aio-pika Documentation

Release 0.9.0

Dmitry Orlov

Apr 09, 2017

RabbitMQ tutorial adopted for aio-pika

1 Tutorial **3**

1.1 Introduction . . . 3
1.1.1 Hello World! . . . 4
1.1.2 Sending . . . 5
1.1.3 Receiving . . . 6
1.1.4 Putting it all together . . . 7
1.2 Work Queues . . . 9
1.2.1 Preparation . . . 10
1.2.2 Round-robin dispatching . . . 10
1.2.3 Message acknowledgment . . . 11
1.2.4 Message durability . . . 12
1.2.5 Fair dispatch . . . 13
1.2.6 Putting it all together . . . 14
1.3 Publish/Subscribe . . . 15
1.3.1 Exchanges . . . 16
1.3.2 Temporary queues . . . 18
1.3.3 Bindings . . . 18
1.3.4 Putting it all together . . . 19
1.4 Routing . . . 21
1.4.1 Bindings . . . 21
1.4.2 Direct exchange . . . 22
1.4.3 Multiple bindings . . . 23
1.4.4 Emitting logs . . . 23
1.4.5 Subscribing . . . 23

1.4.6 Putting it all together . . . 24
1.5 Topics . . . 26
1.5.1 Topic exchange . . . 26
1.5.2 Putting it all together . . . 28
1.6 Remote procedure call (RPC) . . . 30
1.6.1 Client interface . . . 30
1.6.2 Callback queue . . . 31
1.6.3 Correlation id . . . 31
1.6.4 Summary . . . 32
1.6.5 Putting it all together . . . 32
1.7 aio_pika package . . . 35

2 Installation **43**

3 Usage example **45**

4 Thanks for contributing **47**

Python Module Index **49**

aio_pika it's a wrapper for the PIKA for asyncio and humans.

CHAPTER 1

Tutorial

Introduction

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

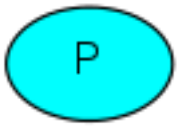
RabbitMQ is a message broker. The principal idea is pretty simple: it accepts and forwards messages. You can think about it as a post office: when you send mail to the post box you're pretty sure that Mr. Postman will eventually deliver the mail to your recipient. Using this metaphor RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is the fact that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data messages.

RabbitMQ, and messaging in general, uses some jargon.

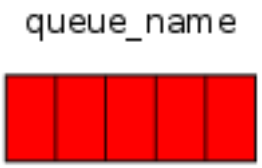
- Producing means nothing more than sending. A program that sends messages is a producer.

We'll draw it like that, with "P":



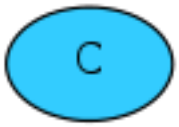
- A queue is the name for a mailbox. It lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can be stored only inside a queue. A queue is not bound by any limits, it can store as many messages as you like it's essentially an infinite buffer. Many producers can send messages that go to one queue, many consumers can try to receive data from one queue.

A queue will be drawn as like that, with its name above it:



- Consuming has a similar meaning to receiving. A consumer is a program that mostly waits to receive messages.

On our drawings it's shown with "C":



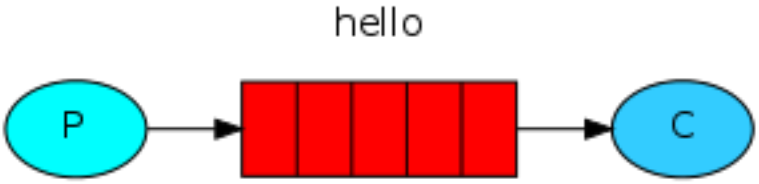
Note: Note that the producer, consumer, and broker do not have to reside on the same machine; indeed in most applications they don't.

Hello World!

Note: Using the aio-pika async Python client

Our "Hello world" won't be too complex let's send a message, receive it and print it on the screen. To do so we need two programs: one that sends a message and one that receives and prints it.

Our overall design will look like:



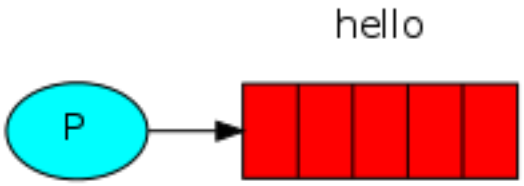
Producer sends messages to the "hello" queue. The consumer receives messages from that queue.

Note: RabbitMQ libraries

RabbitMQ speaks AMQP 0.9.1, which is an open, general-purpose protocol for messaging. There are a number of clients for RabbitMQ in many different languages. In this tutorial series we're going to use aio-pika, which is the

Python client recommended by the RabbitMQ team. To install it you can use the pip package management tool.

Sending



Our first program *send.py* will send a single message to the queue. The first thing we need to do is to establish a connection with RabbitMQ server.

```
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

We're connected now, to a broker on the local machine - hence the localhost. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.

Next, before sending we need to make sure the recipient queue exists. If we send a message to non-existing location, RabbitMQ will just trash the message. Let's create a queue to which the message will be delivered, let's name it *hello*:

```
async def main(loop):
    ...

    # Declaring queue
    queue = await channel.declare_queue('hello')
```

At that point we're ready to send a message. Our first message will just contain a string Hello World! and we want to send it to our hello queue.

In RabbitMQ a message can never be sent directly to the queue, it always needs to go through an exchange. But let's not get dragged down by the details you can read more about exchanges in the *third part of this tutorial*. All we need to know now is how to use a default exchange identified by an empty string. This exchange is special it allows us to specify exactly to which queue the message should go. The queue name needs to be specified in the *routing_key* parameter:

```
async def main(loop):
    ...

    await channel.default_exchange.publish(
```

```
        Message(b'Hello World!'),
        routing_key='hello',
    )
    print(" [x] Sent 'Hello World!'")
```

Before exiting the program we need to make sure the network buffers were flushed and our message was actually delivered to RabbitMQ. We can do it by gently closing the connection.

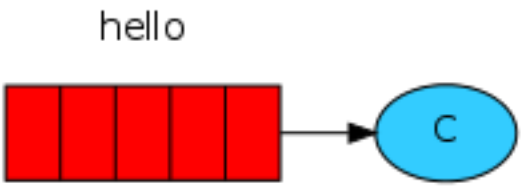
```
async def main(loop):
    ...

    await connection.close()
```

Note: *Sending doesn't work!*

If this is your first time using RabbitMQ and you don't see the "Sent" message then you may be left scratching your head wondering what could be wrong. Maybe the broker was started without enough free disk space (by default it needs at least 1Gb free) and is therefore refusing to accept messages. Check the broker logfile to confirm and reduce the limit if necessary. The configuration file documentation will show you how to set *disk_free_limit*.

Receiving



Our second program *receive.py* will receive messages from the queue and print them on the screen.

Again, first we need to connect to RabbitMQ server. The code responsible for connecting to Rabbit is the same as previously.

The next step, just like before, is to make sure that the queue exists. Creating a queue using *queue_declare* is idempotent we can run the command as many times as we like, and only one will be created.

```
async def main(loop):
    ...

    # Declaring queue
    queue = await channel.declare_queue('hello')
```

You may ask why we declare the queue again we have already declared it in our previous code. We could avoid that if we were sure that the queue already exists. For example if *send.py* program was run before. But we're not yet sure which program to run first. In such cases it's a good practice to repeat declaring the queue in both programs.

Note: Listing queues

You may wish to see what queues RabbitMQ has and how many messages are in them. You can do it (as a privileged user) using the rabbitmqctl tool:

```
$ sudo rabbitmqctl list_queues
Listing queues ...
hello    0
...done.
(omit sudo on Windows)
```

Receiving messages from the queue is simple. It works by subscribing a *callback function* to a queue or using *simple get*.

Whenever we receive a message, this callback function is called by the aio-pika library. In our case this function will print on the screen the contents of the message.

```
import asyncio
from aio_pika import IncomingMessage

def on_message(message: IncomingMessage):
    print(" [x] Received message %r" % message)
    print("     Message body is: %r" % message.body)
```

Next, we need to tell RabbitMQ that this particular callback function should receive messages from our hello queue:

```
import asyncio
from aio_pika import connect, IncomingMessage

def on_message(message: IncomingMessage):
    print(" [x] Received message %r" % message)
    print("     Message body is: %r" % message.body)

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    # Declaring queue
    queue = await channel.declare_queue('hello')

    # Start listening the queue with name 'hello'
    await queue.consume(on_message, no_ack=True)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main(loop))

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    loop.run_forever()
```

The *no_ack* parameter will be described *later on*.

Putting it all together

Full code for *send.py*:

```
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    # Sending the message
    await channel.default_exchange.publish(
        Message(b'Hello World!')
        routing_key='hello',
    )

    print(" [x] Sent 'Hello World!'")

    await connection.close()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Full *receive.py* code:

```
import asyncio
from aio_pika import connect, IncomingMessage


def on_message(message: IncomingMessage):
    print(" [x] Received message %r" % message)
    print("     Message body is: %r" % message.body)


async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    # Declaring queue
    queue = await channel.declare_queue('hello')

    # Start listening the queue with name 'hello'
    await queue.consume(on_message, no_ack=True)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main(loop))

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(" [*] Waiting for messages. To exit press CTRL+C")
    loop.run_forever()
```

Now we can try out our programs in a terminal. First, let's send a message using our send.py program:

```
$ python send.py
[x] Sent 'Hello World!'
```

The producer program send.py will stop after every run. Let's receive it:

```
$ python receive.py
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World!'
```

Hurray! We were able to send our first message through RabbitMQ. As you might have noticed, the *receive.py* program doesn't exit. It will stay ready to receive further messages, and may be interrupted with **Ctrl-C**.

Try to run *send.py* again in a new terminal.

We've learned how to send and receive a message from a named queue. It's time to move on to *part 2* and build a simple work queue.

Work Queues

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

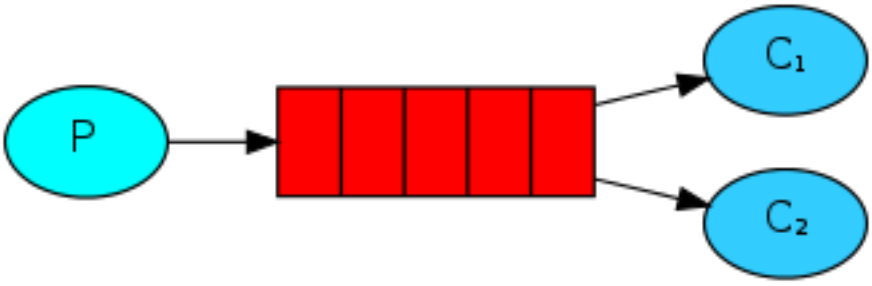
Note: Using the aio-pika async Python client

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.



In the *first tutorial* we wrote programs to send and receive messages from a named queue. In this one we'll create a Work Queue that will be used to distribute time-consuming tasks among multiple workers.

The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We 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. When you run many workers the tasks will be shared between them.

This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

Preparation

In the previous part of this tutorial we sent a message containing *"Hello World!"*. Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the time.sleep() function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by Hello... will take three seconds.

We will slightly modify the send.py code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it *new_task.py*:

```
import sys
from aio_pika import connect, Message

async def main(loop):
    ...

    message = ' '.join(sys.argv[1:]) or "Hello World!"

    # Sending the message
    await channel.default_exchange.publish(
        Message(message_body),
        routing_key='task_queue',
    )

    print(" [x] Sent %r" % message)

...
```

Our old receive.py script also requires some changes: it needs to fake a second of work for every dot in the message body. It will pop messages from the queue and perform the task, so let's call it *worker.py*:

```
import asyncio
from aio_pika import connect, IncomingMessage


loop = asyncio.get_event_loop()


def on_message(message: IncomingMessage):
    print(" [x] Received %r" % body)
    await asyncio.sleep(message.body.count(b'.'), loop=loop)
    print(" [x] Done")
```

Round-robin dispatching

One of the advantages of using a Task 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.

First, let's try to run two *worker.py* scripts at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the worker.py script. These consoles will be our two consumers - C1 and C2.

```
shell1$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
```

```
shell2$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
```

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
shell3$ python new_task.py First message.
shell3$ python new_task.py Second message..
shell3$ python new_task.py Third message...
shell3$ python new_task.py Fourth message....
shell3$ python new_task.py Fifth message.....
```

Let's see what is delivered to our workers:

```
shell1$ python worker.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'First message.'
 [x] Received 'Third message...'
 [x] Received 'Fifth message.....'
```

```
shell2$ python worker.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Second message..'
 [x] Received 'Fourth message....'
```

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

Message acknowledgment

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code once RabbitMQ delivers message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back from 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 understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

Message acknowledgments are turned on by default. In previous examples we explicitly turned them off via the *no_ack=True* flag. It's time to remove this flag and send a proper acknowledgment from the worker, once we're done

with a task.

```
from aio_pika import connect, IncomingMessage

def on_message(message: IncomingMessage):
    print(" [x] Received message %r" % message)
    print("     Message body is: %r" % message.body)
    message.ack()
```

or using special context processor:

```
from aio_pika import connect, IncomingMessage

def on_message(message: IncomingMessage):
    with message.process():
        print(" [x] Received message %r" % message)
        print("     Message body is: %r" % message.body)
```

If context processor will catch an exception, the message will be returned to the queue.

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered.

Note: Forgotten acknowledgment

It's a common mistake to miss the basic_ack. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use rabbitmqctl to print the messages_unacknowledged field:

```
$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
Listing queues ...
hello    0       0
...done.
```

Message durability

We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as *durable*:

```
async def main(loop):
    ...

    # Declaring queue
    queue = await channel.declare_queue('hello', durable=True)
```

Although this command is correct by itself, it won't work in our setup. That's because we've already defined a queue called hello which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters

and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example task_queue:

```
async def main(loop):
    ...

    # Declaring queue
    queue = await channel.declare_queue('task_queue', durable=True)
```

This queue_declare change needs to be applied to both the producer and consumer code.

At that point we're sure that the task_queue queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by supplying a delivery_mode property with a value *PERSISTENT* (see enum *aio_pika.DeliveryMode*).

```
async def main(loop):
    ...
    message_body = b' '.join(sys.argv[1:]) or b"Hello World!"

    message = Message(
        message_body,
        delivery_mode=DeliveryMode.PERSISTENT
    )

    # Sending the message
    await channel.default_exchange.publish(message, routing_key='task_queue')

    print(" [x] Sent %r" % message)
```

Note: Note on message persistence

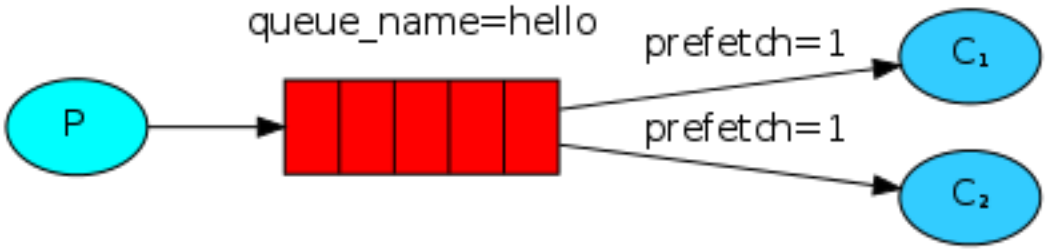
Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message – it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use publisher confirms.

'aio-pika'_ supports 'publisher confirms'_ out of the box.

Fair dispatch

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



In order to defeat that we can use the basic.qos method with the *prefetch_count=1* setting. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```
async def main(loop):
    ...

    await channel.set_qos(prefetch_count=1)
```

Note: Note about queue size

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

Putting it all together

Final code of our *new_task.py* script:

```
import sys
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    message_body = b' '.join(sys.argv[1:]) or b"Hello World!"

    message = Message(
        message_body,
        delivery_mode=DeliveryMode.PERSISTENT
    )

    # Sending the message
    await channel.default_exchange.publish(message, routing_key='task_queue')

    print(" [x] Sent %r" % message)

    await connection.close()

if __name__ == "__main__":
```

```
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

And our *worker.py*:

```
import asyncio
from aio_pika import connect, IncomingMessage


loop = asyncio.get_event_loop()


def on_message(message: IncomingMessage):
    print(" [x] Received %r" % body)
    await asyncio.sleep(message.body.count(b'.'), loop=loop)
    print(" [x] Done")


async def main():
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()
    await channel.set_qos(prefetch_count=1)

    # Declaring queue
    queue = await channel.declare_queue('task_queue', durable=True)

    # Start listening the queue with name 'task_queue'
    await queue.consume(on_message)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main())

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(" [*] Waiting for messages. To exit press CTRL+C")
    loop.run_forever()
```

Using message acknowledgments and prefetch_count you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.

Now we can move on to *tutorial 3* and learn how to deliver the same message to many consumers.

Publish/Subscribe

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

Note: Using the aio-pika async Python client

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

In the *previous tutorial* we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different — we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs — the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

Exchanges

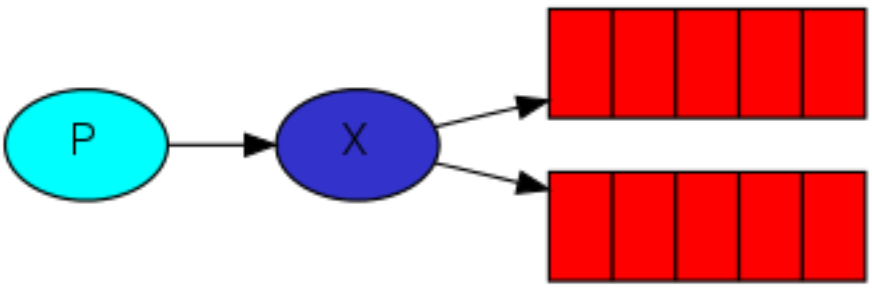
In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

Let's quickly go over what we covered in the previous tutorials:

- A producer is a user application that sends messages.
- A queue is a buffer that stores messages.
- A consumer is a user application that receives messages.

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an exchange. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the exchange type.



There are a few exchange types available: *DIRECT*, *TOPIC*, *HEADERS* and *FANOUT* (see *aio_pika.ExchangeType*). We'll focus on the last one — the fanout. Let's create an exchange of that type, and call it *logs*:

```
from aio_pika import ExchangeType

async def main():
    ...

    logs_exchange = await channel.declare_exchange('logs', ExchangeType.FANOUT)
```

The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

Note: Listing exchanges

To list the exchanges on the server you can run the ever useful rabbitmqctl:

```
$ sudo rabbitmqctl list_exchanges
Listing exchanges ...
logs      fanout
amq.direct      direct
amq.topic       topic
amq.fanout      fanout
amq.headers     headers
...done.
```

In this list there are some *amq.** exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

Nameless exchange

In previous parts of the tutorial we knew nothing about exchanges, but still were able to send messages to queues. That was possible because we were using a default exchange, which we identify by the empty string ("").

Recall how we published a message before:

```
await channel.default_exchange.publish(
    Message(message_body),
    routing_key='hello',
)
```

The exchange parameter is the name of the exchange. The empty string denotes the default or nameless exchange: messages are routed to the queue with the name specified by routing_key, if it exists.

Now, we can publish to our named exchange instead:

```
async def main():
    ...

    await logs_exchange.publish(
        Message(message_body),
        routing_key='hello',
    )

...
```

Temporary queues

As you may remember previously we were using queues which had a specified name (remember *hello* and *task_queue*?). Being able to name a queue was crucial for us — we needed to point the workers to the same queue. Giving a queue a name is important when you want to share the queue between producers and consumers.

But that's not the case for our logger. We want to hear about all log messages, not just a subset of them. We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.

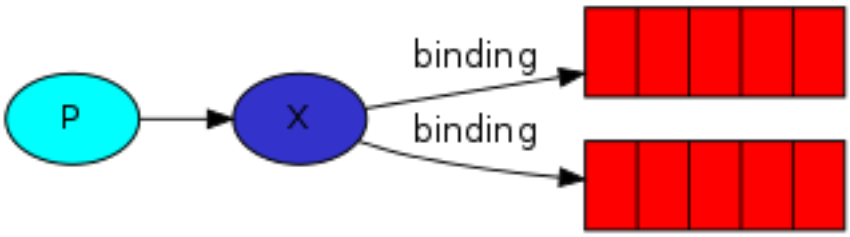
Firstly, whenever we connect to Rabbit we need a fresh, empty queue. To do it we could create a queue with a random name, or, even better - let the server choose a random queue name for us. We can do this by not supplying the queue parameter to *declare_queue*:

```
queue = await channel.declare_queue()
```

Secondly, once we disconnect the consumer the queue should be deleted. There's an exclusive flag for that:

```
queue = await channel.declare_queue(exclusive=True)
```

Bindings



We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a binding.

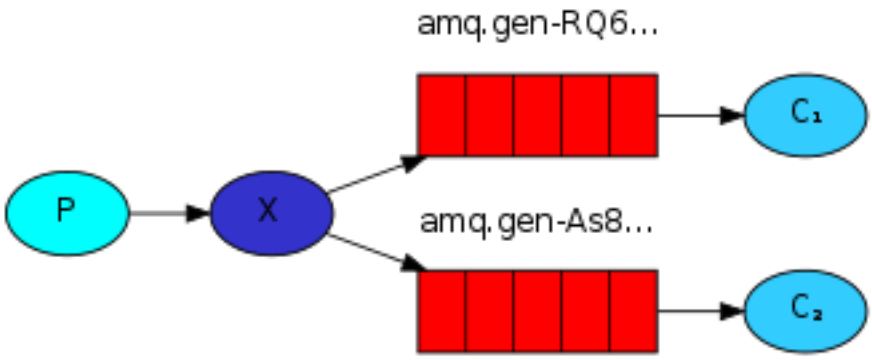
```
await queue.bind(exchange='logs')
```

From now on the logs exchange will append messages to our queue.

Note: Listing bindings

You can list existing bindings using, you guessed it, *rabbitmqctl list_bindings*.

Putting it all together



The producer program, which emits log messages, doesn't look much different from the previous tutorial. The most important change is that we now want to publish messages to our logs exchange instead of the nameless one. We need to supply a routing_key when sending, but its value is ignored for fanout exchanges. Here goes the code for *emit_log.py* script:

```
import sys
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    logs_exchange = await channel.declare_exchange('logs', ExchangeType.FANOUT)

    message_body = b' '.join(sys.argv[1:]) or b"Hello World!"

    message = Message(
        message_body,
        delivery_mode=DeliveryMode.PERSISTENT
    )

    # Sending the message
    await logs_exchange.publish(message, routing_key='task_queue')

    print(" [x] Sent %r" % message)

    await connection.close()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

As you see, after establishing the connection we declared the exchange. This step is necessary as publishing to a non-existing exchange is forbidden.

The messages will be lost if no queue is bound to the exchange yet, but that's okay for us; if no consumer is listening yet we can safely discard the message.

The code for *receive_logs.py*:

```
import asyncio
from aio_pika import connect, IncomingMessage


loop = asyncio.get_event_loop()


def on_message(message: IncomingMessage):
    print("[x] %r" % message.body)


async def main():
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()
    await channel.set_qos(prefetch_count=1)

    logs_exchange = await channel.declare_exchange(
        'logs',
        ExchangeType.FANOUT
    )

    # Declaring queue
    queue = await channel.declare_queue(exclusive=True)

    # Binding the queue to the exchange
    await queue.bind(logs_exchange)

    # Start listening the queue with name 'task_queue'
    await queue.consume(on_message)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main())

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(' [*] Waiting for logs. To exit press CTRL+C')
    loop.run_forever()
```

We're done. If you want to save logs to a file, just open a console and type:

```
$ python receive_logs.py > logs_from_rabbit.log
```

If you wish to see the logs on your screen, spawn a new terminal and run:

```
$ python receive_logs.py
```

And of course, to emit logs type:

```
$ python emit_log.py
```

Using *rabbitmqctl list_bindings* you can verify that the code actually creates bindings and queues as we want. With two *receive_logs.py* programs running you should see something like:

```
$ sudo rabbitmqctl list_bindings
Listing bindings ...
logs    exchange        amq.gen-JzTY20BRgKO-HjmUJj0wLg  queue           []
logs    exchange        amq.gen-vso0PVvyiRIL2WoV3i48Yg  queue           []
...done.
```

The interpretation of the result is straightforward: data from exchange logs goes to two queues with server-assigned names. And that's exactly what we intended.

To find out how to listen for a subset of messages, let's move on to *tutorial 4*

Routing

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

Note: Using the aio-pika async Python client

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

In the *previous tutorial* we built a simple logging system. We were able to broadcast log messages to many receivers.

In this tutorial we're going to add a feature to it — we're going to make it possible to subscribe only to a subset of the messages. For example, we will be able to direct only critical error messages to the log file (to save disk space), while still being able to print all of the log messages on the console.

Bindings

In previous examples we were already creating bindings. You may recall code like:

```
async def main():
    ...

    # Binding the queue to the exchange
    await queue.bind(logs_exchange)

...
```

A binding is a relationship between an exchange and a queue. This can be simply read as: the queue is interested in messages from this exchange.

Bindings can take an extra *routing_key* parameter. To avoid the confusion with a *basic_publish* parameter we're going to call it a *binding key*. This is how we could create a binding with a key:

```
async def main():
    ...

    # Binding the queue to the exchange
    await queue.bind(logs_exchange, routing_key="black")

...
```

The meaning of a binding key depends on the exchange type. The *fanout* exchanges, which we used previously, simply ignored its value.

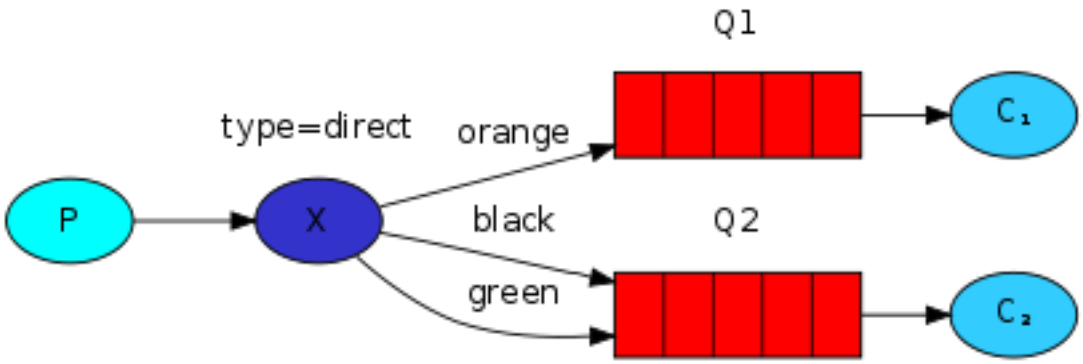
Direct exchange

Our logging system from the previous tutorial broadcasts all messages to all consumers. We want to extend that to allow filtering messages based on their severity. For example we may want the script which is writing log messages to the disk to only receive critical errors, and not waste disk space on warning or info log messages.

We were using a fanout exchange, which doesn't give us too much flexibility — it's only capable of mindless broadcasting.

We will use a direct exchange instead. The routing algorithm behind a direct exchange is simple — a message goes to the queues whose binding key exactly matches the routing key of the message.

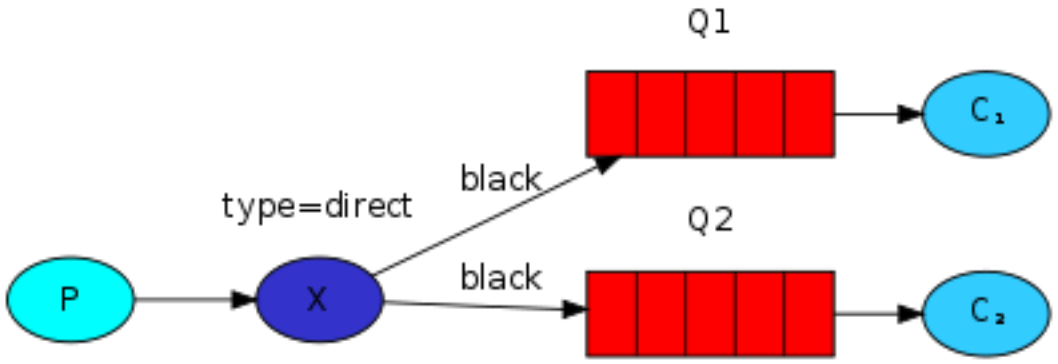
To illustrate that, consider the following setup:



In this setup, we can see the *direct* exchange X with two queues bound to it. The first queue is bound with binding key *orange*, and the second has two bindings, one with binding key *black* and the other one with *green*.

In such a setup a message published to the exchange with a routing key *orange* will be routed to queue *Q1*. Messages with a routing key of *black* or *green* will go to *Q2*. All other messages will be discarded.

Multiple bindings



It is perfectly legal to bind multiple queues with the same binding key. In our example we could add a binding between *X* and *Q1* with binding key *black*. In that case, the *direct* exchange will behave like fanout and will broadcast the message to all the matching queues. A message with routing key black will be delivered to both *Q1* and *Q2*.

Emitting logs

We'll use this model for our logging system. Instead of *fanout* we'll send messages to a *direct* exchange. We will supply the log severity as a *routing key*. That way the receiving script will be able to select the severity it wants to receive. Let's focus on emitting logs first.

Like always we need to create an exchange first:

```
from aio_pika import ExchangeType

async def main():
    ...

    direct_logs_exchange = await channel.declare_exchange('logs', ExchangeType.DIRECT)
```

And we're ready to send a message:

```
async def main():
    ...

    await direct_logs_exchange.publish(
        Message(message_body),
        routing_key=severity',
    )
```

To simplify things we will assume that *'severity'* can be one of *'info'*, *'warning'*, *'error'*.

Subscribing

Receiving messages will work just like in the previous tutorial, with one exception - we're going to create a new binding for each severity we're interested in.

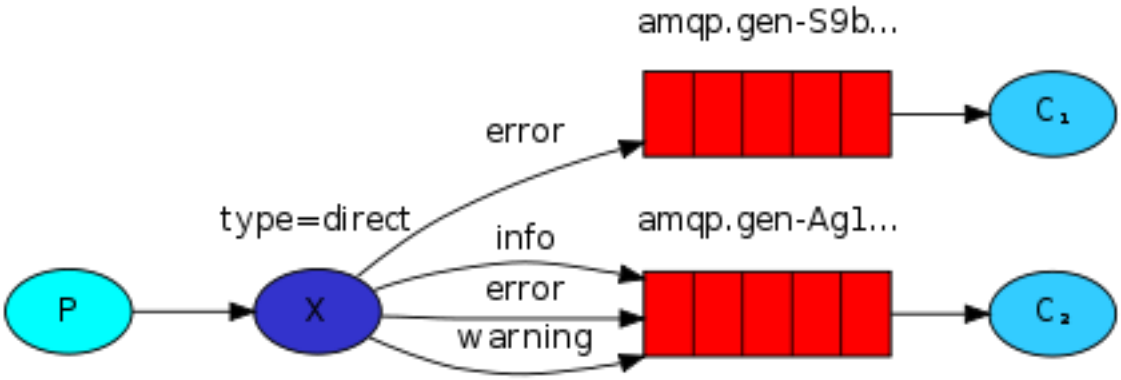
```
async def main():
    ...

    # Declaring queue
    queue = await channel.declare_queue(exclusive=True)

    # Binding the queue to the exchange
    await queue.bind(direct_logs_exchange, routing_key=severity)

...
```

Putting it all together



The code for *emit_log_direct.py*:

```
import sys
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    direct_logs_exchange = await channel.declare_exchange('logs', ExchangeType.DIRECT)

    severity = sys.argv[1] if len(sys.argv) > 2 else 'info'
    message_body = b' '.join(sys.argv[2:]) or b"Hello World!"

    message = Message(
        message_body,
        delivery_mode=DeliveryMode.PERSISTENT
    )

    # Sending the message
    await direct_logs_exchange.publish(message, routing_key=severity)

    print(" [x] Sent %r" % message)
```

```
    await connection.close()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

The code for *receive_logs_direct.py*:

```
import asyncio
from aio_pika import connect, IncomingMessage


def on_message(message: IncomingMessage):
    print(" [x] %r:%r" % (message.routing_key, message.body))


async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()
    await channel.set_qos(prefetch_count=1)

    severities = sys.argv[1:]
    if not severities:
        sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
        sys.exit(1)

    # Declare an exchange
    direct_logs_exchange = await channel.declare_exchange('logs', ExchangeType.DIRECT)

    # Declaring queue
    queue = await channel.declare_queue('task_queue', durable=True)

    for severity in severities:
        await queue.bind(direct_logs_exchange, routing_key=severity)

    # Start listening the queue with name 'task_queue'
    await queue.consume(on_message)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main(loop))

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(" [*] Waiting for messages. To exit press CTRL+C")
    loop.run_forever()
```

If you want to save only *'warning'* and *'error'* (and not *'info'*) log messages to a file, just open a console and type:

```
$ python receive_logs_direct.py warning error > logs_from_rabbit.log
```

If you'd like to see all the log messages on your screen, open a new terminal and do:

```
$ python receive_logs_direct.py info warning error
 [*] Waiting for logs. To exit press CTRL+C
```

And, for example, to emit an error log message just type:

```
$ python emit_log_direct.py error "Run. Run. Or it will explode."
[x] Sent 'error':'Run. Run. Or it will explode.'
```

Move on to *tutorial 5* to find out how to listen for messages based on a pattern.

Topics

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

Note: Using the aio-pika async Python client

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

In the *previous tutorial* we improved our logging system. Instead of using a fanout exchange only capable of dummy broadcasting, we used a direct one, and gained a possibility of selectively receiving the logs.

Although using the direct exchange improved our system, it still has limitations — it can't do routing based on multiple criteria.

In our logging system we might want to subscribe to not only logs based on severity, but also based on the source which emitted the log. You might know this concept from the syslog unix tool, which routes logs based on both severity (*info/warn/crit...*) and facility (*auth/cron/kern...*).

That would give us a lot of flexibility - we may want to listen to just critical errors coming from 'cron' but also all logs from 'kern'.

To implement that in our logging system we need to learn about a more complex topic exchange.

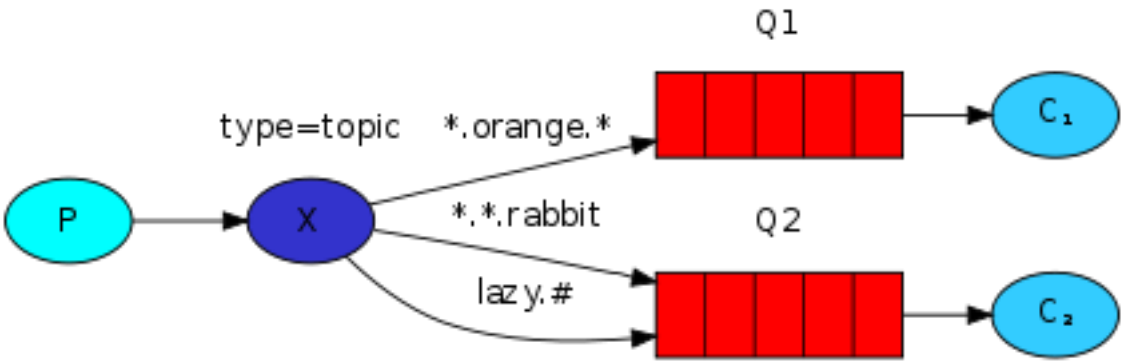
Topic exchange

Messages sent to a topic exchange can't have an arbitrary *routing_key* - it must be a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message. A few valid routing key examples: *"stock.usd.nyse"*, *"nyse.vmw"*, *"quick.orange.rabbit"*. There can be as many words in the routing key as you like, up to the limit of 255 bytes.

The binding key must also be in the same form. The logic behind the topic exchange is similar to a direct one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys:

- * (star) can substitute for exactly one word.
- # (hash) can substitute for zero or more words.

It's easiest to explain this in an example:



In this example, we're going to send messages which all describe animals. The messages will be sent with a routing key that consists of three words (two dots). The first word in the routing key will describe a celerity, second a colour and third a species: *"<celerity>.<colour>.<species>"*.

We created three bindings: *Q1* is bound with binding key *"*.orange.*"* and Q2 with *"*.*.rabbit"* and *"lazy.#"*.

These bindings can be summarised as:

- Q1 is interested in all the orange animals.
- Q2 wants to hear everything about rabbits, and everything about lazy animals.
- A message with a routing key set to *"quick.orange.rabbit"* will be delivered to both queues. Message *"lazy.orange.elephant"* also will go to both of them. On the other hand *"quick.orange.fox"* will only go to the first queue, and *"lazy.brown.fox"* only to the second. *"lazy.pink.rabbit"* will be delivered to the second queue only once, even though it matches two bindings. "quick.brown.fox" doesn't match any binding so it will be discarded.

What happens if we break our contract and send a message with one or four words, like *"orange"* or *"quick.orange.male.rabbit"*? Well, these messages won't match any bindings and will be lost.

On the other hand *"lazy.orange.male.rabbit"*, even though it has four words, will match the last binding and will be delivered to the second queue.

Note: Topic exchange

Topic exchange is powerful and can behave like other exchanges.

When a queue is bound with *"#"* (hash) binding key - it will receive all the messages, regardless of the routing key - like in fanout exchange.

When special characters *"*"* (star) and *"#"* (hash) aren't used in bindings, the topic exchange will behave just like a direct one.

Putting it all together

We're going to use a topic exchange in our logging system. We'll start off with a working assumption that the routing keys of logs will have two words: *"<facility>.<severity>"*.

The code is almost the same as in the *previous tutorial*.

The code for *emit_log_topic.py*:

```
import sys
import asyncio
from aio_pika import connect, Message

async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    topic_logs_exchange = await channel.declare_exchange('topic_logs', ExchangeType.
↪TOPIC)

    routing_key = sys.argv[1] if len(sys.argv) > 2 else 'anonymous.info'
    message_body = b' '.join(sys.argv[2:]) or b"Hello World!"

    message = Message(
        message_body,
        delivery_mode=DeliveryMode.PERSISTENT
    )

    # Sending the message
    await topic_logs_exchange.publish(message, routing_key=routing_key)

    print(" [x] Sent %r" % message)

    await connection.close()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

The code for *receive_logs_topic.py*:

```
import asyncio
from aio_pika import connect, IncomingMessage


def on_message(message: IncomingMessage):
    print(" [x] %r:%r" % (message.routing_key, message.body))


async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()
    await channel.set_qos(prefetch_count=1)
```

```
    # Declare an exchange
    topic_logs_exchange = await channel.declare_exchange('topic_logs', ExchangeType.
↪TOPIC)

    # Declaring queue
    queue = await channel.declare_queue('task_queue', durable=True)

    binding_keys = sys.argv[1:]

    if not binding_keys:
        sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
        sys.exit(1)

    for binding_key in binding_keys:
        await queue.bind(topic_logs_exchange, routing_key=binding_key)

    # Start listening the queue with name 'task_queue'
    await queue.consume(on_message)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main(loop))

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(" [*] Waiting for messages. To exit press CTRL+C")
    loop.run_forever()
```

To receive all the logs run:

```
python receive_logs_topic.py "#"
```

To receive all logs from the facility *"kern"*:

```
python receive_logs_topic.py "kern.*"
```

Or if you want to hear only about *"critical"* logs:

```
python receive_logs_topic.py "*.critical"
```

You can create multiple bindings:

```
python receive_logs_topic.py "kern.*" "*.critical"
```

And to emit a log with a routing key *"kern.critical"* type:

```
python emit_log_topic.py "kern.critical" "A critical kernel error"
```

Have fun playing with these programs. Note that the code doesn't make any assumption about the routing or binding keys, you may want to play with more than two routing key parameters.

Move on to *tutorial 6* to learn about RPC.

Remote procedure call (RPC)

Warning: This is a beta version of the port from official tutorial. Please when you found an error create issue or pull request for me.

Note: Using the aio-pika async Python client

Note: Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (*5672*). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

In the *second tutorial* we learned how to use *Work Queues* to distribute time-consuming tasks among multiple workers.

But what if we need to run a function on a remote computer and wait for the result? Well, that's a different story. This pattern is commonly known as *Remote Procedure Call or RPC*.

In this tutorial we're going to use RabbitMQ to build an RPC system: a client and a scalable RPC server. As we don't have any time-consuming tasks that are worth distributing, we're going to create a dummy RPC service that returns Fibonacci numbers.

Client interface

To illustrate how an RPC service could be used we're going to create a simple client class. It's going to expose a method named call which sends an RPC request and blocks until the answer is received:

```
async def main():
    fibonacci_rpc = FibonacciRpcClient()
    result = await fibonacci_rpc.call(4)
    print("fib(4) is %r" % result)
```

Note: A note on RPC

Although RPC is a pretty common pattern in computing, it's often criticised. The problems arise when a programmer is not aware whether a function call is local or if it's a slow RPC. Confusions like that result in an unpredictable system and adds unnecessary complexity to debugging. Instead of simplifying software, misused RPC can result in unmaintainable spaghetti code.

Bearing that in mind, 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, results are asynchronously pushed to a next computation stage.

Callback queue

In general doing RPC over RabbitMQ is easy. 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. Let's try it:

```
async def main():
    ...

    # Queue for results
    callback_queue = await channel.declare_queue(exclusive=True)

    await channel.default_exchange.publish(
        Message(
            request,
            reply_to=callback_queue.name
        ),
        routing_key='rpc_queue'
    )

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

...
```

Note: Message properties

The AMQP protocol predefines a set of 14 properties that go with a message. Most of the properties are rarely used, with the exception of the following:

- *delivery_mode*: Marks a message as persistent (with a value of 2) or transient (any other value). You may remember this property from the *second tutorial*.
- *content_type*: Used to describe the mime-type of the encoding. For example for the often used JSON encoding it is a good practice to set this property to: application/json.
- *reply_to*: Commonly used to name a callback queue.
- *correlation_id*: Useful to correlate RPC responses with requests.

See additional info in *aio_pika.Message*

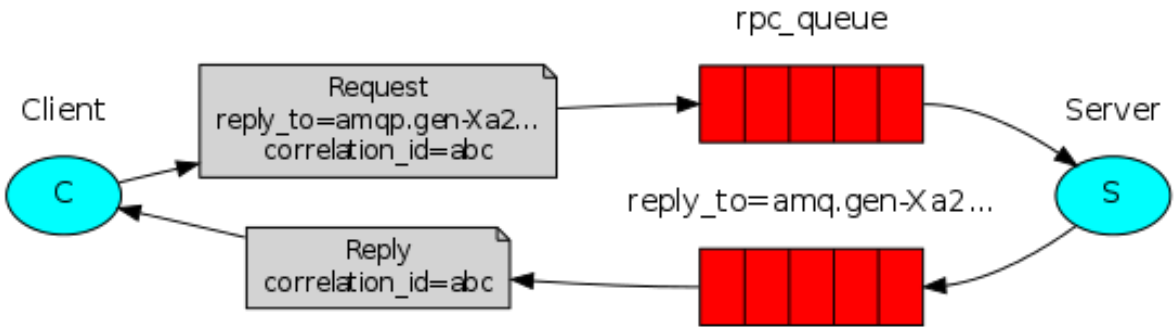
Correlation id

In the method presented above we suggest creating a callback queue for every RPC request. That's pretty inefficient, but fortunately there is a better way - let's create a single callback queue per client.

That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the *correlation_id* property is used. We're going to set it to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request. If we see an unknown *correlation_id* value, we may safely discard the message - it doesn't belong to our requests.

You may ask, why should we ignore unknown messages in the callback queue, rather than failing with an error? It's due to a possibility of a race condition on the server side. Although unlikely, it is possible that the RPC server will die just after sending us the answer, but before sending an acknowledgment message for the request. If that happens, the restarted RPC server will process the request again. That's why on the client we must handle the duplicate responses gracefully, and the RPC should ideally be idempotent.

Summary



Our RPC will work like this:

- When the Client starts up, it creates an anonymous exclusive callback queue.
- For an RPC request, the Client sends a message with two properties: *reply_to*, which is set to the callback queue and *correlation_id*, which is set to a unique value for every request.
- The request is sent to an rpc_queue queue.
- The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from the reply_to field.
- The client waits for data on the callback queue. When a message appears, it checks the *correlation_id* property. If it matches the value from the request it returns the response to the application.

Putting it all together

The code for *rpc_server.py*:

```
import asyncio
from functools import partial
from aio_pika import connect, IncomingMessage, Exchange, Message


def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)


def on_message(exchange: Exchange, message: IncomingMessage):
    with message.process():
        n = int(body)

        print(" [.] fib(%s)" % n)
        response = fib(n)

        exchange.publish(
```

```
            Message(
                body=str(response)
                correlation_id=message.correlation_id
            ),
            routing_key=message.reply_to
        )


async def main(loop):
    # Perform connection
    connection = await connect("amqp://guest:guest@localhost/", loop=loop)

    # Creating a channel
    channel = await connection.channel()

    # Declaring queue
    queue = await channel.declare_queue('rpc_queue')

    # Start listening the queue with name 'hello'
    await queue.consume(
        partial(
            on_message,
            channel.default_exchange
        )
    )


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.add_callback(main(loop))

    # we enter a never-ending loop that waits for data and runs callbacks whenever␣
↪necessary.
    print(" [x] Awaiting RPC requests")
    loop.run_forever()
```

The server code is rather straightforward:

- 33. As usual we start by establishing the connection and declaring the queue.
- (6) We declare our fibonacci function. It assumes only valid positive integer input. (Don't expect this one to work for big numbers, it's probably the slowest recursive implementation possible).
- (15) We declare a callback for basic_consume, the core of the RPC server. It's executed when the request is received. It does the work and sends the response back.

The code for rpc_client.py:

```
import asyncio
from functools import partial
from aio_pika import connect, IncomingMessage, Message


class FibonacciRpcClient:
    def __init__(self, loop):
        self.connection = None
        self.channel = None
        self.callback_queue = None
        self.futures = {}
```

```
12          self.loop = loop
13
14      async def connect(self):
15          self.connection = await connect("amqp://guest:guest@localhost/", loop=loop)
16          self.channel = await connection.channel()
17          self.callback_queue = await channel.declare_queue(exclusive=True)
18
19          await queue.consume(self.on_response)
20
21          return self
22
23      def on_response(self, message: IncomingMessage):
24          future = self.futures.pop(message.correlation_id)
25          future.set_result(message.body)
26
27      async def call(self, n):
28          correlation_id = str(uuid.uuid4())
29          future = loop.create_future()
30
31          self.futures[correlation_id] = future
32
33          self.channel.default_exchange.publish(
34              Message(
35                  bytes(n),
36                  correlation_id=correlation_id,
37                  reply_to=self.callback_queue.name,
38              ),
39              routing_key='rpc_queue',
40          )
41
42          return int(await future)
43
44
45  async def main(loop):
46      fibonacci_rpc = await FibonacciRpcClient(loop).connect()
47      print(" [x] Requesting fib(30)")
48      response = await fibonacci_rpc.call(30)
49      print(" [.] Got %r" % response)
50
51
52  if __name__ == "__main__":
53      loop = asyncio.get_event_loop()
54      loop.run_until_complete(main())
```

The client code is slightly more involved:

- 15. We establish a connection, channel and declare an exclusive 'callback' queue for replies.
- 19. We subscribe to the 'callback' queue, so that we can receive RPC responses.
- 23. The 'on_response' callback executed on every response is doing a very simple job,

  for every response message it checks if the correlation_id is the one we're looking for. If so, it saves the response in self.response and breaks the consuming loop.
- 27. Next, we define our main call method - it does the actual RPC request.
- 28. In this method, first we generate a unique correlation_id number and save it - the 'on_response' callback function will use this value to catch the appropriate response.

- 33. Next, we publish the request message, with two properties: reply_to and correlation_id.
- 42. And finally we return the response back to the user.

Our RPC service is now ready. We can start the server:

```
$ python rpc_server.py
[x] Awaiting RPC requests
```

To request a fibonacci number run the client:

```
$ python rpc_client.py
[x] Requesting fib(30)
```

The presented design is not the only possible implementation of a RPC service, but it has some important advantages:

If the RPC server is too slow, you can scale up by just running another one. Try running a second rpc_server.py in a new console. On the client side, the RPC requires sending and receiving only one message. No synchronous calls like queue_declare are required. As a result the RPC client needs only one network round trip for a single RPC request. Our code is still pretty simplistic and doesn't try to solve more complex (but important) problems, like:

- How should the client react if there are no servers running?
- Should a client have some kind of timeout for the RPC?
- If the server malfunctions and raises an exception, should it be forwarded to the client?
- Protecting against invalid incoming messages (eg checking bounds) before processing.

Note: If you want to experiment, you may find the rabbitmq-management plugin useful for viewing the queues.

---

aio_pika package

aio_pika.**connect**(*url: str = None*, ***, *host: str = 'localhost'*, *port: int = 5672*, *login: str = 'guest'*, *password: str = 'guest'*, *virtualhost: str = '/'*, *ssl: bool = False*, *loop=None*, ***kwargs*) → aio_pika.connection.Connection

Make connection to the broker

Parameters

- **url** – RFC3986 formatted broker address. When `None` will be used keyword arguments.
- **host** – hostname of the broker
- **port** – broker port 5672 by default
- **login** – username string. *'guest'* by default.
- **password** – password string. *'guest'* by default.
- **virtualhost** – virtualhost parameter. *'/'* by default
- **ssl** – use SSL for connection. Should be used with addition kwargs. See pika documentation for more info.
- **loop** – Event loop (`asyncio.get_event_loop()` when `None`)
- **kwargs** – addition parameters which will be passed to the pika connection.

Returns `aio_pika.connection.Connection`

aio_pika.**connect_url**(*url: str*, *loop=None*) → aio_pika.connection.Connection

class aio_pika.**Connection**(*host: str = 'localhost'*, *port: int = 5672*, *login: str = 'guest'*, *password: str = 'guest'*, *virtual_host: str = '/'*, *ssl: bool = False*, ***, *loop=None*, ***kwargs*)

Bases: object

Connection abstraction

add_close_callback(*callback: typing.Callable[[], NoneType]*)

Add callback which will be called after connection will be closed.

Returns None

channel() → aio_pika.channel.Channel

Get a channel

close() → None

Close AMQP connection

closing

Return future which will be finished after connection close.

connect()

Perform connect. This method should be called after aio_pika.connection.Connection.__init__()

is_closed

Is this connection are closed

loop

class aio_pika.**Channel**(*connection*, *loop: asyncio.events.AbstractEventLoop*, *future_store: aio_pika.common.FutureStore*)

Bases: aio_pika.common.BaseChannel

Channel abstraction

add_close_callback(*callback: function*)

close() → None

declare_exchange(*name: str*, *type: aio_pika.exchange.ExchangeType = <ExchangeType.DIRECT: 'direct'>*, *durable: bool = None*, *auto_delete: bool = False*, *arguments: dict = None*, *timeout: int = None*) → aio_pika.exchange.Exchange

declare_queue(*name: str = None*, ***, *durable: bool = None*, *exclusive: bool = False*, *auto_delete: bool = False*, *arguments: dict = None*, *timeout: int = None*) → aio_pika.queue.Queue

Parameters

- **name** – queue name
- **durable** – Durability (queue survive broker restart)
- **exclusive** – Makes this queue exclusive. Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes. Passive declaration of an exclusive queue by other connections are not allowed.
- **auto_delete** – Delete queue when channel will be closed.
- **arguments** – pika additional arguments
- **timeout** – execution timeout

Returns aio_pika.queue.Queue instance

default_exchange

exchange_delete(*exchange_name: str*, *timeout: int = None*, *if_unused=False*, *nowait=False*)

initialize(*timeout=None*) → None

loop

queue_delete(*queue_name: str*, *timeout: int = None*, *if_unused: bool = False*, *if_empty: bool = False*, *nowait: bool = False*)

set_qos(*prefetch_count: int = 0*, *prefetch_size: int = 0*, *all_channels=False*, *timeout: int = None*)

class aio_pika.**Exchange**(*channel: pika.channel.Channel*, *publish_method*, *name: str*, *type: aio_pika.exchange.ExchangeType = <ExchangeType.DIRECT: 'direct'>*, ***, *auto_delete: bool*, *durable: bool*, *arguments: dict*, *loop: asyncio.events.AbstractEventLoop*, *future_store: aio_pika.common.FutureStore*)

Bases: aio_pika.common.BaseChannel

Exchange abstraction

arguments

auto_delete

delete(*if_unused=False*) → asyncio.futures.Future

Delete the queue

Parameters **if_unused** – perform deletion when queue has no bindings.

durable

name

publish(*message: aio_pika.message.Message*, *routing_key*, ***, *mandatory=True*, *immediate=False*)

Publish the message to the queue. *aio_pika* use publisher confirms extension for message delivery.

class aio_pika.**Message**(*body: bytes*, ***, *headers: dict = None*, *content_type: str = None*, *content_encoding: str = None*, *delivery_mode: aio_pika.message.DeliveryMode = <DeliveryMode.NOT_PERSISTENT: 1>*, *priority: int = None*, *correlation_id=None*, *reply_to: str = None*, *expiration: typing.Union[int, datetime.datetime, float, datetime.timedelta, NoneType] = None*, *message_id: str = None*, *timestamp: typing.Union[int, datetime.datetime, float, datetime.timedelta, NoneType] = None*, *type: str = None*, *user_id: str = None*, *app_id: str = None*)

Bases: object

AMQP message abstraction

app_id

body

content_encoding

content_type

correlation_id

delivery_mode

expiration

headers

info()

lock()

locked
is message locked

Returns bool

message_id

priority

properties

reply_to

timestamp

type

user_id

class aio_pika.**IncomingMessage**(*channel: pika.channel.Channel*, *envelope*, *properties*, *body*, *no_ack: bool = False*)
Bases: aio_pika.message.Message

Incoming message it's seems like Message but has additional methods for message acknowledgement.

Depending on the acknowledgement mode used, RabbitMQ can consider a message to be successfully delivered either immediately after it is sent out (written to a TCP socket) or when an explicit ("manual") client acknowledgement is received. Manually sent acknowledgements can be positive or negative and use one of the following protocol methods:

- basic.ack is used for positive acknowledgements
- basic.nack is used for negative acknowledgements (note: this is a RabbitMQ extension to AMQP 0-9-1)
- basic.reject is used for negative acknowledgements but has one limitations compared to basic.nack

Positive acknowledgements simply instruct RabbitMQ to record a message as delivered. Negative acknowledgements with basic.reject have the same effect. The difference is primarily in the semantics: positive acknowledgements assume a message was successfully processed while their negative counterpart suggests that a delivery wasn't processed but still should be deleted.

ack()
Send basic.ack is used for positive acknowledgements

Returns None

cluster_id

consumer_tag

delivery_tag

exchange

info()
Method returns dict representation of the message

process(*requeue=False*, *reject_on_redelivered=False*)
Context manager for processing the message

```
>>> def on_message_received(message: IncomingMessage):
...     with message.process():
...         # When exception will be raised
...         # the message will be rejected
...         print(message.body)
```

Parameters

- **requeue** – Requeue message when exception.
- **reject_on_redelivered** – When True message will be rejected only when message was redelivered.

redelivered

reject(*requeue=False*)
When *requeue=True* the message will be returned to queue. Otherwise message will be dropped.

Parameters **requeue** – bool

routing_key

synchronous

class aio_pika.**Queue**(*loop: asyncio.events.AbstractEventLoop*, *future_store: aio_pika.common.FutureStore*, *channel: pika.channel.Channel*, *name*, *durable*, *exclusive*, *auto_delete*, *arguments*)
Bases: aio_pika.common.BaseChannel

AMQP queue abstraction

arguments

auto_delete

bind(*exchange: aio_pika.exchange.Exchange*, *routing_key: str = None*, *, *arguments=None*, *timeout: int = None*) → asyncio.futures.Future
A binding is a relationship between an exchange and a queue. This can be simply read as: the queue is interested in messages from this exchange.

Bindings can take an extra routing_key parameter. To avoid the confusion with a basic_publish parameter we're going to call it a binding key.

Parameters

- **exchange** – aio_pika.exchange.Exchange instance
- **routing_key** – routing key
- **arguments** – additional arguments (will be passed to *pika*)
- **timeout** – execution timeout

Returns None

consume(*callback: function*, *no_ack: bool = False*, *exclusive: bool = False*, *arguments: dict = None*)
Start to consuming the *Queue*.

Parameters

- **callback** – Consuming callback
- **no_ack** – if True you don't need to call aio_pika.message.IncomingMessage.ack()
- **exclusive** – Makes this queue exclusive. Exclusive queues may only be accessed by the current connection,

and are deleted when that connection closes. Passive declaration of an exclusive queue by other connections are not allowed. :return: None

declare(*timeout: int = None*) → asyncio.futures.Future
Declare queue.

Parameters **timeout** – execution timeout

Returns None

delete(*, *if_unused=True*, *if_empty=True*, *timeout=None*) → asyncio.futures.Future
Delete the queue.

Parameters

- **if_unused** – Perform delete only when unused
- **if_empty** – Perform delete only when empty
- **timeout** – execution timeout

Returns None

durable

exclusive

get(*, *no_ack=False*, *timeout=None*) → aio_pika.message.IncomingMessage
Get message from the queue.

Parameters

- **no_ack** – if True you don't need to call aio_pika.message.IncomingMessage.ack()
- **timeout** – execution timeout

Returns aio_pika.message.IncomingMessage

name

purge(*timeout=None*) → asyncio.futures.Future
Purge all messages from the queue.

Parameters **timeout** – execution timeout

Returns None

unbind(*exchange: aio_pika.exchange.Exchange*, *routing_key: str*, *arguments: dict = None*, *timeout: int = None*) → asyncio.futures.Future
Remove binding from exchange for this *Queue* instance

Parameters

- **exchange** – aio_pika.exchange.Exchange instance
- **routing_key** – routing key
- **arguments** – additional arguments (will be passed to *pika*)
- **timeout** – execution timeout

Returns None

exception aio_pika.**AMQPException**
Bases: Exception

exception aio_pika.**MessageProcessError**
Bases: aio_pika.exceptions.AMQPException

class aio_pika.**ExchangeType**
Bases: enum.Enum

An enumeration.

DIRECT = 'direct'

FANOUT = 'fanout'

HEADERS = 'headers'

TOPIC = 'topic'

class aio_pika.**DeliveryMode**

Bases: enum.IntEnum

An enumeration.

NOT_PERSISTENT = 1

PERSISTENT = 2

CHAPTER 2

Installation

```
pip install aio-pika
```

CHAPTER 3

Usage example

```
from aio_pika import connect


async def main(loop):
    connection = await connect("amqp://guest:guest@127.0.0.1/", loop=loop)

    queue_name = "test_queue"
    routing_key = "test_queue"

    # Creating channel
    channel = await connection.channel()

    # Declaring exchange
    exchange = await channel.declare_exchange('direct', auto_delete=True)

    # Declaring queue
    queue = await channel.declare_queue(queue_name, auto_delete=True)

    # Binding queue
    await queue.bind(exchange, routing_key)


    await exchange.publish(
        Message(
            bytes('Hello', 'utf-8'),
            content_type='text/plain',
            headers={'foo': 'bar'}
        ),
        routing_key
    )

    # Receiving message
    incoming_message = await queue.get(timeout=5)

    # Confirm message
```

```
    incoming_message.ack()

    await queue.unbind(exchange, routing_key)
    await queue.delete()
    await connection.close()


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

CHAPTER 4

Thanks for contributing

- @mosquito (author)
- @hellysmile (bug fixes and smart ideas)
- @adelkhafizova (helps with documentation)

Python Module Index

a

aio_pika, 35

Index

A

ack() (aio_pika.IncomingMessage method), 38
add_close_callback() (aio_pika.Channel method), 36
add_close_callback() (aio_pika.Connection method), 36
aio_pika (module), 35
AMQPException, 40
app_id (aio_pika.Message attribute), 37
arguments (aio_pika.Exchange attribute), 37
arguments (aio_pika.Queue attribute), 39
auto_delete (aio_pika.Exchange attribute), 37
auto_delete (aio_pika.Queue attribute), 39

B

bind() (aio_pika.Queue method), 39
body (aio_pika.Message attribute), 37

C

Channel (class in aio_pika), 36
channel() (aio_pika.Connection method), 36
close() (aio_pika.Channel method), 36
close() (aio_pika.Connection method), 36
closing (aio_pika.Connection attribute), 36
cluster_id (aio_pika.IncomingMessage attribute), 38
connect() (aio_pika.Connection method), 36
connect() (in module aio_pika), 35
connect_url() (in module aio_pika), 35
Connection (class in aio_pika), 36
consume() (aio_pika.Queue method), 39
consumer_tag (aio_pika.IncomingMessage attribute), 38
content_encoding (aio_pika.Message attribute), 37
content_type (aio_pika.Message attribute), 37
correlation_id (aio_pika.Message attribute), 37

D

declare() (aio_pika.Queue method), 39
declare_exchange() (aio_pika.Channel method), 36
declare_queue() (aio_pika.Channel method), 36
default_exchange (aio_pika.Channel attribute), 37
delete() (aio_pika.Exchange method), 37
delete() (aio_pika.Queue method), 40
delivery_mode (aio_pika.Message attribute), 37
delivery_tag (aio_pika.IncomingMessage attribute), 38
DeliveryMode (class in aio_pika), 41
DIRECT (aio_pika.ExchangeType attribute), 40
durable (aio_pika.Exchange attribute), 37
durable (aio_pika.Queue attribute), 40

E

exchange (aio_pika.IncomingMessage attribute), 38
Exchange (class in aio_pika), 37
exchange_delete() (aio_pika.Channel method), 37
ExchangeType (class in aio_pika), 40
exclusive (aio_pika.Queue attribute), 40
expiration (aio_pika.Message attribute), 37

F

FANOUT (aio_pika.ExchangeType attribute), 41

G

get() (aio_pika.Queue method), 40

H

HEADERS (aio_pika.ExchangeType attribute), 41
headers (aio_pika.Message attribute), 37

I

IncomingMessage (class in aio_pika), 38
info() (aio_pika.IncomingMessage method), 38
info() (aio_pika.Message method), 37
initialize() (aio_pika.Channel method), 37
is_closed (aio_pika.Connection attribute), 36

L

lock() (aio_pika.Message method), 37
locked (aio_pika.Message attribute), 37
loop (aio_pika.Channel attribute), 37
loop (aio_pika.Connection attribute), 36

M

Message (class in aio_pika), 37
message_id (aio_pika.Message attribute), 38
MessageProcessError, 40

N

name (aio_pika.Exchange attribute), 37
name (aio_pika.Queue attribute), 40
NOT_PERSISTENT (aio_pika.DeliveryMode attribute), 41

P

PERSISTENT (aio_pika.DeliveryMode attribute), 41
priority (aio_pika.Message attribute), 38
process() (aio_pika.IncomingMessage method), 38
properties (aio_pika.Message attribute), 38
publish() (aio_pika.Exchange method), 37
purge() (aio_pika.Queue method), 40

Q

Queue (class in aio_pika), 39
queue_delete() (aio_pika.Channel method), 37

R

redelivered (aio_pika.IncomingMessage attribute), 39
reject() (aio_pika.IncomingMessage method), 39
reply_to (aio_pika.Message attribute), 38
routing_key (aio_pika.IncomingMessage attribute), 39

S

set_qos() (aio_pika.Channel method), 37
synchronous (aio_pika.IncomingMessage attribute), 39

T

timestamp (aio_pika.Message attribute), 38
TOPIC (aio_pika.ExchangeType attribute), 41
type (aio_pika.Message attribute), 38

U

unbind() (aio_pika.Queue method), 40
user_id (aio_pika.Message attribute), 38