# ДЗ: Онлайн обновление ленты новостей

Задание решено реализовывать на связке: FastAPI Websockets и RabbitMQ (модуль pika для python), поскольку остальная часть проекта уже реализуется на фреймворке FastAPI.

При реализации исходил из следующих соображений:

1. После загрузки страницы скриптом фронта поднимается websocket соединение, в рамках которого ожидается онлайн получение новых постов авторов, на которых подписан пользователь.
2. Старые посты в ленте сообщений загружаются из tarantool кеша при запросе страницы пользователем. В связи с этим:
   1. Очередь сообщения актуальна только пока пользователь подключен к websocket
   2. После загрузки ленты постов из кеша очередь должна быть пуста, т.к. все опубликованные посты уже получены
   3. После отключения пользователя сохранение постов в очередь не требуется.

В связи с этим было решено:

* При начале работы с rabbitmq (инициации подключения) создается отдельный exchange «postedfeed» c типом “topic”. Используется особенность работы rabbitmq – игнорирование декларирования exchange, если он уже существует.
* При подключении пользователя к ws создавать уникальную очередь c routing\_key равным его userid с . При отключении пользователя – очередь удаляется
* При создании поста, новый пост публикуется в “postedfeed” c routing\_key всех подписчиков автора. Тут используется особенность rabbitmq, что задачи с routing\_key для которого отсутствует целевая очередь будут удалены.

Работа с очередями реализована в классе SocialQueue, основные методы которого:

connect() – выполняет подключение к rabbintmq и декларирует создание создание нужного exchange.

create\_userqueue(userid) – создает очередь для пользователя и биндит ее к exchange «postedfeed» с routing\_key = userid

delete\_userqueue(userid) – отбиндивает очередь пользователя userid от exchange и удаляет ее

addpost2feed(userid, post) – публикует сообщение post в exchange для пользователя userid (routing\_key)

При подключении пользователь к websocket выполняется проверка его авторизации (наличие правильного id сессии в auth заголовке), после чего:

1. Создается объект SocialQueue и выполняется подключение к rabbitmq
2. Из сессии извлекается userid и для него создается очередь
3. Запускается бесконечный цикл, в каждой итерации которого:
   1. Выполняется чтение сообщений в очереди пользователя методом basic\_get (от использования start\_comsuming отказался в связи с конфликтом блокироующим асинхронной работой websocket в python), пока в очереди есть сообщения (возвращается не None), для каждого сообщения выполняется send\_json поста пользователю.
   2. Задержка в 5 сек перед следующей итерация (для снижения нагрузки на компоненты сервиса).
4. Дополнительно добавлена проверка отключения пользователя и обработка соответствующего исключения. После обработки исключения выполняется удаление очереди пользователя и отключение от rabbitmq

Также был доработан метод create\_post, в который добавлено:

1. Подключение к rabbitmq
2. Отправка созданного поста в очереди подписчиков автора по routing\_key = userid подписчика.
   1. Пользователь получит пост в рамках websocket подключения в течение 5 секунд от отправки (пауза между проверками сообщений в очереди)

## Масштабирование сервисов:

Линейная масштабируемость websocket сервиса достигается асинхронной реализацией фремворка FastAPI и независимостью веб-сервиса от хранилищ данных, это позволяет запускать параллельно несколько экземпляров и увеличивать производительность проекта.

Для масштабирования rabbitmq применяется deploy c преднастроенной кластеризацией.

В стандартную конфигурацию сервиса добавлены строки автоподключения к кластеру с указанием первых двух нод:

cluster\_formation.peer\_discovery\_backend = classic\_config

cluster\_formation.classic\_config.nodes.1 = rabbit@rabbit.node1

cluster\_formation.classic\_config.nodes.1 = rabbit@rabbit.node2

Конфигурация расположена в каталоге проекта «./db/rabbitmq/cfg», который монтируется к образу в точке /etc/rabbitmq/conf.d для применения при старте.

Также к env контейнера указываются общие дефолтные учетные данные, ерланг кука и использование длинных имен:

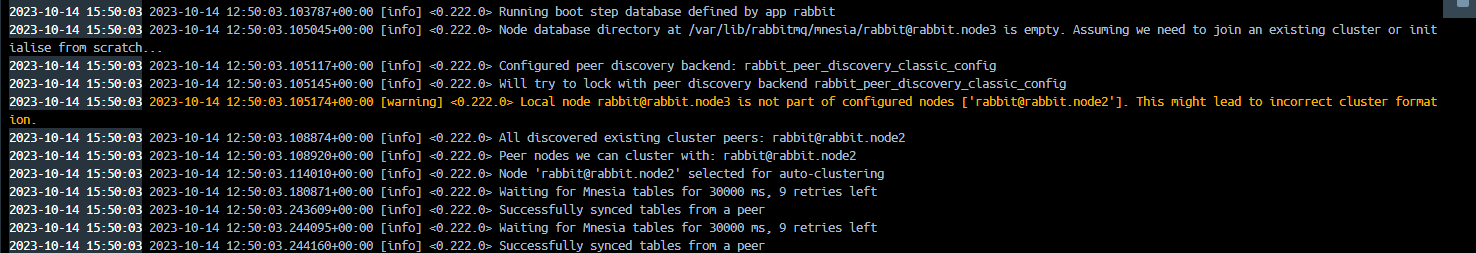
      RABBITMQ\_ERLANG\_COOKIE: <кука>

      RABBITMQ\_DEFAULT\_USER: <логин>

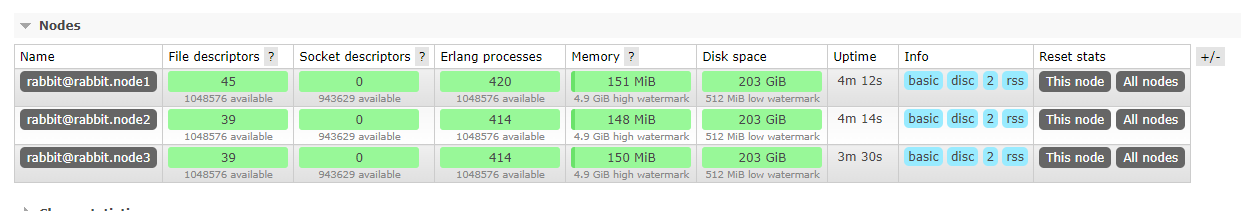
      RABBITMQ\_DEFAULT\_PASS: <пароль>

      RABBITMQ\_USE\_LONGNAME: true

В результате при деплое новой ноды rabbitmq, она автоматически находит существующий кластер и подключается к нему по имени одной из нод в конфиге, о чем можно увидеть в логах старта сервиса:



И через веб-консоль видно, что нода3 успешно присоединилась к кластеру:



Для сохранности данных применяются DISK ноды кластера rabbitmq с примонтированным внешним хранилищем:

    volumes:

      - ./db/rabbitmq/wrk/<нода Х>:/var/lib/rabbitmq

что позволяется перезапускать и обслуживать ноды без потери данных.

**Запуск сервиса:**

Запуск осуществляется командой:

docker-compose -f ./ docker-compose.rabbit.yaml up --build –d

Для масштабирования rabbitmq в docker-compose добавляется описания необходимого количества нод (с уникальными именем хоста, контейнера и каталогами внешнего хранилища)