RabbitMQ 지침서

2020-12-03

목차

[RabbitMQ 지침서 1](#_Toc57886502)

["Hello World!" 4](#_Toc57886503)

[무언가를 하는 가장 간단한 것 4](#_Toc57886504)

[Introduction 4](#_Toc57886505)

["Hello World" 5](#_Toc57886506)

[**(using the .NET/C# Client)** 5](#_Toc57886507)

[Setup 6](#_Toc57886508)

[Sending 7](#_Toc57886509)

[Receiving 10](#_Toc57886510)

[Putting It All Together 11](#_Toc57886511)

[Production [Non-]Suitability Disclaimer 12](#_Toc57886512)

[Getting Help and Providing Feedback 12](#_Toc57886513)

[Help Us Improve the Docs <3 12](#_Toc57886514)

[Work queues 13](#_Toc57886515)

[Work Queues 13](#_Toc57886516)

[(using the .NET Client) 13](#_Toc57886517)

[Preparation 13](#_Toc57886518)

[Round-robin dispatching 15](#_Toc57886519)

[Message acknowledgment 16](#_Toc57886520)

[Message durability 18](#_Toc57886521)

[Putting It All Together 19](#_Toc57886522)

[Publish/Subscribe 21](#_Toc57886523)

[Publish/Subscribe 21](#_Toc57886524)

[(using the .NET Client) 21](#_Toc57886525)

[Exchanges 21](#_Toc57886526)

[Temporary queues 23](#_Toc57886527)

[Bindings 24](#_Toc57886528)

[Putting it all together 25](#_Toc57886529)

[Routing 28](#_Toc57886530)

[Routing 28](#_Toc57886531)

[(using the .NET client) 28](#_Toc57886532)

[Bindings 28](#_Toc57886533)

[Direct exchange 29](#_Toc57886534)

[Multiple bindings 30](#_Toc57886535)

[Emitting logs 30](#_Toc57886536)

[Subscribing 30](#_Toc57886537)

[Putting it all together 31](#_Toc57886538)

[Topics 34](#_Toc57886539)

[Topics 34](#_Toc57886540)

[(using the .NET client) 34](#_Toc57886541)

[Topic exchange 34](#_Toc57886542)

[Putting it all together 36](#_Toc57886543)

[RPC 39](#_Toc57886544)

[Remote procedure call (RPC) 39](#_Toc57886545)

[(using the .NET client) 39](#_Toc57886546)

[Client interface 39](#_Toc57886547)

[Callback queue 41](#_Toc57886548)

[Correlation Id 42](#_Toc57886549)

[Summary 42](#_Toc57886550)

[Putting it all together 43](#_Toc57886551)

[Publisher Confirms 48](#_Toc57886552)

[Publisher Confirms 48](#_Toc57886553)

[(using the .NET client) 48](#_Toc57886554)

[Overview 48](#_Toc57886555)

[Enabling Publisher Confirms on a Channel 48](#_Toc57886556)

[Strategy #1: Publishing Messages Individually 48](#_Toc57886557)

[Strategy #2: Publishing Messages in Batches 49](#_Toc57886558)

[Strategy #3: Handling Publisher Confirms Asynchronously 50](#_Toc57886559)

[Summary 52](#_Toc57886560)

[Putting It All Together 53](#_Toc57886561)

## ["Hello World!"](https://www.rabbitmq.com/tutorials/tutorial-one-python.html)

## https://www.rabbitmq.com/img/tutorials/python-one.png무언가를 하는 가장 간단한 것

## Introduction

RabbitMQ는 메시지 브로커입니다. 메시지를 수락하고 전달합니다. 우체국이라고 생각할 수 있습니다. 게시하려는 메일을 우체통에 넣으면 Mailperson 씨 또는 Ms. Mailperson이 결국받는 사람에게 메일을 배달 할 것임을 확신 할 수 있습니다. 이 비유에서 RabbitMQ는 우체통, 우체국 및 우체부입니다.

**전제 조건**

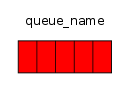
이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

RabbitMQ와 우체국의 주요 차이점은 종이를 처리하지 않고 대신 데이터의 이진 blob (메시지)을 수락, 저장 및 전달한다는 것입니다.

RabbitMQ 및 일반적으로 메시징은 일부 전문 용어를 사용합니다

* https://www.rabbitmq.com/img/tutorials/producer.png생산은 보내는 것 이상을 의미하지 않습니다. 메시지를 보내는 프로그램은 생산자입니다.
* 큐는 RabbitMQ 내부에있는 우체통의 이름입니다. 메시지는 RabbitMQ 및 애플리케이션을 통해 흐르지 만 큐 내부에만 저장할 수 있습니다. 대기열은 호스트의 메모리 및 디스크 제한에 의해서만 제한되며 본질적으로 큰 메시지 버퍼입니다. 많은 생산자가 하나의 대기열로가는 메시지를 보낼 수 있고 많은 소비자가 하나의 대기열에서 데이터를 수신하려고 할 수 있습니다. 이것은 우리가 큐를 나타내는 방법입니다.
* https://www.rabbitmq.com/img/tutorials/consumer.png소비는 받는 것과 비슷한 의미를 갖습니다. 소비자는 대부분 메시지 수신을 기다리는 프로그램입니다.

생산자, 소비자 및 브로커가 동일한 호스트에 있을 필요는 없습니다. 실제로 대부분의 응용 프로그램에서는 그렇지 않습니다. 응용 프로그램은 생산자이자 소비자 일 수도 있습니다.

"Hello World"

**(using the .NET/C# Client)**

튜토리얼의 이 부분에서는 C #으로 두 개의 프로그램을 작성합니다. 단일 메시지를 보내는 생산자와 메시지를 받아 출력하는 소비자. .NET 클라이언트 API의 세부 사항 중 일부를 간단히 살펴보고 시작하기 위해 이 매우 간단한 것에 집중할 것입니다. 메시징의 "Hello World"입니다.

(P) -> [|||] -> (C)아래 다이어그램에서 "P"는 생산자이고 "C"는 소비자입니다. 중간에 있는 상자는 RabbitMQ가 소비자를 대신하여 보관하는 메시지 버퍼 인 대기열입니다

#### The .NET client library

RabbitMQ는 여러 프로토콜을 사용합니다. 이 자습서에서는 메시징을위한 개방형 범용 프로토콜 인 AMQP 0-9-1을 사용합니다. 다양한 언어로 된 RabbitMQ 용 클라이언트가 많이 있습니다. RabbitMQ에서 제공하는 .NET 클라이언트를 사용합니다.

클라이언트는 .NET Core 및 .NET Framework 4.5.1+를 지원합니다. 이 자습서에서는 RabbitMQ .NET 클라이언트 5.0 및 .NET Core를 사용하므로 설치되어 있고 PATH에 있는지 확인합니다.

.NET Framework를 사용하여 이 자습서를 완료 할 수도 있지만 설정 단계는 다릅니다.

RabbitMQ .NET 클라이언트 5.0 이상 버전은 NUGET을 통해 배포됩니다.

이 자습서에서는 Windows에서 powershell을 사용하고 있다고 가정합니다. MacOS 및 Linux에서는 거의 모든 쉘이 작동합니다.

### Setup

먼저 PATH에 .NET Core 도구 체인이 있는지 확인합니다.

dotnet --help

도움말 메시지를 생성해야 합니다.

이제 게시자 용과 소비자 용으로 하나씩 두 개의 프로젝트를 생성 해 보겠습니다.

dotnet new console --name Send

mv Send/Program.cs Send/Send.cs

dotnet new console --name Receive

mv Receive/Program.cs Receive/Receive.cs

그러면 Send 및 Receive라는 두 개의 새 디렉토리가 생성됩니다.

그런 다음 클라이언트 종속성을 추가합니다.

cd Send

dotnet add package RabbitMQ.Client

dotnet restore

cd ../Receive

dotnet add package RabbitMQ.Client

dotnet restore

이제 .NET 프로젝트가 설정되어 코드를 작성할 수 있습니다.

### (P) -> [|||]Sending

메시지 게시자 (발신자) Send.cs 및 메시지 소비자 (수신자) Receive.cs라고합니다. 게시자는 RabbitMQ에 연결하고 단일 메시지를 보낸 다음 종료합니다.

Send.cs에서는 몇 가지 네임 스페이스를 사용해야합니다.

using System;

using RabbitMQ.Client;

using System.Text;

Set up the class

class Send

{

public static void Main()

{

...

}

}

then we can create a connection to the server:

연결은 소켓 연결을 추상화하고 프로토콜 버전 협상 및 인증 등을 처리합니다. 여기에서 로컬 머신의 RabbitMQ 노드에 연결합니다. 따라서 로컬 호스트가 됩니다. 다른 시스템의 노드에 연결하려면 여기에 호스트 이름이나 IP 주소를 지정하기 만하면 됩니다.

class Send

{

public static void Main()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using (var connection = factory.CreateConnection())

{

using (var channel = connection.CreateModel())

{

...

}

}

}

}

다음으로 작업을 수행하기위한 대부분의 API가있는 채널을 만듭니다.

보내려면 보낼 대기열을 선언해야 합니다. 그러면 큐에 메시지를 게시 할 수 있습니다.

큐 선언은 idempotent 적입니다. 큐가 아직 존재하지 않는 경우에만 생성됩니다. 메시지 내용은 바이트 배열이므로 원하는 대로 인코딩 할 수 있습니다.

using System;

using RabbitMQ.Client;

using System.Text;

class Send

{

public static void Main()

{

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);

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();

}

}

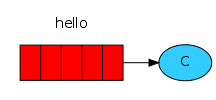
위의 코드 실행이 완료되면 채널과 연결이 삭제됩니다. 게시자를 위한 것입니다.

[Here's the whole Send.cs class](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/Send/Send.cs).

#### Sending doesn't work!

RabbitMQ를 처음 사용하는 경우 "Sent"메시지가 표시되지 않으면 무엇이 잘못되었는지 궁금해 할 수 있습니다. 브로커가 충분한 여유 디스크 공간없이 시작되어 (기본적으로 최소 50MB의 여유 공간이 필요함) 메시지 수락을 거부 할 수 있습니다. 브로커 로그 파일을 확인하여 필요한 경우 제한을 확인하고 줄입니다. 구성 파일 설명서는 disk\_free\_limit를 설정하는 방법을 보여줍니다.

### Receiving

소비자는 RabbitMQ의 메시지를 수신하고 있습니다. 따라서 단일 메시지를 게시하는 게시자와는 달리 소비자가 계속해서 메시지를 듣고 인쇄하도록 할 것입니다.

코드 (Receive.cs)에는 Send와 거의 동일한 using 문이 있습니다.

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System;

using System.Text;

설정은 게시자와 동일합니다. 연결과 채널을 열고 소비 할 큐를 선언합니다. 이것은 Send가 게시하는 대기열과 일치합니다.

class Receive

{

public static void Main()

{

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);

...

}

}

}

}

여기에서도 큐를 선언합니다. 게시자보다 먼저 소비자를 시작할 수 있으므로 메시지를 사용하기 전에 큐가 있는지 확인하려고 합니다.

대기열에서 메시지를 전달하도록 서버에 지시하려고 합니다. 메시지를 비동기 적으로 푸시하므로 콜백을 제공합니다. 이것이 EventingBasicConsumer.Received 이벤트 핸들러가 하는 일입니다.

[Here's the whole Receive.cs class](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/Receive/Receive.cs).

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System;

using System.Text;

class Receive

{

public static void Main()

{

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);

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

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();

}

}

}

### Putting It All Together

두 개의 터미널을 엽니다.

먼저 소비자를 실행하여 토폴로지 (주로 큐)가 제자리에 있도록 합니다.

cd Receive

dotnet run

그런 다음 생산자를 실행합니다.

cd Send

dotnet run

소비자는 RabbitMQ를 통해 게시자로부터 받은 메시지를 인쇄합니다. 소비자는 계속 실행되어 메시지를 기다리므로 (Ctrl-C를 사용하여 중지) 다른 터미널에서 게시자를 실행 해보십시오.

2 부로 넘어 가서 간단한 작업 대기열을 만들 시간입니다.

## Production [Non-]Suitability Disclaimer

이 튜토리얼과 다른 튜토리얼은 튜토리얼이라는 것을 명심하십시오. 그들은 한 번에 하나의 새로운 개념을 보여주고 일부는 의도적으로 지나치게 단순화하고 다른 것은 제외시킬 수 있습니다. 예를 들어 연결 관리, 오류 처리, 연결 복구, 동시성 및 메트릭 수집과 같은 항목은 간결성을 위해 대부분 생략되었습니다. 이러한 단순화 된 코드는 프로덕션 준비 상태로 간주해서는 안됩니다.

앱을 사용하기 전에 나머지 [documentation](https://www.rabbitmq.com/documentation.html)를 살펴 보시기 바랍니다. 특히 [Publisher Confirms and Consumer Acknowledgements](https://www.rabbitmq.com/confirms.html), [Production Checklist](https://www.rabbitmq.com/production-checklist.html) 및 [Monitoring](https://www.rabbitmq.com/monitoring.html) 가이드를 권장합니다.

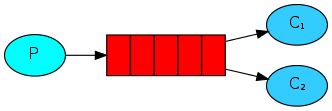
## Getting Help and Providing Feedback

이 튜토리얼의 내용이나 RabbitMQ와 관련된 다른 주제에 대한 질문이 있으면 주저하지 말고 [RabbitMQ mailing list](https://groups.google.com/forum/#!forum/rabbitmq-users)에 질문하십시오.

## Help Us Improve the Docs <3

사이트 개선에 기여하고 싶다면 [available on GitHub](https://github.com/rabbitmq/rabbitmq-website)에서 해당 소스를 사용할 수 있습니다. 저장소를 포크하고 pull 요청을 제출하기 만하면 됩니다. 감사합니다!

## [Work queues](https://www.rabbitmq.com/tutorials/tutorial-two-python.html)

작업자에게 작업 분배(the [competing consumers pattern](http://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html))

## Work Queues

### https://www.rabbitmq.com/img/tutorials/python-two.png(using the .NET Client)

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

작업자 간에 작업 배포 첫 번째 자습서([first tutorial](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html))에서는 명명 된 큐에서 메시지를 보내고 받는 프로그램을 작성했습니다. 여기에서는 여러 작업자에게 시간이 많이 걸리는 작업을 분산하는 데 사용할 작업 대기열을 만듭니다.

작업 대기열 (일명 작업 대기열)의 기본 아이디어는 리소스 집약적 인 작업을 즉시 수행하지 않고 완료 될 때까지 기다려야하는 것입니다. 대신 나중에 수행 할 작업을 예약합니다. 작업을 메시지로 캡슐화하여 대기열로 보냅니다. 백그라운드에서 실행되는 작업자 프로세스는 작업을 표시하고 결국 작업을 실행합니다. 많은 작업자를 실행하면 작업이 그들간에 공유됩니다.

이 개념은 짧은 HTTP 요청 창 동안 복잡한 작업을 처리 할 수 없는 웹 애플리케이션에서 특히 유용합니다.

## Preparation

이 자습서의 이전 부분에서 "Hello World!"가 포함 된 메시지를 보냈습니다. 이제 복잡한 작업을 나타내는 문자열을 보낼 것입니다. 크기를 조정할 이미지 나 렌더링 할 pdf 파일과 같은 실제 작업이 없으므로 바쁜 척하면서 가짜로 만들어 보겠습니다. - Thread.Sleep () 함수를 사용하여(스레딩 API에 액세스하려면 파일 상단 근처에 System.Threading을 사용하여 추가해야합니다.).

문자열의 점 수를 복잡성으로 간주합니다. 모든 점은 1 초의 "작업"을 설명합니다. 예를 들어 Hello ...가 설명하는 가짜 작업은 3 초가 걸립니다.

명령 줄에서 임의의 메시지를 보낼 수 있도록 이전 예제의 Send 프로그램을 약간 수정합니다. 이 프로그램은 작업 대기열에 작업을 예약하므로 이름을 NewTask로 지정하겠습니다.

튜토리얼 1과 마찬가지로 두 개의 프로젝트를 생성해야합니다.

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

channel.BasicPublish(exchange: "",

routingKey: "task\_queue",

basicProperties: properties,

body: body);

dotnet new console --name NewTask

mv NewTask/Program.cs NewTask/NewTask.cs

dotnet new console --name Worker

mv Worker/Program.cs Worker/Worker.cs

cd NewTask

dotnet add package RabbitMQ.Client

dotnet restore

cd ../Worker

dotnet add package RabbitMQ.Client

dotnet restore

명령 줄 인수에서 메시지를 얻는 데 도움이 됩니다.

var message = GetMessage(args);

private static string GetMessage(string[] args)

{

return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");

}

이전 Receive.cs 스크립트도 약간의 변경이 필요합니다. 메시지 본문의 모든 점에 대해 1 초의 작업을 위조해야 합니다. RabbitMQ에서 전달한 메시지를 처리하고 작업을 수행하므로 Worker 프로젝트에 복사하고 수정하겠습니다.

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

};

channel.BasicConsume(queue: "task\_queue", autoAck: true, consumer: consumer);

실행 시간을 시뮬레이션하는 가짜 작업 :

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

## Round-robin dispatching

작업 대기열 사용의 장점 중 하나는 작업을 쉽게 병렬 처리 할 수 있다는 것입니다. 작업 백 로그를 구축하는 경우 작업자를 더 추가하기 만하면 쉽게 확장 할 수 있습니다.

먼저 두 개의 Worker 인스턴스를 동시에 실행 해 보겠습니다. 둘 다 대기열에서 메시지를 가져 오지만 정확히 어떻게 할까요? 보자.

세 개의 콘솔이 열려 있어야합니다. 두 사람은 Worker 프로그램을 실행합니다. 이 콘솔은 C1과 C2의 두 소비자가 될 것입니다.

*# shell 2*

cd Worker

dotnet run

*# => [\*] Waiting for messages. To exit press CTRL+C*

*# shell 1*

cd Worker

dotnet run

*# => [\*] Waiting for messages. To exit press CTRL+C*

세 번째 항목에서는 새 작업을 게시합니다. 소비자를 시작한 후에는 몇 가지 메시지를 게시 할 수 있습니다.

*# shell 3*

cd NewTask

dotnet run "First message."

dotnet run "Second message.."

dotnet run "Third message..."

dotnet run "Fourth message...."

dotnet run "Fifth message....."

작업자에게 전달되는 내용을 살펴 보겠습니다.

*# shell 2*

*# => [\*] Waiting for messages. To exit press CTRL+C*

*# => [x] Received 'Second message..'*

*# => [x] Received 'Fourth message....'*

*# shell 1*

*# => [\*] Waiting for messages. To exit press CTRL+C*

*# => [x] Received 'First message.'*

*# => [x] Received 'Third message...'*

*# => [x] Received 'Fifth message.....'*

기본적으로 RabbitMQ는 각 메시지를 순서대로 다음 소비자에게 보냅니다. 평균적으로 모든 소비자는 동일한 수의 메시지를 받습니다. 이러한 메시지 배포 방법을 라운드 로빈이라고 합니다. 3 명 이상의 작업자와 함께 사용해보십시오.

## Message acknowledgment

작업을 수행하는 데 몇 초가 걸릴 수 있습니다. 소비자 중 한 명이 긴 작업을 시작하고 일부만 수행 한 채로 죽으면 어떻게 될지 궁금 할 것입니다. 현재 코드를 사용하면 RabbitMQ가 소비자에게 메시지를 전달하면 즉시 삭제 표시를 합니다. 이 경우 작업자를 죽이면 방금 처리 중이던 메시지를 잃게 됩니다. 또한이 특정 작업자에게 발송되었지만 아직 처리되지 않은 모든 메시지가 손실됩니다.

그러나 우리는 어떤 일도 잃고 싶지 않습니다. 작업자가 사망하면 작업이 다른 작업자에게 전달되기를 바랍니다.

메시지가 손실되지 않도록 하기 위해 RabbitMQ는 메시지 확인을 지원([message acknowledgments](https://www.rabbitmq.com/confirms.html))합니다. ack(nowledgement)은 특정 메시지가 수신되고 처리되었으며 RabbitMQ가 자유롭게 삭제할 수 있음을 RabbitMQ에 알리기 위해 소비자에 의해 다시 전송됩니다.

소비자가 ack를 보내지 않고 죽으면 (채널이 닫히거나, 연결이 닫히거나, TCP 연결이 끊어지면) RabbitMQ는 메시지가 완전히 처리되지 않았 음을 인식하고 다시 큐에 넣습니다. 동시에 온라인에 다른 소비자가 있는 경우 다른 소비자에게 신속하게 다시 제공합니다. 이렇게 하면 작업자가 가끔 죽어도 메시지가 손실되지 않도록 할 수 있습니다.

메시지 시간 제한이 없습니다. RabbitMQ는 소비자가 사망하면 메시지를 다시 전달합니다. 메시지를 처리하는 데 아주 오랜 시간이 걸리더라도 괜찮습니다.

수동 메시지 확인([Manual message acknowledgments](https://www.rabbitmq.com/confirms.html))은 기본적으로 켜져 있습니다. 이전 예에서는 autoAck("자동 승인 모드") 매개 변수를 true로 설정하여 명시 적으로 해제했습니다. 작업이 완료되면 이 플래그를 제거하고 작업자로부터 적절한 승인을 수동으로 보낼 때입니다.

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (sender, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

*// Note: it is possible to access the channel via*

*// ((EventingBasicConsumer)sender).Model here*

channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);

};

channel.BasicConsume(queue: "task\_queue", autoAck: false, consumer: consumer);

이 코드를 사용하면 메시지를 처리하는 동안 CTRL + C를 사용하여 작업자를 죽이더라도 아무것도 손실되지 않는다는 것을 확신 할 수 있습니다. 작업자가 사망 한 후 곧 확인되지 않은 모든 메시지가 다시 전달됩니다.

전달을 받은 동일한 채널에서 확인을 보내야합니다. 다른 채널을 사용하여 승인하려고 하면 채널 수준 프로토콜 예외가 발생합니다. 자세한 내용은 확인 문서 가이드([doc guide on confirmations](https://www.rabbitmq.com/confirms.html))를 참조하세요.

#### Forgotten acknowledgment

BasicAck를 놓치는 것은 흔한 실수입니다. 쉬운 오류이지만 그 결과는 심각합니다. 클라이언트가 종료되면 메시지가 재전송되지만 (무작위 재전송처럼 보일 수 있음) RabbitMQ는 확인되지 않은 메시지를 해제 할 수 없기 때문에 점점 더 많은 메모리를 소모합니다.

이런 종류의 실수를 디버깅하기 위해 rabbitmqctl을 사용하여 messages\_unacknowledged 필드를 인쇄 할 수 있습니다.

sudo rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

Windows에서는 sudo를 삭제하십시오.

rabbitmqctl.bat list\_queues name messages\_ready messages\_unacknowledged

## Message durability

우리는 소비자가 죽더라도 작업이 손실되지 않도록 하는 방법을 배웠습니다. 그러나 RabbitMQ 서버가 중지되면 우리의 작업은 여전히 손실됩니다.

RabbitMQ가 종료되거나 충돌하면 사용자가 지정하지 않는 한 큐와 메시지를 잊습니다. 메시지가 손실되지 않도록 하려면 두 가지가 필요합니다. 대기열과 메시지를 모두 내구성 있는 것으로 표시해야 합니다.

먼저 RabbitMQ 노드를 다시 시작해도 큐가 유지되는지 확인해야합니다. 이렇게 하려면 내구성을 선언해야 합니다.

channel.QueueDeclare(queue: "hello",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

이 명령은 그 자체로는 정확하지만 현재 설정에서는 작동하지 않습니다. 그것은 우리가 이미 내구성이 없는 hello라는 큐를 정의했기 때문입니다. RabbitMQ는 다른 매개 변수로 기존 큐를 재정의하는 것을 허용하지 않으며 이를 시도하는 모든 프로그램에 오류를 반환합니다. 그러나 빠른 해결 방법이 있습니다. 예를 들어 task\_queue와 같이 다른 이름으로 대기열을 선언 해 보겠습니다.

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

이 QueueDeclare 변경은 생산자와 소비자 코드 모두에 적용되어야합니다.

이 시점에서 RabbitMQ가 다시 시작 되더라도 task\_queue 대기열이 손실되지 않을 것입니다. 이제 IBasicProperties.SetPersistent를 true로 설정하여 메시지를 지속적으로 표시해야 합니다.

#### Note on Message Persistence

메시지를 지속성으로 표시한다고해서 메시지가 손실되지 않는다는 보장은 없습니다. RabbitMQ에 메시지를 디스크에 저장하도록 지시하지만 RabbitMQ가 메시지를 수락하고 아직 저장하지 않은 경우에는 여전히 짧은 시간 창이 있습니다. 또한 RabbitMQ는 모든 메시지에 대해 fsync (2)를 수행하지 않습니다. 캐시에 저장되고 실제로 디스크에 기록되지 않을 수 있습니다. 지속성 보장은 강력하지 않지만 간단한 작업 대기열에는 충분합니다. 더 강력한 보증이 필요한 경우 게시자 확인([publisher confirms](https://www.rabbitmq.com/confirms.html))을 사용할 수 있습니다.

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

## Putting It All Together

두 개의 터미널 열기

먼저 소비자 (작업자)를 실행하여 토폴로지 (주로 큐)가 제자리에 있도록 합니다. 전체 코드는 다음과 같습니다.

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

using System.Threading;

class Worker

{

public static void Main()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);

Console.WriteLine(" [\*] Waiting for messages.");

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (sender, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

*// Note: it is possible to access the channel via*

*// ((EventingBasicConsumer)sender).Model here*

channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);

};

channel.BasicConsume(queue: "task\_queue",

autoAck: false,

consumer: consumer);

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

Console.ReadLine();

}

}

}

이제 작업 게시자를 실행합니다. 최종 코드는 다음과 같습니다.

using System;

using RabbitMQ.Client;

using System.Text;

class NewTask

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

channel.BasicPublish(exchange: "",

routingKey: "task\_queue",

basicProperties: properties,

body: body);

Console.WriteLine(" [x] Sent {0}", message);

}

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

Console.ReadLine();

}

private static string GetMessage(string[] args)

{

return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");

}

}

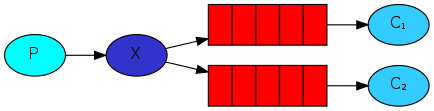
[(NewTask.cs source)](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/NewTask/NewTask.cs)

메시지 확인 및 BasicQos를 사용하여 작업 대기열을 설정할 수 있습니다. 내구성 옵션을 사용하면 RabbitMQ를 다시 시작하더라도 작업이 유지됩니다.

IModel 메서드 및 IBasicProperties에 대한 자세한 내용은 온라인으로 RabbitMQ .NET 클라이언트 API 참조([RabbitMQ .NET client API reference online](https://www.rabbitmq.com/releases/rabbitmq-dotnet-client/v3.6.10/rabbitmq-dotnet-client-3.6.10-client-htmldoc/html/index.html))를 찾아 볼 수 있습니다.

이제 자습서 3([tutorial 3](https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html))으로 이동하여 동일한 메시지를 많은 소비자에게 전달하는 방법을 배울 수 있습니다.

## [Publish/Subscribe](https://www.rabbitmq.com/tutorials/tutorial-three-python.html)

한 번에 많은 소비자에게 메시지 보내기

## Publish/Subscribe

### (using the .NET Client)

이전 자습서에서 작업 대기열을 만들었습니다. 작업 대기열 뒤에 있는 가정은 각 작업이 정확히 한 명의 작업자에게 전달된다는 것입니다. 이 부분에서 우리는 완전히 다른 것을 할 것입니다. -- 여러 소비자에게 메시지를 전달합니다. 이 패턴을 "게시/구독"이라고 합니다.

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

패턴을 설명하기 위해 간단한 로깅 시스템을 구축 할 것입니다. 두 프로그램으로 구성됩니다. -- 첫 번째는 로그 메시지를 내보내고 두 번째는 수신하여 인쇄합니다.

로깅 시스템에서 실행중인 모든 수신자 프로그램 사본은 메시지를 받습니다. 이렇게 하면 하나의 수신기를 실행하고 로그를 디스크로 보낼 수 있습니다. 동시에 다른 수신기를 실행하고 화면에서 로그를 볼 수 있습니다.

기본적으로 게시 된 로그 메시지는 모든 수신자에게 브로드 캐스트됩니다.

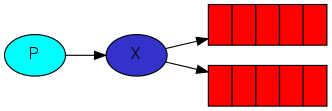
## Exchanges

튜토리얼의 이전 부분에서는 큐와 메시지를 주고 받았습니다. 이제 Rabbit에서 전체 메시징 모델을 소개 할 때입니다.

이전 튜토리얼에서 다룬 내용을 빠르게 살펴 보겠습니다.

* 생산자는 메시지를 보내는 사용자 응용 프로그램입니다.
* 큐는 메시지를 저장하는 버퍼입니다.
* 소비자는 메시지를 수신하는 사용자 응용 프로그램입니다.

RabbitMQ 메시징 모델의 핵심 아이디어는 생산자가 메시지를 대기열에 직접 보내지 않는다는 것입니다. 사실, 생산자는 메시지가 어떤 큐로 전달 될지 조차 알지 못하는 경우가 많습니다.

대신 생산자는 교환에만 메시지를 보낼 수 있습니다. 교환은 매우 간단합니다. 한쪽에서는 생산자로부터 메시지를 받고 다른 쪽에서는 큐로 전달합니다. 교환기는 수신 한 메시지로 무엇을 해야하는지 정확히 알아야합니다. 특정 대기열에 추가해야 합니까? 많은 대기열에 추가해야 합니까? 아니면 버려야합니다. 이에 대한 규칙은 교환 유형에 의해 정의됩니다.

사용 가능한 몇 가지 교환 유형이 있습니다: 직접, 주제, 헤더 및 팬 아웃. 마지막 팬 아웃에 초점을 맞출 것입니다. 이 유형의 교환을 만들고 로그라고 합시다.

channel.ExchangeDeclare("logs", ExchangeType.Fanout);

팬 아웃 교환은 매우 간단합니다. 이름에서 짐작할 수 있듯이 수신하는 모든 메시지를 알고있는 모든 큐에 브로드 캐스트합니다. 이것이 바로 로거에 필요한 것입니다.

#### Listing exchanges

서버에서 교환을 나열하려면 유용한 rabbitmqctl을 실행할 수 있습니다.

sudo rabbitmqctl list\_exchanges

이 목록에는 amq.\* 교환과 기본 (이름 없음) 교환이 있습니다. 기본적으로 생성되지만 지금은 사용할 필요가 없을 것입니다.

#### The default exchange

튜토리얼의 이전 부분에서 우리는 교환에 대해 전혀 몰랐지만 여전히 큐에 메시지를 보낼 수있었습니다. 빈 문자열 ("")로 식별되는 기본 교환을 사용했기 때문에 가능했습니다.

이전에 메시지를 게시 한 방법을 기억하십시오.

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "",

routingKey: "hello",

basicProperties: null,

body: body);

첫 번째 매개 변수는 교환의 이름입니다. 빈 문자열은 기본 또는 이름없는 교환을 나타냅니다. 메시지가있는 경우 routingKey에서 지정한 이름을 사용하여 큐로 라우팅됩니다.

이제 대신 명명 된 교환에 게시 할 수 있습니다.

## Temporary queues

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "logs",

routingKey: "",

basicProperties: null,

body: body);

이전에 기억할 수 있듯이 특정 이름을 가진 대기열을 사용했습니다 (hello 및 task\_queue를 기억하십니까?). 대기열의 이름을 지정할 수 있는 것은 우리에게 매우 중요했습니다. 작업자에게 동일한 대기열을 지정해야 했습니다. 생산자와 소비자간에 대기열을 공유하려는 경우 대기열에 이름을 지정하는 것이 중요합니다.

그러나 그것은 우리 로거의 경우가 아닙니다. 일부 로그 메시지가 아닌 모든 로그 메시지에 대해 듣고 싶습니다. 우리는 또한 오래된 메시지가 아닌 현재 흐르는 메시지에만 관심이 있습니다. 이를 해결하려면 두 가지가 필요합니다.

첫째, Rabbit에 연결할 때마다 새롭고 빈 대기열이 필요합니다. 이를 위해 임의의 이름으로 대기열을 만들거나 더 나은 방법으로 서버가 임의의 대기열 이름을 선택하도록 할 수 있습니다.

둘째, 소비자의 연결을 끊으면 대기열이 자동으로 삭제됩니다.

.NET 클라이언트에서 QueueDeclare()에 매개 변수를 제공하지 않으면 생성 된 이름으로 비 영구적이고 배타적 인 자동 삭제 대기열을 만듭니다.

var queueName = channel.QueueDeclare().QueueName;

대기열 가이드([guide on queues](https://www.rabbitmq.com/queues.html))에서 배타적(exclusive)플래그 및 기타 대기열 속성에 대해 자세히 알아볼 수 있습니다.

이 시점에서 queueName에는 임의의 대기열 이름이 포함됩니다.

예를 들어 amq.gen-JzTY20BRgKO-HjmUJj0wLg처럼 보일 수 있습니다.

둘째, 소비자 연결이 닫히면 대기열을 삭제해야 합니다. 이에 대한 독점(exclusive)플래그가 있습니다.

result = channel.queue\_declare(queue='', exclusive=True)

대기열 가이드([guide on queues](https://www.rabbitmq.com/queues.html))에서 배타적(exclusive) 플래그 및 기타 대기열 속성에 대해 자세히 알아볼 수 있습니다.

## https://www.rabbitmq.com/img/tutorials/bindings.pngBindings

우리는 이미 팬 아웃 교환과 대기열을 만들었습니다. 이제 우리는 큐에 메시지를 보내도록 교환에 알려야 합니다. 교환과 대기열 간의 관계를 바인딩이라고 합니다.

channel.QueueBind(queue: queueName,

exchange: "logs",

routingKey: "");

이제부터 로그(logs) 교환은 큐에 메시지를 추가합니다.

#### Listing bindings

당신은 그것을 사용하여 기존 바인딩을 나열 할 수 있습니다.

rabbitmqctl list\_bindings

## https://www.rabbitmq.com/img/tutorials/python-three-overall.pngPutting it all together

로그 메시지를 내보내는 생산자 프로그램은 이전 튜토리얼과 크게 다르지 않습니다. 가장 중요한 변경 사항은 이제 이름없는 메시지 대신 logs 교환에 메시지를 게시하려고한다는 것입니다. 보낼 때 routingKey를 제공해야하지만 팬 아웃 교환에서는 해당 값이 무시됩니다. EmitLog.cs 파일의 코드는 다음과 같습니다.

using System;

using RabbitMQ.Client;

using System.Text;

class EmitLog

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "logs",

routingKey: "",

basicProperties: null,

body: body);

Console.WriteLine(" [x] Sent {0}", message);

}

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

Console.ReadLine();

}

private static string GetMessage(string[] args)

{

return ((args.Length > 0)

? string.Join(" ", args)

: "info: Hello World!");

}

}

[(EmitLog.cs source)](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/EmitLog/EmitLog.cs)

보시다시피 연결을 설정 한 후 교환을 선언했습니다. 이 단계는 존재하지 않는 거래소에 대한 게시가 금지되어 있으므로 필요합니다.

아직 교환에 바인딩 된 대기열이 없으면 메시지가 손실되지만 괜찮습니다. 아직 듣고있는 소비자가 없는 경우 메시지를 안전하게 삭제할 수 있습니다.

ReceiveLogs.cs의 코드

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

class ReceiveLogs

{

public static void Main()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);

var queueName = channel.QueueDeclare().QueueName;

channel.QueueBind(queue: queueName,

exchange: "logs",

routingKey: "");

Console.WriteLine(" [\*] Waiting for logs.");

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] {0}", message);

};

channel.BasicConsume(queue: queueName,

autoAck: true,

consumer: consumer);

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

Console.ReadLine();

}

}

}

[(ReceiveLogs.cs source)](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/ReceiveLogs/ReceiveLogs.cs)

튜토리얼 1([tutorial one](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html))의 설정 지침에 따라 EmitLogs 및 ReceiveLogs 프로젝트를 생성합니다.

로그를 파일에 저장하려면 콘솔을 열고 다음을 입력하십시오.

cd ReceiveLogs

dotnet run > logs\_from\_rabbit.log

화면에서 로그를 보려면 새 터미널을 생성하고 다음을 실행하십시오.

cd ReceiveLogs

dotnet run

물론 로그를 내보내려면 다음을 입력하십시오.

cd EmitLog

dotnet run

rabbitmqctl list\_bindings를 사용하면 코드가 실제로 원하는 대로 바인딩과 큐를 생성하는지 확인할 수 있습니다. 두 개의 ReceiveLogs.cs프로그램이 실행되면 다음과 같은 내용이 표시됩니다.

sudo rabbitmqctl list\_bindings

*# => Listing bindings ...*

*# => logs exchange amq.gen-JzTY20BRgKO-HjmUJj0wLg queue []*

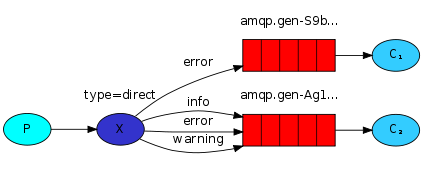
*# => logs exchange amq.gen-vso0PVvyiRIL2WoV3i48Yg queue []*

*# => ...done.*

결과의 해석은 간단합니다. 교환 로그(logs)의 데이터는 서버가 할당 한 이름을 사용하여 두 개의 대기열로 이동합니다. 이것이 바로 우리가 의도 한 것입니다.

메시지 하위 집합을 수신하는 방법을 알아 보려면 자습서 4([tutorial 4](https://www.rabbitmq.com/tutorials/tutorial-four-dotnet.html))로 이동하겠습니다.

## [Routing](https://www.rabbitmq.com/tutorials/tutorial-four-python.html)

선택적으로 메시지 받기

## Routing

### (using the .NET client)

이전 튜토리얼([previous tutorial](https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html))에서 우리는 간단한 로깅 시스템을 구축했습니다. 로그 메시지를 많은 수신자에게 브로드 캐스트 할 수 있었습니다.

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

이 자습서에서는 여기에 기능을 추가 할 것입니다. 메시지의 하위 집합에만 구독 할 수 있도록 할 것입니다. 예를 들어, 콘솔에서 모든 로그 메시지를 인쇄 할 수 있는 동안에도 중요한 오류 메시지 만 로그 파일로 보낼 수 있습니다 (디스크 공간 절약).

## Bindings

이전 예제에서 우리는 이미 바인딩을 만들고있었습니다. 다음과 같은 코드를 기억할 수 있습니다.

channel.QueueBind(queue: queueName,

exchange: "logs",

routingKey: "");

바인딩은 교환과 대기열 간의 관계입니다. 이것은 간단히 읽을 수 있습니다. 큐는 이 교환의 메시지에 관심이 있습니다.

바인딩은 추가 routingKey 매개 변수를 사용할 수 있습니다. BasicPublish 매개 변수와의 혼동을 피하기 위해 이를 binding key라고 부를 것입니다. 다음은 키로 바인딩을 만드는 방법입니다.

channel.QueueBind(queue: queueName,

exchange: "direct\_logs",

routingKey: "black");

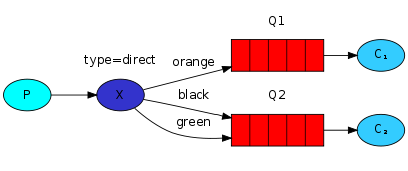
바인딩 키의 의미는 교환 유형에 따라 다릅니다. 이전에 사용했던 fanout 거래소는 단순히 그 가치를 무시했습니다.

## Direct exchange

이전 자습서의 로깅 시스템은 모든 메시지를 모든 소비자에게 브로드 캐스트합니다. 심각도에 따라 메시지를 필터링 할 수 있도록 확장하려고 합니다. 예를 들어, 로그 메시지를 디스크에 기록하는 스크립트가 심각한 오류 만 수신하고 경고 또는 정보 로그 메시지에 디스크 공간을 낭비하지 않도록 할 수 있습니다.

우리는 많은 유연성을 제공하지 않는 fanout 교환을 사용하고있었습니다. 이는 무의미한 방송 만 가능합니다.

대신 direct 교환을 사용합니다. direct 교환 뒤의 라우팅 알고리즘은 간단합니다. 메시지는 binding key가 메시지의 routing key와 정확히 일치하는 큐로 이동합니다.

이를 설명하기 위해 다음 설정을 고려하십시오.

이 설정에서는 두 개의 대기열이 바인딩 된 direct 교환 X를 볼 수 있습니다. 첫 번째 대기열은 바인딩 키 orange으로 바인딩되고 두 번째 큐에는 바인딩 키가 black이고 다른 하나는 green 인 두 개의 바인딩이 있습니다.

이러한 설정에서 라우팅 키가 orange 인 Exchange에 게시 된 메시지는 Q1 대기열로 라우팅됩니다. 라우팅 키가 black 또는 green 인 메시지는 Q2로 이동합니다. 다른 모든 메시지는 삭제됩니다.

## https://www.rabbitmq.com/img/tutorials/direct-exchange-multiple.pngMultiple bindings

동일한 바인딩 키로 여러 큐를 바인딩하는 것은 완벽하게 합법적입니다. 이 예에서는 바인딩 키가 black인 X와 Q1 사이에 바인딩을 추가 할 수 있습니다. 이 경우 direct 교환은 fanout 처럼 작동하고 모든 일치하는 대기열에 메시지를 브로드 캐스트합니다. 라우팅 키가 black인 메시지는 Q1 및 Q2 모두에 전달됩니다.

## Emitting logs

로깅 시스템에 이 모델을 사용할 것입니다. fanout 대신 direct 교환에 메시지를 보냅니다. 로그 심각도를 routing key로 제공합니다. 이렇게 하면 수신 스크립트가 수신하려는 심각도를 선택할 수 있습니다. 먼저 로그를 내보내는 데 중점을 둡니다.

항상 그렇듯이 먼저 교환을 만들어야합니다.

channel.ExchangeDeclare(exchange: "direct\_logs", type: "direct");

이제 메시지를 보낼 준비가되었습니다.

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "direct\_logs",

routingKey: severity,

basicProperties: null,

body: body);

단순화하기 위해 '심각도'는 '정보', '경고', '오류'중 하나 일 수 있다고 가정합니다.

## Subscribing

메시지 수신은 이전 튜토리얼과 동일하게 작동하지만, 관심있는 각 심각도에 대한 새 바인딩을 만들 것입니다.

var queueName = channel.QueueDeclare().QueueName;

foreach(var severity in args)

{

channel.QueueBind(queue: queueName,

exchange: "direct\_logs",

routingKey: severity);

}

## https://www.rabbitmq.com/img/tutorials/python-four.pngPutting it all together

EmitLogDirect.cs 클래스의 코드:

using System;

using System.Linq;

using RabbitMQ.Client;

using System.Text;

class EmitLogDirect

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "direct\_logs",

type: "direct");

var severity = (args.Length > 0) ? args[0] : "info";

var message = (args.Length > 1)

? string.Join(" ", args.Skip( 1 ).ToArray())

: "Hello World!";

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "direct\_logs",

routingKey: severity,

basicProperties: null,

body: body);

Console.WriteLine(" [x] Sent '{0}':'{1}'", severity, message);

}

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

Console.ReadLine();

}

}

ReceiveLogsDirect.cs의 코드:

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

class ReceiveLogsDirect

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "direct\_logs",

type: "direct");

var queueName = channel.QueueDeclare().QueueName;

if(args.Length < 1)

{

Console.Error.WriteLine("Usage: {0} [info] [warning] [error]",

Environment.GetCommandLineArgs()[0]);

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

Console.ReadLine();

Environment.ExitCode = 1;

return;

}

foreach(var severity in args)

{

channel.QueueBind(queue: queueName,

exchange: "direct\_logs",

routingKey: severity);

}

Console.WriteLine(" [\*] Waiting for messages.");

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

var routingKey = ea.RoutingKey;

Console.WriteLine(" [x] Received '{0}':'{1}'",

routingKey, message);

};

channel.BasicConsume(queue: queueName,

autoAck: true,

consumer: consumer);

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

Console.ReadLine();

}

}

}

평소와 같이 프로젝트를 만듭니다 (자문은 [tutorial one](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html) 참조).

'경고'및 '오류'( '정보'제외) 로그 메시지 만 파일에 저장하려면 콘솔을 열고 다음을 입력하십시오.

cd ReceiveLogsDirect

dotnet run warning error > logs\_from\_rabbit.log

화면에서 모든 로그 메시지를 보려면 새 터미널을 열고 다음을 수행하십시오.

cd ReceiveLogsDirect

dotnet run info warning error

*# => [\*] Waiting for logs. To exit press CTRL+C*

예를 들어 error 로그 메시지를 내보내려면 다음을 입력하십시오.

cd EmitLogDirect

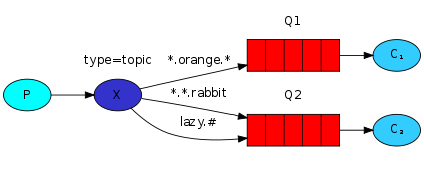
dotnet run error "Run. Run. Or it will explode."

*# => [x] Sent 'error':'Run. Run. Or it will explode.'*

([(EmitLogDirect.cs source)](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/EmitLogDirect/EmitLogDirect.cs) 및 [(ReceiveLogsDirect.cs source)](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/ReceiveLogsDirect/ReceiveLogsDirect.cs)의 전체 소스 코드)

패턴을 기반으로 메시지를 수신하는 방법을 알아 보려면 [tutorial 5](https://www.rabbitmq.com/tutorials/tutorial-five-dotnet.html)로 이동하십시오.

## [Topics](https://www.rabbitmq.com/tutorials/tutorial-five-python.html)

패턴 (주제)을 기반으로 메시지 받기

## Topics

### (using the .NET client)

이전 튜토리얼에서 우리는 로깅 시스템을 개선했습니다. 더미 방송 만 가능한 fanout 교환기를 사용하는 대신 direct 교환기를 사용하여 로그를 선별 적으로 수신 할 수 있는 가능성을 얻었습니다.

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

직접 교환을 사용하면 시스템이 개선되었지만 여전히 한계가 있습니다. 여러 기준에 따라 라우팅을 수행 할 수 없습니다.

로깅 시스템에서 심각도뿐만 아니라 로그를 생성 한 소스에 따라 로그를 구독 할 수 있습니다. 심각도 (info / warn / crit ...)와 기능 (auth / cron / kern ...)을 기반으로 로그를 라우팅하는 [syslog](http://en.wikipedia.org/wiki/Syslog) unix 도구에서 이 개념을 알 수 있습니다.

그것은 우리에게 많은 유연성을 제공 할 것입니다. 우리는 'cron'에서 오는 중요한 오류 뿐만 아니라 'kern'의 모든 로그를 듣고 싶을 수 있습니다.

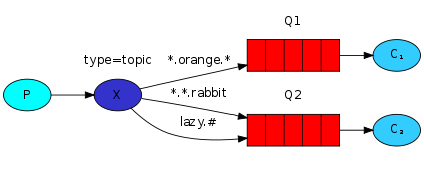
로깅 시스템에서이를 구현하려면 더 복잡한 topic 교환에 대해 배워야합니다.

## Topic exchange

topic 교환으로 전송 된 메시지는 임의의 routing\_key를 가질 수 없습니다. 점으로 구분 된 단어 목록이어야 합니다. 단어는 무엇이든 될 수 있지만 일반적으로 메시지에 연결된 일부 기능을 지정합니다. 몇 가지 유효한 라우팅 키 예: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit". 라우팅 키에는 최대 255 바이트까지 원하는 만큼 많은 단어가 포함될 수 있습니다.

바인딩 키도 동일한 형식이어야 합니다. topic 교환의 논리는 direct 논리와 유사합니다. 특정 라우팅 키로 전송 된 메시지는 일치하는 바인딩 키로 바인딩 된 모든 큐로 전달됩니다. 그러나 키 바인딩에는 두 가지 중요한 특수한 경우가 있습니다.

* \* (별표)는 정확히 한 단어를 대체 할 수 있습니다.
* # (해시)는 0개 이상의 단어를 대체 할 수 있습니다.

예를 들어 설명하는 것이 가장 쉽습니다.

이 예에서 우리는 모두 동물을 설명하는 메시지를 보낼 것입니다. 메시지는 세 단어 (두 개의 점)로 구성된 라우팅 키와 함께 전송됩니다. 라우팅 키의 첫 번째 단어는 속도, 두 번째는 색상, 세 번째 단어는 “<speed>.<colour>.<species>"입니다.

세 가지 바인딩을 만들었습니다. Q1은 바인딩 키 "\*.orange.\*"로 바인딩되고 Q2는 "\*.\*.rabbit"및 "lazy.#"로 바인딩됩니다.

이러한 바인딩은 다음과 같이 요약 할 수 있습니다.

* Q1은 모든 주황색 동물에 관심이 있습니다.
* Q2는 토끼에 대한 모든 것, 게으른 동물에 대한 모든 것을 듣고 싶습니다.

라우팅 키가 "quick.orange.rabbit"로 설정된 메시지가 두 대기열에 모두 전달됩니다. "lazy.orange.elephant"메시지도 둘 다에 전달됩니다. 반면에 "quick.orange.fox"는 첫 번째 대기열로만 이동하고 "lazy.brown.fox"는 두 번째 대기열로만 이동합니다. "lazy.pink.rabbit"는 두 개의 바인딩과 일치하더라도 두 번째 큐에 한 번만 전달됩니다. "quick.brown.fox"는 바인딩과 일치하지 않으므로 폐기됩니다.

계약을 어 기고 "orange"또는 "quick.orange.male.rabbit"과 같이 1 ~ 4 개의 단어로 메시지를 보내면 어떻게 되나요? 이 메시지는 어떤 바인딩과도 일치하지 않으며 손실됩니다.

반면에 "lazy.orange.male.rabbit"은 4 개의 단어가 있어도 마지막 바인딩과 일치하고 두 번째 대기열로 전달됩니다.

## Putting it all together

#### Topic exchange

주제 교환은 강력하며 다른 교환과 같이 작동 할 수 있습니다.

큐가 "#"(해시) 바인딩 키로 바인딩되면 fanout 교환에서와 같이 라우팅 키에 관계없이 모든 메시지를 수신합니다.

특수 문자 "\*"(별표) 및 "#"(해시)가 바인딩에 사용되지 않으면 토픽 교환이 직접 교환처럼 작동합니다.

로깅 시스템에서 topic 교환을 사용할 것입니다.

로그의 라우팅 키에 " <facility>.<severity>"라는 두 단어가 있다는 가정부터 시작하겠습니다.

코드는 이전 튜토리얼([previous tutorial](https://www.rabbitmq.com/tutorials/tutorial-four-dotnet.html))과 거의 동일합니다.

EmitLogTopic.cs의 코드:

using System;

using System.Linq;

using RabbitMQ.Client;

using System.Text;

class EmitLogTopic

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "topic\_logs",

type: "topic");

var routingKey = (args.Length > 0) ? args[0] : "anonymous.info";

var message = (args.Length > 1)

? string.Join(" ", args.Skip( 1 ).ToArray())

: "Hello World!";

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "topic\_logs",

routingKey: routingKey,

basicProperties: null,

body: body);

Console.WriteLine(" [x] Sent '{0}':'{1}'", routingKey, message);

}

}

}

ReceiveLogsTopic.cs의 코드:

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

class ReceiveLogsTopic

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "topic\_logs", type: "topic");

var queueName = channel.QueueDeclare().QueueName;

if(args.Length < 1)

{

Console.Error.WriteLine("Usage: {0} [binding\_key...]",

Environment.GetCommandLineArgs()[0]);

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

Console.ReadLine();

Environment.ExitCode = 1;

return;

}

foreach(var bindingKey in args)

{

channel.QueueBind(queue: queueName,

exchange: "topic\_logs",

routingKey: bindingKey);

}

Console.WriteLine(" [\*] Waiting for messages. To exit press CTRL+C");

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

var routingKey = ea.RoutingKey;

Console.WriteLine(" [x] Received '{0}':'{1}'",

routingKey,

message);

};

channel.BasicConsume(queue: queueName,

autoAck: true,

consumer: consumer);

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

Console.ReadLine();

}

}

}

다음 예제를 실행하십시오.

모든 로그를 수신하려면:

cd ReceiveLogsTopic

dotnet run "#"

*:'Run. Run. Or it will explode.'*

기능 "kern"에서 모든 로그를 수신하려면:

cd ReceiveLogsTopic

dotnet run "kern.\*"

또는 "critical"로그에 대해서만 듣고 싶은 경우:

cd ReceiveLogsTopic

dotnet run "\*.critical"

여러 바인딩을 만들 수 있습니다.

cd ReceiveLogsTopic

dotnet run "kern.\*" "\*.critical"

라우팅 키가 "kern.critical"인 로그를 내보내려면 다음을 입력하십시오.

cd EmitLogTopic

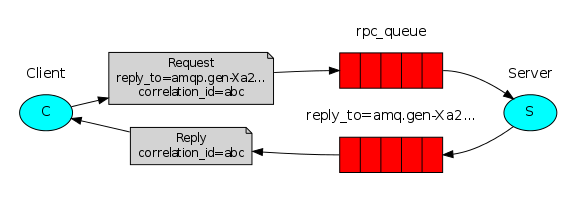
dotnet run "kern.critical" "A critical kernel error"

이 프로그램을 재미있게 즐기십시오. 코드는 라우팅 또는 바인딩 키에 대해 어떠한 가정도 하지 않으므로 두 개 이상의 라우팅 키 매개 변수를 사용하여 플레이 할 수 있습니다.

([EmitLogTopic.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/EmitLogTopic/EmitLogTopic.cs) 및 [ReceiveLogsTopic.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/ReceiveLogsTopic/ReceiveLogsTopic.cs)의 전체 소스 코드)

다음으로 튜토리얼 6([tutorial 6](https://www.rabbitmq.com/tutorials/tutorial-six-dotnet.html))에서 원격 프로 시저 호출로 왕복 메시지를 수행하는 방법을 알아 봅니다.

## [RPC](https://www.rabbitmq.com/tutorials/tutorial-six-python.html)

요청/응답 패턴([Request/reply pattern](http://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html)) 예

## Remote procedure call (RPC)

### (using the .NET client)

두 번째 자습서([second tutorial](https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html))에서는 작업 대기열을 사용하여 여러 작업자에게 시간이 많이 걸리는 작업을 배포하는 방법을 배웠습니다.

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

하지만 원격 컴퓨터에서 기능을 실행하고 결과를 기다려야한다면 어떨까요? 음, 그것은 다른 이야기입니다. 이 패턴은 일반적으로 원격 프로시저 호출 또는 RPC로 알려져 있습니다.

이 튜토리얼에서는 RabbitMQ를 사용하여 RPC 시스템, 즉 클라이언트 및 확장 가능한 RPC 서버를 구축 할 것입니다. 배포 할 가치가 있는 시간 소모적 인 작업이 없으므로 피보나치 수를 반환하는 더미 RPC 서비스를 만들 것입니다.

### Client interface

RPC 서비스를 사용하는 방법을 설명하기 위해 간단한 클라이언트 클래스를 만들 것입니다. RPC 요청을 보내고 응답을 받을 때까지 차단하는 Call이라는 메서드를 노출합니다:

var rpcClient = new RPCClient();

Console.WriteLine(" [x] Requesting fib(30)");

var response = rpcClient.Call("30");

Console.WriteLine(" [.] Got '{0}'", response);

rpcClient.Close();

#### A note on RPC

RPC는 컴퓨팅에서 매우 일반적인 패턴이지만 종종 비판을 받습니다. 프로그래머가 함수 호출이 로컬인지 또는 느린 RPC인지 알지 못할 때 문제가 발생합니다. 이와 같은 혼동은 예측할 수 없는 시스템을 초래하고 디버깅에 불필요한 복잡성을 추가합니다. 소프트웨어를 단순화하는 대신 RPC를 잘못 사용하면 유지 관리 할 수 없는 스파게티 코드가 발생할 수 있습니다.

이를 염두에 두고 다음 조언을 고려하십시오.

* 어떤 함수 호출이 로컬이고 어떤 것이 원격인지 분명해야 합니다.
* 시스템을 문서화하십시오. 구성 요소 간의 종속성을 명확히 합니다.
* 오류 사례를 처리합니다. RPC 서버가 오랫동안 다운되었을 때 클라이언트는 어떻게 반응해야 합니까?

의심스러운 경우 RPC를 피하십시오. 가능하다면 비동기 파이프 라인을 사용해야합니다. RPC와 같은 차단 대신 결과가 비동기 적으로 다음 계산 단계로 푸시 됩니다.

### Callback queue

일반적으로 RabbitMQ를 통해 RPC를 수행하는 것은 쉽습니다. 클라이언트는 요청 메시지를 보내고 서버는 응답 메시지로 응답합니다. 응답을 받으려면 요청과 함께 '콜백' 대기열 주소를 보내야합니다.

#### Message properties

AMQP 0-9-1 프로토콜은 메시지와 함께 제공되는 14 개의 속성 집합을 미리 정의합니다. 다음을 제외하고 대부분의 속성은 거의 사용되지 않습니다.

* Persistent: 메시지를 지속적 (값 2) 또는 일시적 (기타 값)으로 표시합니다. [두 번째 튜토리얼](https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html)을 살펴보십시오.
* DeliveryMode: 프로토콜에 익숙한 사용자는 Persistent 대신이 속성을 사용하도록 선택할 수 있습니다. 그들은 같은 것을 제어합니다.
* ContentType: 인코딩의 MIME 유형을 설명하는 데 사용됩니다. 예를 들어 자주 사용되는 JSON 인코딩의 경우이 속성을 application/json으로 설정하는 것이 좋습니다.
* ReplyTo: 일반적으로 콜백 대기열의 이름을 지정하는 데 사용됩니다.
* CorrelationId: RPC 응답을 요청과 연관시키는 데 유용합니다.

var props = channel.CreateBasicProperties();

props.ReplyTo = replyQueueName;

var messageBytes = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "",

routingKey: "rpc\_queue",

basicProperties: props,

body: messageBytes);

*// ... then code to read a response message from the callback\_queue ...*

### Correlation Id

위에 제시된 방법에서는 모든 RPC 요청에 대해 콜백 대기열을 만드는 것이 좋습니다. 이는 매우 비효율적이지만 다행히 더 나은 방법이 있습니다. 클라이언트 당 단일 콜백 대기열을 생성 해 보겠습니다.

그 큐에서 응답을 받았기 때문에 응답이 어떤 요청에 속하는지 명확하지 않은 새로운 문제가 발생합니다. 이때 CorrelationId 속성이 사용됩니다. 모든 요청에 대해 고유 한 값으로 설정합니다. 나중에 콜백 큐에서 메시지를 받으면 이 속성을 살펴보고 이를 기반으로 응답을 요청과 일치시킬 수 있습니다. 알 수 없는 CorrelationId 값이 표시되면 메시지를 안전하게 삭제할 수 있습니다. 이는 요청에 속하지 않습니다.

오류로 인해 실패하는 대신 콜백 대기열에서 알 수 없는 메시지를 무시해야하는 이유는 무엇입니까? 이는 서버 측의 경쟁 조건 가능성 때문입니다. 가능성은 낮지 만 RPC 서버가 응답을 보낸 직후 요청에 대한 승인 메시지를 보내기 전에 죽을 가능성이 있습니다. 이 경우 다시 시작된 RPC 서버가 요청을 다시 처리합니다. 그렇기 때문에 클라이언트에서 중복 응답을 우아하게 처리해야하며 RPC는 이상적으로는 멱 등성이 있어야합니다.

### https://www.rabbitmq.com/img/tutorials/python-six.pngSummary

RPC는 다음과 같이 작동합니다.

* 클라이언트가 시작되면 익명의 배타적 콜백 대기열이 생성됩니다.
* RPC 요청의 경우 클라이언트는 콜백 대기열에 설정된 ReplyTo와 모든 요청에 대해 고유 한 값으로 설정된 CorrelationId의 두 가지 속성이 있는 메시지를 보냅니다.
* 요청은 rpc\_queue 대기열로 전송됩니다.
* RPC 작업자 (일명: 서버)는 해당 큐에서 요청을 기다리고 있습니다. 요청이 나타나면 작업을 수행하고 ReplyTo 속성의 큐를 사용하여 결과가 포함 된 메시지를 클라이언트에 다시 보냅니다.
* 클라이언트는 콜백 대기열에서 데이터를 기다립니다. 메시지가 나타나면 CorrelationId 속성을 확인합니다. 요청의 값과 일치하면 응용 프로그램에 대한 응답을 반환합니다.

## Putting it all together

피보나치 작업:

private static int fib(int n)

{

if (n == 0 || n == 1) return n;

return fib(n - 1) + fib(n - 2);

}

피보나치 함수를 선언합니다. 유효한 양의 정수 입력 만 가정합니다. (이것이 큰 숫자에 대해 작동 할 것이라고 기대하지 마십시오. 아마도 가능한 가장 느린 재귀 구현 일 것입니다).

RPC 서버 [RPCServer.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/RPCServer/RPCServer.cs)의 코드는 다음과 같습니다:

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

class RPCServer

{

public static void Main()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using (var connection = factory.CreateConnection())

using (var channel = connection.CreateModel())

{

channel.QueueDeclare(queue: "rpc\_queue", durable: false,

exclusive: false, autoDelete: false, arguments: null);

channel.BasicQos(0, 1, false);

var consumer = new EventingBasicConsumer(channel);

channel.BasicConsume(queue: "rpc\_queue",

autoAck: false, consumer: consumer);

Console.WriteLine(" [x] Awaiting RPC requests");

consumer.Received += (model, ea) =>

{

string response = null;

var body = ea.Body.ToArray();

var props = ea.BasicProperties;

var replyProps = channel.CreateBasicProperties();

replyProps.CorrelationId = props.CorrelationId;

try

{

var message = Encoding.UTF8.GetString(body);

int n = int.Parse(message);

Console.WriteLine(" [.] fib({0})", message);

response = fib(n).ToString();

}

catch (Exception e)

{

Console.WriteLine(" [.] " + e.Message);

response = "";

}

서버 코드는 다소 간단합니다.

finally

{

var responseBytes = Encoding.UTF8.GetBytes(response);

channel.BasicPublish(exchange: "", routingKey: props.ReplyTo,

basicProperties: replyProps, body: responseBytes);

channel.BasicAck(deliveryTag: ea.DeliveryTag,

multiple: false);

}

};

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

Console.ReadLine();

}

}

*///*

*/// Assumes only valid positive integer input.*

*/// Don't expect this one to work for big numbers, and it's*

*/// probably the slowest recursive implementation possible.*

*///*

private static int fib(int n)

{

if (n == 0 || n == 1)

{

return n;

}

return fib(n - 1) + fib(n - 2);

}

}

* 평소와 같이 연결, 채널을 설정하고 대기열을 선언하는 것으로 시작합니다.
* 둘 이상의 서버 프로세스를 실행할 수 있습니다. 부하를 여러 서버에 균등하게 분산하려면 channel.BasicQos에서 prefetchCount 설정을 설정해야 합니다.
* BasicConsume을 사용하여 큐에 액세스합니다. 그런 다음 작업을 수행하고 응답을 다시 보내는 전달 핸들러를 등록합니다.

RPC 클라이언트 [RPCClient.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/RPCClient/RPCClient.cs)의 코드:

using System;

using System.Collections.Concurrent;

using System.Text;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

public class RpcClient

{

private readonly IConnection connection;

private readonly IModel channel;

private readonly string replyQueueName;

private readonly EventingBasicConsumer consumer;

private readonly BlockingCollection<string> respQueue = new BlockingCollection<string>();

private readonly IBasicProperties props;

public RpcClient()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

connection = factory.CreateConnection();

channel = connection.CreateModel();

replyQueueName = channel.QueueDeclare().QueueName;

consumer = new EventingBasicConsumer(channel);

props = channel.CreateBasicProperties();

var correlationId = Guid.NewGuid().ToString();

props.CorrelationId = correlationId;

props.ReplyTo = replyQueueName;

consumer.Received += (model, ea) =>

{

var body = ea.Body.ToArray();

var response = Encoding.UTF8.GetString(body);

if (ea.BasicProperties.CorrelationId == correlationId)

{

respQueue.Add(response);

}

};

}

public string Call(string message)

{

var messageBytes = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(

exchange: "",

routingKey: "rpc\_queue",

basicProperties: props,

body: messageBytes);

channel.BasicConsume(

consumer: consumer,

queue: replyQueueName,

autoAck: true);

return respQueue.Take();

클라이언트 코드는 약간 더 복잡합니다.

}

public void Close()

{

connection.Close();

}

}

public class Rpc

{

public static void Main()

{

var rpcClient = new RpcClient();

Console.WriteLine(" [x] Requesting fib(30)");

var response = rpcClient.Call("30");

Console.WriteLine(" [.] Got '{0}'", response);

rpcClient.Close();

}

}

* 우리는 연결과 채널을 설정하고 응답을 위한 독점적 인 '콜백'큐를 선언합니다.
* 우리는 RPC 응답을 받을 수 있도록 '콜백'큐를 구독합니다.
* Call 메서드는 실제 RPC 요청을합니다.
* 여기에서 먼저 고유 한 CorrelationId 번호를 생성하고이를 저장하여 도착 시 적절한 응답을 식별합니다.
* 다음으로, 두 가지 속성 인 ReplyTo 및 CorrelationId와 함께 요청 메시지를 게시합니다.
* 이 시점에서 우리는 적절한 응답이 도착할 때까지 앉아서 기다릴 수 있습니다.
* 모든 응답 메시지에 대해 클라이언트는 CorrelationId가 우리가 찾고있는 것인지 확인합니다. 그렇다면 응답을 저장합니다.
* 마지막으로 응답을 사용자에게 반환합니다.

클라이언트 요청하기:

var rpcClient = new RPCClient();

Console.WriteLine(" [x] Requesting fib(30)");

var response = rpcClient.Call("30");

Console.WriteLine(" [.] Got '{0}'", response);

rpcClient.Close();

이제 [RPCClient.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/RPCClient/RPCClient.cs) 및 [RPCServer.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/RPCServer/RPCServer.cs)에 대한 전체 예제 소스 코드 (기본 예외 처리 포함)를 살펴볼 좋은 시간입니다.

평소대로 설정합니다 ([tutorial one](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html) 참조).

이제 RPC 서비스가 준비되었습니다. 서버를 시작할 수 있습니다.

cd RPCServer

dotnet run

*# => [x] Awaiting RPC requests*

피보나치 번호를 요청하려면 클라이언트를 실행하십시오.

cd RPCClient

dotnet run

*# => [x] Requesting fib(30)*

여기에 제시된 디자인은 RPC 서비스를 구현할 수 있는 유일한 방법은 아니지만 몇 가지 중요한 이점이 있습니다.

* RPC 서버가 너무 느린 경우 다른 서버를 실행하여 확장 할 수 있습니다. 새 콘솔에서 두 번째 RPCServer를 실행 해보십시오.
* 클라이언트 측에서 RPC는 하나의 메시지 만 보내고 받아야합니다. QueueDeclare와 같은 동기 호출이 필요하지 않습니다. 결과적으로 RPC 클라이언트는 단일 RPC 요청에 대해 하나의 네트워크 왕복 만 필요합니다.

우리의 코드는 여전히 매우 단순하며 다음과 같이 더 복잡하지만 중요한 문제를 해결하려고 하지 않습니다.

* 실행중인 서버가 없는 경우 클라이언트는 어떻게 반응해야 합니까?
* 클라이언트에 RPC에 대한 일종의 시간 제한이 있어야합니까?
* 서버가 오작동하고 예외가 발생하면 클라이언트로 전달해야 합니까?
* 처리하기 전에 유효하지 않은 수신 메시지 (예: 경계, 유형 확인)로부터 보호합니다.

실험하고 싶다면 대기열을 보는 데 [유용한 관리 UI](https://www.rabbitmq.com/management.html)를 찾을 수 있습니다.

## [Publisher Confirms](https://www.rabbitmq.com/tutorials/tutorial-seven-java.html)

게시자 확인을 통한 안정적인 게시

## Publisher Confirms

게시자 확인([Publisher confirms](https://www.rabbitmq.com/confirms.html#publisher-confirms))은 안정적인 게시를 구현하기 위한 RabbitMQ 확장이다. 게시자 확인이 채널에서 활성화되면 클라이언트가 게시하는 메시지는 브로커에 의해 비동기 적으로 확인됩니다. 즉, 서버 측에서 처리되었습니다.

**전제 조건**

이 가이드에서는 RabbitMQ가 [표준 포트](https://www.rabbitmq.com/networking.html#ports) (5672)의 localhost에 [설치](https://www.rabbitmq.com/download.html)되어 실행되고 있다고 가정합니다. 다른 호스트, 포트 또는 자격 증명을 사용하는 경우 연결 설정을 조정해야 합니다.

**도움을 받을 수 있는 곳**

이 튜토리얼을 진행하는 데 문제가 있는 경우 [메일링 리스트](https://groups.google.com/forum/#!forum/rabbitmq-users) 또는 [RabbitMQ 커뮤니티 Slack](https://rabbitmq-slack.herokuapp.com/)을 통해 문의 할 수 있습니다.

### (using the .NET client)

### Overview

이 튜토리얼에서는 발행인 확인을 사용하여 발행 된 메시지가 브로커에 안전하게 도달했는지 확인합니다. 게시자 확인을 사용하는 몇 가지 전략을 다루고 장단점을 설명합니다.

### Enabling Publisher Confirms on a Channel

게시자 확인은 AMQP 0.9.1 프로토콜에 대한 RabbitMQ 확장이므로 기본적으로 활성화되어 있지 않습니다. 게시자 확인은 ConfirmSelect 메서드를 사용하여 채널 수준에서 활성화됩니다.

var channel = connection.CreateModel();

channel.ConfirmSelect();

이 메소드는 게시자 확인을 사용하려는 모든 채널에서 호출되어야 합니다. 확인은 게시 된 모든 메시지가 아닌 한 번만 활성화해야 합니다.

### Strategy #1: Publishing Messages Individually

확인을 사용하여 게시하는 가장 간단한 방법, 즉 메시지를 게시하고 확인을 동 기적으로 대기하는 것부터 시작하겠습니다.

while (ThereAreMessagesToPublish())

{

byte[] body = ...;

BasicProperties properties = ...;

channel.BasicPublish(exchange, queue, properties, body);

*// uses a 5 second timeout*

channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));

}

이전 예에서는 평소와 같이 메시지를 게시하고 Channel#WaitForConfirmsOrDie(TimeSpan) 메서드로 확인을 기다립니다. 이 메서드는 메시지가 확인되는 즉시 반환됩니다. 시간 초과 내에 메시지가 확인되지 않거나 메시지가 확인되지 않은 경우 (브로커가 어떤 이유로 메시지를 처리 할 수 없음을 의미) 메서드는 예외를 throw합니다. 예외 처리는 일반적으로 오류 메시지 로깅 및 / 또는 메시지 전송 재시 도로 구성됩니다.

클라이언트 라이브러리마다 게시자 확인을 동기식으로 처리하는 방법이 다르므로 사용중인 클라이언트의 문서를 주의 깊게 읽어야합니다.

이 기술은 매우 간단하지만 중요한 단점도 있습니다. 메시지 확인이 모든 후속 메시지의 게시를 차단하므로 **게시 속도가 크게 느려**집니다. 이 접근 방식은 초당 수백 개의 게시 된 메시지 처리량을 제공하지 않습니다. 그럼에도 불구하고 이것은 일부 응용 프로그램에 충분할 수 있습니다.

#### Are Publisher Confirms Asynchronous?

처음에 브로커가 게시 된 메시지를 비동기 적으로 확인한다고 언급했지만 첫 번째 예제에서는 메시지가 확인 될 때까지 코드가 동 기적으로 대기합니다. 클라이언트는 실제로 비동기 적으로 확인을 수신하고 그에 따라 WaitForConfirmsOrDie에 대한 호출을 차단 해제합니다. WaitForConfirmsOrDie를 내부적으로 비동기 알림에 의존하는 동기식 도우미로 생각하십시오.

### Strategy #2: Publishing Messages in Batches

이전 예제를 개선하기 위해 메시지 배치를 게시하고 전체 배치가 확인 될 때까지 기다릴 수 있습니다. 다음 예에서는 배치 100을 사용합니다.

var batchSize = 100;

var outstandingMessageCount = 0;

while (ThereAreMessagesToPublish())

{

byte[] body = ...;

BasicProperties properties = ...;

channel.BasicPublish(exchange, queue, properties, body);

outstandingMessageCount++;

if (outstandingMessageCount == batchSize)

{

channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));

outstandingMessageCount = 0;

}

}

if (outstandingMessageCount > 0)

{

channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));

}

일련의 메시지가 확인 될 때까지 기다리면 개별 메시지에 대한 확인을 기다리는 것보다 처리량이 크게 향상됩니다 (원격 RabbitMQ 노드의 경우 최대 20-30 회). 한 가지 단점은 실패 시 무엇이 잘못되었는지 정확히 알지 못하므로 의미 있는 것을 기록하거나 메시지를 다시 게시하기 위해 전체 배치를 메모리에 보관해야 할 수 있다는 것입니다. 이 솔루션은 여전히 동기식이므로 메시지 게시를 차단합니다.

### Strategy #3: Handling Publisher Confirms Asynchronously

브로커는 게시 된 메시지를 비동기 적으로 확인합니다. 이러한 확인에 대한 알림을 받으려면 클라이언트에 콜백을 등록하기 만하면 됩니다.

var channel = connection.CreateModel();

channel.ConfirmSelect();

channel.BasicAcks += (sender, ea) =>

{

*// code when message is confirmed*

};

channel.BasicNacks += (sender, ea) =>

{

*//code when message is nack-ed*

};

두 개의 콜백이 있습니다. 하나는 확인 된 메시지 용이고 다른 하나는 nack-ed 메시지 (브로커가 손실 된 것으로 간주 할 수 있는 메시지) 용입니다. 두 콜백에는 다음을 포함하는 해당 EventArgs 매개 변수 (ea)가 있습니다.

* 배달 태그: 확인되거나 확인 된 메시지를 식별하는 시퀀스 번호. 이를 게시 된 메시지와 연관시키는 방법을 곧 살펴 보겠습니다.
* 배수: 부울 값입니다. 거짓이면 하나의 메시지 만 확인 / 확인되고 참이면 시퀀스 번호가 더 낮거나 같은 모든 메시지가 확인 / 확인됩니다.

게시하기 전에 Channel#NextPublishSeqNo를 사용하여 시퀀스 번호를 얻을 수 있습니다.

var sequenceNumber = channel.NextPublishSeqNo;

channel.BasicPublish(exchange, queue, properties, body);

메시지를 시퀀스 번호와 연관시키는 간단한 방법은 사전을 사용하는 것입니다. 문자열이 게시를 위해 바이트 배열로 변환하기 쉽기 때문에 문자열을 게시하려고 한다고 가정 해 보겠습니다. 다음은 사전을 사용하여 게시 시퀀스 번호를 메시지의 문자열 본문과 연관시키는 코드 샘플입니다.

var outstandingConfirms = new ConcurrentDictionary<ulong, string>();

*// ... code for confirm callbacks will come later*

var body = "...";

outstandingConfirms.TryAdd(channel.NextPublishSeqNo, body);

channel.BasicPublish(exchange, queue, properties, Encoding.UTF8.GetBytes(body));

이제 게시 코드는 사전을 사용하여 아웃 바운드 메시지를 추적합니다. 확인이 도착하면 이 딕셔너리를 정리하고 메시지가 nack-ed 될 때 경고를 기록하는 것과 같은 작업을 수행해야합니다.

이전 샘플에는 확인이 도착하면 사전을 정리하는 콜백이 포함되어 있습니다. 이 콜백은 단일 및 다중 확인을 모두 처리합니다. 이 콜백은 확인이 도착할 때 사용됩니다 (Channel#BasicAcks). nack-ed 메시지에 대한 콜백은 메시지 본문을 검색하고 경고를 발행합니다. 그런 다음 이전 콜백을 다시 사용하여 미결 확인 사전을 정리합니다 (메시지가 확인되었는지 여부에 관계없이 사전에서 해당 항목을 제거해야 함).

var outstandingConfirms = new ConcurrentDictionary<ulong, string>();

void cleanOutstandingConfirms(ulong sequenceNumber, bool multiple)

{

if (multiple)

{

var confirmed = outstandingConfirms.Where(k => k.Key <= sequenceNumber);

foreach (var entry in confirmed)

{

outstandingConfirms.TryRemove(entry.Key, out \_);

}

}

else

{

outstandingConfirms.TryRemove(sequenceNumber, out \_);

}

}

channel.BasicAcks += (sender, ea) => cleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);

channel.BasicNacks += (sender, ea) =>

{

outstandingConfirms.TryGetValue(ea.DeliveryTag, out string body);

Console.WriteLine($"Message with body {body} has been nack-ed. Sequence number: {ea.DeliveryTag}, multiple: {ea.Multiple}");

cleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);

};

*// ... publishing code*

#### How to Track Outstanding Confirms?

샘플은 ConcurrentDictionary를 사용하여 미결 확인을 추적합니다. 이 데이터 구조는 여러 가지 이유로 편리합니다. 시퀀스 번호를 메시지와 쉽게 연관시킬 수 있고 (메시지 데이터가 무엇이든) 주어진 시퀀스 ID까지 항목을 쉽게 정리할 수 있습니다 (여러 confirms/nacks를 처리하기 위해). 마지막으로 동시 액세스를 지원합니다. 확인 콜백이 클라이언트 라이브러리가 소유 한 스레드에서 호출되므로 게시 스레드와 다르게 유지되어야 합니다.

간단한 동시 해시 테이블 및 변수를 사용하여 게시 시퀀스의 하한을 추적하는 것과 같이 정교한 사전 구현을 사용하는 것보다 미결 확인을 추적하는 다른 방법이 있지만 일반적으로 더 복잡하고 자습서에 속하지 않습니다.

요약하자면 게시자 확인을 비동기식으로 처리하려면 일반적으로 다음 단계가 필요합니다.

* 게시 순서 번호를 메시지와 연관시키는 방법을 제공합니다.
* 게시자 acks/nacks가 도착하면 알림을 받을 채널에 확인 리스너를 등록하여 nack-ed 메시지를 기록하거나 다시 게시하는 것과 같은 적절한 작업을 수행합니다. 시퀀스 번호와 메시지 간의 상관 관계 메커니즘은이 단계 동안 일부 정리가 필요할 수도 있습니다.
* 메시지를 게시하기 전에 게시 순서 번호를 추적하십시오.

#### Re-publishing nack-ed Messages?

해당 콜백에서 nack-ed 메시지를 다시 게시하고 싶을 수 있지만 채널이 작업을 수행하지 않아야하는 I/O 스레드에서 확인 콜백이 발송되므로 피해야합니다. 더 나은 솔루션은 게시 스레드에 의해 폴링되는 메모리 내 대기열에 메시지를 넣는 것입니다. ConcurrentQueue와 같은 클래스는 확인 콜백과 게시 스레드간에 메시지를 전송하기에 좋은 후보입니다.

### Summary

게시 된 메시지가 브로커에 전송되었는지 확인하는 것은 일부 애플리케이션에서 필수적 일 수 있습니다. 게시자는 이 요구 사항을 충족하는 데 도움이 되는 RabbitMQ 기능을 확인합니다. 게시자 확인은 본질적으로 비동기 적이지만 동 기적으로 처리 할 수도 있습니다. 게시자 확인을 구현하는 확실한 방법은 없습니다. 이는 일반적으로 애플리케이션과 전체 시스템의 제약으로 귀결됩니다. 일반적인 기술은 다음과 같습니다.

* 메시지를 개별적으로 게시하고 동시에 확인을 기다립니다. 단순하지만 매우 제한된 처리량.
* 일괄 적으로 메시지를 게시하고 일괄 확인을 위해 동 기적으로 대기: 간단하고 합리적인 처리량이지만 문제가 발생했을 때 추론하기 어렵습니다.
* 비동기 처리: 최상의 성능 및 리소스 사용, 오류 발생시 좋은 제어, 하지만 올바르게 구현하는 데 관여 할 수 있습니다.

## Putting It All Together

[PublisherConfirms.cs](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/dotnet/PublisherConfirms/PublisherConfirms.cs) 클래스에는 우리가 다룬 기술에 대한 코드가 포함되어 있습니다. 컴파일하고있는 그대로 실행하고 각각의 성능을 확인할 수 있습니다.

dotnet run

출력은 다음과 같습니다.

Published 50,000 messages individually in 5,549 ms

Published 50,000 messages in batch in 2,331 ms

Published 50,000 messages and handled confirms asynchronously in 4,054 ms

클라이언트와 서버가 동일한 시스템에 있는 경우 컴퓨터의 출력은 유사하게 표시됩니다. 메시지를 개별적으로 게시하면 예상대로 성능이 좋지 않지만 비동기 처리 결과는 일괄 게시에 비해 약간 실망스럽습니다.

게시자 확인은 네트워크에 매우 의존적이므로 클라이언트와 서버가 일반적으로 프로덕션에서 동일한 시스템에 있지 않기 때문에 더 현실적인 원격 노드로 시도하는 것이 좋습니다. PublisherConfirms.cs는 로컬이 아닌 노드를 사용하도록 쉽게 변경할 수 있습니다.

private static IConnection CreateConnection()

{

var factory = new ConnectionFactory { HostName = "remote-host", UserName = "remote-host", Password = "remote-password" };

return factory.CreateConnection();

}

클래스를 다시 컴파일하고 다시 실행 한 다음 결과를 기다립니다.

Published 50,000 messages individually in 231,541 ms

Published 50,000 messages in batch in 7,232 ms

Published 50,000 messages and handled confirms asynchronously in 6,332 ms

이제 개별 게시가 끔찍하게 수행되는 것을 볼 수 있습니다. 그러나 클라이언트와 서버 간의 네트워크를 사용하면 이제 일괄 게시와 비동기 처리가 비슷하게 수행되고 게시자 확인의 비동기 처리에 대한 작은 이점이 있습니다.

일괄 게시는 구현하기 쉽지만 부정적인 게시자 승인의 경우 브로커에 전달할 수 없는 메시지를 쉽게 알 수 없습니다. 게시자를 비동기식으로 처리하는 것이 구현에 더 많이 관련되어 있음을 확인하지만 게시 된 메시지가 잠길 때 수행 할 작업에 대해 더 나은 세분성과 더 나은 제어를 제공합니다.