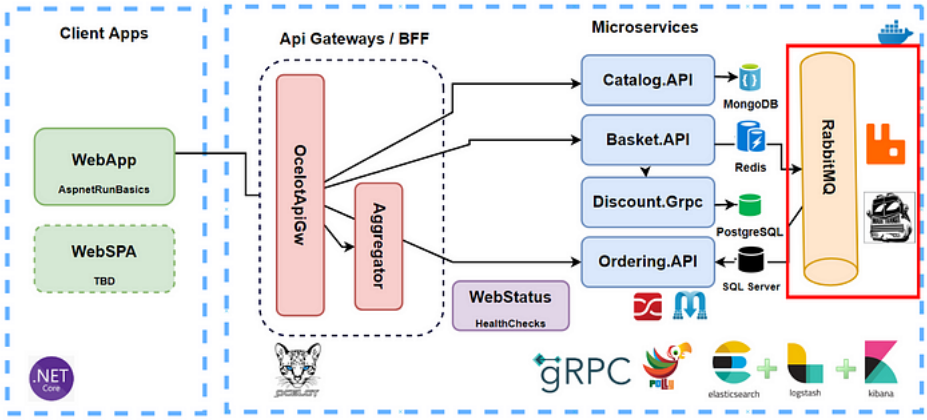
Microservices Event Driven Architecture with RabbitMQ and Docker Container on .NET

Building Microservices Async Communication w/ RabbitMQ & MassTransit
for Checkout Order use cases Between Basket-Ordering Microservices with docker-compose.

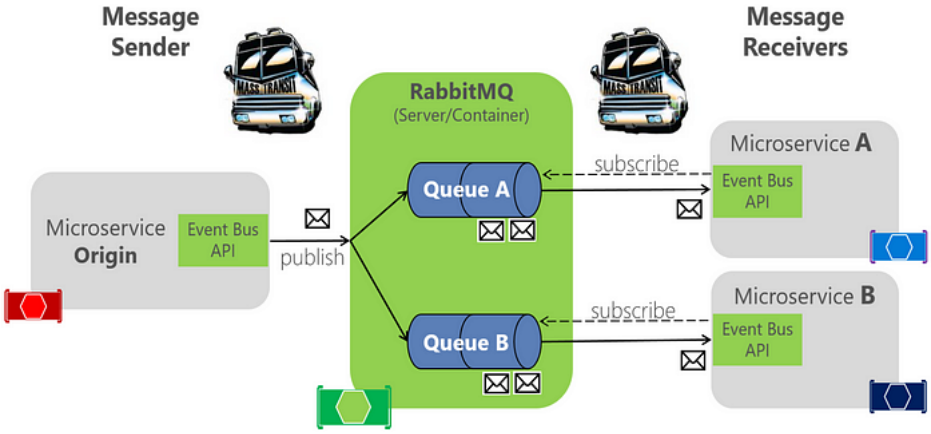


Introduction

In this article we will show how to perform **RabbitMQ connection** on **Basket** and **Ordering** microservices when producing and consuming events. Developing Microservices Async Communication w/ **RabbitMQ & MassTransit** for **Checkout Order** use cases Between Basket-Ordering Microservices.

By the end of the article, we will have a library which provide to **communication along Basket and Ordering microservices** with creating a **EventBus.Messages Common Class Library** project.

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



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

- Async **Microservices Communication** with **RabbitMQ Message-Broker Service**
- Using **RabbitMQ Publish/Subscribe Topic Exchange Model**
- Using **MassTransit** for abstraction over **RabbitMQ** Message-Broker system
- **Publishing BasketCheckout event** queue from Basket microservices
  and **Subscribing** this event from **Ordering microservices**
- Create **RabbitMQ EventBus.Messages Common Class Library** and add references Microservices
- **Containerize** RabbitMQ Message Queue system with Basket and Ordering microservices using Docker Compose

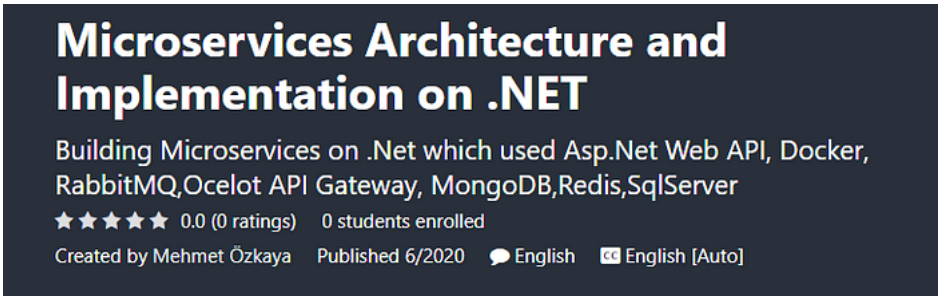
Background

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

Pub/Sub RabbitMQ Architecture

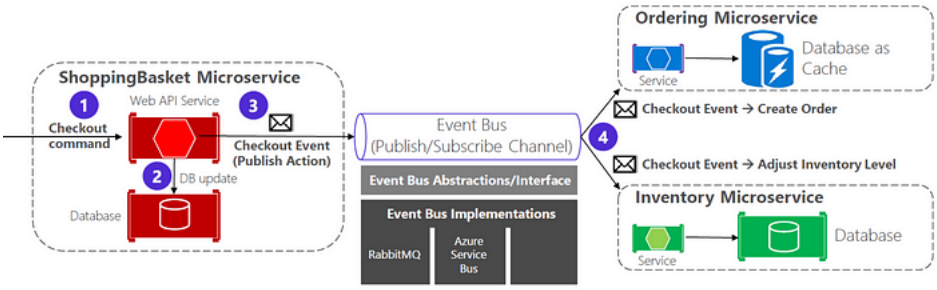
We will Analysis and Architecting of Microservices Async Communication w/ **RabbitMQ** & **MassTransit**.
We will create RabbitMQ connection on Basket and Ordering microservices when producing and consuming events.

Basket microservice which includes;
— **Publish BasketCheckout Queue** with using MassTransit and RabbitMQ

Ordering Microservice which includes;
— **Consuming RabbitMQ BasketCheckout event** queue with using MassTransit-RabbitMQ Configuration

Publisher/Subscriber of BasketCheckout Event w/ Basket and Ordering Microservices

Here is the another view of Publisher/Subscriber of BasketCheckout Event.
This is the E2E use case of BasketCheckout event.



1- **BasketCheckout command** will comes from the client application
2- Basket microservices perform their operations like **removing basket** in redis database, because this is going to be an order
3- Basket microservices **publish BasketCheckout event** to the **RabbitMQ** with using **MassTransit**
4- This queue typically implemented with **topic messaging thecnology**
So Subscriber microservices, in our case it will be Ordering microservices will **consume** this **event** and **Create an order** in their **sql server database.**

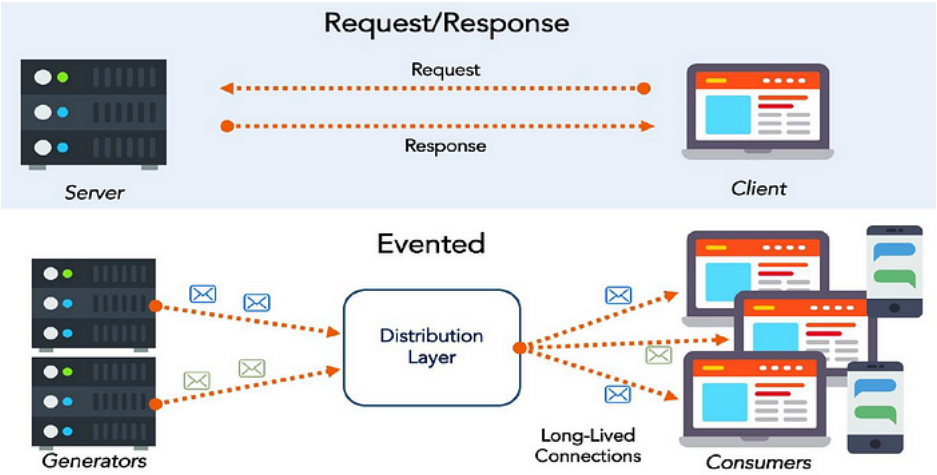
By the end of the section, we will have a library which provide to communication along Basket and Ordering microservices with creating a class library project.
The event bus implementation with RabbitMQ that microservices publish events, and receive events.

Before we start to developments, we should understand the microservices communications and RabbitMQ.

Microservices Communication Types Request-Driven or Event-Driven Architecture

Here you can see the microservices communication types.

Microservices communications can be divide by 3;
Request-Driven Architecture
Event-Driven Architecture
Hybrid Architecture



Request-Driven Architecture

In the request-response based approach, services communicate using HTTP or RPC. In most cases, this is done using REST HTTP calls. But we have also developed that gRPC communication with Basket and Discount microservices. That communication was Request-Driven Architecture.

Benefits
There is a clear control of the flow, looking at the code of the orchestrator, we can determine the sequence of the actions.
Tradeoffs
If one of the dependent services is down, there is a high chance to exclude calls to the other services.

Event-Driven Architecture

This communication type microservices dont call each other, instead of that they created events and consume events from message broker systems in an async way. In this section we will follow this type of communication for
Basket microservice
Publish BasketCheckout Queue with using MassTransit and RabbitMQ
Ordering Microservice
Consuming RabbitMQ BasketCheckout event queue with using MassTransit-RabbitMQ Configuration

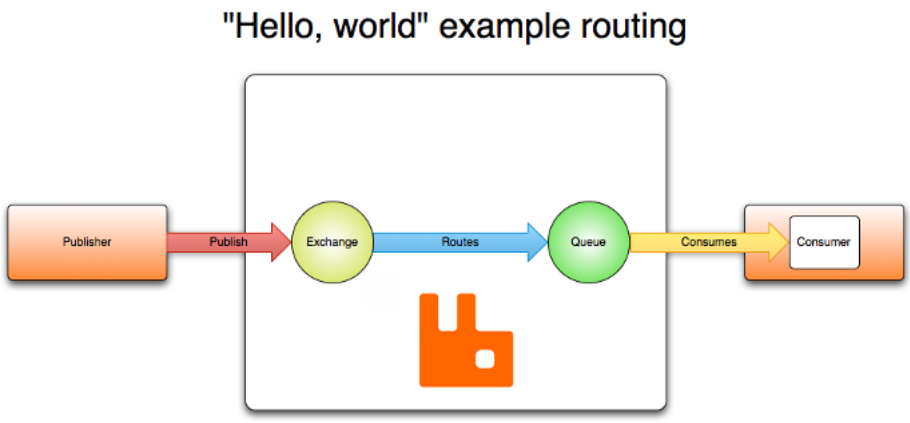
Benefits
The producer service of the events does not know about its consumer services. On the other hand, the consumers also do not necessarily know about the producer. As a result, services can deploy and maintain independently. This is a key requirement to build loosely coupled microservices.

Tradeoffs
There is no clear central place (orchestrator) defining the whole flow.

And the last one is using hybrid architecture. So it means that depending on your custom scenario, you can pick one of this type of communication and perform it.
So in our reference application, we also choose Request-Driven Architecture when communication Basket and Discount with gRPC calls.
And now we are going to use Event-Driven Architecture for producing and consuming basket checkout event between Basket and Ordering microservices.

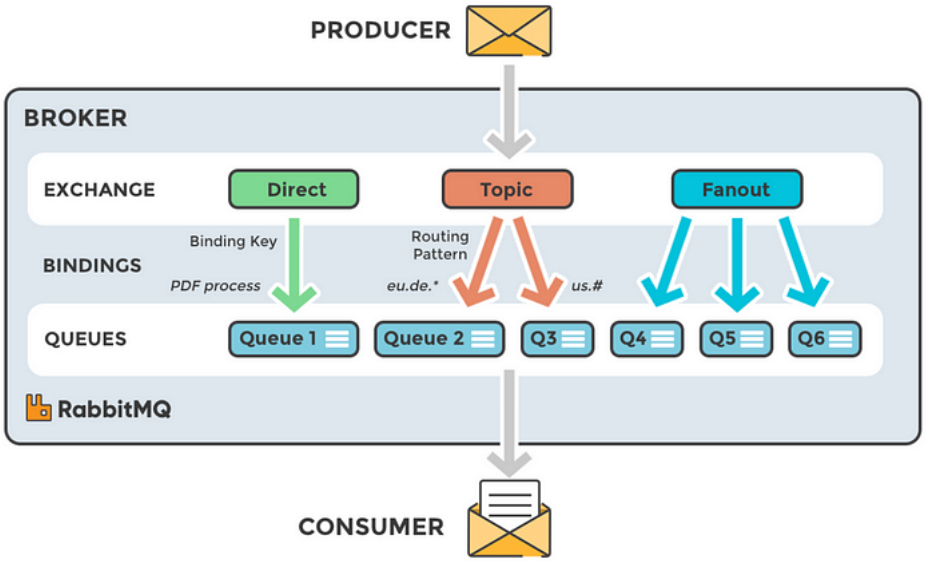
RabbitMQ

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.



Main Logic of RabbitMQ

Producer: The source of the message is the application.
Queue: Where messages are stored. The sent messages are put in a queue before they are received. All incoming messages are stored in Queue, that is memory.

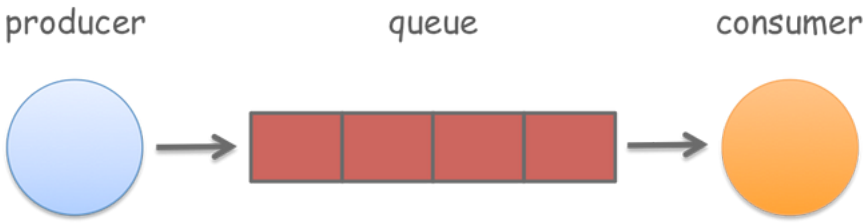


Consumer: It is the server that meets the sent message. It is the application that will receive and process the message on the queue.

Message: The data we send on the queue.

Exchange: It is the structure that decides which queues to send the messages. It makes the decision according to routing keys.

Binding: The link between exchance and queue.
FIFO: The order of processing of outgoing messages in RabbitMQ is first in first out.



The Producer project sends a message to be queued. The message is received by the Exchange interface and redirects to one or more queues according to various rules.

Queue Properties

Name: The name of the queue we have defined.

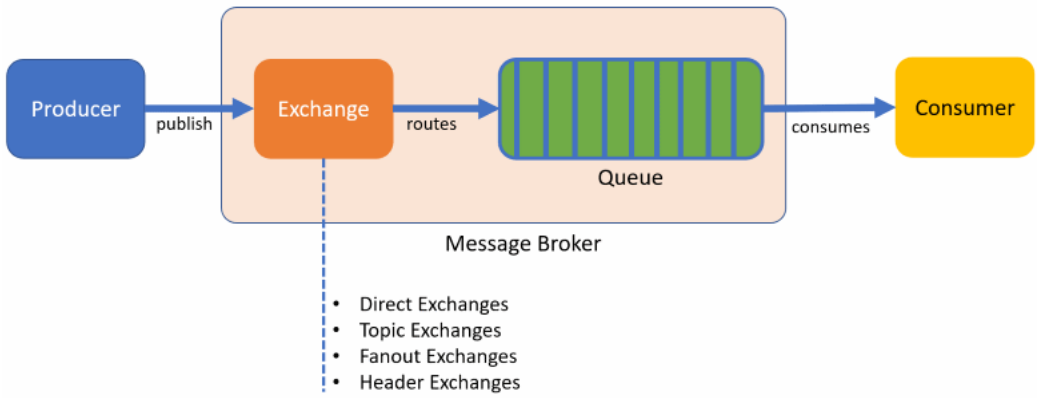
Durable: Determines the lifetime of the queue. If we want persistence, we have to set it true. We use it in-memory in the project. In this case, the queue will be deleted when the broker is restart.

Exclusive: Contains information whether the queue will be used with other connections.

AutoDelete: Contains information about deletion of the queue with the data sent to the queue passes to the consumer side.

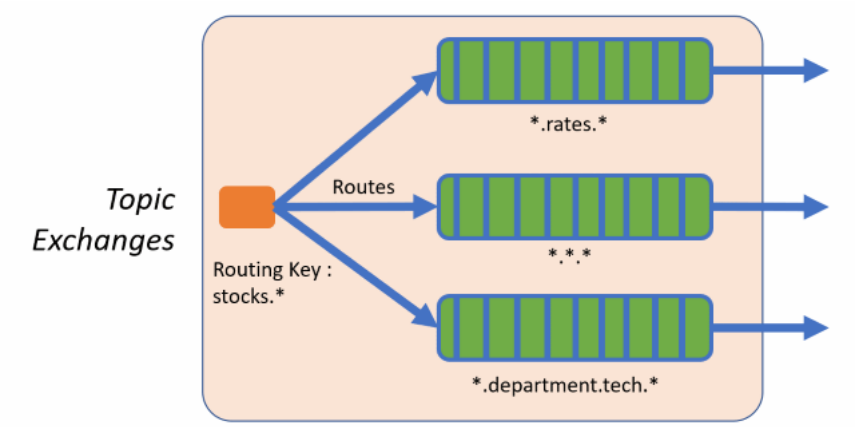
RabbitMQ Exchange Types

RabbitMQ is based on a messaging system like below.

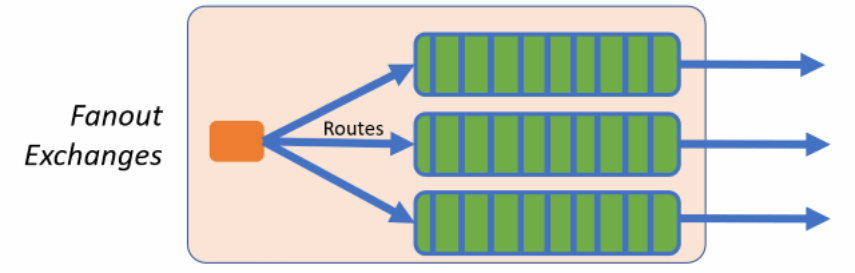


Direct Exchange: The use of a single queue is being addressed. A routing key is determined according to the things to be done and accordingly, the most appropriate queue is reached with the relevant direct exchange.

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 RabbitMQ & BuildingBlocks EventBus.Messages

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 are going to make Async communication between Basket and Ordering microservices with using RabbitMQ and MassTransit. When basket checkout operation performed, we are going to create basketcheckout event and consume from Ordering microservices.

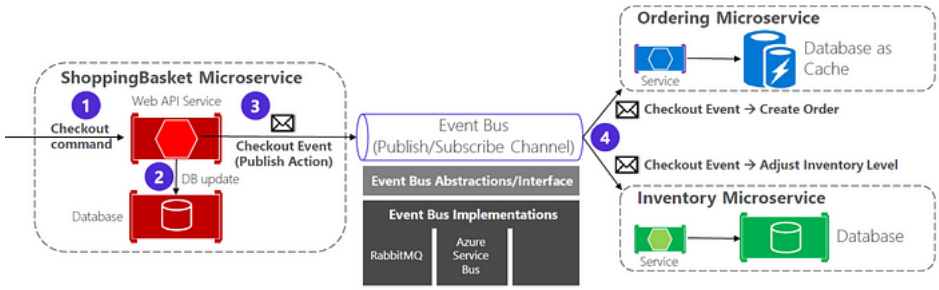
MassTransit

Let me give some brief information about MassTransit. An open-source lightweight message bus framework for .NET. MassTransit is useful for routing messages over MSMQ, RabbitMQ, TIBCO, and ActiveMQ service busses, with native support for MSMQ and RabbitMQ. MassTransit also supports multicast, versioning, encryption, sagas, retries, transactions, distributed systems, and other features. We are going to use the publish/sbuscribe feature over the Rabbitmq for our application for publishing the basket checkout orders.

So we are going to develop BuildingBlocks EventBus.Messages Class Library.
Inside of this common class libary we will create BasketCheckoutEvent event class.

Publisher/Subscriber of BasketCheckout Event

Here is the another view of Publisher/Subscriber of BasketCheckout Event.
This is the E2E use case of BasketCheckout event.



1- **BasketCheckout command** will comes from the client application
2- Basket microservices perform their operations like **removing basket** in redis database, because this is going to be an order
3- Basket microservices **publish BasketCheckout event** to the **RabbitMQ** with using **MassTransit**
4- This queue typically implemented with **topic messaging thecnology**
So Subscriber microservices, in our case it will be Ordering microservices will **consume** this **event** and **Create an order** in their **sql server database.**

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

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 if you run the **docker ps** command, the rabbitMQ container, the details of which can be found.

Or we can add RabbitMQ image into Docker-Compose File for Multi-Container Docker Environment.

Now we can add rabbitmq image into our docker-compose.yml files

```
docker-compose.ymlrabbitmq:
 image: rabbitmq:3-management-alpinedocker-compose.override.ymlrabbitmq:
 container_name: rabbitmq
 restart: always
 ports:
 – “5672:5672”
 – “15672:15672”
```

— We set rabbitmq configuration as per environment variables.

Finally, we can create RabbitMq image for our Basket and Ordering microservices.

Open In Terminal, RUN with below command on that location;

```
docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d
```

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

http://localhost:15672/#/queues

username: guest

password: guest

The above image includes RabbitMQ dashboard metrics.

Library & Frameworks

Setting Up The The Publisher Microservice

- Install-Package MassTransit
- Install-Package MassTransit.RabbitMQ
- Install-Package MassTransit.AspNetCore

Developing BuildingBlocks EventBus.Messages Class Library

We are going to Develop BuildingBlocks EventBus.Messages Class Library. We are going to create below folder structure — Shared Model Library.

BuildingBlocks
EventBus.Messages
IntegrationEvent
BasketCheckoutEvent

Create "**Events**" folder

Add New Class — IntegrationBaseEvent.cs

```
public class IntegrationBaseEvent
 {
 public IntegrationBaseEvent()
 {
 Id = Guid.NewGuid();
 CreationDate = DateTime.UtcNow;
 }

 public IntegrationBaseEvent(Guid id, DateTime createDate)
 {
 Id = id;
 CreationDate = createDate;
 }

 public Guid Id { get; private set; }

 public DateTime CreationDate { get; private set; }
 }
```

Now we can create our basket checkout event in here.
We create this event in the class library because this data will be shared for Basket and Ordering microservices.

Add New Class
BasketCheckoutEvent.cs

```
public class BasketCheckoutEvent : IntegrationBaseEvent
 {
 public string UserName { get; set; }
 public decimal TotalPrice { get; set; }// BillingAddress
```

```
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public string EmailAddress { get; set; }
 public string AddressLine { get; set; }
 public string Country { get; set; }
 public string State { get; set; }
 public string ZipCode { get; set; }// Payment
 public string CardName { get; set; }
 public string CardNumber { get; set; }
 public string Expiration { get; set; }
 public string CVV { get; set; }
 public int PaymentMethod { get; set; }
 }
```

Produce RabbitMQ Event From Basket Microservice Publisher of BasketCheckoutEvent

We are going to Produce RabbitMQ Event From Basket Microservice Publisher of BasketCheckoutEvent.

We are going to make Async communication between Basket and Ordering microservices with using RabbitMQ and MassTransit.
When basket checkout operation performed, we are going to create basketcheckout event and consume from Ordering microservices.
In this section, we are focusing on producing event in Basket microservices.

Before we start, we need to add project refernces.

Go to Basket.API

Add Project Reference
EventBus.Messages

This will also adding project references on Ordering.API later — common library for eventbus messages.

Install required Nuget Packages — Setting Up The The Publisher Microservice

- Install-Package MassTransit
- Install-Package MassTransit.RabbitMQ
- Install-Package MassTransit.AspNetCore

We need to configure MassTransit in order to connect with RabbitMQ in our aspnet project.

Configuring MassTrasit — Add DI Configuration

Basket.API — Startup.cs

```
// MassTransit-RabbitMQ Configuration
    services.AddMassTransit(config => {
        config.UsingRabbitMq((ctx, cfg) => {
            cfg.Host(Configuration["EventBusSettings:HostAddress"]);
        });
    });appsettings.json"EventBusSettings": {
    "HostAddress": "amqp://guest:guest@localhost:5672"
  },
```

Adds the MassTransit Service to the ASP.NET Core Service Container.
Creates a new Service Bus using RabbitMQ. Here we pass paramteres like the host url, username and password. Don’t forget to move host name as a configuration value from appsettings json file.

Publish BasketCheckout Queue Message Event in Basket.API Controller Class

We are going to Publish BasketCheckout Queue Message Event in Basket.API Controller Class.

Developing BasketCheckout API Method — BasketController.cs

```
private readonly IBasketRepository _repository;
 private readonly DiscountGrpcService _discountGrpcService;
 private readonly IPublishEndpoint _publishEndpoint;
 private readonly IMapper _mapper;public BasketController(IBasketRepository repository, DiscountGrpcService discountGrpcService, IPublishEndpoint publ
 {
 _repository = repository ?? throw new ArgumentNullException(nameof(repository));
 _discountGrpcService = discountGrpcService ?? throw new ArgumentNullException(nameof(discountGrpcService));
 _publishEndpoint = publishEndpoint ?? throw new ArgumentNullException(nameof(publishEndpoint));
 _mapper = mapper ?? throw new ArgumentNullException(nameof(mapper));
 }[Route(“[action]”)]
 [HttpPost]
 [ProducesResponseType((int)HttpStatusCode.Accepted)]
 [ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<IActionResult> Checkout([FromBody] BasketCheckout basketCheckout)
 {
 // get existing basket with total price
 // Set TotalPrice on basketCheckout eventMessage
 // send checkout event to rabbitmq
 // remove the basket// get existing basket with total price
 var basket = await _repository.GetBasket(basketCheckout.UserName);
```

```
if (basket == null)
{
return BadRequest();
}// send checkout event to rabbitmq
var eventMessage = _mapper.Map<BasketCheckoutEvent>(basketCheckout);
eventMessage.TotalPrice = basket.TotalPrice;
await _publishEndpoint.Publish(eventMessage);// remove the basket
await _repository.DeleteBasket(basket.UserName);return Accepted();
}
```

Publishing an event to the rabbit mq is very easy when you are using masstransit. In order to publish messages to rabbitmq, the important object is IPublishEndpoint.

As you can see that we have Published BasketCheckout Queue Message Event in Basket.API Controller Class.

Consume RabbitMQ Event From Ordering Microservice Subscriber of BasketCheckoutEvent

We are going to Consume RabbitMQ Event From Ordering Microservice which is Subscriber of BasketCheckoutEvent.

We are going to make Async communication between Basket and Ordering microservices with using RabbitMQ and MassTransit.
When basket checkout operation performed, we are going to create basketcheckout event and consume from Ordering microservices.

In the last section, we have developed for publishing Basket Checkout Queue from Basket API. In this section, we are focusing on consuming event in Ordering microservices.

Go to Ordering.API — Add Project Reference

- EventBus.Messages

Setting Up The The Publisher Microservice

- Install-Package MassTransit
- Install-Package MassTransit.RabbitMQ
- Install-Package MassTransit.AspNetCore

We need to configure MassTransit in order to connect with RabbitMQ in our aspnet project.

Add DI Configuration — Ordering.API — Startup.cs

```
// MassTransit-RabbitMQ Configuration
services.AddMassTransit(config => {config.AddConsumer<BasketCheckoutConsumer>();config.UsingRabbitMq((ctx, cfg) => {
cfg.Host(Configuration[“EventBusSettings:HostAddress”]);

cfg.ReceiveEndpoint(EventBusConstants.BasketCheckoutQueue, c => {
c.ConfigureConsumer<BasketCheckoutConsumer>(ctx);
});
});
});
services.AddMassTransitHostedService();appsettings.json“EventBusSettings”: {
“HostAddress”: “amqp://guest:guest@localhost:5672”
},
```

Now we can create consumer class.

— **Ordering.API -BasketCheckoutConsumer**

Create Folder
“**EventBusConsumer**”

Add Class — BasketCheckoutConsumer

```
public class BasketCheckoutConsumer : IConsumer<BasketCheckoutEvent>
{
public async Task Consume(ConsumeContext<BasketCheckoutEvent> context)
{
var command = _mapper.Map<CheckoutOrderCommand>(context.Message);
var result = await _mediator.Send(command);_logger.LogInformation(“BasketCheckoutEvent consumed successfully. Created Order Id : {newOrderId}”, resul
}
}
```

As you can see that we have successfully Subscribed BasketCheckout Queue Message Event in Ordering.API BasketCheckoutConsumer Class.

Test BasketCheckout Event in Basket.API and Ordering.API Microservices

We are going to Test BasketCheckout Event in Basket.API and Ordering.API Microservices.

Before we start to test, verify that rabbitmq docker image is running well. Also it is good practice to run docker-compose command in order to validate to work all microserices what we have developed so far.

Open in Terminal
Start Docker

```
docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d
```

RUN on DOCKER
See from Rabbit Management Dashboard
http://localhost:15672

Testing

First we need to create basket
POST
/api/v1/Basket

```
POST PAYLOAD
{
 “UserName”: “swn”,
 “Items”: [
 {
 “Quantity”: 2,
 “Color”: “Red”,
 “Price”: 500,
 “ProductId”: “60210c2a1556459e153f0554”,
 “ProductName”: “IPhone X”
 },
 {
 “Quantity”: 1,
 “Color”: “Blue”,
 “Price”: 500,
 “ProductId”: “60210c2a1556459e153f0555”,
 “ProductName”: “Samsung 10”
 }
 ]
}returns
 {
 “userName”: “swn”,
…
 “totalPrice”: 1100
 }
```

After that checkout the basket
POST
/api/v1/Basket – checkoutPOST PAYLOAD

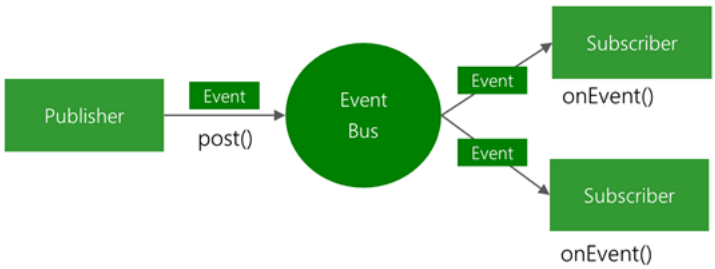
```
 {
 “userName”: “swn”,
 “totalPrice”: 0,
 “firstName”: “swn”,
 “lastName”: “swn”,
 “emailAddress”: “string”,
 “addressLine”: “string”,
 “country”: “string”,
 “state”: “string”,
 “zipCode”: “string”,
 “cardName”: “string”,
 “cardNumber”: “string”,
 “expiration”: “string”,
 “cvv”: “string”,
 “paymentMethod”: 1
 }
```

Check RabbitMQ Dashboard
http://localhost:15672/#/

— See that exchange types created successfully.
— See Queues
Queue basketcheckout-queue

— See exchanges and bindings.

SUCCESS !!!



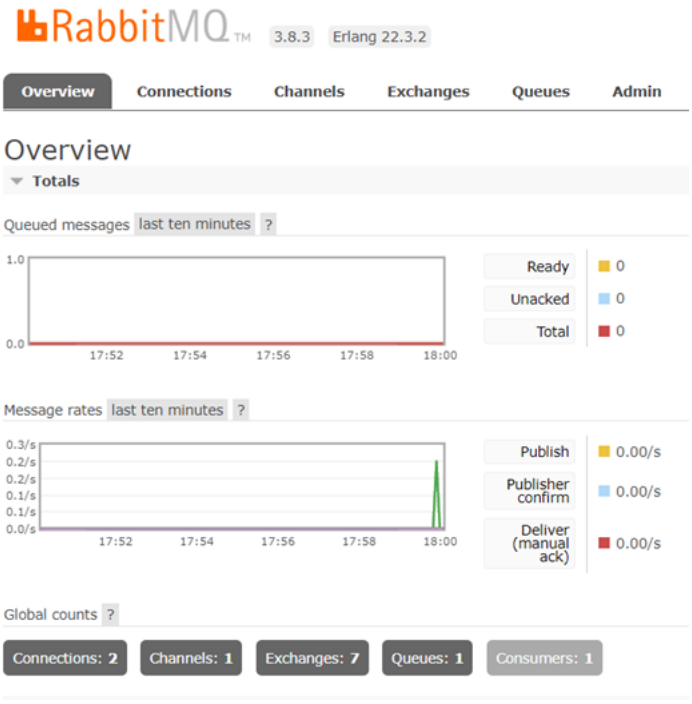
As you can see that we have successfully Tested BasketCheckout Event publish/subscriber models in Basket.API and Ordering.API Microservices.

Conclusion

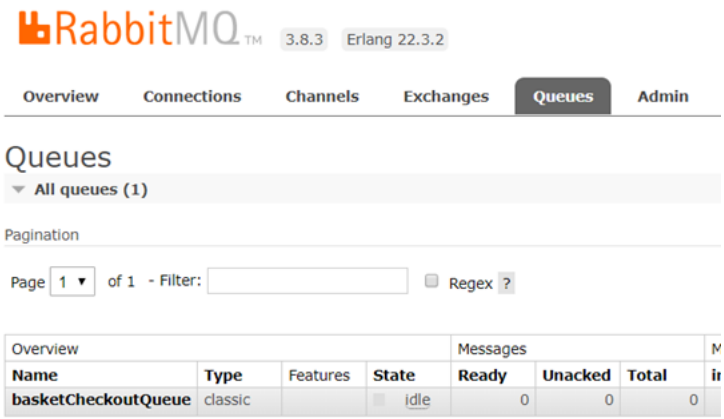
See from **Rabbit Management Dashboard;**

http://localhost:15672/#/queues/%2F/basketCheckoutQueue

The queue should be pop 1 and will consume from Ordering microservices.



Queue Record created as below name what we defined our code.



We created a library which provide to **communication along Basket and Ordering microservices** with creating a class library project.

Also we saw that the **event bus implementation** with RabbitMQ & MassTransit that **microservices publish** events, and **receive** events.