ИТМО

Итоговый проект

Группа M3203
Лакеев Георгий

Методы анализа данных
Университет ИТМО
Санк-Петербург, Россия

29 июня 2024 г.

Оглавление

Подготовка окружения

Docker — это проект с открытым исходным кодом для автоматизации развертывания приложений в виде переносимых автономных контейнеров, выполняемых в облаке или локальной среде. Он позволяет изолировать исполняемую программу со всеми ее зависимостями от операционной системи, таким образом однажды собранный контейнер можно запустить на любой машине, где установлен докер.

Прежде всего необходимо собрать контейнер докера, для этого необходим образ. Docker image (образ) - это шаблон, который содержит все необходимое для запуска приложения, помещенного в контейнер:

1. Код
2. Среду выполнения
3. Зависимости
4. Переменные окружения
5. Конфигурационные файлы

В Dockerfile записываются команды и опции создания образа, а также некоторые настройки будущего контейнера, такие как порты, переменные окружения и другие опции. Крупнейшим хранилищем образов является Docker Hub, который содержит более 100000 образов контейнеров от поставщиков программного обеспечения, а также проекты с открытым исходным кодом и сообщества. Docker Compose - это инструмент для определения и запуска многоконтейнерных приложений. Это ключ к упрощению и эффективности процесса разработки и развертывания. Compose упрощает управление всем вашим стеком приложений, упрощая управление службами, сетями и томами с помощью единого понятного файла конфигурации YAML.

Так как необходимо развернуть СУБД PostgreSQL, то установим образ PostgreSQL на локальной машине. Для запуска bash скриптов, генерации данных, исполнения файлов миграции, определения переменных среды и указания томов необходимо сконфигурировать docker-compose.yaml файл.

```yaml
version: '3.9' # версия docker compose

services: # Определяем сервисы, которые необходимо развернуть

  db:
    image: postgres:alpine3.19 #Версия образа из Docker Hub
    container_name: business_broker # Имя контейнера
    environment: # Определяем переменные среды
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
      MIGRATION_VERSION: ${MIGRATION}
    volumes: # Определяем тома
      - ./.env:/docker-entrypoint-initdb.d/env
      - ./Data generators:/docker-entrypoint-initdb.d/Data generators
      - ./Migrations:/docker-entrypoint-initdb.d/Migrations
      - ./initialize.sh:/docker-entrypoint-initdb.d/1.sh
    ports: # Определяем порты локальной машины и внутренние порты СУБД
      - 5433:5432
```

Создание отношений

На этом этапе необходимо добавить отношения в БД. За основу возьмем нормализованную на 3 нормальную форму схему БД из предыдущего этапа.

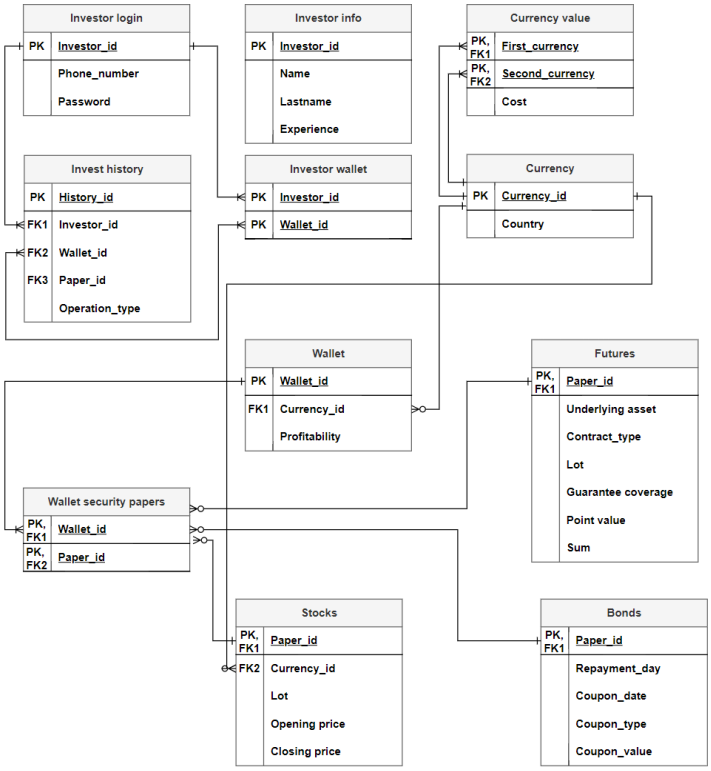

Рис. 2.1: Схема базы данных

Для добавления отношений необходимо определить идемпотентые файлы миграции. Файлы миграций должны соответствовать конверсии о семантическом версионировании. Так же при создании нового контейнера миграции должны исполняться автоматически. Каждая новая правка или исправление должна быть в новом файле.

В docker-compose.yaml мы указывали тома, одним из которых является папка с описанием файлов миграции, а также конфигурационный файл .env, в котором указываются версия миграции, сколько данных необходимо нагенерировать в каждую таблицу, а также пользователи для регистрации в бд. Для инициализации будем использовать bash-скрипт: initialize.sh,. Этот скрипт считывает перменные среды из .env, запускает файлы миграции, sql-генерацию значения для таблиц и добавляет пользователей. initialize.sh копируется в директорию /docker-entrypoint-initdb.d/, СУБД при разворачивании запускает скрипты, лежащие в этой директории, таким образом после запуска docker-compose.yaml у нас поднимется БД, создадуться отношения и заполнятья значениями.

Значения и роли

Как уже было сказано, генерация значений происходит на этапе инициализации СУБД. Все данные создаются при помощи языка SQL и гарантируется соблюдения всех ограничений БД, например foreign key или размера типа varchar(100). Так как разные файлы миграции задают разные состояния БД, то для генерации значений также необходимо соблюдать версионирование, чтобы избежать ситуации добавления значений в таблицу или в столбец, которых в данной версии еще нет. Для этого используются все то же значение переменной окружения, задающее значение версии миграции. Стоит упомянуть, что исполняется не один файл с последней версией, а все файлы до, так как именно так сохраняются значения в идемпотентых файлах миграции. Таким образом, все генерируемые значения являются допустимыми и не нарушают целостности данных.

Для работы с БД была разработана система ролей и пользователей. Пользователь - это конкретный человек или группа людей, которые имеют доступ к БД, а также набор привилегий и разрешений для работы с ней. В случае PostgreSQL понятия пользователя и роли сливаются в одно, так как USER - это alias над понятием ROLE с одним лишь только отличием, что по умолчанию пользователь может заходить в БД, в то время как ROLE не имеет привилегии на доступ к БД. В нашей СУБД были добавлены роли writer и reader, пользователь analytic, который может анализировать только одну таблицу. Т.е analytic имеет доступ к таблице Bonds с правами на чтение, а также групповая роль developers. Это роль обьеденяет в себе совокупность прав и разрешений на использования БД, однако нельзя использовать PostgreSQL от имени этой роли. Существует группа пользователь, задающихся в конфигурационном файле .env, которые являются developers, т.е наследуют от группы все ее права и дополняют разрешением на доступ к субд.

Бекапы

Для создаия бекапов используется утилита PostgreSQL "pg_dumpall". "pg_dumpall это утилита для записи (“сброса”) всех баз данных PostgreSQL кластера в один файл скрипта. Файл скрипта содержит SQL-команды, которые могут быть использованы в качестве входных данных для psql для восстановления баз данных. Это достигается путем вызова "pg_dump"для каждой базы данных в кластере. "pg_dumpall"также сохраняет глобальные объекты, общие для всех баз данных, а именно роли базы данных, табличные пространства и права доступа для параметров конфигурации. ("pg_dump"не сохраняет эти объекты.)

Был написан скрипт "backup.sh которые парсит ".env"файл, достает значения N и M и в фоновом режиме делает проверку на время.

Patroni

Patroni — это Python-приложение для создания высокодоступных PostgreSQL кластеров на основе потоковой репликации. Таким образом, было достигнуто горизонтальное масштабирование с поддержкой репликации. В случае падения одного из узлов, остальные продолжают работу, перенаправляя запросы на рабочие nodы. Также с помощью внутреннего устройства ETCD (часть кластера на Patroni) были настроены алгоритмы для выбора главной ноды, на которую перенаправляются запросы на изменения БД,а все остальные узлы подтягивают изменения с нее.

Мониторинг

Последней задачей стало внедрение мониторинга. Мониторинг необходим для выявления аномалий в работе БД. Для этого были выбраны такие инструменты, как Prometheus, Postgres-exporter и Grafana. С помощью postgres-exporter собираются метрики из БД, Prometheus прослушивает нужный порт и ретранслирует по другому порту grafana. Последний же сервис нужен для настройки Gui, в котором можно было бы поставить dashboard.

Метрики будем настраивать с помощью созданного файла queries.yaml (настройка для postgres-exporter). Благодаря ему помимо метрик по умолчанию вытянутся еще и те, что мы сами зададим. Пропишем в нем следующее

```
pg_database:
  query: "SELECT pg_database.datname,
  pg_database_size(pg_database.datname) as size_bytes FROM pg_database"
  metrics:
    - datname:
        usage: "LABEL"
        description: "Name of the database"
    - size_bytes:
        usage: "GAUGE"
        description: "Disk space used by the database"

pg_statements_time:
    query:
    "SELECT sum(total_exec_time) / sum(calls) as execution_time FROM pg_stat_statements "
    metrics:
        - execution_time:
            usage: "GAUGE"
            description: "Average time of script execution"

pg_database_error:
    query: "select sum(xact_rollback) as errors_count from pg_stat_database"
    metrics:
```

```
        - errors_count:
            usage: "GAUGE"
            description: "Counts of errors in database"

pg_database_queries:
    query: "select sum(xact_commit + xact_rollback) as queries_count from pg_stat_database"
    metrics:
        - queries_count:
            usage: "GAUGE"
            description: "TPS"

pg_database_calls:
    query: "select sum(calls) as calls from pg_stat_statements"
    metrics:
        - calls:
            usage: "GAUGE"
            description: "QPS"


```

Выводы

Таким образом, по схеме, разработанной на I этапе, была развернута БД в PostgreSQL, которая поднята в docker на локальной машине. Эта СУБД предоставляет набор ролей и пользователей, которые имеют различные уровни доступа к БД и разные права. Для заполнения отношений тестовыми даннами были написани SQL скрипты. Используется bash скрипт, который отвечает за первоначальную инициализацию БД, т.е запуск файлов миграций, генерации данных, а также добавления пользователей. Было добавлено горизонтальное масштабирование, бекапы и мониторинг.