Соединяем несоединимое

Давно не писал ничего умного и полезного. И вот время пришло.

Что, если у нас есть две системы, которые должны обрабатывать одни и те же данные, но только с разными: форматом данных, логикой их обработки и способом хранения?

Предположим, для простоты начала, что события изменения данных в одной из систем являются источником новых данных для второй системы. И, что система-источник – это наша система, к кодовой базе которой у нас есть доступ.

Чтобы не было совсем просто, пусть система-приёмник будет для нас “чёрным ящиком”, у которого нам доступен только REST API для взаимодействия с системой-источником.

Наша задача – подружить эти две системы, чтобы их взаимодействие было асинхронно и масштабируемо.

Какие у нас есть варианты?

1. Мы можем при каждом изменении данных (создание, изменение, удаление) в системе-источнике, отправлять REST-запрос в систему приёмник. Для этого нам придётся внести много изменений в код системы-источника, что потребует значительных затрат на разработку и тестирование, т.к. гарантированно увеличит количество багов в этой системе. Всё это нам придётся повторять при добавлении каждого нового типа данных, который мы захотим передавать в систему-приёмник.

В случае небольшой кодовой базы системы-источника и небольшого количества типов передаваемых данных, этот вариант нам мог бы подойти. Но трезвое видение текущей ситауции показывает, что это не наш случай.

2. Мы можем отслеживать изменения записей в таблицах базы данных системы-источника. Так мы избежим сильного вмешательства в код системы-источника. Можем даже написать отдельное приложение, которое будет этим заниматься, чтобы совсем не трогать систему-источник.

Этот вариант уже ближе к нашему видению прекрасного. Как мы можем его реализовать?

2.1. Можно обложить нужные нам таблички в базе данных триггерами, написать там же функции и создать таблички, в которые эти функции будут записывать все изменения нужных нам данных. Из нашего приложения мы будем периодически опрашивать таблички с изменениями на наличие новых записей.

Этот вариант сильно похож на п.1. С той лишь разницей, что мы переносим разработку в базу данных и нагружаем работой базу данных. На Хабре есть статья на близкую тему (https://habr.com/ru/post/317866/) , прочитав которую, мы можем прийти к пониманию того, что в большой системе с большим количесвом данных лучше так не делать.

2.2. Можно использовать Kafka Connect. Эта штука умеет интегрировать между собой различные базы данных, не требует вмешательства в кодовую базу системы-источника или системы-приёмника.

Но у нас система-приёмник, которая не умеет принимать ничего, кроме REST-запросов. Помните?

Не беда. Получать данные от Kafka Connect из топиков брокера Kafka будет отдельное приложение, а затем оно будет передавать их системе-приёмнику REST-запросами.

Хорошо. Пока нет противоречий с тем, что нам нужно. Исследуем вопрос дальше.

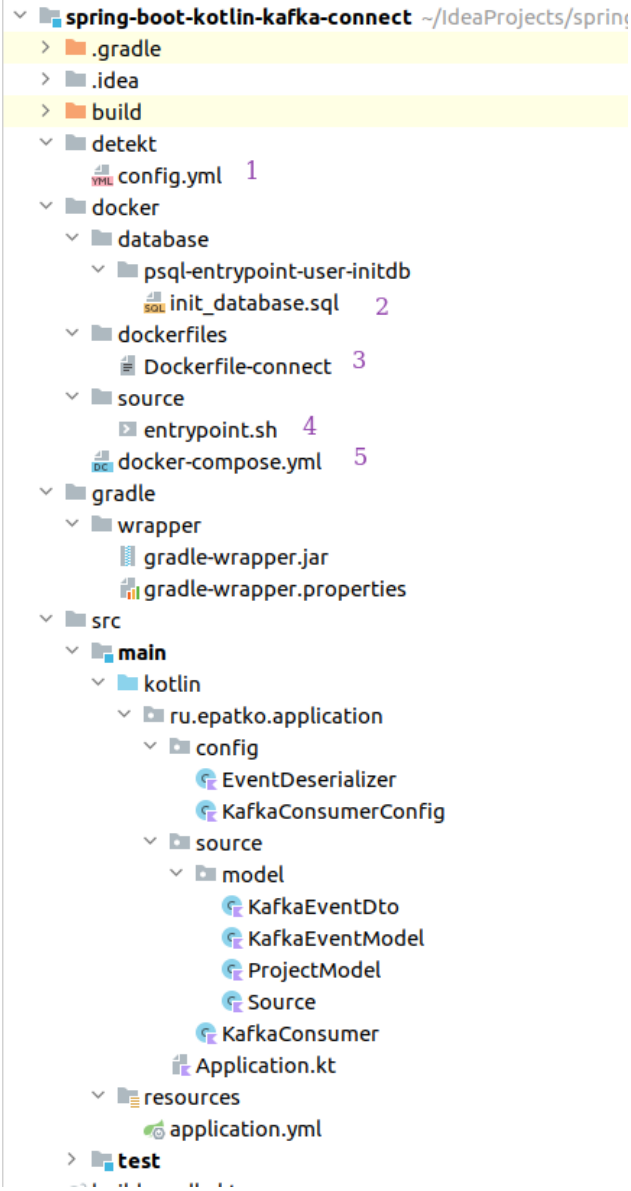
По Kafka Connect есть подробная документация (https://docs.confluent.io/platform/current/connect/userguide.html#how-to-use-kconnect-long-getting-started). Основное оттуда:

- для подключения к базам данных Kafka Connect использует плагины;

- для базы данных системы-источника (PostgreSQL) плагины есть: jdbc и debezium. Берём debezium, чтобы не мучить базу данных лишними запросами,

- документация по плагину debezium тоже в отличном состоянии (https://debezium.io/documentation/reference/stable/connectors/postgresql.html#debezium-connector-for-postgresql).

Собственно, этого уже достаточно, чтобы на коленке, за 5 минут, накидать MVP.



Вот такой у нас получился MVP.

Из необычного в нём:

1 – конфиг для статического анализатора кода на Kotlin

2 – скрипт инициализации базы данных, к которой будет подключаться коннектор

3 – кастомный Dockerfile с образом коннектора и нужным плагином. Решил сделать так, чтобы не собирать каждый раз новый образ Kafka Connect с плагином. Помните: Kafka Connect – отдельно, плагины – отдельно?

4 – скрипт запуска Kafka Connect, который передаёт ему конфиг для подключения к базе данных

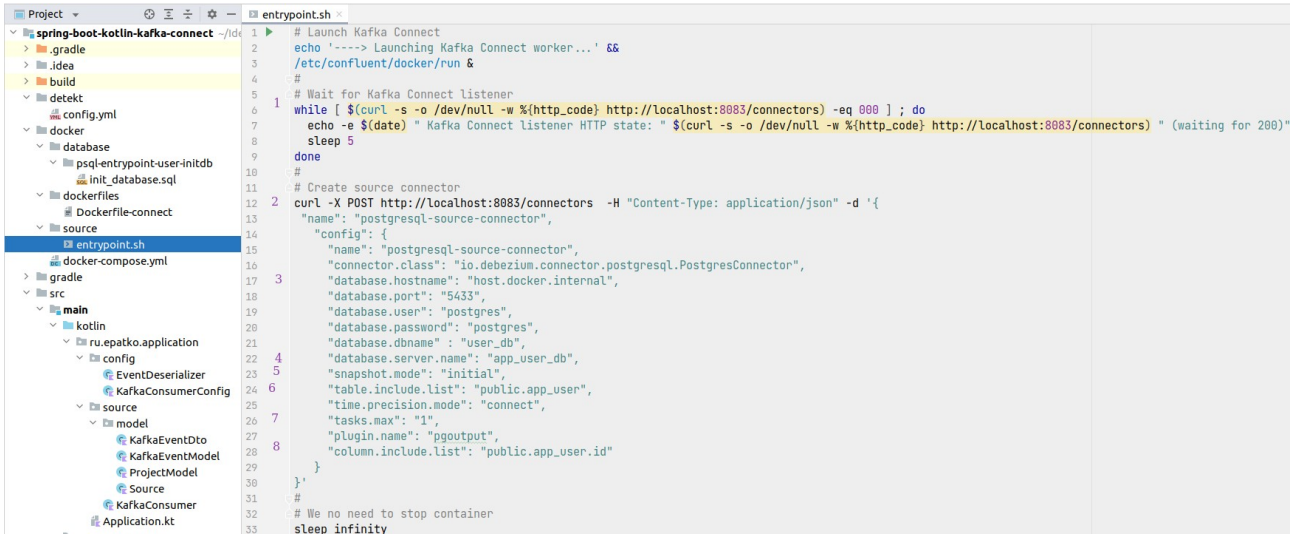
5 – конфигурация Docker Compose для запуска нашего MVP. Все сторонние продукты, которые нужны нам для теста MVP, мы запустим в Docker Compose.

Остальные части проекта должны быть более/менее понятны вем, кто пишет на Java/Kotlin.

Содержимое файла Dockerfile-connect.

Здесь мы пишем инструкцию по сборке Docker-образа Kafka Connect с нужным нам плагином debezium-connector-postgresql:1.9.3.

Образ будет собран один раз при первом запуске всего окружения в Docker Compose. И при последующих запусках будет браться из нашего Docker Registry.



Содержимое файла entrypoint.sh.

1 – скрипт, который проверяет: запущен ли брокер Kafka. Без брокера Connect работать не сможет.

2 – через REST API Kafka Connect передаём ему конфиг для подключения к базе данных.

3 – имя хоста с базой данных. В данном случае, host.docker.internal – это псевдоним localhost.

4 – псевдоним сервера базы данных. К реальному названию сервера этот псевдоним не имеет отношения. Можно указать любое имя. Главное – не забыть, что оно есть, когда будем искать топики с событиями, которые нагенерировал Kafka Connect.

5 – способ создания снэпшотов базы данных.

6 – список таблиц, которые мы отслеживаем.

7 – уровень параллелизма. Указывает, сколько инстансов Kafka Connect одновременно обрабатывает события изменения отслеживаемых таблиц. Почему здесь “1”? Портому что нам важно, чтобы события были положены в топик брокера и доставлялись в систему-приёмник в той же последователььности, в которой они происходят в базу данных системы-источника.

8 – список колонок отслеживаемой таблицы, которые Kafka Connect передаст в топик брокера в случае изменения записи (строки) в таблице. Почему у нас передаётся только id? Это связано с особенностями работы debezium-connector-postgresql и postgresql. В случае

выполнения над таблицей операций INSERT или UPDATE, коннектор отправить в топик значения из всех колонок, которые мы в этом месте укажем. В случае выполнения операции DELETE, в топик попадёт только replica identity записи (в данном случае – это id).

Если бы у нас стояла задача просто скопировать содержимое базы данных системы-источника в базу данных системы-приёмника, то мы могли бы в п.8 перечислить все колонки, которые хотим скопировать, включая id и передавать при операциях INSERT или UPDATE – все колонки, а при DELETE – только id. И нам этого хватило бы. Но у нас две перпендикулярные системы с разной логикой и способами хранения данных. В случае удаления записи из отслеживаемой таблицы системы-источника, система-приёмник получает id далённой записи. Такого id у неё нет. В её системе хранения данных у хранимых сущностей вообще нет такого id, как в базе данных системы-источника. А ничего другого, кроме id, debezium-connector-postgresql и postgresql нам не передадут.

Поэтому, в целях упрощения и универсализации:

- при любых операциях в базе данных системы-источника получаем id записи;

- по id достаём изменённую запись (или её копию – в случае удаления) из базы данных источника;

- готовим из того, что достали, данные, которые поймёт система-приёмник и отправляем их ей.

init_database.sql

```sql
select 'create database user_db'
where not exists(select from pg_database where datname = 'user_db')
\gexec


\c user_db;


alter system set wal_level = 'logical';


create table if not exists app_user
(
    id          bigserial primary key,
    name        varchar(100),
    avatar      varchar(100),
    position    varchar(100),
    created_at  timestamptz default current_date,
    modified_at timestamptz default current_date
);

create table if not exists app_user_deleted
(
    id              bigserial primary key,
    user_id         bigint,
    name            varchar(100),
    avatar          varchar(100),
    position        varchar(100),
    delivery_status varchar(20)
);


insert into app_user (id, name, avatar, position)
values (1, 'UserName-1', 'UserAvatar-1', 'UserPosition-1'),
       (2, 'UserName-2', 'UserAvatar-2', 'UserPosition-2'),
       (3, 'UserName-3', 'UserAvatar-3', 'UserPosition-3')
on conflict do nothing;


create or replace function add_app_user_deleted() returns trigger AS
$$
begin
    insert into app_user_deleted (user_id, name, avatar, position)
    select id, name, avatar, position from app_user where id = OLD.id;
    return old;
end;
$$ language plpgsql;


drop trigger if exists delete_app_user on app_user;


create trigger delete_app_user
    before delete
    on app_user
    for each row
execute procedure add_app_user_deleted();

```

Готовим базу данных системы-источника.

В прошлом посте мы придумали, как передавать изменённые записи из базы данных источника, получая от коннектора и базы данных только id этих записей.

Но что насчёт удалённых записей? Их id мы получим уже после того, как они были удалены. И где нам их тогда искать, если их уже нет в отслеживаем таблице?

Конечно же, в таблице, куда они будут скопированы при удалении!

На скрине – содержимое файла init_database.sql, который:

1 – создаст нам базу данных,

2 – подключится к ней

3 – установит уровень логирования базы данных в “logical” – необходимо для работы debezium-connector-postgresql. Для переключения уровня логирования потребуется перезапустить базу данных. Этот уровень логирования накладывает ряд требований к создаваемым в базе данных таблицам. Основное – каждая таблица должна иметь replica identity (идентификатор репликации). В простейшем случае – это id записи. За более сложными случаями – добро пожаловать в документацию PostgreSQL.

4 – создаст нам таблицу app_user с данными,

5 – создаст таблицу app_user_deleted, в которую будут копироваться удаляемые записи,

6 – создаст функцию, которая будет копировать удаляемые записи,

7 – создаст для таблицы app_user триггер на удаление записей.

Условия и порядок запуска приложения подробно описаны в README.md проекта.