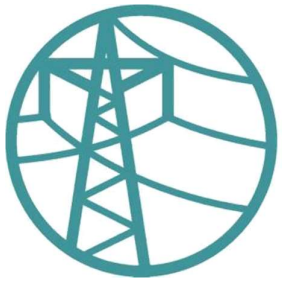


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Temat projektu:

Integracja microservices za pomocą RabbitMQ z wykorzystaniem języka C#

Imię i nazwisko: Szymon Fornal
Nr albumu: 168141
Rok: 4EF-ZI

Rzeszów, 2024

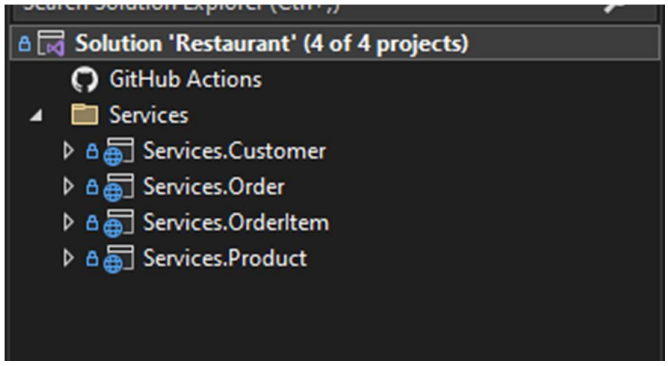
Spis treści

Wstęp

W tej dokumentacji zostanie przedstawiony przewodnik dotyczący integracji mikroserwisów za pomocą RabbitMQ w środowisku opartym na języku C#. Mikroserwisy, jako popularne podejście do projektowania aplikacji, wymagają efektywnego zarządzania komunikacją między nimi. RabbitMQ, będąc zaawansowanym brokerem komunikatów, odgrywa kluczową rolę w ułatwianiu tej wymiany informacji. Dokumentacja zawiera przykłady oraz omówienia kluczowych aspektów konfiguracji, implementacji umożliwiając programistom skuteczną integrację mikroserwisów w projektach opartych na języku C#.

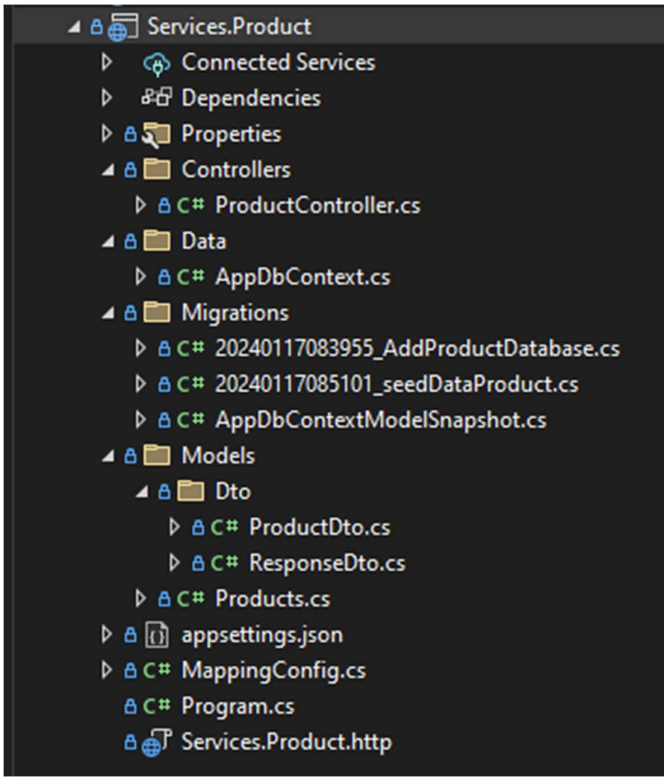
1. Opis mikroserwisów

W ramach projektu zbudowano cztery usługi: Customer, Order, OrderItem i Product. Każda z tych usług została stworzona z myślą o ułatwieniu integracji przy użyciu RabbitMQ, umożliwiając efektywną wymianę informacji między nimi. Usługa Customer odpowiada za obsługę informacji dotyczących klientów, Order zarządza zamówieniami, OrderItem reprezentuje poszczególne pozycje w zamówieniach, natomiast Product odpowiada za dane związane z produktami. Integracja RabbitMQ między tymi usługami pozwala na płynne i efektywne działanie systemu, umożliwiając im współpracę w dynamicznym środowisku mikroserwisów.



2. Jak wygląda poszczególny microservice

Zostanie opisany microservice Product. Cała reszta jest zbudowana w praktycznie ten sam sposób. Jest to szablon projektu ASP.NET Core Web API z wykorzystaniem controllers. Na rysunku 2.1. została pokazana struktura plików. Z najważniejszych możemy wyróżnić Models czyli miejsce przechowywania klas projektów. Znajduje się w nim również katalog DTO, który został dodany do przepływ danych pomiędzy usługami. W katalogu Data znajduję się połączenie z bazą danych (dokładnie z usługą Entity Framework). W katalogu Migrations tworzone są migracje, które odpowiedzialne są za tworzenie struktury bazy danych poszczególnej usługi. W Controllers znajduje się kontroler, który odpowiedzialny jest za tworzenie zapytań CRUD usługi. W pliku MappingConfig.cs została skonfigurowana usługa AutoMapper, która transferuje obiekty DTO do ich poszczególnych klas. W pliku Program.cs zostały powiązane cykle życia poszczególnych komponentów usługi, takie jak np. połączenie z bazą danych, dodanie usługi automappera. Jest również odpowiedzialna za odpalenie całej aplikacji.



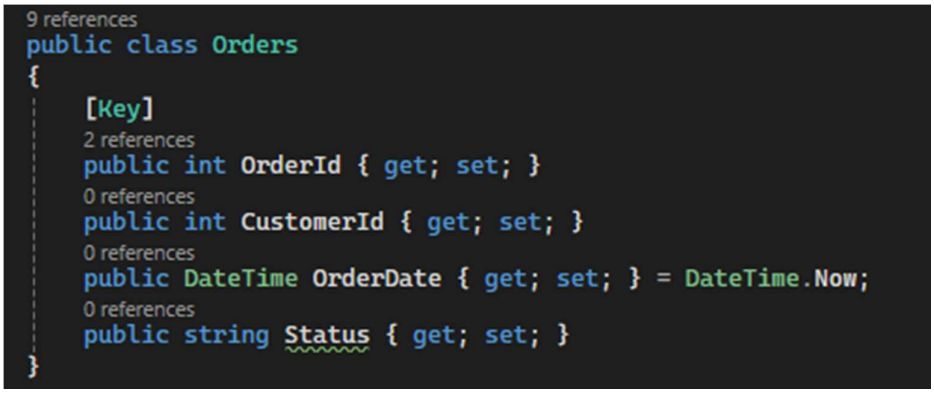
Rys 2.1. Struktura plików usługi Product

3. Modele, reprezentujące bazy danych

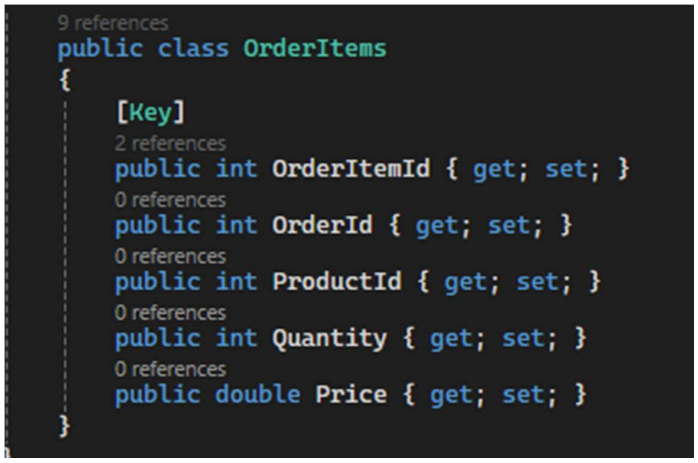
Wyróżniamy tutaj 4 modele Customers, Orders, OrderItems, Products. Są one odpowiedzialne za stworzenie struktur baz danych.

```
public class Customers
{
    [Key]
    2 references
    public int CustomerId { get; set; }
    0 references
    public string FirstName { get; set; }
    0 references
    public string LastName { get; set; }
    [EmailAddress]
    0 references
    public string Email { get; set; }
    0 references
    public string Address { get; set; }
}
```

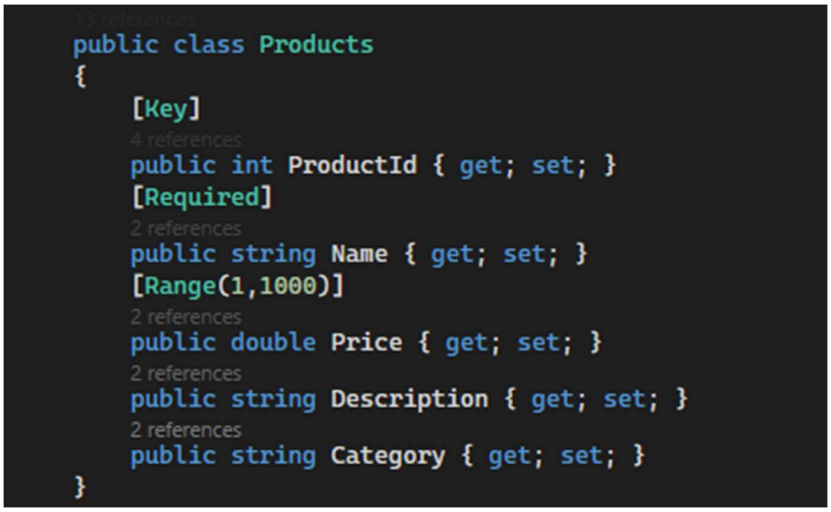
Rys 3.1. Klasa Customers



Rys 3.2. Klasa Orders



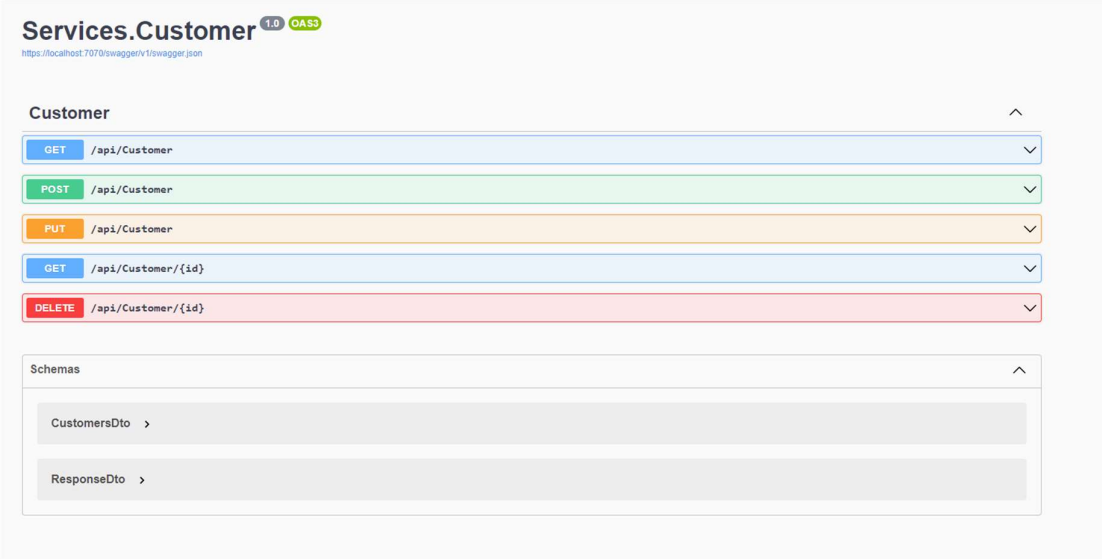
Rys 3.3. Klasa OrderItems



Rys 3.4. Klasa Products

4. Swagger do zapytań CRUD

Do komunikacji w usługę będę wykorzystywał Swagger'a, który pomoże mi do wysyłania zapytań CRUD. Czyli jeśli uruchomimy nasze usługi, będą one wyglądały jak na rysunku 4.1. Posiada on podstawowe zapytania GET, POST, PUT, DELETE.



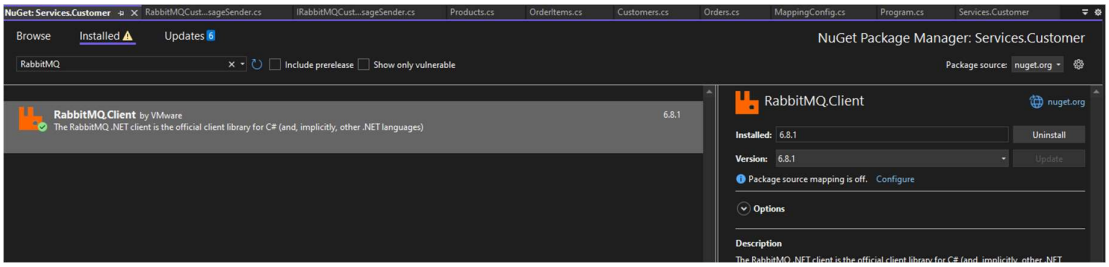
4.1. Swagger w aplikacji

5. RabbitMQ do czego jest potrzebny w aplikacji?

Dodamy do aplikacji system przekazywania komunikatów (message broker). Będzie służył do dodawania wiadomości do kolejki. Kolejka w RabbitMQ to abstrakcyjny mechanizm przechowywania i przesyłania komunikatów między producentem (usługa, która wyślę wiadomość) a konsumentem (usługa, która odbierze wiadomość). Dzięki dodaniu kolejkowania, nie musimy martwić się o wiadomości kiedy jedna z naszych usług się wyłączy. Po ponownym uruchomieniu wszystkie wiadomości dodane do kolejki zostaną odebrane. Jednak jedną z zalet jest to, że możemy wysłać od naszego producenta komunikat do wielu producentów, co właśnie będzie bardzo pomagało w komunikacji między mikrousługami.

6. Dodanie RabbitMQ do projektu

W Manage NuGet Packages musimy zainstalować RabbitMQ.Client tak jak na rysunku 6.1.



6.1. RabbitMQ instalacja w NuGet Packages

Następnie potrzebujemy stworzyć interfejs (Rys 6.2.), który będzie zawierał obiekt oraz nazwę kolejki. Potem w utworzonej klasie zaimplementujemy ten interfejs, żeby móc się zalogować do usługi RabbitMQ oraz utworzny metodę SendMessage, która będzie wysyłać komunikaty do Konsumentów (Rys 6.3.).

```
4 references
public interface IRabbitMQCustomerMessageSender
{
    2 references
    void SendMessage(Object message, string queueName);
}
```

6.2. Stworzenie interfejsu IRabbitMQCustomerMessageSender

```
public class RabbitMQCustomerMessageSender : IRabbitMQCustomerMessageSender
{
    private readonly string _hostName;
    private readonly string _username;
    private readonly string _password;
    private IConnection _connection;

    0 references
    public RabbitMQCustomerMessageSender()
    {
        _hostName = "localhost";
        _username = "guest";
        _password = "guest";
    }
    2 references
    public void SendMessage(object message, string queueName)
    {
        if (ConnectionExists())
        {

            using var channel = _connection.CreateModel();
            channel.QueueDeclare(queueName, false, false, false, null);
            var json = JsonConvert.SerializeObject(message);
            var body = Encoding.UTF8.GetBytes(json);
            channel.BasicPublish(exchange: "", routingKey: queueName, null, body: body);
        }

    }
```

6.3. Stworzenie klasy RabbitMQCustomerMessageSender oraz metody SendMessage

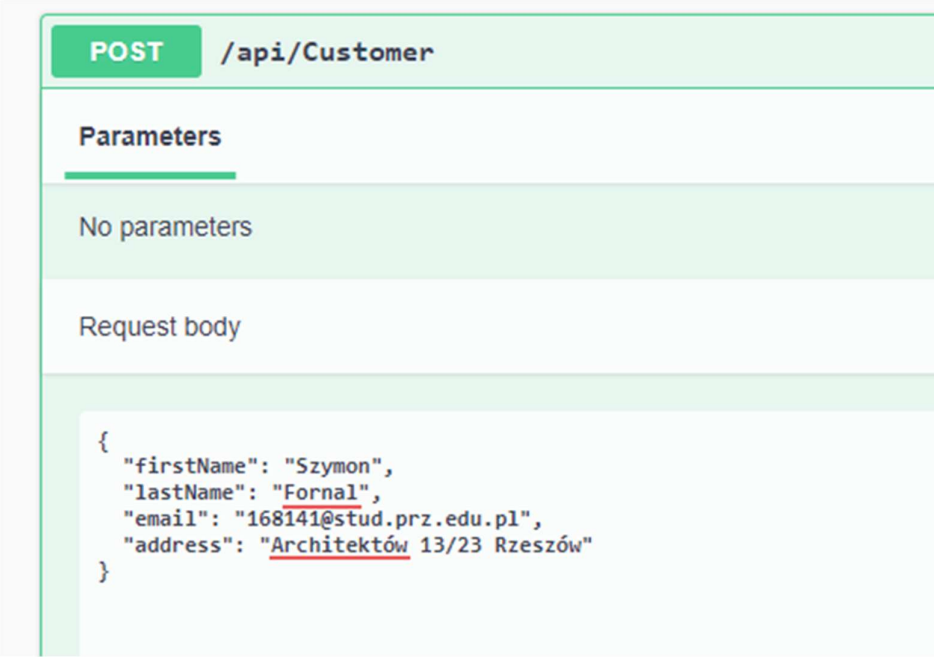
7. Wysyłanie komunikatu

W naszym kontrolerze dodaliśmy do konstruktora wywołanie interfejsu IRabbitMQCustomerSender i zapisaliśmy go jako zmienną _messageSender. Teraz po odwołaniu do tej zmiennej możemy wysłać komunikat jak na rysunku 6.3. z obiektem customersDto oraz podając nazwę kolejki przechowywaną w appsettings.json.

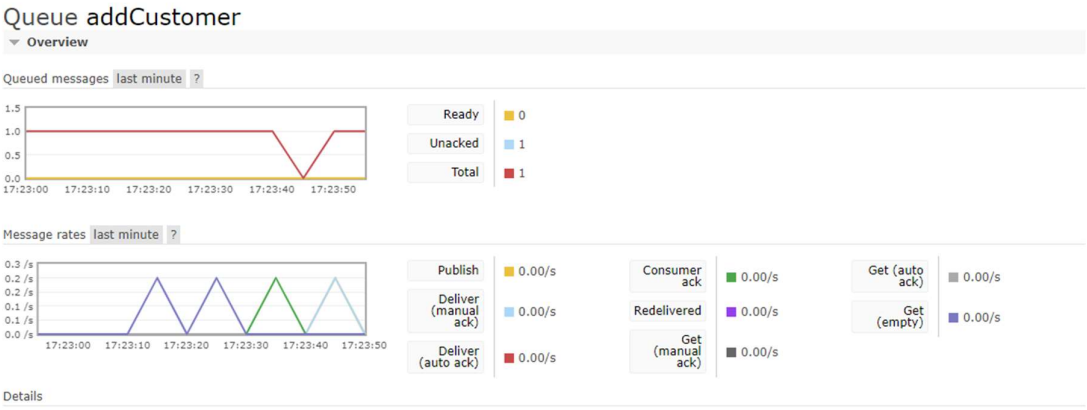
```
[HttpPost]
0 references
public ResponseDto Post(CustomersDto customersDto)
{
    try
    {
        Customers customer = _mapper.Map<Customers>(customersDto);
        _db.Customers.Add(customer);
        _db.SaveChanges();


        _db.Customers.Update(customer);
        _db.SaveChanges();
        _response.Result = _mapper.Map<CustomersDto>(customer);
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    _messageSender.SendMessage(customersDto, _configuration.GetValue<string>("TopicAndQueueNames:AddCustomerQueue"));
    return _response;
}
```

7.1. Kod wysyłania wiadomości

Teraz wystarczy wysłać zapytanie POST i utworzyć nowy obiekt klienta dla naszej aplikacji (Rys 7.2.). Po utworzeniu obiektu i wyłaniu możemy zobaczyć nowy komunikat jak na Rysunku 7.3.



7.2. Stworzenie nowego obiektu klienta



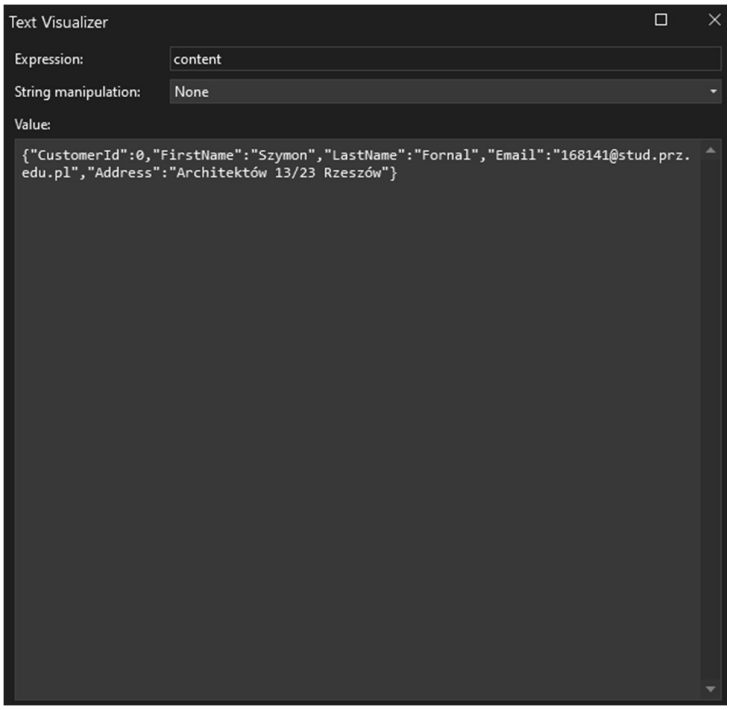
7.3. Otrzymanie nowego komunikatu

8. Odbieranie komunikatu w RabbitMQ

Teraz, żeby odebrać taki komunikat musimy stworzyć klasę RabbitMQCustomerConsumer, która będzie również implementowała klasę BackgroundService. Ta klasa jest bardzo ważna, ponieważ zaimplementowaliśmy w Program.cs cykl życia jako HostedService i będzie ona odpytywać naszą aplikację, z każdym wysłaniem komunikatu. Bardzo ważne jest to, żeby podać wszystkie parametry takie same jak w implementacji wysyłanego komunikatu. Jeśli coś będzie się różniło nie otrzymamy wiadomości. Dzięki nadpisaniu metody ExecuteAsync (Rys 8.1.), możemy odebrać teraz nasze wysłane zapytanie. Jeśli ustawimy debuggera na zmiennej customerDto z rysunku 8.1. możemy otrzymać komunikat, który został wysłany z Rys 7.2. Wygląda on jak na rysunku 8.2. w foramcie json.

```
2 references
public class RabbitMQCustomerConsumer : BackgroundService
{
    private readonly IConfiguration _configuration;
    private IConnection _connection;
    private IModel _channel;
    0 references
    public RabbitMQCustomerConsumer(IConfiguration configuration)
    {
        _configuration = configuration;
        var factory = new ConnectionFactory
        {
            HostName = "localhost",
            Password = "guest",
            UserName = "guest",
        };
        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
        _channel.QueueDeclare(_configuration.GetValue<string>("TopicAndQueueNames:AddCustomerQueue")
            , false, false, false, null);

        var consumer = new EventingBasicConsumer(_channel);
    }

    0 references
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        stoppingToken.ThrowIfCancellationRequested();

        var consumer = new EventingBasicConsumer(_channel);
        consumer.Received += (ch, ea) =>
        {
            var content = Encoding.UTF8.GetString(ea.Body.ToArray());
            CustomersDto customerDto = JsonConvert.DeserializeObject<CustomersDto>(content);
           //HandleMessage(customerDto).GetAwaiter().GetResult();

            _channel.BasicAck(ea.DeliveryTag, false);
        };

        _channel.BasicConsume(_configuration.GetValue<string>("TopicAndQueueNames:AddCustomerQueue"), false, consumer);

        return Task.CompletedTask;
    }
```

8.1. Implementacja klasy RabbitMQCustomerConsumer



8.2. Otrzymanie komunikatu w formacie json

9. Wysyłanie do kolejki fanout

Kolejka typu "fanout" w RabbitMQ jest jednym z rodzajów wymiany (exchange), który umożliwia wysyłanie kopii każdego przesłanego komunikatu do wszystkich powiązanych kolejek. Jest to mechanizm rozgłaszania, gdzie każda kolejka związana z danym exchange otrzymuje identyczną kopię komunikatu.

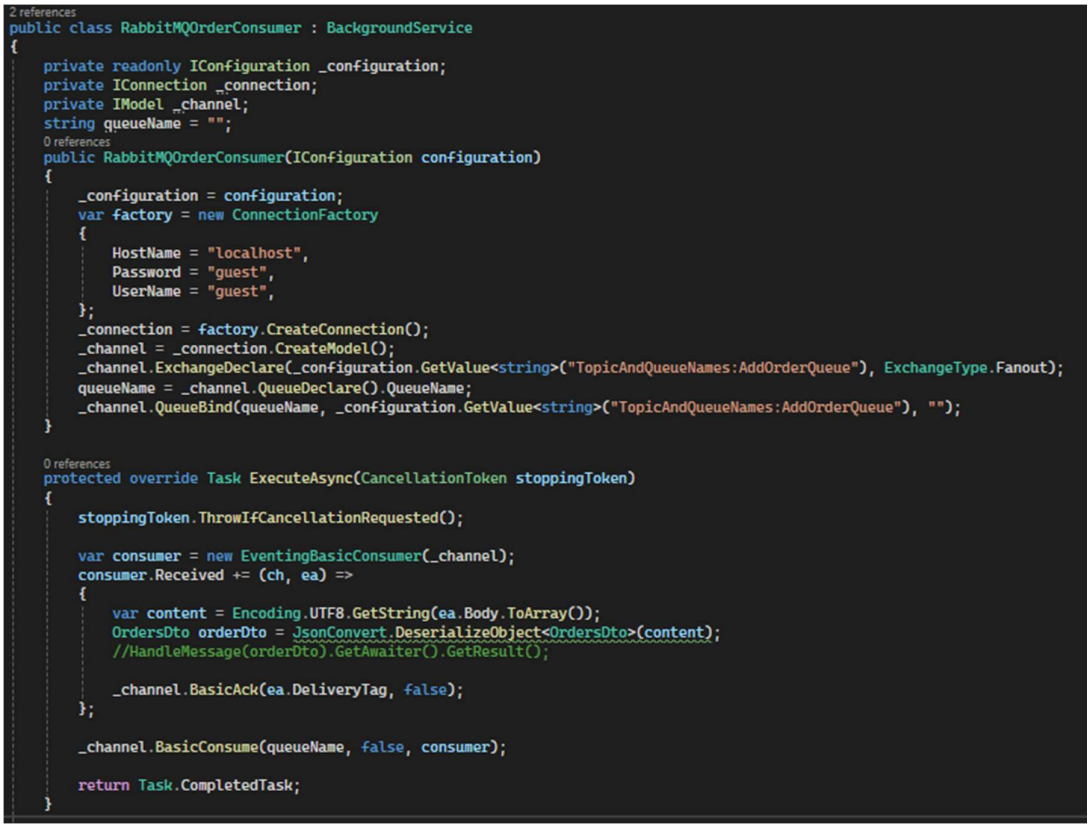
Zaimplementujemy taką kolejkę w usługę Order (Rys 9.1.). Robi się to tam samo jak w poprzednim przypadku (Rys 6.3.), jednak ze zmianą metody na EchangeDeclare. Teraz my nie definiujemy nazwy kolejki tylko definuje je zmienna w RabbitMQ exchangeName. W ten właśnie sposób, będziemy się komunikować ze wszystkimi kolejkami.

```
0 references
public RabbitMQOrderMessageSender()
{
    _hostName = "localhost";
    _username = "guest";
    _password = "guest";
}
2 references
public void SendMessage(object message, string exchangeName)
{
    if (ConnectionExists())
    {

        using var channel = _connection.CreateModel();
        channel.ExchangeDeclare(exchangeName, ExchangeType.Fanout, durable: false);
        var json = JsonConvert.SerializeObject(message);
        var body = Encoding.UTF8.GetBytes(json);
        channel.BasicPublish(exchange: exchangeName, "", null, body: body);
    }

}
```

9.1. Implementacja RabbitMQOrderMessageSender z kolejką fanout

10. Odbieranie wiadomości od kolejki fanout

Podobnie jak wcześniej teraz musimy odebrać taki komunikat więc stworzymy podobnie jak na rysunku 8.1. klasę, która będzie służyła do odbierania takiego komunikatu. Będzie się ona różniła tym, że musi odbierać takie same parametry jak w implemetacji na rysunku 9.1. więc zamienimy QueueDeclare na ExchangeDeclare (Rys 9.1.). Taką klasę mamy stworzoną w usługę Order oraz OrderItem. Po wysłaniu wiadomości i stworzenia nowego zamówienia obydwie usługi dostaną wiadomość. Właśnie w tym stylu komunikują się mikrousługi za pomocą RabbitMQ.



```
2 references
public class RabbitMQOrderConsumer : BackgroundService
{
    private readonly IConfiguration _configuration;
    private IConnection _connection;
    private IModel _channel;
    string queueName = "";
    0 references
    public RabbitMQOrderConsumer(IConfiguration configuration)
    {
        _configuration = configuration;
        var factory = new ConnectionFactory
        {
            HostName = "localhost",
            Password = "guest",
            UserName = "guest",
        };
        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
        _channel.ExchangeDeclare(_configuration.GetValue<string>("TopicAndQueueNames:AddOrderQueue"), ExchangeType.Fanout);
        queueName = _channel.QueueDeclare().QueueName;
        _channel.QueueBind(queueName, _configuration.GetValue<string>("TopicAndQueueNames:AddOrderQueue"), "");
    }

    0 references
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        stoppingToken.ThrowIfCancellationRequested();

        var consumer = new EventingBasicConsumer(_channel);
        consumer.Received += (ch, ea) =>
        {
            var content = Encoding.UTF8.GetString(ea.Body.ToArray());
            OrdersDto orderDto = JsonConvert.DeserializeObject<OrdersDto>(content);
            //HandleMessage(orderDto).GetAwaiter().GetResult();

            _channel.BasicAck(ea.DeliveryTag, false);
        };

        _channel.BasicConsume(queueName, false, consumer);

        return Task.CompletedTask;
    }
```

10.1. Implementacja RabbitMQOrderConsumer do odbierania kolejki fanout

Podsumowanie

Kolejka typu "fanout" w RabbitMQ stanowi potężne narzędzie w kontekście systemów opartych na mikroserwisach, umożliwiając rozgłaszanie komunikatów do wielu kolejek bez konieczności definiowania reguł routingu. To szczególnie przydatne w przypadku, gdy wiele komponentów systemu, reprezentujących różne mikroserwisy, potrzebuje odbierać identyczne informacje jednocześnie.

W kontekście mikroserwisów, stosowanie kolejek, takich jak "fanout", wprowadza asynchroniczność do komunikacji między usługami. Mikroserwisy, reprezentujące różne funkcje systemu, mogą bezproblemowo komunikować się poprzez przesyłanie komunikatów do kolejek. To podejście umożliwia elastyczne skalowanie poszczególnych usług, ponieważ komunikacja między nimi nie jest bezpośrednia.

Wysyłanie wiadomości przez kolejki w usługach mikroserwisów eliminuje bezpośrednią zależność pomiędzy usługami, co sprzyja luźnemu powiązaniu. Każda usługa może działać niezależnie, a zmiany w jednej usłudze nie wymagają natychmiastowej aktualizacji pozostałych. Kolejki umożliwiają obsługę różnych prędkości przetwarzania i dostarczają mechanizm odporności na awarie.

Podsumowując, wykorzystanie kolejki typu "fanout" oraz wysyłanie wiadomości przez kolejki w architekturze mikroserwisów przyczynia się do elastyczności, skalowalności i odporności na awarie w rozproszonych systemach, tworząc luźno powiązane i wysoko interoperacyjne środowisko usług.