Cодержание

[1. Задание 2](#_Toc128682828)

[2. Анализ проекта из первой лабораторной 3](#_Toc128682829)

[3. Реализация проекта 4](#_Toc128682830)

[3.1. Служба аутентификации и авторизации 4](#_Toc128682831)

[3.1.1. Описание службы 4](#_Toc128682832)

[3.1.2. Описание кода сервиса 4](#_Toc128682833)

[3.1.3. Описание докерфайла 4](#_Toc128682834)

[3.2. Служба управления пользователями 4](#_Toc128682835)

[3.2.1. Описание службы 4](#_Toc128682836)

[3.2.2. Описание кода 5](#_Toc128682837)

[3.2.3. Описание докерфайла 5](#_Toc128682838)

[3.3. Служба управления продуктами 5](#_Toc128682839)

[3.3.1. Описание службы 5](#_Toc128682840)

[3.3.2. Описание кода 5](#_Toc128682841)

[3.3.3. Описание докерфайла 5](#_Toc128682842)

[3.4. Служба управления корзиной 5](#_Toc128682843)

[3.4.1. Описание службы 5](#_Toc128682844)

[3.4.2. Описание кода 6](#_Toc128682845)

[3.4.3. Описание докерфайла 6](#_Toc128682846)

[3.5. Служба управления заказами 6](#_Toc128682847)

[3.5.1. Описание службы 6](#_Toc128682848)

[3.5.2. Описание кода 6](#_Toc128682849)

[3.5.3. Описание докерфайла 6](#_Toc128682850)

[3.6. Регистрация в Eureka 6](#_Toc128682851)

[3.6.1. Описание 6](#_Toc128682852)

[3.6.2. Описание кода 7](#_Toc128682853)

[3.6.3. Описание докерфайла 7](#_Toc128682854)

[3.7. Доступ через Gateway 7](#_Toc128682855)

[3.7.1. Описание 7](#_Toc128682856)

[3.7.2. Описание кода 7](#_Toc128682857)

[3.7.3. Описание докерфайла 7](#_Toc128682858)

[3.8. Конфигурация Config-Server 7](#_Toc128682859)

[3.8.1. Описание 7](#_Toc128682860)

[3.8.2. Описание кода 7](#_Toc128682861)

[3.8.3. Описание докерфайла 7](#_Toc128682862)

[3.9. Описание docker-compose.yml 7](#_Toc128682863)

1. Задание

Декомпозировать на микросервисы систему из лабораторной работы 1.

* Микросервисы должны регистрироваться в Eureka <https://spring.io/projects/spring-cloud-netflix>
* Микросервисы должны подтягивать конфигурацию из Config-Server https://spring.io/projects/spring-cloud-config
* Микросервисы должны быть доступны через Spring Gateway <https://spring.io/projects/spring-cloud-gateway>
* Для взаимодействия между микросервисами использовать Feign Client
* Внедрить Circuit Breaker https://cloud.spring.io/spring-cloud-netflix/multi/multi\_\_circuit\_breaker\_hystrix\_clients.html
* Решить проблему с авторизацией между микросервисами
* Минимум один микросервис должен быть написан с использованием Reactor и r2dbc
* Минимум один микросервис должен быть написан с использованием Reactor и spring-data-jpa
* Минимум один микросервис должен использовать только jdbc

Предоставленный проект из первой лабораторной работы является монолитным приложением, что означает, что весь функционал упакован в одно приложение. Чтобы разложить это приложение на микросервисы, нам нужно определить различные части приложения и их зависимости. Затем на основе идентифицированных частей мы можем разбить приложение на более мелкие независимые службы.

1. Анализ проекта из первой лабораторной

Из проекта на GitHub мы можем выделить следующие части:

* Аутентификация и авторизация
* Управление пользователями (CRUD-операции для пользователей)
* Управление продуктом (CRUD-операции для продуктов)
* Управление корзиной (добавление/удаление товаров из корзины)
* Управление заказами (размещение и отслеживание заказов)

Чтобы разложить это монолитное приложение на микросервисы, мы могли бы рассмотреть следующий подход:

1. Служба аутентификации и авторизации: эта служба будет обрабатывать аутентификацию и авторизацию пользователей. Он будет отвечать за проверку личности и учетных данных пользователя, создание и проверку токенов доступа и применение политик безопасности.
2. Служба управления пользователями: эта служба будет обрабатывать все операции CRUD, связанные с пользователями. Он будет отвечать за создание, чтение, обновление и удаление пользовательской информации. Он также предоставит API для регистрации пользователей, сброса пароля и других задач, связанных с пользователем.
3. Служба управления продуктами: эта служба будет обрабатывать все операции CRUD, связанные с продуктами. Он будет отвечать за создание, чтение, обновление и удаление информации о продукте. Он также предоставит API для поиска и фильтрации продуктов на основе различных критериев.
4. Служба управления корзиной: эта служба будет обрабатывать добавление/удаление товаров из корзины. Он будет нести ответственность за поддержание состояния корзины покупок пользователя и ее обновление по мере того, как пользователь добавляет или удаляет продукты.
5. Служба управления заказами: эта служба занимается размещением и отслеживанием заказов. Он будет отвечать за обработку заказов, генерацию подтверждений заказов и предоставление API для отслеживания статуса заказов.
6. Реализация проекта
   1. Служба аутентификации и авторизации
      1. Описание службы

Эта служба будет обрабатывать аутентификацию и авторизацию пользователей. Он будет отвечать за проверку личности и учетных данных пользователя, создание и проверку токенов доступа и применение политик безопасности.

* + 1. Описание кода сервиса

Весь код проекта для этого сервиса находится по пути labatwo/authentication-and-authorization-service.

Каталог src содержит несколько пакетов, которые упорядочивают компоненты приложения. К ним относятся:

* config: этот пакет содержит классы Java, которые определяют конфигурацию приложения, такую как конфигурация безопасности и конфигурация Spring RestTemplate.
* controller: этот пакет содержит класс AuthController, который определяет конечные точки REST для проверки подлинности и обновления токена.
* model: этот пакет содержит несколько классов Java, которые определяют модели данных, используемые приложением, такие как AuthenticationRequest и AuthenticationResponse.
* security: этот пакет содержит несколько классов, определяющих конфигурацию безопасности для приложения, например JwtTokenProvider, который отвечает за создание и проверку токенов JWT.
* service: этот пакет содержит несколько классов Java, определяющих бизнес-логику приложения, например UserService, который отвечает за получение сведений о пользователе из пользовательской службы.

В целом, микросервис предназначен для предоставления централизованной конечной точки аутентификации, которая может аутентифицировать пользователей в пользовательской службе и генерировать токены JWT. Приложение также включает функции для обновления токенов и проверки токенов для безопасного доступа к защищенным ресурсам.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Служба управления пользователями
     1. Описание службы

Эта служба будет обрабатывать все операции CRUD, связанные с пользователями. Он будет отвечать за создание, чтение, обновление и удаление пользовательской информации. Он также предоставит API для регистрации пользователей, сброса пароля и других задач, связанных с пользователем.

* + 1. Описание кода

Весь код проекта для этого сервиса находится по пути labatwo/user-management-service.

Репозиторий содержит исходный код пользовательского сервиса высоконагруженного приложения. Вот обзор кода:

* Каталог src/main/java содержит исходный код Java для службы.
* Пакет com.example.userservice содержит основной класс UserServiceApplication с основным методом и аннотацией @SpringBootApplication.
* Пакет конфигурации содержит классы конфигурации для службы, например класс CorsConfig, который настраивает совместное использование ресурсов между источниками (CORS) для службы.
* Пакет контроллера содержит конечные точки REST для пользовательской службы. Класс UserController имеет методы для обработки HTTP-запросов, связанных с созданием, поиском и удалением пользователей.
* Пакет исключений содержит настраиваемые классы исключений для службы, такие как UserNotFoundException, которое выдается, когда запрошенный пользователь не найден в базе данных.
* Пакет модели содержит класс модели User, который представляет пользователя и имеет такие поля, как идентификатор, адрес электронной почты, имя и фамилия.
* Пакет репозитория содержит интерфейс UserRepository, который расширяет интерфейс Spring JpaRepository и определяет методы взаимодействия с базой данных для выполнения операций CRUD над объектами User.
* Пакет службы содержит интерфейс UserService и его класс реализации UserServiceImpl, которые определяют бизнес-логику службы. Методы службы взаимодействуют с UserRepository для выполнения операций CRUD над объектами User.
* Файл resources/application.properties содержит свойства конфигурации службы, например свойства подключения к базе данных.

В целом код соответствует стандартной структуре и соглашениям службы Spring Boot. Он использует Spring Data JPA для взаимодействия с базой данных и предоставляет конечные точки REST для управления пользователями.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Служба управления продуктами
     1. Описание службы

Эта служба будет обрабатывать все операции CRUD, связанные с продуктами. Он будет отвечать за создание, чтение, обновление и удаление информации о продукте. Он также предоставит API для поиска и фильтрации продуктов на основе различных критериев.

* + 1. Описание кода

Весь код проекта для этого сервиса находится по пути labatwo/product-management-service.

Класс ProductController определяет конечные точки API для управления продуктами, включая операции GET, POST, PUT и DELETE. Аннотация @RequestMapping указывает базовый путь маршрута для всех вызовов API, а аннотация @Autowired вводит объект ProductService для обработки бизнес-логики управления продуктами.

Аннотация @GetMapping определяет конечную точку для получения всех продуктов или одного продукта по идентификатору. Аннотация @PostMapping определяет конечную точку для добавления нового продукта. Аннотация @PutMapping определяет конечную точку для обновления существующего продукта. Аннотация @DeleteMapping определяет конечную точку для удаления продукта.

Класс UriComponentsBuilder используется в методе addProduct для создания URI для вновь добавленного продукта, который возвращается в заголовках ответа. Класс ResponseEntity используется для возврата ответов HTTP с соответствующими кодами состояния и заголовками.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Служба управления корзиной
     1. Описание службы

Эта служба будет обрабатывать добавление/удаление товаров из корзины. Он будет нести ответственность за поддержание состояния корзины покупок пользователя и ее обновление по мере того, как пользователь добавляет или удаляет продукты.

* + 1. Описание кода

Весь код проекта для этого сервиса находится по пути labatwo/basket-management-service.

Этот класс BasketController с четырьмя конечными точками API:

* GET /api/v1/basket/{id} для получения корзины по ID
* POST /api/v1/basket/ для создания новой корзины
* PUT /api/v1/basket/{id} для обновления существующей корзины по идентификатору
* DELETE /api/v1/basket/{id}, чтобы удалить корзину по идентификатору

Эти конечные точки используют класс BasketService для взаимодействия с базой данных и выполнения необходимых операций.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Служба управления заказами
     1. Описание службы

Эта служба занимается размещением и отслеживанием заказов. Он будет отвечать за обработку заказов, генерацию подтверждений заказов и предоставление API для отслеживания статуса заказов.

* + 1. Описание кода

Весь код проекта для этого сервиса находится по пути labatwo/order-management-service.

В этом классе контроллера OrderController с четырьмя различными методами API, которые обрабатывают HTTP-запросы GET, POST, PUT и DELETE. Аннотация @RequestMapping используется для указания базового пути URL для всех методов в этом контроллере. Аннотация @Autowired используется для внедрения экземпляра класса OrderService, который используется для извлечения данных заказа и управления ими.

Метод getOrderById извлекает заказ по его идентификатору, используя аннотацию @GetMapping и указывая идентификатор заказа в качестве переменной пути. Если заказ не найден, возвращается ответ 404 NOT FOUND, в противном случае заказ возвращается с ответом 200 OK.

Метод createOrder создает новый заказ, принимая тело запроса JSON, содержащее сведения о заказе. Аннотация @PostMapping используется для указания того, что этот метод обрабатывает HTTP-запросы POST. Вновь созданный заказ возвращается в теле ответа с кодом состояния 201 CREATED.

Метод updateOrder обновляет существующий заказ, принимая тело запроса JSON, содержащее обновленные сведения о заказе, и указывая идентификатор заказа в качестве переменной пути. Если заказ не найден, возвращается ответ 404 NOT FOUND, в противном случае обновленный заказ возвращается с ответом 200 OK.

Метод deleteOrder удаляет существующий заказ, указывая идентификатор заказа в качестве переменной пути. Если заказ успешно удален, возвращается ответ 204 NO CONTENT, в противном случае возвращается ответ 404 NOT FOUND.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Регистрация в Eureka
     1. Описание

Eureka — это компонент реестра служб проекта Spring Cloud Netflix, который позволяет микросервисам обнаруживать и взаимодействовать друг с другом в распределенной системе.

В распределенной системе каждый микросервис может работать на разных машинах и портах, а их расположение может часто меняться из-за увеличения или уменьшения масштаба. Eureka предоставляет централизованный реестр, который каждая микрослужба может использовать для регистрации своего местоположения и состояния.

Затем другие микросервисы могут использовать Eureka для обнаружения местоположения и состояния микросервисов, с которыми им необходимо взаимодействовать, без жесткого кодирования IP-адресов или других сведений о соединении. Eureka также включает в себя такие функции, как балансировка нагрузки и отказоустойчивость, упрощающие создание устойчивых архитектур микросервисов.

В целом, Eureka упрощает процесс создания микросервисов и управления ими, позволяя разработчикам сосредоточиться на построении своей основной бизнес-логики, а не на механике обнаружения сервисов и коммуникации.

* + 1. Описание кода

EurekaServerApplication.java

package com.example.eurekaserver;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServerApplication.class, args);

    }

}

Приведенный выше код представляет собой класс Java для приложения Spring Boot, которое настраивает сервер Eureka. Вот что делает каждый раздел кода:

package: Этот оператор указывает пакет, к которому принадлежит класс.

import: Эти операторы импортируют различные классы, которые используются в классе.

@SpringBootApplication: эта аннотация используется для включения функций Spring Boot для приложения. Это комбинация трех аннотаций: @Configuration, @EnableAutoConfiguration и @ComponentScan. @Configuration используется для указания классов конфигурации, @EnableAutoConfiguration включает автоматическую настройку Spring Boot, а @ComponentScan сообщает Spring, где искать такие компоненты, как контроллеры и службы.

@EnableEurekaServer: эта аннотация используется для включения функций сервера Eureka в приложении. Он настраивает приложение как реестр служб, который другие микрослужбы могут использовать для обнаружения и взаимодействия друг с другом.

открытый класс EurekaServerApplication: это основной класс приложения.

public static void main(String[] args): это основной метод приложения. Это точка входа приложения, и она запускает приложение Spring Boot, вызывая SpringApplication.run().

SpringApplication.run(EurekaServerApplication.class, args): этот метод запускает приложение Spring Boot и инициализирует контекст Spring.

В целом, этот класс Java настраивает приложение Spring Boot с включенной функциональностью сервера Eureka. Это простая, но необходимая настройка для создания микросервисов, которым необходимо взаимодействовать друг с другом в распределенной системе. Когда сервер Eureka запущен, другие микрослужбы могут зарегистрироваться на сервере Eureka и использовать его для обнаружения местоположений и состояний других микрослужб в системе.

application.yml

spring:

  application:

    name: eureka-server

server:

  port: ${SERVER\_PORT:8761}

eureka:

  client:

    register-with-eureka: false

    fetch-registry: false

Приведенный выше код представляет собой файл конфигурации YAML для приложения Spring Boot, которое настраивает сервер Eureka. Вот что делает каждый раздел конфигурации:

* spring: это раздел конфигурации верхнего уровня для приложений Spring Boot. Он используется для настройки различных аспектов среды Spring.
* application: в этом разделе задается имя серверного приложения Eureka «eureka-server». Это имя используется для регистрации сервера Eureka в других службах системы.
* server: в этом разделе задается порт, который будет прослушивать сервер Eureka. В этом случае он использует порт по умолчанию 8761, но также позволяет переменной среды SERVER\_PORT переопределять значение по умолчанию.
* eureka: в этом разделе настраивается клиент Eureka для сервера Eureka.
* client.register-with-eureka: для этого свойства установлено значение false, что означает, что сам сервер Eureka не будет зарегистрирован ни в каком другом экземпляре сервера Eureka. Это полезно, когда сервер Eureka используется как автономный реестр служб.
* client.fetch-registry: для этого свойства также установлено значение false, что означает, что сервер Eureka не будет пытаться получить информацию реестра из других экземпляров сервера Eureka. Опять же, это полезно, когда сервер Eureka используется в качестве автономного реестра служб.

В целом, этот файл конфигурации настраивает автономный сервер Eureka, который будет прослушивать порт по умолчанию 8761 и не будет регистрироваться с другими экземплярами сервера Eureka или извлекать информацию из реестра из других экземпляров. Это простая конфигурация, которую можно использовать в качестве отправной точки для создания более сложных настроек реестра служб.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Доступ через Gateway
     1. Описание

Spring Cloud Gateway — это легкий шлюз API с широкими возможностями настройки, созданный на основе Spring Framework 5, Spring Boot 2 и Project Reactor. В качестве шлюза API он действует как точка входа для входящих HTTP-запросов от клиентов и направляет их в соответствующие микросервисы, которые могут обрабатывать запросы.

Некоторые из ключевых особенностей Spring Cloud Gateway:

* Маршрутизация: Spring Cloud Gateway предоставляет гибкий механизм маршрутизации, который можно использовать для маршрутизации входящих запросов к соответствующим нижестоящим службам на основе различных критериев, таких как путь запроса, заголовки, параметры запроса и т. д.
* Фильтры: Spring Cloud Gateway поддерживает широкий спектр фильтров, которые можно использовать для изменения входящих запросов и исходящих ответов. Фильтры можно использовать для реализации аутентификации, ограничения скорости, формирования трафика и других общих функций шлюза API.
* Балансировка нагрузки: Spring Cloud Gateway предоставляет встроенные возможности балансировки нагрузки, которые можно использовать для распределения входящих запросов между несколькими экземплярами службы.
* Прерыватель цепи: Spring Cloud Gateway интегрируется с Netflix Hystrix, чтобы обеспечить функциональность прерывателя цепи, которую можно использовать для предотвращения каскадных сбоев в распределенной системе.
* Интеграция с Spring Cloud: Spring Cloud Gateway интегрируется с другими компонентами Spring Cloud, такими как Eureka и Consul, для обнаружения сервисов и управления конфигурацией.
* Модель реактивного программирования: Spring Cloud Gateway построен с использованием модели реактивного программирования, которая позволяет ему обрабатывать большое количество одновременных подключений с низким использованием ресурсов.

В целом, Spring Cloud Gateway предоставляет мощную и гибкую платформу для создания шлюзов API для архитектур микросервисов. Его модульная архитектура и широкие возможности настройки делают его отличным выбором для создания масштабируемых и отказоустойчивых распределенных систем.

* + 1. Описание кода

application.yml

server:

  port: ${SERVER\_PORT:8080}

eureka:

  client:

    service-url:

      default-zone: http://localhost:8761/eureka

  instance:

    prefer-ip-address: true

spring:

  application:

    name: gateway-service

  config:

    import: optional:configserver:http://${CONFIG\_HOST:localhost}:${CONFIG\_PORT:8888}

Это файл конфигурации для службы Spring Cloud Gateway, которая действует как обратный прокси-сервер и балансировщик нагрузки для микрослужб в распределенной системе. Разберем ключевые элементы конфигурации:

* server.port: ${SERVER\_PORT:8080} — устанавливает номер порта для службы шлюза. Номер порта считывается из переменной среды с именем «SERVER\_PORT». Если переменная среды не задана, используется номер порта по умолчанию 8080.
* eureka.client.service-url.default-zone: http://localhost:8761/eureka — устанавливает URL-адрес по умолчанию для сервера Eureka, который используется для обнаружения служб. Служба шлюза зарегистрируется на сервере Eureka, используя этот URL-адрес.
* eureka.instance.prefer-ip-address: true — устанавливает предпочтение службы шлюза для использования своего IP-адреса вместо имени хоста при регистрации на сервере Eureka.
* spring.application.name: gateway-service — устанавливает имя приложения Spring Boot на «шлюз-сервис».
* spring.config.import: optional:configserver:http://${CONFIG\_HOST:localhost}:${CONFIG\_PORT:8888} — импортирует конфигурацию с сервера конфигурации. Служба шлюза попытается получить файлы конфигурации с сервера конфигурации, работающего по URL-адресу, указанному в переменных среды «CONFIG\_HOST» и «CONFIG\_PORT». Если переменные не установлены, используются значения по умолчанию «localhost» и 8888. Ключевое слово «необязательный» указывает, что служба шлюза все равно будет работать, если она не может подключиться к серверу конфигурации.

В целом, этот файл конфигурации настраивает службу шлюза Spring Cloud, которая регистрируется на сервере Eureka для обнаружения службы, прослушивает указанный порт и извлекает файлы конфигурации с сервера конфигурации, если они доступны.

GatewayRoutingConfig.java

package com.example.gatewayservice;

import lombok.RequiredArgsConstructor;

import lombok.extern.slf4j.Slf4j;

import org.springframework.cloud.gateway.route.RouteLocator;

import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

@RequiredArgsConstructor

@Slf4j

public class GatewayRoutingConfig {

    @Bean

    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

        return builder.routes()

                .route(route -> route.path("/auth").uri("lb://user-service"))

                .route(route -> route.path("/api/v1/orders/\*\*").uri("lb://facade"))

                .route(route -> route.path("/api/v1/products/\*\*").uri("lb://facade"))

                .route(route -> route.path("/api/v1/basket/\*\*").uri("lb://facade"))

                .route(route -> route.path("/\*\*").uri("lb://reactive-facade"))

                .build();

    }

}

Это метод в приложении Spring Boot, который возвращает RouteLocator, который используется для определения маршрутов, которые будут обрабатываться службой Spring Cloud Gateway. Разберем код:

customRouteLocator (построитель RouteLocatorBuilder) — этот метод принимает объект RouteLocatorBuilder в качестве параметра, который используется для определения маршрутов.

return builder.routes() — начинается определение маршрутов.

* .route(route -> route.path("/auth").uri("lb://user-service")) — определяет маршрут для запросов авторизации, соответствующих пути /auth. Маршрут перенаправляет запрос экземпляру службы пользовательской службы, используя балансировку нагрузки.
* .route(route -> route.path("/api/v1/orders/\*\*").uri("lb://facade")) — определяет маршрут для запросов к заказам, соответствующих /api/v1/orders /\*\* путь. Маршрут перенаправляет запрос экземпляру фасадной службы, используя балансировку нагрузки.
* .route(route -> route.path("/api/v1/products/\*\*").uri("lb://facade")) — определяет маршрут для запросов к продуктам, соответствующих /api/v1/products /\*\* путь. Маршрут перенаправляет запрос экземпляру фасадной службы, используя балансировку нагрузки.
* .route(route -> route.path("/api/v1/basket/\*\*").uri("lb://facade")) — определяет маршрут для запросов к корзине, соответствующих /api/v1/basket /\*\* путь. Маршрут перенаправляет запрос экземпляру фасадной службы, используя балансировку нагрузки.
* .route(route -> route.path("/\*\*").uri("lb://reactive-facade")) — определяет универсальный маршрут для запросов, которые не соответствуют ни одному из предыдущих маршрутов. Маршрут перенаправляет запрос экземпляру службы реактивного фасада с использованием балансировки нагрузки.
* .build() — завершает определение маршрутов и возвращает объект RouteLocator.

В целом этот код определяет службу Spring Cloud Gateway, которая обрабатывает запросы по нескольким различным путям и перенаправляет их в соответствующие экземпляры службы с помощью балансировки нагрузки.

GatewayServiceApplication.java

package com.example.gatewayservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class GatewayServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayServiceApplication.class, args);

    }

}

Приведенный выше код представляет собой класс Java для приложения Spring Boot, которое настраивает службу шлюза. Вот что делает каждый раздел кода:

* package: Этот оператор указывает пакет, к которому принадлежит класс.
* import: Эти операторы импортируют различные классы, которые используются в классе.
* @SpringBootApplication: эта аннотация используется для включения функций Spring Boot для приложения. Это комбинация трех аннотаций: @Configuration, @EnableAutoConfiguration и @ComponentScan. @Configuration используется для указания классов конфигурации, @EnableAutoConfiguration включает автоматическую настройку Spring Boot, а @ComponentScan сообщает Spring, где искать такие компоненты, как контроллеры и службы.
* @EnableEurekaClient: эта аннотация используется для включения функций клиента Eureka в приложении. Он регистрирует службу шлюза на сервере Eureka, что позволяет другим микрослужбам обнаруживать службу шлюза и взаимодействовать с ней.
* открытый класс GatewayServiceApplication: это основной класс приложения.
* public static void main(String[] args): это основной метод приложения. Это точка входа приложения, и она запускает приложение Spring Boot, вызывая SpringApplication.run().
  + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Конфигурация Config-Server
     1. Описание

Spring Cloud Config — это серверный компонент среды Spring Cloud, который обеспечивает централизованное внешнее управление конфигурацией для распределенных систем. Он позволяет централизованно управлять конфигурациями нескольких микрослужб и хранить их, а затем распространять эти конфигурации среди микрослужб по запросу.

Ключевые особенности Spring Cloud Config:

* Централизованное управление конфигурацией: Spring Cloud Config предоставляет центральное место для хранения конфигураций для нескольких микросервисов. Файлы конфигурации можно хранить в репозитории Git или любом другом хранилище, поддерживающем абстракцию Spring Resource.
* Контроль версий: Spring Cloud Config поддерживает контроль версий для файлов конфигурации. Вы можете отслеживать изменения в файлах конфигурации с течением времени, выполнять откат к предыдущим версиям и просматривать изменения с помощью стандартных инструментов Git.
* Безопасность: Spring Cloud Config поддерживает защиту конфигураций с помощью Spring Security. Вы можете настроить детальное управление доступом для разных пользователей и ролей.
* Масштабируемость: Spring Cloud Config разработан с учетом высокой масштабируемости. Вы можете развернуть несколько экземпляров сервера конфигурации и использовать балансировщик нагрузки для распределения запросов между экземплярами.
* Динамическое обновление: Spring Cloud Config поддерживает динамическое обновление конфигураций. Изменения в файлах конфигурации можно распространить на микрослужбы без перезапуска.
* Интеграция с Spring Cloud: Spring Cloud Config интегрируется с другими компонентами среды Spring Cloud, такими как Eureka, для обнаружения сервисов и балансировки нагрузки.

В целом Spring Cloud Config предоставляет мощную и гибкую платформу для управления конфигурациями распределенных систем. Его способность управлять версиями, динамически обновлять конфигурации и интегрироваться с другими компонентами Spring Cloud делает его отличным выбором для создания масштабируемых и устойчивых архитектур микросервисов.

* + 1. Описание кода

ConfigServerApplication.java

package com.example.configserver;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(ConfigServerApplication.class, args);

    }

}

application.yml

spring:

  application:

    name: config-server

  cloud:

    config:

      server:

        git:

          uri: https://github.com/slovak12345/labatwo-config.git

          default-label: master

server:

  port: ${SERVER\_PORT:8888}

eureka:

  client:

    service-url:

      default-zone: http://localhost:8761/eureka

  instance:

    prefer-ip-address: true

Это файл конфигурации для Spring Cloud Config Server, который отвечает за обслуживание файлов конфигурации для микросервисов в распределенной системе. Разберем ключевые элементы конфигурации:

spring.application.name: config-server — устанавливает имя приложения Spring Boot на «config-server».

spring.cloud.config.server.git.uri: https://github.com/slovak12345/labatwo-config.git — устанавливает URI для репозитория Git, содержащего файлы конфигурации. Сервер конфигурации будет использовать этот репозиторий Git для получения файлов конфигурации для микросервисов.

spring.cloud.config.server.git.default-label: master — устанавливает для ветки Git по умолчанию значение «master». Сервер конфигурации будет использовать эту ветвь для получения файлов конфигурации, если в запросе не указана конкретная ветвь.

server.port: ${SERVER\_PORT:8888} — устанавливает номер порта для сервера конфигурации. Номер порта считывается из переменной среды с именем «SERVER\_PORT». Если переменная среды не задана, используется номер порта по умолчанию 8888.

eureka.client.service-url.default-zone: http://localhost:8761/eureka — устанавливает URL-адрес по умолчанию для сервера Eureka, который используется для обнаружения служб. Сервер конфигурации зарегистрируется на сервере Eureka, используя этот URL-адрес.

eureka.instance.prefer-ip-address: true — устанавливает предпочтение сервера конфигурации для использования своего IP-адреса вместо имени хоста при регистрации на сервере Eureka.

В целом, этот файл конфигурации настраивает сервер конфигурации Spring Cloud, который извлекает файлы конфигурации из репозитория Git и регистрируется на сервере Eureka для обнаружения служб. Он также указывает номер порта и другие параметры конфигурации для сервера конфигурации.

Шаблонное описание хэндлера сервиса для сервиса конфигурации сервера, как и в остальных примерах.

* + 1. Описание докерфайла

# установливаем образ версии Maven + название этапа сборки

FROM maven AS builder

# создать директорию

RUN mkdir -p $HOME

# путь к рабочему каталогу

WORKDIR $HOME

ADD pom.xml $HOME

# Эта команда создает проект, запускает все тестовые примеры и выполняет любые проверки результатов интеграционных тестов,

# чтобы убедиться, что критерии качества соблюдены.

RUN mvn verify --fail-never

ADD . $HOME

# выполнить командную строку внутри контейнера: mvn clean package для сборки зависимостей в jar

# скипаем выполнение тестов, который нет пока

RUN mvn clean package -DskipTests

# используем многоступенчатую сборку

# исполняемый код меняется чаще, чем зависимости

FROM openjdk:17-jdk-alpine

ARG JAR\_FILE=/target/\*.jar

# копируем jar из мавен контейнера в контейнер с приложением

COPY --from=builder ${JAR\_FILE} app.jar

# точка входа

# исполняемый файл, который будет запускаться при загрузке контейнера

ENTRYPOINT ["java", "-jar", "app.jar"]

* 1. Описание docker-compose.yml
     1. Описание

docker-compose.yml — это файл YAML, который используется для определения многоконтейнерного приложения Docker. Он используется для определения служб, сетей и томов, необходимых для запуска приложения. docker-compose — это инструмент командной строки, который используется для запуска и остановки приложения, управления контейнерами и настройки сети и томов.

* + 1. Описание кода

version: "3.7"

services:

  labtwo\_pg:

    image: postgres:14

    container\_name: labtwo\_pg

    ports:

      - "5433:5432"

    environment:

      POSTGRES\_DB: labtwo

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 12345

    volumes:

      - ./postgres/db:/var/lib/postgresql/data

  eureka-server:

    image: aixoxs/eureka-server:latest

    container\_name: eureka-server

    ports:

      - "8761:8761"

    expose:

      - "8761"

    environment:

      SERVER\_PORT: 8761

  config-server:

    image: aixoxs/config-server:latest

    container\_name: config-server

    ports:

      - "8888:8888"

    environment:

      SERVER\_PORT: 8888

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

  gateway\_service:

    image: aixoxs/gateway-service:latest

    container\_name: gateway-service

    ports:

      - "8080:8080"

    environment:

      SERVER\_PORT: 8080

      CONFIG\_HOST: config-server

      CONFIG\_PORT: 8888

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

  auth-facade:

    image: aixoxs/auth-facade:latest

    container\_name: auth-facade

    ports:

      - "8086:8086"

    environment:

      SERVER\_PORT: 8086

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

  auth-reactive-facade:

    image: aixoxs/auth-reactive-facade:latest

    container\_name: auth-reactive-facade

    ports:

      - "8087:8087"

    environment:

      SERVER\_PORT: 8087

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

  user-managment-service:

    image: aixoxs/user-managment-service:latest

    container\_name: user-managment-service

    ports:

      - "8084:8084"

    environment:

      SERVER\_PORT: 8074

      CONFIG\_HOST: config-server

      CONFIG\_PORT: 8888

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 12345

      spring.r2dbc.url: r2dbc:postgresql://labtwo\_pg/labtwo?currentSchema=user\_info

      spring.liquibase.url: jdbc:postgresql://labtwo\_pg:5433/labtwo?currentSchema=user\_info

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

      - labtwo\_pg

  basket-managment-service:

    image: aixoxs/basket-managment-service:latest

    container\_name: basket-managment-service

    ports:

      - "8084:8084"

    environment:

      SERVER\_PORT: 8075

      CONFIG\_HOST: config-server

      CONFIG\_PORT: 8888

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 12345

      spring.r2dbc.url: r2dbc:postgresql://labtwo\_pg/labtwo?currentSchema=basket\_info

      spring.liquibase.url: jdbc:postgresql://labtwo\_pg:5433/labtwo?currentSchema=basket\_info

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

      - labtwo\_pg

  product-managment-service:

    image: aixoxs/product-managment-service:latest

    container\_name: product-managment-service

    ports:

      - "8084:8084"

    environment:

      SERVER\_PORT: 8076

      CONFIG\_HOST: config-server

      CONFIG\_PORT: 8888

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 12345

      spring.r2dbc.url: r2dbc:postgresql://labtwo\_pg/labtwo?currentSchema=product\_info

      spring.liquibase.url: jdbc:postgresql://labtwo\_pg:5433/labtwo?currentSchema=product\_info

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

      - labtwo\_pg

  order-managment-service:

    image: aixoxs/order-managment-service:latest

    container\_name: order-managment-service

    ports:

      - "8084:8084"

    environment:

      SERVER\_PORT: 8077

      CONFIG\_HOST: config-server

      CONFIG\_PORT: 8888

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 12345

      spring.r2dbc.url: r2dbc:postgresql://labtwo\_pg/labtwo?currentSchema=order\_info

      spring.liquibase.url: jdbc:postgresql://labtwo\_pg:5433/labtwo?currentSchema=order\_info

      eureka.client.service-url.defaultZone: http://eureka-server:8761/eureka

    depends\_on:

      - eureka-server

      - config-server

      - labtwo\_pg

Это файл docker-compose.yml, который используется для определения и запуска многоконтейнерных приложений Docker. В этом файле используется версия 3.7, которая представляет собой синтаксическую версию Docker Compose.

Файл определяет несколько служб, каждая из которых работает в отдельном контейнере, с соответствующими конфигурациями. Первая служба — это служба базы данных PostgreSQL с именем labtwo\_pg, которая запускает последнюю версию образа Postgres с сопоставлением портов с хоста 5433 на контейнер 5432. Для базы данных устанавливаются переменные среды, включая имя базы данных, имя пользователя и пароль, и создается том для сохранения данных в каталоге ./postgres/db на хосте.

Следующая служба — это сервер Eureka, на котором работает образ aixoxs/eureka-server с сопоставлением портов с хоста 8761 на контейнер 8761. Поле expose используется для предоставления порта контейнера связанным службам, а переменные среды устанавливаются для сервера. порт.

Следующие службы — это микрослужбы, зависящие от сервера Eureka и сервера Config. Эти службы создаются с использованием соответствующих образов и запускаются в отдельных контейнерах с сопоставлением портов, переменными среды и зависимостями.

Служба config-server запускает образ aixoxs/config-server и использует сопоставление портов с хоста 8888 на контейнер 8888. Она также устанавливает переменную среды для порта сервера и eureka.client.service-url.defaultZone на URL-адрес сервера Eureka.

Служба gateway\_service запускает образ aixoxs/gateway-service и использует сопоставление портов с хоста 8080 на контейнер 8080. Переменные среды устанавливаются для порта сервера, CONFIG\_HOST для URL-адреса сервера конфигурации и eureka.client.service-url.defaultZone для URL-адрес сервера Эврика. Служба зависит от сервера Eureka и сервера Config.

Служба auth-facade запускает образ aixoxs/auth-facade и использует сопоставление портов с хоста 8086 на контейнер 8086. Переменная среды устанавливается для порта сервера, а eureka.client.service-url.defaultZone — для URL-адреса сервера Eureka. Служба зависит от сервера Eureka и сервера Config.

Служба auth-reactive-facade запускает образ aixoxs/auth-reactive-facade и использует сопоставление портов с хоста 8087 на контейнер 8087. Переменная среды устанавливается для порта сервера, а eureka.client.service-url.defaultZone — для URL-адрес сервера Эврика. Служба зависит от сервера Eureka и сервера Config.

Служба управления пользователями запускает образ aixoxs/user-managment-service и использует сопоставление портов с хоста 8084 на контейнер 8084. Переменные среды устанавливаются для порта сервера, CONFIG\_HOST для URL-адреса сервера конфигурации и POSTGRES\_USER, POSTGRES\_PASSWORD, spring.r2dbc.url и spring.liquibase.url для подключения к базе данных. Служба зависит от сервера Eureka, сервера конфигурации и службы labtwo\_pg.

Служба управления корзиной запускает образ aixoxs/basket-managment-service и использует сопоставление портов с хоста 8084 на контейнер 8084. Переменные среды устанавливаются для порта сервера, CONFIG\_HOST для URL-адреса сервера конфигурации и POSTGRES\_USER, POSTGRES\_PASSWORD, spring.r2dbc.url и spring.liquibase.url для подключения к базе данных. Служба зависит от сервера Eureka, сервера конфигурации и службы labtwo\_pg.

Микросервис product-managment-service основан на образе Docker aixoxs/product-managment-service:latest. Он имеет имя контейнера product-managment-service и сопоставлен с портом 8084 на хосте. Он также указывает несколько переменных среды, включая SERVER\_PORT, CONFIG\_HOST, CONFIG\_PORT, POSTGRES\_USER, POSTGRES\_PASSWORD, spring.r2dbc.url, spring.liquibase.url и eureka.client.service-url.defaultZone. Поле depend\_on указывает, что эта служба зависит от трех других служб: eureka-server, config-server и labtwo\_pg.

Микросервис управления заказами аналогичен микросервису управления продуктами. Он основан на образе Docker aixoxs/order-managment-service:latest, имеет имя контейнера order-managment-service и сопоставлен с портом 8084 на хосте. Он также указывает несколько переменных среды, включая SERVER\_PORT, CONFIG\_HOST, CONFIG\_PORT, POSTGRES\_USER, POSTGRES\_PASSWORD, spring.r2dbc.url, spring.liquibase.url и eureka.client.service-url.defaultZone. Поле depend\_on указывает, что эта служба зависит от трех других служб: eureka-server, config-server и labtwo\_pg.

В целом, этот код представляет собой конфигурацию двух микрослужб, для правильной работы которых требуется несколько других служб и переменных среды. Микросервисы также используют базы данных PostgreSQL с определенными схемами (product\_info и order\_info) и регистрируются на сервере Eureka по адресу http://eureka-server:8761/eureka.