МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО» (НИУ ИТМО)
Факультет программной инженерии и компьютерной техники

Многоуровневая организация программных систем.

Лабораторная работа

Разработка простого IoT сервиса

Исполнитель:

студент гр. P4116 Д.А.Ермолаев

Руководитель:

Преподаватель И.А.Перл

Санкт-Петербург

2024

Цель: отработка принципов и подходов к разработке современных многоуровневых сервисов при решении практической задачи.

Задача: Разработать простое IoT решение и показать применение основных принципов разработки, которые обсуждались на лекции.

Примерная структура решения, которое необходимо разработать:

Лабораторное задание, изображение №1

Компоненты системы

IoT контроллер - сервис, который принимает входные пакеты с данными от «устройств», подключенных к системе. Принимаемые пакеты валидируются и сохраняются в базу данных MongoDB.

Rule engine - простой обработчик правил. Должен уметь обрабатывать мгновенные правила, т.е. основанные на конкретном пакете, и длящиеся, основанные на нескольких пакетах. Пакеты для обработки компонент получает от IoT контроллера через очередь сообщений.

Мгновенное правило - Значение поля A от устройства 42 больше 5.

Длящееся правило - Значение поля A от устройства 42 больше 5 на протяжении 10 пакетов от этого устройства.

Data simulator - Простой генератор данных для разрабатываемого IoT решения. Позволяет указать количество симулируемых устройств и частоту сообщений, которые генерируются каждым из них. Например, 100 устройств и 1 сообщение в секунду с устройства.

Дополнительные компоненты

MongoDB - база данных, в которой хранятся IoT сообщения и отметки (например, алёрты) о срабатываниях правил, которые заложены в Rule Engine

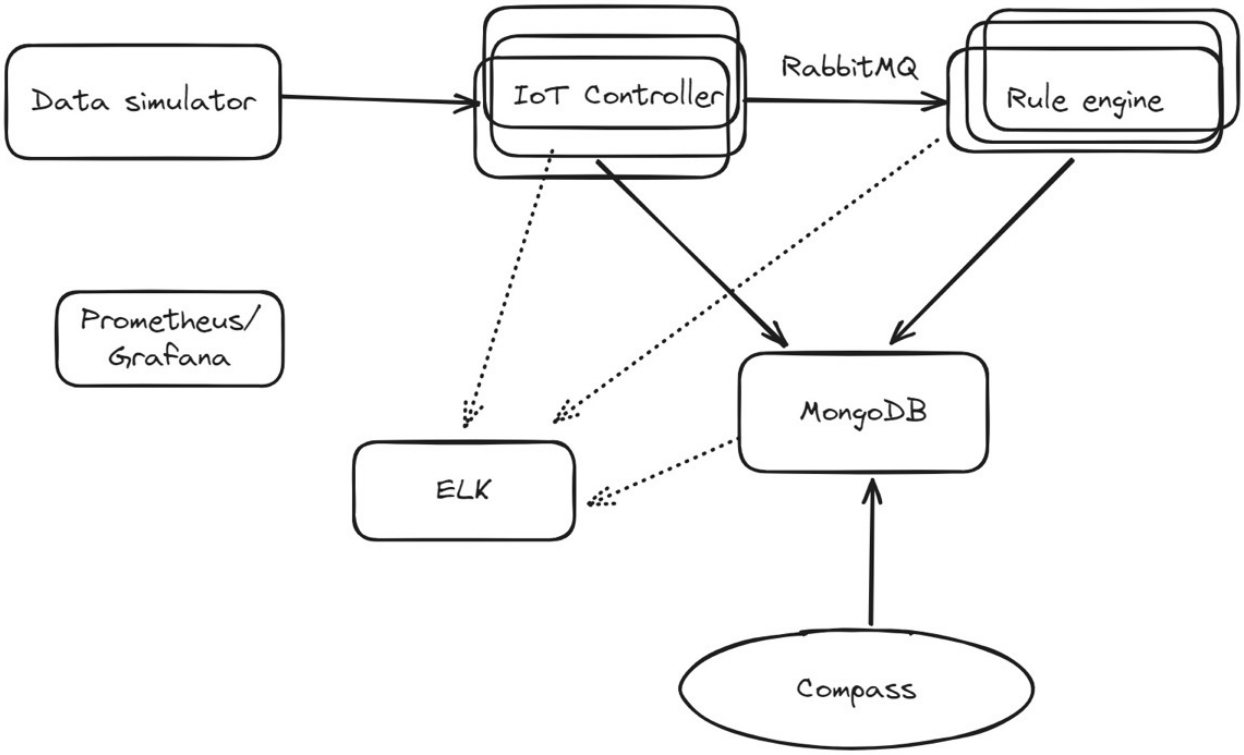
Compass - приложение для просмотра содержимого базы данных MongoDB, будет использоваться вместо пользовательского интерфейса приложения.

RabbitMQ - очередь сообщений для обмена данными между IoT Контроллером и Rule Engine

Postgres/Graphana - система для сбора и мониторинга метрик о работе приложения

ELK Stack - система для сбора и просмотра логов разрабатываемого решения.

При выполнении работы можно использовать любой удобный язык программирования. Желательно выполнять работу на чистом Docker окружении, чтобы максимально разобраться в том, как работают компоненты на низком уровне.



Реализация.

Имитатор устройств.(Data simulator)

Данный проект решил реализовывать на питоне, т.к это новый для меня язык, в нем я не работал от слова совсем, но почему бы и не попробовать сделать что – то новое.

Соответственно на питоне я создал эмулятор устройств **device_client.py** , которое создает 100 устройств и начинает спамить по tsp пакеты с данными.

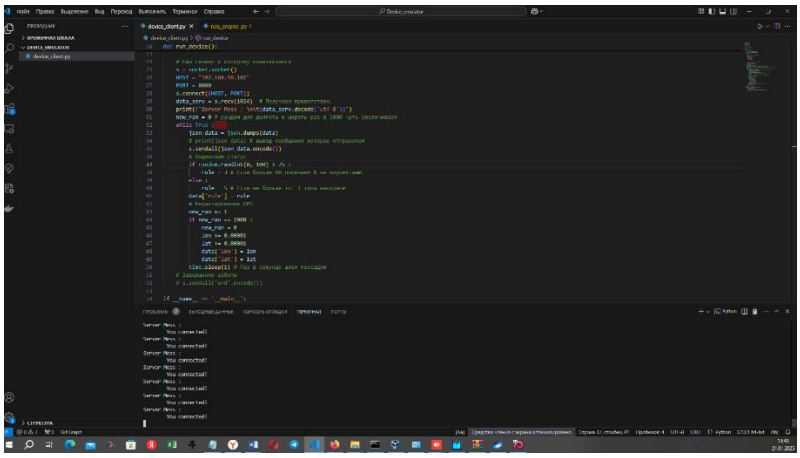
Формат данных JSON принял следующей структурой:

```
{
    "id_device": id,          #  ид девайса
    "lon": lon,               #  долгота (увеличиваем на 0.001раз в 1000 сообщений)
    "lat": lat,               # широта(увеличиваем на 0.001раз в 1000 сообщений)
    "rule": rule              #  правило ( в 75%  = 5 в остальных = 3)
}
```

lon и lat изначально формируется рандомно, а rule = 1.

Пакеты отправляются раз в секунду, на порт 8080(в нашем случае приемник располагался на вертуальной машине с Debian).

Устройства создаются в отельном потоке, ввиду того, что их немного и посылки раз в секунду — это допустимо.



IoT контроллер(IoT)

По заданию мы упаковываем наш сервис(IoT) в контейнер. Тут в дело вступают Docker для упаковки наших сервисов в контейнеры и Docker

Compos для реализации взаимодействия между микросервисами. Соотвественно для контенризации сервисов написанных на питоне я создал фаил Dockerfile к котором описан процесс так сказать контейнеризации, а для всего IoT я создал docker-compose.yml в котор ом описано сценарий взаимодействия контейнеров между собой.

В docker-compose.yml описываем запуск и открываем порты, для работы с деайсами.

```yaml
version: "3.8"

services:
  iotcontroller:
    build: server_iot/
    command: python ./server_iot.py
    image: iotcontroller:latest
    ports:
      - 8080:8080
    depends_on:
      - rabbitmq
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
    networks:
     iot:
       aliases:
         - iotcontroller

  rule:
    build: ./rule_engine/
    #command: python ./rule_engine.py
    image: rule:latest
    ports:
      - 8081:8081
    depends_on:
      - mongodb
      - rabbitmq
    deploy:
      replicas: 3
```

Прием осуществляем на сервер данных tsp реализованный так же на питоне на порт 8080.

```
import socket
import json
import pika
import math
import time
import threading

delitel = 10

# Пул потоков для нашего сервера
class ClientThread(threading.Thread):

    def __init__(self, conn, details):
        self.conn = conn
        self.details = details
        threading.Thread.__init__ ( self )

        print (f"[+] New server socket thread started for \t{HOST} : \t{PORT}")
        self.conn.sendall("You connected!".encode('utf-8'))

    def run(self):
        queue_br  = ""
        #RabbitMQ
        connect_rabbitmq = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq_server'))# Устанавливаем
соединение с сервером RabbitMQ
        channel = connect_rabbitmq.channel()
        while True :
            json_data = self.conn.recv(1024)
            data = json_data.decode('utf-8')
            if data == 'end':
                print('end') # окончание работы соединения
```

И Dockerfile для контейнеризации

```
FROM python:latest

RUN python -m pip install pika
RUN pip show pika

ADD server_iot.py /server_iot/

WORKDIR /server_iot/
```

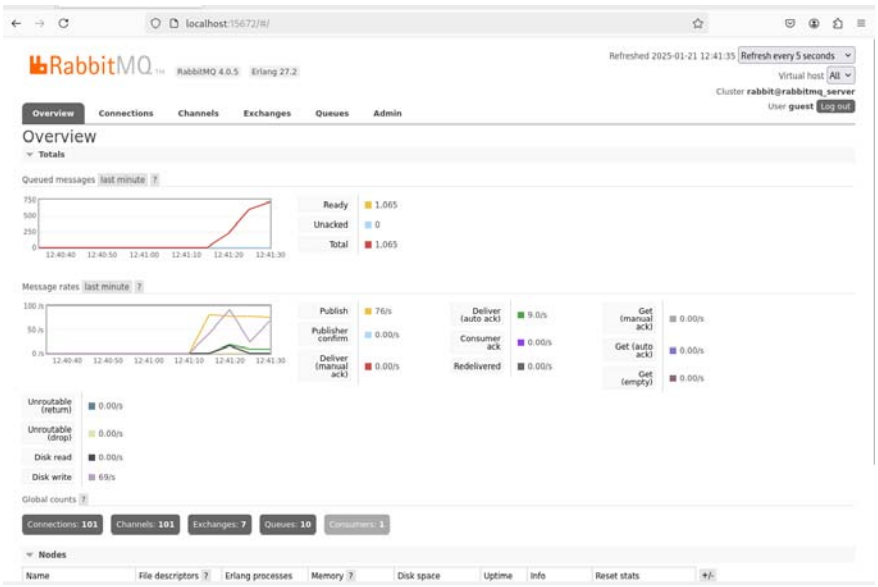
Далее делим по id сообщения от девайсов на очереди (в среднем 100 это 10 очередей) и отправляем сообщения в RabbitMQ также располагается в контейнере Docker.

RabbitMQ

Docker скачает и поставит и настроит контейнер, но для этого надо прописать docker-compose.yml , сервис:

```yaml
rabbitmq:
  image: rabbitmq:management
  hostname: rabbitmq_server
  container_name: rabbitmq_server
  ports:
    - 5672:5672
    - 15672:15672
    - 15692:15692  #prometheus
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  networks:
    iot:
      aliases:
        - rabbitmq
```

Прописав внешний порт 15672, можно зайти через браузер и посмотреть его работу, когда контейнер запущен.



Далее сообщения с броккера обрабатываются **rule_engine.**py по умолчанию приняли что мгновенное в бд заносим ид девайс == 51., мгновенное правило если rule == 3 (примерно 25% в соответствии с генерацией дата эмулятором), а длящееся правило , что как только к нам приходят 10 пакетов подряд с rule == 5 мы заносим эти пакеты в нашу бд.

```
import json
import pymongo
import math
import pika
import time

id_device = 51
#мгновенное правило
light_rule = 3 #мгновенное правило берем пакеты равные 3( 25% примерно)
#Обработка длящегося правила
long_rule = 5 #берем пакеты равные 5( 75% примерно)
count_long_rule = 10 # как только подряд приходит 10 пакетов

time.sleep(40)

#RabbitMQ
delitel = 10
queue_br = 'rule_control_'+str(math.ceil(id_device/delitel))
connect_rabbitmq = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq_server'))# Устанавливаем
соединение с сервером RabbitMQ
channel = connect_rabbitmq.channel()
channel.queue_declare(queue = queue_br)

#mongo db
connect_db = pymongo.MongoClient('mongo_server')
db = connect_db['mopsdb']
collection = db['my_light_engine']
collection_2 = db['my_long_engine']
arr =[]
def callback(ch, method, properties, data):
```

Соответственно к этому сервису так же прилагается Docker.

```
FROM python:latest

RUN pip install --user --upgrade pip
RUN python -m pip install pymongo
RUN pip show pymongo
RUN python -m pip install pika
RUN pip show pika

ADD rule_engine.py /rule_engine/

WORKDIR /rule_engine/

CMD ["python3", "rule_engine.py"]
```

Для запуска этого сервиса в купе, с остальными мы прописываем в docker-compose.yml его настройки:

```yaml
rule:
  build: ./rule_engine/
  #command: python ./rule_engine.py
  image: rule:latest
  ports:
    - 8081:8081
  depends_on:
    - mongodb
    - rabbitmq
  deploy:
    replicas: 3
    restart_policy:
      condition: on-failure
  networks:
    iot:
      aliases:
        - rule
```

Prometheus и Grafana

Также для мониторинга мы используем Prometheus и Grafana.

Prometheus — это система мониторинга, которая собирает и хранит метрики временных рядов от различных ресурсов: программ, программных систем и оборудования.

Grafana — это инструмент для визуализации данных, который позволяет создавать наглядные дашборды. Поддерживает множество источников данных, включая Prometheus

Для запуска этоих сервиса в купе, с остальными мы прописываем в docker-compose.yml его настройки:

```yaml
prometheus:
  image: prom/prometheus
  ports:
    - 9091:9090
  volumes:
    - ./config:/etc/prometheus
  command:
    - "--config.file=/etc/prometheus/prometheus.yml"
  networks:
    iot:
      aliases:
        - prometheus
  deploy:
    replicas: 1

grafana:
  image: grafana/grafana
  restart: unless-stopped
  ports:
    - 3001:3000
  volumes:
    - ./grafana:/var/lib/grafana
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin_1
  networks:
    iot:
      aliases:
        - grafana
  deploy:
    replicas: 1
```

MongoDB

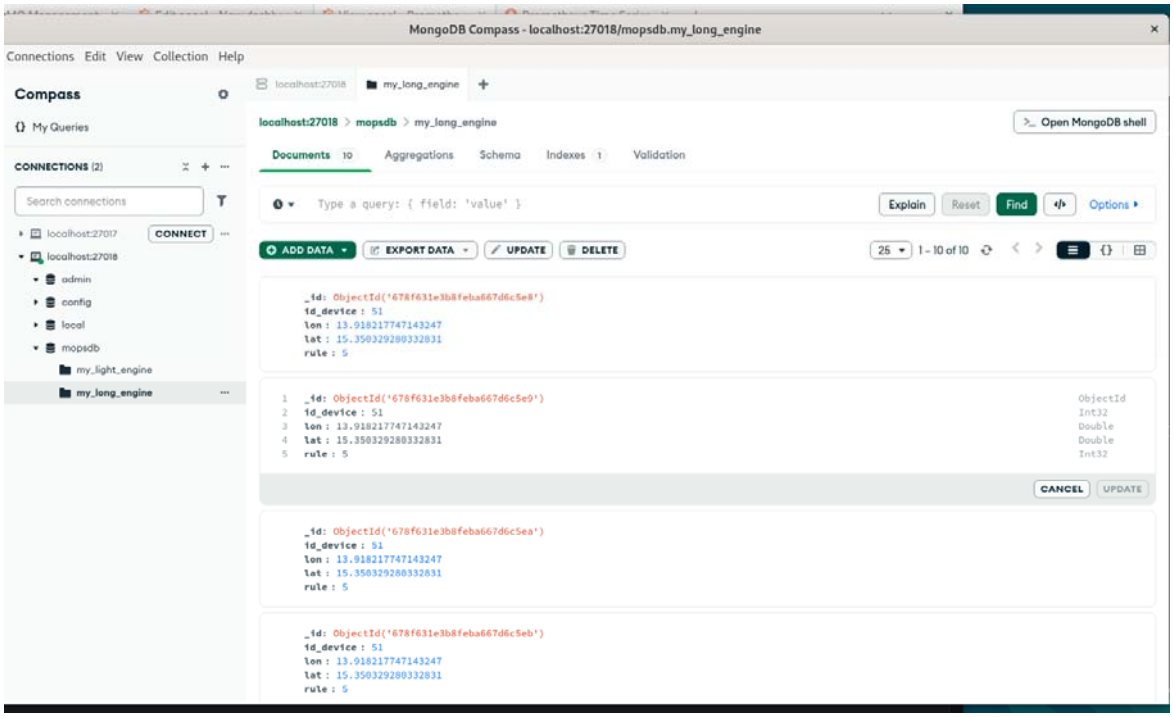
В docker-compose.yml для работы контейнера Mongodb прописан сервис:

```yaml
mongodb:
  image: mongo
  container_name: mongo_server
  ports:
    - 27018:27017
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  networks:
    iot:
      aliases:
        - mongodb
```

Соответственно в rule_engine.py вместо «localhost» в нашем случае мы прописываем присвоенное container_name: mongo_server.

При запуске rule_engine.py в Mongodb мы создаем бд mopsdb, в данной бб мы создаем две коллекции, для так сказать для мгновенного ('my_light_engine') и длящегося правила 'my_long_engine'.

Для просмотра содержимого Mongodb я установил Compass.



Для запуска стека контейнеров необходимо выполнить команду docker-compose up –d, мониторинг можно осуществлять или docker desktop или командами в консоли

Сложность возникла в стыковки компонентов, а именно правильно прокинуть сеть, и настроить компоненты.

Основные дизайн принципы.

Detect for Serf-Healing (Проектирование для самовосстановление) в нашем случае интегрированы инструменты для мониторинга Prometheus и Grafana, который в данном случае помогают отслеживать стабильность работы нашей мини системы.

Так же для обеспечения работы данного принципа используется RabbitMQ, который позволяет реализовать асинхронную передачу данных между компонентами. Так же приложение спроектировано так, основные компоненты запущены по 3 компонента с помощью DockerSwrm, что гарантирует переключение на резервные компоненты. Таким образом, в случае возникновения ошибки работы определенного контейнера наш балансировщик нагрузки автоматически будет передавать запросы на рабочие контейнеры, что не скажется работоспособности всего приложения.

Если бы наша система была более масштабной с точки зрения функционального обеспечения рассматриваемого дизайн принципа можно было бы внести такой функционал как:0

1. Откат транзакций (это позволит гарантировать целосность данных).
2. Ограничение частоты запросов (rate limiting) гарантия от нежелательных воздейтсвий)
3. Реализовать систему таким образом, чтобы в случае отключения какого-либо компонетна, пользователи могли продолжить использовать сновные функции приложения Т.е. сли БД недоступна, то система отключает функционал, связанный с ней и оставляет функционал остальной части системы приложения.

Make all thigs redundadant(Сделайте все избыточным) данный принцип гласит, что наличие нескольких экземпляров каждого важного компонента системы повышает ее надежность и устойчивость. Т.е если один экземпляр выйдет из строя, другой сможет взять на себя его нагрузку, не приводя к поной остановке системы.

В этом пункте идет речь в основном об аппаратной части проектируемой системы.

Для гарантии целостности данных хорошим паттерном будет репликация в базах данных. Это должно выполняться для повышения доступности и устойчивости. Можно настроить репликацию между несколькими БД, чтобы в случае сбоя все данные не были утеряны.

У нашего случая, данный элемент выполнен с помощью использования volumes (механизма для хранения данных, который позволяет сохранять их вне контейнера, обеспечивая персистентность данных) Т.е. в случае отключения и пергрузки контейнера БД мы не потеряем уже хранившиеся данные

Насчет регионально зонирования стоит задуматься в случае распределения инфраструктуры по нескольким регионам или уцентрам обработки данных Т.е обеспечение безопасности от катастрофических сбоев, которые могу затронуть одну территориальную зону (например, природные катастрофы).

Minimiz coordination (минимизируем координацию)

Данный принцип нацелен на снижение необходимости в координации между компонентами системы. Т.е рассматривая данный подход мы должны избежать едиой точки отказа и проблем узкое горлышко.

В нашей системе есть место, в котором реализован данный принцип. С помощью RabbitMQ мы проходим к ассинхронно коммуникации сервисов Т.е, один сервис Iotcontroller отправляет очередь сообщений, а другой сервис (RuleEngine) обрабатывает это сообщение тогда, когда до них дойдет очередь. Но RabbitMQ гарантию того, что сообщения будут доставлены и обработаны.

Disain for scale-out (Отказ от масштабирования) принцип проектирования для масштабирования вширь означает создание архитектуры, которая способна масштабироваться не за счет усиления текущих компонентов (вертикального масштабирования), а за счет добавления новых экземпляров компонентов (горизонтальное масштабирование). Это ключевая стратегия для построения систем, которые могут адаптироваться к растущим нагрузкам и обеспечивать высокую доступность.

Здесь важно помнить о том, чтобы архитектура могла адаптироваться к увеличению нагрузки. Т.е. система должна быть способна справляться с увеличением количества пользователей, запросов и данных. Если не продумать данный пункт, то можно прийти к проблеме узкого горлышка.

Стоит отметить, в нашем приложении реализация была с помощью Docker Swarm который выступил в роли балансировщика нагрузки, распределяя

нагрузку равномерно между экземплярами, и не привязываясь к конкретным сервисам

Partitioning (деление) Данный принцип в проектировании ПО предполагает разделение на более мелкие части для улучшения масштабируемости, производительности и управления данными. Этот подход помогает эффективно справляться с большим объёмами данных и нагрузкой, а также минимизировать узкие места в системе. Например, в случае больших объемов данных узким местом может стать БД, но в нашем случае, ввиду небольшого заносимого объема данных это исключено. Но в случае, когда БД может стать источником узких мест таких как ограничение по производительности ввода/вывода, сетевая нагрузка и ограничение на количество сессий, разбиение на базы данных позволило бы разделить данные на части, который можно обрабатывать независимо, что снижает нагрузку на один сервер или параллельно обрабатывать запросы БД.

Хорошим примером внедрения горизонтального разбиения является разбиение пользователь по региональному принципу или по типам устройств, сли мы, например, взаимодействем с разными устройствами.

Design for operations (Дизайн операций), данный принципе предполагает созданиесистемы таким образом, чтобы эксплуатация, поддержка и масштабирование были простыми и эффективными.

Здесь стоит сконцентрироваться на развертывании, наблюдаемости, эскалации, реагирование на инциденты и аудит безопасности.

В нашей системе для обеспечения данного дизайн принципа проектирования были интегрированы Prometheus и Grafana, что помогаю отслеживать состояние нашей системы в реальном времени.

Т.е данный принцип фокусируется на создании системы, которую легко поддерживать в реальном времени

User manager services данный принцип ориентирован на то, что бы пользователь, то есть организация или команда разработчиков, использовала облачные платформы и управляемые сервисы вместо того, чтобы управлять инфраструктурой в ручную. Это упрощает разработку, эксплуатации добавления бизнес функционала, а не на управлении низкоуровневыми техническими аспектами.

Данный принцип должен привезти к снижению сложности нагрузки на команду разработки, автоматическому масштабированию, снижению затрат на обслуживание, высокой доступности и отказоустойчивости.

Для использования данного принципа можно использовать такие Paas-сервисы, как: Horoku. Google App Engine. AWS Elastic Beanstalk.

Centralized identily Централизованная идентификация и проектирование архитектуры программного обеспечения акцентируемого внимание на создание единого, централизованного механизма управления идентификацией и авторизацией пользователей. Т.е упрощение процессов аутентификации и управления доступом, уменьшение рисков безопасности и повышение удобства использования.

Если бы была необходимость в реализации системы регистрации, аутентификации и авторизации в системе, то грамотным шагом было бы использование готовых решений, предоставляемые как внутренними так и внешними сервисами.

Принцип Centralized identily предполагает, что управление идентификации и авторизации должно быть централизованным и стандартизированным, а не реализоваться с 0. Использование готовых решений, как внутреннего библиотеки для авторизации), так и внешнего(QAuth-поставщиков), позволяет повысить безопасность и упростить разработку. Современные функции аутентификации, такие как многофакторная, SSO, аудит, позволяют не только повысить уровень безопасности, но и улучшить удобство работы с системой для пользователей.

Design for evolution данный принцип фокусируется на создании архитектуры, которая легко адаптируется к изменениям в будущем. Системы должны быть гибкими и масштабируемыми, чтобы поддерживать новые функции, требования и технологии без необходимости полностью переписывать их. Данный принцип особенно важен в наше время ввиду быстрого развития технологий.

В нашей системе можно выделить тот факт, что каждый из компонентов отвечает за собственную функциональность, что демонстрирует разграничение ответственности между сервисами. Так же в нашей системе используется асинхронная передача сообщений, что улучшит производительность системы.

Design for business needs принцип проектирования с учетом бизнес-требований ориентирован на создание архитектуры ПО, которая напрямую отвечает на потребность бизнеса. Это включает в себя правильное определение бизнес-полей, понимание рисков и возможностей, а также выстраивание архитектуры так, чтобы она могла поддерживать текущие и бедующие требования бизнеса. Процесс проектирования должен учитывать, как функциональные так, и не функциональные требования, а также обеспечить соответствие финансовым и операционным показателям.

Если придерживаться данного принципа, то необходимо:

1. Понять, какие цели бизнес ставит перед системой. Это может правильно направить усилия на те области, которые имеют наибольшее значение для бизнеса.
2. Учесть какие риски отказа могут повлиять на бизнес процессы и выделить мероприятия по нивелированию, которые минимизируют выделенные риски.
3. Спроектировать систему не только с точки зрения технических компонентов, но и с точки зрения бизнес логики. Подразумевается построение диаграмм, описывающих, как различные компоненты системы взаимодействуют между собой и пользователем.
4. Выделять функциональные и не функциональные требования к системе.
5. Необходимо проектировать архитектуру так, чтобы она поддерживала бизнес модель в соответствии с финансовыми ограничениями. Важно учитывать стоимость инфраструктуры, разработки, тестирования и поддержки системы.
6. Прогнозировать и управлять затратами (Например, например, использование облачных сервисов с возможность динамического масштабирования позволяет снижать избыточные расходы на инфраструктуру в периоды низкой нагрузки).

Resiliency checklist данный принцип касается проектирования и эксплуатации системы таким образом, чтобы она могла эффективно справляться со сбоями, восстанавливаться после них и обеспечивать высокую доступность и надежность. Это важный аспект архитектуры, поскольку система должна быть способна продолжать свою работу даже в случае частичных отказов.

В нашем случае производиться мониторинг производительности системы, который помогает своевременно обнаружить перегрузки, утечки памяти, высокие задержки и другие проблемы (Prometheus и Grafana).

Failure modes analysis FMA Это процесс идентификации возможных сбоев в системе, их последствий и разработки стратегии реагирования. Цель FMA заключается в том, чтобы заблаговременно выявить потенциальные проблемы и подготовиться к ним, что значительно повышает устойчивость системы и минирует риски.

В нашем случае есть выбор сервисов, каждый из которых имеет те или иные зависимости. В Docker Compos файле указано, какие компоненты зависят от других компонентов.

Так же в более масштабных приложениях было бы полезно для каждого компонента возможные риски и отказы. Т.е. выделяем факты, которые

могут привести сбою, например, отказ сети, ошибка, перегрузка сервера и т.д.

Далее необходимо было бы оценить, какое воздействие на приложение будет нести сбой. Важно понимать, какие последствия будут иметь сбои в работе системы, и какие них критичны для пользователей и бизнеса. Например, БД будет недоступна более 24 часов, это может вызвать значительные финансовые потери, потерю клиентов и ухудшение репутации компании.

И в результате нам надо будет сформулировать, как будет реагировать приложение на тот или иной сбой. То есть нам надо было разработать стратегию, как приложение будет вести себя в каждого из выделенных сбоев. Это позволяет минимизировать влияние отказов и быстро восстанавливать систему. Например, в случае сбоя БД система может автоматически переключаться на резервную копию данных или активировать систему копирования, чтобы временно обслуживать запросы до восстановления основной БД.

Выводы.

Целью данного проекта, было ознакомиться и реализовать основные принципы разработки, а также произвести отработку основных навыков и подходов к разработке современных многоуровневых сервисов при решении данной задачи.

Проект выложен в репозитории GitHub:

https://github.com/den251965/MOPS