Opracowanie: Kamil Mrowiec

Wprowadzenie

RabbitMQ jest szeroko rozpowszechnionym open source'owym brokerem wiadomości.

Wykorzystywany jest w ponad 35 000 projektach wdrożeniowych zarówno w startup'ach jak i w masywnych przedsiębiorstwach o ugruntowanej pozycji na rynku.

Jest lekki i łatwy we wdrożeniu zarówno stacjonarnie jak i w rozwiązaniach chmurowych.

Wspiera wiele protokołów komunikacyjnych, systemów operacyjnych oraz rozwiązań chmurowych. Zawiera szerokie spektrum narzędzi deweloperskich do najpopularniejszych języków.

Przedsiębiorstwa wykorzystujące RabbitMQ

reddit

Code School

SendGrid

9GAG

GoSquared

Shutterstock

CircleCI

Sentry

Sauce Labs

Moduły w .NET

- **RabbitMQ .NET Client** (wspiera .NET Core and .NET 4.5.1+)
- **EasyNetQ**, łatwe w użyciu .NET API dla RabbitMQ
- **NServiceBus**, najpopularniejsza magistrala serwisowa typu open-souce dla .NET.
- **RawRabbit**, klient wyższego poziomu, dla ASP.NET vNext i .NET Core.
- **Restbus**, zorientowany na serwisy framework dla .NET
- **RabbitMQTools**, moduł Powershella posiadający cmdlets do zarządzania RabbitMQ

Czym jest message broker ?

Message broker akceptuje oraz przekierowuje wiadomości.

Można go porównać do tradycyjnego systemu pocztowego.

W rzeczywistym świecie po włożeniu listu do skrzynki pocztowej możemy być pewni, że listonosz dostarczy wiadomość do określonego odbiorcy.

RabbitMQ jest zarówno skrzynką pocztową, listonoszem jak i samą instytucją pocztową.

1

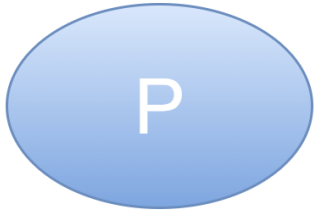
Podstawowe elementy

Czyli jakie komponenty są niezbędne do implementacji systemu komunikacji

Producent (Producer)

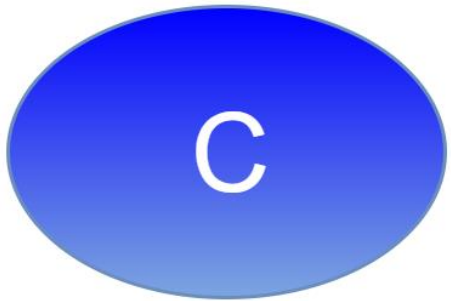
Producentem jest program, który wysyła wiadomości.

W odróżnieniu od tradycyjnej poczty zamiast papierem operujemy binarnymi blokami danych.



Konsument (Consumer)

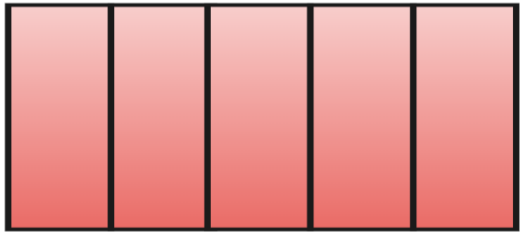
Konsumentem jest program, który oczekuje na odebranie wiadomości.



Kolejka wiadomości (Queue)

Kolejka to kontener („skrzynka na listy”) w postaci bufora, która zawiera wiadomości przesyłane pomiędzy producentami a konsumentami.

Jest zdefiniowana wewnątrz RabbitMQ.



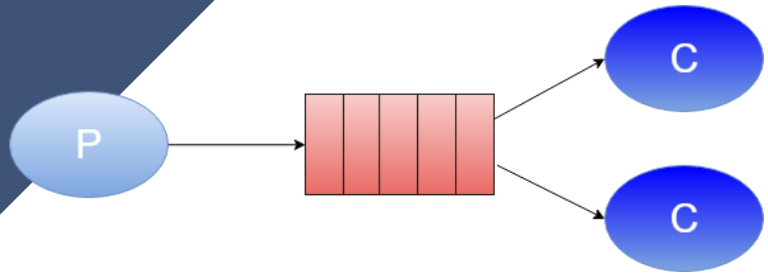
Uzupełnienie

- Wielu producentów może wysyłać wiadomości do jednej kolejki, a także wielu konsumentów może próbować odebrać wiadomości znajdujące się w niej.
- Producent oraz konsument w większości przypadków nie są umieszczeni na tym samym hoście (maszynie) jednakże aplikacja może pracować jednocześnie zarówno jako producent i konsument.

2

Podstawowe mechanizmy

Czyli jak utworzyć prosty system komunikacyjny przesyłający „Hello World!”



Utworzenie Producenta

- Dodanie referencji do **RabbitMQ.Client**.
- Wyszczególnienie adresu IP połączenia (**localhost**).
- Utworzenie **połączenia** oraz **kanału komunikacyjnego**.

```
using RabbitMQ.Client;

0 references
class Send
{
    0 references
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
```

- Utworzenie nazwanej kolejki – „hello”
- Utworzenie wiadomości wraz z konwersją do **tablicy bajtów**.
- Wysłanie wiadomości z domyślnym **exchange**.

```
    {
        channel.QueueDeclare(queue: "hello",
                             durable: false,
                             exclusive: false,
                             autoDelete: false,
                             arguments: null);

        string message = "Hello World!";
        var body = Encoding.UTF8.GetBytes(message);

        channel.BasicPublish(exchange: "",
                             routingKey: "hello",
                             basicProperties: null,
                             body: body);
        Console.WriteLine(" [x] Sent {0}", message);
    }

    Console.WriteLine(" Press [enter] to exit.");
    Console.ReadLine();
}
```

Utworzenie Konsumenta

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

namespace RabbitMQ_Receive_1
{
    0 references
    class Receive
    {
        0 references
        static void Main(string[] args)
        {
            var factory = new ConnectionFactory() { HostName = "localhost"};
            using (var connection = factory.CreateConnection())
            using (var channel = connection.CreateModel())
            {
                channel.QueueDeclare(queue: "hello",
                                     durable: false,
                                     exclusive: false,
                                     autoDelete: false,
                                     arguments: null);
```

- Połączenie z **message brokerem**.
- Utworzenie kolejki (zabezpieczenie przed wywołaniem konsumenta wcześniej niż producenta).

- Utworzenie konsumenta, który **nasłuchuje na zdarzenia.**
- Zadeklarowanie pętli nasłuchującej. W jej ciele przetwarzane są wysyłane przez producentów wiadomości.
- Połączenie **kanałem** konsumenta z nazwaną kolejką „hello”.

```
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body;
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine(" [x] Received {0}", message);
};
channel.BasicConsume(queue: "hello",
                     autoAck: true,
                     consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");
Console.ReadLine();
```

Wyniki wywołania programu

```
Desktop\RabbitMQ\Send_1>dotnet run
[x] Sent Hello World!
Press [enter] to exit.

Desktop\RabbitMQ\Send_1>dotnet run
[x] Sent Hello World!
Press [enter] to exit.
```

Producent

```
Desktop\RabbitMQ\Receive_1>dotnet run
Press [enter] to exit.
[x] Received Hello World!
[x] Received Hello World!
```

Konsument

3

Cechy wiadomości

Czyli trwałość i sposoby przydzielania wiadomości do konsumentów.

Metoda przydzielania Round-Robin

Metoda przydzielania Round-Robin przypisuje po kolei wiadomości znajdujące się w kolejce kolejnym konsumentom znajdującym się w pewnej sekwencji o ile z daną kolejką został skojarzony więcej niż 1 konsument.

Potwierdzenie dostarczenia wiadomości

Potwierdzenia dostarczenia wiadomości pozwalają zapewnić, że **wiadomość nie została utracona**.

Po otrzymaniu wiadomości, od konsumenta **wysyłane jest powrotne potwierdzenie**, aby poinformować brokera o fakcie poprawnego dostarczenia wiadomości. Po tym zdarzeniu wiadomość może zostać usunięta.

Trwałość wiadomości

W przypadku gdy broker RabbitMQ z bliżej niewyjaśnionych przyczyn przestanie działać, wiadomości oraz kolejki są tracone bezpowrotnie.

Takiemu obrotowi spraw można zapobiec stosując **flagi durable**.

Aby zapewnić żywotność tych elementów należy oznaczyć zarówno **kolejkę** jak i same **wiadomości** jako właśnie **durable (trwałe)**.

Fair Dispatch

Fair Dispatch pozwala na zmianę domyślnej metody przydzielania wiadomości do danego konsumenta. Dzięki wykorzystaniu metody BasicQos można sterować ilością przydzielonych jednocześnie wiadomości danemu klientowi.

Właściwość **prefetchCount** znajdująca się w wyżej wymienionej metodzie pozwala na **przydzielenie określonej liczby wiadomości**. Dopiero w momencie przetworzenia danej wiadomości oraz wysłania potwierdzenia, nowa wiadomość może zostać przydzielona do danego konsumenta.

```
channel.BasicQos(0, 1, false);
```

4

Pełny system komunikacyjny

Czyli czym jest exchange oraz jak działają metody routingu.

Exchange

Jednym z założeń mechanizmu komunikacyjnego w RabbitMQ jest **nie przesyłanie bezpośrednio wiadomości do kolejki przez producenta**.

Producent może wysłać wyłącznie wiadomość do exchange.

Producent sam w sobie **nie ma informacji do jakiej kolejki** (czy do żadnej/czy do wielu) **wysyłana jest dana wiadomość.**

Exchange jest mechanizmem pośredniczącym w transferze informacji. Odbiera wiadomości wysyłane przez producentów i kieruje je do wskazanych kolejek na podstawie danych zawartych w publikacji.

E

Temporary Queues

W większości zastosowań zamiast wykorzystanej poprzednio kolejki nazwanej stosuje się **kolejki tymczasowe**.

Kolejki tymczasowe sprawdzają się, gdy zawsze po połączeniu z Rabbit potrzebujemy **świeżej, pustej kolejki** natomiast po rozłączeniu konsumenta **kolejka ma zostać usunięta**. Możemy tworzyć **nietrwałą, automatycznie kasowaną kolejkę** z generowaną nazwą przy pomocy:

```
var queueName = channel.QueueDeclare().QueueName;
```

Binding

Po utworzeniu exchange oraz kolejki należy przekazać informację do exchange, aby komponent ten przekierowywał wiadomości do danej kolejki.

Binding jest to relacja pomiędzy exchange a kolejką określająca powiązanie kolejki z danym exchange.

Exchange *„logs”* będzie przekazywał wiadomości do kolejki o nazwie *„queueName”*.

```
channel.QueueBind(queue: queueName,
                  exchange: "logs",
                  routingKey: "");
```

Wybrane rodzaje Exchange

Exchange może działać na kilka sposobów:

- Direct
- Topic
- Headers
- Fanout
- Default

Deklaracja nowego exchange o nazwie *„logs”* i typie *„fanout”*

```
channel.ExchangeDeclare("logs", "fanout");
```

Default Exchange

Default exchange zawsze kieruje wiadomości do kolejki o nazwie wprowadzonej do pola *routingKey*.

Oznaczany jest przy publikacji jako "" (pusty string).

```
channel.BasicPublish(exchange: "",
                     routingKey: "hello",
                     basicProperties: null,
                     body: body);
```

Fanout Exchange

Fanout Exchange kieruje wiadomości do wszystkich kolejek o istnieniu których ma informacje.

Ten typ exchange ignoruje pole *routingKey* znajdujące się w bindingu kolejki.

```
channel.ExchangeDeclare("logs", "fanout");
```

Direct Exchange

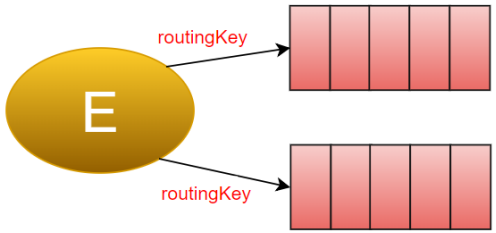
Direct Exchange kieruje wiadomości do kolejek, których *routingKey bindingu* jest identyczny z *routingKey* przesyłanej wiadomości.

Ten typ exchange działa na zasadzie sortowania, gdzie **dla każdej kolejki przesyłane są wiadomości adresowane konkretnie do niej**.

Multiple Binding

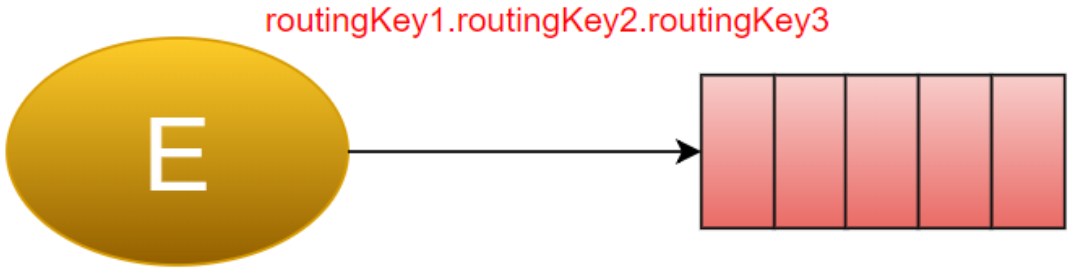
Multiple Binding polega na przypisaniu takiego samego *routingKey bindingu* dwóm lub większej ilości kolejek.

W przypadku wykorzystania exchange typu **direct**, ale z wykorzystaniem **multiple binding** otrzymamy **exchange**, który przekierowuje wiadomości jednocześnie do dwóch lub większej ilości kolejek **o identycznym warunku**.



Topic Exchange

Topic Exchange jest rozszerzeniem Direct Exchange w którym to *binding routingKey* nie jest jednym słowem lecz listą słów pozwalających na zdefiniowanie nie 1 lecz wielu kryteriów przekierowania wiadomości.



Określanie kryterium dla tego typu **exchange** wygląda następująco:

„car.orange.1”, gdzie „.” oddziela poszczególne kryteria.

Dodatkowo można wykorzystać znaki specjalne:

- *** - zastępuje dokładnie 1 słowo**
- **# - zastępuje 0 lub większą dowolną ilość słów**

Limitem ilości kryteriów jest rozmiar 255 bajtów.

5

Remote Procedure Call

Czyli jak za pomocą wiadomości uruchamiać czasochłonne operacje wymagające informacji zwrotnej

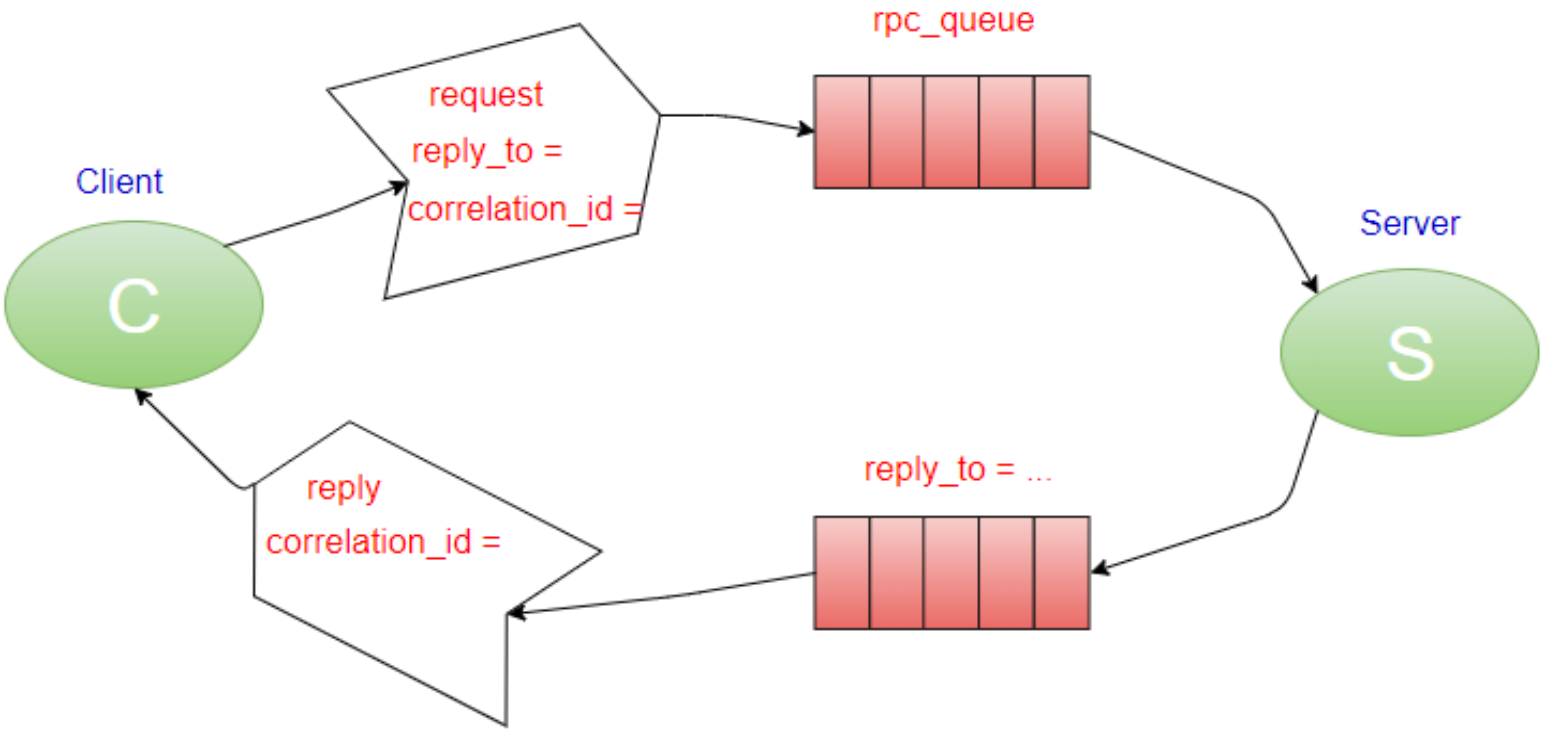
RPC

RPC (Remote Procedure Call) jest to zdalne wywołanie procedury. W przypadku brokera wiadomości RabbitMQ wykorzystywane jest do zadysponowania poprzez wiadomość zadania do wykonania przez odbiorcę, a następnie po określonym czasie odebrania z kolejki zwrotnej uzyskanego wyniku.

Założenia działania RPC

- Każdy klient tworzy po swojej inicjalizacji **wyłączną kolejkę zwrotną**.
- Każdy **request**, który klient wysyła do serwera identyfikowany jest przez 2 właściwości: ***ReplyTo*** oraz ***CorrelationId***.
- **Requesty wysyłane są do kolejki** jednostki wykonawczej RPC (servera).

- Kiedy server **odczyta wiadomość**, przystąpi do **wykonywania zdefiniowanego w requeście zadania**.
- Po zakończeniu pracy **server wysyła** na podstawie parametru *ReplyTo* **wyniki do kolejki zwrotnej** konkretnego klienta.
- **Klient** sprawdza swoją kolejkę i **dopasowuje otrzymane wiadomości** do konkretnego **requesta** na podstawie ***Correlation Id***.


rpc_queue
request
reply_to =
correlation_id =
Client
C
Server
S
reply_to = ...
reply
correlation_id =

Struktura request'u

Dodatkowy parametr ***ReplyTo*** **służy do identyfikacji kolejki** do której jednostka wykonawcza RPC ma wysłać rezultaty wykonania przydzielonego zadania.

```
var props = channel.CreateBasicProperties();
props.ReplyTo = replyQueueName;

var messageBytes = Encoding.UTF8.GetBytes(message);
channel.BasicPublish(exchange: "",
                     routingKey: "rpc_queue",
                     basicProperties: props,
                     body: messageBytes);
```

Correlation Id

Klient, który zlecał wykonanie zadania **sprawdza cyklicznie kolejkę zwrotną**. Na podstawie parametru ***Correlation Id*** jest w stanie powiązać **wysłany** przez siebie **request** z konkretną **wiadomością znajdującą się w kolejce zwrotnej**.

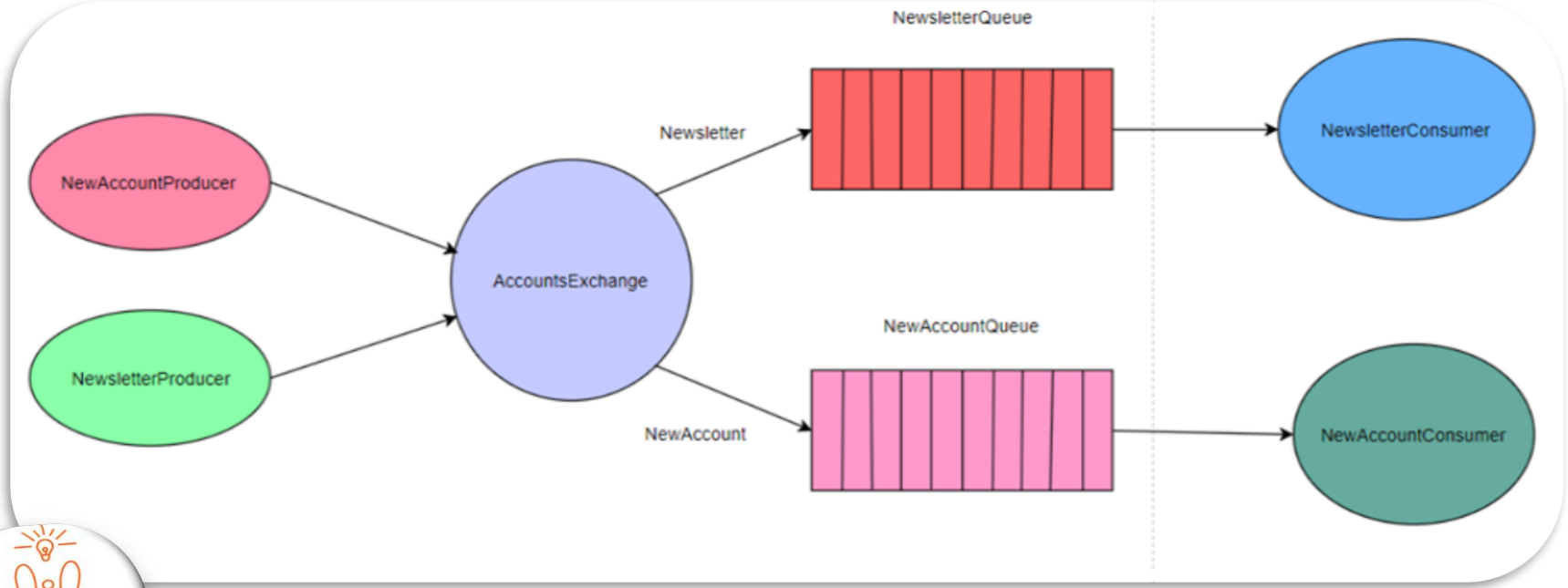
```
var props = channel.CreateBasicProperties();
var correlationId = Guid.NewGuid().ToString();
props.CorrelationId = correlationId;
props.ReplyTo = replyQueueName;
```

```
consumer.Received += (model, ea) =>
{
    var body = ea.Body;
    var response = Encoding.UTF8.GetString(body);
    if (ea.BasicProperties.CorrelationId == correlationId)
    {
        respQueue.Add(response);
    }
};
```

6

Podstawowy system obsługi kont w sklepie internetowym z wykorzystaniem **RabbitMQ**

Schemat systemu



PREZENTACJA DZIAŁANIA PROGRAMU

Opracowanie: Kamil Mrowiec

https://www.rabbitmq.com/