УДК 004.9

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА СЕРВЕРНОГО ПРИЛОЖЕНИЯ ДЛЯ УПРАВЛЕНИЯ КЛАСТЕРОМ KUBERNETES

Кондрашов Е. Ю. (Университет ИТМО, Санкт-Петербург)
Научный руководитель – ассистент, Терёшкин С. Е.
(Университет ИТМО, Санкт-Петербург)

Введение. В связанных с информационными технологиями компаниях может возникать необходимость предоставлять своим сотрудникам доступ к какому-либо ПО в облаке. Например, специалисту Data Science могут понадобиться развернутые JupyterHub и БД Clickhouse, обрабатывающие достаточно тяжелые данные. В таком случае личный ПК сотрудника становится затруднительно использовать из-за недостаточного количества вычислительных ресурсов. Для решения этой проблемы компания может выбрать платную подписку на SaaS-платформу; однако, если у неё есть свои серверы (как у ИТМО), то она может разворачивать экземпляры ПО на них, используя Kubernetes.

При выборе второго варианта для безопасности и удобства использования конечным пользователем требуется приложение, которое будет управлять кластером Kubernetes, учитывая необходимую бизнес-логику. В частности, оно должно создавать группы объектов Kubernetes в кластере, позволяя пользователю менять строго определённые параметры конфигурации.

Кроме того, пользователям приложения необходимо получать данные об актуальном состоянии объектов в кластере Kubernetes. Такую информацию можно получить через Kubernetes API Server.

Однако для внутреннего взаимодействия объекты Kubernetes тоже используют API Server. Большая нагрузка от пользователей приложения может привести к задержкам во внутренней работе Kubernetes; тем более, запросы на получение списков объектов обрабатываются Kubernetes API Server в разы дольше, чем остальные запросы [1].

Основная часть. Для запуска экземпляра ПО под конкретного пользователя в коммерческих SaaS решениях (Google Collab, Yandex Managed Clickhouse и др.) обычно используются виртуальные машины. Однако в рамках работы во внутреннем контуре компании использование Kubernetes имеет преимущества: отсутствуют накладные расходы создания виртуальных машин и проблемы динамического распределения ресурсов.

В качестве посредника между пользователем приложения и кластером будем использовать бэкенд на Python FastAPI. Для взаимодействия с API Kubernetes через Python можно использовать клиентскую библиотеку kubernetes.

Поскольку для запускаемого экземпляра требуется постоянное хранилище и доступ в сеть, то в кластере Kubernetes нужно создать ряд объектов: Deployment, PersistentVolume, PersistentVolumeClaim, Service, Namespace. Некоторые свойства этих объектов должны различаться в зависимости от пользователя (например, ярлыки). Кроме того, хочется дать пользователям возможность менять определённые параметры конфигурации (версию Docker-изображения для Jupyter, количество доступных ресурсов).

Существует готовое ПО для достижения похожих задач. Например, с помощью Postgres Operator можно управлять PostgreSQL в кластере Kubernetes. Однако такие операторы имеют множество дополнительного, далеко не всегда нужного функционала. К тому же интегрировать их в общую логику приложения сложнее, чем в случае описанного далее решения.

Создание объектов в Kubernetes можно реализовать с помощью конфигурационных файлов в формате YAML. Чтобы объекты различались по необходимым признакам, можно применить шаблоны конфигурационных файлов, которые заполняются проверенными аргументами с помощью шаблонизатора Jinja2. Валидация входных данных можно осуществить с помощью моделей Pydantic.

Уменьшить нагрузку на Kubernetes API Server можно с помощью кэширования. В качестве технологии для кэширования выбран Redis, так как он обеспечивает низкие задержки при работе с данными в формате JSON, среди прочего из-за их хранения в оперативной памяти [2].

Одним из популярных подходов является кэширование полного ответа сервера в зависимости от URL и параметров запроса. Однако в случае со списком подов (абстрактная единица в кластере Kubernetes) этот подход не даст ощутимого прироста производительности.

Состояние подов должно кэшироваться на небольшой срок, чтобы быть актуальным. Список подов всегда запрашивается с фильтрами: по пользователю и/или проекту. Из-за этих двух причин велик шанс, что в кэше не будет нужного пользователю ответа.

Таким образом, при кэшировании полного ответа сервера кэш будет покрывать только малую часть запросов. Основное же их количество всё-равно будет приходиться на Kubernetes API Server.

Как решение, была предложена и реализована следующая стратегия кэширования. Периодически данные по всем подам из кластера запрашиваются через Kubernetes API Server, а затем сохраняются в Redis по каждому поду отдельно.

Чтобы формировать ответы пользователям, используются запросы к хранилищу Redis, которые обрабатываются Redis Search - частью Redis Stack. Далее, библиотека Redis OM позволяет переводить данные, хранящиеся в Redis, в объекты Python, что удобно для работы с ними в FastAPI.

Выводы. Практическим результатом этой работы является разработанный бэкенд микросервис, который управляет состоянием кластера Kubernetes, учитывая необходимую бизнес логику, а также осуществляет мониторинг состояния кластера.

С помощью разработанной системы кэширования нагрузка от выполняемых пользователями приложения запросов, касающихся состояния кластера Kubernetes, перекладывается на Redis и не замедляет работу кластера.

Благодаря, в том числе, этому микросервису, IT-компания может иметь альтернативу платным SaaS-платформам для запуска ПО в “облаке” при наличии собственных серверов. Приложение, частью которого является разработанный микросервис, используется внутренними командами университета ИТМО, а также индустриальным партнёром.

Список использованных источников:

1. Раздел с описанием нагрузки на Kubernetes API Server от различных запросов в документации Kubernetes [Электронный ресурс]. - URL: https://kubernetes-io.translate.goog/docs/concepts/cluster-administration/flow-control/?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=sc#seats-occupied-by-a-request (дата обращения 02.02.2023)
2. Бенчмаркинг Redis JSON в сравнении с MongoDB и ElasticSearch [Электронный ресурс]. - URL: https://redis.com/blog/redisjson-public-preview-performance-benchmarking/ (дата обращения 14.02.2023).