [Online]. Available: dzone.com. Accessed: 23/9/2019
2. “What is Microservices?”, Smart bear. [Online]. Available: smartbear.com. Accessed: 15/7/19
3. “.NET Microservices: Architecture for Containerized .NET Applications”, Microsof. [Online]. Available: docs.microsoft.com. Accessed: 23/9/19
4. “What is Docker?”, Open Source. [Online]. Available: opensource.com. Accessed: 24/9/2019
5. “C#”, Tutorials Point. [Online]. Available: tutorialspoint.com. Accessed: 24/9/2019
6. “MS SQL Server – Overview”, Tutorials Point. [Online]. Available: tutorialspoint.com. Accessed: 24/9/2019
7. “Introduction to Redis”, Redis. [Online]. Available: redis.io. Accessed: 5/10/2019
8. “An introduction to Message Brokers”, Medium. [Online]. Available: medium.com. Accessed: 5/10/2019
9. “Introduction to RabbitMQ”, Tutlane. [Online]. Available: tutlane.com. Accessed: 5/10/2019
10. “Swagger UI”, Swagger. [Online]. Available: swagger.io. Accessed: 13/10/2019
11. “ASP.NET Core 2.0 E-Commerce website based on microservices”, Udemy. [Online]. Available: udemy.com. Accessed: 25/9/2019