Microservices Architecture & Step-by-Step Implementation on .NET

aspnetrun

run-aspnetcore-microservices

Beginner to Intermediate

Mehmet Ozkaya

EDITION v1.2

DOWNLOAD available at:

https://github.com/aspnetrun/run-aspnetcore-

microservices

PUBLISHED BY

AspnetRun Corporation

Bahcelievler, Istanbul 98052-6399 Copyright

Author:

Mehmet Ozkaya (https://github.com/mehmetozkaya)

Level:

Beginner to Intermediate

Contents

Table of Contents

SECTION 1

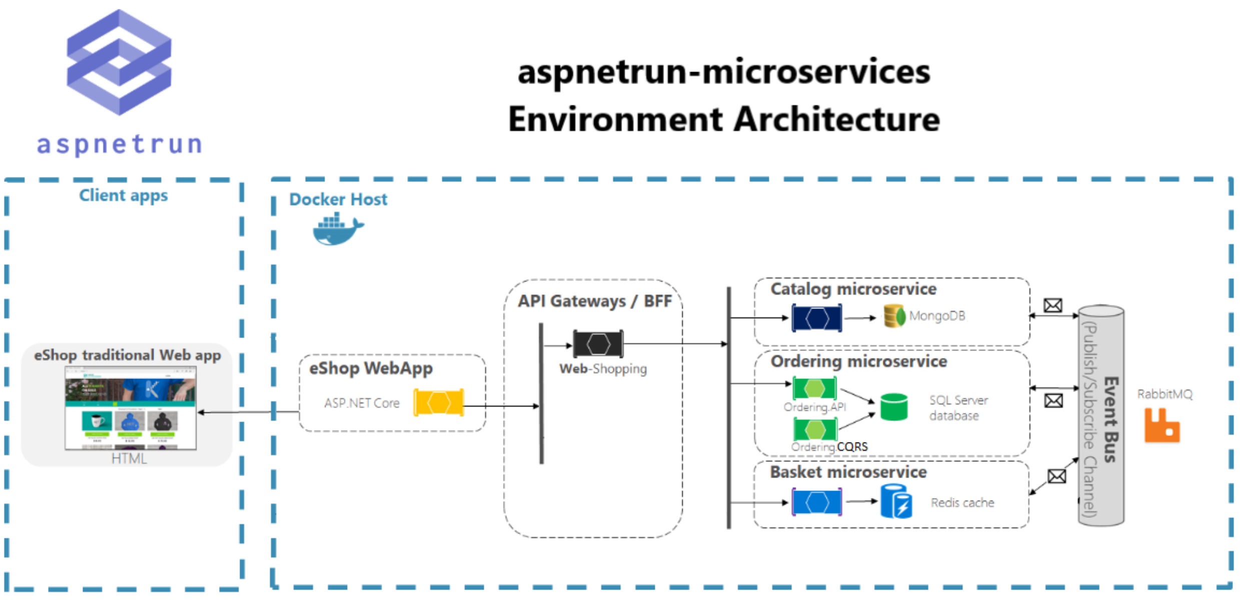
Introduction

*This book explains **aspnetrun-microservices repository** of GitHub. This book intended for the beginner to intermediate of asp.net core and microservices.*

In this book we will show **how to build microservices on .net environments** with using **ASP.NET Core Web API** applications, Entity Framework Core and different databases platforms **NoSQL** and **Relational databases**.

By the end of the book, you will have a couple of microservices which implemented e-commerce modules over **Catalog, Basket** and **Ordering microservices** with communicating over **RabbitMQ** Event Driven Communication.

Look at the big picture of final architecture of the system.



According to this design, we have a web application which basically implement e-commerce domain. This application has below functionalities;

- Retrieving Products and Categories and listing, filtering them
- Add products to Basket with applying quantity, color and calculating total basket price

- Check out Basket and create order with submitting the basket

Along with this you'll develop following microservices and items:

- **Catalog microservice** which includes;
    - ASP.NET Core Web API application
    - REST API principles, CRUD operations
    - Mongo DB NoSQL database connection on docker
    - N-Layer implementation with repository pattern
    - Swagger Open API implementation
    - Dockerfile implementation
- **Basket microservice** which includes;
    - ASP.NET Core Web API application
    - REST API principles, CRUD operations
    - Redis database connection on docker
    - Redis connection implementation
    - RabbitMQ trigger event queue when checkout cart
    - Swagger Open API implementation
    - Dockerfile implementation
- **RabbitMQ messaging library** which includes;
    - Base EventBus implementation and add references Microservices
- **Ordering microservice** which includes;
    - ASP.NET Core Web API application
    - Entity Framework Core on SQL Server docker
    - REST API principles, CRUD operations
    - Consuming RabbitMQ messages
    - Clean Architecture implementation with CQRS Pattern
    - Event Sourcing
    - Implementation of MediatR, Autofac, FluentValidator, AutoMapper
    - Swagger Open API implementation
    - Dockerfile implementation
- **API Gateway Ocelot microservice** which includes

- Routing to inside microservices
- Dockerization api-gateway

- **WebUI ShoppingApp microservice** which includes
  - Asp.net core Web Application with Razor template
  - Call Ocelot APIs with HttpClientFactory
  - Asp.net Core Razor Tools — View Components, partial Views, Tag Helpers, Model Bindings and Validations, Razor Sections etc.
- **Docker Compose establishment** with all microservices on docker;
  - Dockerization of microservices
  - Dockerization of database
  - Override Environment variables

Source Code

Get the Source Code from AspnetRunMicroservices Github Repository — 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.

Build Web Application with ASP.NET Core, Entity Framework Core
Github : https://github.com/aspnetrun/run-aspnetcore-microservices

Prerequisites

- Install the .NET Core 3.x or above SDK
- Install Visual Studio 2019 v16.x or above
- Install Docker Desktop

Run Application

Before running the application, start the **Docker Desktop.**

Follow these steps to get your development environment set up:

- Clone the repository
- And 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
```

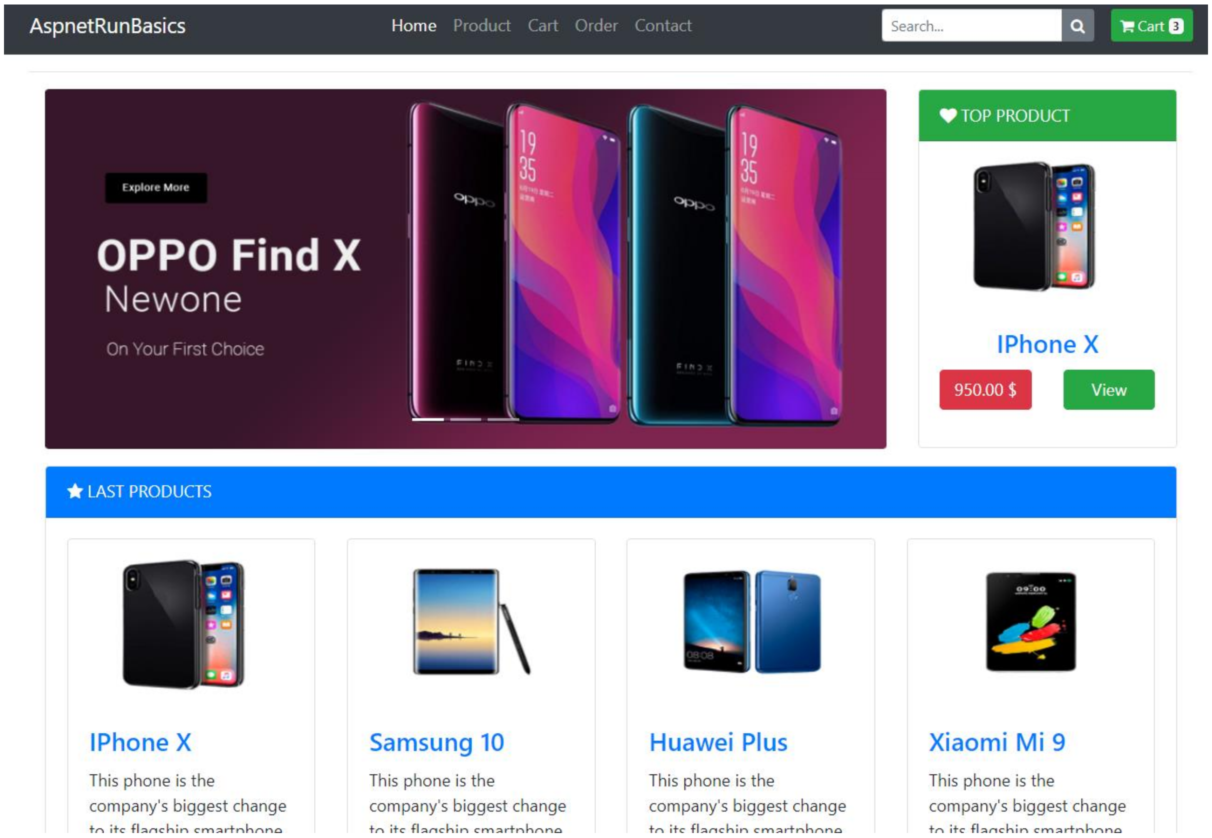
- Wait for docker finished to compose all microservices.

That's it!

You can **launch microservices** as below URLs:

RabbitMQ -> http://localhost:15672/
Catalog API -> http://localhost:8000/swagger/index.html
Basket API -> http://localhost:8001/swagger/index.html
Order API -> http://localhost:8002/swagger/index.html
API Gateway -> http://localhost:7000/Order?username=swn
Shopping Web UI -> http://localhost:8003/

You can use **Shopping Web UI** Project in order to call microservices over API Gateway. When you are **checkout the basket** you can follow queue record on **RabbitMQ dashboard**.



This phone is the company's biggest change to its flagship smartphone

This phone is the company's biggest change to its flagship smartphone

This phone is the company's biggest change to its flagship smartphone

This phone is the company's biggest change to its flagship smartphone

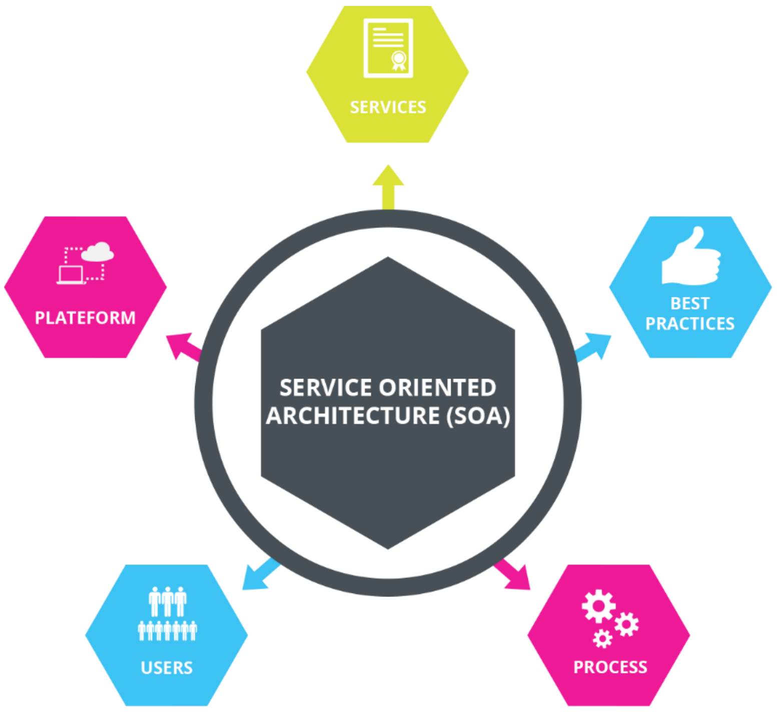
Background

Before start to microservices, you should now some concepts and use them in order to understand microservice development process.

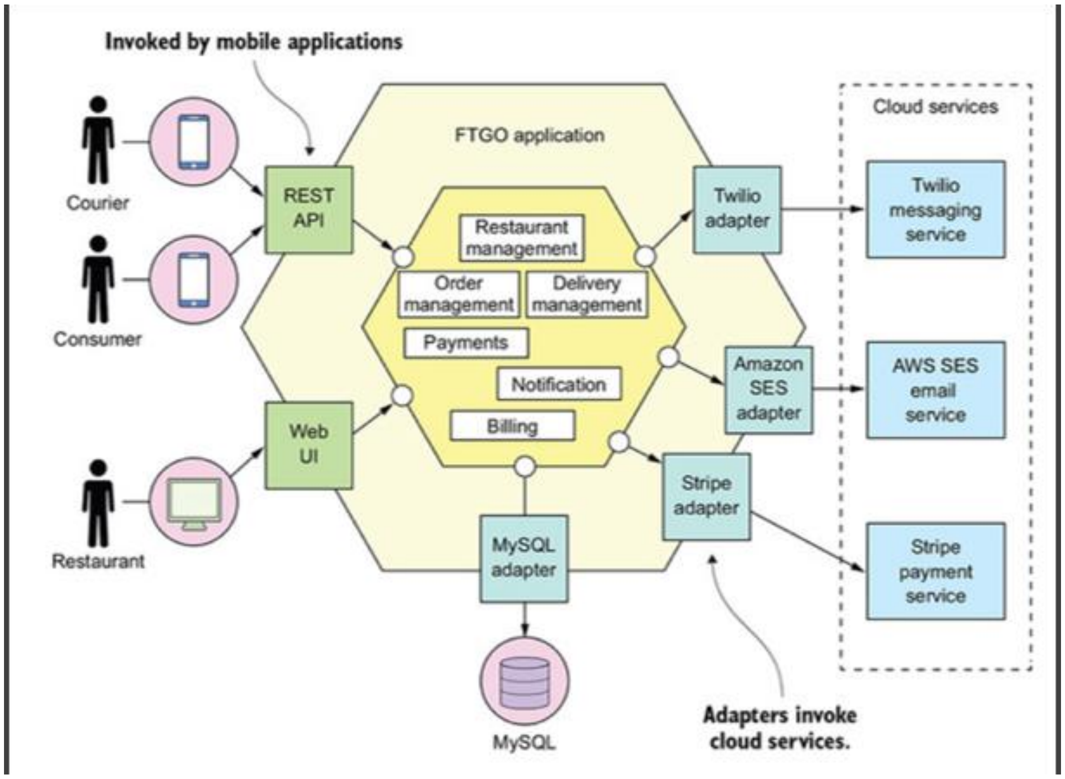
The starting point of the Microservice architecture is **SOA**. Microservice is an architecture based on SOA, or Services Oriented Architecture.

Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an architecture that can enable services to communicate in the distribution system when applications are distributed to each other over a single machine or network over multiple machines.

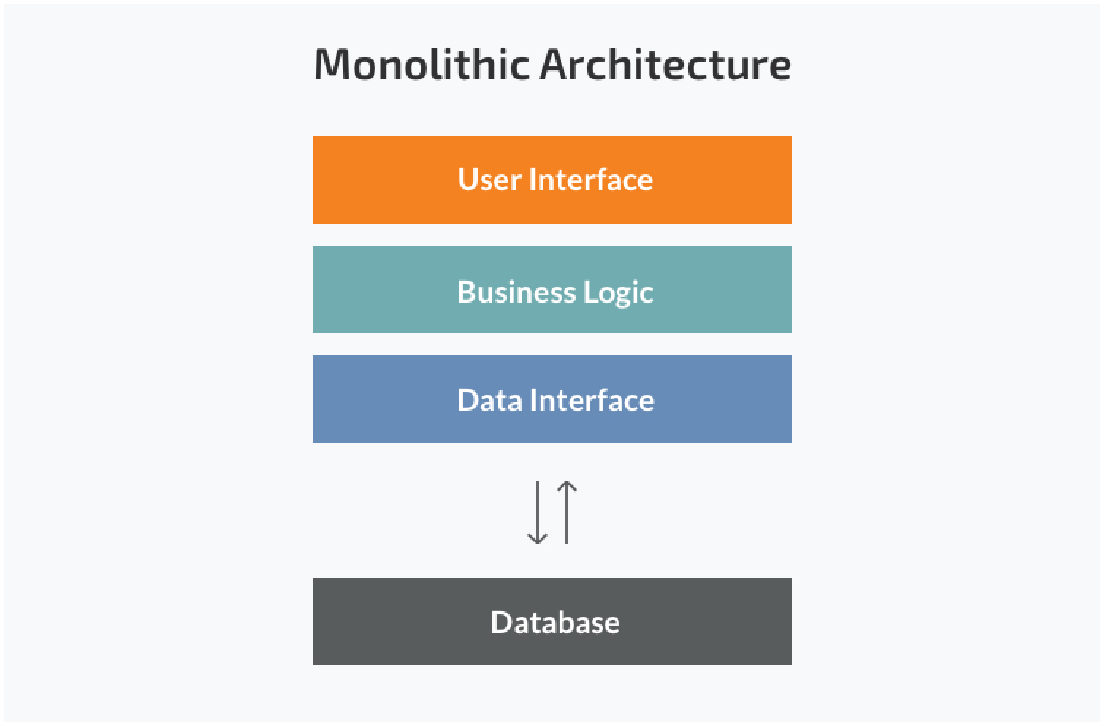


The name we give to the architecture we use for services that are used for multiple applications that want to use the same data layer and / or business together with the need for integration or Omni-Channel and applications that consume these services.

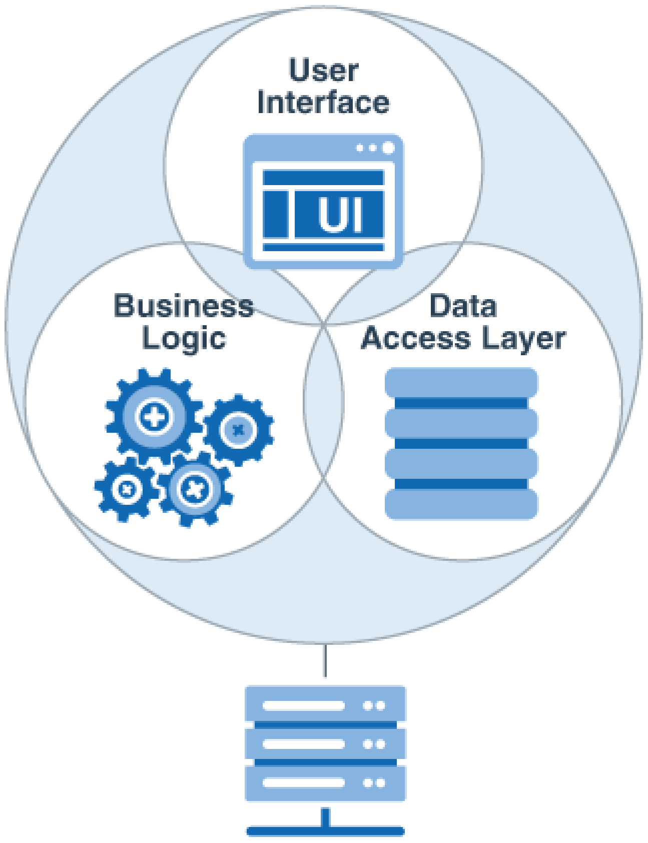


Monolithic Architecture

Monolithic is an architectural form that contains all layers and modules of the developed application in a single project. In Monolithic architecture, layers such as authorization, presentation, business logic, data can take place. Imagine you are developing an e-commerce application, at least you have a business and data layer where you keep user information, log in, list products and get paid. It is the architecture that offers all these modules in the same project without breaking them apart.



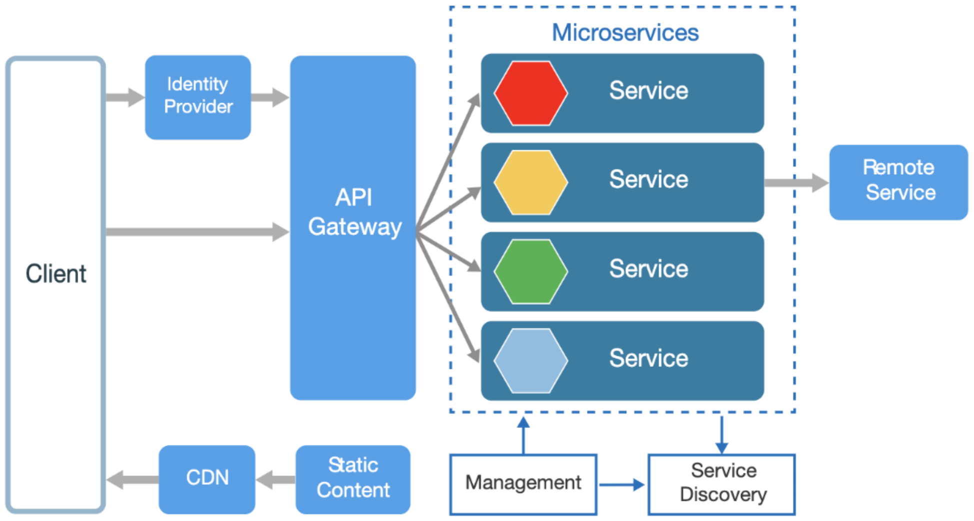
Monolithic architecture means that the software is designed as self-contained. We can also say that it is formed as a “single piece” in line with a standard. The components in this architecture are designed as interdependent rather than loosely coupled.



As time progresses, we begin to face various problems in SOAs designed with Monolithic Architecture. With the increasing requirements, integration needs or the need to implement different modules, our service application begins to grow. With this growth, adding new developers to the tight coupling service project is pushing the project towards chaos.

Microservices

Microservice architecture is an approach that has a mechanism that is not too complicated and has a low dependence on other services, each of which should be considered as a small service in a modular structure when developing a single application and each service can carry out its own business and communication.



These services are based on a single job that they are responsible for and can operate independently and have an automated deployment mechanism. It should be free from central management mechanisms. It can be developed in different programming languages and different database technologies can be used.

Each microservice works with other services (**loosely coupled**) with very little dependency. These services are self-contained and offer a single functionality (or a group of common functionalities).

On any service developed in accordance with the Microservice architecture, the developer can work independently, thereby facilitating the development phase.

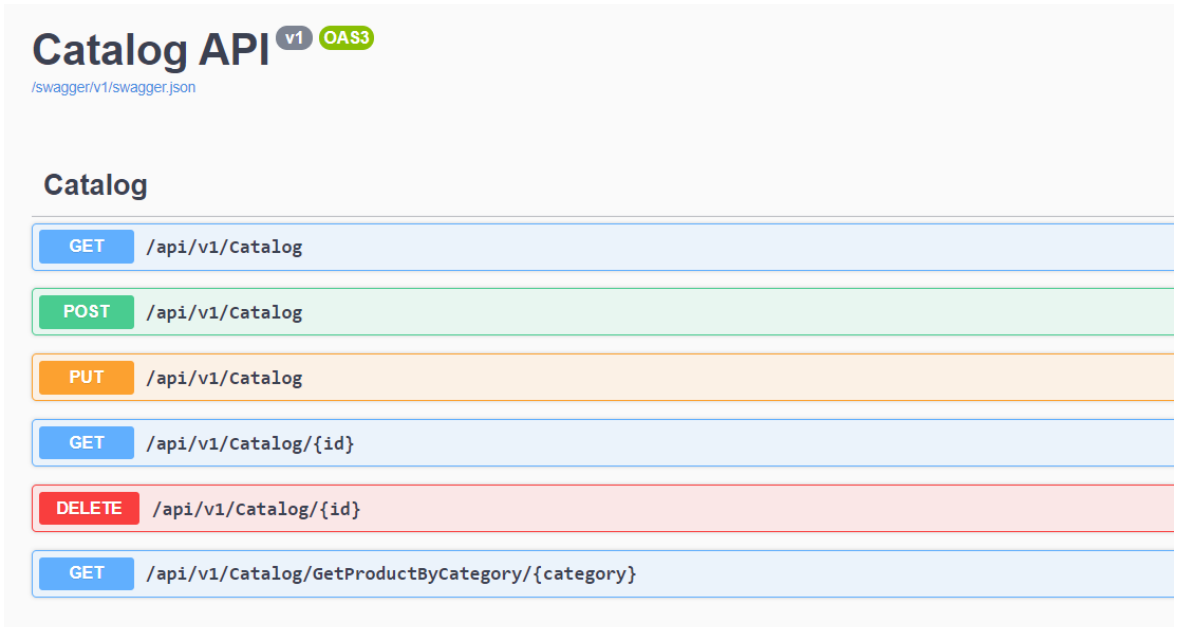
SECTION 2

Building Catalog Microservices

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

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

Look at the final swagger application.



```
/api/v1/Catalog
/api/v1/Catalog
/api/v1/Catalog
/api/v1/Catalog/{id}
/api/v1/Catalog/{id}
/api/v1/Catalog/GetProductByCategory/{category}
```

You'll learn how to Create **Catalog microservice** which includes;

- ASP.NET Core **Web API** application

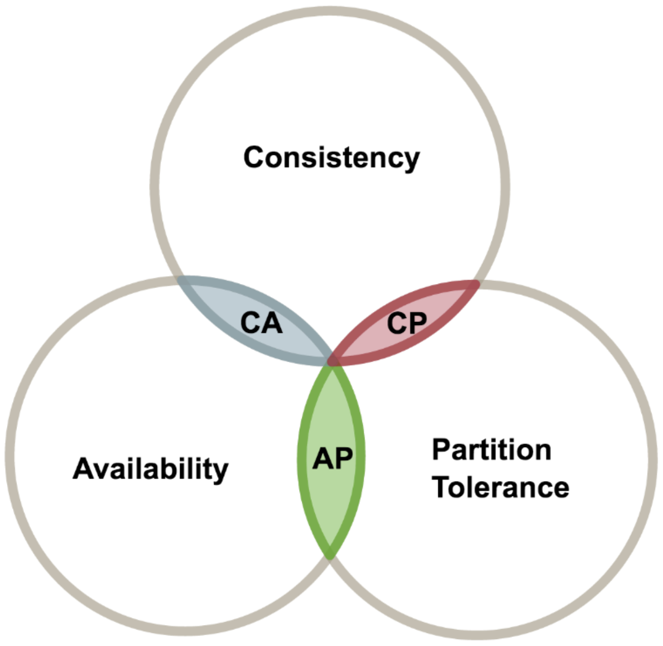
- **REST** API principles, CRUD operations
- **Mongo DB** NoSQL database connection on docker
- **N-Layer** implementation
- **Repository** Design Pattern
- **Swagger** Open API implementation
- **Dockerfile** implementation

Background

Before we start, we will start with definitions and explanations of catalog microservices on Asp.Net Core Web API.

CAP Theorem

We talked about **NoSQL's compromise on data consistency** for performance and accessibility.
Speaking without compromise, it is also necessary to touch on the **CAP theorem**.
We need to understand that all three features cannot be a NoSQL architecture either. Let's look at C, A, P of CAP with a sentence;

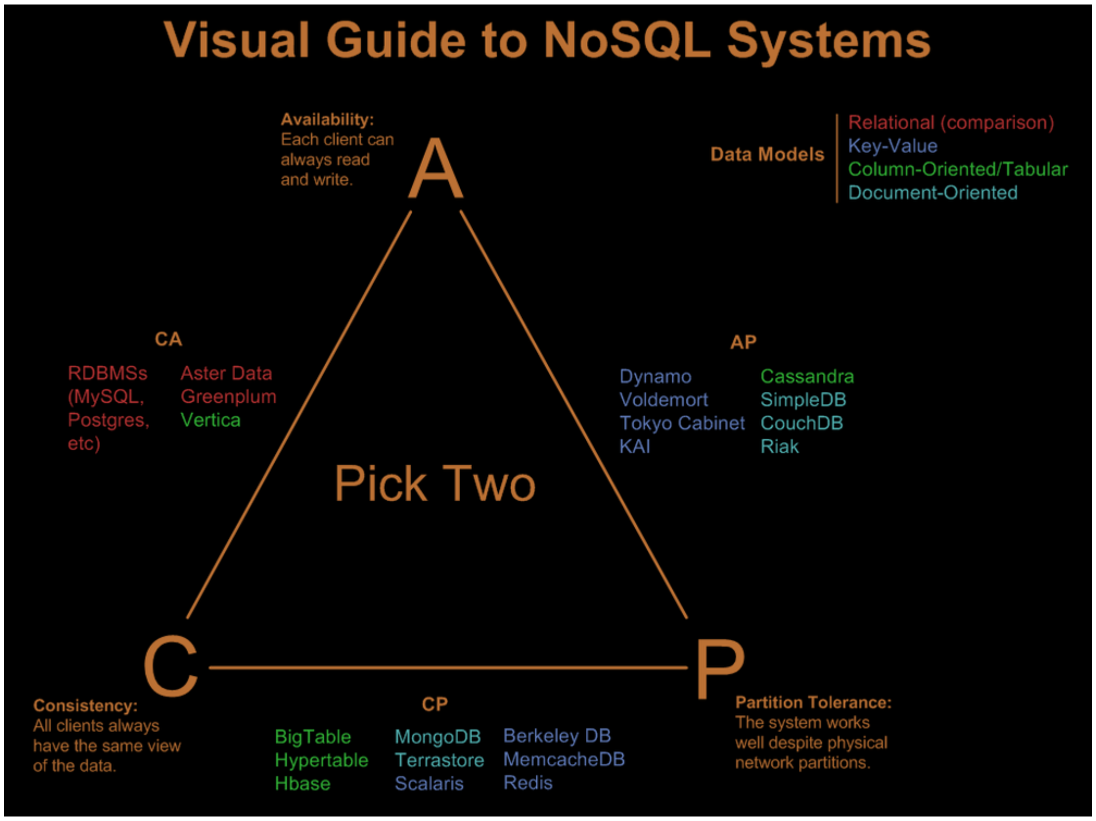


The theorem is that in a distributed system, 3 features cannot be provided simultaneously for the service provided over the data. As seen in the pictures above, the system you designed can be **CA, CP or AP. There can be no CAP**.

Consistency: When all nodes in the distributed system have the same data.

Availability: Status of every request made to the system, whether successful or unsuccessful. (Even if he doesn't have the most up-to-date data.)

Partition Tolerance: It is the ability of the system to continue operating even if a problem occurs in some of the existing nodes for network or other reasons and becomes inaccessible.



Finally, NoSQL database types and their preferred application types are as follows;

Document Based (MongoDB, CouchDB, etc.) E-commerce sites, Content management systems etc.

Key / Value (Redis etc.) User session information storage, Shopping cart data storage etc.

Graph Based (Neo4J etc.) Social media applications, Graph based search applications etc.

Column Based (Cassandra, HBase etc.) Transaction logging, IoT applications etc.

MongoDB

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

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

To download MongoDB, you need to go to the address in the docker hub.
After that you can use below commands;

mongo --version
If you run the command, you will receive version information about MongoDB installed on your computer.

mongo
command will activate the MongoDB Command Line Interface. All commands written after this command are interpreted in MongoDB and output is produced.

show dbs
command will bring all the databases on the server.

use [database name]
command will select the named database as available.

show collecitons
The command will bring the collections in the database being worked on.

Analysis & Design

This project will be the REST APIs which basically perform CRUD operations on Catalog databases.
We should define our Catalog use case analysis. In this part we will create Product— Category entities. Our main use case are Listing Products and Categories, able to search products. Also performed CRUD operations on Product entity.

Our main use cases;

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

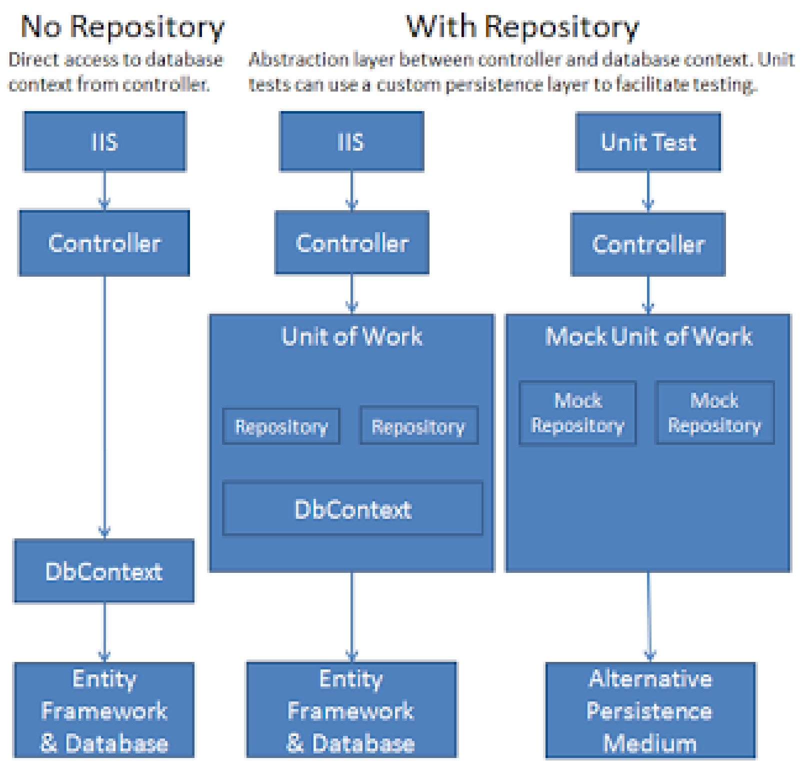
Along with this we should design our APIs according to REST perspective.

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

Repository Pattern

Repository is a concept that is used in order to write the information of all entity and value objects in an aggregate to the database. For each aggregate itself, we will perform the DB operations over the Repository in the transactional structure. Repository basically **prevents database work** from being moved from the workstation to a database, thus preventing query and code repetition. In other words, the main purpose is to process data and interrogations into a central structure avoiding repetitions.

In this way, we stay away from writing our database operations again and again in the business layer. **The Repository Design Pattern** has brought the logic of the sections that make the actual work in your program and the sections that access the data from each other. That is, it **acts as an interface** between the **data layer and the business layer** that uses this layer, and it also acts as an abstraction between these two layers.


No Repository
Direct access to database context from controller.
With Repository
Abstraction layer between controller and database context. Unit tests can use a custom persistence layer to facilitate testing.
IIS
Controller
DbContext
Entity Framework & Database
IIS
Controller
Unit of Work
Repository
Repository
DbContext
Entity Framework & Database
Unit Test
Controller
Mock Unit of Work
Mock Repository
Mock Repository
Alternative Persistence Medium

Database Setup with Docker

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

Setup Mongo Database

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

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

```
docker pull mongo
```

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

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

See the logs;

```
docker logs -f aspnetrun-mongo
```

Now we had a Mongo docker image. So now we can go into mongo image and run mongo commands in order to create database and collections;

```
docker exec -it aspnetrun-mongo /bin/bash
```

Now we are in mongo db. Let’s check folders;

```
ls
```

In order to use mongo commands, we should start with mongo command;

```
mongo
```

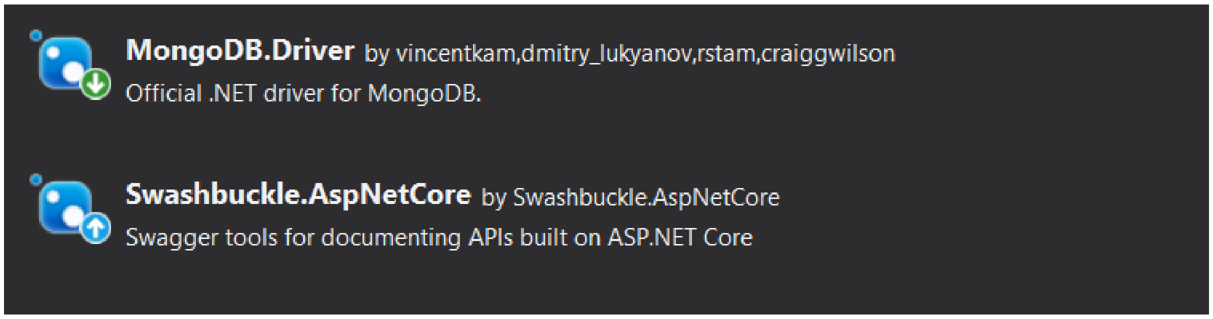
Below one are mongo commands which orderly show database – create CatalogDb database – cereate Products collection – add multiple row to collection.

```
show dbs
use CatalogDb  --> for create db on mongo
db.createCollection('Products')  --> for create people collection

db.Products.insertMany(
                [
                    {
                        "Name": "Asus Laptop",
                        "Category": "Computers",
                        "Summary": "Summary",
                        "Description": "Description",
                        "ImageFile": "ImageFile",
                        "Price": 54.93
                    },
                    {
                        "Name": "HP Laptop",
                        "Category": "Computers",
                        "Summary": "Summary",
                        "Description": "Description",
                        "ImageFile": "ImageFile",
                        "Price": 88.93
                    }
                ])

db.Products.find({}).pretty()
db.Products.remove({})

show databases
show collections
db.Products.find({}).pretty()
```

Library & Frameworks

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

1- Mongo.DB.Driver – To connect mongo database
2- Swashbuckle.AspNetCore – To generate swagger index page

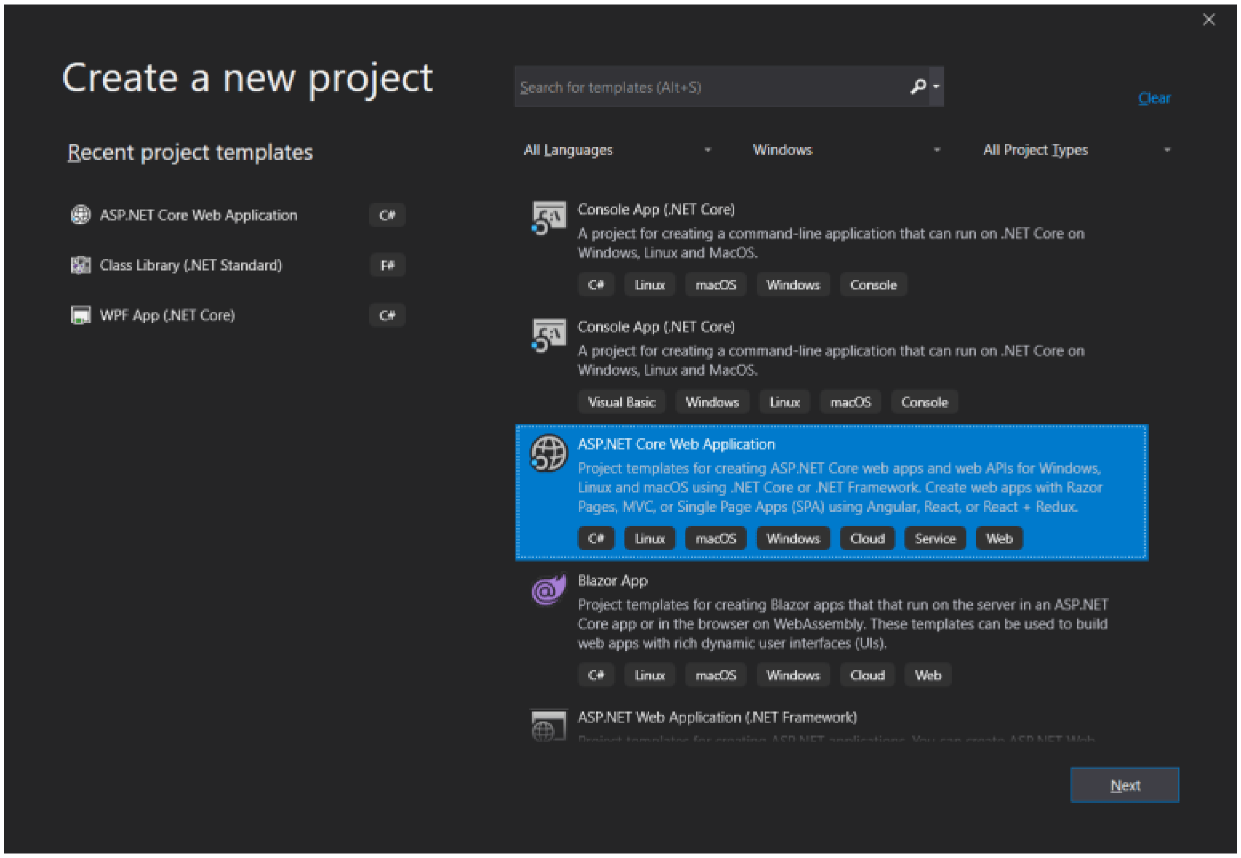

MongoDB.Driver by vincentkam,dmitry_lukyanov,rstam,craiggwilson
Official .NET driver for MongoDB.

Swashbuckle.AspNetCore by Swashbuckle.AspNetCore
Swagger tools for documenting APIs built on ASP.NET Core

Developing Application

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

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic Windows Linux macOS Console

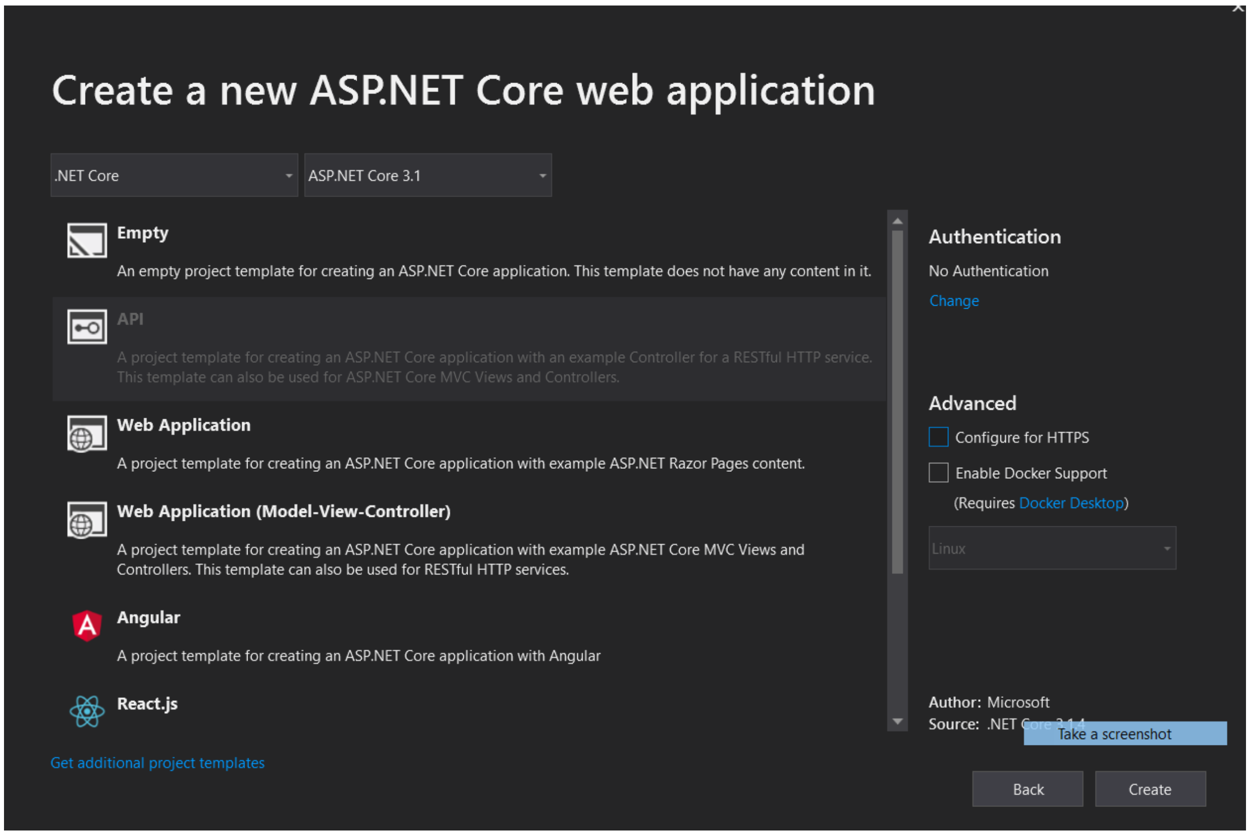
ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# Linux macOS Windows Cloud Service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

Blazor App
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).
C# Linux macOS Windows Cloud Web

ASP.NET Web Application (.NET Framework)

Next

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 Core 3.1.4

Add New Web API project under below location and name;
src/catalog/Catalog.API

Create Entities

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

Add New Class -> Product

Entities/Product.cs

```
public class Product
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        [BsonElement("Name")]
        public string Name { get; set; }
        public string Category { get; set; }
        public string Summary { get; set; }
        public string Description { get; set; }
        public string ImageFile { get; set; }
        public decimal Price { get; set; }
    }
```

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

Create Data Layer

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

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

Create **ICatalogContext** class.

```
Data/Interfaces/ICatalogContext.cs
public interface ICatalogContext
    {
        IMongoCollection<Product> Products { get; }
    }
```

Basically, we expect from our db context object is **Products** collections.
Continue with implementation and create **CatalogContext** class.

```
Data/CatalogContext.cs
public class CatalogContext : ICatalogContext
    {
        public CatalogContext(ICatalogDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);

            Products = database.GetCollection<Product>(settings.CollectionName);
            CatalogContextSeed.SeedData(Products);
        }

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

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

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

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

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

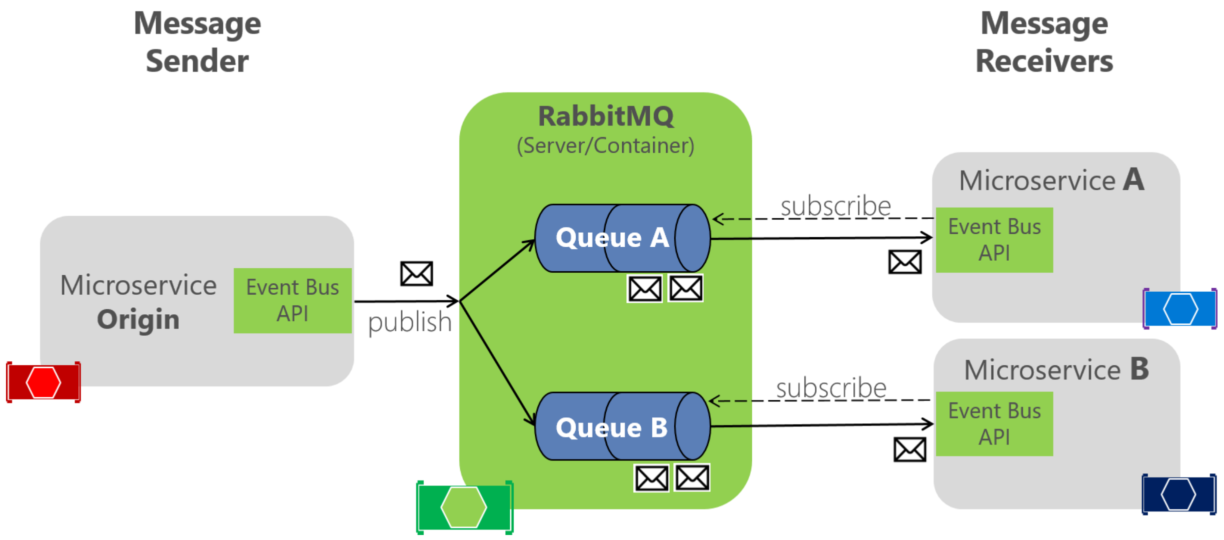
SECTION 3

Microservices Communication with Building RabbitMQ Library

In this section we will show how to perform **RabbitMQ connection** on **Basket** and **Ordering** microservices when producing and consuming events.

By the end of the section, we will have a library which provide to **communication along Basket and Ordering microservices** with creating a class library project.

The event bus implementation with RabbitMQ that microservices publish events, and receive events, as shown in below figure.



You'll learn how to Create basic **Event Bus with RabbitMQ** which includes;

- **Microservice communication** with **RabbitMQ** implementing **SAGA pattern**
- **Class Library** development
- **RabbitMQ Producer** on **Basket** Microservice Web API
- **RabbitMQ Consumer** on **Ordering** Microservice Web API
- **AutoMapper** implementation when mapping Event to Microservices entity
- **RabbitMQ Docker** Implementation

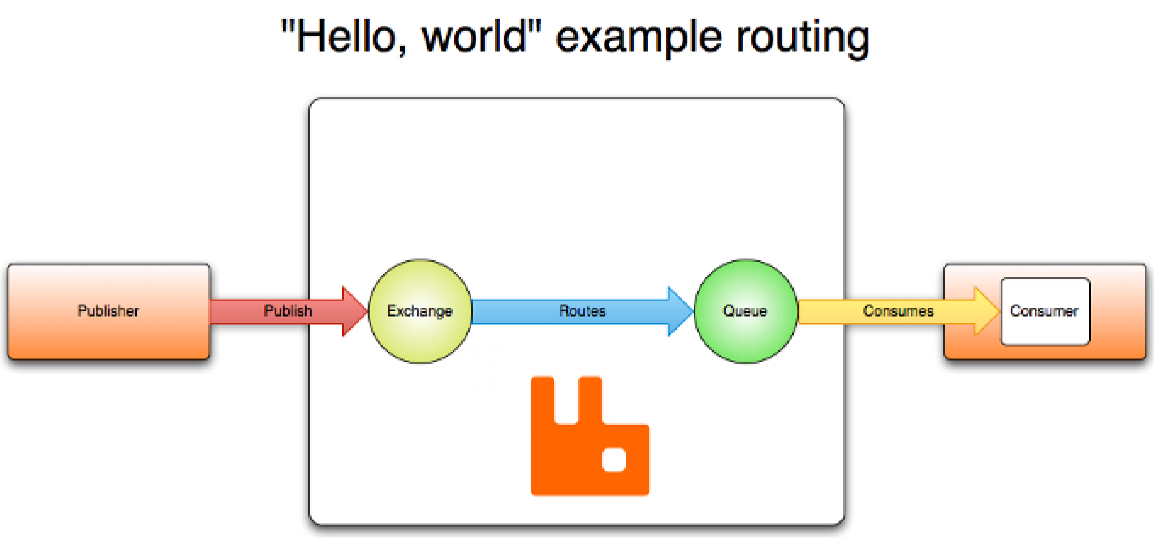
Background

Before we start, we will start with definitions and explanations of **RabbitMQ Library** and prerequests.

RabbitMQ

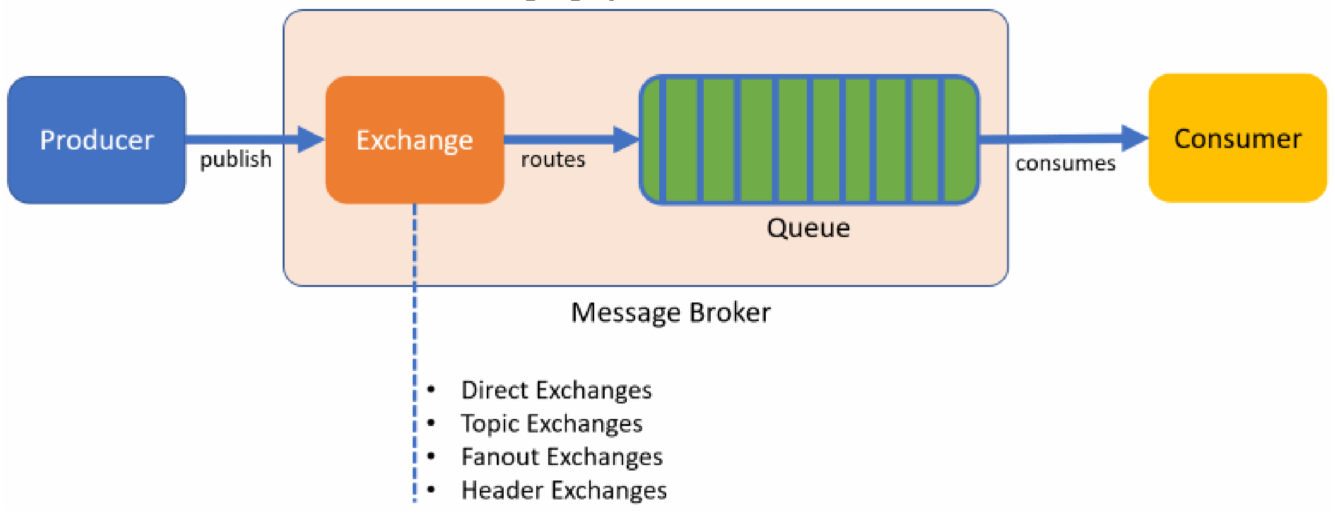
RabbitMQ is a message queuing system. Similar ones can be listed as Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ. Its purpose is to transmit a message received from any source to another source as soon as it is their turn.

In other words, all transactions can be listed **in a queue** until the source to be transmitted gets up. RabbitMQ's support for multiple operating systems and open source code is one of the most preferred reasons.



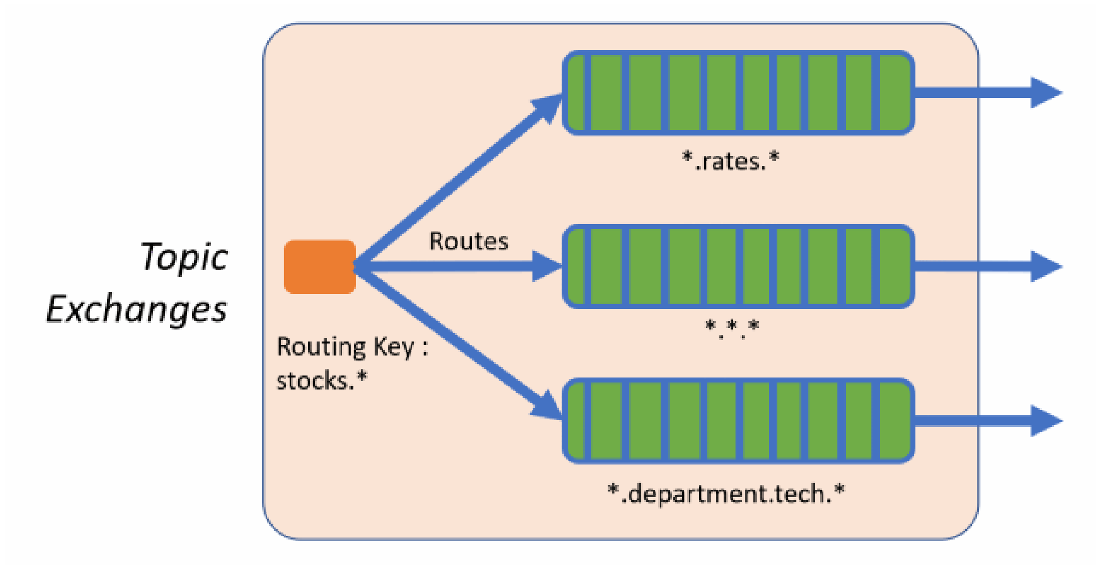
RabbitMQ Exchange Types

RabbitMQ is based on a messaging system like below.

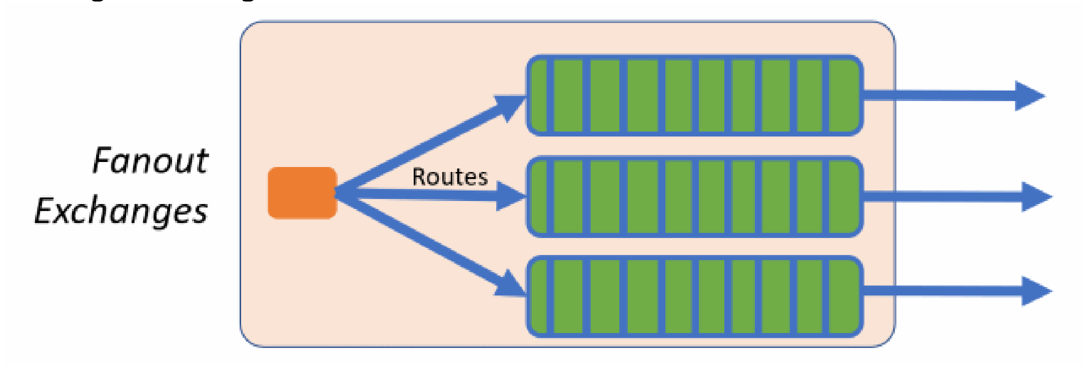


Direct Exchange: The use of a single queue is being addressed. A routing key is determined according to the things to be done and accordingly, the most appropriate queue is reached with the relevant direct exchange.

Topic Exchange: In Topic Exchanges, messages are sent to different queues according to their subject. The incoming message is classified and sent to the related queue. A route is used to send the message to one or more queues. It is a variation of the **Publish / Subscribe pattern**. If the problem concerns several consumers, Topic Exchange should be used to determine what kind of message they want to receive.



Fanout Exchange: It is used in situations where the message should be sent to more than one queue. It is especially applied in **Broadcasting systems**. It is mainly used for games for global announcements.



Headers Exchange: Here you are guided by the features added to the header of the message. **Routing-Key** used in other models is not used. Transmits to the correct queue with a few features and descriptions in message headers. The attributes on the header and the attributes on the queue must match each other's values.

Analysis & Design

This project will be the **Class Library** which basically perform **event bus operations** with **RabbitMQ** and use this library when **communicating Basket and Ordering** microservices.
We should define our Event Bus use case analysis.

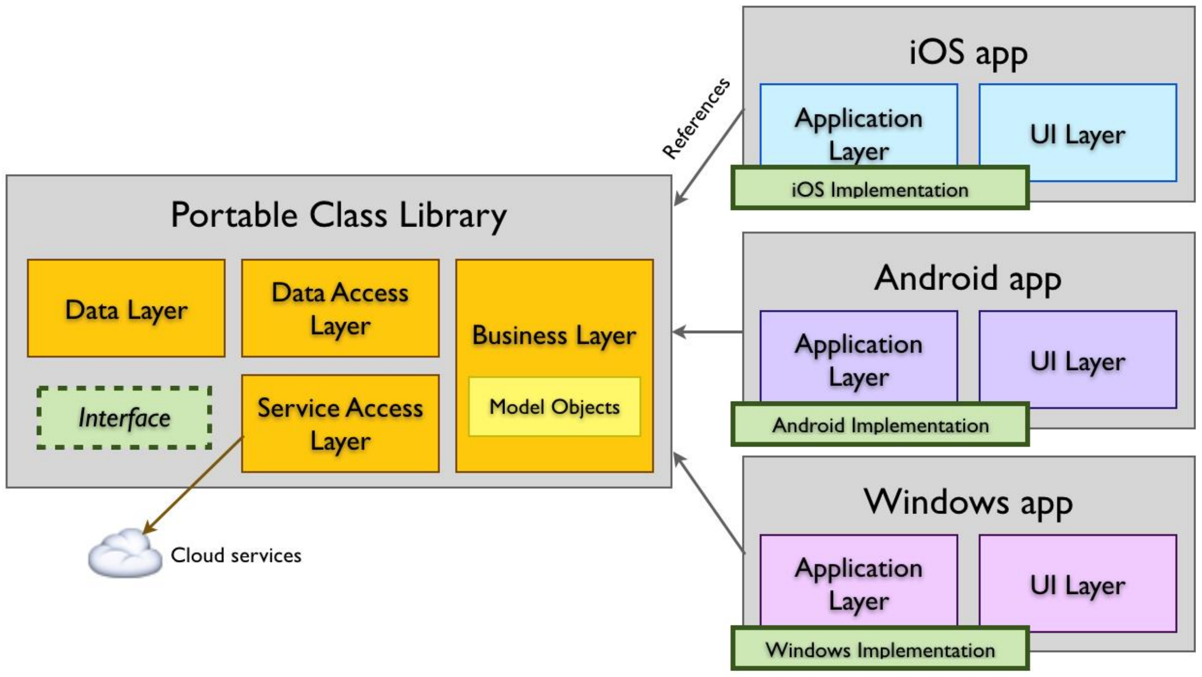
Our main use cases;

- Create RabbitMQ Connection
- Create BasketCheckout Event
- Develop Basket Microservices as Producer of BasketCheckout Event
- Develop Ordering Microservices as Consumer of BasketCheckout Event
- Update Basket and Items (add – remove item on basket)

- Delete Basket
- Checkout Basket

Architecture

In order to use different microservices, we will create a common class library which provide to connection with RabbitMQ and perform event bus operations.



According to this architecture, we encapsulate RabbitMQ operations into **EventBusRabbitMQ Class Library** project under the Common folder.

Basket microservices will be depended on this class library and use it when producer of BasketCheckout event in Checkout method of Basket API.

Ordering microservices will be depended on this class library and use it when consumer of BasketCheckout event in Checkout method of Basket API. When get the event Ordering microservice create an order record in their relational Sql Server database.

RabbitMQ Setup with Docker

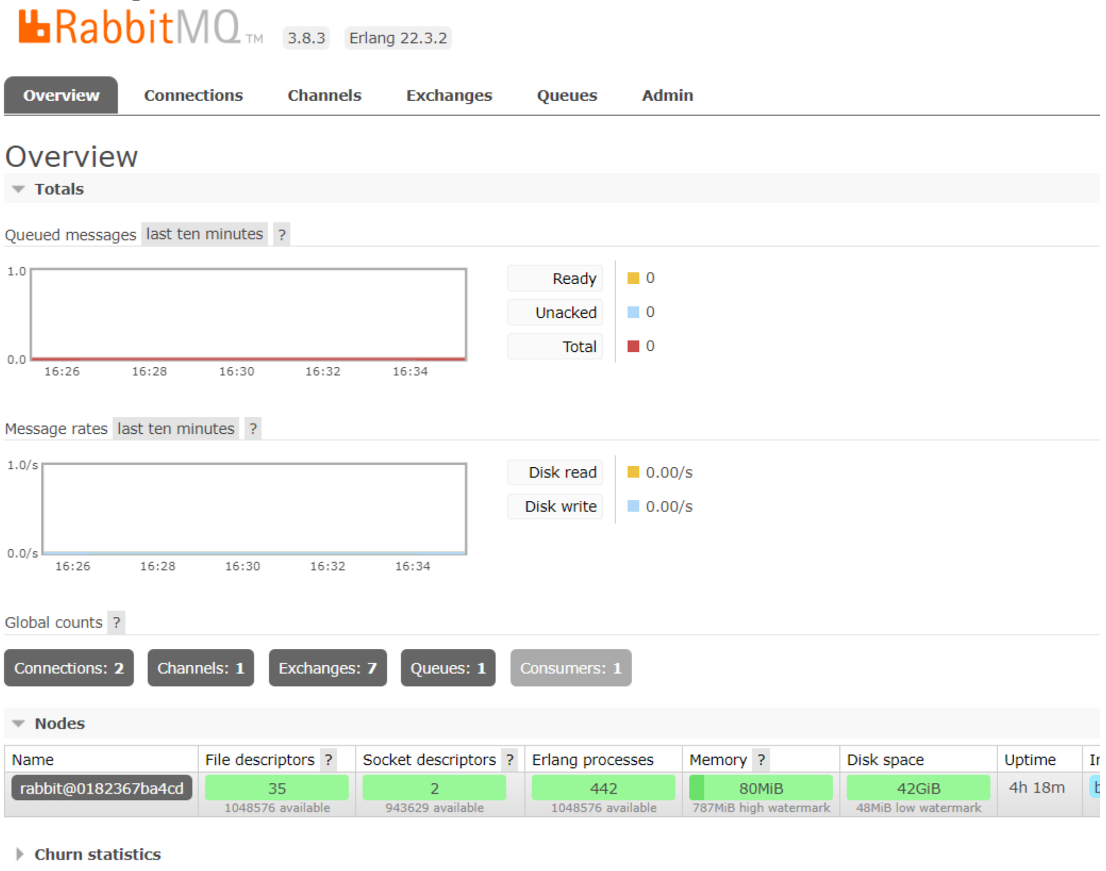
Here is the docker commands that basically download RabbitMQ in your local and use make collections.

In order to download and run RabbitMQ from docker hub use below commands;

```
	docker run -d --hostname my-rabbit --name some-rabbit -p 15672:15672 -p
5672:5672 rabbitmq:3-management
```

After this command you can watch your events from RabbitMQ Dashboard in 15672 port;

http://localhost:15672/#/queues
username: guest
password: guest



Ready	🟨 0
Unacked	🟦 0
Total	🟥 0

Disk read	🟨 0.00/s
Disk write	🟦 0.00/s

Connections: 2 Channels: 1 Exchanges: 7 Queues: 1 Consumers: 1

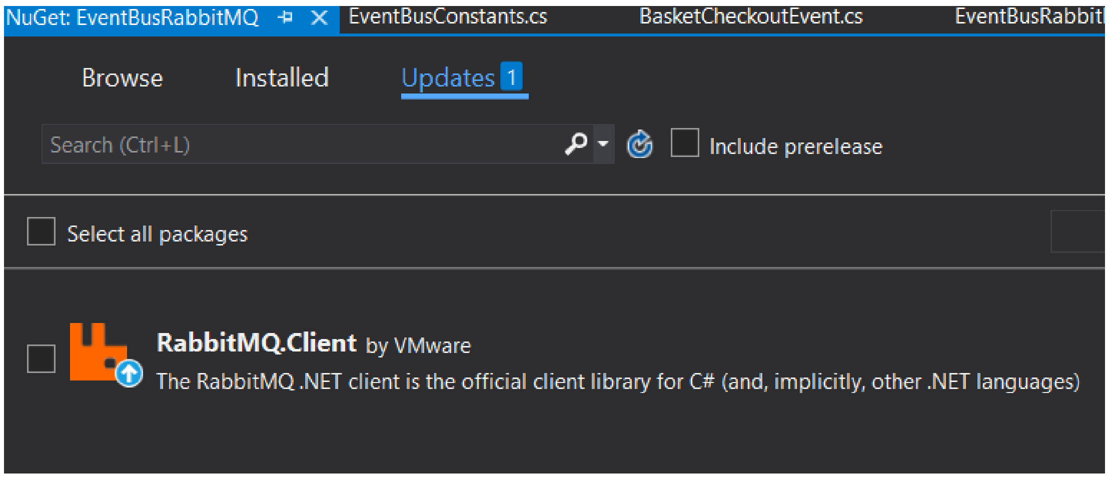
Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Ir
rabbit@0182367ba4cd	35 1048576 available	2 943629 available	442 1048576 available	80MiB 787MiB high watermark	42GiB 48MiB low watermark	4h 18m	b

The above image includes RabbitMQ dashboard metrics.

Library & Frameworks

For EventBusRabbitMQ Class Library project, we have to libraries in our Nuget Packages,

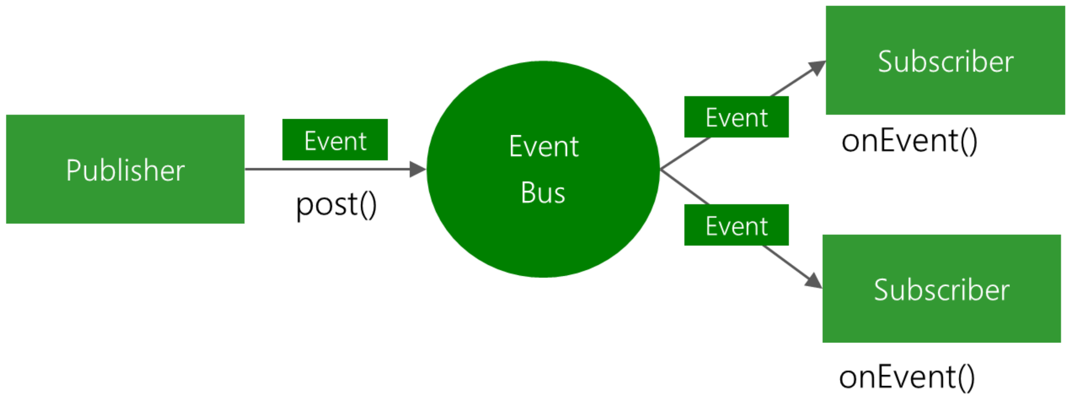
- RabbitMQ.Client – To connect RabbitMQ



RabbitMQ.Client by VMware
The RabbitMQ .NET client is the official client library for C# (and, implicitly, other .NET languages)

Developing Application

Instead of using RabbitMQ methods directly, we will encapsulate and write **wrapper EventBus structure** over RabbitMQ in order to manage queue operations.



Create new Class Library project with Visual Studio.

First, open **File -> New -> Project**. Select Class Library, give your project a name as a **EventBusRabbitMQ** and select OK.

Add New Class -> IRabbitMQConnection.cs

```
IRabbitMqConnection.cs
  public interface IRabbitMQConnection : IDisposable
    {
        bool IsConnected { get; }
        bool TryConnect();
        IModel CreateModel();
    }
```

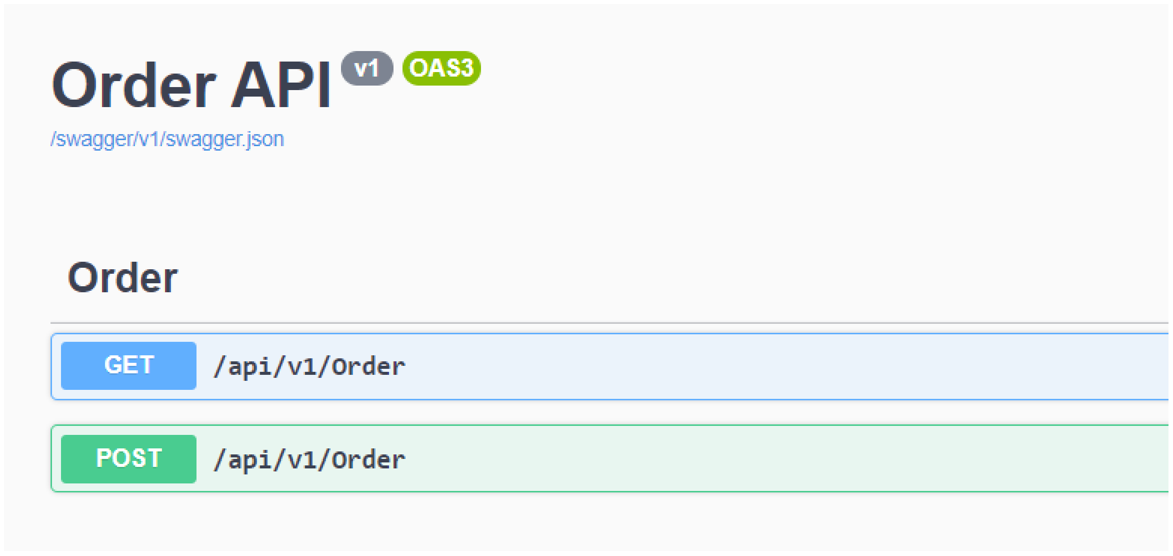
SECTION 5

Building Ordering Microservices with Clean Architecture and CQRS

In this section we will show how to perform Ordering microservices operations on ASP.NET Core Web application using **Entity Framework Core with Sql Server** Database applying **Clean Architecture** and **CQRS**.

By the end of the section, we will have a Web API which implemented basic CRUD operations over Order entity and perform as a **RabbitMQ Listener which consume basketCheckout events from Basket Microservices**.

Look at the final swagger application.



You'll learn how to Create **Ordering microservice** which includes;

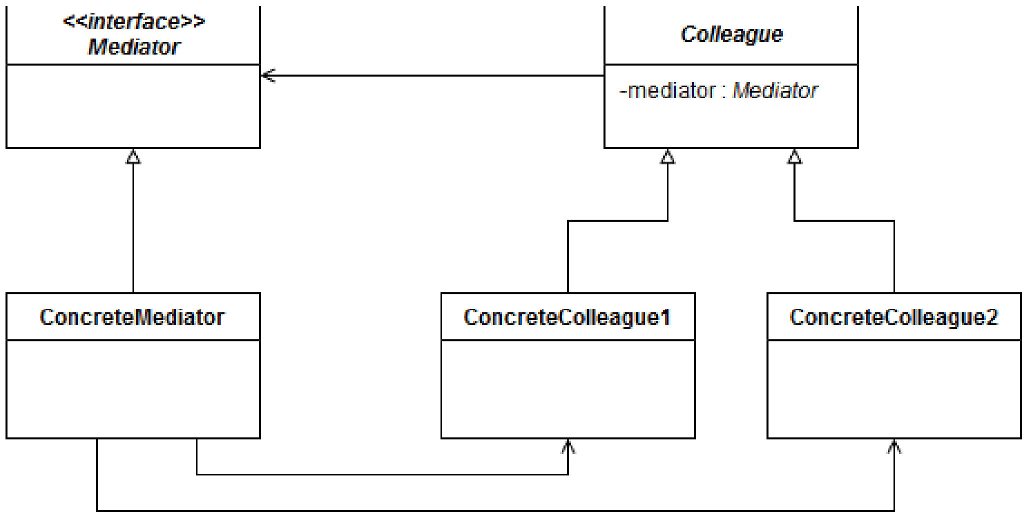
- ASP.NET Core **Web API** application
- **REST** API principles, CRUD operations
- **Entity Framework Core** code-first approach
- **SQL Server database** connection on docker
- **Clean Architecture** implementation with applying **SOLID principles**
- **CQRS implementation** on commands and queries
- N-Layer **Hexagonal architecture** (Core, Application, Infrastructure and Presentation Layers)
- **Domain Driven Design** (Entities, Repositories, Domain/Application Services, DTO's...)
- **Swagger** Open API implementation
- **Dockerfile** implementation

Background

Before we start, we will start with definitions and explanations of Ordering microservices on Asp.Net Core Web API.

Mediator Design Pattern

As can be understood from the name of **Mediator**, it is a class created with the logic of performing this operation by using an **intermediate class** to perform the connection and other operations between the classes derived from the same interface.



One of the most frequently given examples is the tower structure, which gives the permission of the aircraft at the airports. Airplanes do not get permission from each other. All airplanes only get permission from the tower and the operations take place accordingly. The vehicle here becomes the tower.

We will use this **Mediator Pattern** when using **MediatR** nuget packages in **Ordering.Application** project.

- **MediatR** – To implement Mediator Pattern

Analysis & Design

This project will be the REST APIs which basically perform CRUD operations on Ordering databases. And act as a **RabbitMQ Listener** for **basketCheckout** events, with this event trigger to **CQRS command handlers**.

Our main use cases;

- Get Orders with username
- Consume **basketCheckout event** from Rabbit MQ and trigger to **OrderCommand** which provide to **insert** Order record into **Sql Server** database

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

Method	**Request URI**	**Use Case**
GET	api/v1/Order	Get Orders with username

Architecture

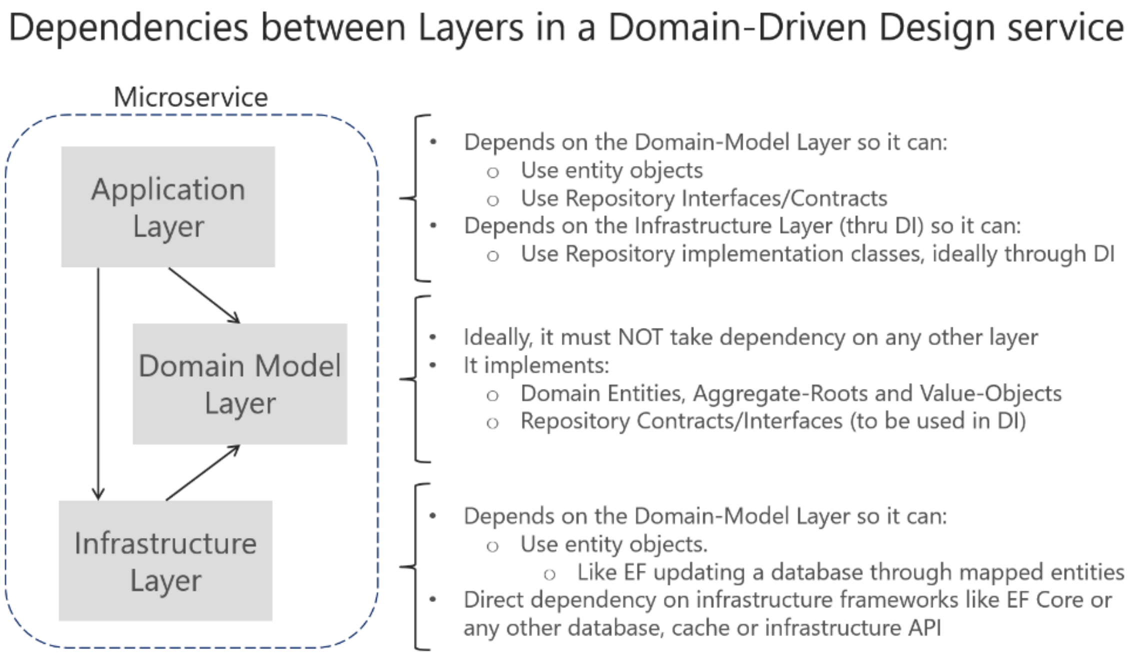
Ordering microservice use **ASP.NET Core Web API reference application** with **Entity Framework Core**, demonstrating a layered application architecture with DDD best practices. Implements N-Layer **Hexagonal architecture** (Core, Application, Infrastructure and Presentation Layers) and **Domain Driven Design** (Entities, Repositories, Domain/Application Services, DTO's...) and aimed to be a **Clean Architecture**, with applying **SOLID principles** in order to use for a project template.

Also implements **CQRS Design Pattern** into their layers in order to separate **Queries** and **Command** for Ordering microservice.

Before start with the architecture, we should know the **Design Principles** and some of the **Design Patterns** over the clean architecture.

Domain Driven Design (DDD)

Domain Driven Design (DDD) is not an improved technology or specific method. Domain Driven Design (DDD) is an approach that tries to bring solutions to the basic problems frequently experienced in the development of **complex software systems** and in ensuring the continuity of our applications after the implementation of these opposing projects. To understand Domain Driven Design (DDD), some basic concepts need to be mastered. With these concepts, we can enter the Domain Driven Design (DDD).


Dependencies between Layers in a Domain-Driven Design service
Microservice
Application Layer
Domain Model Layer
Infrastructure Layer
• Depends on the Domain-Model Layer so it can:
o Use entity objects
o Use Repository Interfaces/Contracts
• Depends on the Infrastructure Layer (thru DI) so it can:
o Use Repository implementation classes, ideally through DI
• Ideally, it must NOT take dependency on any other layer
• It implements:
o Domain Entities, Aggregate-Roots and Value-Objects
o Repository Contracts/Interfaces (to be used in DI)
• Depends on the Domain-Model Layer so it can:
o Use entity objects.
o Like EF updating a database through mapped entities
• Direct dependency on infrastructure frameworks like EF Core or any other database, cache or infrastructure API

Ubiquitous Language
It is one of the **cornerstones** of Domain Driven Design (DDD). We need to be able to produce the desired output of the software developers and to ensure the

continuity of this output to be able to **speak the same language** as the **Domain Expert.** Afterwards, we must transfer this experience to the methods and classes that we use while developing our applications by giving the names of the concepts used by experts. Every service we will use in our project must have a response in the domain. Thus, everyone involved in the project can **speak this common language** and understand each other.

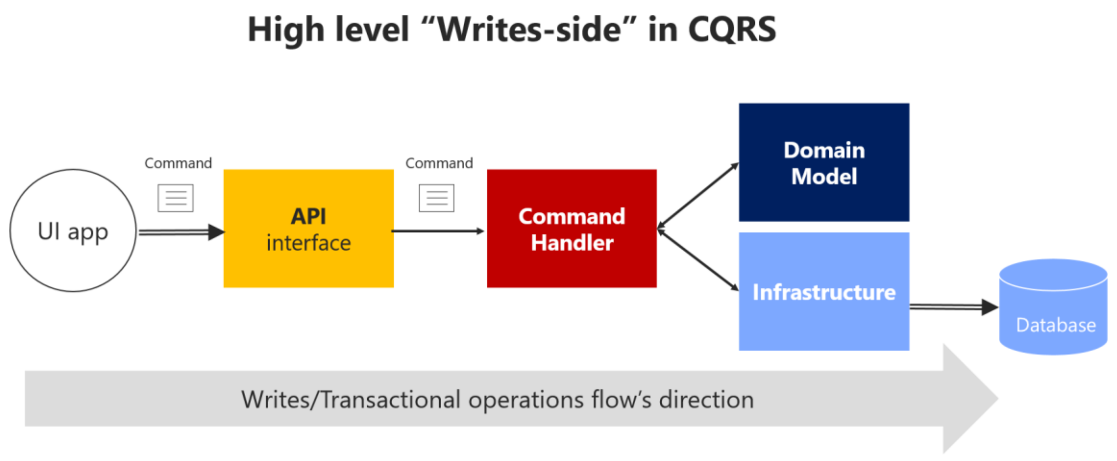
Bounded Context
A recommended approach to use in Domain Driven Design (DDD) complex systems. The complex may **contain sub domains** within a domain. It should also include Domain Driven Design (DDD). Example of e-commerce site; · Order management · Customer management · Stock management · Delivery management · Payment System management · Product management · User management may contain many sub domains. As these **sub domains are grouped, Bounded Context** refers to structures in which the group of individuals most logically associated with each other in terms of the rules of the **Aggregate Roots** are **grouped together** and the **responsibilities** of this group are clearly defined.

CQRS (Command Query Responsibility Segregation) Design Pattern

CQRS means **separation of command** and **query responsibilities**. Although it has become an increasingly popular pattern in recent years, it has increased in popularity after this article written by Martin Fowler when he heard this pattern by Greg Young.

We can say that it is based on the **CQS (Command Query Separation)** principle for the CQRS architecture. The main idea of CQS is to separate the interfaces between our operations that read the data and the operations that update the data. In CQRS, this is added to the separation of our business models.

CQRS is a software development pattern based on conducting reading and writing / updating processes on different models. The data you read and the data you write are stored in **different database tools**.



Command - First of all, command type is the only way to change something in the system. If there is no command, the state of the system remains unchanged. Command types should not return any value. Command types often act as plain objects that are sent to **CommandHandler** as parameters when used with **CommandHandler** and **CommandDispatcher** types.

Query - The only way to do similar reading is the type of Query. They cannot change the state of the system. They are generally used with **QueryHandler** and **QueryDispatcher** types.

After doing Command and Query parsing in our system, it will be clear that the domain model we use for data writing is not suitable for reading data.

Ordering Microservices follow the CQRS and DDD principles as below figure;

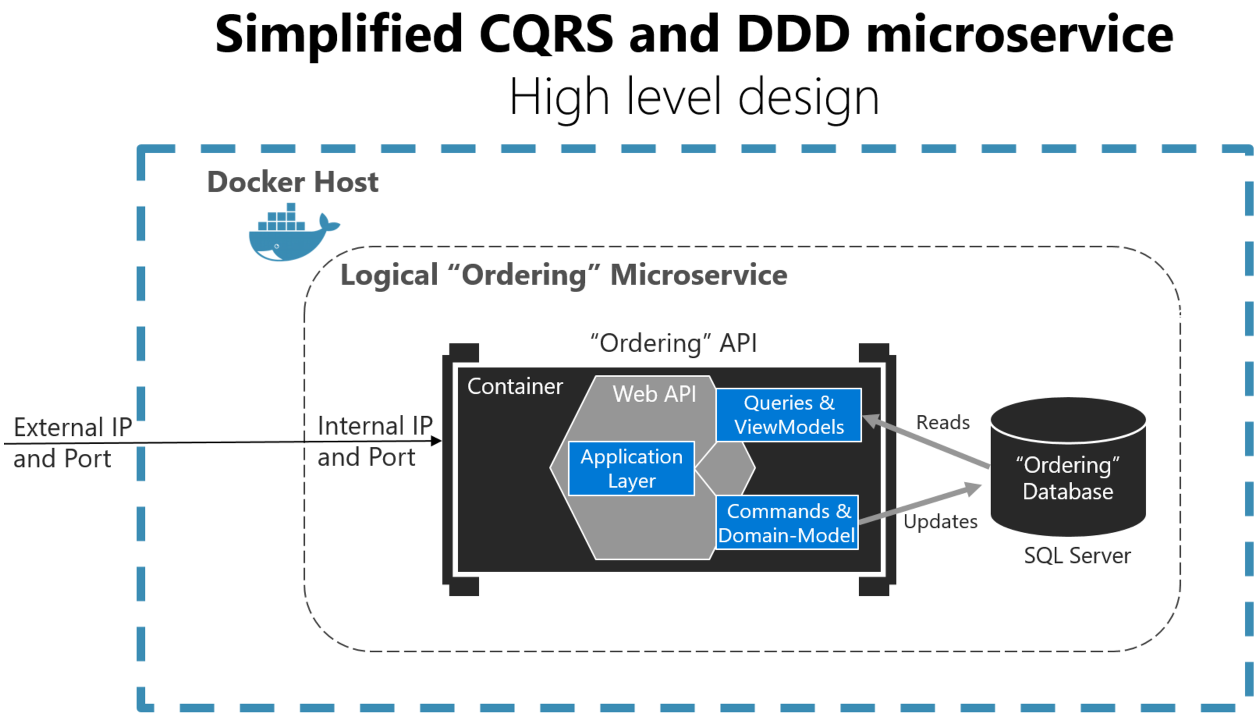


Figure: Ordering CQRS- and DDD-based microservice

So, after these definitions, we can start with set-up our Data Source Layer that means, create **Sql Server Database** docker in our docker desktop and learn basics commands on command line.