**Модуль 8: NoSQL бази даних**

**Заняття 2: Redis. Брокер повідомлень RabbitMQ**

**Redis**

База даних **[Redis](https://redislabs.com/lp/python-redis/" \t "_blank)** (https://developer.redis.com/develop/python/)дуже швидка, тому що використовує для своєї роботи тільки оперативну пам'ять (RAM) і зберігає там усі дані. Це, звичайно, може бути проблемою, якщо зникло живлення на сервері і потрібно відновити стан, але Redis вміє зберігати періодично свій стан на жорсткий диск, тому це зменшує ефект від втрати даних.

Redis має багато можливостей. Сценарій, що найчастіше використовується, — це кешування даних (кеш). Справа в роботі алгоритму TTL (Time To Live), який вбудований у Redis і дозволяє встановлювати для будь-якого запису час "життя" (TTL). Щойно проходить вказаний час, запис автоматично видаляється. Таким чином, розробник може використовувати Redis як тимчасове сховище даних для зменшення навантаження на реляційну базу даних (RDb) або сервер. При цьому не потрібно замислюватися над тим, як інвалідувати кеш.

INFO

Інвалідація кеша - це процес видалення всіх кешованих об'єктів, пов'язаних зі змінами у стані вашої моделі. Найпоширенішим типом інвалідації є пряме видалення об'єктів.

Інша вбудована особливість Redis — це встановлення обмеження на розмір займаної RAM. Використовуючи це налаштування, можна вказати Redis зберігати записи доти, доки є місце (ліміт пам'яті не вичерпаний). Після вичерпання ліміту Redis автоматично видаляє найстаріші записи, щоб записати нові.

База даних підтримує наступні типи даних:

* рядки
* словники (hashes)
* списки
* множини
* сортовані множини.

Як ключі використовуються рядки. Якщо ви спробуєте використати якийсь інший тип даних як ключ, то Redis автоматично приведе цей тип до рядка. Про це потрібно пам'ятати, оскільки це неявне приведення типів може бути несподіваним.

Запустити Redis локально можна за допомогою офіційного [Docker-образу](https://hub.docker.com/_/redis) (https://hub.docker.com/\_/redis).

Щоб працювати з Redis із Python, потрібно встановити один із пакетів для роботи, наприклад [redis](https://pypi.org/project/redis/) (https://pypi.org/project/redis/):

pip install redis

Приклад створення з'єднання:

import redis

r = redis.Redis(host="localhost", port=6379, password=None)

Приклад запису рядка в базу Redis - метод set, та читання за ключем - метод get:

r.set('foo', 'bar')

value = r.get('foo')

print(value) # bar

Сприймайте Redis як великий словник, де як ключі завжди рядки. Пари ключ-значення можуть автоматично видалятися за деякою заданою умовою (вийшов час, закінчилося місце). Значення можуть бути рядками, числами, словниками, множинами.

Крім методів set та get, Redis має великий набір вбудованих методів для маніпуляції даними: обмеження на розмір списку, вставлення зліва або справа, інкремент на n, декремент на n і [багато інших](https:// redis.readthedocs.io/en/latest/commands.html).

Також важливо відзначити, що в Redis "з коробки" реалізовано паттерн проектування [Publish/Subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) (https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe\_pattern).

Коли кілька клієнтів (Subscribers) можуть підключитися до Redis, чекаючи на події, і Redis автоматично надішле всім клієнтам події, коли вони настануть.

Настання події контролює інший тип клієнта (Publisher), який відправляє на сервер повідомлення про подію і сервер (Redis) вже подбає, щоб усі зацікавлені клієнти отримали повідомлення.

**Кешування**

Найчастіше Redis використовують для кешування. Щоб реалізувати [LRU кеш](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)) за допомогою бази даних redis, розгорнемо докер контейнер наступною командою в терміналі.

docker run --name redis-cache -d -p 6379:6379 redis

Перевіримо командою docker ps, що контейнер вдало запустився.

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

55eb7355e280 redis:latest "docker-entrypoint.s…" 3 hours ago Up 3 hours 0.0.0.0:6379->6379/tcp redis-cache

Як відомо, необхідно, щоб кеш не збільшувався дуже сильно, і бажано мав закінчення терміну дії. Це можна реалізувати самостійно, розібравшись як реалізувати [кешування в Python](https://realpython.com/lru-cache-python/), а потім розібратися з реалізацією LRU через redis за допомогою цієї статті ["Using Redis in Python LRU cache"](https://medium.com/@thebestchef/using-redis-in-python-lru-cache-56594df3582c). Але ми не винаходитимемо велосипед і використаємо модуль redis-lru.

ВИНАЙТИ ВЕЛОСИПЕД

Намагатися придумати нове рішення в тому випадку, коли вже є надійне, перевірене рішення

Встановимо цей модуль у корені проекту:

pip install redis-lru

Тепер наведемо приклад реалізації. Виконаємо кешування обчислення функції

import redis

from redis\_lru import RedisLRU

client = redis.StrictRedis(host="localhost", port=6379, password=None)

cache = RedisLRU(client)

@cache

def f(x):

print(f"Function call f({x})")

return x

if \_\_name\_\_ == '\_\_main\_\_':

print(f"Result f(3): {f(3)}")

print(f"Result f(3): {f(3)}")

Функція f обгортається декоратором @cache, який і зберігає значення функції протягом 15 хвилин всередині кеша. Ми бачимо, що виклик функції відбувся один раз, при повторному зверненні значення було взято з кеша.

Виведення:

Function call f(3)

Result f(3): 3

Result f(3): 3

Тривалість зберігання даних у кеші можна встановити параметром default\_ttl при виклику RedisLRU. Розмір кешу визначається параметром max\_size.

Живий приклад, запустіть та перевірте код:

NOTE

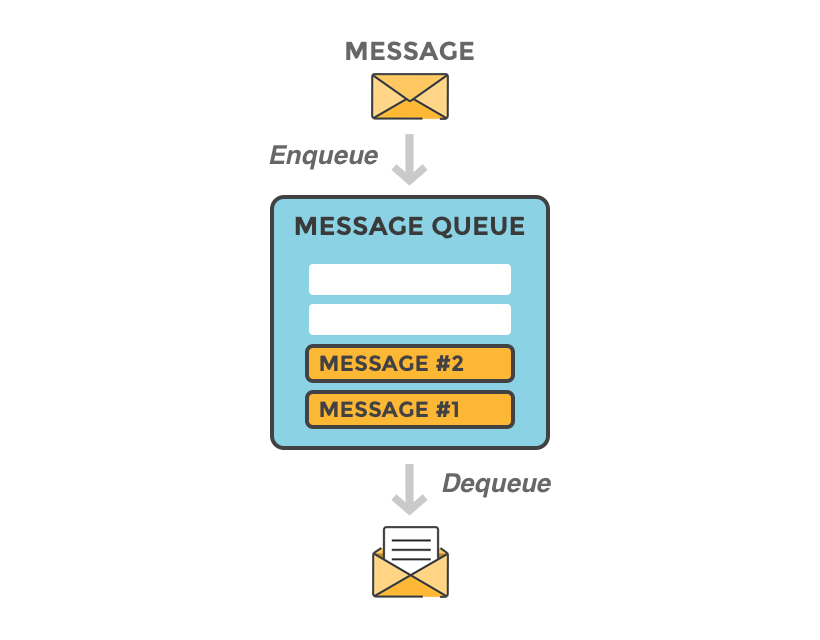
В живому прикладі ви можете взагалі не побачити повідомлення Function call f(3), оскільки хтось вже запустив приклад і помістив результат виконання функції f у кеш.

**Брокер повідомлень RabbitMQ**

[**RabbitMQ**](https://www.rabbitmq.com/) - це програмне забезпечення для черги повідомлень, також відоме як брокер повідомлень або менеджер черг. Простіше кажучи — це програмне забезпечення, де визначаються черги, до яких підключаються застосунки для передачі повідомлення або повідомлень.

Це дуже потужний інструмент для обробки подій, який реалізує гнучкий протокол обміну повідомленнями [**AMQP**](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol)**.**

Повідомлення може містити будь-яку інформацію. Програмне забезпечення диспетчера черг зберігає повідомлення доти, доки приймаючий застосунок не підключиться і не прийме повідомлення позачергово. Потім приймаючий застосунок обробляє повідомлення.



**Повідомлення RabbitMQ**

Основні можливості:

* підтвердження доставки повідомлення клієнту або підтвердження обробки клієнтом повідомлення;
* підтвердження отримання повідомлення на сервер (повідомлення збереглося у черзі і не може загубитися);
* відправлення повідомлень у чергу на зберігання та подальшу обробку;
* відправлення повідомлення в кілька черг за деяким правилом;
* складні правила пересилання повідомлень.

INFO

RabbitMQ — досить надійний інструмент і, якщо адреса (правило кому відправляти) існує, то RabbitMQ гарантує доставку повідомлення.

Найчастіше брокер повідомлень виступає посередником для різних послуг,наприклад, для веб-застосунків. Він може бути використаний для зниження навантажень на основний сервер застосунку, шляхом делегування завдань іншому сервісу. Як правило, такі завдання займають багато часу або ресурсів в основного сервера і віддають їх на обробку іншій стороні.



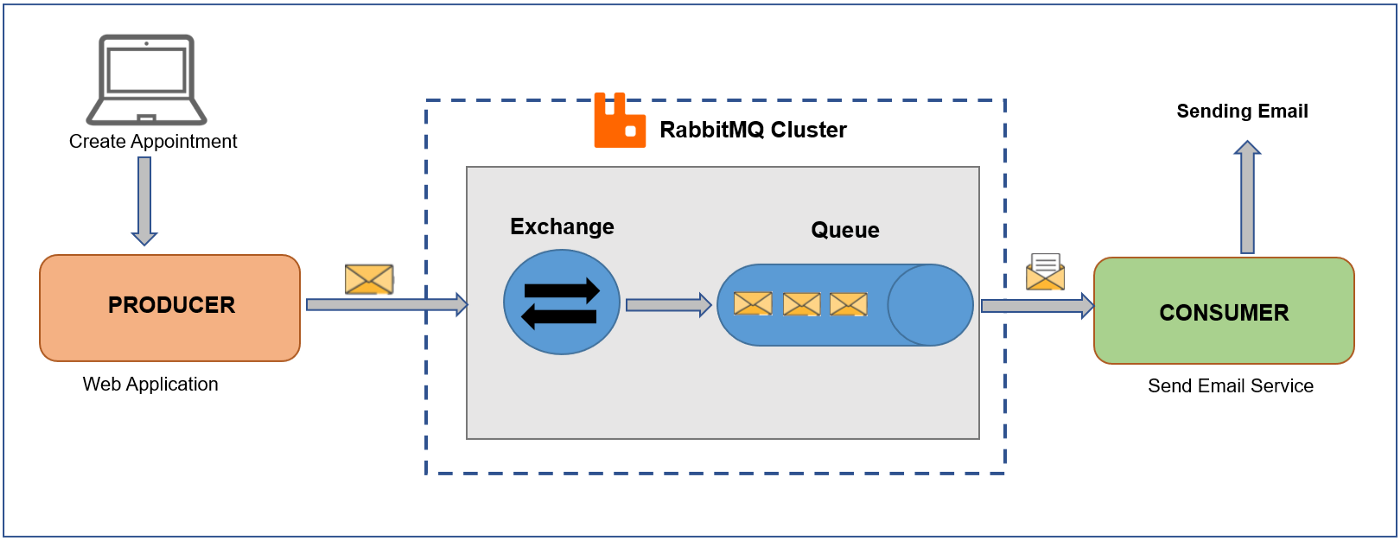
**Базова архітектура RabbitMQ**

Базова архітектура черги повідомлень проста - є клієнтські застосунки, що називаються**Producer** (виробниками), які створюють повідомлення і доставляють їх в **Broker**(черга повідомлень). Інші застосунки, які називаються **Consumer**(споживачами), підключаються до черги та підписуються на обробку повідомлень. Повідомлення, поміщені у чергу, зберігаються доти, доки споживач не отримає їх.

**Приклад роботи RabbitMQ**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_01#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4-%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B8-rabbitmq)

Черга повідомлень дозволяє веб-серверам швидко реагувати на запити, замість того, щоб бути змушеним виконувати ресурсомісткі операції та затримувати час відгуку для клієнта. Черга повідомлень також хороша для поширення повідомлень серед кількох споживачів або балансування навантаження між працівниками.



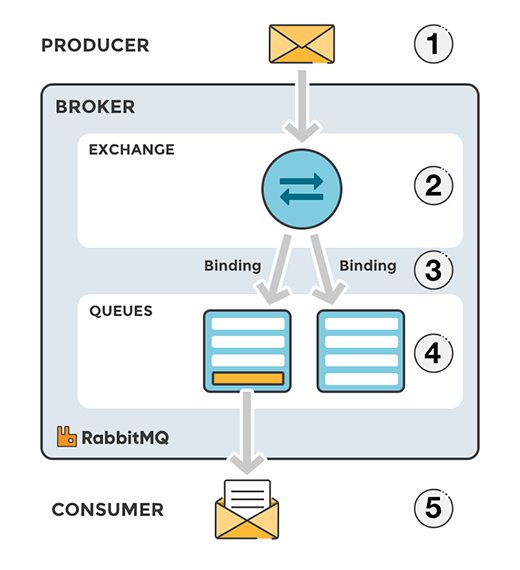
**Приклад застосування RabbitMQ**

1. Користувач реєструється у вебзастосунку, йому повинен прийти email з підтвердженням реєстрації. Але веб-застосунок відразу формує відгук для користувача, він не чекає завершення відправлення email, а, наприклад, перенаправляється у свій профіль.
2. Веб застосунок (виробник) відправляє повідомлення в RabbitMQ, яке включає дані: ім'я та адресу електронної пошти користувача.
3. **Exchange** (біржа) приймає повідомлення від виробника та відправляє їх у чергу повідомлень (Queue) для відправлення email.
4. **Consumer** (споживач) отримує повідомлення із завданням із черги (Queue) та починає відправлення email.

**Exchange**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_01#exchange)

Як бачимо повідомлення не публікуються безпосередньо в черзі, натомість виробник відправляє повідомлення на **Exchange** (біржу або обмінник). Exchange відповідає за маршрутизацію повідомлення у різні черги за допомогою прив'язок (binding) та ключів маршрутизації. Прив'язка – це зв'язок між чергою та біржею.



**Потік повідомлень в RabbitMQ**

Таким чином потік повідомлень у RabbitMQ відбувається так:

1. Виробник публікує повідомлення на біржі. Під час створення біржі необхідно вказати її тип (див. нижче).
2. Біржа отримує повідомлення і тепер відповідає за маршрутизацію повідомлення. Біржа враховує різні атрибути повідомлень, такі як ключ маршрутизації, залежно від типу біржі.
3. Прив'язки повинні створюватися з біржі у черзі. На ілюстрації дві прив'язки з біржі до двох різних черг. Біржа надсилає повідомлення у черзі, залежно від атрибутів повідомлення.
4. Повідомлення залишаються в черзі доти, доки вони не будуть оброблені споживачем.
5. Споживач обробляє повідомлення.

Типи бірж:

* direct: Повідомлення надсилається у черзі, ключ прив'язки яких точно збігається з ключем маршрутизації повідомлення.
* fanout: Цей тип бірж направляє повідомлення на всі черги, прив'язані до нього.
* topic: Біржа цього типу виконує зіставлення підстановних знаків між ключем маршрутизації та шаблоном маршрутизації, зазначеного у прив'язці.
* headers: Біржа використовує атрибути заголовків повідомлень для маршрутизації.

У прикладах ми переважно використовуватимемо тип бірж direct.

**Початок роботи з RabbitMQ і Python**

Перш ніж ми перейдемо до прикладів коду, давайте визначимося з термінологією RabbitMQ:

* **Producer:** Застосунок, який надсилає повідомлення.
* **Consumer:** Застосунок, який отримує повідомлення.
* **Queue:** Черга, в якій зберігаються повідомлення.
* **Message:** Інформація, яка відправляється від Producer до Consumer через RabbitMQ.
* **Connection:** TCP-з'єднання між застосунком та брокером RabbitMQ.
* **Channel:** Канал. Віртуальне підключення всередині Connection. Під час публікації або використання повідомлень з черги - все це виконується каналом.
* **Exchange:** Біржа. Отримує повідомлення від Producer та відправляє їх у Queue, залежно від свого типу. Для отримання повідомлень необхідно, щоб Queue була прив'язана принаймні до однієї біржі.
* **Binding:** Прив'язка – це зв'язок між чергою (Queue) та біржею (Exchange).
* **Routing key:** Ключ, який переглядає біржа, щоб вирішити, як спрямовувати повідомлення в чергу. Думайте про ключі маршрутизації як про адресу повідомлення.
* **AMQP:** Розширений протокол черги повідомлень — це протокол, який використовується RabbitMQ для обміну повідомленнями.

INFO

Робота з RabbitMQ дуже детально описана на офіційній сторінці з прикладами на Python.

Встановити локально сервіс RabbitMQ можна за допомогою [Docker-образу](https://hub.docker.com/_/rabbitmq/).

docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.11-management

NOTE

Якщо під час навчання виникнуть проблеми зі встановленням RabbitMQ через Docker контейнер, то як заміну можна використовувати хмарний сервіс CloudAMQP

Для роботи з RabbitMQ в Python необхідно використовувати пакет [**pika**](https://pika.readthedocs.io/en/stable/):

pip install pika

**Базовий приклад**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#%D0%B1%D0%B0%D0%B7%D0%BE%D0%B2%D0%B8%D0%B9-%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4)

**Producer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#producer)

Розглянемо найпростіший приклад сервера надсилання повідомлень, як завжди, це буде Hello world. Почнемо з Producer

**producer.py**

import pika

def main():

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

channel.queue\_declare(queue='hello\_world')

channel.basic\_publish(exchange='', routing\_key='hello\_world', body='Hello world!'.encode())

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

connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Спочатку нам потрібно створити з'єднання з RabbitMQ:

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

За замовчуванням користувач для Docker-контейнера має нікнейм guest з паролем guest. В результаті в змінній channel ми отримуємо **Channel.**

Далі потрібно створити чергу повідомлень і дамо їй ім'я 'hello\_world':

channel.queue\_declare(queue='hello\_world')

RabbitMQ проігнорує повідомлення, якщо воно відправлене в неіснуючу чергу. Далі надсилаємо саме повідомлення:

channel.basic\_publish(exchange='', routing\_key='hello\_world', body='Hello world!'.encode())

Тут ми вказали exchange порожнім рядком, у цьому випадку RabbitMQ буде використовувати біржу за замовчуванням (**AMQP default**) типу direct. Повідомлення надсилаються в чергу, ім'я якої суворо збігається з routing\_key.

Після відправлення потрібно обов'язково закрити з'єднання:

connection.close()

**Consumer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#consumer)

Отримання повідомлень — трохи складніший процес. Справа в тому, що pika реалізує реактивний підхід в обробці повідомлень. Передбачається, що клієнт постійно в режимі очікування нових повідомлень та обробки тих, що надійшли. Для цього точно потрібно підключитися до брокера,

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(

pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

створити чергу (повторно створювати чергу можна, нічого не зламається),

channel.queue\_declare(queue='hello\_world')

зареєструвати функцію для обробки повідомлення та викликати блокуючий метод start\_consuming, який отримуватиме повідомлення та надсилатиме їх у функцію зворотного виклику - callback. Щоразу, коли ми отримуємо повідомлення, ця функція зворотного виклику викликається бібліотекою pika. У нашому випадку ця функція друкуватиме на екрані вміст повідомлення.

def callback(ch, method, properties, body):

print(f" [x] Received {body}")

channel.basic\_consume(queue='hello\_world', on\_message\_callback=callback, auto\_ack=True)

print(' [\*] Waiting for messages. To exit press CTRL+C')

channel.start\_consuming()

У методі start\_consuming ми додали параметр auto\_ack=True — прапор автоматичного підтвердження всіх повідомлень, що відправляються (не вимагає відповіді від Consumer) - працює швидко, але не дає жодних гарантій успішної обробки повідомлень. У наступному прикладі ми розберемо момент із підтвердженням доставки.

Повний код **Consumer** представлений нижче.

**consumer.py**

import pika

import sys

def main():

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(

pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

channel.queue\_declare(queue='hello\_world')

def callback(ch, method, properties, body):

print(f" [x] Received {body}")

channel.basic\_consume(queue='hello\_world', on\_message\_callback=callback, auto\_ack=True)

print(' [\*] Waiting for messages. To exit press CTRL+C')

channel.start\_consuming()

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except KeyboardInterrupt:

print('Interrupted')

sys.exit(0)

Функція callback повинна обов'язково приймати 4 аргументи:

* ch — поточний канал комунікації (цей об'єкт може перервати виконання циклу всередині start\_consuming, якщо потрібно);
* method — детальна інформація про повідомлення;
* properties — службова інформація про повідомлення;
* body — тіло повідомлення у форматі bytes рядка.

NOTE

Передбачається, що RabbitMQ встановлений та запущений на localhost на стандартному порту 5672.

Запустимо **Consumer**і в терміналі повинні побачити наступні повідомлення:

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

**Consumer** у режимі очікування та чекає повідомлення. Після цього виконаємо **Producer**:

[x] Sent 'Hello World!'

Термінал **Consumer** тепер повинен виглядати так:

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

[x] Received b'Hello world!'

Як бачимо, ми успішно отримали повідомлення b'Hello world!'

**Приклад роботи з чергою завдань**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4-%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B8-%D0%B7-%D1%87%D0%B5%D1%80%D0%B3%D0%BE%D1%8E-%D0%B7%D0%B0%D0%B2%D0%B4%D0%B0%D0%BD%D1%8C)

Однією з переваг використання черги завдань є можливість легко розпаралелювати роботу. Ми можемо просто додати більше працівників і таким чином легко масштабуватись.

**Producer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#producer-1)

Цього разу як **Producer** у нас виступатиме скрипт create\_tasks.py, який створюватиме і відправлятиме у RabbitMQ 5 умовних завдань:

**create\_tasks.py**

import pika

from datetime import datetime

import sys

import json

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(

pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

channel.exchange\_declare(exchange='task\_mock', exchange\_type='direct')

channel.queue\_declare(queue='task\_queue', durable=True)

channel.queue\_bind(exchange='task\_mock', queue='task\_queue')

def main():

for i in range(5):

message = {

"id": i + 1,

"payload": f"Task #{i + 1}",

"date": datetime.now().isoformat()

}

channel.basic\_publish(

exchange='task\_mock',

routing\_key='task\_queue',

body=json.dumps(message).encode(),

properties=pika.BasicProperties(

delivery\_mode=pika.spec.PERSISTENT\_DELIVERY\_MODE

))

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

connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Тут ми визначаємо біржу task\_mock типу direct

channel.exchange\_declare(exchange='task\_mock', e

xchange\_type='direct')

Декларуємо чергу task\_queue та зв'язуємо її з нашою біржею task\_mock.

channel.queue\_declare(queue='task\_queue', durable=True)

channel.queue\_bind(exchange='task\_mock', queue='task\_queue')

Коли RabbitMQ завершує роботу звичайним чином або аварійно, він забуває про черги та повідомлення. Щоб такої ситуації не відбувалося, нам необхідно вказати прапор durable=True під час визначення черги.

Ми формуємо якийсь словник message і додаємо до нього імітацію корисного навантаження. І відправляємо повідомлення на біржу, попередньо запакувавши його в байт-рядок json.dumps(message).encode():

channel.basic\_publish(

exchange='task\_mock',

routing\_key='task\_queue',

body=json.dumps(message).encode(),

properties=pika.BasicProperties(

delivery\_mode=pika.spec.PERSISTENT\_DELIVERY\_MODE

))

Під час надсилання через параметр properties вказуємо delivery\_mode для кожного повідомлення - так звана *ознака персистентності* - це означає, що повідомлення буде збережено на диску і не зникне у разі перезавантаження RabbitMQ.

* delivery\_mode=pika.spec.TRANSIENT\_DELIVERY\_MODE - не зберігати повідомлення та працює швидше.
* delivery\_mode=pika.spec.PERSISTENT\_DELIVERY\_MODE - зберігати повідомлення на диску, повільніше, але надійніше, та використовується майже завжди.

**Consumer**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_02#consumer-1)

Як Consumer буде скрипт worker.py:

**worker.py**

import pika

import time

import json

credentials = pika.PlainCredentials('guest', 'guest')

connection = pika.BlockingConnection(

pika.ConnectionParameters(host='localhost', port=5672, credentials=credentials))

channel = connection.channel()

channel.queue\_declare(queue='task\_queue', durable=True)

print(' [\*] Waiting for messages. To exit press CTRL+C')

def callback(ch, method, properties, body):

message = json.loads(body.decode())

print(f" [x] Received {message}")

time.sleep(1)

print(f" [x] Done: {method.delivery\_tag}")

ch.basic\_ack(delivery\_tag=method.delivery\_tag)

channel.basic\_qos(prefetch\_count=1)

channel.basic\_consume(queue='task\_queue', on\_message\_callback=callback)

if \_\_name\_\_ == '\_\_main\_\_':

channel.start\_consuming()

Цей код відрізняється від попереднього прикладу та вимагає додаткових пояснень. Виконання завдання може зайняти кілька секунд і можна поставити питання, що станеться, якщо споживач починає довге завдання, і воно завершується до його завершення. У попередньому прикладі, коли RabbitMQ доставляє повідомлення споживачеві, він негайно позначає його видалення. У разі, якщо Consumer не обробив повідомлення, воно втрачено. Але ми не хочемо втрачати жодних завдань, ми хочемо, щоб завдання було передано іншому працівникові (Consumer).

Щоб переконатися, що повідомлення ніколи не буде втрачено, RabbitMQ підтримує механізм підтвердження повідомлень. Якщо споживач (Consumer) помирає з якоїсь причини без надсилання підтвердження, RabbitMQ розуміє, що повідомлення не було опрацьовано повністю, і повторно ставить його в чергу. Якщо у цей час є інші споживачі, він передає це повідомлення іншому споживачеві. Таким чином, можна бути впевненим, що жодне повідомлення не буде втрачено, навіть якщо робітники (Consumer) час від часу вмирають.

INFO

За замовчуванням тайм-аут підтвердження доставки споживачеві дорівнює 30 хвилин. Це допомагає виявити застряглих споживачів, які ніколи не повернуть підтвердження доставки.

Тому рядок коду ch.basic\_ack(delivery\_tag=method.delivery\_tag) наприкінці функції callback відповідає за відправлення відповідного підтвердження від працівника, як тільки виконання завдання завершиться.

Також у коді з'явився рядок channel.basic\_qos(prefetch\_count=1) — це так зване чесне відправлення. Параметр prefetch\_count=1 каже, щоб RabbitMQ не надсилав нове повідомлення для працівника доти, доки він не обробить і не підтвердить обробку попереднього повідомлення. Натомість він відправить його наступному працівникові, який ще не зайнятий.

Тепер можемо запустити **Producer** create\_tasks.py:

[x] Sent {'id': 1, 'payload': 'Task #1', 'date': '2022-12-02T22:23:48.452124'}

[x] Sent {'id': 2, 'payload': 'Task #2', 'date': '2022-12-02T22:23:48.452124'}

[x] Sent {'id': 3, 'payload': 'Task #3', 'date': '2022-12-02T22:23:48.453125'}

[x] Sent {'id': 4, 'payload': 'Task #4', 'date': '2022-12-02T22:23:48.453125'}

[x] Sent {'id': 5, 'payload': 'Task #5', 'date': '2022-12-02T22:23:48.453125'}

А потім запустимо для обробки повідомлень **Consumer** worker.py:

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

[x] Received {'id': 1, 'payload': 'Task #1', 'date': '2022-12-02T22:23:48.452124'}

[x] Done: 1

[x] Received {'id': 2, 'payload': 'Task #2', 'date': '2022-12-02T22:23:48.452124'}

[x] Done: 2

[x] Received {'id': 3, 'payload': 'Task #3', 'date': '2022-12-02T22:23:48.453125'}

[x] Done: 3

[x] Received {'id': 4, 'payload': 'Task #4', 'date': '2022-12-02T22:23:48.453125'}

[x] Done: 4

[x] Received {'id': 5, 'payload': 'Task #5', 'date': '2022-12-02T22:23:48.453125'}

[x] Done: 5

Як бачимо, все чудово працює.

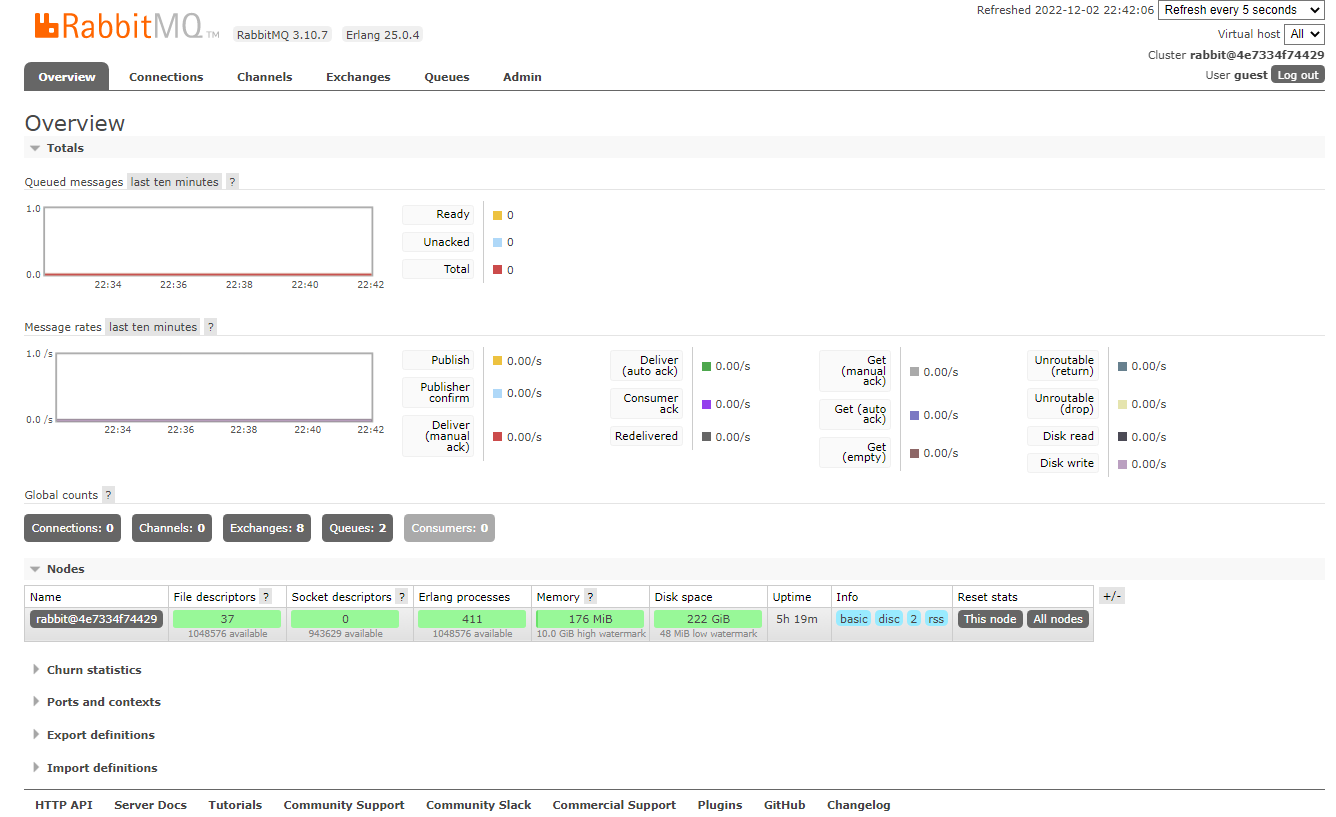
**Інтерфейс управління RabbitMQ**

RabbitMQ надає веб-інтерфейс для управління та моніторингу Сервер RabbitMQ — плагін **RabbitMQ Management .** З інтерфейсу управління можна обробляти, створювати, видаляти списки черг. Також можна відстежувати довжину черги, перевіряти швидкість повідомлень, змінювати та додавати дозволи користувачів та багато іншого. Після запуску Docker-контейнера можна авторизуватися в [веб-інтерфейсі](http://localhost:15672/), де логін і пароль за замовчуванням - guest/guest

INFO

RabbitMQ Management - це плагін, який можна увімкнути для RabbitMQ. Коли ми запускали Docker-контейнер, то слово management у назві образу rabbitmq:3.11-management якраз і каже, що плагін включений до складу образу. Він пропонує набір статичних HTML-сторінок, які виконують фонові запити в HTTP API для RabbitMQ. Інформація з інтерфейсу управління може бути корисною, коли виконується відлагодження застосунків або коли вам потрібен огляд усієї системи. Якщо ви бачите, що кількість неприкріплених повідомлень починає збільшуватись, це може означати, що ваші споживачі стають повільними. Якщо потрібно перевірити, чи працює біржа, можна спробувати відправити тестове повідомлення.

**Головна сторінка Overview**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BD%D0%B0-%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-overview)



**Головна сторінка Overview**

Зверху праворуч - ім'я кластера@сервера rabbit@4e7334f74429, ім'я користувача guest. Зверху - версія rabbit та erlang'а, трохи нижче - вкладки

* Overview - загальні дані всього кластера (у цьому разі кластер з однієї ноди)
* Connections – відомості про з'єднання
* Channels - відомості про канали
* Exchanges - відомості про біржі
* Queues - відомості про черги
* Admin – функції адміністрування

Два верхні графіки - загальна кількість повідомлень у всіх чергах у кластері

Queued messages - кількісний показник

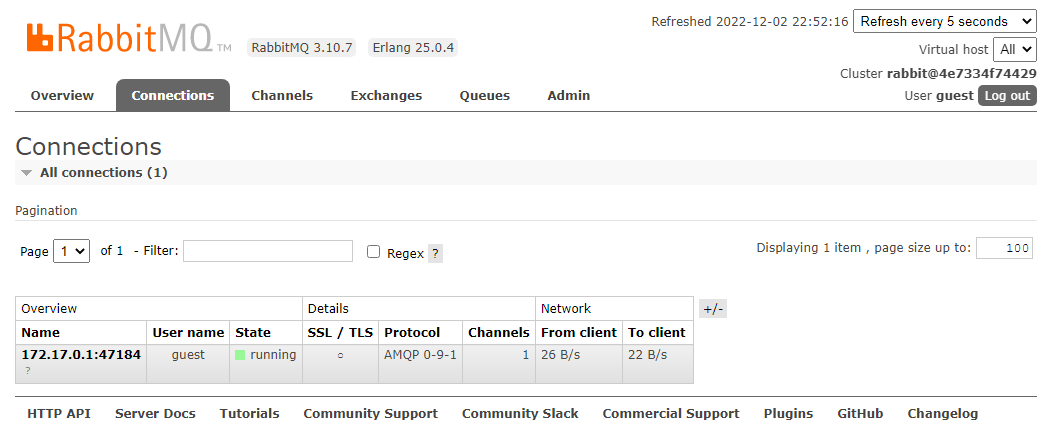
* Ready - Повідомлення, що очікують обробки
* Unacked - Повідомлення, передані Consumer на обробку і чекають підтвердження.
* Total - Сума Reade+Unacked

Message rates - показник швидкості обробки

* Publish - Швидкість надсилання, запису повідомлень у черзі
* Consumer ack - Швидкість підтвердження обробки повідомлень Consumer
* Redelivered - Швидкість повернення повідомлень у черзі у разі неуспішної обробки (nack)

**Сторінка Connections**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-connections)

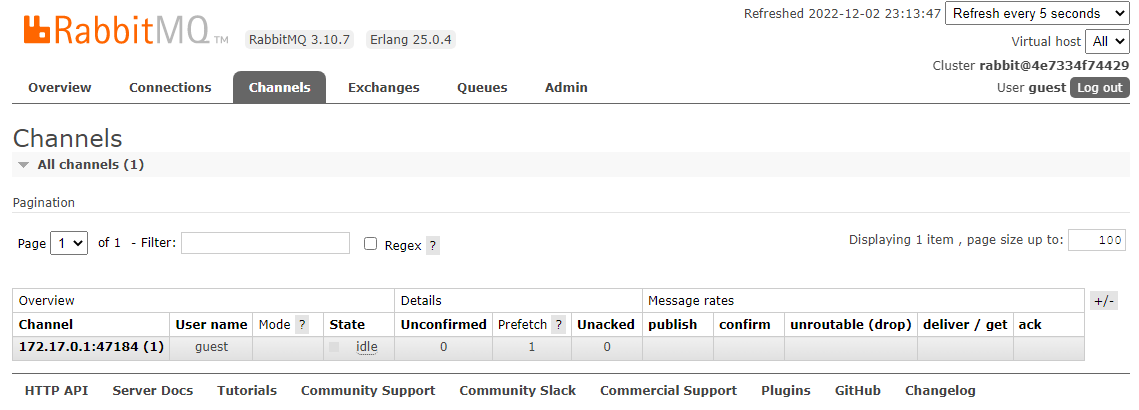


**Сторінка Connections**

Сторінка зі списком з'єднань. На скріншоті з'єднання Consumer з останнього прикладу. Всередині кожного з'єднання можна переглянути індивідуальні графіки та параметри, також там можна примусово завершити з'єднання.

**Сторінка Channels**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-channels)

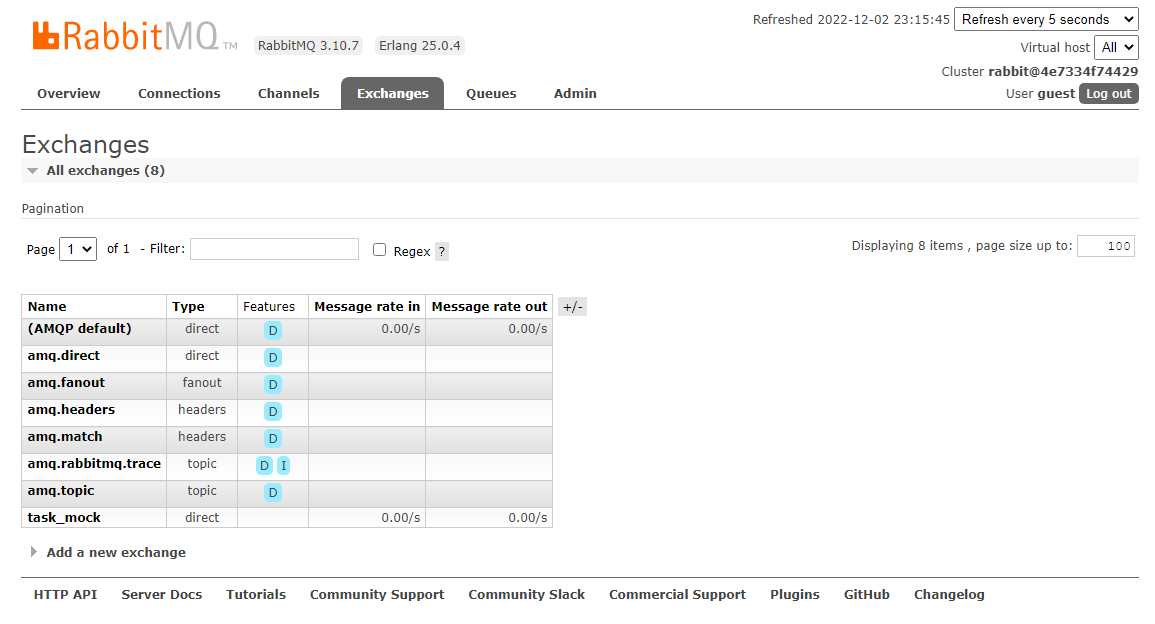


**Сторінка Channels**

Сторінка зі списком каналів. Все аналогічно до списку з'єднань, тільки по суті "канал". Всередині аналогічно можна переглянути подробиці по кожному каналу

**Сторінка Exchanges**

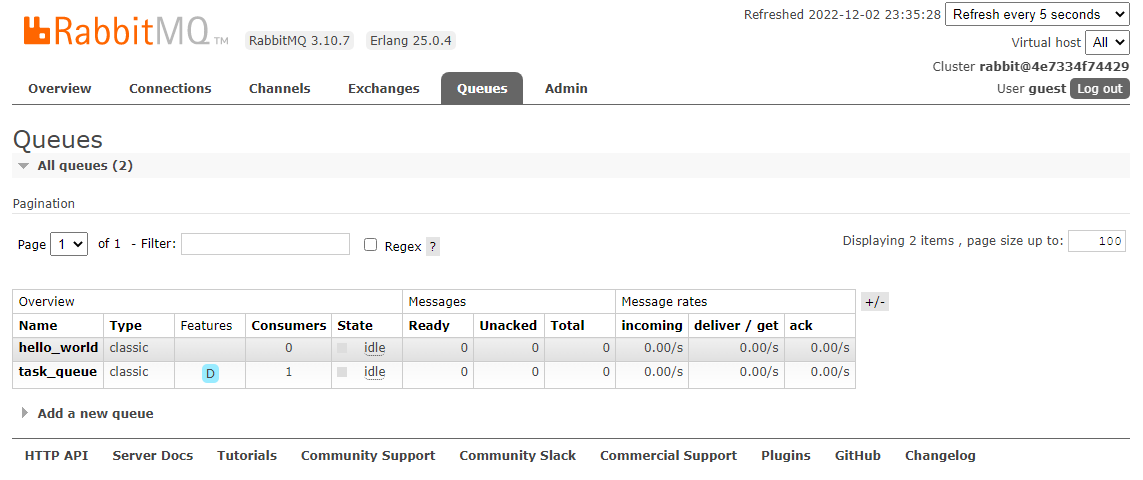
[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-exchanges)



**Сторінка Exchanges**

Сторінка зі списком exchanges. Бачимо, як службові exchanges, так і створені програмно task\_mock. Також на цій сторінці можна створити exchange вручну. Зайшовши всередину exchange, можна подивитися навантаження, подивитися/створити/видалити binding до черг. Також там можна опублікувати повідомлення в exchange, вказавши при цьому потрібний routing key. При цьому delivery\_mode повідомлень дорівнюватиме 2 - persistent.

**Сторінка Queues**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-queues)



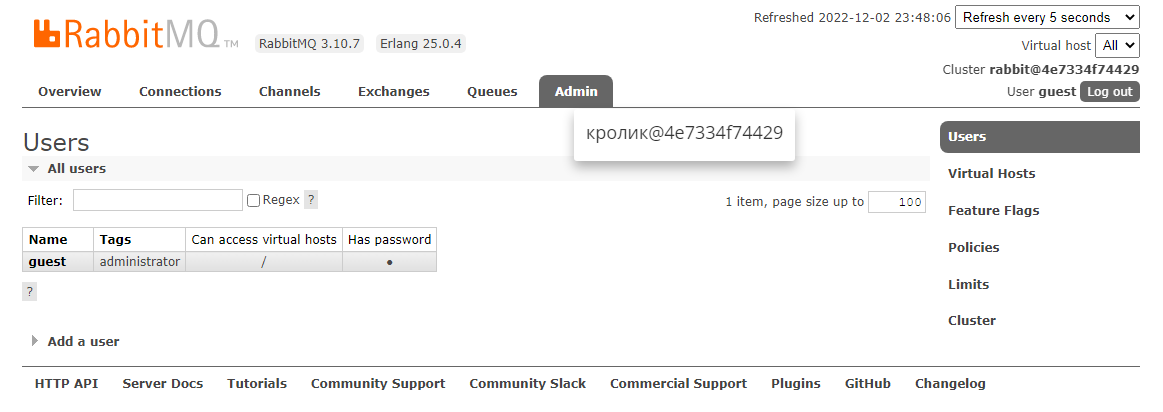
**Сторінка Queues**

Сторінка зі списком черг. Рекомендується додати стовпчик consumers - показує кількість підключених до черги Consumers, додається через кнопку +/- праворуч від таблиці. Внизу інтерфейс створення нової черги. Якщо зайти в саму чергу, ми побачимо графіки за кількістю повідомлень у конкретній черзі і за швидкістю публікації/обробки повідомлень.

**Сторінка Admin**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-08/module-08-02/rabbit_03_03#%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0-admin)

Вкладка для адміністрування деяких параметрів RabbitMQ



**Сторінка Admin**

Зокрема:

* Users - управління користувачами та їх правами;
* Virtual Hosts – управління віртуальними хостами - це ізольовані оточення всередині одного кластера;
* Feature Flags - відображення увімкненого функціоналу RabbitMQ
* Policies - можливість додавати атрибути для черг та бірж без повторного їх оголошення.
* Limits - налаштування обмежень
* Cluster - параметри кластера, але фактично можна змінити тільки назву кластера, наприклад, наш rabbit@4e7334f74429 на деяке осмислене значення.

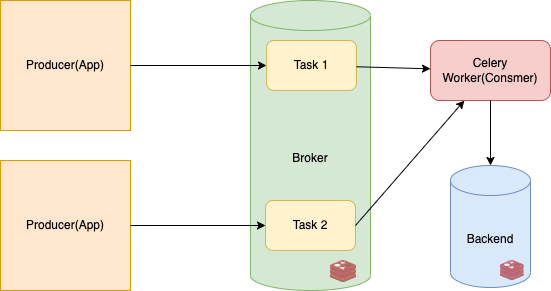
**Основи роботи з Celery**

Налаштування брокерів повідомлень (*RabbitMQ* або *Redis*) може бути достатньо нетривіальним завданням і неспецифічний синтаксис інструментів "протікає" в Python код. Щоб зменшити прив'язку до конкретного інструменту, можна скористатися інструментом, який створює додатковий шар абстракції поверх "голого" брокера. Таким інструментом можна вважати [**Celery**](https://docs.celeryproject.org/en/stable/getting-started/first-steps-with-celery.html)**.**

INFO

Celery може використовувати RabbitMQ або Redis для обміну повідомленнями та надає зручні інтерфейси для надсилання завдань на виконання "працівниками" (worker).

У Celery може бути багато варіантів використання. Це надсилання електронних листів, завантаження зображень або інших видів документів через веб-сторінку та подальша обробка цих документів. Також Celery може виступати як планувальник і може використовуватися для виконання періодичних завдань.



**Базова архітектура Celery**

**Producer**— Цей застосунок відповідає за надсилання повідомлень з усією необхідною інформацією.

**Broker**— цей модуль фактично слугує чергою повідомлень, тут можна використовувати такі застосунки, як Redis або RabbitMQ.

**Task** — це функція або завдання Python, яка ставиться в чергу у брокері після серіалізації. Потім функція завдання вибирається працівником (worker), який відповідає за її десеріалізацію та подальше виконання. Формат серіалізації за замовчуванням — JSON, який ви можете змінити на msgpack, YAML або pickle.

**Backend** — цей компонент відповідає за збереження результату, отриманого **worker.**

Для початку створимо застосунок Celery.

Нам знадобиться брокер, ми будемо використовувати Redis. Тому використовуємо docker для підняття контейнера Redis.

docker run -d -p 6379:6379 redis

Також встановимо сам Celery:

pip install celery

Тепер у нас встановлений Celery, і база даних Redis готова до приймання даних. Створимо завдання (Task) у файлі mytasks.py.

**mytasks.py**

from celery import Celery

BROKER\_URL = 'redis://localhost:6379/0'

BACKEND\_URL = 'redis://localhost:6379/1'

celery = Celery('tasks', broker=BROKER\_URL, backend=BACKEND\_URL)

@celery.task(name='Add two numbers')

def add(x, y):

return x + y

В якості аргументів ми вказали рядок з'єднання з брокером BROKER\_URL = 'redis://localhost:6379/0', а також рядок для збереження результатів обчислення задачі BACKEND\_URL = 'redis://localhost:6379/1'.

from celery import Celery

BROKER\_URL = 'redis://localhost:6379/0'

BACKEND\_URL = 'redis://localhost:6379/1'

celery = Celery('tasks', broker=BROKER\_URL, backend=BACKEND\_URL)

Далі створимо функцію, яку виконуватиме наш віддалений "працівник" (worker):

@celery.task(name='Add two numbers')

def add(x, y):

return x + y

Тепер ми можемо відправити завдання на виконання на віддаленому сервері, передавши аргументи для задачі x та y. Напишемо скрипт app.py з наступним вмістом:

**app.py**

from mytasks import add

if \_\_name\_\_ == '\_\_main\_\_':

result = add.delay(1, 1)

print(result.id)

Додавання функції буде поставлено в чергу в Redis за допомогою функції delay(), яка викличе функцію apply\_async(), яка в свою чергу серіалізує функцію для подальшої обробки. Тепер запустимо app.py, щоб поставити завдання в чергу. Після запуску він дасть результат, як показано нижче:

bda3267c-f2f2-4579-863b-779fdbd2b8ce

Як бачите, він повертає ідентифікатор завдання. У самому Redis він збереже завдання у вигляді JSON:

{

"body": "W1sxLCAxXSwge30sIHsiY2FsbGJhY2tzIjogbnVsbCwgImVycmJhY2tzIjogbnVsbCwgImNoYWluIjogbnVsbCwgImNob3JkIjogbnVsbH1d",

"content-encoding": "utf-8",

"content-type": "application/json",

"headers": {

"lang": "py",

"task": "Add two numbers",

"id": "bda3267c-f2f2-4579-863b-779fdbd2b8ce",

"shadow": null,

"eta": null,

"expires": null,

"group": null,

"group\_index": null,

"retries": 0,

"timelimit": [

null,

null

],

"root\_id": "bda3267c-f2f2-4579-863b-779fdbd2b8ce",

"parent\_id": null,

"argsrepr": "(1, 1)",

"kwargsrepr": "{}",

"origin": "gen5032@DESKTOP-SNTSLVR",

"ignore\_result": false

},

"properties": {

"correlation\_id": "bda3267c-f2f2-4579-863b-779fdbd2b8ce",

"reply\_to": "da2ac493-af2b-3f10-b127-2b74abd2a502",

"delivery\_mode": 2,

"delivery\_info": {

"exchange": "",

"routing\_key": "celery"

},

"priority": 0,

"body\_encoding": "base64",

"delivery\_tag": "74413d4d-30e5-480d-8da6-9f031fe3550c"

}

}

Щоб запустити "працівника", потрібно виконати в консолі таку команду:

celery -A mytasks worker --loglevel=INFO

CAUTION

Для Windows потрібно вказати додатковий параметр --pool solo. Подробиці дивіться [тут](https://stackoverflow.com/questions/37255548/how-to-run-celery-on-windows)

celery -A mytasks worker --loglevel=INFO --pool solo

Ключ -A використовується для імені застосунку, яка є нашим завданням. Сама назва походить від імені файлу завдання mytasks.py. Потім ми викликаємо працівника разом із журналом логування worker --loglevel=INFO.

Отримаємо наступне виведення у терміналі:

--------------- celery@DESKTOP-SNTSLVR v5.2.7 (dawn-chorus)

--- \*\*\*\*\* -----

-- \*\*\*\*\*\*\* ---- Windows-10-10.0.19044-SP0 2022-11-21 01:26:46

- \*\*\* --- \* ---

- \*\* ---------- [config]

- \*\* ---------- .> app: tasks:0x1e7db3a2760

- \*\* ---------- .> transport: redis://localhost:6379/0

- \*\* ---------- .> results: redis://localhost:6379/1

- \*\*\* --- \* --- .> concurrency: 12 (solo)

-- \*\*\*\*\*\*\* ---- .> task events: OFF (enable -E to monitor tasks in this worker)

--- \*\*\*\*\* -----

-------------- [queues]

.> celery exchange=celery(direct) key=celery

[tasks]

. Add two numbers

[2022-11-21 01:26:46,733: INFO/MainProcess] Connected to redis://localhost:6379/0

[2022-11-21 01:26:46,740: INFO/MainProcess] mingle: searching for neighbors

[2022-11-21 01:26:47,757: INFO/MainProcess] mingle: all alone

[2022-11-21 01:26:47,773: INFO/MainProcess] celery@DESKTOP-SNTSLVR ready.

[2022-11-21 01:26:47,775: INFO/MainProcess] Task Add two numbers[bda3267c-f2f2-4579-863b-779fdbd2b8ce] received

[2022-11-21 01:26:47,778: INFO/MainProcess] Task Add two numbers[bda3267c-f2f2-4579-863b-779fdbd2b8ce] succeeded in 0.0s: 2

Ви можете помітити завдання Task Add two numbers[bda3267c-f2f2-4579-863b-779fdbd2b8ce] разом з ідентифікатором завдання, яке спочатку було отримане received, а потім успішно виконане succeeded in 0.0s: 2 Зверніть увагу на цифру 2, яка є сумою 1і1 (add.delay(1, 1)`).

Оскільки ми також використовуємо бекенд BACKEND\_URL = 'redis://localhost:6379/1', то результат виконання завдання зберігається в Redis з ключем celery-task-meta-bda3267c-f2f2-4579-863b-779fdbd2b8ce, як показано нижче:

{

"status": "SUCCESS",

"result": 2,

"traceback": null,

"children": [],

"date\_done": "2022-11-20T23:26:47.775917",

"task\_id": "bda3267c-f2f2-4579-863b-779fdbd2b8ce"

}

Celery найчастіше використовують для завдань, які вимагають відкладеного виконання, а гнучка архітектура робить його корисним для багатьох цілей.