Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет
“Высшая школа экономики”»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

Программный проект на тему

“Разработка сайта "Play.ClickHouse“”

Выполнил студент группы 176,
Калмыков Азат Асхатевич

Руководитель КР:
Руководитель разработки Clickhouse,
Миловидов Алексей Николаевич

Москва 2020

Оглавление

1 Цель

Цель проекта - создать веб-сайт, который позволит пользователям познакомиться с базой данных (БД) Clickhouse. На этом сайте должна быть возможность запускать произвольные SQL-запросы для Clickhouse и делиться результатами с другими пользователями.

2 Актуальность

Проект крайне актуален. Сервис может понадобиться обычным разработчиками, использующими Clickhouse, чтобы получше познакомиться с этой БД, чтобы эффективнее коммуницировать с другими разработчиками (например на StackOverflow). Сервис также найдёт применение и среди людей, которые создают Clickhouse, чтобы демонстрировать какое-то поведение и делиться друг с другом результатами.

3 Требования

У пользователей должны быть следующие возможности:

1. Запускать read и write SQL-запросы и получать результаты.
2. Делиться полученными по запросу результатами с другими пользователями с доступом по ссылке.
3. Запускать запросы на нескольких версиях Clickhouse.

Дополнительные технические требования:

1. Запрос по возможности не должен быть запущен ещё раз, нужно использовать кэширование результатов.
2. Сервис не должен вызывать сильные задержки при увеличении числа пользователей.

3. Пользовательский код должен быть изолирован. Пользователь не должен иметь возможность делать что-то помимо непосредственного взаимодействия с БД.

4 Задачи

Задачи, возникшие в течение выполнения работы:

1. Изучить документацию всех необходимых мне технологий (список далее)
2. Изучить код существующих открытых решений
3. Продумать архитектуру проекта, взаимодействие сервисов друг с другом
4. Написать основной бэкенд-сервис, принимающий пользовательские запросы.
5. Написать сервис, запускающий код на чистой БД Clickhouse.
6. Настроить взаимодействие сервисов через RabbitMQ.
7. Подключить базу данных для хранения результатов.
8. Упаковать проект в Kubernetes.
9. Настроить в Kubernetes масштабирование сервиса запуска кода.
10. Написать простой фронтенд, подключить к Kubernetes.
11. Запустить проект в Google Kubernetes Engine с доступом через внешний Интернет.

5 Существующие решения

Существует немало известных веб-сайтов, позволяющих запускать пользовательские SQL-запросы. Самых известных из них два - DB Fiddle и SQL Fiddle. У первого закрытый исходный код, поэтому он не представляет большого интереса для проекта КР.

У SQL Fiddle открытый исходный код, я изучил его на Github[1]. Проект работает на Kubernetes, который я также планировал использовать. Однако в проекте нет масштабирования, поэтому он не сможет выдержать возросшие нагрузки. Меня это не устраивало. К тому же код проекта было сложно изучать при моих начальных на тот момент знаниях в Kubernetes и смежных технологиях. По этим причинам я решил не оглядываться на исходный код этого проекта и действовать самостоятельно.

6 Реализация

6.1 Технологический стек

В своей работе я использовал множество технологий. Для применения каждой есть чёткое обоснование.

6.1.1 Golang

Я выделил следующие достоинства этого языка:

1. Простота. Этот язык прост в изучении и использовании, в нём нет сложных конструкций.
2. Типизированность. Это позволяет писать более понятный код, быстрее отлаживать проблемы.
3. Наличие библиотек для работы с распределёнными системами.

К тому же я был хорошо знаком с этим языком, мне не пришлось изучать почти ничего нового.

6.1.2 Docker

Требования к изолированности уже значат, что нужно использовать контейнеризацию. Docker - общепринятая технология для этого.

6.1.3 Kubernetes

Я предъявил требования к масштабируемости. С учётом предыдущего, это значит, что нужно использовать средство оркестризации контейнеров. Kubernetes - общепринятая технология для этого.

6.1.4 Helm

Helm позволяет облегчить задачу развёртывания приложений в Kubernetes. С помощью него можно легко установить разные компоненты в мой кластер. Например, базу данных.

6.1.5 RabbitMQ

RabbitMQ реализует очередь сообщений в распределённой системе. Благодаря нему мои сервисы могут взаимодействовать друг с другом нужным образом.

6.1.6 KEDA

Встроенные возможности Kubernetes в масштабировании довольно ограничены. Kubernetes может масштабировать сервисы в зависимости от нагрузки на процессор, от занятости оперативной памяти и ещё нескольких метрик. KEDA (Kubernetes Event-Driven Autoscaling) позволяет настроить масштабирование по многим другим метрикам. В том числе по количеству необработанных сообщений в очереди, мне нужен был этот вариант.

6.1.7 ReactJS

Я не тратил время на изучение достоинств и недостатков разных фреймворков, так как ко фронтенду не было сильных требований. Мне нужно было наличие большого сообщества у фреймворка, чтобы я мог быстро находить ответы на возникающие вопросы. Поэтому я выбрал ReactJS - по многим оценкам самый популярный фронтенд-фреймворк.

6.1.8 Google Kubernetes Engine

Мне нужно было выложить Kubernetes проект в Интернет. Для этого есть сервисы от разных облачных провайдеров. Я выбрал наугад один из самых популярных и работал с ним.

6.2 Архитектура

В процессе работы было опробовано 4 варианта реализации архитектуры. Первые 3 не сработали. Я расскажу о двух вариантах: предпоследнем и последнем, который и используется в итоге.

6.2.1 Неудачная реализация

Пользовательский запрос через фронтенд отправляется на основной бэкенд-сервис. Бэкенд-сервис через очередь сообщений пересылает SQL-запрос сервису executor. Executor состоит из 2 контейнеров, существующих в общем дисковом и сетевом пространстве. Первый контейнер - это клиент на Go, написан мной. Второй контейнер - это Clickhouse. Внутри и клиент, и сервер. Контейнер-клиент запускает бинарный исполняемый файл clickhouse-client из соседнего контейнера, подаёт ему на вход SQL-запрос и сохраняет вывод. Эти данные пересылается обратно в очередь и читаются оттуда основным бэкенд-сервером. Дальше пересылаются нужному пользователю.

Нам нужно каждый раз работать с чистым Clickhouse контейнером, поэтому мой клиент должен был после исполнения пользовательского запроса убивать соседа. После этого Kubernetes перезапускал соседа в чистом состоянии.

Однако эта схема не сработала, потому что Kubernetes не расчитан на такое использование. Он считает, что когда контейнер регулярно падает - это признак сбоя. Поэтому со временем он начинает откладывать перезапуск, из-за этого возникают огромные задержки.

Это поведение нельзя отключить в Kubernetes, поэтому мне в очередной раз пришлось перейти к другой схеме.

6.2.2 Итоговая реализация

Рассмотрим все компоненты из которых состоит система.

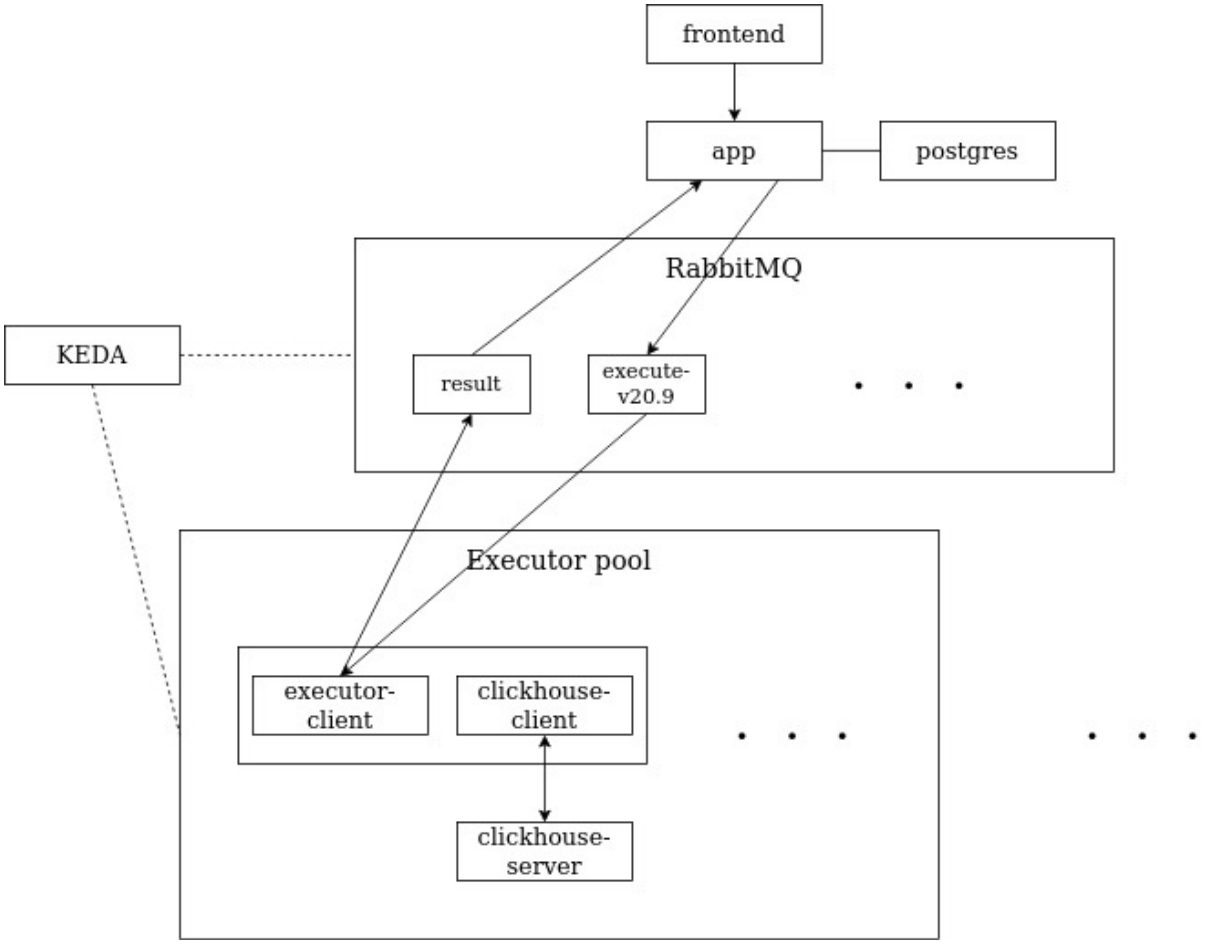

Рис. 6.1: Итоговая архитектура системы.

6.2.3 Микросервис app

Это главный сервис в бэкенде, с ним взаимодействует фронтенд.

В этом сервисе есть 2 обрабатываемых адреса: /api/exec, /run/id. Фронтенд присылает на первый адрес JSON с текстом SQL-запроса и версией Clickhouse, на которой нужно запустить этот запрос. Каждый пользовательский запрос обрабатывается в отдельной горутине (корутина в Golang). Таким образом задержки в обработке запроса одного пользователя скорее всего не повлияют на время обработки для другого.

При обработке запроса сервис отправляет сообщение с текстом запроса в очередь RabbitMQ, соответствующую нужной версии. При этом к каждому сообщению прикрепляется уникальный идентификатор запроса.

Сервис в отдельной горутине читает сообщения из очереди ответов. Она общая для всех версий Clickhouse. Ответ соединяется с нужным пользовательским запросом по упомянутому до этого идентификатору и отсылается,

куда нужно.

При обработке второго адреса сервис возвращает информацию об уже совершённом запуске. Это нужно, чтобы можно было делиться результатами по ссылке.

6.2.4 Микросервисы executor-{{versionID}}

У нас есть микросервис executor для каждой поддерживаемой версии Clickhouse. Он состоит из 2 контейнеров. В первом контейнере работает приложение на Go, написанное мной, назовём его executor-client. В соседнем контейнере работает клиент Clickhouse нужной версии.

Так как контейнеры существует на общем дисковом пространстве, executor-client может запустить исполняемый файл clickhouse-client из соседнего контейнера и с помощью него выполнить пользовательский код. Именно это он и делает для каждого прочитанного из нужной очереди сообщения. Потом возвращает результат в общую очередь ответов.

6.2.5 Фронтенд

Написан на ReactJS. Состоит из нескольких React компонент. Используются библиотека для роутинга, для отображения таблиц.

6.3 Разворачивание

Как уже было сказано, система живёт в Kubernetes кластере. Соответственно, всё разворачивание проходит с помощью Kubernetes.

Микросервис app составляет 1 deployment с 1 pod внутри.

Каждый микросервис executor-versionID составляет 1 deployment с переменным количеством pod внутри. Это количество регулируется автоскейлером от Keda, который используется в кластере. Автоскейлер использует информацию о количестве сообщений в очереди. Если их уже набралось несколько, то создаются дополнительные pod для того, чтобы справится с

нагрузкой.

Доступ к проекту из внешнего Интернета обеспечивается с помощью Kubernetes ingress. Таким образом фронтенд и бэкенд могут взаимодействовать друг с другом бесшовно.

Для упрощения процесса разработки я настроил разворачивание с помощью GNU make. В Makefile я прописал все команды, нужные для корректной инициализации кластера, и зависимости между ними. В частности, установку Postgres, KEDA, сборку микросервисов. Благодаря этому я могу запускать проект в Интернете одной командой make deploy, а сворачивать тоже одной командой make tear-down. Make сам разберётся, что конкретно ему нужно делать.

7 Итог

Получен рабочий продукт в альфа-версии. Все базовые требования выполнены. Написано около 2 тысяч строк кода. Продукт понравился разработчикам Clickhouse.

8 Дальнейшая работа

Далее продукт будет доделан до production-ready состояния и выложен на explorer.clickhouse.tech с помощью проксирования запросов.

Список источников

1. Исходный код SQLFiddle [Электронный ресурс] : репозиторий на GitHub. URL : https://github.com/zzzprojects/sqlfiddle3 .