Федеральное государственное образовательное бюджетное учреждение

высшего образования

«Финансовый университет

при Правительстве Российской Федерации»

Департамент информационных технологий и анализа больших данных

Проектная работа по дисциплине «Программная инженерия»

по теме

**«Реализация микросервисной архитектуры для удовлетворения требований бизнеса (ресторана) с использованием облачных технологий»**

Выполнил:

студент группы ПИ21-3

Супрунов Матвей

Проверил:

Маслов С. С.

Москва 2024

## **Цель работы**

Реализовать контейнеризованные микросервисы на Python для преобразования потоковых данных ресторана и развернуть их в оркестраторе контейнеров.

## **Задачи работы**

1. Считать потоковые данные с помощью брокера сообщений, преобразовать их и направить в хранилище данных.
2. Поднять необходимые сервисы для проекта в облаке.
3. Организовать контейнеризацию сервисов и реализовать менеджирование (оркестрацию) подов с контейнерами с помощью оркестратора контейнеров для обеспечения бесперебойной работы контейнеров и перераспределения ресурсов в зависимости от нагрузки.

## **Терминология**

* **Требования бизнеса** - описание глобальных целей и нужд проекта с точки зрения пользы для бизнеса.
* **Нефункциональные требования** - технические детали по задаче.
* **Функциональные требования** - ключевые детали задачи, которые надо знать инженеру для разработки решения. Операции стриминг-сервиса с данными. Среди основных: десериализация, фильтрация, дедупликация, трансформация, обогащение, сериализация, таргет. Нефункциональные — технические особенности решения. В их числе допустимая временная задержка, пропускная способность системы, масштабируемость, отказоустойчивость, допустимая потеря данных.
* **Пользовательские требования** - рамочно сформулированная задача по настройке системы, которая позвоолит реализовать бизнес-цели.

## **Функциональные требования**

На вход подаются данные формата JSON. В каждом заказе есть категория. Для категорий надо завести счётчики с конкретными порогами. Все расчёты ведутся только по закрытым заказам со статусом CLOSED - фильтрация.

В хранилище данных будут реализованы следующие слои:

* **Staging (STG)** - слой с исходными данными (as-is). Необходим для отслеживания причины ошибок или восстановления потерянной информации.
* **Detail Data Store (DDS)** - слой детализированных данных для содержания информации в удобном для управления виде – обогащение данных.
* **Common Data Marts (CDM)** – витрины с атрибутами, обозначающими значимые бизнес-метрики. В CDM нужны две витрины. Первая витрина — счётчик заказов по блюдам; вторая — счётчик заказов по категориям товаров.

## **Нефункциональные требования**

Информация по заказам — это потоковые данные, которые будут передаваться через брокер сообщений. Нагрузка на систему заказов — 5 заказов в минуту.

Необходимо обеспечить идемпотентность обработки сообщений из брокера (свойство системы, когда на одни и те же входные данные система выдаёт одинаковый результат. В нашем случае при повторной обработке сообщения состояние БД не должно измениться).

Во всех схемах взаимодействия важны контракты: структура сообщений, отправляемых в Kafka, должна быть фиксированной, чтобы обработчик смог распарсить сообщение.

## **Подбор технологий**

В DWH данные будут передаваться через Kafka. Данные, полученные через Kafka, буду хранить в PostgreSQL. В сообщениях есть id, которые необходимо будет перевести в имена. Например, блюда, рестораны, пользователи. Для этого потребуется key-value хранилище Redis. Для контейнеризации сервисов буду использовать Docker. Для их оркестрации хорошая связка для Docker – Kubernetes. В качестве пакетного менеджера для удобного взаимодействия с манифестами и версионирования релизов буду использовать пакетный менеджер Helm.

Бизнес-задача — «тегирование гостей».

С функциональной точки зрения структура хранилища стандартная: слои STG, DDS, CDM.

Особенности слоёв:

* В STG — исходные данные as is.
* В CDM — две витрины. Первая витрина — счётчик заказов по блюдам; вторая — счётчик заказов по категориям товаров.
* В DDS — модель данных Data Vault.

Данные из системы-источника передаются по двум каналам:

* Первый канал — это поток заказов, который идёт в Kafka (5 заказов в минуту).
* Второй канал — это словарные данные (блюда, рестораны, пользователи), которые идут в Redis.

В качестве БД используется PostgreSQL. Логику обработки данных нужно написать на Python, она будет разворачиваться в Kubernetes. Брокер сообщений как на вход, так и для обмена данными между сервисами — Kafka.

Построение дашборды на основе витрин данных в DWH.

## **Оценка ресурсов**

Исходя из требований для данной задачи было принято решение использовать минимальные ресурсы сервисов в облаке, однако для полноценной реализации потребуется отказоустойчивая система (превосходящая текущие характеристики как минимум в 5 раз) с небольшим количеством ресурсов в запасе на случай сбоев.

Текущие технические характеристики сервисов:

* **PostgreSQL** - 2 vCPU, 50% vCPU rate, 4 GB RAM, 16 GB network-hd
* **Redis** - 2 vCPU, 50% vCPU rate, 4 GB RAM, 10 GB network-ssd
* **Kafka** - 2 vCPU, 50% vCPU rate, 4 GB RAM, 12 GB network-ssd

## **Проектирование решения**

1. Сначала нужно поднять сервисы, в которые поступают входные данные (Redis, Kafka). С помощью этого получится сразу изучить оригинальную информацию
2. Поднять PostgreSQL в облаке. Развернуть инфраструктуру, чтобы все части DWH были готовы к разработке.
3. Завести Container Registry — инструмент для запуска сервисов.
4. Создать шаблон сервиса и выкатить в Kubernetes. Отладить процесс сборки и релиза.
5. Поочередно заполнить слои хранилища

Полная архитектура решения:

A diagram of a service

Description automatically generated

В проектной работе будет реализована микросервисная архитектура, предусмотрен асинхронный способ взаимодействия сервисов.

Сервис-отправитель посылает сообщение в систему-посредник — это система класса «Очередь», например Kafka.

Сообщение сохраняется в Kafka до тех пор, пока сервис-обработчик не будет готов эту информацию принять.

В удобный момент обработчик вычитывает сообщение из Kafka. Иногда сервис может отправить сообщение обратно, также через очередь.

A diagram of a diagram

Description automatically generated

## **Стратегия выбора модели данных для слоя DDS**

В качестве модели данных для данной проектной работы изначально рассматривались подходы Билла Инмана, Ральфа Кимбалла, Data Vault и Anchor Model. После анализа качества интеграции моделей в текущую задачу было принято решение использовать Data Vault, так как организация DDS в виде Data Vault позволяет параллельно загружать данные. Можем сначала прогрузить хабы, потом линки для них, потом их сателлиты. Все это намного оптимальнее и быстрее, чем классические цепочки ETL в классических хранилищах по Кимбаллу и Инману.

Можно также выделить такие плюсы, как:

* Удобный аудит преобразований данных.
* Быстрый старт – не надо долго думать над моделью и проектировать ее целиком
* Гибкость и расширяемость модели данных – можно легко добавлять новые сущности и атрибуты
* Нет избыточности информации
* Данные загружаются быстрее, но считываются медленнее. Лучше для OLTP систем.

Если же говорить об Anchor Model, то в ней модель данных состоит из элементов трёх типов: хабов, таблиц связей и сателлитов. Все отношения должны быть нормализованы до 6НФ. В неизменяемом хабе может быть только один элемент — суррогатный ключ. Считаю, что нет необходимости декомпозировать объекты базы данных до минимума. В таком случае у нас итоговое количество отношений станет равно изначальному количеству атрибутов в нашей схеме, что вызовет дополнительную нагрузку на кластер вследствие значительного количества соединений таблиц.

Способы проектирования Кимбалла и Инмана не подходят, так как если модели модифицируются и расширяются, то возникают проблемы. Приходится пересобирать и заново проектировать, дорабатывать детали процесса сборки. Также в них есть проблемы, связанные с денормализацией данных, данные не в конечной НФ.

Модель данных Data Vault в слое DDS:

A screenshot of a computer

Description automatically generated

## **Планирование логики сервиса**

Вкратце: чтение сообщений из Kafka, наполнение stg-слоя, обогащение данными из Redis и отправка обогащённых сообщений.

Будет построена система, которая состоит из обработчиков сообщений. Каждый сервис — это отдельный обработчик, который получает и отдаёт сообщение. При этом в процессе обработки сервис порождает новые артефакты, оседающие в системе, — например, заполняет таблицы разных слоёв данных в БД.

Формат входного сообщения для STG-слоя из Kafka - JSON. Выходное сообщение должно содержать всю необходимую информацию о передаваемых объектах. Части этой информации нет во входном сообщении, а именно: данных о покупателях, ресторане (лишь id) и продукте, т. е. заказанном блюде. Эту информацию необходимо будет доложить.

С заданной периодичностью планировщик будет запускать функцию run, внутри которой вычитываются и обрабатываются сообщения из Kafka. Таким образом, будет реализован micro-batch processing — обработка небольших пакетов сообщений.

## **Обработка сообщения**

* Сложить исходное сообщение в PostgreSQL в таблицу stg.order\_events – это действие обязательно надо реализовать в сервисе. STG-слой будет источником правды и хранилищем всех сообщений.
* Обогатить сообщение данными из Redis: забрать информацию из Redis по id объекта.
* Отправить обогащённое сообщение в Kafka, но в другой топик. Данные понадобятся следующему сервису, заполняющему DDS-слой.

Подход для обработки повторов – upsert (в PostgreSQL – ON CONFLICT(\_\_\_) DO UPDATE SET \_\_\_ = EXCLUDED.\_\_\_). Он позволяет проверить существование строки в БД с тем же object\_id. Если существует — обновить.

Порядок действий при обработке сообщения:

1. Сложить сообщение as-is в БД по логике upsert.
2. Достать user\_id из сообщения.
3. Получить user из Redis по user\_id.
4. Достать restaurant\_id из сообщения.
5. Получить restaurant из Redis по restaurant\_id.
6. Для каждого product\_id в сообщении:
7. Достать product\_id.
8. Получить product из Redis (нужна категория).
9. Сформировать выходное сообщение.
10. Отправить сообщение в Kafka.

Ссылка на dashboard: <https://datalens.yandex.com/fg5q9yvb5d723-yandexstreamdash>

## **Вывод**

В результате данной проектной работы была выполнена ее цель, а именно была реализована система микросервисов обработки сообщений с данными от ресторана, состоящая из 3 слоев. Все 3 сервиса были упакованы в контейнеры и обернуты с помощью оркестратора контейнеров Kubernetes и управляются пакетным менеджером Helm.

## **Приложения**

1. Поднятие сервиса в локальном докере

A screenshot of a computer

Description automatically generated

A black screen with blue text

Description automatically generated

A screenshot of a running program

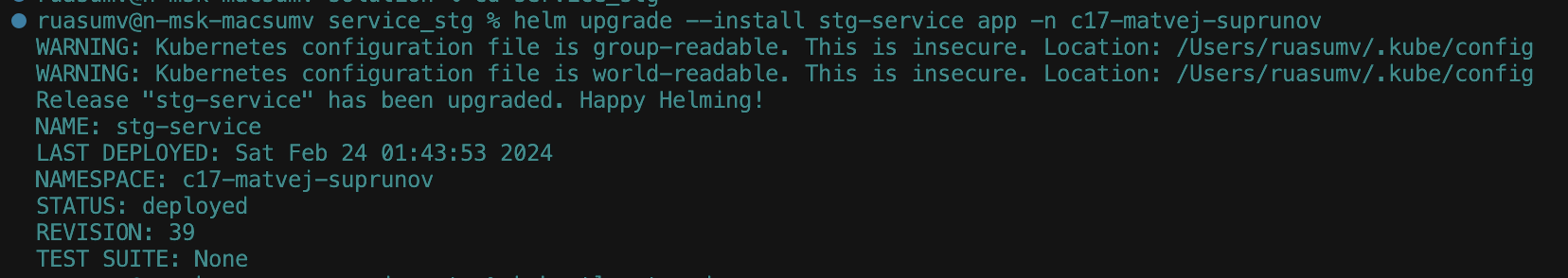
Description automatically generated

1. Пуш контейнера в Container Registry

A computer screen with white text

Description automatically generated

1. Релиз сервиса с помощью пакетного менеджера Helm



1. Проверка того, что сервис запущен, с помощью kubectl

A screenshot of a computer

Description automatically generated

1. Наполнение витрин данных в хранилище

A screenshot of a computer

Description automatically generated