Отправляем и получаем

```
import asyncio
import os

from aio_pika import connect, Message, IncomingMessage


async def create_connection():
    return await connect(url=os.getenv("RABBITMQ_URL"))


async def on_message(message: IncomingMessage):
    """
    on_message doesn't necessarily have to be defined as async.
    Here it is to show that it's possible.
    """
    print(" [x] Received message %r" % message)
    print("Message body is: %r" % message.body)
    print("Before sleep!")
    await asyncio.sleep(5)  # Represents async I/O operations
    print("After sleep!")


async def receiver():
    # Perform connection
    connection = await create_connection()

    # 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)
```

```
async def sender():
    # Perform connection
    connection = await create_connection()

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

    # Declaring queue
    await channel.declare_queue("hello")

    # Sending the message
    for i in range(2):
        await channel.default_exchange.publish(
            Message(f"{i} - Hello World!".encode()),
            routing_key="hello",
        )

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

    await connection.close()


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(receiver())
    loop.run_until_complete(sender())

    print(" [*] Waiting for messages.")
    loop.run_forever()
```

Запустить пример можно в mercury:

```
python3 connectors/rabbit/intro.py
```

Рассмотрим функции подробнее.

receiver

- вызвать функцию *connect* для создания подключения к Rabbit
- получить *channel*. Channel — это логическое соединение. По сути channel — это то же самое, что и connection, только без реального TCP-соединения. В рамках одного физического подключения может быть несколько логических.
- объявить очередь с помощью функции *declare_queue*. Функцию declare_queue можно вызывать несколько раз, ошибок она не вызовет. Такое поведение называется идемпотентным.
- начать слушать сообщения из очереди с помощью функции *queue.consume*

queue.consume

- Когда сообщение поступит в очередь, и до него дойдет очередь, то будет вызван *callback on_message*
- Параметр *no_ack=True* означает, что обработку сообщений не нужно подтверждать. Об этом расскажем подробнее чуть позже
- Есть более привычный для asyncio синтаксис без callback, с использованием асинхронного итератора:

```
async with queue.iterator() as queue_iter:
    async for message in queue_iter:
        print(message.body)
```

sender

- нужно вызвать функцию *connect* для создания подключения к Rabbit
- получить *channel*
- проверить, создана ли очередь с помощью функции *declare_queue*. Если мы отправим сообщение в несуществующую очередь, RabbitMQ просто удалит это сообщение
- отправить сообщение с помощью функции *channel.default_exchange.publish*. В RabbitMQ сообщение не может быть отправлено напрямую в очередь, оно всегда должно проходить через *exchange,* о чем поговорим в следующей главе. Все, что нам нужно знать сейчас — как использовать обмен по умолчанию, идентифицируемый пустой строкой *(default_exchange).* Этот обмен особенный: он позволяет точно указать, в какую очередь должно идти сообщение. Имя очереди необходимо указать в параметре routing_key.

Задание для самопроверки

Разделите пример на 2 разных файла. В одном должна быть отправка сообщений в очередь, в другом должно быть принятие сообщений из очереди.

sender должен запускаться, класть в очередь сообщения и завершаться.

receiver должен запускаться и ожидать задачи из очереди, то есть быть «демоном». Если вы не хотите использовать слово «демон», такие виды программ можно называть «сервисы», как в Windows.

Нужно попробовать:

- запустить несколько раз sender без receiver. Затем запустить receiver и увидеть, что выполнится сразу несколько задач
- запустить receiver и затем запускать sender. Сообщения должны появляться в то же время, в какое произошла отправка
- запустить несколько receiver и начать отправлять сообщения с помощью sender. Сообщения будут появляться по очереди: сначала в первом receiver, потом во втором. Такой алгоритм распределения называется round robin и используется RabbitMQ по умолчанию. Он позволяет распределять нагрузку между consumer и гибко масштабировать приложения. Если очередь начала увеличиваться, можно запустить несколько consumer дополнительно, чтобы ее разобрать.

Задание нужно для того, чтобы понять, что RabbitMQ хранит сообщения и позволяет организовать межпроцессорное взаимодействие. Более того, процессы могут находиться на разных серверах.