#### **RabbitMQ 주요 컴포넌트**

* **프로듀서 (Producer)**: 메시지를 생성하여 RabbitMQ로 전송.  
  메시지는 Exchange (교환기)로 먼저 전달.
* **Exchange (교환기)**: 프로듀서가 보낸 메시지를 받아 적절한 큐(Queue)로 라우팅.  
  Exchange 타입에 따라 메시지를 큐로 분배하는 방식이 달라짐.
  + - **Direct Exchange**: 라우팅 키(Routing Key) 기반 매칭
    - **Fanout Exchange**: 모든 바인딩된 큐에 메시지 전달 (Broadcast)
    - **Topic Exchange**: 특정 패턴(와일드카드 포함)에 맞는 큐로 메시지 전달
    - **Headers Exchange**: 메시지 헤더 값을 기반으로 큐를 선택
* **큐 (Queue)**: 메시지를 저장하는 공간이며, FIFO 방식으로 동작.
  + 컨슈머가 메시지를 가져갈 때까지 메시지를 유지.
  + 내구성(Durability) 옵션을 설정하면 RabbitMQ 서버 재시작 시에도 메시지 유지.
* **바인딩 (Binding)**: Exchange와 Queue를 연결하는 설정.
  + 메시지가 어떤 조건으로 어떤 큐에 들어갈지 결정.
  + 하나의 Queue가 여러 Exchange에 연결될 수 있고,  
    하나의 Exchange가 여러 Queue에 연결될 수 있다.  
    하나의 Queue가 하나의 Exchange에 bindingKey만 달리해서 여러번 연결도 가능
* **컨슈머 (Consumer)**: 큐에서 메시지를 가져와 처리.
  + 메시지를 가져가는 방식:
    - **Pull 방식**: 컨슈머가 직접 큐에서 메시지를 요청해서 가져감.
    - **Push 방식**: RabbitMQ가 컨슈머에게 메시지를 자동으로 전달.
* **채널 (Channel)**: 하나의 TCP 연결 내에서 여러 개의 가상 연결을 만들 수 있도록 하는 논리적인 단위. TCP 연결을 효율적으로 사용하기 위해 사용.
* **Connection (TCP 연결)**: 클라이언트(Producer, Consumer)와 RabbitMQ 브로커 간의 물리적인 연결.
  + 성능 최적화를 위해 하나의 Connection에서 여러 개의 Channel을 사용하는 것이 일반적.

#### Broker는 Exchange, Binding, Queue로 구성된다.

routingKey와 bindingKey 가 같아도, 대상하는 Exchange가 다르면 메시지 수신이 안된다.

#### 

#### **RabbitMQ 메시지 흐름**

1. 프로듀서가 메시지를 생성하여 특정 Exchange로 보냄.
2. Exchange는 바인딩된 큐에 메시지를 전달 (Exchange 타입에 따라 다르게 라우팅).
3. 큐는 메시지를 보관하고, 컨슈머가 메시지를 소비할 때까지 유지.
4. 컨슈머가 메시지를 가져와 처리한 후,  
   RabbitMQ에 메시지 처리가 완료 되었음을 알림 (ACK).  
   \* 기본적으로 메시지는 컨슈머가 ACK를 보내기 전까지 큐에 남아 있음.

#### 

+

**AMQP?**

Advanced Message Queuing Protocol의 약자로,

MQ(Message Queuing)기반의 프로토콜을 의미함.

**C# 프로젝트 생성 및 RabbitMQ.Client 설치**

| dotnet new console --name <projectName> dotnet add package RabbitMQ.Client |
| --- |

#### **Docker로 RabbitMQ 실행**

| docker run -d --rm --name rabbit-container -p 5672:5672 rabbitmq:3-management |
| --- |

#### 

#### **RabbitMQ 추가 개념**

**Ack (Acknowledgment)**:  
컨슈머가 메시지를 정상적으로 처리했음을 RabbitMQ에 알리는 방식.

* **Auto Ack:**  
  사용하면 RabbitMQ는 메시지를 즉시 삭제하지만, 실패 시 메시지 유실 위험이 있음.
* 메시지 확인(acknowledgment) 기능은 소비자가 RabbitMQ에 특정 메시지가 수신되어 처리되었으며, 이제 RabbitMQ가 삭제해도 된다는 신호를 보내는 방식.
* Ack를 누락하는 것은 흔한 실수이며, acknowledgment가 누락되면 클라이언트가 종료될 때 메시지가 재전송되고, RabbitMQ는 미확인 메시지를 반환할 수 없기 때문에 점점 더 많은 메모리를 소비하게 되므로 메시지를 처리한 후 반드시 Ack을 보내야 한다.
* 소비자에게 메시지를 전달한 후에는 30분(기본값)의 타임아웃이 적용되어  
  소비자가 ack을 보내지 않으면 자동으로 메시지가 재전송됨.

**QoS (Quality of Service)**:  
컨슈머가 처리할 수 있는 메시지 개수를 제한하는 기능.

* ex) basic.qos(prefetch=1) 설정 시, 컨슈머는 한 번에 하나의 메시지만 처리.
* prefetchSize: 소비자가 한 번에 받을 수 있는 메시지의 총 크기를 설정.
* **prefetchCount**: 소비자가 한 번에 가져올 수 있는 메시지 수를 설정.
* global: prefetchCount가 큐 단위가 아닌, 채널 전체에 적용될지를 설정.

**Dead Letter Queue (DLQ)**:  
실패하거나 만료된 메시지를 저장하는 큐.

* 메시지 TTL(Time-To-Live)이 지나거나, 최대 재시도 횟수를 초과하면 DLQ로 이동.

**Cluster & High Availability**:   
RabbitMQ는 여러 개의 노드로 구성된 **클러스터**를 지원하여 확장성과 내구성을 높일 수 있음.

* Mirrored Queue를 사용하면 여러 노드에 메시지를 복제하여 장애 시에도 메시지 유실을 방지 가능.

**라운드 로빈 분배**:  
RabbitMQ는 기본적으로 각 메시지를 라운드 로빈 방식으로  
소비자에게 순차적으로 전송하여 평균적으로 각 소비자가 동일한 수의 메시지를 받게 함.

**영속성 설정:**

기본적으로 RabbitMQ는 서버가 중지되면 큐와 메시지가 손실된다.

메시지가 손실되지 않게 하려면 두 가지를 설정해야 한다.

영속성 설정을 한 항목들은 디스크에 저장되어, 서버 재시작 시 이를 복구할 수 있다.

큐 영속성 설정:

| await channel.QueueDeclareAsync(queue: "hello",  durable: true, exclusive: false,  autoDelete: false, arguments: null); |
| --- |

메시지 영속성 설정:

| var properties = new BasicProperties { Persistent = true }; await channel.BasicPublishAsync(exchange: string.Empty,  routingKey: queueName,  mandatory: true,  basicProperties: properties,  body: body); |
| --- |

#### **기본 교환기 (Default Exchange)**

RabbitMQ에서는 빈 문자열("")이 기본 교환기를 나타냄.  
기본 교환기는 routingKey와 동일한 이름을 가진 큐가 있으면, 해당 큐로 메시지를 자동으로 라우팅.

즉, Direct Exchange로 동작함.

| var body = Encoding.UTF8.GetBytes(message); await channel.BasicPublishAsync(exchange: string.Empty,  routingKey: "hello",  body: body); |
| --- |

**임시 큐 (Temporary Queues)**

새로운 연결마다 빈 큐를 생성하고, 소비자가 연결을 종료하면 큐가 자동으로 삭제되도록 설정.

* 비내구성 (Non-durable): 서버가 재시작되면 사라짐.
* 독점적 (Exclusive): 현재 연결에서만 사용 가능, 연결이 끊어지면 삭제됨.
* 자동 삭제 (Auto-delete): 소비자가 연결을 종료하면 큐 삭제됨.

| var queueName = await channel.QueueDeclareAsync(); |
| --- |

**메시지 속성**

AMQP 0-9-1 프로토콜은 메시지와 함께 사용되는 14개의 속성을 미리 정의.

* Persistent: 메시지를 영속적(true) 또는 일시적(false)으로 설정.
* DeliveryMode: 이 속성을 Persistent 대신 사용할 수 있으며, 동일한 것을 제어.
* ContentType: 인코딩의 MIME 타입을 설명하는 데 사용.
* ReplyTo: 콜백 큐의 이름을 지정하는 데 일반적으로 사용.
* CorrelationId: RPC 응답과 요청을 연결하는 데 유용.

**구현 참고:**[**https://www.rabbitmq.com/tutorials/tutorial-one-dotnet**](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet)

**WorkQueues**

“direct exchange으로 설정” << 큐에 바로 넣어놔야 해서 direct로 하나..?

작업을 수행하는 데 시간이 걸리는 작업들을 분배하는 데 유용.

#### **Publish/Subscribe**

“fanout exchange으로 설정.”

하나의 메시지를 여러 소비자에게 전달하는 방식.

발행된 로그 메시지가 모든 수신자에게 브로드캐스트(broadcast)됨.

생산자는 메시지를 직접 큐로 보내지 않고 "교환기(exchange)"로 메시지를 보냄.

**Topic Exchange**

“topic exchange으로 설정.”

토픽 익스체인지로 전송되는 메시지는 점(dot)으로 구분된 단어 목록 형태의 routing\_key를 가짐.

* 각 단어는 메시지와 관련된 특징을 나타냄.
* binding\_key 역시 같은 형식을 따라야 함.

바인딩 키(binding key)에는 두 가지 특별한 기호가 있음:

* \* (별표): 정확히 한 단어를 대체할 수 있음.
* # (해시): 0개 이상의 단어를 대체할 수 있음.
* #만 사용: fanout처럼 동작 (모든 메시지 수신).
* \*, # 없이 사용: 직접(direct) 익스체인지처럼 동작 (완전히 일치하는 메시지만 수신).
* \*, #을 조합: 토픽(topic) 익스체인지의 강력한 패턴 매칭 기능 활용 가능.

**RPC (Remote Procedure Call)**

클라이언트는 요청 메시지를 보내고 서버는 응답 메시지로 회신하며,

응답을 받기 위해서는 요청과 함께 '콜백' 큐 주소를 보내야 함.

RPC는 다음과 같이 작동:

1. 클라이언트 시작: 클라이언트가 시작되면, 전용 콜백 큐를 생성.
2. RPC 요청: RPC 요청을 보낼 때,  
   클라이언트는 ReplyTo (콜백 큐)와 CorrelationId (고유한 값) 속성을 가진 메시지를 보냄.
3. 서버 처리: RPC 워커(서버)는 rpc\_queue 큐에서 요청을 기다림. 요청이 도착하면, 서버는 작업을 처리하고, 그 결과를 클라이언트에게 ReplyTo 속성에 설정된 큐로 전송하여 응답.
4. 클라이언트 응답 처리: 클라이언트는 콜백 큐에서 응답을 기다림. 메시지가 도착하면, 클라이언트는 CorrelationId 속성을 확인하여 요청에서 받은 값과 일치하는지 확인하고 응답을 애플리케이션에 반환.

**Publisher Confirms**

퍼블리셔 확인은 AMQP 0.9.1 프로토콜에 대한 RabbitMQ 확장이며 기본적으로 비활성 상태다.

메시지가 정상적으로 큐에 도달했는지 확인할 수 있는 기능이다.

publisher confirms는 confirm\_select 명령을 통해 활성화 된다.

**클러스터:**

<https://jeongchul.tistory.com/815>

<https://dodghek.tistory.com/35>

**RabbitMQ의 2가지 노드 타입:**

Disc 노드:

디스크에 메타데이터(큐 정의, 사용자 정보, Exchange, Binding 등)를 저장한다.

클러스터의 주요 정보를 관리하며, 최소 1개 이상의 Disc 노드가 필요하다.

주의할 것은 메타데이터가 저장된다는 뜻이다. 메시지가 저장된다는 뜻이 아니다.   
메시지의 경우, durable 및 persistent로 설정된 메시지만 디스크에 저장된다.

클러스터 내의 최소한 하나 이상의 노드는 반드시 Disc 노드여야 한다.

클러스터의 데이터 일관성과 지속성을 보장한다.

모든 클러스터 노드가 메타데이터를 공유하도록 한다.

RAM 노드:

메모리에만 메타데이터를 저장한다.

성능을 높이기 위해 사용되며, 메타데이터가 손실될 수 있다.

메모리에 저장되므로 더 빠른 응답 시간과 성능을 제공한다.

클러스터의 메타데이터 복제를 받지만, 디스크에 영구적으로 저장하지 않는다.

RAM 노드가 장애가 발생하면 클러스터 메타데이터를 잃을 수 있지만,

Disc 노드가 여전히 살아있다면 전체 클러스터에는 큰 영향이 없다.

**클러스터 구성:**

모든 노드를 Disc 노드로 구성:

메타데이터를 디스크에 저장하여 모든 노드가 장애 발생 시에도 데이터를 보존할 수 있다. 일반적으로 소규모 클러스터에서는 모든 노드를 Disc 노드로 설정하여 관리가 간단해진다.

혼합 구성:

일부 노드는 Disc 노드로 설정하고, 다른 노드는 RAM 노드로 설정한다.

메타데이터의 안정성과 클러스터 성능 간의 균형을 맞추는 용도이다.

이는 대규모 클러스터에서 자주 사용하는 방법이다.

ex\_)

\* 3개 노드 구성: 1개 Disk + 2개 RAM 노드

\* 5개 노드 구성: 2개 Disk + 3개 RAM 노드

**유일한 1개의 Disk 노드가 죽으면 데이터 손실이 발생하는가?**

Disk가 저장했던 내용들을 RAM노드가 동기화 받을 수 없어, 손실 발생한다.

**RabbitMQ의 클러스터는 기본적으로 큐 데이터를 공유하지 않는다.**

특정 노드가 다운되면 메시지가 유실될 수 있다.

노드들은 서로 Exchange와 Binding을 기본적으로 공유하지만,

Queue는 각자 가지기 때문에 특정 노드가 다운되면

해당 노드의 데이터 유실은 피할 수 없다.

메시지 손실을 감수할 수 있는 경우에는 고가용성을 지원하지 않아도 되겠다.

고가용성을 지원하지 않으면 성능은 더 잘나온다.

**Mirrored Queue:**

RabbitMQ 클러스터에서는 고가용성을 위해 큐를 미러링할 수 있다.

기본적으로 RabbitMQ의 큐는 싱글 노드에만 존재하지만,

미러링 큐(HA Queue)를 사용하면 여러 노드에 복제된다.

**미러링 큐 구성:**

Master 노드:

큐의 주된(Primary) 노드로, 큐의 메시지를 처리하고 저장하는 주체이다.

실제로 큐에 메시지를 추가하고 삭제하는 역할을 담당한다.

메시지가 소비될 때도 master 노드에서 소비되고 처리된다.

Master 노드는 다른 노드들에 비해 큐의 "진정한 상태"를 보유하고 있으며,

메시지의 소비와 ACK 처리가 이루어진다.

Slave 노드:

Slave 노드는 메시지를 실제로 처리하거나 소비하지 않으며,

단지 Master 노드의 데이터를 복제(백업)하는 역할을 한다.

**동기화 및 장애 처리:**

기본적으로 Message는 Master QeueuMQ에 넣고,  
그 후 모든 Slave Queue에 미러링 되어야,

Message를 보낸 Producer에게 ACK를 보낸다.

Master 노드가 장애가 발생하면,

미러링이 다 안된 노드는 제외하고,  
가장 오랫동안 Running 상태인 Slave 노드가 Master가 된다.

모두 미러링이 완전히 안된 상태에서는,

설정 옵션에 따라서 Master 노드가 선정 된다.

ex\_1) 그나마 최신 상태의 Slave 노드 선택

ex\_2) Master 복구될 때 까지 대기..

**미러링 큐 팁:**

1. 미러링 큐 구성 시 성능을 잘 고려해야 한다.  
   미러링을 많이할 수록 메시지 처리량이 떨어진다.  
   복제 할게 많으니까 당연히 느려진다.
2. 클러스터 내에 특정 큐만 미러링 할 수 있다.  
   일반 큐 + 미러링 큐 조합을 해도 된다.
3. 미러링 구성 시, LB 같은 걸 사용하는게 좋다.  
   producer, consumer는 master 노드를 대상으로만 동작을 하지만,  
   리더 노드 장애 시 slave 노드가 승격되며, 해당 노드에 접근해야 한다.  
   이를 위해 LB 같은게 필요할 수 있다

**Dead Letter Queue:**

RabbitMQ로 메시지를 처리하다 문제가 발생하면,

다음 메시지를 처리하기 위해 이 메시지를 건너뛰어야 하는 경우가 있다.

이 메시지를 건너뛰어야 꾸준히 들어오는 다음 메시지를 처리할 수 있기 때문이다.

일시적으로 다른 API 호출이나 연결 문제가 있는 메시지는

나중에 재시도 하면 정상처리 될 수 있기 때문에 문제가 되는 메시지는 잠시 다른 곳에 보관한다.

queue에 dead letter 설정을 해두고, nack나 reject를 보내면 된다.

이 때, requeue는 false로 설정해서 보내야 한다.

auto ack가 true로 설정되어 있으면,

메시지를 가져오면 자동으로 ack를 보내서 RabbitMQ 서버에서 메시지가 사라지게 된다.

메시지를 가져와서 처리가 끝날 때 수동으로 ack를 보낼 수 있도록 auto ack를 false로 설정하자.

메시지를 가져와서 ack를 보내기 전 까지는

이 메시지를 다른 컨슈머에게 주지도 않고, 큐에서 제거하지도 않는다.

정상적으로 처리가 되지 않을 때, ack 대신 nack를 보내면 된다.

nack나 reject를 보낼 때 requeue=false로 설정하지 않으면,

이 메시지는 큐의 원래 위치로 돌아가게 된다.

그러면 이 컨슈머나 혹은 다른 컨슈머가 그걸 다시 가져와서 처리하게 된다.

이렇게 다른 컨슈머가 바로 처리해도 되는 비즈니스가 맞는 곳이라면

dead letter를 설정하지 말고 그대로 requeue 하면 된다.

**Transaction:**

1. 프로듀서 트랜잭션
   * 메시지를 MQ에 전송하기 전 트랜잭션 시작
   * 메시지를 큐에 게시한 후 트랜잭션 커밋 or 롤백
   * 실패 시 메시지가 큐로 producing 되지 않음
   * 단, 속도 문제가 있으며 비동기라는 특징에 부합하지 않음
2. **컨슈머 트랜잭션**
   * 메시지를 수신한 후 트랜잭션 시작 - RabbitMQ 브로커에 도달했음을 알림
   * 메시지를 처리한 뒤 커밋
   * 실패 시 메시지를 다시 처리하거나 DLQ로 이동
   * 대규모 메시지 처리 방식에 권장

**트랜잭션 메시징의 한계**

1. **성능 오버헤드 :**트랜잭션은 추가적인 작업(예: 디스크 동기화)을 요구하므로 성능이 떨어질수있음  
   **대량 메시지 처리**가 필요한 경우 부적합.
2. **복잡성:**트랜잭션 처리를 잘못 구성하면 메시지가 중복 처리되거나 손실될 가능성 증가**.**
3. **분산 트랜잭션을 완벽하게 보장하지는 않음.**   
   데이터베이스 작업과 메시지 전송 사이의 완벽한 원자성을 위해서는 추가적인 메커니즘 필요
4. **AMQP Confirm Select 추천:**  
   RabbitMQ에서는 트랜잭션 대신 **Publisher Confirms**(AMQP Confirm Select)이 효율적

**필요하면 더 찾아보기:**

**Stream:**

큐와 동일한 작업을 수행할 수 있는 영속적인 복제 데이터 구조이다.

이는 생산자로부터 메시지를 버퍼링하고 소비자가 이를 읽는 방식이다.

그러나 스트림은 메시지가 저장되고 소비되는 방식에서 큐와 두 가지 중요한 차이점이 있다.

스트림은 만료될 때까지 반복해서 읽을 수 있는 메시지의 추가 전용 로그를 모델링한다.

스트림은 항상 영속적이고 복제된다.

스트림의 메시지를 읽으려면 하나 이상의 소비자가 해당 스트림에 구독하여

원하는 만큼 동일한 메시지를 읽을 수 있다.  
  
**Super Stream:**

분할 스트림이다.

**Direct reply to:**

Direct reply-to를 사용하면 RPC 클라이언트는 응답 큐를 거치지 않고 직접 RPC 서버로부터 응답을 받게 된다. 여기서 "직접"이라는 말은 여전히 동일한 채널과 RabbitMQ 노드를 거친다는 의미이다. 즉, RPC 클라이언트와 RPC 서버 프로세스 간에 직접적인 네트워크 연결은 없다.

이 메커니즘을 사용하여 보내는 응답 메시지는 일반적으로 내결함성이 없다.  
원래 요청을 발행한 클라이언트가 이후에 연결을 끊으면 메시지가 폐기된다.

이 경우 RPC 클라이언트가 재연결하여 다른 요청을 제출할 것이라는 가정이 있다.

**Local Random Exchange:**

RabbitMQ 4.0에서 새롭게 추가된 교환 유형이다.

주로 요청-응답("RPC") 사용 사례를 위해 설계되었다.

이 교환 유형에서는 메시지가 항상 로컬 큐에 전달되며,

이를 통해 최소한의 발행자 대기 시간이 보장된다.

만약 여러 개의 로컬 큐가 교환에 바인딩되어 있으면,

그 중 하나가 무작위로 선택되어 메시지를 전달한다.

**Blocked Connection Notifications:**

클라이언트가 브로커의 리소스(메모리 또는 디스크)가 부족해

연결이 차단될 때 알림을 받는 것이 바람직할 수 있다.

이를 위해 AMQP 0-9-1 프로토콜 확장을 도입하여,

브로커가 연결이 차단되었을 때 클라이언트에게 connection.blocked 메서드를 보내고,

연결이 차단 해제되었을 때는 connection.unblocked 메서드를 보낸다.

**Exchange to Exchange Bindings:**

exchange.bind 메서드가 도입되어 한 익스체인지를 다른 익스체인지에 바인딩할 수 있다.

이 바인딩은 익스체인지-큐 바인딩과 의미적으로 동일하다다.

단방향이며, 바인딩 키와 익스체인지 유형은 정상적으로 작동하지만,

바인딩의 두 끝점은 모두 익스체인지이다.

이를 통해 훨씬 더 풍부한 라우팅 토폴로지를 생성할 수 있다.

**Delay Queue:**

메시지가 큐에 도달한 후 일정 시간 동안 **지연된 후**에만 소비되도록 설정된 큐입니다. 기본적으로 메시지가 큐에 들어가면 즉시 소비가 가능하지만, **지연 큐**는 **미리 설정된 시간만큼 기다린 후**에만 메시지를 처리할 수 있게 합니다.

기능을 따로 지원하는 건 아니고, TTL이랑 DLX로 직접 구현하는 듯함.

ex\_)

예약된 이메일 전송이나 알림을 특정 시간 후에 보내는 경우

실패한 작업을 일정 시간 뒤에 재시도 할 경우

**Alternate Exchange:**

때때로 클라이언트가 익스체인지가 라우팅할 수 없는 메시지를   
처리하도록 하는 것이 바람직할 수 있다.

ex\_) 바인딩된 큐가 없거나 일치하는 바인딩이 없는 경우

구성된 AE(대체 익스체인지)가 있는 익스체인지에 메시지를 발행 했을 때,   
해당 메시지가 어떤 큐로도 라우팅되지 않으면, 채널은 메시지를 지정된 AE로 다시 발행 한다.

**Federation:**

다양한 RabbitMQ 인스턴스나 클러스터가 서로 데이터를 공유할 수 있도록 해주는 기능이다.이를 통해 서로 다른 RabbitMQ 서버 간에 메시지를 주고받을 수 있으며, 분산 메시징 시스템을 구현할 때 유용하다. 페더레이션은 RabbitMQ 클러스터 간에 메시지를 **동기적으로** 복제하지 않고, **비동기적으로** 메시지를 전달한다. 즉, 다른 RabbitMQ 인스턴스의 큐나 교환소(exchange)에 있는 메시지를 가져와서 처리할 수 있다.

사용 예시)

여러 지리적 위치에서 RabbitMQ 서버가 있을 때,

페더레이션을 통해 서로 다른 위치에 있는 메시지를 공유할 수 있다.