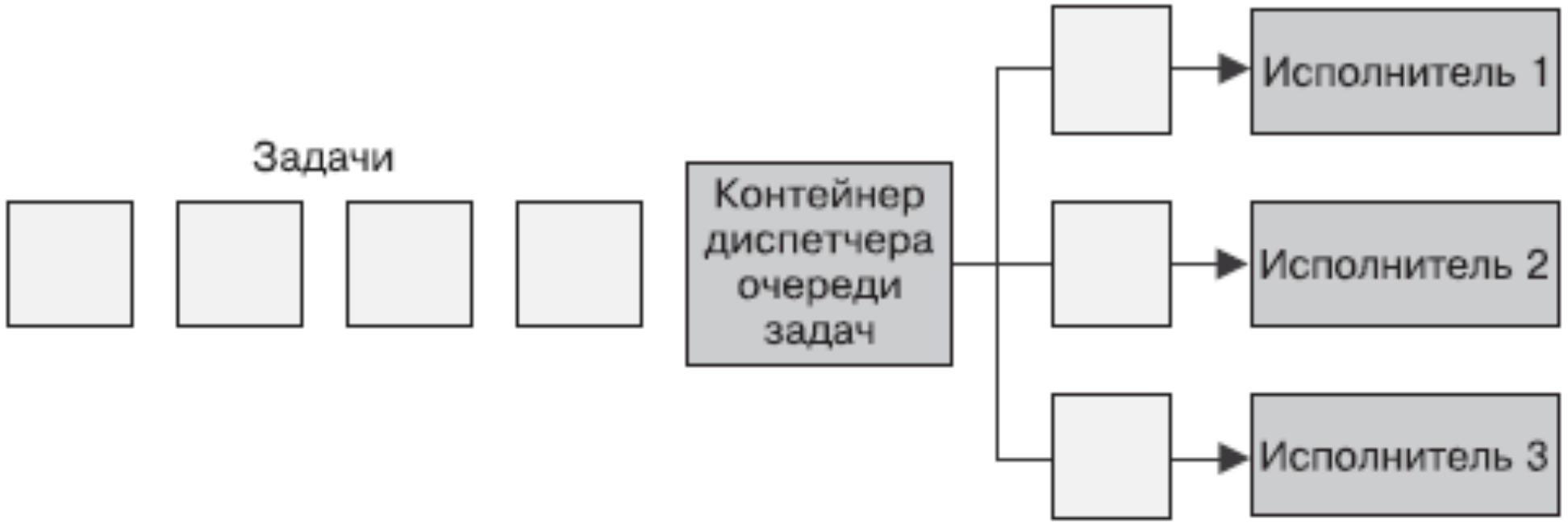
FastAPI+Celery+RabbitMQ

МАКСИМОВСКАЯ
АНАСТАСИЯ

Очередь задач

- В системе с очередью задач есть набор задач, которые должны быть выполнены.
- Каждая задача полностью независима от остальных и может быть обработана без всяких взаимодействий с ними.
- В общем случае цель системы с очередью задач — обеспечить выполнение каждого этапа работы в течение заданного промежутка времени.
- Количество рабочих потоков увеличивается либо уменьшается сообразно изменению нагрузки.

Очередь задач



Источник: URL

Очередь задач

- Когда мы создаем веб-приложения/сервисы, которые выполняют тяжелую работу на сервере, требующую времени (более нескольких миллисекунд) или длительной работы, следует использовать очередь задач.
- Вы можете думать об этом как об асинхронности, поднятой на новый уровень. Это помогает нам поставить задачу в очередь на обработку и отправить клиенту какое-то подтверждение непосредственно перед тем, как мы выполним фактическую обработку и перейдем к следующему запросу
- Другой сервер просто проверит список , если есть какие-либо ожидающие задачи, и обработает их Как только это будет выполнено с заданием, оно подтвердит сервер API, который сообщит клиенту, что задание выполнено

Термины

Queue: Очереди похожи на настоящие очереди, в которых похожие задания/задачи сгруппированы вместе, ожидая обработки работником в порядке FIFO (первым поступил – первым обслужен).

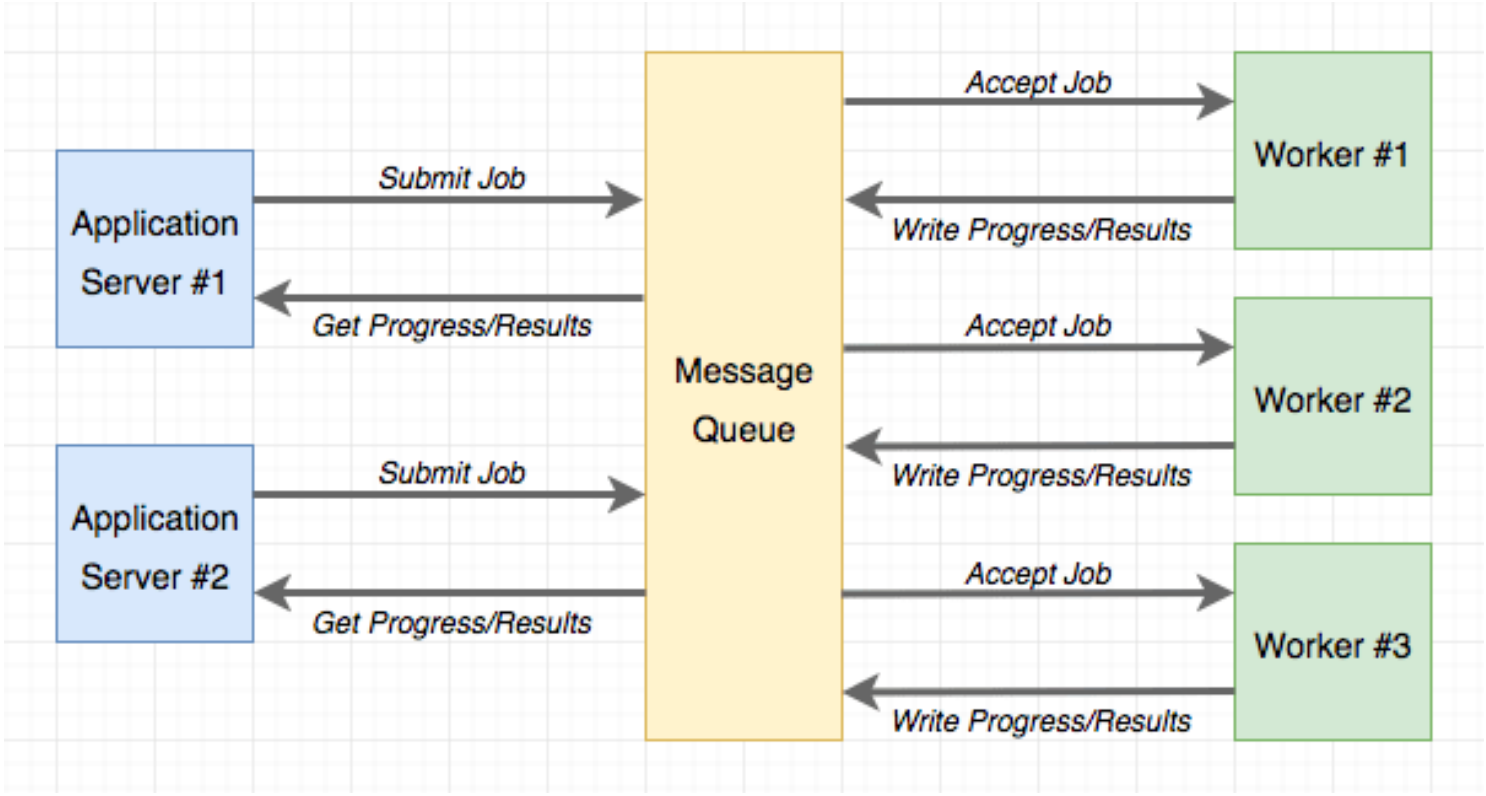
Jobs/Tasks: это объекты, которые содержат фактические сведения о задании, ожидающем обработки.

Publisher : это тот, кто добавляет задачу в очередь.

Consumer: он отслеживает очередь заданий на предмет любого ожидающего задания и отправляет его на обработку.

Worker: фактическая машины, которая обрабатывает задание и уведомляет, было ли оно успешным или нет.

Схема


Application Server #1
Application Server #2
Submit Job
Get Progress/Results
Submit Job
Get Progress/Results
Message Queue
Accept Job
Write Progress/Results
Accept Job
Write Progress/Results
Accept Job
Write Progress/Results
Worker #1
Worker #2
Worker #3

Действия

- Сначала мы настроим сервер API с некоторыми эндпоинтами, которые будут отвечать на HTTP-запросы клиента.
- Сервер API публикует задание в соответствующей очереди и отправляет клиенту какое-то подтверждение (успех / неуспех и тд)
- Потребитель наблюдает и использует очередь, отправляя задачу для обработки воркеру.
- Рабочий процесс обрабатывает задание (одно или несколько за раз), сообщает о ходе выполнения (при желании) и отправляет событие после завершения задания. Вы можете заметить, что задача может завершиться ошибкой и на этом этапе, поэтому она отправляет событие успеха или ошибки, которое можно обработать соответствующим образом.

Действия

- Сервер API запрашивает ход выполнения и сообщает об этом клиенту, чтобы приложение могло отображать хороший индикатор выполнения в пользовательском интерфейсе.
- Он также отслеживает события успеха или неудачи и отправляет уведомление клиенту.
- Теперь клиент может запросить ресурс через другой вызов API, а сервер отвечает клиенту запрошенным ресурсом и закрывает соединение.

Где тут сельдерей?

Почему celery?

Плюсы:

- Опен сорс
- Легко ставить
- Поддерживает много брокеров, в том числе redis и rabbitmq
- Интеграция с фласком и другими веб фреймворками

Почему celery?

Минусы:

- Мало поддержки для больших корпораций
- User experience
- Проблемы с некоторыми интеграциями (некоторые жалуются на редис)

Где тут заяц?

Почему RabbitMQ?

Плюсы:

- Опенсорс
- Поддерживает стандартные message protocols
- Много в каких компаниях используется
- Юзер-френдли
- Масштабируемый

Почему RabbitMQ?

Минусы:

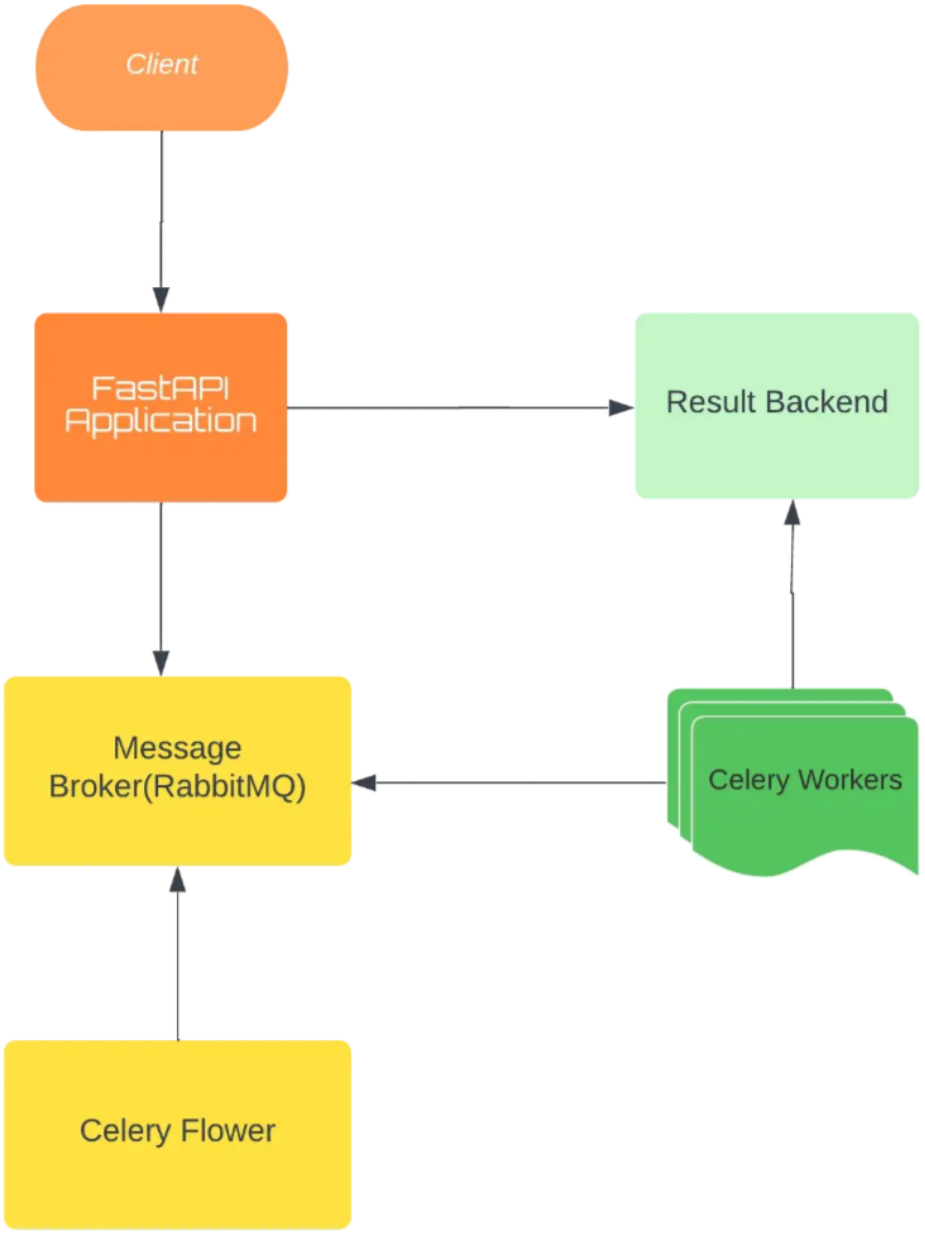
- Медленно обрабатывает большие наборы данных
- Довольно сложный в обслуживании
- Он имеет услуги интеграции премиум-класса
- Плохая документация

Что мы хотели сделать в прошлый раз?

Юз-кейсы:

- Отправка электронных писем в качестве фоновых задач в приложении.
- Обработка загруженных изображений в фоновом режиме.
- Оффлайн обучение моделей машинного обучения.
- Периодические задачи, такие как создание отчетов или удаление веб-страниц.

Архитектура



Что нужно сделать?

1. Настроить и установить все нужное
2. Настроить брокер сообщений
3. Добавить сельдерей
4. Добавить задачи сельдерея
5. Добавить APIмаршрутизатор
6. Запустить приложение и рабочий сервер сельдерея.
7. Протестировать приложение
8. Следить за задачами

Настройка и установка

1. Создать проект и virtual environment (многие IDE, например, PyCharm, сделают за Вас)
2. Поставить нужные библиотеки для Вашего проекта + fastapi, uvicorn, celery, flower
3. Сделать pip freeze > requirements.txt

Настройка брокера

1. brew install rabbitmq
2. brew services start rabbitmq
3. http://localhost:15672 – открываем, вводим логин-пароль guest

Добавляем сельдерей

```python
from config.celery_utils import create_celery
from routers import sentiments


def create_app() -> FastAPI:
    current_app = FastAPI(title="Asynchronous tasks processing with Celery and RabbitMQ",
                          description="Sample FastAPI Application to demonstrate Event "
                                      "driven architecture with Celery and RabbitMQ",
                          version="1.0.0", )

    current_app.celery_app = create_celery()
    current_app.include_router(sentiments.router)
    return current_app


app = create_app()
celery = app.celery_app


if __name__ == "__main__":
    uvicorn.run("main:app", port=9000, reload=True)
```

Конфигурация сельдерея

```python
import os
from functools import lru_cache
from kombu import Queue


def route_task(name, args, kwargs, options, task=None, **kw):
    if ":" in name:
        queue, _ = name.split(":")
        return {"queue": queue}
    return {"queue": "celery"}


class BaseConfig:
    CELERY_BROKER_URL: str = os.environ.get("CELERY_BROKER_URL", "amqp://guest:guest@localhost:5672//")
    CELERY_RESULT_BACKEND: str = os.environ.get("CELERY_RESULT_BACKEND", "rpc://")

    CELERY_TASK_QUEUES: list = (
        # default queue
        Queue("celery"),
        # custom queue
        Queue("polarity"),
    )
```

Конфигурация сельдерея

```
    CELERY_TASK_ROUTES = (route_task,)


class DevelopmentConfig(BaseConfig):
    pass


@lru_cache(maxsize=128)
def get_settings():
    config_cls_dict = {
        "development": DevelopmentConfig,
    }
    config_name = os.environ.get("CELERY_CONFIG", "development")
    config_cls = config_cls_dict[config_name]
    return config_cls()


settings = get_settings()
```

Instance (экземпляр) сельдерея

```python
from celery import current_app as current_celery_app
from celery.result import AsyncResult

from .celery_config import settings


def create_celery():
    celery_app = current_celery_app
    celery_app.config_from_object(settings, namespace='CELERY')
    celery_app.conf.update(task_track_started=True)
    celery_app.conf.update(task_serializer='pickle')
    celery_app.conf.update(result_serializer='pickle')
    celery_app.conf.update(accept_content=['pickle', 'json'])
    celery_app.conf.update(result_persistent=True)
    celery_app.conf.update(worker_send_task_events=False)
    celery_app.conf.update(worker_prefetch_multiplier=1)

    return celery_app
```

Откуда берем настройки?

`result_persistent`

Default: Disabled by default (transient messages).

If set to **`True`**, result messages will be persistent. This means the messages won't be lost after a broker restart.

`task_track_started`

Default: Disabled.

If **`True`** the task will report its status as 'started' when the task is executed by a worker. The default value is **`False`** as the normal behavior is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a 'started' state can be useful for when there are long running tasks and there's a need to report what task is currently running.

Откуда берем настройки?

Добавляем по мере необходимости гуглежом или смотрим в доку:

https://docs.celeryq.dev/en/stable/userguide/configuration.html#std-setting-task_track_started

Создаем таски

И настраиваем по необходимости. Shared_task ниже позволяет не привязываться к инстансу селери, альтернатива – app.task

```python
from celery import shared_task

from api import polarity

@shared_task(bind=True,autoretry_for=(Exception,), retry_backoff=True, retry_kwargs={"max_retries": 5},
             name='polarity:get_polarity_one')
def get_polarity_task(review: str):
    sentiment = polarity.get_polarity_one(review)
    return sentiment
```

Создаем таски

`Task.retry_backoff`

A boolean, or a number. If this option is set to `True`, autoretries will be delayed following the rules of exponential backoff. The first retry will have a delay of 1 second, the second retry will have a delay of 2 seconds, the third will delay 4 seconds, the fourth will delay 8 seconds, and so on. (However, this delay value is modified by **`retry_jitter`**, if it is enabled.) If this option is set to a number, it is used as a delay factor. For example, if this option is set to `3`, the first retry will delay 3 seconds, the second will delay 6 seconds, the third will delay 12 seconds, the fourth will delay 24 seconds, and so on. By default, this option is set to `False`, and autoretries will not be delayed.

Создаем таски

Параметры по необходимости гуглежом или из доки:

https://docs.celeryq.dev/en/stable/userguide/tasks.html#Task

Добавляем API routers

```python
from fastapi import APIRouter

from api import polarity
from config.celery_utils import get_task_info
from schemas.schemas import Review

router = APIRouter(prefix='/polarities', tags=['Polarity'], responses={404: {"description": "Not found"}})

@router.post("/")
def get_polarities(review: Review) -> dict:
    return polarity.get_polarity_one(review.review)

@router.get("/task/{task_id}")
async def get_task_status(task_id: str) -> dict:
    return get_task_info(task_id)
```

Запускаем

python main.py в терминале

http://localhost:9000/docs -- наслаждаемся)

Без /docs не работает, тк мы не прописали гет эндпоинт

celery -A main.celery worker --loglevel=info -Q polarity

Запускаем селери воркер

Мониторим задачи цветочком:

celery -A main.celery flower --port=5555

Задания

Добавьте тесты, которые проверяют правильность работы модели

Например, что на негативный текст она даст оценку ниже 0

Библиотека на Ваш вкус, советую начать с pytest. https://docs.pytest.org/en/7.2.x/

Попробуйте линтеры

Например, pylint https://towardsdatascience.com/using-pylint-to-write-clean-python-code-660eff40ed8

Добавьте асинхронность

По аналогии с get_universities_async в статье

https://medium.com/cuddle-ai/async-architecture-with-fastapi-celery-and-rabbitmq-c7d029030377