

Применение брокера сообщений RabbitMQ для выполнения задач в фоновом режиме

Фатеенков Данила Витальевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной статье рассматривается популярный брокер сообщений с открытым исходным кодом RabbitMQ. Описываются его особенности, назначение и принцип работы. Также описан процесс интеграции брокера сообщений с API-сервисом, которое реализовано на языке программирования Python с применением модуля FastAPI. Для работы с брокером сообщений на стороне API-сервиса используется модуль Aio-Pika.

Ключевые слова: брокер сообщений, RabbitMQ, очередь сообщений, Python, Pika, AMQP протокол

Using the RabbitMQ message broker to perform tasks in the background

Fateenkov Danila Vitalievich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article discusses the popular open-source message broker RabbitMQ. It describes its features, purpose and principle of operation. It also describes the process of integrating the message broker with the API-service, which is implemented in the Python programming language using the FastAPI module. To work with the message broker on the API-service side, the Aio-Pika module is used.

Keywords: message broker, RabbitMQ, message queue, Python, Pika, AMQP protocol

1. Введение

1.1 Актуальность

В настоящее время ряд асинхронных задач принято обрабатывать через специальные брокеры сообщений, которые выступают посредниками между источником и приёмником. Такой архитектурный шаблон позволяет значительно ускорить работу приложения за счёт выполнения части задач на фоне во время работы всех остальных процессов.

Актуальность брокеров сообщений в современных системах обусловлена их способностью решать ключевые задачи, связанные с обменом данными, асинхронной обработкой, масштабируемостью и интеграцией

различных компонентов системы. Наиболее популярные брокеры: Kafka, RabbitMQ, ActiveMQ.

RabbitMQ актуален благодаря своей надежности, гибкости, поддержке сложных сценариев маршрутизации и интеграции с современными технологиями. Он продолжает использоваться в микросервисной архитектуре, для фоновой обработки задач, в event-driven системах и многих других сценариях.

1.2 Обзор исследований

А.А. Дмитриева представила исследование зависимости времени обработки сообщений от количества нод в кластере RabbitMQ [1]. Исследование проводилось на 5-и серверах с использованием модуля для нагрузочного тестирования Jmeter.

Г.В. Васильев в своей научной работе представил методику по реализации отказоустойчивой информационной системы для взаимодействий микросервисов с использованием брокера сообщений RabbitMQ [2].

С.В. Шиловский и М.И. Давидовская провели исследование на тему применения системы очередей сообщений в программном обеспечении на основе сервисной архитектуры [3].

Д.А. Золотарев посвятил свою научную работу сравнению популярных брокеров сообщений Apache ActiveMQ Artemis и RabbitMQ [4]. В статье рассматриваются основные характеристики, преимущества и недостатки каждой из систем. Кроме того, в статье описываются функциональные и нефункциональные особенности каждой из систем.

О.В. Натроби́на и В.Ю. Пашечко описали процесс разработки модуля гарантированной доставки сообщений в брокере RabbitMQ для встраивания его в веб-приложения, написанные на .NET Core с использованием микросервисной архитектуры [5].

1.3 Цель исследования

Цель – реализовать очередь сообщений с помощью брокера сообщений RabbitMQ и внедрить данную очередь в API-сервис, написанный на языке программирования Python.

2 Материалы и методы

Для реализации поставленной задачи используется язык программирования Python, модуль для взаимодействия с брокером сообщений Pika, модуль для запуска API-сервиса FastAPI.

3 Результаты и обсуждения

В первую очередь нужно определить структуру тестового проекта, в котором будет происходить взаимодействие между API-сервисом и брокером сообщений.

В проекте предусмотрены 3 основных Python-скрипта:

1. main.py – основной скрипт, в котором описаны конечные точки сервиса и данные для подключения к обработчику задач Celery.
2. tasks.py – скрипт с задачей, которая будет ставиться в очередь.
3. utils.py – скрипт, который будет возвращать список всех созданных, но ещё не обработанных сообщений.

Также проект будет развёрнут в виде контейнера в Docker с использованием Docker Compose. На данном этапе можно определить сервисы, которые должны быть установлены и запущены в контейнере: брокер сообщений RabbitMQ, асинхронная очередь задач Celery и сам сервер, в котором будут располагаться скрипты для работы API-сервиса.

Для Celery нужно также определить команду для запуска очереди, чтобы сервис работал корректно. В файле «docker-compose.yml» необходимо определить команду запуска очереди, указав в качестве параметров задачи, которые должны быть запущены, сам обработчик заданий worker и уровень логирования:

```
celery -A app.tasks.celery_app worker --loglevel=info
```

RabbitMQ работает по AMQP. AMQP – это протокол передачи сообщений, предназначенный для обеспечения надежной и безопасной передачи данных между приложениями. Он был разработан как альтернатива существующим решениям для обмена сообщениями, таким как JMS (Java Message Service) или MQTT, с акцентом на переносимость, гибкость и высокую производительность.

Для RabbitMQ достаточно указать порты, по которым будет доступно подключение к брокеру (по умолчанию это 5672 и 15672).

Таким образом итоговый файл «docker-compose.yml» содержит следующие сервисы:

```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "8001:80"
    depends_on:
      - rabbitmq
      - celery

  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"

  celery:
    build: .
```

```
command: celery -A app.tasks.celery_app worker --
loglevel=info
depends_on:
  - rabbitmq
```

В Dockerfile достаточно указать базовый образ для запуска сервера (так как приложение построено на FastAPI, то таковым будет образ языка программирования Python), а также указать рабочую директорию, установить все необходимые модули (список которых лучше всего сохранить в отдельный текстовый файл requirements.txt). После подготовки образа содержимое проекта достаточно скопировать в рабочую директорию проекта и запустить сервер Uvicorn.

Получился следующий Dockerfile:

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

Теперь нужно правильно настроить конечные точки и данные для подключения в main.py. В настройках Celery нужно указать:

1. Название скрипта с задачей, которая должна быть запущена брокером сообщений.
2. URL для подключения к брокеру через протокол AMQP (нужно указать логин пароль, а также порт для подключения).
3. URL для backend-составляющей проекта (это RPC протокол).

Созданный объект класса Celery с указанными настройками достаточно сохранить в отдельную переменную, которую затем можно будет задействовать в любой части скрипта:

```
celery_app = Celery(
    'tasks',
    broker='amqp://guest:guest@rabbitmq:5672//',
    backend='rpc://'
)
```

В main.py также указан класс модели, которая будет задействована в конечной точке отправки сообщения на почту пользователя. Класс в данном примере намеренно простой и содержит всего один текстовый атрибут: email пользователя, который можно указать в теле POST-запроса:

```
class EmailRequest(BaseModel):  
    email: str
```

Для GET-запроса определена конечная точка по следующему пути: localhost/queue/. При обращении к этой точке происходит вызов специальной функции `get_messages_from_queue`, которая находится в скрипте `utils.py` и возвращает список сообщений, которые ещё не были обработаны. Функция, с которой взаимодействует сервер возвращает список всех необработанных сообщений:

```
@app.get("/queue/")  
async def get_queue():  
    messages = get_messages_from_queue()  
    return messages
```

POST-запрос состоит из следующих шагов:

1. Сначала определяется UUID случайным образом для отправки, а также время отправки сообщения в ISO-формате.

2. Создаётся задание с помощью метода `create_async` для функции задачи `send_email_task`, которая определена в `tasks.py`. Задаче передаются следующие параметры: время задержки выполнения (`countdown`) и входные аргументы в виде почты пользователя, UUID и времени отправки (`args`).

3. В консоль выводится сообщение о создании новой задачи и после завершения обработки задачи выводится также содержимое отправки.

Итоговая функция для конечной точки (расположена по адресу: localhost/send-email) выглядит следующим образом:

```
@app.post("/send-email/")  
async def send_email(email_request: EmailRequest):  
    message_id = str(uuid.uuid4())  
    timestamp = datetime.now().isoformat()  
    task = send_email_task.apply_async(  
        args=[email_request.email, message_id, timestamp],  
        countdown=5  
    )  
    print(f"Created new task: {task.id}")  
    return {"message_id": message_id, "timestamp":  
timestamp, "task_id": task.id}
```

Главный скрипт готов и теперь нужно реализовать задачу, которую будет выполнять сервере с помощью брокера сообщений. В качестве основы была реализована задача по выводу содержимого отправки, которая в дальнейшем будет развита в задачу по отправке письма на указанную пользователем почту.

В скрипте `tasks.py` необходимо также указать данные для подключения к очереди Celery схожим образом, а сами задачи должны быть представлены как функции и «обёрнуты» в специальный декоратор `«celery_app.task»`:

```

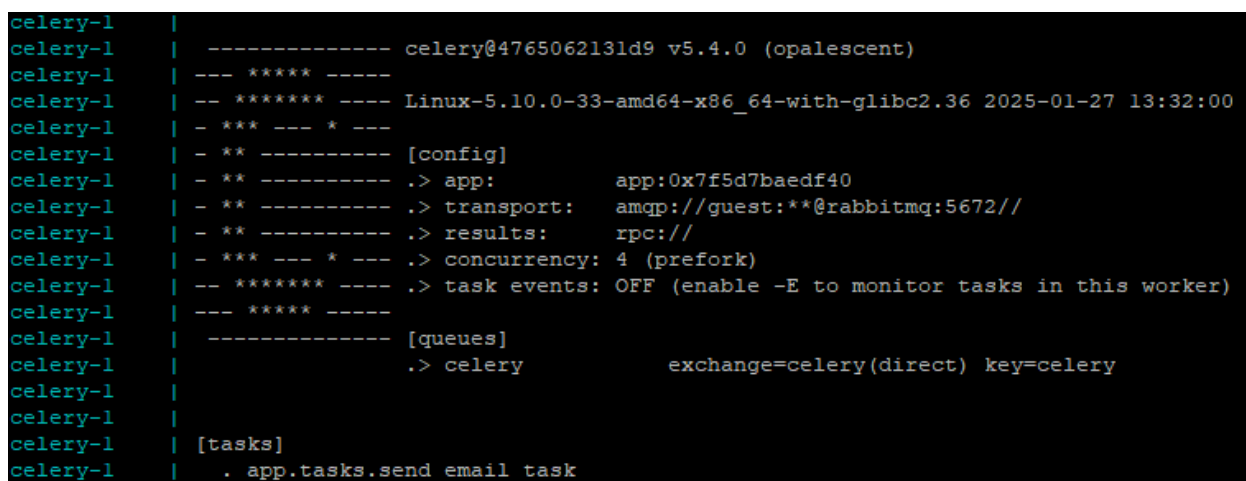
from celery import Celery

celery_app = Celery(
    'app',
    broker='amqp://guest:guest@rabbitmq:5672//',
    backend='rpc://'
)

@celery_app.task
def send_email_task(email, message_id, timestamp):
    log_message = f"Sending email to: {email}\nMessage ID: {message_id}\nTimestamp: {timestamp}"
    print(log_message)

```

На данном этапе можно протестировать работу приложения и попробовать создать сообщение и отправить его в RabbitMQ. Если структура проекта была определена правильно и worker смог подключиться к задаче, то об этом будет выведено соответствующее сообщение в логе проекта (см. рис. 1).



```

celery-1 | ----- celery@4765062131d9 v5.4.0 (opalescent)
celery-1 | --- ***** ---
celery-1 | -- ***** --- Linux-5.10.0-33-amd64-x86_64-with-glibc2.36 2025-01-27 13:32:00
celery-1 | - *** --- * ---
celery-1 | - ** ----- [config]
celery-1 | - ** ----- .> app: app:0x7f5d7baedf40
celery-1 | - ** ----- .> transport: amqp://guest:**@rabbitmq:5672//
celery-1 | - ** ----- .> results: rpc://
celery-1 | - *** --- * --- .> concurrency: 4 (prefork)
celery-1 | -- ***** --- .> task events: OFF (enable -E to monitor tasks in this worker)
celery-1 | --- ***** ---
celery-1 | ----- [queues]
celery-1 | .> celery exchange=celery(direct) key=celery
celery-1 |
celery-1 | [tasks]
celery-1 | . app.tasks.send_email_task

```

Рисунок 1. Сообщение о начале работы очереди Celery и успешном подключении к брокеру RabbitMQ

Одним из главных достоинств RabbitMQ можно считать графическая оболочка, доступная в браузере и позволяющая разработчику просматривать очередь сообщений в текущий момент времени (см. рис. 2).

На странице брокера сообщений представлена детальная информация о выполнении каждого сообщения, ведётся статистика о количестве сообщений, находящихся в обработке и неудачно обработанных. Также можно просмотреть потребление ресурсов брокером, подключения, очереди, каналы и др.

Также отображается информация о версиях брокера сообщений и языка программирования Erlang, на котором реализован функционал RabbitMQ.



Рисунок 2. Панель администратора брокера сообщений RabbitMQ в браузере

Также FastAPI автоматически создаёт документацию к конечным точкам благодаря модулю Swagger (см. рис. 3). В документации можно протестировать работу API, а также изучить структуру классов, которые задействованы в проекте.

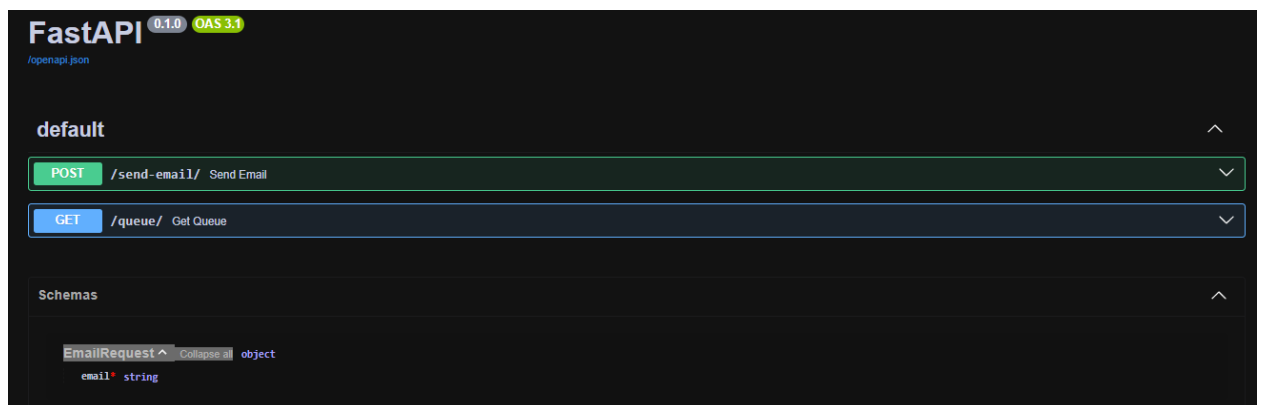
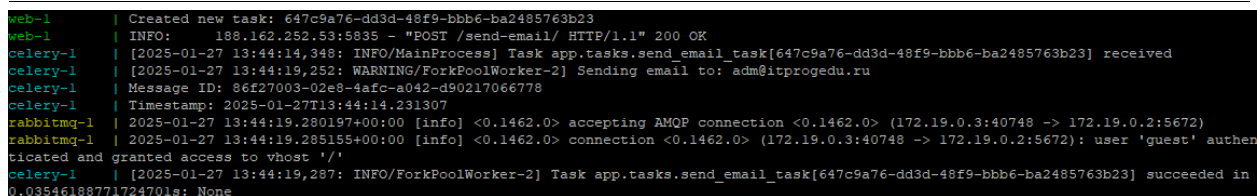


Рисунок 3. Документация к API, сформированная через Swagger

Проверка работы POST-запроса осуществлялась через документацию, в которой есть возможность ввода тела запроса с последующей отправкой cURL-запроса на сервер.

В качестве тестового значения была указана почта «adm@itprogedu.ru». После отправки сообщения в консоли отобразилась информация о создании нового сообщения и сохранении его в соответствующей очереди (см. рис. 4). Также по завершении обработки сообщения в консоль была выведена информация об email, UUID и времени создания сообщения, после чего Celery сообщил об успешном выполнении задачи.



```

web-1 | Created new task: 647c9a76-dd3d-48f9-bbb6-ba2485763b23
web-1 | INFO: 188.162.252.53:5835 - "POST /send-email/ HTTP/1.1" 200 OK
celery-1 | [2025-01-27 13:44:14,348: INFO/MainProcess] Task app.tasks.send_email_task[647c9a76-dd3d-48f9-bbb6-ba2485763b23] received
celery-1 | [2025-01-27 13:44:19,252: WARNING/ForkPoolWorker-2] Sending email to: adm@itprogedu.ru
celery-1 | Message ID: 86f27003-02e8-4afc-a042-d90217066778
celery-1 | Timestamp: 2025-01-27T13:44:14.231307
rabbitmq-1 | 2025-01-27 13:44:19.280197+00:00 [info] <0.1462.0> accepting AMQP connection <0.1462.0> (172.19.0.3:40748 -> 172.19.0.2:5672)
rabbitmq-1 | 2025-01-27 13:44:19.285155+00:00 [info] <0.1462.0> connection <0.1462.0> (172.19.0.3:40748 -> 172.19.0.2:5672): user 'guest' authenticated and granted access to vhost '/'
celery-1 | [2025-01-27 13:44:19,287: INFO/ForkPoolWorker-2] Task app.tasks.send_email_task[647c9a76-dd3d-48f9-bbb6-ba2485763b23] succeeded in 0.03546188771724701s: None

```

Рисунок 4. Лог обработки сообщения на сервере

После реализации простого примера с выводом сообщения в консоль скрипт `tasks.py` был изменён для отправки писем на указанную почту. Для реализации такого функционала используется модуль `email.mime.text`. В скрипте были указаны данные для подключения к почте, с которой будет отправляться письмо, а также данные пользователя, которому письмо нужно отправить (это параметр `email`, получаемый через POST-запрос). Также необходимо указать сервер SMTP и протокол для корректного подключения к почтовому серверу (протокол был указан по умолчанию 587). Итоговый код функции `send_email_task` в файле `tasks.py` получился следующим:

```

def send_email_task(email, message_id, timestamp):
    msg = MIMEText(f"Message ID: {message_id}\nTimestamp: {timestamp}")
    msg['Subject'] = 'Test Email'
    msg['From'] = 'loginmail@login.com'
    msg['To'] = email

    with SMTP('smtp.server.com', 587) as server:
        server.starttls()
        server.login('loginmail@login.com', 'password')
        server.send_message(msg)

```

Рисунок 5. Результат работы задачи отправки писем на указанную в POST-запросе электронную почту (в логе сервера)

Также если подключение к почтовому ящику отправителя было настроено правильно, то письмо будет успешно отправлено на указанный электронный адрес (см. рис. 6).

Для получения списка сообщений в очереди в данный момент времени была реализована функция `get_messages_from_queue`, в которой производится подключение к RabbitMQ, затем через стандартный канал (после подключения к очереди задач Celery) выбираются все сообщения, которые ещё не были выполнены. Результирующий список сообщений возвращается в качестве результата работы функции:

```

import pika

def get_messages_from_queue():
    connection_params =
pika.ConnectionParameters('rabbitmq')

```

```

        with pika.BlockingConnection(connection_params) as
connection:
        with connection.channel() as channel:
            channel.queue_declare(queue='celery',
durable=True)

            messages = []
            while True:
                method_frame, header_frame, body =
channel.basic_get(queue='celery', auto_ack=False)
                if method_frame:
                    messages.append(body.decode())

channel.basic_ack(delivery_tag=method_frame.delivery_tag)
            else:
                break

        return messages

```

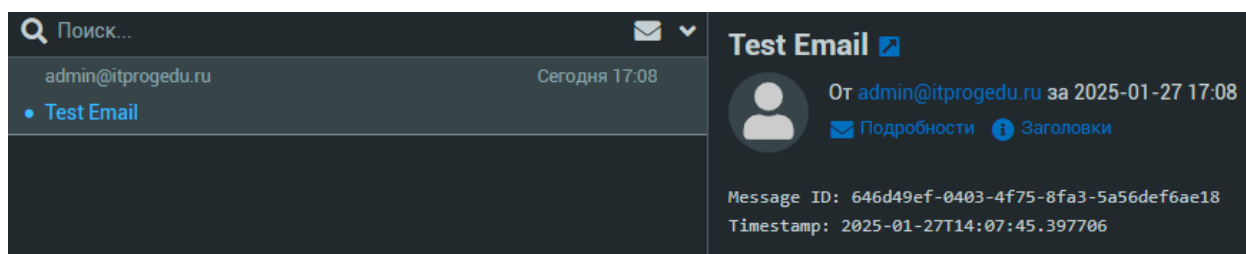


Рисунок 6. Отправленное письмо с данными о задаче на сторонний почтовый адрес (результат выполнения POST-запроса)

4 Выводы

Был продемонстрирован пример реализации взаимодействия между брокером сообщений и API-сервером, работающим на Uvicorn и FastAPI. В статье представлено 2 примера: вывод сообщений в консоль с задержкой и отправка письма с такой же задержкой (с предварительным указанием адреса электронной почты получателя).

Второй пример будет задействован для реализации системы отчётов в информационной системе обучения программированию, которая работает с 2023-го года на базе Приамурского государственного университета им. Шолом-Алейхема. Планируется внедрение возможности создания дисциплин, в которых студенты будут загружать отчёты на сервер, а преподаватель будет получать об этом сообщение на почту.

Библиографический список

1. Дмитриева А.А. Тестирование производительности RabbitMQ // В сборнике: Современные проблемы цивилизации и устойчивого развития в информационном обществе. сборник материалов VIII Международной научно-практической конференции. Москва. 2022. С. 131-133.

2. Васильев Г.В. Эффективное построение микросервисной архитектуры с помощью брокера сообщений RabbitMQ // В сборнике: Молодежь и наука: актуальные проблемы фундаментальных и прикладных исследований. Материалы VI Всероссийской национальной научной конференции молодых учёных. Комсомольск-на-Амуре. 2023. С. 455-458.
3. Шиловский С.В., Давидовская М.И. Анализ использования системы очередей сообщений на примере RabbitMQ // Научные исследования XXI века. 2023. № 3 (23). С. 53-58.
4. Золотарев Д.А. Сравнительный анализ Apache ActiveMQ Artemis и RabbitMQ // Тенденции развития науки и образования. 2023. № 103-8. С. 143-147.
5. Натроби́на О.В., Пашечко В.Ю. Разработка встраиваемого модуля гарантированной доставки сообщений в брокере RabbitMQ // Дневник науки. 2022. № 10 (70).