# RabbitMQ

- Distributed **message broker**
  - **Message broker:** software that allows applications to communicate with each other by exchanging messages
    - Ex: user signed in to account from a new device
- **Advantages:** allows for loosely-coupled applications and new parts of systems to be seamlessly added without affecting existing systems.
- **Commonly** used in **Microservices Style Architecture**
- **Asynchronous**
- Can be **deployed** on many clouds as well as on premise.
- **Default messaging protocol:** AMQP

## Windows and Docker Installation

- Install *Erlang (≥ 26.0 && < 27.0)*
  - https://erlang.org/download/otp_versions_tree.html
- Install RabbitMQ
  - https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.13.3/rabbitmq-server-3.13.3.exe

| **RabbitMQ version** | **Minimum required Erlang/OTP** | **Notes** |
|---|---|---|
| 3.13.0 | 26.0 | • The 3.13 release is compatible with Erlang<br>• *Erlang 27 (latest)* is not supported |

## Chocolatey Installation (Preferred)

- To install chocolatey, run the following command:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile
-InputFormat None -ExecutionPolicy Bypass -Command
"[System.Net.ServicePointManager]::SecurityProtocol = 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/instal
l.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

  - Type choco or choco -? now
- To install RabbitMQ, run the following command (installs erlang as well):

```
choco install rabbitmq
```

### Start the server as application

- Run command prompt as an administrator
- Navigate to ***sbin*** folder where RabbitMQ is installed and execute the commands below
- To execute commands without using fully qualified paths
  - Create a system environment variable (e.g. `RABBITMQ_SERVER`) for `"C:\Program Files\rabbitmq_3.13.3"`
  - Append the literal string `"%RABBITMQ_SERVER%\sbin"` to your system path (aka `%PATH%`).

```
rabbitmq-plugins enable rabbitmq_management
rabbitmq-server start -detached
```

### Stop RabbitMQ server on localhost

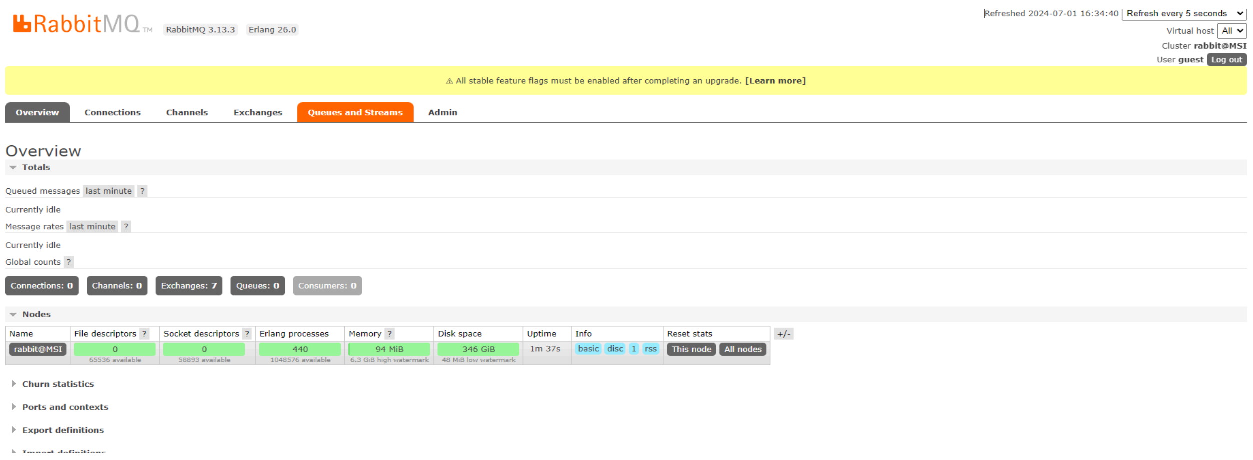
```
rabbitmqctl stop
```

## Management UI Access

- Provides an HTTP-based API for management and monitoring of RabbitMQ nodes and clusters

### Management UI Access

- Can be accessed using a Web browser at `http://localhost:15672/`


RabbitMQ 3.13.3 Erlang 26.0

Refreshed 2024-07-01 16:34:40 Refresh every 5 seconds
Virtual host All
Cluster **rabbit@MSI**
User **guest** Log out

⚠ All stable feature flags must be enabled after completing an upgrade. **[Learn more]**

Overview | Connections | Channels | Exchanges | Queues and Streams | Admin

Overview

**Totals**

Queued messages last minute ?
Currently idle
Message rates last minute ?
Currently idle
Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

**Nodes**

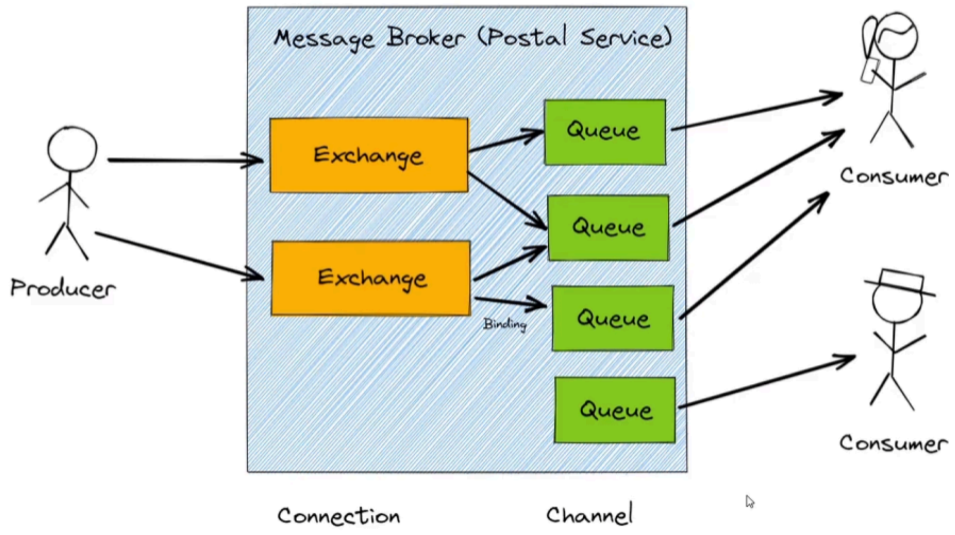
| Name | File descriptors ? | Socket descriptors ? | Erlang processes | Memory ? | Disk space | Uptime | Info | Reset stats |
|---|---|---|---|---|---|---|---|---|
| rabbit@MSI | 0<br>65536 available | 0<br>58893 available | 440<br>1048576 available | 94 MiB<br>6.3 GiB high watermark | 346 GiB<br>48 MiB low watermark | 1m 37s | basic disc 1 rss | This node All nodes |

**Churn statistics**
**Ports and contexts**
**Export definitions**

## Default User Access

- The broker creates a user guest with password guest
  - **By default**, these credentials can only be used when connecting to the broker as localhost

# Core Concepts



## Message Broker

- Accepts and forwards messages (e.g. post office)
  - binary blobs of data – *messages*
- RabbitMQ is a post box, a post office, and a letter carrier

## Producer

- A program that sends messages is a ***producer***.

## Consumer

- A ***consumer*** is a program that mostly waits to receive messages.

## Exchanges

- The *producer* never sends any messages directly to a queue. Instead, the *producer* can only send messages to an ***exchange***
- An ***exchange*** receives messages from *producers* and then pushes them to *queues*.
- The ***exchange*** must know exactly what to do with a message it receives.
    - **Four exchange types to choose from:**
        - **Direct** – the exchange forwards the message to a queue based on a routing key
        - **Fanout** – the exchange ignores the routing key and forwards the message to all bounded queues
        - **Topic –** the exchange routes the message to bounded queues using the match between a pattern defined on the exchange and the routing keys attached to the queues
        - **Headers** – the message header attributes are used, instead of the routing key, to bind an exchange to one or more queues
    - **Declare properties of exchange:**
        - **Name** – the name of the exchange
        - **Durability** – if enabled, the broker will not remove the exchange in case of a restart
        - **Auto-Delete –** when this option is enabled, the broker deletes the exchange if it is not bound to a queue
        - **Optional arguments**
- Therefore, the *exchange* decides if the message goes to one queue, to multiple queues, or is simply discarded.

## Queues

- name for the post box in RabbitMQ

- A ***queue*** is only bound by the host's memory & disk limits; it's essentially a large message buffer that delivers messages to *consumers* based on a **FIFO model.**
- Many *producers* can send messages that go to one ***queue***, and many *consumers* can try to receive data from one ***queue***
- We can define **several properties of the queue:**
    - **Name –** the name of the queue. If not defined, the broker will generate one
    - **Durability** – if enabled, the broker will not remove the queue in case of a restart
    - **Exclusive –** if enabled, the queue will only be used by one connection and will be removed when the connection is closed
    - **Auto-delete –** if enabled, the broker deletes the queue when the last consumer unsubscribes
    - **Optional arguments**

## Bindings

- The relationship between *exchange* and a *queue* is called a ***binding***.

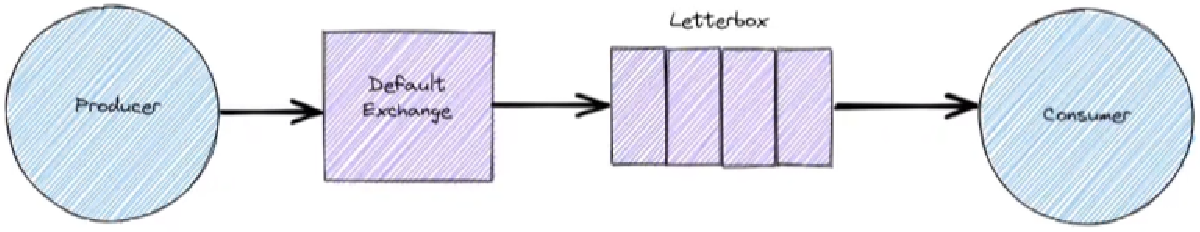
## Connections

- **The first step a client/producer must take to interact with a RabbitMQ broker is to establish a *Connection*.**
- This can be a regular TCP connection or an encrypted one using TLS.

## Channels

- **An AMQP *channel* is a mechanism that allows multiplexing multiple logic flows on top of a single connection.**
    - Allows better **resource usage** both on the client and server side since setting up a connection is a relatively expensive operation.
- A client creates one or more ***channels*** so it can send commands to the *broker*. This includes commands related to sending and/or receiving messages.
- **Note:** if we close a connection, all associated channels will also be closed.

## First Program - Python



(using the Pika Python client)

- **two small programs in Python:**
    - a producer (sender) that sends a single message
    - a consumer (receiver) that receives messages and prints them out.
- We will use **AMQP 0-9-1** as the protocol
    - open, general-purpose protocol for messaging

### Sending

- to establish a connection with RabbitMQ server locally

```
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

- before sending, must create a queue to which the message will be delivered

```
channel.queue_declare(queue='letterbox')
```

- In RabbitMQ a **message can never be sent directly to the queue**, must go through **exchange**
    - For the purpose of this tutorial, we will just use *default exchange*

```
message = "Hello world"
channel.basic_publish(exchange='', routing_key='letterbox', body=message)
# queue name needs to be specified in routing_key
```

### Receiving

- first we need to connect to RabbitMQ server

```
import pika
```

```
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

- make sure that the queue exists by using **.queue_declare()**
  - **.queue_declare()** is *idempotent* **–** can call as many times but only one queue will be created

```
channel.queue_declare(queue='letterbox')
```

- To receive message,
  - we **subscribe** callback to a *queue*

```
def callback(ch, method, properties, body):
    print(f"Received new message: {body}")
# this is called by the pika library whenever we receive a message
```

  - then, consume the message by telling RabbitMQ that this particular callback function should receive messages

```
channel.basic_consume(queue='letterbox',
                      auto_ack=True,
                      on_message_callback=callback)
```

  - Lastly consume the message

```
channel.start_consuming()
```

## Listing queues

- You may want to see how many queues RabbitMQ has. You can do so (as a **privileged user**) using:

```
rabbitmqctl.bat list_queues
```

## AMQP for RabbitMQ

- An open standard for passing business messages between applications or organizations.
- Uses a **remote procedure call pattern** (**RPC**) to allow one computer (ex: client) to execute programs or methods on another computer (ex: broker)
  - *Two-way communication*: both the broker and client can use RPC to run programs or call methods on each other
- When a **command** is sent to or from a *RabbitMQ broker*, all the data needed to

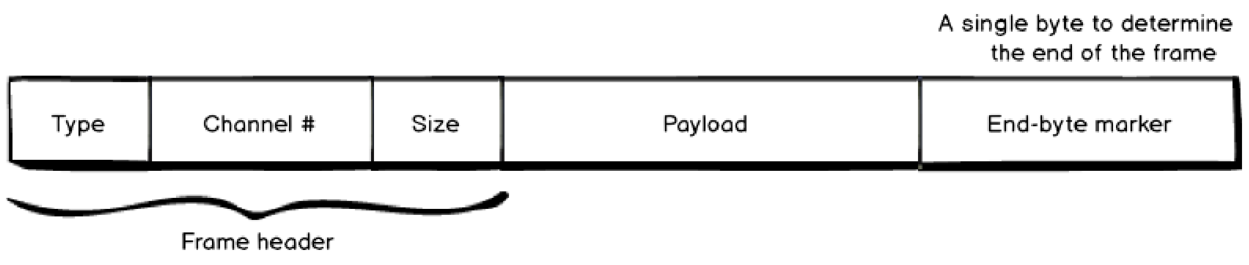
execute the command is included in a data structure called a **frame.**

## Key Features

- AMQP was designed with the following main characteristics as goals:
    - Security
    - Reliability
    - Interoperability
    - Standard
    - Open

## Frames

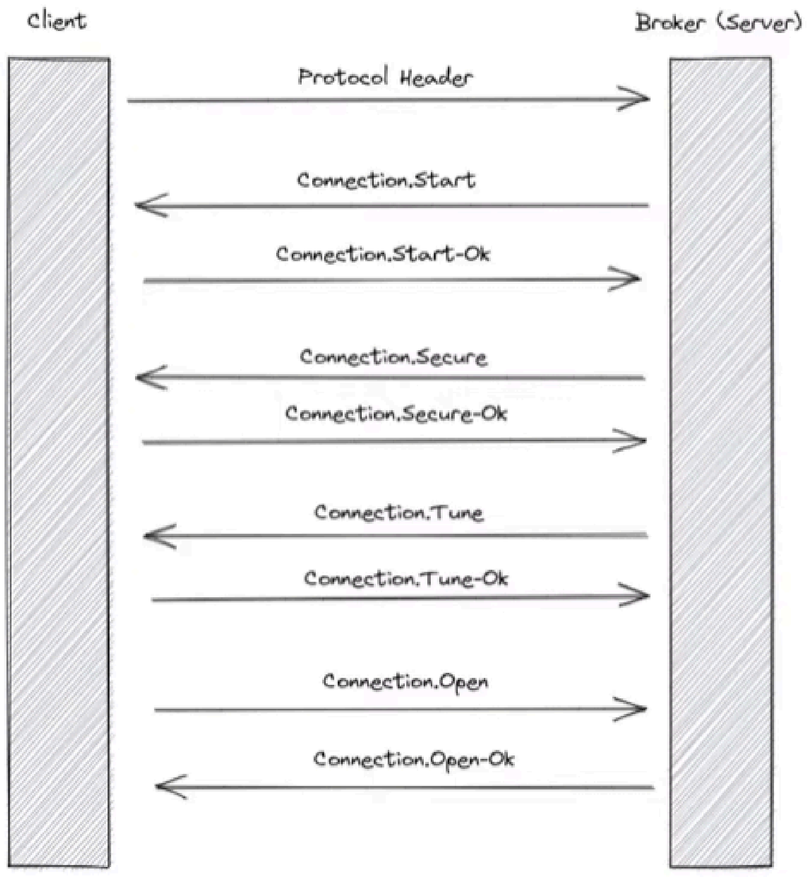
- A **frame** is a chunk of data that is used to send information from RabbitMQ to your application and vice-versa
- Every frame will have the same basic structure:



    - Five parts to a frame:
        - **Header:** first three
            - type of frame (0 - 1; 1 byte)
            - the channel the frame belongs to (1 - 3; 2 bytes)
            - size, in bytes (3 -7; 4 bytes)
        - **Payload:** varies accordingly with frame type (7 - size + 7)
        - **End-byte marker:** to determine end of frame (size + 8)
- **Types of frames**:
    - Protocol header: the frame sent to establish a new connection between the broker (RabbitMQ) and a client
    - Method frame: Carries a RPC request or response.
        - Ex: when we are publishing a message, our application calls Basic.Publish, and this message is carried in a method frame that will tell RabbitMQ that a client is going to publish a message.

    - Content header: Certain specific methods carry a content (like Basic.Publish, for instance, that carries a message to be published). The *content header frame* is used to send the properties of this content.
    - Body: the frame with the actual content of your message, and can be split into multiple different frames if the message is too big
        - **Default size:** 131KB
    - Heartbeat: Used to confirm that a given client is still alive. If no response in a timely fashion, the client will be disconnected (considered dead).

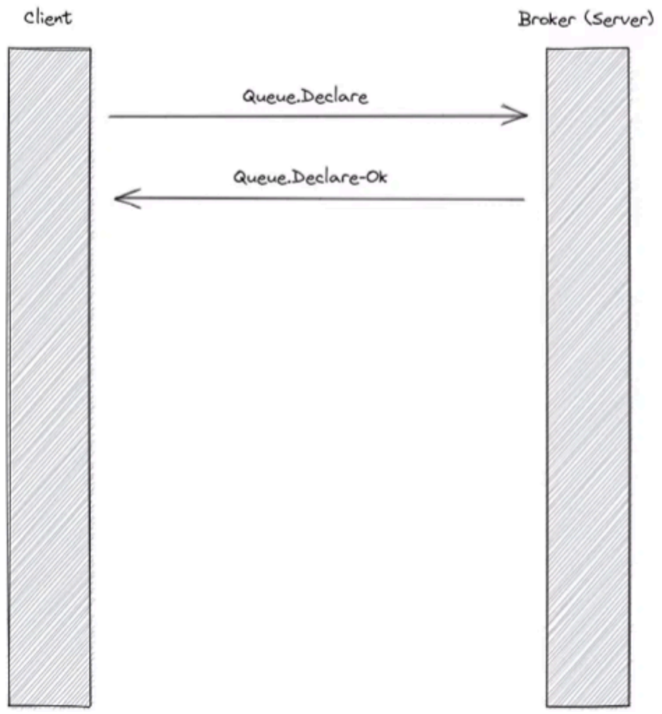
## Establishing a Connection



- **Protocol header:** this header specifies **protocol and version** being used (e.g. AMQP 0-9-1)
- **Connection.Start:** this method includes parameters such as the server properties, supported authentication mechanisms, and supported locales. (**connection parameters + authentication mechanism and locale**)

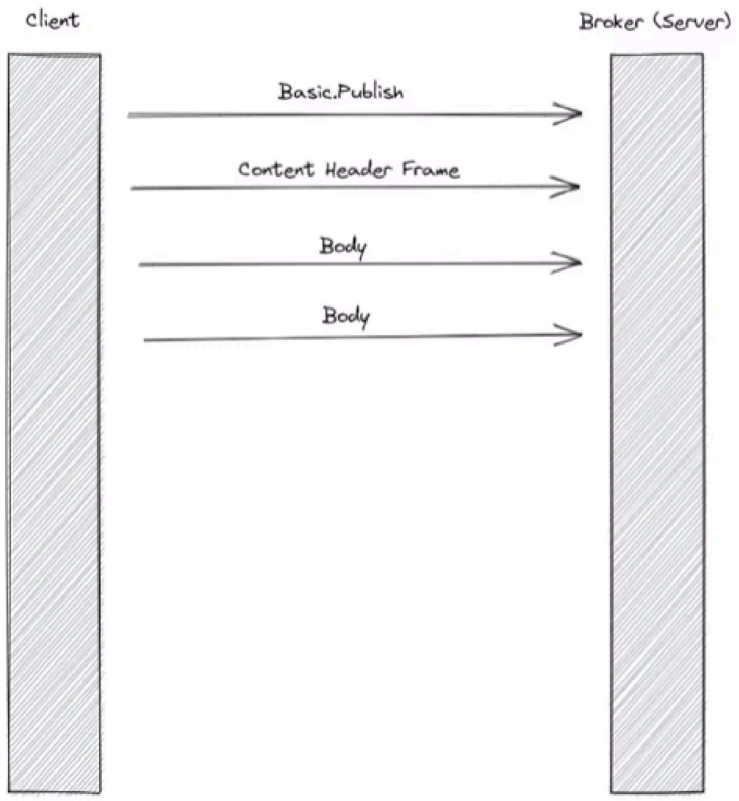
- **Connection.Secure:** a secure connection negotiation, which could request security- related parameters from client (**security settings**)
- **Connection.Tune:** this method includes tuning parameters such as channel frame, frame max, heartbeat interval, etc. (**connection configuration**)
- **Connection.Open:** indicates connection setup is now complete (**connection open**)
- **Connection._-Ok:** **acknowledgments** indicating that the client has received and accepted the broker's requests

## Declaring a Queue



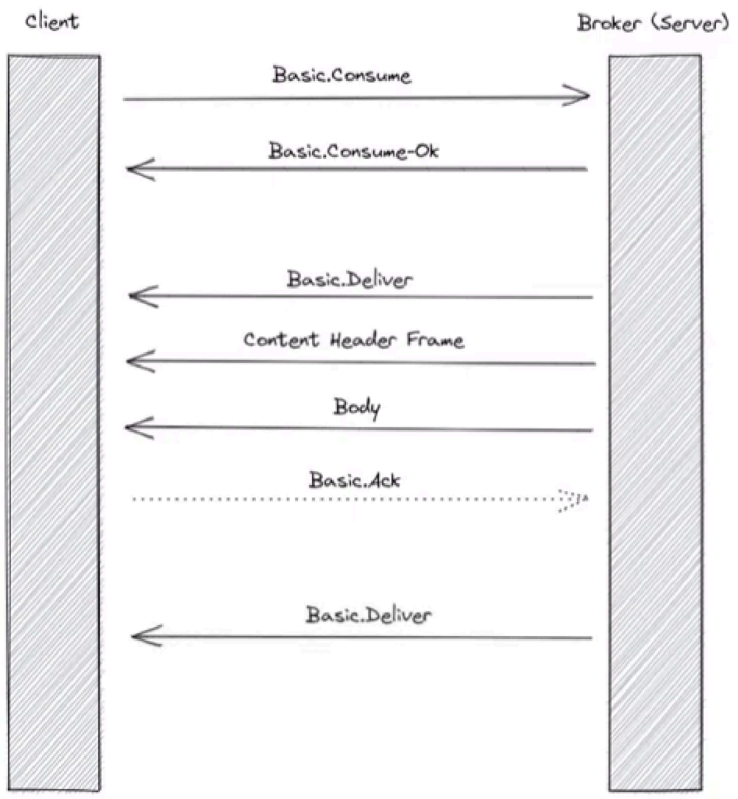
- **Queue.Declare:** client sends this method to the broker to declare (create) a queue. It **specifies the queue's properties** such as its *name*, *durability*, *exclusivity*, and *auto-delete behavior*.
- **Queue.Declare-Ok:** confirm that the queue has been successfully declared. It **includes details about the queue**, such as its *name*, *message count*, and *consumer count*.

## Publishing a Message



- **Basic.Publish:** initiate the publishing of a message. It **specifies the target exchange and the routing key** for the message.
- **Content Header Frame:** provide **metadata about the message**, such as its *size*, *content type*, and *delivery mode*.
- **Body:** contains the **actual message payload**. Can be multiple frames (if too big)

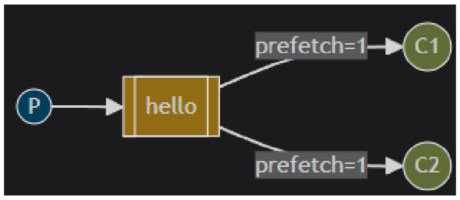
## Consuming a Message



- **Basic.Consume:** tells the broker to **start consuming messages from a specified queue**; includes parameters such as the *queue name* and *consumer tag*.
- **Basic.Consume-Ok:** confirms that the broker is ready to deliver messages
- **Basic.Deliver:** Used to deliver a message. It includes **metadata** such as the *delivery tag*, *exchange*, and *routing key*.
- **Content Header Frame:** provide **metadata about the message**, such as its *size*, *content type*, and *delivery mode*.
- **Body:** contains the **actual message payload**. Can be multiple frames (if too big)
- **Basic.Ack:** client **acknowledges it has received and processed message**. This then allows broker to remove message from queue.

# Messaging Patterns

## Competing Consumers



- The **computing consumers** or **work queue** pattern is used to distribute messages to multiple workers. **Each task is delivered to exactly one worker.**
- **Main idea:**
  - Avoid doing a resource-intensive task (ex: processing an image) immediately in which we have to wait for it to complete. Instead, we schedule the task to be done later.
  - Encapsulate a task as a message and send it to the queue. A worker process running in the background will pop the tasks and eventually execute the job.
- This concept is especially **useful in web applications** where it's impossible to handle a complex task during a short HTTP request window.

### Round-robin dispatching

- One of the advantages of using a **Work Queue** is the ability to easily **parallelise work.** If we are building up a backlog of work, we can just **add more workers** and that way, scale easily (Scalability ↑ Reliability ↑).
- **By default**, RabbitMQ will send each message to the next consumer, in sequence

### Message Acknowledgement

- An **ack(nowledgement)** is sent back by the consumer to tell RabbitMQ that a particular message had been received, processed and that RabbitMQ is free to delete it.
- If a *consumer dies* (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, **RabbitMQ will re-queue it** and deliver it to another worker, if available.
- **Manual message acknowledgments** are **turned on by default.**

```
ch.basic_ack(delivery_tag = method.delivery_tag)
```

### Message Durability

- When **RabbitMQ quits or crashes** it will forget the queues and messages
- To make sure that *messages aren't lost*: we need to **mark both the queue and messages as durable.**
    - **Note:** RabbitMQ doesn't allow you to redefine an existing queue with different parameters. So you must make a new queue if needed.

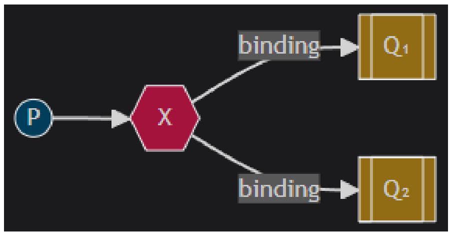
```
# queue
channel.queue_declare(queue='task_queue', durable=True)

# message
channel.basic_publish(exchange='',
                      routing_key="task_queue",
                      body=message,
                      properties=pika.BasicProperties(
                         delivery_mode = pika.DeliveryMode.Persistent
                      ))
```

### Fair Dispatch

- In situations where **one worker consumes messages faster** than another (*unequal distribution due to round-robin dispatch*).
- **Solution:** set prefetch count with the value of 1 (tells RabbitMQ to not give more than 1 message at a time to a worker)

```
 channel.basic_qos(prefetch_count=1)
```

## Publish/Subscribe



- **Main idea:**
    - Deliver messages to multiple consumers (*multiple consumers receive the*

*same message*).
        - **Note:** The message is not saved multiple times in memory, rather, the binded queues just all have the same reference to that message in memory
    - The producer sends messages directly to the **exchange**, where it follows its rules for distributing the messages.

## Exchanges

- Uses the **exchange type fanout**

```
channel.exchange_declare(exchange='logs',
                         exchange_type='fanout')

channel.basic_publish(exchange='logs',
                      routing_key='',
                      body=message)
# the ‘exchange’ parameter is the name of the exchange
# no routing_key to send messages to all queues
```

## Temporary Queue

- Since, we want to hear all messages, we need:
    - a **fresh empty queue** everytime we connect to **RabbitMQ**

```
result = channel.queue_declare(queue='')
# queue='' => server chooses a random queue name
```

- If the **consumer connection is closed**, the **queue** should be **deleted**

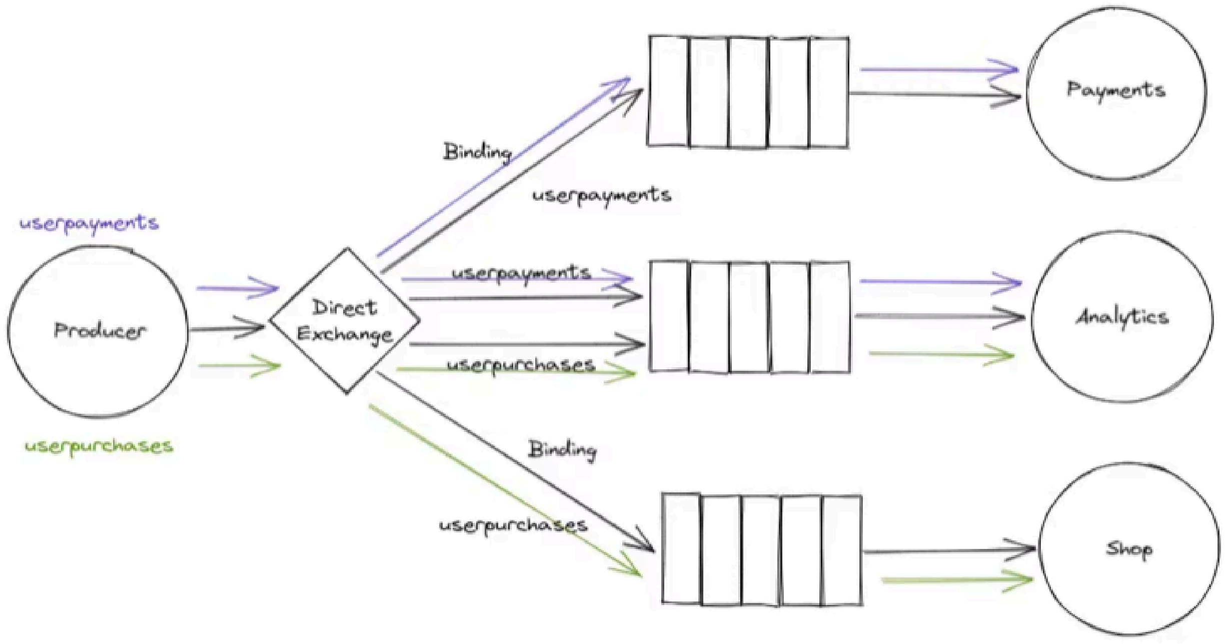
```
result = channel.queue_declare(queue='', exclusive=True)
```

## Bindings

- The relationship between exchange and a queue is called a **binding**

```
channel.queue_bind(exchange='logs',
                   queue=result.method.queue)
```

## Routing



- **Main idea:**
    - Similar to **pub/sub** but only subscribe to a **subset** of the messages
- **Advantage:**
    - Multiple **queues** can be bound to the **direct exchange** using the same **binding keys**
    - A **single** **queue** can also have **multiple bindings**
- **Limitation:** can't do routing on multiple criterias
    - **Solution: Topic Exchange**

### Direct Exchange

- Uses the **exchange type direct** and a **routing_key**
- A **message** goes to the **queues** whose **binding key** exactly matches the **routing key** of the message

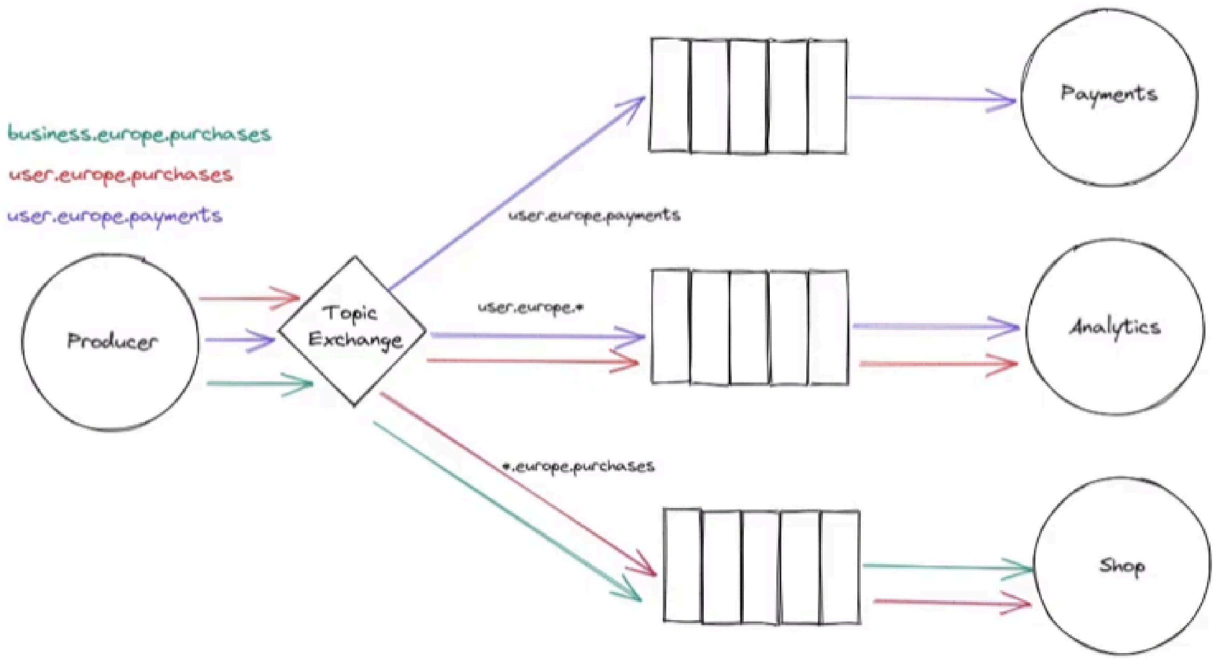
```
channel.exchange_declare(exchange='direct_logs',
                         exchange_type='direct')

channel.basic_publish(exchange='direct_logs',
                      routing_key='black',
                      body=message)
```

## Bindings

- Must create a **binding** with a **key**

```
channel.queue_bind(exchange=exchange_name,
                   queue=queue_name,
                   routing_key='black')
```

## Topics



- **Main Idea:**
  - A **message** sent with a *particular* **routing key** will be delivered to all the queues that are bound with a *matching* **binding key**
- Messages sent to a **topic** can't have an *arbitrary* **routing key** – must be a list of words, delimited by dots.
  - **Words:** can be anything – usually specify features connected to message
    - Ex: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit"
    - **Limit:** 255 bytes
- **Binding key** must also be in the same form (list of words, delimited by dots). However, there are two special cases:
  - *** (star)** can substitute for exactly one word.
  - **# (hash)** can substitute for zero or more words.

## Topic Exchange

- Uses the **exchange type topic** and a **routing_key**
- A **message** goes to the **queues** whose **binding key** exactly matches the **routing key** of the message
    - *** (star)** can substitute for exactly one word.
    - **# (hash)** can substitute for zero or more words.

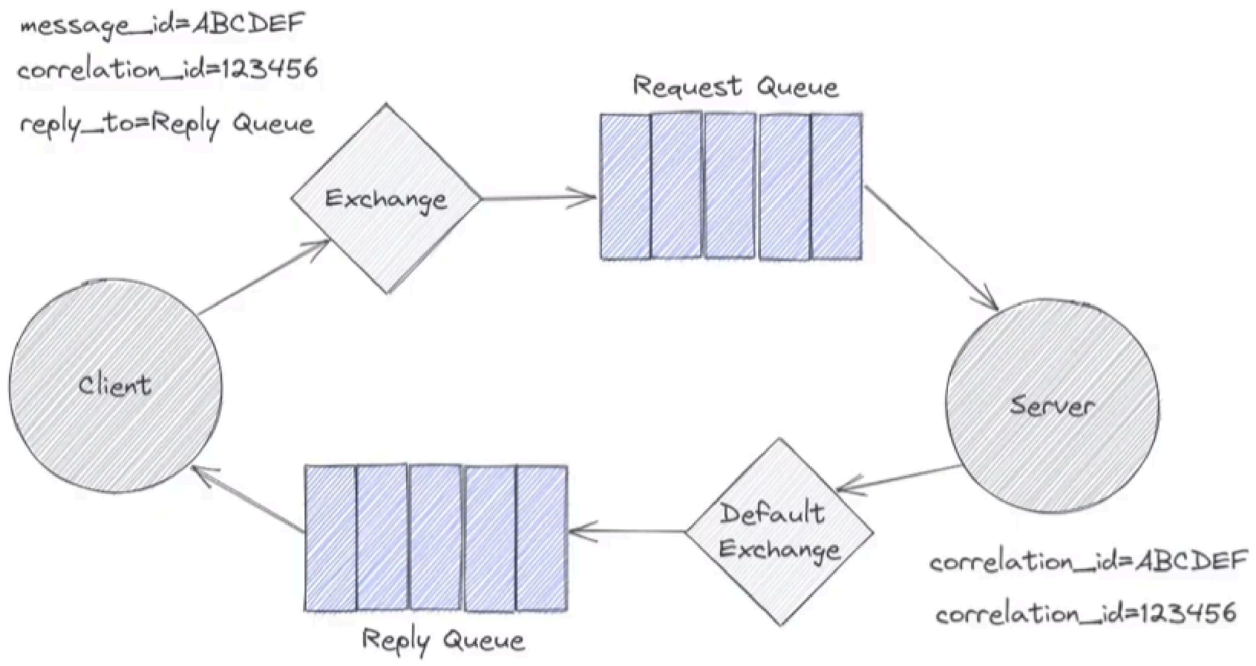
```
channel.exchange_declare(exchange='topic_logs',
                         exchange_type='topic')

# <celerity>.<color>.<species>
channel.basic_publish(exchange='topic_logs',
                      routing_key='lazy.pink.rabbit',
                      body=message)

channel.basic_publish(exchange='topic_logs',
                      routing_key='quick.orange.rabbit',
                      body=message)
```

## Bindings

- Must create a **binding** with a **key**

```
# interested in all black animals
channel.queue_bind(exchange=exchange_name,
                   queue=queue_name,
                   routing_key='*.black.*')

# interested in all quick animals
channel.queue_bind(exchange=exchange_name,
                   queue=queue_name,
                   routing_key='quick.#')
```

## Request Reply Protocol



- So far, our patterns focused on *one-way communication*. Sometimes, we may want *two-way conversation*. This brings us to the **Request Reply Protocol**.
- **Request-Reply** has two participants:
    - **Requestor** – Sends a request message and waits for a reply message.
        - The request is either broadcast to all interested parties (*pub/sub*) or processed by a single consumer (*direct*)
        - 
    - **Replier** – Receives the request message and responds with a reply message.
        - Almost always point to point (*direct*)

### Receiving a Reply

- The **requestor** has two approaches to receive a reply:
    - **Synchronous Block**
        - *Mechanism:* A single thread sends a request, blocks to wait for the reply message, then processes the reply.
        - *Pros:* Simple to implement.
        - *Cons:* Difficult to recover if the requestor crashes. Only one outstanding request per thread, as the reply channel is private.
    - **Asynchronous Block**

        - *Mechanism:* One thread sends a request and sets up a callback for the reply. A separate thread listens for replies and invokes the appropriate callback.
        - *Pros:* Allows multiple outstanding requests to have a shared reply channel. Easier to recover if the requestor crashes by restarting the reply thread.
        - *Cons:* More complex due to the need to re-establish the caller's context in the callback.

## Remote Procedure Call (RPC)

- **RPC** is a method in which a computer program executes a procedure in another address space (a remote computer) while waiting for the result.
- Can be a powerful tool in setting up *distributed systems*.
- By using **RabbitMQ** for **RPC**, you can ensure that your systems are *loosely coupled*, *scalable*, and *resilient to failures*.
- To build an **RPC** we need a *client* and a *scalable RPC server*.

### Client Interface

- Exposes a method 'call' which sends an RPC request and blocks until the answer is received

```
fibonacci_rpc = FibonacciRpcClient()
result = fibonacci_rpc.call(4)
print(f"fib(4) is {result}")
```

#### Note on RPC

- Consider the following advice:
    - Make sure it's obvious which function call is local and which is remote.
    - Document your system. Make the dependencies between components clear.
    - Handle error cases. How should the client react when the RPC server is down for a long time?
- When in doubt avoid RPC. If you can, you should use an *asynchronous pipeline* - instead of RPC-like blocking

## Callback Queue

- A **client** sends a *request message* and a **server** replies with a response *message*.
- In order to receive a response the **client** needs to send a **'callback' queue** address with the request.

```
result = channel.queue_declare(queue='', exclusive=True)
callback_queue = result.method.queue

channel.basic_publish(exchange='',
                      routing_key='rpc_queue',
                      properties=pika.BasicProperties(
                            reply_to = callback_queue,
                            ),
                      body=request)

# ... and some code to read a response message from the
callback_queue ...
```

### Message Properties

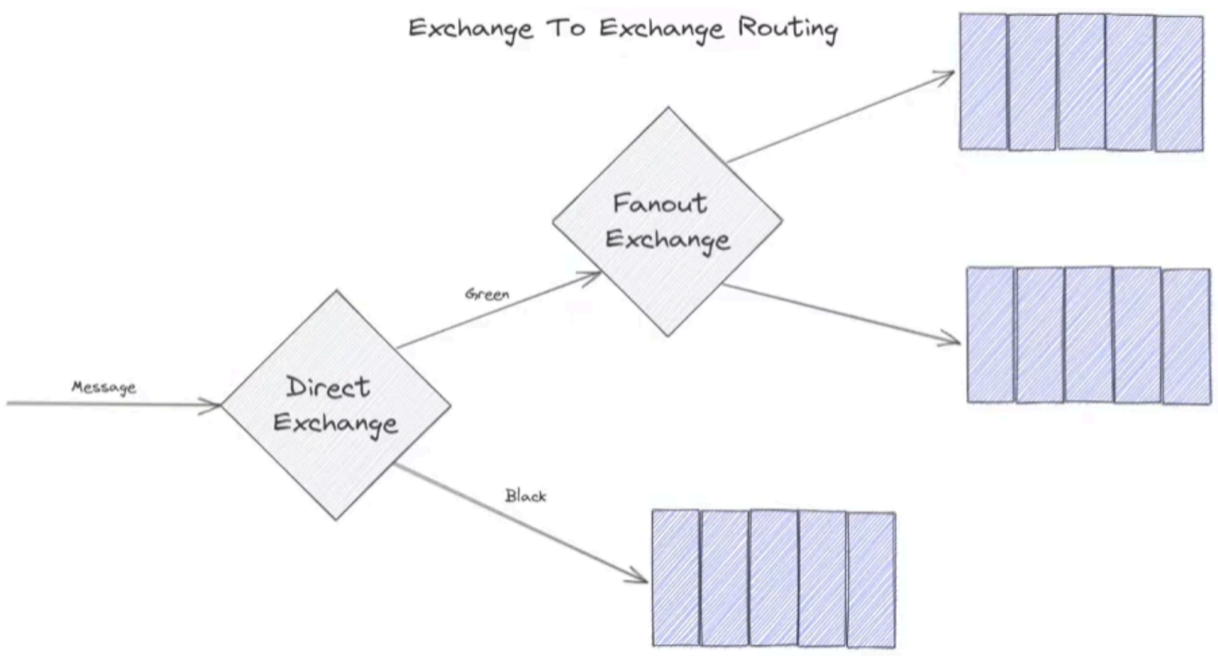
- **delivery_mode:** Marks a message as persistent (with a value of 2) or transient (any other value).
- **content_type:** Used to describe the mime-type of the encoding. Ex: application/json.
- **reply_to:** Commonly used to name a callback queue.
- **correlation_id:** Useful to correlate RPC responses with requests.

## Correlation Id

- It's better to create a *single callback queue* per **client.**
- How to then identify which *request* the *response* belongs to? **correlation_id** (unique)
    - Unknown *correlation_id messages* will be discarded
        - Why ignore rather than fail? **race condition** can arise with server

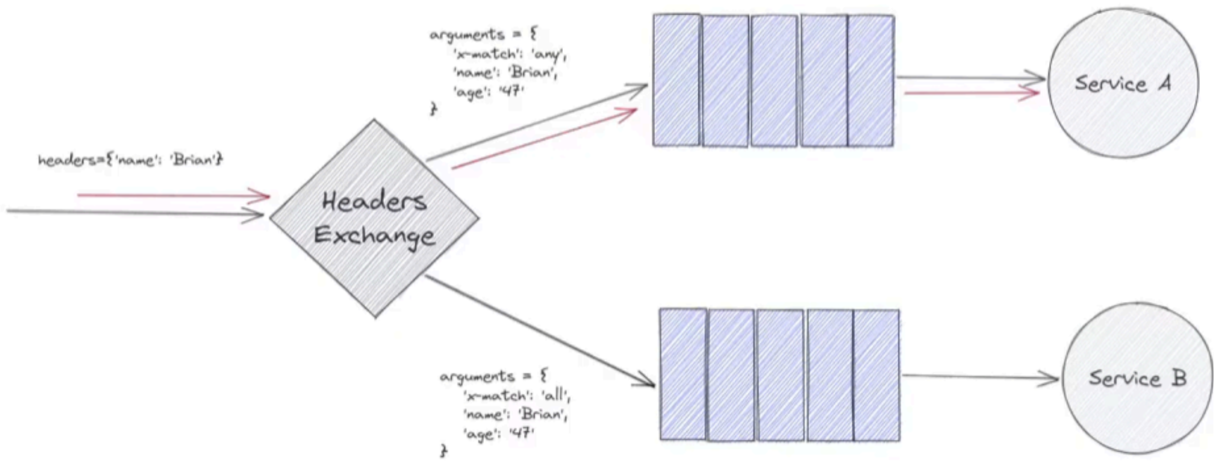
## Other Exchanges

### Exchange to Exchange Routing



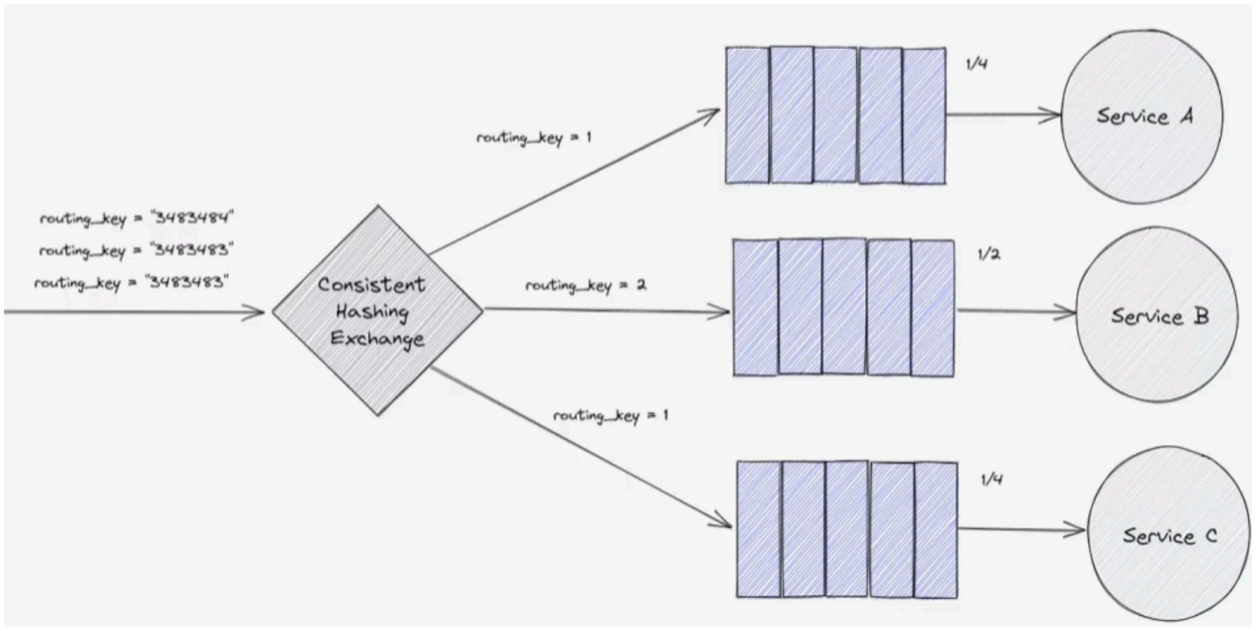
- **Exchanges** cannot only be bound to **queues**, but they can also be bound to other **exchanges**.
    - Flexibility ↑

## Headers Exchange


headers={'name': 'Brian'}
Headers Exchange
arguments = { 'x-match': 'any', 'name': 'Brian', 'age': '47' }
arguments = { 'x-match': 'all', 'name': 'Brian', 'age': '47' }
Service A
Service B

- Uses the contents of a **headers table** to determine where to route the message rather than the *routing key*.

## Consistent Hashing Exchange



- A plugin that allows us to **possibly equally distribute** messages between the bound queues.

### Weights

- When a queue is bound to a **Consistent Hash exchange**, the binding key is a **number** which indicates the binding weight (*allocated hash space*)
    - **1** ensures reasonably even balancing (uniform distribution)
- **Note:** be careful when adding additional bindings once the original bindings have been added

### Routing

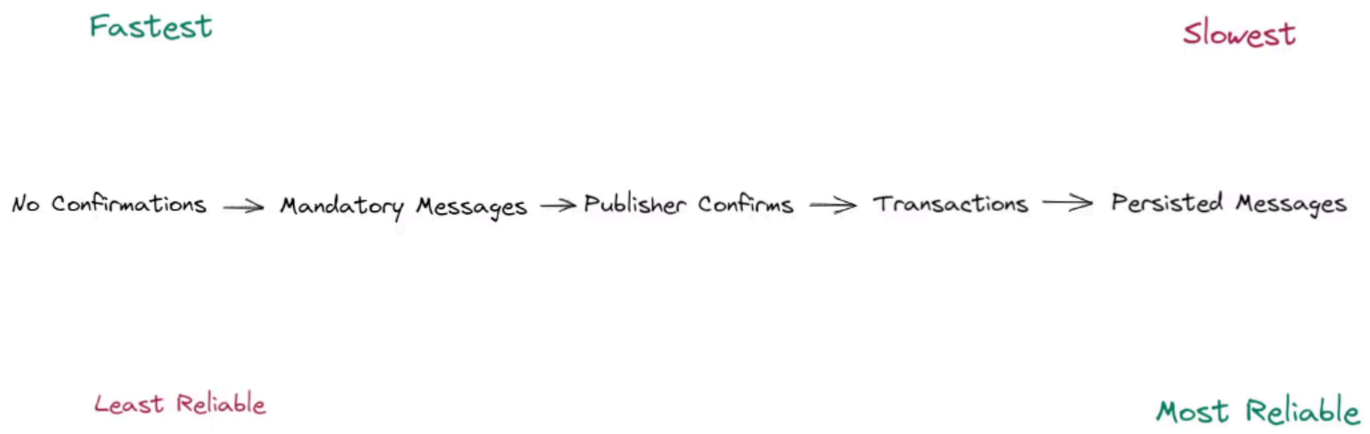
- **Hashing** distributes *routing keys* among **queues**, not message payloads among queues; all messages with the *same routing key/hash* will go the *same queue*
- In other words, queues with a higher binding weight or hash space, will have more buckets for which a *hash partition* can fall into.
- Each message gets delivered to **at most one queue**. On average, a message gets delivered to exactly one queue.

# Publishing Options

## Basic Properties

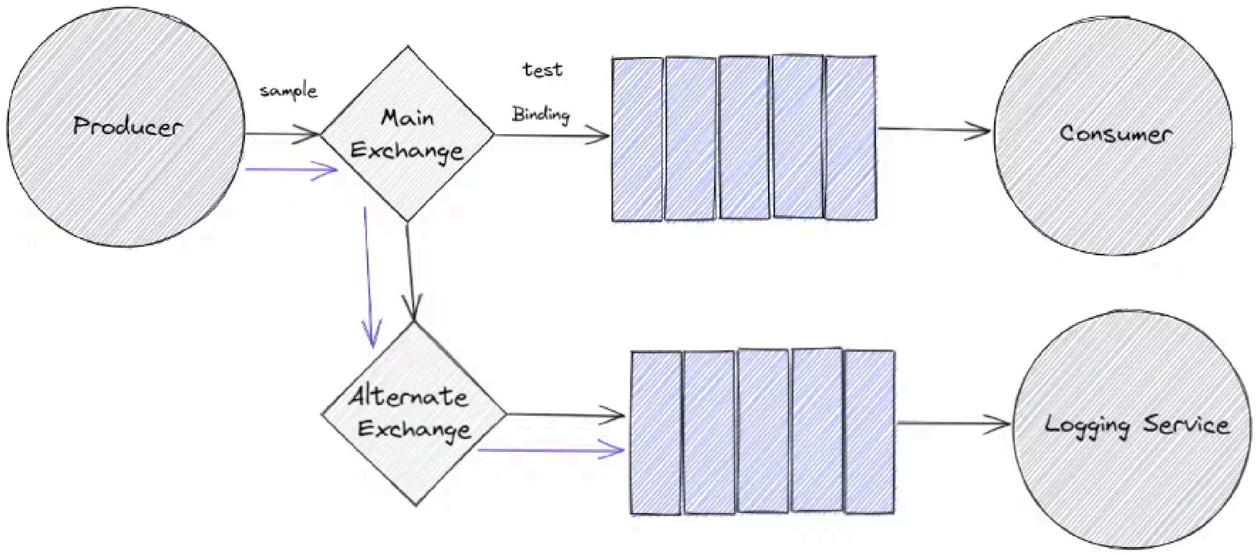
- **contentType**: MIME type of the message (e.g., `text/plain`, `application/json`, `application/pdf`).
- **contentEncoding**: Encoding type (e.g., `gzip`, `UTF-8`, `deflate`, `compress`).
- **headers**: Custom metadata as key-value pairs.
- **deliveryMode**: Message persistence level (`0` for non-persistent, `1` for persistent).
- **priority**: Message priority (higher value = higher priority).
- **correlationId**: ID for correlating RPC requests and responses.
- **replyTo**: Queue name for RPC responses.
- **expiration**: Message TTL (time-to-live).
- **messageId**: Unique identifier for the message.
- **timestamp**: Time when the message was created.
- **type**: Message type (e.g., `order`, `invoice`).
- **userId**: ID of the user sending the message (must be the same userId as loggedIn user).
- **appId**: ID of the application sending the message.
- **clusterId**: ID of the RabbitMQ cluster.

### Speed vs Resiliency Trade Offs


Fastest
Slowest
No Confirmations → Mandatory Messages → Publisher Confirms → Transactions → Persisted Messages
Least Reliable
Most Reliable

## Exchange Options

### Alternate Exchanges

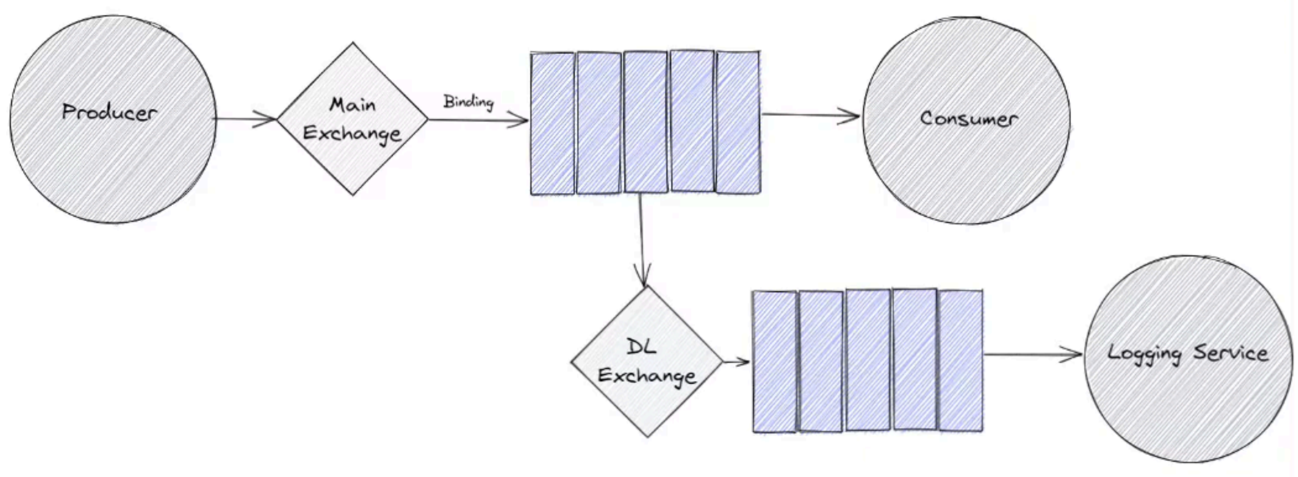


- **Purpose**: Handle messages that an exchange cannot route (no bound queues or no matching bindings).
- **Exchange type:** can be any, but fanout is most common
- **Use Cases**:
    - Detecting when clients publish unroutable messages.
    - Implementing "or else" routing where unroutable messages are handled by a generic handler.

### How to Define an Alternate Exchange

- **Via Policies**: Recommended method as it simplifies changes (e.g., during upgrades).
    - **Note**: The default exchange does not support alternate exchanges as it is a special-case in the code, not a real exchange.
- **Via Exchange Arguments**: Optional method at exchange declaration time.
    - If both policy and arguments specify an AE, the argument-specified AE takes precedence.

## Dead Letter Exchanges



- **Purpose**: Republish messages from a queue to an exchange when certain events occur.
- **Exchange Type:** can be of the usual exchange types and are declared as normal

### Events Triggering Dead-Lettering:

- Message negatively acknowledged (`basic.reject` or `basic.nack` with `requeue=false`).
- Message expires due to per-message TTL.
- Queue exceeds its length limit.
- Message returned more times to a quorum queue than the delivery limit.
- **Note:** If an entire queue expires, its messages are not dead-lettered.

### How to Define a DLX

- **Via Queue Arguments**: Specified at queue declaration time.
- **Via Policies**: Recommended for easier reconfiguration without redeployment.
    - If both policy and arguments specify a DLX, the argument-specified DLX takes precedence.

# Message Acknowledgements

## basic.ack (Acknowledgment)

- Confirms successful processing of one (`multiple=false`) or more messages (`multiple=true`).
- Removes message(s) from the queue.
- Acknowledges messages up to a specified delivery tag.

## basic.nack (Negative Acknowledgment)

- Indicates failure to process one (`multiple=false`) or more messages (`multiple=true`).
- Allows requeueing (`requeue=true`) or discarding (`requeue=false`) messages.
- Applies to messages up to a specified delivery tag.

## basic.reject

- Rejects individual messages.
- Allows requeueing (`requeue=true`) or discarding (`requeue=false`) of each message.
- Operates on messages identified by their delivery tag.

**Delivery Tag:** Unique identifier for each message, used by acknowledgment methods to specify ranges of messages to acknowledge or reject.

## Queue Options

### Durable

- Determines if the queue survives broker restarts.
- **Usage:** `durable=True` ensures the queue persists messages to disk.

### Exclusive

- Restricts queue usage to the connection that declared it.
- **Usage:** `exclusive=True` makes the queue accessible only to the declaring connection.

### Auto-Delete

- Automatically deletes the queue when no longer in use.
- **Usage:** `auto_delete=True` removes the queue when all consumers have finished using it.

### Arguments

- Additional properties defined by key-value pairs.
- **Usage:** Customizes queue behavior, e.g., setting message TTL or maximum length.

### Maximum Length

- Limits the number of messages a queue holds.
- **Usage:** `max_length=N` restricts the queue to N messages.

## Message TTL (Time-to-Live)

- Defines how long messages remain in the queue before expiration.
- **Usage:** `x-mxessage_ttl=milliseconds` removes messages older than the specified time.

## Auto-Expire

- Automatically deletes the queue when no longer used or expired.
- **Usage:** `expires=milliseconds` removes the queue if unused or expired.

# Troubleshooting

## Cannot Delete epmd.exe after Uninstalling RabbitMQ and Erlang

This error happens when you uninstall RabbitMQ (and optionally Erlang) but cannot delete one or more programs or folders associated with RabbitMQ or Erlang.

One common example would be "epmd.exe". This occurs because these files are still being held open by an active process and can present a problem. To fix this:

- Open the command prompt as Administrator and run the tasklist command.
- Find the epmd.exe process (or whichever process cannot be deleted) and note the process ID by running the taskkill /pid {PROCESSID} /F command.

```
taskkill /pid 1234 /f
```

- Delete the file or folder.

## References

- https://www.youtube.com/playlist?list=PLalrWAGybpB-UHbRDhFsBgXJM1g6T4IvO
- https://www.rabbitmq.com/docs
- https://www.brianstorti.com/speaking-rabbit-amqps-frame-structure/
- https://www.baeldung.com/java-rabbitmq-channels-connections
- https://github.com/rabbitmq/rabbitmq-server/blob/main/deps/rabbitmq_consistent_hash_exchange/README.md
- https://www.cogin.com/articles/rabbitmq/rabbitmq-exchanges-guide.php
- https://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html