Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №3:**

**«*Брокеры очередей и тестирование веб API*»**

Антон Гатченко Б22-525

2024 г.

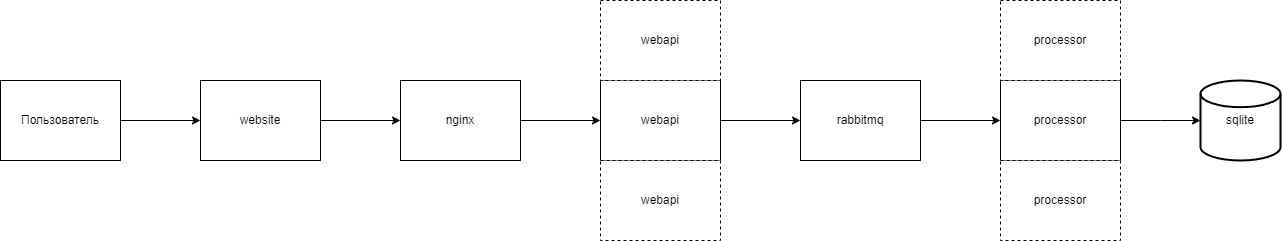
*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit

*Ход работы:*

В ходе работы был реализован проект по сервису онлайн-магазина TNS. У пользователей есть возможность делать заказы неких товаров.

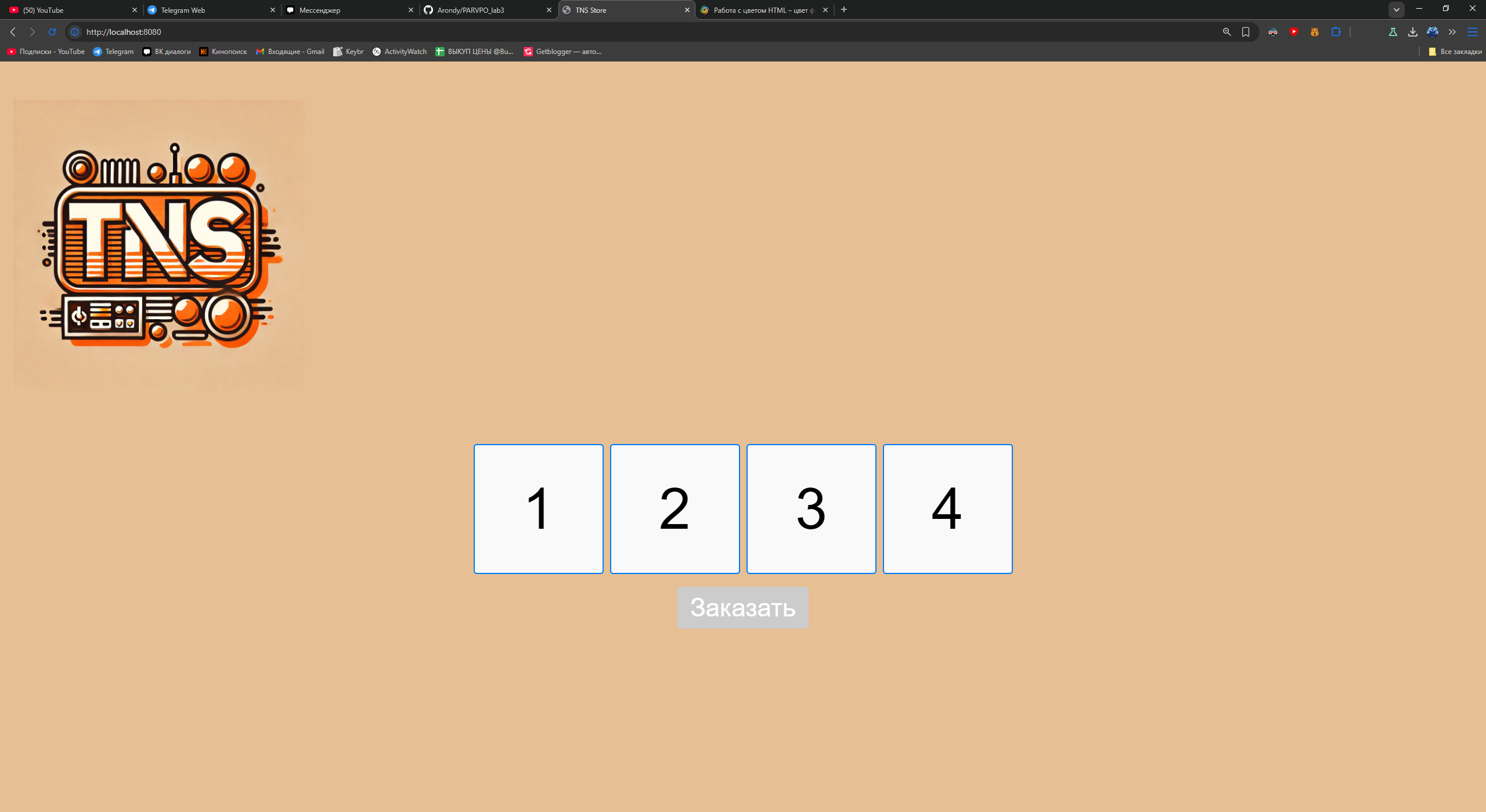
Схема проекта:



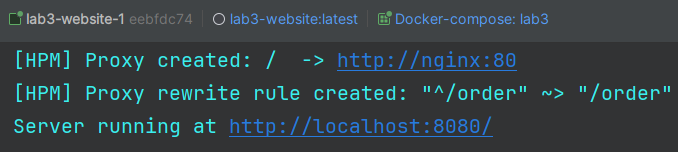
В проект входят следующие контейнеры:

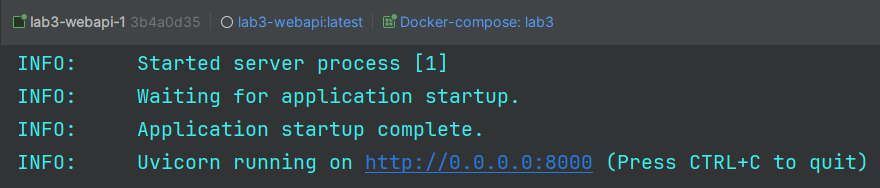
* Website – фронтенд для подключения пользователей к сервису. Через него происходит взаимодействие с системой, где пользователь размещает заказ, посредством отправки POST-запроса от скрипта script.js, встроенного в страницу, на локальный сервер, откуда он пересылается. Изначально планировалось реализовать его на FastApi и Python, поскольку другие компоненты использовали этот фреймворк, однако по каким-то причинам (выяснить которые не удалось) POST-запрос не отправлялся с ошибкой 405 Method Not Allowed (Приложение 2). По этой причине данный элемент был реализован на базе Node.js.
* Nginx – посредник между сайтом и webapi, reverse proxy (выполняет сложные задачи: аутентификацию и снятие https). Если потребуется несколько серверов webapi, nginx обеспечит корректную пересылку на все сервера.
* WebApi – получает запросы от nginx и добавляет их в очередь брокера rabbitmq. Может горизонтально масштабироваться. Реализован на FastApi, Python.
* Rabbitmq – брокер сообщений, реализует очередь запросов, которые отсылаются по мере возможностей серверов-приемщиков. Выбран из-за простоты пользования, а практическая польза обоснована возможным наплывом заказов на распродаже/при выпуске нового, крайне привлекательного для пользователей товара. Связь с остальными элементами с помощью асинхронной библиотеки Python – aio-pika. Был выбор между асинхронной и последовательной (pika) библиотеками, выбран первый вариант из-за более высокой производительности.
* Processor – обработчик запросов. Получает данные из запроса и добавляет их в БД (sqlite). В дальнейшем эти данные понадобятся для получения статуса заказа. Может горизонтально масштабироваться.

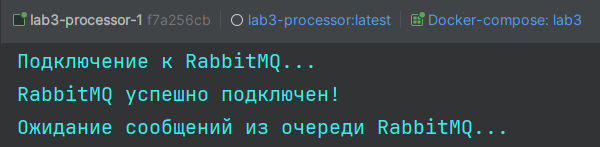
*Вид сайта:*



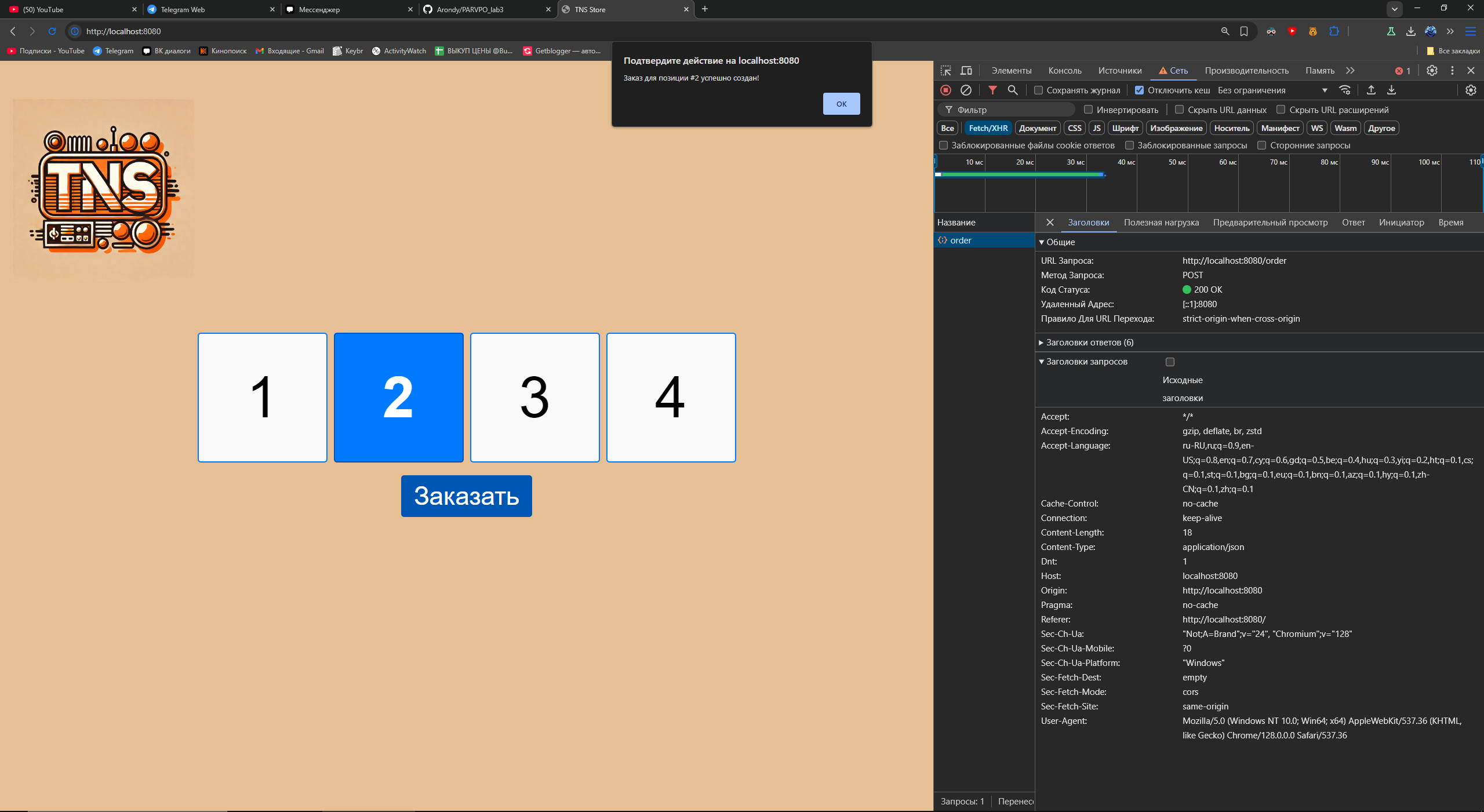
*Логи контейнеров при запуске:*

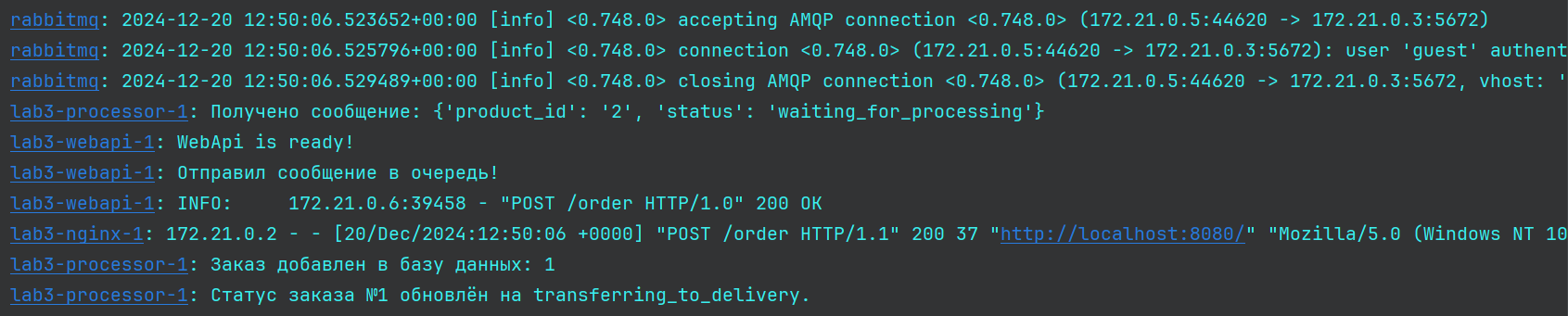






*При заказе:*





Статус заказа обновляется после некоторой задержки, имитируя обработку, после чего его состояние в БД изменяется.

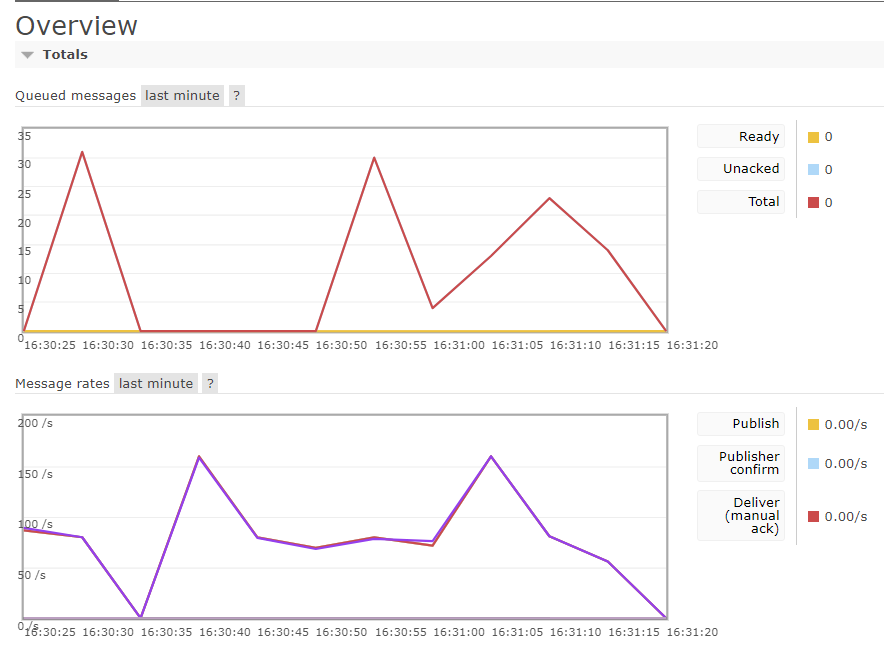
*Нагрузочное тестирование:*

***Единовременный всплеск активности:***

С помощью aiohttp отправлялись 5000 запросов. Пока что оставляем 1 processor. Посмотрим на показатели, когда значения CPU для всех контейнеров равны.

**Ограничение по CPU для всех контейнеров (кроме nginx): 0.5 CPU**

Максимальное количество запросов в очереди – 31, обрабатывались 100-150 сообщений в секунду. Время обработки всех сообщений – около минуты.



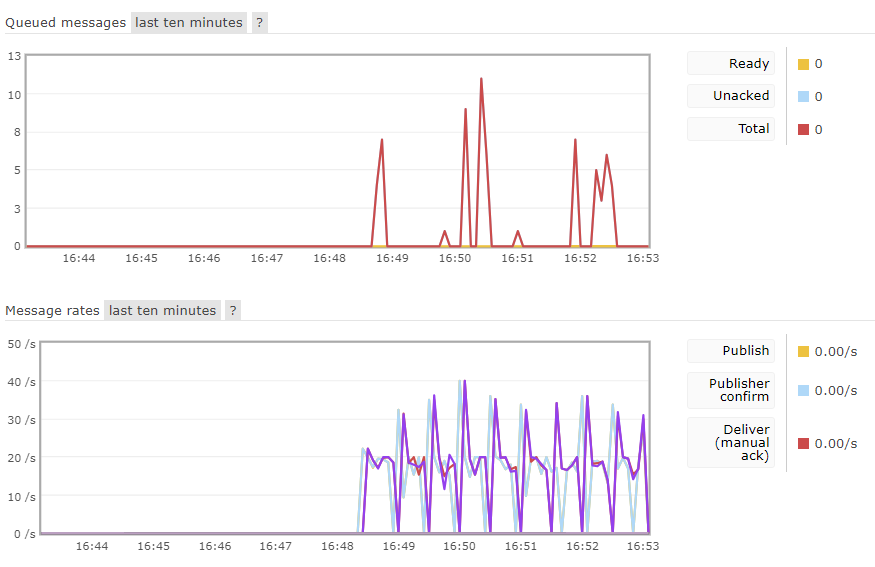
**Ограничение по CPU для всех контейнеров (кроме nginx): 0.25 CPU**

Максимальное количество запросов в очереди – 14, обрабатывались 50-90 сообщений в секунду. Время обработки всех сообщений – около 2 минут.



**Ограничение по CPU для всех контейнеров (кроме nginx): 0.1 CPU**

Максимальное количество запросов в очереди – 11, обрабатывались 20-40 сообщений в секунду. Время обработки всех сообщений – около 5 минут.



Как видно из графиков, время обработки всех сообщений растет, а количество обслуженных запросов в секунду падает. Это связано с уменьшением ресурсов также для website и webapi, которые не успевают передавать запросы. Увеличим их значение CPU. Посмотрим на показатели, когда значения CPU для rabbitmq и processor равны.

**Ограничение по CPU для website, webapi: 0.5 CPU; для rabbitmq, processor: 0.25 CPU**

Максимальное количество запросов в очереди – 26, обрабатывались 90+ сообщений в секунду (пик – 150+). Время обработки всех сообщений – <1 минуты.



**Ограничение по CPU для website, webapi: 0.5 CPU; для rabbitmq, processor: 0.1 CPU**

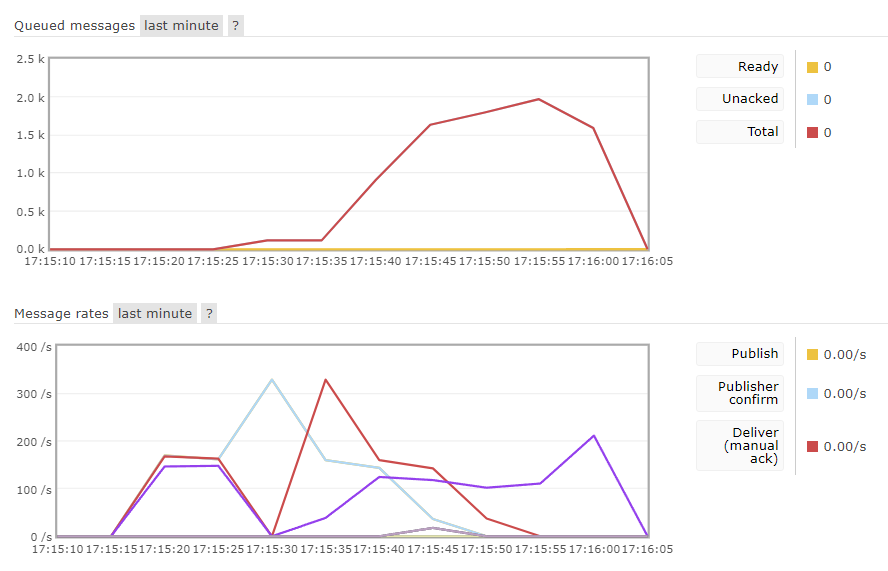
Максимальное количество запросов в очереди – 13, обрабатывались 40-70 сообщений в секунду. Время обработки всех сообщений – 2 минуты.



Увеличим параметр CPU для rabbitmq и web-составляющей, чтобы посмотреть, как processor станет «узким горлышком» системы.

**Ограничение по CPU для website, webapi: 1 CPU; для rabbitmq: 2 CPU, processor: 0.25 CPU**

Максимальное количество запросов в очереди – 2 тысячи, обрабатывались 100-200 сообщений в секунду. Время обработки всех сообщений – <1 минуты.



**Ограничение по CPU для website, webapi: 1 CPU; для rabbitmq: 2 CPU, processor: 0.1 CPU**

Максимальное количество запросов в очереди – 3.3 тысячи, обрабатывалось до 100 сообщений в секунду в пике. Время обработки всех сообщений – около 2 минут.



При значении 0.1 CPU для processor заметно, что он не успевает обработать такое количество сообщений. Увеличим производительность системы с помощью горизонтального масштабирования, добавив второй экземпляр processor с помощью --scale.

**Ограничение по CPU для website, webapi: 1 CPU; для rabbitmq: 2 CPU, processor (x2): 0.1 CPU**

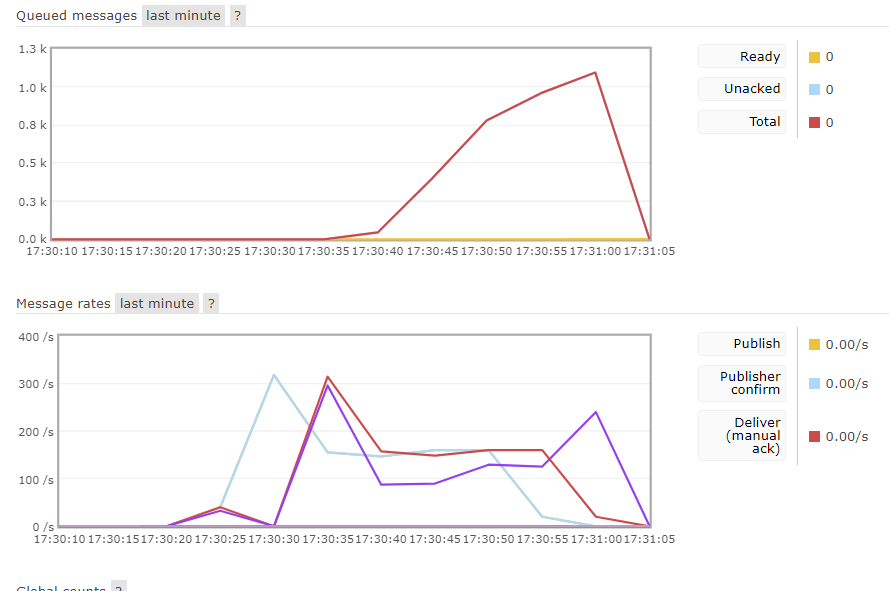
Максимальное количество запросов в очереди – 1.8 тысячи, обрабатывалось до 150 сообщений в секунду. Время обработки всех сообщений – 1 минута.



Увеличим число processor до 3.

**Ограничение по CPU для website, webapi: 1 CPU; для rabbitmq: 2 CPU, processor (x3): 0.1 CPU**

Максимальное количество запросов в очереди – 1.1 тысячи, обрабатывалось до 300 сообщений в секунду. Время обработки всех сообщений – <1 минуты.

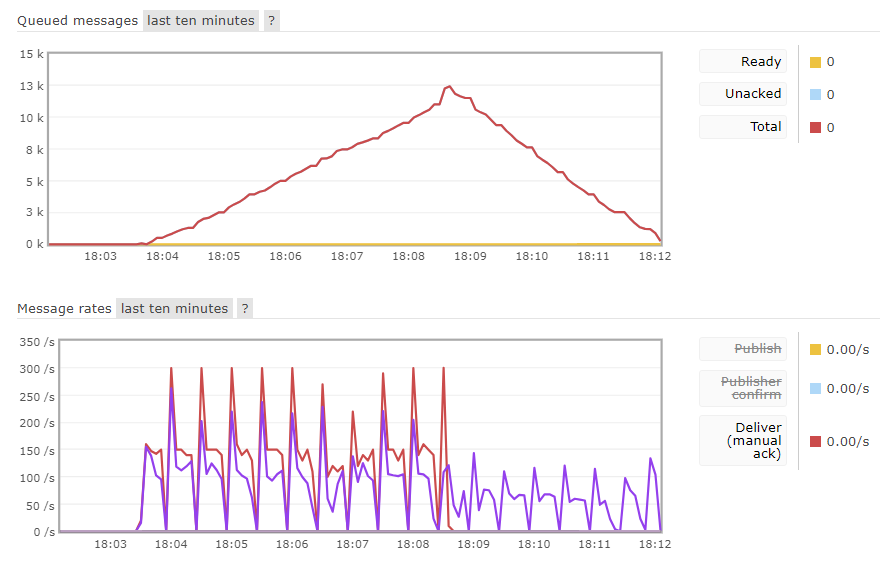


***Продолжительная нагрузка:***

Ограничение по CPU для website, webapi: 1 CPU; для rabbitmq: 2 CPU, processor (x3): 0.1 CPU

**Заданная скорость отправки запросов - 150 в секунду.**

Как видно, это слишком много для наших processor, даже в количестве 3 штук. Они успевают обрабатывать около 100 запросов в секунду. Попробуем выставить это число в качестве лимита скорости отправки запросов.



**Заданная скорость отправки запросов – 100 в секунду.**

В таком режиме в очереди находятся до 100 сообщений, их количество стабильно остается ниже данного значения. Попробуем немного увеличить количество запросов.



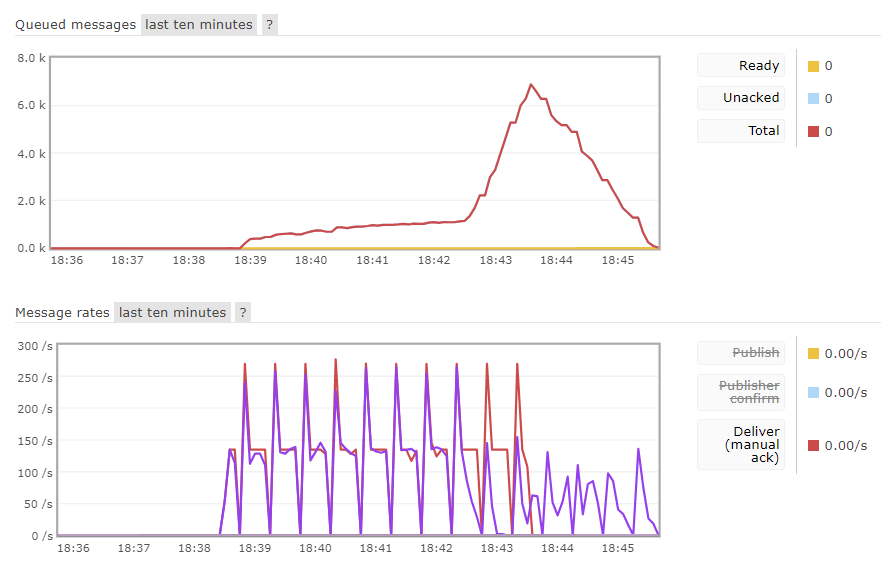
**Заданная скорость отправки запросов - 120 в секунду.**

В таком режиме в очереди находятся до 150 сообщений, их количество стабильно остается ниже данного значения.



**Заданная скорость отправки запросов - 135 в секунду.**

Очередь начинает постоянно расти, с резким увеличением до 7 тысяч запросов под конец, когда скорость обработки по какой-то причине упала до менее чем 100 сообщений в секунду.



*Заключение:*

В ходе лабораторной работы был разработан и протестирован проект системы онлайн-магазина, включающий в себя фронтенд, брокер очередей, API и обработчик запросов. Были проанализированы характеристики системы при различных ограничениях на ресурсы CPU, а также проведено нагрузочное тестирование с изменением конфигураций контейнеров, для двух типов нагрузки: единовременного всплеска активности и продолжительной нагрузки.

На основании экспериментов были сделаны следующие выводы:

1. Горизонтальное масштабирование эффективно увеличивает пропускную способность системы и дает возможность справляться с ростом запросов путем добавления новых контейнеров. Это позволяет лучше справиться с "узкими местами", распределив нагрузку по нескольким серверам или контейнерам, и быстрее обрабатывать запросы.
2. Система демонстрирует устойчивую работу при настройке оптимального числа процессоров и ресурсов для брокера сообщений, API и веб-части. Очередь сообщений позволяет обеспечить работоспособность проекта при жестких ограничениях производительности. Пользователям придется ждать чуть дольше (или не чуть в крайних сценариях), однако отказоустойчивость повышается в разы.

Данная система может обрабатывать до 120 запросов в секунду при 3 серверах-обработчиках с лимитом 0.1 CPU на каждом, что вполне достаточно для небольшого сервиса. Это число может масштабироваться как вертикально, например для 12-16 CPU на все обработчики для современных 6-8 ядерных/12-16 поточных процессорах в домашних системах, что позволит обрабатывать 4000-5000 запросов в секунду, так и горизонтально, в случае бОльших потребностей.

*Приложение:*

1. git репозиторий с исходным кодом:

<https://github.com/Arondy/PARVPO_lab3>

1. Скриншот с ошибкой реализации website на FastApi:

